UNIVERSIDADE DE COIMBRA
FACULDADE DE CIÊNCIAS E DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

DEPENDABILITY MECHANISMS FOR DESKTOP GRIDS

Patrício Rodrigues Domingues

DOUTORAMENTO EM ENGENHARIA INFORMÁTICA

2008

UNIVERSIDADE DE COIMBRA
FACULDADE DE CIÊNCIAS E DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

DEPENDABILITY MECHANISMS FOR DESKTOP GRIDS

Patrício Rodrigues Domingues

DOUTORAMENTO EM ENGENHARIA INFORMÁTICA

2008

Tese orientada pelo Prof. Doutor Luís Moura e Silva

This work was partially supported by the *Programa de Formação Avançada de Docentes do Ensino Superior Medida 5/Acção 5.3* (PRODEP III), by the Portuguese Foundation for Science and Technology under the POSI programme, by the FEDER programme of the European Union through the R&D unit 326/94 (CISUC) and by the CoreGRID programme funded by the European Commission (Contract IST-2002-004265).

Abstract

It is a well-known fact that most of the computing power spread over the Internet simply goes unused, with CPU and other resources sitting idle most of the time: on average less than 5% of the CPU time is effectively used. Desktop grids are software infrastructures that aim to exploit the otherwise idle processing power, making it available to users that require computational resources to solve long-running applications. The outcome of some efforts to harness idle machines can be seen in public projects such as SETI@home and Folding@home that boost impressive performance figures, in the order of several hundreds of TFLOPS each. At the same time, many institutions, both academic and corporate, run their own desktop grid platforms. However, while desktop grids provide free computing power, they need to face important issues like fault tolerance and security, two of the main problems that harden the widespread use of desktop grid computing.

In this thesis, we aim to exploit a set of fault tolerance techniques, such as checkpointing and redundant executions, to promote faster turnaround times. We start with an experimental study, where we analyze the availability of the computing resources of an academic institution. We then focus on the benefits of sharing checkpoints in both institutional and wide-scale environments. We also explore hybrid schemes, where the traditional centralized desktop grid organization is complemented with peer-to-peer resources.

Another major issue regarding desktop grids is related with the level of trust that can be achieved relatively to the volunteered hosts that carry out the executions. We propose and explore several mechanisms aimed at reducing the waste of computational resources needed to detect incorrect computations. For this purpose, we detail a checkpoint-based scheme for early detection of errors. We also propose and analyze an invitation-based strategy coupled to a credit rewarding scheme, to allow the enrollment and filtering of more trustworthy and more motivated resource donors.

To summarize, we propose and study several fault tolerance methodologies oriented toward a more efficient usage of resources, resorting to techniques such as checkpointing, replication and sabotage tolerance to fasten and to make more reliable executions that are carried over desktop grid resources. The usage of techniques like these ones will be of ultimate importance for the wider deployment of applications over desktop grids.

KEYWORDS: Fault tolerance, desktop grids, volunteer computing, checkpointing, scheduling, sabotage-tolerance.

Resumo Estendido

Introdução

É um facto sabido que uma vasta percentagem do poder computacional de computadores pessoais acaba perdida. De facto, no âmbito desta dissertação constatou-se que a percentagem média de inactividade dos processadores de mais de centena e meia de computadores de uma instituição académica era de 97,93%. Este valor confirma a regra empírica dos 5/95%, regra essa que afirma que um CPU tem uma taxa média de utilização de cerca 5%, sendo os restantes 95% desaproveitados. Ironicamente, o constante aumento do poder computacional leva a que o poder computacional desaproveitado aumente de ano para ano. O recente aparecimento de arquitecturas que integram vários núcleos independentes (*multi-cores*) num mesmo processador leva a que possam existir vários *cores* que, grande parte do tempo, não estão a ser usados pelo utilizador da máquina. Deste modo, perde-se poder computacional, uma vez que recursos como memória e CPU não utilizados num dado instante não podem ser armazenados para posterior uso.

Se a maioria dos utilizadores de computadores pessoais apenas recorre à totalidade das capacidades da máquina por curtos períodos de tempos para suprir necessidades pontuais, outros utilizadores estão fortemente dependentes de elevados recursos computacionais para a execução de aplicações nas mais diversas áreas do conhecimento, desde o cálculo das propriedades de compostos químicos, a construção de imagens 3-D, a detecção de ondas gravíticas, o experimentar de modelos de propagação de doenças, ou para uma das muitas outras áreas do saber que estão fortemente dependentes de poder computacional. Para estes utilizadores, todo o poder computacional que possam usar é bem-vindo. Obviamente, essa classe de utilizadores preferia ter acesso a recursos computacionais dedicados, mas esses ou não existem ou não estão disponíveis devido a restrições orçamentais.

O estudo de técnicas orientadas para o aproveitamento de recursos computacionais não dedicados data dos finais da década de 80. A generalização do computador pessoal e da Internet contribuíram ainda mais para o despontar de sistemas distribuídos de larga escala orientados para o aproveitamento de recursos denominados de *voluntários*. Esta designação advém do facto dos donos/responsáveis destes recursos os disponibilizarem voluntariamente. Estes sistemas, também designados pelo termo anglo-saxónico *volunteer computing*, foram popularizados pelo seu uso em projectos de computação voluntária tais como o *distributed.net*, o *SETI@home*, o *Folding@home*, entre muitos outros. Na sequência do enorme sucesso do projecto SETI@home, e cientes das dificuldades técnicas que sentiram na criação e manutenção do referido projecto, os seus promotores implementaram a plataforma *Berkeley Open Infrastructure for Network Computing* (BOINC). Desta forma, o BOINC foi desenvolvido com o intuito de tornar mais simples e flexível a instalação e manutenção de um projecto de computação voluntária. O BOINC é hoje umas das principais plataformas para computação voluntária para ambientes de larga escala, sendo empregue em cerca de trinta projectos públicos de computação voluntária tais como o *Einstein@home*, o *SIMAP@home*, o *Rosetta@home* e o *SZTAKI Desktop Grid*, entre outros. O *XtremWeb* é outro exemplo de plataforma para aproveitamento de recursos não dedicados, embora esteja mais vocacionado como plataforma para experimentação de conceitos e técnicas relacionadas com o *desktop grid*.

Apesar dos custos associados ao uso de recursos voluntários serem reduzidos relativamente ao poder computacional que pode ser alcançado, algumas limitações restringem o seu uso. As limitações dos ambientes de computação voluntária relacionam-se com (1) a baixa prioridade de execução concedida às aplicações externas, e com (2) as assimetrias das redes de computadores que levam a que, muitas vezes, duas ou mais máquinas não possam comunicar directamente entre si. De facto, os recursos não dedicados executam as aplicações externas com um nível de prioridade inferior ao empregue para as aplicações dos utilizadores interactivos, e, nalguns casos, as aplicações externas apenas são executadas quando não existe uso interactivo dos recursos, isto é, quando nenhum utilizador está a usar a máquina. Nestes casos, a execução de uma aplicação externa é suspensa logo que seja detectada utilização interactiva da máquina. Adicionalmente, um recurso pode repentinamente ser desligado por um período de tempo indeterminado ou mesmo por tempo infinito (por exemplo, o dono do recurso decidiu cessar a partilha do mesmo). A conjugação destes dois factores – baixa prioridade no acesso aos recursos e imprevisibilidade da disponibilidade dos recursos – leva a que os recursos não dedicados apresentem elevada volatilidade, uma característica a ponderar por quem adapta as aplicações externas a ambientes não dedicados. A técnica habitual para lidar com este tipo de situações é a da salvaguarda periódica do estado da aplicação para suporte persistente (*checkpointing*). Na sequência de uma interrupção, e logo que o recurso esteja de novo disponível, a execução da aplicação reinicia-se a partir do último ponto de execução salvaguardado.

No que respeita à assimetria das comunicações, os mecanismos de *firewalls* e de translação de endereços (vulgo *Network Address Translation*, NAT) originam redes assimétricas, nas quais, as máquinas não podem comunicar directamente umas com as outras, ou então, a comunicação até pode ser feita directamente, mas apenas pode ocorrer num determinado sentido. É aliás a assimetria nas comunicações que leva a que os ambientes de computação *desktop grids* sejam baseados no modelo de *master-worker*, no qual a iniciativa da comunicação parte do lado do *worker*. Outra limitação dos canais de comunicação relaciona-se com a largura de banda. De facto, para além da heterogeneidade ao nível da capacidade dos canais de comunicação, com a existência de recursos com diferentes velocidades de acesso à rede, existe ainda assimetria nas larguras de banda, com débitos de envio diferentes dos débitos de recepção. Adicionalmente, uma percentagem significativa de máquinas está limitada em termos de volume de tráfego no acesso à Internet, não podendo ultrapassar uma determinada quantidade de tráfego mensal, sob pena do tráfego adicional ser fortemente taxado.

A par da volatilidade dos recursos, as limitações ao nível das comunicações condicionam o tipo de aplicações que pode ser eficientemente suportado pelos ambientes não dedicados de computação. De facto, mesmo se a existência de recursos distribuídos permite a distribuição da aplicação por várias tarefas, as limitações resultantes da elevada volatilidade dos recursos e da assimetria ao nível das comunicações obrigam a que essas tarefas sejam independentes, no sentido que a execução de uma qualquer tarefa não pode depender em nada da execução de outra tarefa. Isto é, as tarefas de uma mesma aplicação têm que ser autónomas no que respeita à sua execução. Deste modo, torna-se difícil a execução de aplicações paralelas ou de aplicações distribuídas cujas tarefas estejam fortemente acopladas, pela falta de disponibilidade das máquinas e pela dificuldade de estabelecer conectividade directa entre os *workers*.

Objectivos da Dissertação

Esta dissertação visa alcançar os seguintes objectivos na área da computação avançada:

1. Estudo do nível médio de uso dos recursos computacionais existentes em laboratórios académicos;
2. Especificação de metodologias de escalonamento para aplicações distribuídas de elevado desempenho executadas em recursos não dedicados de ambientes institucionais. Concretamente, procura-se diminuir o tempo de execução das aplicações, intervindo ao nível do escalonamento das tarefas individuais com migrações, replicações, e ainda com a partilha de *checkpointing*;
3. Adaptação dos mecanismos de partilha de checkpointing a ambientes de larga escala não dedicados;
4. Extensão do modelo tradicional de *desktop grid* por forma a optimizar a partilha de dados de entrada e de *checkpoints* em ambientes institucionais e em ambientes de larga escala;
5. Detecção precoce de erros através da validação de resultados intermédios baseada na comparação de ficheiros de *checkpointing*;
6. Apresentação de várias metodologias para incrementar o número de nodos voluntariados para a computação bem como o seu nível de contribuição e ainda minorar a ocorrência de comportamentos maliciosos.

De seguida, descrevem-se os referidos objectivos, sendo apresentados os principais resultados alcançados relativos a cada um deles.

Nível de Uso dos Recursos Computacionais

Foi realizado um estudo no âmbito desta dissertação que abrangeu 169 computadores pessoais de 11 laboratórios de informática de uma instituição académica. Durante os 77 dias que durou a monitorização, foi observado um nível médio de inactividade de CPU de 97,93%. Este valor médio baixou para 94,24% aquando da presença de um utilizador interactivo e aumentou para 99,71% aquando da inexistência de utilizador interactivo. Este último valor indica um consumo residual de 0,3% de CPU quando a máquina não está a ser utilizada. Os elevados valores de inactividade ilustram o grande potencial que pode constituir o aproveitamento de recursos computacionais não dedicados.

No que respeita à memória RAM, a taxa média de ocupação observada no referido estudo foi de 58,94%, sendo de 67,53% aquando da presença de utilizador interactivo e de 54,81% na ausência de uso interactivo. Importa notar que a interpretação das taxas médias de ocupação deve ser feita com cuidado, pois a variação da quantidade de memória entre as máquinas era apreciável, existindo máquinas com 128 MB, outras com 256 MB e outras com 512 MB. Ao nível da utilização da rede, denotou-se um padrão de uso *cliente-servidor*, com as máquinas estudadas a desempenhar o papel de cliente. De facto, em média, o volume de tráfego recebido por máquina era várias vezes superior ao volume de tráfego enviado.

Os valores observados no citado estudo de monitorização confirmam o sobredimensionamento dos computadores pessoais em geral e dos CPUs em particular para a maioria das actividades desenvolvidas em computadores, nomeadamente o uso de ferramentas apelidadas de produtividade (processador de texto, folha de cálculo, etc.) e de utilitários de comunicação (correio electrónico, WEB, etc.). Assim, essas máquinas apresentam um elevado potencial que pode ser aproveitado para a execução de aplicações de computação intensiva.

Escalonamento Orientado para a Execução mais Rápida das Aplicações

Mesmo que um número significativo de aplicações não possa ser adaptado ao modelo da tarefa independente, e portanto não são passíveis de serem executadas em ambientes *desktop grids*, o número de aplicações apropriadas ou adaptáveis a este paradigma é largamente suficiente para justificar o uso em larga escala deste tipo de plataformas. Prova disso é o crescente número de projectos assentes em plataformas de computação voluntária.

Uma particularidade deste tipo de plataforma é o facto das políticas de escalonamento de tarefas privilegiarem a taxa global de execução de tarefas (*throughput*) em detrimento da velocidade individual de cada aplicação. Deste modo, embora o sistema seja eficiente quando analisado na perspectiva da taxa global de execução de tarefas, não o é quando visto na perspectiva de um utilizador individual que necessita que a sua aplicação seja executada o mais rapidamente possível (por exemplo, para que possa analisar os resultados da execução corrente e planear as execuções seguintes de acordo com os resultados obtidos). Um exemplo clássico desta situação é o *síndroma da última tarefa*, no qual o término de uma aplicação composta por múltiplas tarefas independentes acaba por ser retardado pelo facto da última tarefa estar a ser executada numa máquina lenta ou instável, obrigando a muitas reinicializações da tarefa que executa. Nesta dissertação exploram-se metodologias de escalonamento que, combinadas com os indispensáveis mecanismos de *checkpointing* nas plataformas de aproveitamento de recursos não dedicados, permitem tornar mais célere a execução de aplicações, seja através de migração e replicação de tarefas, ou mediante modelos que fornecem uma previsão a curto prazo sobre a disponibilidade dos recursos computacionais. Com essa finalidade, analisam-se várias metodologias de escalonamento assentes em mecanismos de salvaguarda que privilegiem a execução rápida de aplicações individuais, mesmo que isso ocorra em detrimento da taxa global de execução de tarefas. Concretamente, as metodologias de escalonamento analisadas e/ou propostas foram:

- FCFS (*First Come First Served*): metodologia que se limita a atribuir ao computador cliente uma tarefa cujos requisitos sejam apropriados à máquina. Esta metodologia é tradicionalmente empregue nos sistemas de *high throughput computing*. De modo a adaptar o escalonamento FCFS à prossecução de resultados orientados para a execução rápida de aplicações, foi empregue a partilha de ficheiros de salvaguarda, sendo os mesmos guardados num servidor. Esta metodologia diverge do modelo tradicional em que os ficheiros de salvaguarda de uma tarefa são mantidos numa só máquina e consequentemente são privados à máquina. Note-se que as restantes metodologias são variantes da metodologia FCFS;

- FCFS-AT (*First Come First Served with Adaptive Timeout*): baseada na metodologia FCFS, o escalonamento FCFS-AT define, aquando da atribuição de tarefa a uma dada máquina requerente, um tempo máximo de execução. Esse tempo máximo é determinado em função da velocidade relativa da máquina cliente encarregue de executar a tarefa. Caso a máquina cliente ultrapasse o tempo de execução definido, o supervisor considera a tarefa como perdida e volta a disponibilizá-la para novo escalonamento;

- FCFS-TR (*FCFS with Transfer Replication*): o escalonamento FCFS-TR introduz a capacidade de replicação de uma tarefa. Concretamente, no escalonamento FCFS-TR, o supervisor pode proceder à replicação da execução de uma tarefa, atribuindo a uma máquina requerente uma tarefa que já se encontre em execução numa ou mais máquinas. Essa replicação ocorre apenas quando já não restam tarefas por atribuir e existam máquinas livres, ou seja, quando a execução da aplicação se encaminha para o seu término. Em ambientes com armazenamento de ficheiros de salvaguarda num servidor, a criação da réplica de uma tarefa é feita a partir da última salvaguarda disponível, reaproveitando-se deste modo a computação salvaguardada. Nos casos em que haja replicação para uma máquina mais rápida do que aquela que originalmente acolheu a tarefa, há uma execução mais rápida da tarefa, factor importante para um mais célere término da aplicação. Mesmo que a replicação seja feita para uma máquina de igual ou menor desempenho do que a original, a replicação aumenta a capacidade de tolerância a falhas, permitindo reduzir ou mesmo mascarar os custos caso ocorra interrupção na execução de uma das instâncias da tarefa.

- FCFS-TR-DMD (*FCFS with Task Replication on Demand*): a metodologia FCFS-TR-DMD acrescenta ao escalonamento FCFS-TR a capacidade de activar o mecanismo de *salvaguarda a pedido*. Como o nome sugere, este mecanismo permite que o escalonador requisite uma operação de salvaguarda a uma máquina que se encontre a processar uma determinada tarefa. Deste modo, torna-se possível replicar uma tarefa com um estado actualizado, maximizando o reaproveitamento de computação;

- FCFS-TR-DMD-PRDCT (*FCFS with Task Replication on Demand with Prediction*): em relação à metodologia anterior, este escalonamento introduz o uso de metodologias para a previsão da disponibilidade de curto prazo dos recursos computacionais. Assim, caso a previsão aponte que uma máquina tem elevada probabilidade de ser interrompida num futuro próximo, a metodologia potencia a replicação da tarefa que possa estar em execução nessa máquina. Similarmente, caso uma máquina requerente de tarefa apresente uma elevada probabilidade de ser interrompida, o escalonador poderá atribuir-lhe uma função de menor importância, por exemplo, a execução da réplica de uma tarefa. No caso do FCFS-TR-DMD-PRDCT foi seguida a metodologia de previsão *Sequential Minimal Optimization*.

A respeito das metodologias de escalonamento acima expostas, importa notar que se assume um escalonador centralizado e com um conhecimento médio do estado do sistema. Por conhecimento médio entende-se que o escalonador recebe periodicamente (por exemplo, todos os n minutos) as actualizações referentes ao estado das máquinas, nomeadamente sobre a disponibilidade ou não das mesmas para a execução de tarefas externas.

As metodologias foram simuladas através do `DGSchedSim`, um simulador desenvolvido para o efeito. A designação *tempo total de execução* corresponde ao intervalo de tempo que medeia desde a submissão da primeira tarefa até ao término da última tarefa. As simulações foram realizadas com recurso à técnica de *trace-*

driven tendo sido empregues os registos de actividade das máquinas capturados num ambiente real pela ferramenta WindowsDDC. Os parâmetros considerados para as simulações foram o número de tarefas por aplicação, o tamanho individual das tarefas, o número, a capacidade e disponibilidade computacional das máquinas, a frequência de *checkpointing* por tarefa e ainda o período de execução (dias *úteis* ou fim-de-semana).

Os principais resultados referentes às metodologias de escalonamento acima apresentadas foram:

- A partilha de ficheiros de *checkpointing* origina melhorias de desempenho significativas com redução dos tempos de execução que podem ir até aos 60%;

- Para aplicações de curta duração (até uma hora), a frequência de *checkpointing* tem pouca influência. Este facto sugere a viabilidade da execução de aplicações de muito curta duração sem que sejam empregues mecanismos de *checkpointing* nos ambientes computacionais considerados;

- A replicação de tarefas pode trazer proveitos importantes ao nível do desempenho, especialmente em ambientes heterogéneos onde coexistam máquinas com diferentes velocidades computacionais. De facto, o replicar de uma tarefa para uma máquina mais rápida do que a original possibilita uma execução mais rápida. Adicionalmente, o mecanismo de replicação melhora significativamente o desempenho se acoplado com *checkpointing* a pedido.

- A previsão da disponibilidade dos recursos computacionais possibilita algum ganho ao nível do desempenho, embora apenas em aplicações de longa duração (superior a quatro horas), revelando-se inapropriado para aplicações de curta duração;

- A hora do dia e o dia da semana são dois factores a ter em conta no escalonamento de aplicações em recursos não dedicados. De facto, os recursos computacionais considerados exibem um padrão regular de uso interactivo, com os recursos a encontrarem-se livres durante os períodos nocturnos e especialmente aos fins-de-semana. Daí resulta que para os fins-de-semana, uma simples metodologia baseada em replicação como a FCFS-TR seja suficiente para a obtenção de um bom desempenho. Similarmente, a execução de tarefas medianamente longas (até dois dias) e que não possuam suporte para salvaguarda periódica do respectivo estado, deve realizar-se aos fins-de-semana de modo a maximizar a probabilidade de término.

Mecanismos de Salvaguarda para Ambientes de Larga Escala

Para além das metodologias que favorecem a execução rápida em ambientes de área local, esta dissertação foca ainda os sistemas de aproveitamento de recursos em larga escala, sempre no âmbito da acoplagem com os mecanismos de salvaguarda. Concretamente, é proposto e simulado o sistema *chkpt2chkpt*, no qual é empregue uma *distributed hash table* (DHT) para vigiar o estado de execução de cada tarefa e a localização (em termos de máquina) dos ficheiros de salvaguarda que são periodicamente guardados em suporte persistente numa das máquinas cujo dono/responsável disponibiliza os seus recursos para o efeito. É associada a cada instância da tarefa[1] um nodo de monitorização designado de *guardian*, cujo propósito é o de manter na DHT uma entrada referente ao estado corrente de cada uma das instâncias da tarefa. É ainda mantida na DHT a localização dos ficheiros de salvaguarda correspondentes aos instantes $t_1, t_2, ..., t_N$. Sempre que é detectada uma tarefa interrompida – a tarefa não actualiza na DHT o seu estado de execução por um período de tempo superior ao predeterminado – o sistema lança novamente a tarefa, sendo essa re-execução iniciada a partir do último estado salvaguardado, caso esteja disponível. Para tal, é efectuada uma consulta à DHT tendo em conta o último ponto t_p de salvaguarda que foi registado. Se este último ponto t_p não se encontrar disponível, por exemplo porque a máquina que alberga os ficheiros de *checkpointing* pretendidos não estar disponível, o sistema procura o ponto imediatamente anterior, isto é, t_{p-1} e assim sucessivamente, até que seja encontrado um ponto de *checkpointing* disponível. Caso nenhum ponto de *checkpointing* esteja disponível, a tarefa é reiniciada. A simulação da metodologia *chkpt2chkpt* revelou que o sistema se torna rentável a partir de uma taxa de falha de 5%. De facto, acima desse patamar, os benefícios do recurso aos estados salvaguardados para retomar execuções interrompidas tornam-se superiores aos custos incorridos na manutenção da DHT e na replicação dos ficheiros de *checkpointing*.

[1]O sistema recorre à replicação de tarefas para validar os resultados, comparando no final das execuções, as saídas produzidas por cada uma das instâncias. Deste modo, torna-se necessário a execução de várias instâncias de uma mesma tarefa para que possam ser validados os resultados.

Exploração de Novos Paradigmas de Cooperação entre Tarefas

Um factor pouco explorado dos ambientes de aproveitamento de recursos computacionais não dedicados prende-se com a capacidade de várias máquinas cooperarem na execução de um conjunto de tarefas. Para este efeito, os recursos são vistos numa perspectiva federada, sendo analisados dois tipos de federações: (1) *ambientes institucionais* e (2) *ambientes peer-to-peer*.

O primeiro tipo de ambiente contempla os recursos de uma mesma instituição, existentes numa mesma zona geográfica, de tal modo que os recursos se encontrem ligados por tecnologia de rede local e sob o controlo administrativo de uma só entidade que por si só pode decidir a forma com os recursos são postos à disposição de ambientes de execução não dedicados. São exemplos desta configuração, *campus* académicos, parques informáticos de empresas, etc. Importa notar que os ciclos de CPU excedentários dos ambientes institucionais são usualmente disponibilizados para utilizadores da própria instituição e não para o exterior.

Nesta dissertação, a metodologia seguida de cooperação em ambientes institucionais assenta na existência de um serviço de coordenação de partilha, o *procurador local* (LPS, seguindo a designação anglo-saxónica de *Local Proxy Server*). O LPS coordena os recursos e age como ponte para o exterior, explorando possíveis simbioses (e.g., partilha de ficheiros de dados necessários à computação das tarefas, partilha de tarefas, etc.), entre as máquinas integrantes do ambiente institucional.

Por sua vez, os ambientes *peer-to-peer* são formados por recursos não relacionados entre si, em que os donos/responsáveis dos recursos muito possivelmente não se conhecem, e sobre os quais não existe nenhuma autoridade central. Um exemplo de ambiente *peer-to-peer* é o formado pelos recursos voluntariados para determinado projecto (e.g. *SETI@home*, *Folding@home*) e que possuem determinadas características que tornam a sua associação viável (por exemplo, as máquinas possuem ligações de rede bastante rápidas entre elas e são publicamente endereçáveis do exterior). Pela ausência de controlo centralizado, os ambientes *peer-to-peer* são bastante mais voláteis do que os ambientes institucionais. Adicionalmente, a exploração de ambientes *peer-to-peer* requer protecção contra comportamentos gananciosos e/ou maliciosos, comportamentos esses que podem surgir tanto por parte dos donos dos recursos como dos utilizadores[2].

[2]Neste caso, *utilizador* designa o indivíduo que pretende ver as suas aplicações executadas e que eventualmente pode maldosamente querer causar prejuízos aos recursos voluntariados.

Níveis de cooperação

Como complemento ao modelo tradicional de tarefas independentes, nesta dissertação são propostos os seguintes níveis de cooperação entre as tarefas de uma mesma aplicação:

- partilha dos dados de entrada;
- partilha dos ficheiros de *checkpointing*;
- comunicação entre nodos;

De seguida, cada um dos níveis de cooperação é descrito.

Partilha dos dados de entrada

Os dados de entrada de uma tarefa correspondem à informação necessária à computação da tarefa, sendo que muitas vezes a própria computação incide sobre esses dados (aplicação de determinado(s) algoritmo(s) aos dados, etc.). O tamanho dos dados de entrada depende não só do problema que a aplicação pretende resolver, como da divisão de trabalho efectuados.

A partilha de dados de entrada consiste no reaproveitamento de ficheiros de dados de entrada entre máquinas afectas a um mesmo projecto. Deste modo, uma aplicação pode receber os dados de entrada de que necessita a partir de outra máquina que já os tenha e não apenas a partir da autoridade central (parte supervisora).

Para que a partilha de dados de entrada faça sentido, é necessário que os mesmos ficheiros de entrada sejam empregues por várias tarefas diferentes, uma característica totalmente dependente da aplicação. Saliente-se que não é conveniente considerar a partilha de dados de entrada entre instâncias de uma mesma tarefa, pois isso levaria a que os recursos executantes das várias instâncias passassem a ter conhecimento uns dos outros. Tal situação potenciaria o conluio, em que uma maioria de executores pudessem maldosamente combinar entre si a apresentação do mesmo resultado falsificado que, por via da maioria, iria ser aceite como correcto pela entidade supervisora.

Por esses dados serem apenas de leitura, a partilha dos mesmos revela-se relativamente fácil, sendo somente necessário verificar que os dados de entrada obtidos via terceiros são cópia conforme aos dados originais, prevenindo desta forma não só erros de transmissão mas também adulterações maliciosas. Esta verificação pode ser feita com recursos a mecanismos de certificação do conteúdo de ficheiros tais como os algoritmos de *hash Message Digest 5* (MD5) e *Secure Hash Algorithm* (SHA), sendo calculado para a cópia recebida o código de certificação que é comparado com o código disponibilizado pela parte servidora.

Partilha dos ficheiros de *checkpointing*

A partilha dos ficheiros de *checkpointing* visa permitir a migração de tarefas entre máquinas, por forma a que uma tarefa interrompida numa máquina possa ser continuada a partir do seu último ponto salvaguardado numa outra máquina. Desta forma, na interrupção de uma tarefa, apenas é perdida a computação efectuada desde o último ponto de *checkpointing* até ao instante em que a tarefa foi efectivamente interrompida. Isto permite não só o retomar da execução de uma tarefa interrompida, como possibilita a migração e/ou replicação preventiva de tarefas, através da qual uma tarefa é migrada e/ou replicada para uma máquina mais rápida ou mais confiável (isto é, com um menor historial de falhas) com o objectivo de conseguir uma execução mais rápida.

A partilha dos ficheiros de *checkpointing* requer que os mesmos estejam acessíveis mesmo quando a máquina na qual eles foram criados esteja indisponível (por exemplo, desligada). Deste modo, os ficheiros de *checkpointing* deixam de ser privados, havendo a necessidade de recorrer à replicação dos referidos ficheiros em tempo oportuno, idealmente, logo após serem criados, sendo as réplicas guardadas num servidor local (esta solução é apenas viável num ambiente institucional) ou noutras máquinas (ambiente *peer-to-peer*).

Similarmente ao que sucede com a partilha de dados de entrada, é necessário validar a integridade de um ficheiro de salvaguarda, pois este pode ser corrompido, seja acidental ou maldosamente. Contudo, ao invés dos dados de entrada que são produzidos exclusivamente pela parte servidora do sistema que deste modo mantém pleno controlo sobre os mesmos, podendo disponibilizar códigos de verificação (MD5, SHA, etc.), os ficheiros de salvaguarda são produzidos pelas máquinas executantes, e portanto a verificação de integridade é mais complexa. De facto, tal verificação requer, ora a confiança na entidade que produziu o ficheiro de salvaguarda, ora a validação por execução redundante, na qual várias máquinas independentes executam a mesma tarefa sendo o resultado final validado mediante a comparação entre as várias máquinas executantes.

Comunicação entre nodos

A execução de aplicações compostas por tarefas dependentes em ambientes distribuídos requer apropriados canais de comunicação, por forma a que haja comunicação e sincronização entre as tarefas dependentes.

A comunicação pode realizar-se de forma indirecta, no qual dois ou mais nodos comunicam entre si recorrendo ao paradigma de memória partilhada (vulgo *tuple space*). Assim, o nodo emissor deposita a mensagem na memória partilhada sobre a forma de um tuplo, identificado por uma chave única. Este tuplo é posteriormente acedido por outro nodo que apresente a mesma chave. A metodologia é apelidade de indirecta porque não há lugar à comunicação directa entre as entidades comunicantes.

Na comunicação directa as entidades comunicantes trocam mensagens entre si. Contudo, pode existir uma entidade intermédia encarregue de encaminhar as mensagens entre origem e destino, por forma a ultrapassar as barreiras criadas pelos sistemas de *Network Address Translation* (NAT).

Validação de Resultados através da Comparação de *checkpoints*

Os esquemas tradicionais de computação voluntária recorrem à redundância para a validação de resultados. Concretamente, a mesma tarefa é escalonada por N máquinas independentes (N é no mínimo dois), sendo que quando todas as execuções estiverem completas, os resultados são comparados, sendo descartados aqueles que estejam em minoria. Por exemplo, num esquema de redundância tripla, $N = 3$, se existirem dois resultados idênticos e um terceiro diferente, então este último é considerado inválido e consequentemente descartado. Embora a abordagem da execução redundante seja relativamente simples de implementar, a detecção de resultados incorrectos apenas ocorre, na melhor perspectiva, quando existir já uma maioria de execuções completadas, uma solução que se revela insatisfatória quando as execuções são muito demoradas (várias horas, possivelmente dias) e/ou se pretende uma rápida obtenção de resultados devidamente validados.

Nesta dissertação é proposto e simulado um esquema de validação intermédio de resultados aproveitando os já existentes mecanismos de salvaguarda assentes em *checkpointing*. Concretamente, são empregues os ficheiros de salvaguarda de pontos de execução intermédios (por exemplo, após o primeiro terço de execução) das execuções redundantes, sendo calculada, via *hashing*, a assinatura dos ficheiros de salvaguarda de cada ponto de execução intermédio. As assinaturas resultantes de um ponto de execução equivalente das várias execuções redundantes de uma mesma tarefa são comparadas, sendo que assinaturas divergentes indicam que pelo menos uma das execuções está incorrecta. Deste modo, e relativamente à metodologia tradicional que apenas detecta erros através da comparação dos resultados finais, a comparação de pontos de salvaguarda intermédios permite uma detecção mais precoce de execuções erradas. Isto verifica-se em todos os casos excepto se o erro ocorrer na última parte da computação. Neste caso, a detecção do erro apenas é possível no final da execução quando são comparados os resultados finais.

A metodologia proposta permite não só uma detecção e reacção mais rápida a erros, mas também possibilita que o re-escalonamento de uma execução redundante que vise substituir uma execução identificada num ponto intermédio como errada, possa ser iniciada a partir do último ponto de execução intermédio que foi validado pelas várias execuções redundantes. Deste modo, caso o último estado salvaguardado esteja disponível, a nova execução pode ser retomada do referido ponto intermédio, permitindo uma execução mais célere. De salientar que a transferência de ficheiros de salvaguarda requer a existência de comunicação directa ou indirecta entre os vários nodos executantes por forma a que o referido estado possa ser transferido para o novo nodo executante.

Reputação e Confiabilidade em Ambientes de Computação Voluntária

Um dos problemas actuais da computação em ambientes não dedicados de larga escala relaciona-se com a necessidade de cativar pessoas que voluntariem os seus recursos computacionais. Se bem que alguns utilizadores sejam atraídos pelo espírito pioneiro e científico do(s) projecto(s) a que se decidam associar, outros voluntariam os seus recursos meramente pela perspectiva de competição, dado que muitos projectos avaliam o desempenho dos seus participantes recompensando-os com créditos virtuais, créditos esses que são publicitados e que dão origem a classificações. Contudo, apesar da existência de um sistema de classificações e recompensa elevar o entusiasmo de certos participantes, também induz a comportamentos desonestos, em que certos utilizadores não hesitam em falsificar resultados ou em optimizar o próprio binário das tarefas para incrementar o número de tarefas que processam por unidade de tempo, tudo numa perspectiva de receber mais créditos e consequentemente alcançarem uma melhor classificação. Este comportamento é também facilitado pela inexistência de identificação fidedigna de um voluntário, dado o acesso a um projecto apenas requerer um endereço válido de correio electrónico, endereço esse que pode ser facilmente criado de forma praticamente anónima na Internet. Se bem que os projectos possuam mecanismos de validação de resultados, muitas vezes baseada em execução redundante, a existência de tais comportamentos em nada beneficia a imagem do projecto, podendo mesmo levar à desistência de voluntários desgostosos com situações menos correctas.

De modo a incentivar os voluntários a recrutarem outros voluntários com elevada produtividade, nesta dissertação propõe-se um esquema assente em convites para a disponibilização voluntária de recursos a projectos de computação de larga escala. Neste esquema, um voluntário que já tenha alcançado um determinado patamar de qualidade no projecto (este patamar é atingido pela execução com sucesso de um número mínimo de tarefas) recebe convites que pode endereçar a aspirantes a voluntários. Por forma a incentivar a distribuição de convites, o utilizador que convida recebe um bónus em créditos correspondente a determinada percentagem dos créditos ganhos não só pelo utilizador que integrou o sistema via convite, mas também pelos convidados do convidado e assim sucessivamente até à geração N. N é um parâmetro configurável do sistema. Se $N = 1$, então apenas os convidados directos geram bónus. Contudo, para evitar que sejam convidados elementos de menos valia, os resultados errados submetidos por convidados (ou descendentes de convidados) penalizam o elemento que convidou. Deste modo, incentiva-se ao convite de utilizadores de qualidade, que são eles próprios incentivados a recrutarem elementos de qualidade e assim sucessivamente.

Contribuições da Dissertação

As principais contribuições desta dissertação são:

- Desenvolvimento da plataforma WindowsDDC que possibilita a execução organizada de programas apelidados de *programas sonda* em conjuntos de máquinas não dedicadas Windows, sem que para tal seja necessária a instalação de qualquer software nos sistemas remotos;

- Caracterização do nível de uso dos recursos computacionais de uma instituição académica, com constatação da existência de uma taxa de inactividade de CPU a rondar os 98%, confirmando a regra empírica dos 5% de uso, 95% de inactividade;

- Desenvolvimento do simulador `DGSchedSim` orientado para o estudo das políticas de escalonamento de aplicações distribuídas constituídas por tarefas independentes e executadas em ambientes de computação *desktop grids* não dedicados. As políticas de escalonamento são assentes na migração e replicação de tarefas baseado na partilha de ficheiros de *checkpointing*;

- Proposta, simulação e análise de esquemas de *checkpointing* empregues juntamente com várias metodologias de escalonamento na execução de aplicações distribuídas em ambientes *desktop grid* de área local e de área alargada;

- Proposta para a extensão do modelo usualmente empregue na computação em recursos não dedicados. A extensão consiste no estabelecimento de uma rede *peer-to-peer* hierarquizada em *nodos* e *super nodos*, organizados através de uma tabela de *hash* distribuída (DHT, *Distributed Hash Table*) ou, alternativamente, através de passagem de mensagens num ambiente *peer-to-peer*. O modelo procura melhorar o uso de mecanismos como o *caching* de dados de entrada e a partilha de *checkpoints*;

- Proposta, modelação e análise de uma metodologia de validação de resultados parciais, baseada na comparação de ficheiros de *checkpointing* provenientes de execuções redundantes. O objectivo é a detecção de falhas logo que essas ocorram, procurando diminuir o efeito nefasto das mesmas;

- Proposta e modelação de um sistema baseado em convites para o recrutamento de utilizadores que voluntariem os seus recursos computacionais para a execução de tarefas em projectos de computação pública. De modo a estimular a angariação de voluntários de qualidade – voluntários fidedignos que se esforcem por contribuir positivamente para o projecto – os angariadores recebem um bónus indexado ao contributo (medido em *créditos*) dos voluntários que convidaram com êxito. Contudo, de modo a evitar a angariação de voluntários não fidedignos, os erros de computação induzem penalizações não só nos donos/responsáveis pelos recursos, mas também nos indivíduos que angariaram os recursos que produziram resultados erróneos.

PALAVRAS-CHAVE: tolerância a falhas, *desktop grid*, computação em larga escala, escalonamento, confiabilidade, tolerância à sabotagem.

Acknowledgments

This journey was only possible because many people helped me along the way. This section is an attempt to express my gratitude to them.

First of all, I would like to thank my supervisor, Professor Luis Moura Silva from the University of Coimbra. His never ending energy, though questions and dynamism were an inspiration to me. Moreover, his persistence in creating research opportunities within CoreGRID was important for my work. A word of praise also for Professor João Gabriel Silva whose smart questions and comments in Athens provided extra motivation for this work.

A big thanks goes to Paulo Marques who was always available to answer my questions or to narrate an interesting episode about his many technical activities. I was also fortunate to collaborate with Filipe "Mac" Araujo, who had the strength for reviewing some chapters of this thesis. I would like also to thank (in no particular order), Artur Andrzejak (ZIB), Derrick Kondo (INRIA) and Bruno Sousa (CISUC) with whom I had the pleasure to cooperate with.

I cannot forget the friendship of Bruno ".Net" Cabral, with whom I shared many good (and healthy) laughs, and of José "Magic" Feiteirinha who, fortunately, is a much better programmer than magician.

I am grateful to the Engineering Informatics Department, the School of Technology of Management of Leiria and the Polytechnic Institute of Leiria for the conditions that allowed me to focus on research. My acknowledgments go not only to the institutions but also to the great people that work there – they are too many to cite them. Likewise, I would like to thank the Software and Systems Engineering Group, CISUC, the European network CoreGRID and PRODEP for their support.

Last but not least, I would like to thank all the anonymous people that are close to me, and who, in their own particular way, contributed to the success of this long journey. To all of them: *Obrigado e Saúde*!

Leiria, 2008
Patrício Rodrigues Domingues

Contents

Contents **i**

List of Figures **vii**

List of Tables **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 4

1.3 Organization of the Dissertation 5

1.3.1 Reading Map 7

1.4 Publication Record 8

2 Desktop Grids **11**

2.1 Motivation for Desktop Grids 11

2.2 Characterization of Desktop Grids 12

2.2.1 Generic Nomenclature 12

2.3 Strengths and Limitations of Desktop Grids 13

2.3.1 Strengths 14

2.3.2 Limitations 14

2.4 Components of a Desktop Grid 19

2.4.1 Desktop Computers 20

2.4.2 Communication Networks 20

2.4.3 Middleware for Desktop Grid 21

2.5 Types of Desktop Grids 23

2.6 Major Desktop Grid Middleware 25

2.6.1 Condor 25

2.6.2 BOINC 29

2.6.3 XtremWeb 40
2.6.4 GridMP 43
2.6.5 P2P-based Architectures 45
2.7 Summary 47

3 Resource Usage in Desktop Grids **49**
3.1 Introduction 49
3.2 The WindowsDDC Framework 50
3.2.1 Experiments and iterations 52
3.2.2 Remote Execution 55
3.2.3 Post-collecting code 56
3.3 Methodology and Monitored Metrics 57
3.3.1 Methodology 57
3.3.2 Monitored Metrics 57
3.4 Experiment 59
3.4.1 Computing Environment 59
3.4.2 Settings and limitations 60
3.5 Results 64
3.5.1 Machines Availability 66
3.5.2 Stability of Machines 68
3.5.3 Group Stability 70
3.5.4 User Sessions 72
3.5.5 Global Resource Usage 73
3.5.6 Weekly Analysis 76
3.5.7 Equivalence ratio 78
3.6 Related Work 79
3.7 Summary and Discussion 82

4 Fault-Tolerant Scheduling **85**
4.1 Introduction 85
4.1.1 Fault Tolerance and Checkpointing 85
4.1.2 Institutional Desktop Grids 87
4.1.3 Bag-of-tasks Applications 88
4.1.4 Turnaround Time of Bag-of-Tasks Applications 88
4.2 Scheduling Policies 89
4.2.1 Scheduler Knowledge 89
4.2.2 FCFS 90

4.2.3 FCFS-AT 91
4.2.4 FCFS-TR 92
4.2.5 FCFS-TR-DMD 92
4.2.6 FCFS-PRDCT-DMD 93
4.3 Segmented Execution with Shared Checkpoints 94
4.3.1 Example of a segmented execution 95
4.4 The DGSchedSim Simulator 97
4.4.1 Requirements 97
4.4.2 Input 98
4.4.3 Output 100
4.5 Summary 102

5 Evaluation of Fault-Tolerant Scheduling 103
5.1 Computing Environment 103
5.1.1 Machines 103
5.2 Trace 104
5.2.1 Characterization of the Trace 105
5.2.2 CPU Idle Threshold 107
5.3 Main Results 109
5.3.1 Ideal Execution Time 109
5.3.2 Simulated Tasks 110
5.4 Presentation of Results 112
5.4.1 Shared versus Private Checkpointing 112
5.4.2 Shared-based Policies 120
5.5 Related Work 125
5.6 Summary and Discussion 127

6 Sharing Checkpoints over Wide-Scale Desktop Grids 129
6.1 Introduction 129
6.2 Motivation 130
6.3 Overview 131
6.4 Description of *chkpt2chkpt* 135
6.4.1 Basic Components 135
6.4.2 Processing a Task 136
6.4.3 Starting and Resuming a Task 137
6.4.4 Separation of Processing and Storage 138
6.4.5 Managing the Checkpoints 140

6.5 Garbage Collection 141
6.6 Evaluation 141
6.7 Related Work 144
6.8 Summary 147

7 Desktop Grid Topologies for Sharing Input Data and Checkpoints 149
7.1 Introduction 149
7.2 Levels of Cooperation 150
7.2.1 Assessing the Benefits of Cooperation 152
7.3 Federating Institutional Desktop Grids 155
7.3.1 Overview 156
7.3.2 Functions of the LPS 157
7.4 Desktop Grids for Unrelated Peers 158
7.4.1 A Model for Grouping Unrelated Peers 159
7.4.2 Costs of Replicating Checkpoints 169
7.5 Related Work 171
7.6 Summary 175

8 Sabotage Tolerance through Comparisons of Checkpoints 177
8.1 Introduction 177
8.2 Results Validation Techniques 180
8.2.1 Majority Voting 180
8.2.2 Spot-checking 182
8.2.3 Credibility-based Validation 183
8.3 Assumptions and Definitions 184
8.4 Comparison of Equivalent Checkpoint Digests 185
8.4.1 Reducing the Time to Detect an Error 186
8.4.2 Theoretical Analysis 187
8.5 Checkpoint-based Task Replication 190
8.6 Experimental Results 192
8.7 Related Work 194
8.8 Summary 195

9 Reputation and Trust Management in Volunteer Computing 197
9.1 Introduction 198
9.2 The Problem of Identity 198
9.3 The Invitation System 200

9.3.1 Overview 201
9.3.2 Invitation Cards 204
9.3.3 Relationship Threshold Distance 204
9.3.4 Bootstrapping the Invitation System 206
9.3.5 Management Overhead 206
9.3.6 Collusion Avoidance 206
9.3.7 Preventing Misuse 207
9.4 Relationship Between Inviter-Invitees 208
9.4.1 Theoretical Analysis 208
9.5 Sharing Reputation Across Volunteer Projects 213
9.5.1 Implementation 215
9.6 Related Work 215
9.7 Summary 217

10 Conclusion and Future Work **219**
10.1 Conclusions 219
10.2 Main Contributions 221
10.3 Future Work 222

Bibliography **227**

List of Figures

1.1 Reading Map. 7

2.1 Distribution of active hosts. 16
2.2 Distribution of hosts per CPU architecture in SETI@home . . 19
2.3 Distribution of Machines per OS in SETI@home 19
2.4 Generic representation of BOINC. 32
2.5 BOINC's client-side components. 34

3.1 Overview of WindowsDDC architecture. 53
3.2 Machines sorted by their relative computing power. 61
3.3 Interactive sessions samples grouped by occurrence. 63
3.4 Count of powered on machines. 67
3.5 Machines' uptime and availability. 67
3.6 Average uptime per power cycle. 70
3.7 Stability of machines per classroom. 71
3.8 Count of stability periods over N machines. 71
3.9 Distribution of the duration of user-sessions. 73
3.10 Average CPU idleness. 74
3.11 Average CPU idleness after boot up. 75
3.12 Sum of free memory of the machines. 75
3.13 Cumulated free disk space of the machines. 76
3.14 Weekly distribution of powered on machines 77
3.15 Weekly distribution of free memory and network traffic. . . 78
3.16 Cluster equivalence ratio. 79

4.1 Optimal segmented execution of 5 tasks over 3 machines. . . 96
4.2 Example of a `DGSchedSim`'s graphical output. 101

5.1 Count of accessible machines. 105

5.2 Samples per machine per day and CPU idleness per machine 106
5.3 Variation count per machine and variation ratio 107
5.4 Variation count per machine and variation ratio for a 90% CIT 108
5.5 Variation count and variation ration for CIT=0% and CIT=90% 109
5.6 Turnaround for shared and private FCFS (0% CIT, weekdays) 113
5.7 Turnaround for shared and private AT (0% CIT, weekdays) . 114
5.8 Turnaround for shared and private TR (0% CIT, weekdays) . 114
5.9 Turnaround for shared/private TR-DMD (0% CIT, weekdays) 115
5.10 Turnaround for shared/private TR-PRDCT-DMD (0% CIT) . 115
5.11 Turnaround for shared and private AT (0% CIT, weekends) . 116
5.12 Turnaround for shared and private TR (0% CIT, weekends) . 117
5.13 Turnaround for shared and private FCFS (90% CIT, weekdays) 118
5.14 Turnaround for shared and private TR (90% CIT, weekdays) 118
5.15 Turnaround for shared/private TR-DMD (90% CIT, weekdays) 119
5.16 Turnaround for shared/private TR-PRDCT-DMD (90% CIT) 119
5.17 Turnaround for shared and private AT (90% CIT, weekends) 120
5.18 Turnaround for shared and private TR (90% CIT, weekends) 120
5.19 Slowdown ratios for 25/1800 tasks on weekdays (0% CIT). . 121
5.20 Slowdown ratios for 25/7200 tasks on weekdays (0% CIT). . 122
5.21 Slowdown ratios for 75/1800 tasks on weekdays (0% CIT). . 123
5.22 Slowdown ratios for 75/7200 tasks on weekdays (0% CIT). . 123
5.23 Slowdown ratios for 75/7200 tasks on weekends (0% CIT). . 124
5.24 Slowdown ratios for 75/7200 tasks on weekdays (90% CIT). 125
5.25 Slowdown ratios for 75/7200 tasks on weekends (90% CIT). 125

6.1 Components and interactions of *chkpt2chkpt* 133
6.2 Locating and retrieving a checkpoint (example) 137
6.3 Turnaround time with crash-recovery failures. 143
6.4 Turnaround time with crash-stop failures. 144
6.5 Turnaround time for varying checkpoint availability. 145

7.1 A Local Proxy Server in a IDG. 157
7.2 Architectural organization of SNBDG. 160
7.3 Recovery Protocol for the DHT-based Approach. 167
7.4 Recovery Protocol for the NMS-based Approach. 170

8.1 Probability of undetected errors on majority voting. 181

8.2 Three-segmented execution of a task 185
8.3 Benefit (W) as a function of the probability of error (p) 189
8.4 Benefit (W) as a function of checkpointing frequency (m) . . 190
8.5 Benefit (W) relative to expected maximum time (T) 193

9.1 Workflow of the Inviter-Invitee Relationship. 203
9.2 An Example of an Invitation System Tree. 205
9.3 $E(x,n,\theta)/W(x,n)$ ratio for the linear penalty model. 211
9.4 Effects of varying rates on the penalty (linear model). 212
9.5 $E(x,n,\theta)/W(x,n)$ ratio for the cubic penalty model. 213
9.6 Effects of varying rates on the penalty (cubic model). 214

List of Tables

3.1 Main characteristics of the monitored machines. 61
3.2 Samples grouped by their relative time occurrence. 63
3.3 Global resource usage 64
3.4 Average CPU idleness right after boot up. 74

4.1 Timeout tolerance factors for FCFS-AT and derived policies. 92

5.1 Sets of simulated machines and their performances. 104
5.2 Main metrics of trace *class* 108
5.3 Ideal Execution Time (IET) 110

6.1 Parameter definitions. 135
6.2 Settings of the experiment. 143

7.1 Execution statistics regarding several BOINC-based projects. 153
7.2 Effect on reducing by Δ_ε the failure rate P_ε 155

8.1 Parameter definitions. 181

9.1 Parameters, functions and constant definitions. 209
9.2 Penalty factors yield by varying α in the linear model. 211
9.3 Penalty factors yield by varying α in the cubic penalty model. 212

1 Introduction

In this opening chapter, we lay out the motivation for the theme of this thesis – *dependability mechanisms in the context of desktop grids*. We then present the main contributions of this thesis and outline its organization. We end this introductory chapter with the list of publications that support this work.

1.1 Motivation

It is a well known fact that many computing activities, specially the ones that are dependent on direct human input through a keyboard, a mouse or any other input device, barely load the machine, leaving plenty of unused CPU. For instance, the use of a word processor, a text editor or a spreadsheet program rarely demands more than a few percent of CPU usage[1]. There are several references in the literature like [Arpaci *et al.* 1995; Heap 2003; Anderson & Fedak 2006] that support the idea that average CPU idleness is well above 90%. As we shall see in Chapter 3, we also found out a near 98% CPU average idleness among Windows-based machines of an academic environment [Domingues *et al.* 2005b]. On top of that, the ever growing capabilities of PCs means that more powerful resources are left idle or unused. Interestingly, while many users have over-powered machines for their regular activities, and thus have plenty of cycles to spare, other users such as researchers and engineers are limited in their activities due to lack of enough computing power. Thus, in this context, the emergence of middleware to harness the resources of networked desktop machines, that would otherwise be left idle, appears as natural. These sets

[1]The most notable exceptions are graphical computer games, which are usually highly demanding on resources.

of machines whose unused resources are volunteered for hosting demanding computations are commonly referred as *desktop grids* [Chien *et al.* 2003; Anderson 2004]. However, besides the difficulties that are usual in distributed environments, desktop grids introduce an additional challenge at the level of availability: user-induced faults. Indeed, in desktop grids faults are the rule, not the exception. In part, this is due to a basic, yet fundamental, principle for using non-dedicated grid computing: the *owner* of the machine has full priority in accessing and using the machine, while applications submitted to the desktop grid middleware are run at lower priority. Thus, no interference whatsoever should be caused to the machine hosting a *foreign* application, with the whole process being totally transparent. In practice, this means that applications executed over non-dedicated desktop grid resources may be suspended or even aborted at any moment, like for instance, if a machine gets powered off, rebooted or its owner simply decides to no longer contribute to the desktop grid system[2]. Therefore, both the hosted applications and the supporting desktop grid middleware should be prepared to deal with sudden failures, independently of the cause of the failures: malfunctioning hardware or an owner claiming back her resources. To cope with failures in such unstable environments, middleware of desktop grids, like BOINC [Anderson 2004], XtremWeb [Fedak *et al.* 2001], and United Devices's GridMP [uniteddevices 2007], resort to several fault tolerance techniques such as *checkpointing* and *redundant computing*. Checkpointing consists in periodically saving enough state of an application to stable storage [Silva & Silva 1998]. Whenever the application needs to be restarted, the last stable and available checkpoint can be used to resume the application. Redundant computing means to replicate the computation throughout several independent machines, not only for coping with machines that might never finish the computation, but also for assessing the soundness of the computed results, comparing the results returned by the multiple instances. However, despite fault tolerance mechanisms like checkpointing and redundant computing that are supported by some desktop grid middleware, the usage of desktop grids is still encumbered by some limitations. Indeed, the main paradigm of most desktop grids is oriented toward the delivery of high throughput computing, with

[2]For instance, as of June 2007, out of around 1,032,000 registered users in BOINC-based projects, less than 338,000 were active, that is, they had contacted at least a BOINC server project in the last 30 days. (source: http://www.boincstats.com).

the systems geared to maximize the usage of resources instead of the execution speed of individual applications. This significantly hampers the use of desktop grids for executing applications with soft deadlines, making users whose applications require fast turnaround times to ignore desktop grids as a viable computing platform for their needs. Security, both from the perspective of the resource donor and of the resource user, is another issue that prevents a wider adoption of desktop grids, especially open desktop grids, that is, desktop grids comprising Internet's connected resources. Indeed, resources are vulnerable to malicious applications that can disrupt them, while desktop grid users can have their applications and the associated results tampered with.

In this thesis, we aim to enhance already existing fault tolerant mechanisms such as checkpointing and redundancy to improve the dependability provided by desktop grids in the execution of applications comprised of independent tasks. Specifically, we focus on improving turnaround time, reducing the time elapsed from the application submission to the completion of its last task. For instance, by sharing checkpoints amongst worker nodes over LAN and WAN environments, we strive to promote faster execution times by fostering a better usage of checkpointed computations, aiming to reduce the needs to redo computation. Likewise, by comparing sets of checkpoints from partial execution points, we aim to provide earlier detection of interrupted or faulty executions, in order to allow a prompter reaction, so that replacement tasks can be scheduled as soon as an abnormality is detected, again promoting a faster completion of the whole application.

Another major issue regarding desktop grids, especially the ones that are based on the paradigm of public volunteered resources [Anderson 2004; Fedak *et al.* 2001], is the level of trust that can be achieved relatively to the volunteered hosts that carry out the executions. In open systems like the Internet, anyone can volunteer computing resources in an anonymous way. Indeed, in most cases, only a valid email address is required. Such an email address can be easily created in one of the many free email services that exist. In this way, malicious users can attempt, and have done so, to deceive volunteer-based computing projects by faking results, and if caught can reengage on the project under a newly created email identity. In this work, we propose an invitation-based system upon which access to a volunteer computing project requires an invitation by an already contributing member. To held accountable inviters, they receive positive/negative credits according to the performance and behavior of their invitees. Therefore, both schemes promote the recruitment of good contributors to the system, and try to filter out the enrollment of malicious workers.

1.2 Contributions

The main contributions of this dissertation are:

- Development of the WindowsDDC framework. WindowsDDC allows the organized execution of user-supplied binary *probes* over a pool of non-dedicated Windows machines without requiring the installation of any software at the remote machines.

- Characterization of the level of computing resource usage in an academic desktop grid, where a near 98% CPU idleness average was found, confirming that the informal *95% CPU idleness rule* holds in these environments.

- Development of the trace-driven simulator `DGSchedSim` for the study of the effects on execution turnaround time of various shared checkpointing policies executed over institutional desktop grids.

- Proposal, simulation and analysis of shared checkpointing schemes coupled with several scheduling policies for the execution of applications over both institutional and wide-scale Internet-based desktop grids.

- Proposal for an extension of the standard model for desktop grid computing. The extension is based on a peer-to-peer network of super nodes, either organized under a DHT-based infrastructure or resorting to message passing between neighbor nodes. The model resorts to checkpointing to tolerate failures of super nodes.

- Proposal and modeling of an error detection mechanism that exploits the comparison of intermediate checkpoints from redundant executions to validate partial results. This scheme promotes an earlier detection of errors in computations, since divergent results can be spotted right after the first occurrence of the divergent checkpoints.

- Proposal for an invitation-based system for recruiting volunteers in public computing projects. The system rewards with bonus credits the recruiters who enroll active and well-behaved volunteers. Conversely, the system penalizes recruiters who enroll less reliable resource donors.

1.3 Organization of the Dissertation

This thesis is organized in three main parts:

1. *Presentation and Characterization of Desktop Grids*

2. *Checkpoint Management in Desktop Grid Environments*

3. *Fault Tolerance and Trust for Wide Scale Desktop Grids*

The first part presents and characterizes the concept of desktop grid. It includes Chapter 2 and Chapter 3. Chapter 2 sets the scene for the desktop grid topic, analyzing the main motivations behind the use of desktop grid and reviewing the most relevant concepts and systems. In Chapter 3, we characterize the resources typically found in networked pools of desktop computers, evaluating the potential of an academic-based desktop grid. First, we present the *Windows Distributed Data Collector* (WindowsDDC), a tool we developed for the purpose of automating the collection of resource usage traces in desktop grid environments. Then, as a case-study, we examine the resource usage collected for 77-consecutive days over 169 machines from 11 computing classrooms of an academic institution. We conclude that a high level of resource idleness exists on desktop machines, namely an average of 97.94% regarding CPU idleness.

The second part of the dissertation centers on the usage of fault tolerant mechanisms, namely checkpointing, to improve turnaround time of task parallel applications executed over desktop grids. It includes chapters 4, 5, 6 and 7.

In Chapter 4, we introduce fault-tolerant scheduling, proposing several checkpoint-based approaches to reduce the turnaround time of so called bag-of-tasks applications [Cirne *et al.* 2003] executed over single geographical institutional desktop grids. Specifically, we propose several scheduling policies, where worker nodes share checkpoints to more efficiently reuse computation, thus reducing the impact induced by failures. The chapter terminates with the presentation of the DGSchedSim simulator that was developed to evaluate the proposed scheduling policies. In Chapter 5, we assess through trace-driven simulations, the performances of the scheduling policies that we introduced in Chapter 4. We first describe the simulation scenarios, characterizing the trace, the machines and the set of tasks studied. Then, we present and analyze the main results, from the point of view of execution turnaround times.

In Chapter 6, we extend the concept of shared checkpoints to wide-scale desktop grid environments. Specifically, we present the *chkpt2chkpt* system that resorts to a distributed hash table-based (DHT) infrastructure for promoting the sharing of checkpoints among the workers of wide-scale, possibly Internet-wide desktop grid projects. The basic idea is to organize the worker nodes into a peer-to-peer (P2P) DHT, so that they can resort to this P2P organization to monitor the execution of the tasks, as well as tracking, sharing and managing checkpoint files.

Finally, in Chapter 7, we focus on alternative topologies for desktop grids. Specifically, we start by proposing a proxy-based model for local institutional desktop grids, with support for data exchange and indirect communication among worker nodes of a same local environment. We then tackle unstructured desktop grids, analyzing a peer-to-peer and super node-based model for extending the current centralized desktop grid middleware solutions. This concludes the second part of the thesis.

In the third part of the thesis, we broaden the horizon of our study, focusing on wide-scale systems like public computing systems that supports Internet-wide projects such as *SETI@home* [seti 2007], *Einstein@home* [einstein 2007] and *SZTAKI Desktop Grid* [sztaki 2007], to name just a few.

We target error detection in computation performed over desktop grids in Chapter 8. We present a checkpoint and replication-based error detection technique that simultaneously exploits checkpointing and redundancy. Specifically, we promote comparison of intermediate checkpoints in schemes that resort to n-replication for validating results. This way, we are able to speed up error detection in most of the cases, and therefore to promote faster responses to these errors by rescheduling the faulty tasks.

In Chapter 9, we analyze how social relationships and ties among volunteers can be used to improve the reliability of public computing volunteered infrastructures, namely at the level of workers. We propose an invitation-based system, upon which potential workers can only donate resources to a project if they are invited by nodes which had and are still significantly contributing to the system. Moreover, to promote good recruitment, inviters are made responsible for their invitees' behavior, receiving credit bonuses for positive performance, and penalties whenever recruited workers return erroneous results. Additionally, to promote trust among inviters and potential invitees, we propose a simple credential scheme, upon which a computing project can make available the main events and statistics related to a worker who has donated or is currently donating resources. To preserve the privacy of donors, the information can only be disclosed by the resource owners themselves when applying for an invitation to another project.

Finally, Chapter 10 concludes this thesis by summarizing the main results and suggesting possible directions for future research.

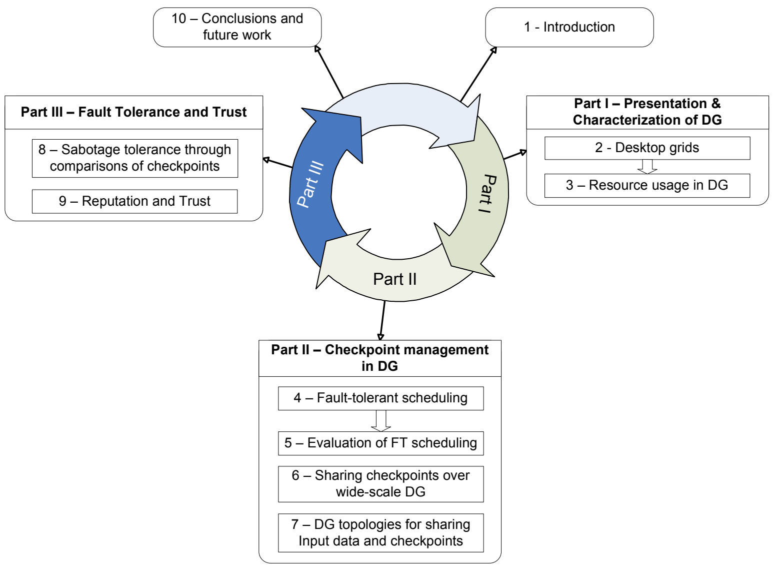

10 – Conclusions and future work
1 - Introduction
Part III – Fault Tolerance and Trust
8 – Sabotage tolerance through comparisons of checkpoints
9 – Reputation and Trust
Part I – Presentation & Characterization of DG
2 - Desktop grids
3 – Resource usage in DG
Part III
Part I
Part II
Part II – Checkpoint management in DG
4 – Fault-tolerant scheduling
5 – Evaluation of FT scheduling
6 – Sharing checkpoints over wide-scale DG
7 – DG topologies for sharing Input data and checkpoints

Figure 1.1: Reading map highlighting dependencies between chapters.

1.3.1 Reading Map

Although the thesis was devised for a sequential and full reading, the writing has also taken in account readers who might only be interested in a given subtopic. In this way, there is a varying degree of independence among chapters, with some subtopics being covered in a single chapter, and others spread over two consecutive ones. A reading map that highlights dependencies between chapters is shown in Figure 1.1 (page 7).

1.4 Publication Record

Subsets of the work towards this dissertation or related to it have been published in refereed international journals, conferences and workshops as follows:

- Patricio Domingues, Paulo Marques, Luís Silva. "*Distributed Data Collection through Remote Probing in Windows Environments*", in Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 59-65, PDP'05, Lugano, Switzerland, February 2005. [Domingues *et al.* 2005a]

  This paper presents WindowsDDC.

- Patricio Domingues, Paulo Marques, Luís Moura Silva. "*Resource usage of Windows computer laboratories*", in Proceedings of the International Conference Parallel Processing, pp. 469-476, ICPP 2005 - Workshops on Parallel Processing, Oslo, Norway, June 2005. [Domingues *et al.* 2005b]

  This paper studies the resource usage of desktop machines of an academic environment over 77-consecutive days.

- Artur Andrzejak, Patricio Domingues, Luís Moura Silva, "*Classifier-Based Capacity Prediction for Desktop Grids*", 1st Workshop of Integrated Research in Grid Computing - CoreGRID, Pisa, Italy, 28-30 November 2005. [Andrzejak *et al.* 2005]

  This paper assesses five classification algorithms over traces collected from classrooms of an academic environment.

- Patricio Domingues, Artur Andrzejak, Luís Moura Silva, "*Scheduling for Fast Turnaround Time on Institutional Desktop grid*", 2nd CoreGRID Workshop on Grid and Peer-to-Peer Systems Architecture, 16-17 January 2006, Paris, France. [Domingues *et al.* 2006c]

This paper reports the main results regarding the scheduling policies simulated over real desktop grid traces.

- Patricio Domingues, Paulo Marques, Luís Moura Silva, "*DGSchedSim: a Trace-driven Simulator to Evaluate Scheduling Algorithms for Desktop Grid Environments*", Proceedings of the 14th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'06), pp 83-90, Montbéliard, France. IEEE Computer Society Press, February 2006. [Domingues *et al.* 2006b]

  This paper describes the DGSchedSim simulator used to perform the trace-driven simulations of scheduling policies over desktop grids.

- Artur Andrzejak, Patricio Domingues, Luís Moura Silva, "*Predicting Machine Availabilities in Desktop Pools*", (short paper). Proceedings of IEEE/IFIP Network Operations & Management Symposium (NOMS 2006), Vancouver, Canada, 3-7 April 2006. [Andrzejak *et al.* 2006]

  This paper evaluates several algorithms for the prediction of the availability of machines that comprise a pool of desktop computers.

- Patricio Domingues, João G. Silva, Luis Moura Silva, "*Sharing Checkpoints to Improve Turnaround Time in Desktop Grid*", pp. 301-306, 20th IEEE International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06), Vienna, Austria, 18-20 April 2006. [Domingues *et al.* 2006d]

  This paper gives a broad overview of the sharing checkpoint concept.

- Patricio Domingues, Artur Andrzejak, Luís Moura Silva, "*Using Checkpointing to Enhance Turnaround Time on Institutional Desktop Grids*", Proceedings of 2nd IEEE International Conference on e-Science and Grid Computing, Amsterdam, The Netherlands, 4-6 December 2006. [Domingues *et al.* 2006e]

  This paper reports the main results regarding the scheduling policies simulated over a real desktop grid trace, introducing the concept of *CPU Idle Threshold* (CIT).

- Patricio Domingues, Filipe Araujo, Luís Moura Silva, "*A DHT-based Infrastructure for Sharing Checkpoints in Desktop Grid Computing*", Proceedings of 2nd IEEE International Conference on e-Science and Grid Computing, Amsterdam, The Netherlands, 4-6 December 2006. [Domingues *et al.* 2006a]

This paper describes *chkpt2chkpt*, a peer-to-peer based system for sharing checkpoints in a wide-scale desktop grid.

- Filipe Araújo, Patricio Domingues, Derrick Kondo, Luís Moura Silva, "*Validating desktop grid results by comparing intermediate checkpoints*". In Achievements in European Research on Grid Systems, CoreGRID Integration Workshop 2006 (Selected papers), pp 13-24, Krakow, Poland, October 2006. Springer-Verlag. [Araujo *et al.* 2006]

  This paper presents a result validation scheme based on the comparison of intermediate checkpoints.

- Patricio Domingues, Bruno Sousa, Luís Moura Silva, "*Sabotage-tolerance and Trust Management in Desktop Grid Computing*", in Journal of Future Generation Computing Systems, December 2006. [Domingues *et al.* 2007] (DOI:10.1016/j.future.2006.12.001)

  This paper gives an overview of existing sabotage-tolerance techniques and presents two sabotage-tolerance oriented mechanisms: the *Invitation System* and the *Reputation Sharing* scheme.

- Derrick Kondo, Filipe Araújo, Patricio Domingues, Luís Moura Silva, "*Result Error Detection on Heterogeneous and Volatile Resources via Intermediate Checkpointing*", CoreGRID Workshop on Grid Programming, Model Grid and P2P Systems Architecture Grid Systems, Tools and Environments, pp 72-80, Hellas Heraklion, Crete, Greece, June 2007. [Kondo *et al.* 2007]

  This paper broadens the approach of validating intermediate results via comparison of equivalent checkpoints.

2 Desktop Grids

In this chapter, we identify the main motivations for exploiting desktop grids. Then, we characterize our understanding of desktop grids, describing their basic components and pointing out their main strengths and weaknesses. In addition, we distinguish between the somewhat controlled and closed institutional desktop grids, and the open public desktop grids that support the volunteer-based *@home* projects [Bohannon 2005a]. Finally, we review the major middleware that support scavenging of desktop grid resources.

2.1 Motivation for Desktop Grids

Desktop grids are becoming increasingly attractive for performing computations. It is a well documented fact that desktop machines used for regular activities, ranging from electronic office operations (word processing, spreadsheets and other document preparation) to communication (email, instant messaging), and information browsing have often very low resource usage. Indeed, most computing activities, and especially the ones dependent on human interactive input, barely load the machines. Several studies confirm the 95% CPU idleness estimate that is normally associated to desktop machines, making these resources interesting for harvesting [Heap 2003; Domingues *et al.* 2005b].

Ironically, while a vast majority of users barely load their machines, others are always limited in their work and research because they do not have enough computing power, nor the budget to acquire and maintain it. Indeed, many problems in areas such as biology, medicine, cryptography, earth sciences, cosmology and high-energy physics, to name just a few, require massive computing power to be effectively tackled [Bohannon

2005a]. Thus, it appears logical that individuals or organizations in need of computing power and who cannot afford dedicated high performance computing facilities, aim to exploit resources that would otherwise be left idle. In fact, although resorting to desktop grids requires adapting applications to the desktop grid paradigms, the *return on investment* (ROI) for applications is frequently high [Anderson 2004].

2.2 Characterization of Desktop Grids

In the context of this thesis, the designation *desktop grids* refers to computational aggregates formed by non-dedicated desktop machines whose resources, such as CPU, memory, network bandwidth, storage space and, more recently, graphical processing unit (GPU) [Fan *et al.* 2004], can be harvested for running non-local applications in a loosely coupled fashion. Examples of desktop grids include the thousands of volunteer computers that support emblematic public computing projects such as SETI@home [seti 2007], Einstein@home [einstein 2007], Rosetta@home [rosetta 2006], QMC@-home [qmc 2007], Folding@home [Larson *et al.* 2002], Climateprediction.net [Stainforth *et al.* 2002] and SZTAKI Desktop Grid [sztaki 2007], among many others.

The usage of public resources for public computing projects is often designated as *public resource computing* (PRC) or alternatively as *public-based desktop grids* [Anderson *et al.* 2002; Martin *et al.* 2005]. Another form of desktop grid is related to the use of private computing resources existing at corporations and institutions such as academic campuses. This type of desktop grid is termed as *institutional desktop grid* or as *enterprise desktop grid* [Kondo *et al.* 2004a], or as *local desktop grid*[Balaton *et al.* 2007]. In this thesis, we study both public resource computing and institutional desktop grids. Next, we present the generic desktop grid nomenclature that is used throughout this work.

2.2.1 Generic Nomenclature

Although some variability exists across the various desktop grid models and implementations, they all share a common substrate. Next, we define the main nomenclature focusing on the logical elements that comprise desktop grids.

- **task**: the entity that gets executed by harvested resources. An equivalent designation found in the literature is **work unit** or **workunit** [Anderson 2004].

- **application**: an application is comprised of several tasks, possibly independent from one another and executed over different machines. The execution of an application is only completed when all of its tasks have been terminated.

- **worker**: the resource whose otherwise idle time is used for executing tasks.

- **resource donor**: the individual or institution who is making available the computing resources, inside which the workers run.

- **submitter**: the person or entity that submits an application. Contrary to some desktop grid related literature, we do not use the term *client* since it can be confused with the scavenged machine, that is, the machine that actually performs the computation.

- **supervisor**: this designates the software entity that controls the execution of an application. Typical functions of a supervisor include the scheduling of individual tasks, the reception of the results computed by the workers, and dealing with errors. Also, in some desktop grid environments, an application can only be submitted through the supervisor (this is often the case in public computing). The designation *server-side* or *master server* is also used with the same meaning.

- **manager**: this is the human entity that controls and manages the resources that run the supervisor and other related management software (for instance, databases for storing tasks and results, and verification software for validating the results). Furthermore, this entity authorizes the applications that can be submitted over the desktop grid infrastructure.

2.3 Strengths and Limitations of Desktop Grids

In this section, we discuss the main strengths and analyze the major limitations of desktop grids.

2.3.1 Strengths

Low cost: From the submitter point of view, the core advantage of a desktop grid environment is definitively the benefit of having access to a large computing power at low cost. In fact, even in institutional environments like an academic campus or a corporation facility, which have to support the costs of their infrastructures (power, cooling, etc.), desktop grids present the benefit of exploiting already existing resources, meaning that no major additional investment is required in hardware nor infrastructure[1]. At most, a couple of machines set as servers might be needed to support the specificities of the desktop grid middleware, along with some personnel attached to the management of the infrastructure. Thus, exploiting idle resources in the context of desktop grids improves the return on investment of these resources [Anderson 2004]. In public desktop grids, the ratio *computing power/cost to submitter* is even more favorable, since the costs are distributed among the volunteers, with each resource donor supporting her own machine(s). An interesting issue regarding public-based desktop grids, is that, even considering a stable population of resource donors, the potential computing power naturally progress over the years, due to the upgrades that owners regularly perform in their machines.

Computing power: Especially in large public desktop grids that assemble tens of hundreds of machines, the available computing power can be immense. In fact, two of the most popular public projects, SETI@home and Einstein@home report, at the time of this writing, 304,812 and 81,800 active computers, respectively. Combined together, these computers boost a performance of 365.1 TFlops and 88.2 TFlops, respectively[2].

2.3.2 Limitations

Although desktop grids present some clear advantages, namely, low cost and high computing power, they are also hindered by several limitations. Next, we describe the main restrictions of desktop grids.

Volatility: Volatility of resources is a major limitation of desktop grids. Indeed, besides the failures that normally affect computers, the tasks run-

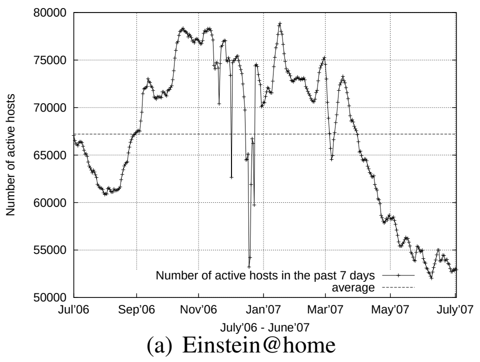
[1]This might not always be the case: when pushed near their full capacity, computers generate more heat and noise. This way, certain rooms that are suited for handling regular usage of PCs, might require air cooling for dealing with intensive desktop grid usage.

[2]http://www.boincstats.com, June 2007.

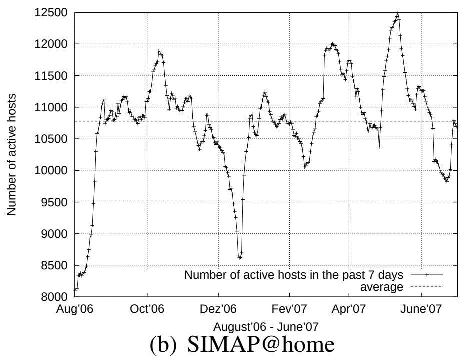
ning over desktop grids are prone to get interrupted when resource owners claim back their machines. In desktop grids, failures are not the exception but the rule. Choi et al. categorize failures of harvested resources in two main classes: *volatility failures* and *interference failures* [Choi *et al.* 2004]. The former comprises failures like network and machine crashes. The latter is a consequence of the shared nature of resources where the interactive users of a machine have priority over volunteer computation in accessing the resources of the machine. Note that some desktop grid middleware allow resource owners to define the conditions that mark an owner claiming back her machine's resources and implicitly forcing the interruption of any *foreign* task that might be running in their machine [Thain *et al.* 2005; Anderson 2004]. It can be, for instance, the simple existence of keyboard and/or mouse activity, or that the owner's usage of resources, namely CPU, is above a given threshold, or simply a combination of both (input devices and CPU). Other owners might be more permissive and let foreign tasks run jointly with their own local applications, relying on the operating system priority mechanisms for non-obtrusiveness of foreign tasks (hosted tasks are usually run at the lowest priority). From the point of view of a hosted task, the cause of the failure is normally irrelevant since the net outcome is that the task gets evicted from the resource.

Figure 2.1(a) plots the evolution of the number of active hosts participating in the Einstein@home project between July 2006 and June 2007. It can be seen, that the number of active hosts varies between 52,000 and slightly above 79,000 (a 50% variation), averaging to 67,208 machines (represented by the flat line), with a standard deviation of 7806.73. The plot gives a rough example of the volatility of resources in a wide-scale public computing project. The sharp drops of the plots seen at the end of 2006 correspond to somewhat long downtime periods (days) of the *Einstein@home's* server side that occurred at this time.

A similar example of the resource volatility of public computing is given by Figure 2.1(b), which plots the number of active hosts for SIMAP@home [simap 2007] for the period comprised between August 2006 and the end of June 2007. SIMAP@home is yet another BOINC-based public project, which aims to build a database of precomputed matrix of protein similarities. During the observed eleven months, the count of active workers varied between a minimum of 8,000 to a peak of almost 12,500, averaging to 10,768 (shown in the plot by the flat line), with a standard deviation of 816.80.



(a) Einstein@home



(b) SIMAP@home

Figure 2.1: Evolution of the number of active hosts of the Einstein@home and SIMAP@home projects throughout the 2^{nd} half of 2006 and the 1^{st} half of 2007.

Security: Security is another serious issue of desktop grid. In such environments, threats exist for both the donated resources and the submitters. Indeed, the former can be harmed by running a malicious foreign task, either explicitly (the attack disrupts the machines), or in a hidden way, for instance, with the privacy of the host machines silently jeopardized by *botnets* [McCarty 2003]. Security issues also exist relatively to submitters, since a worker might spy upon the task it is executing. Worse, a malicious local user might temper with the task data or/and algorithms, trying to extract (for example, via reverse engineering) any valuable information. Additionally, results can be silently faked, in order to disrupt the computation, or simply for claiming undue credits for work not performed as it occurs in public computing projects [Molnar 2000].

All of these are serious issues that need to be properly addressed. As we shall see in section 2.6, *sandboxing*, resource virtualization, code signing, encryption of data and communications, and replication with result comparison are some of the techniques implemented by middleware environments to provide for a certain level of trustworthiness and security.

Communication model: Depending on the underlying network infrastructure and on the level of trustworthiness amongst donated peers, communications can be another limitation of desktop grids. Internet-based computing [Cappello *et al.* 2005] and particularly public-resource computing [Anderson *et al.* 2002; Martin *et al.* 2005], present a much more challenging communication model. Indeed, over open and consequently non-trusted networks, several issues arise that seriously limit the type of application that can be run over desktop grids. First of all, many machines do not have a public IP address since they are behind *network address translation* services (NAT). Jointly with strict firewall rules, many hosts that access Internet have asymmetrical connectivity: they can perform connections to outside hosts, but only over well-defined ports, such as the HTTP port and alike [Son & Livny 2003]. Furthermore, such hosts are not addressable by outside peers, and thus they can only act as clients.

Besides addressing and reachability issues, bandwidth and limitation over network traffic can pose additional restrictions on desktop grid. In fact, despite bandwidth growth and the emergence of broadband, Internet connections are still significantly lower than local area ones. Moreover, a significant number of users with broadband connections have an upload bandwidth much lower than the download one, causing a further network asymmetry. For these reasons, communication is severely limited in desktop grid environments. This restricts the applications that can be run over harvested resources to coarse-grained ones, with tasks having a high CPU to communication ratio. Additionally, tasks need to be totally loose from one another, with no communication occurring among them. This way, for success, applications run over desktop grids need to follow the embarrassingly parallel paradigm, upon which tasks are completely independent from one another. Although this constitutes a serious restriction, many research, industrial and scientific applications follow this paradigm, and, in fact, no scarcity of applications seems to exist for desktop grids [Bohannon 2005a].

Heterogeneity: A subtle limitation that arises in desktop grids relate to the heterogeneity of platforms. Indeed, if the code executed by workers is implemented through a native binary, and the desktop grid aggregates several types of non-compatible platforms, like Windows, Mac OS and Linux as it is frequently the case, exploiting all of these platforms require the development and maintenance of several trees in the source code, although this can be somewhat mitigated by writing portable code. Another platform-related effect respects the accuracy of numerical computations, with different hardware and operating system platforms potentially yielding approximate, yet different results[3]. Depending on the application, this might need to be taken into consideration by the submitter when validating and analyzing the results [Taufer *et al.* 2005a].

A possible way to deal with heterogeneity of platforms is to resort to interpreted code run over software virtual machines, like for instance, JAVA or .NET. Such approach solves the portability issues, and may also provide additional security thanks to the virtual machine sandboxing mechanism, but poses two additional problems:

1. Need for volunteer resources to have the proper virtual machine installed.

2. The performance penalty that is normally imposed by software-level virtual machines.

Another approach yet is to resort to operating system level virtual machines like VMWare [vmware 2007], QEMU [Bellard 2005], Xen [Barham *et al.* 2003] and VirtualBox [VirtualBox 2007], to name just a few. These virtual environments isolate the foreign task from the hosting machine, protecting them from each other. This approach allows to setup an execution environment tailored for the harvesting task, like for instance, a different operating system than the one running on the hosting machine [Figueiredo *et al.* 2003]. Drawbacks include the need to download, install and setup the appropriate virtual machine, the cost of the virtual machine software (although free and open source solutions exist like QEMU and Xen) and the resource overhead it imposes on the host machines.

Figure 2.2 and Figure 2.3 (page 19) illustrate the level of heterogeneity in the volunteer hosts of the SETI@home project. Specifically, Figure 2.2 plots the distribution of the CPU architectures of the hosts participating in the project. The data refer to the 932,696 most productive machines registered in the project as of August 2006. Note that the x86 architecture comprises more than 96% of the machines. Figure 2.3 depicts the distribution of machines accordingly to their operating systems. Although Windows variants represent more than 87%, and thus, it might appear that low heterogeneity exists, it can be seen that Windows variants are themselves spread in

[3]For certain numerically sensible algorithms, the cumulated divergences in computation performed in different platforms yield non-compatible results.

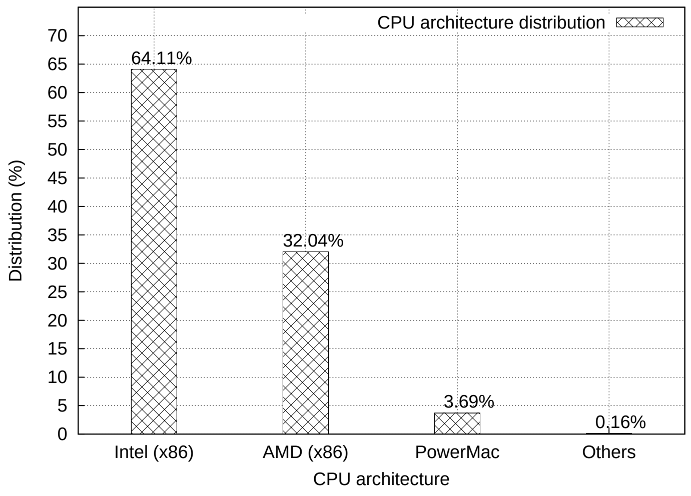
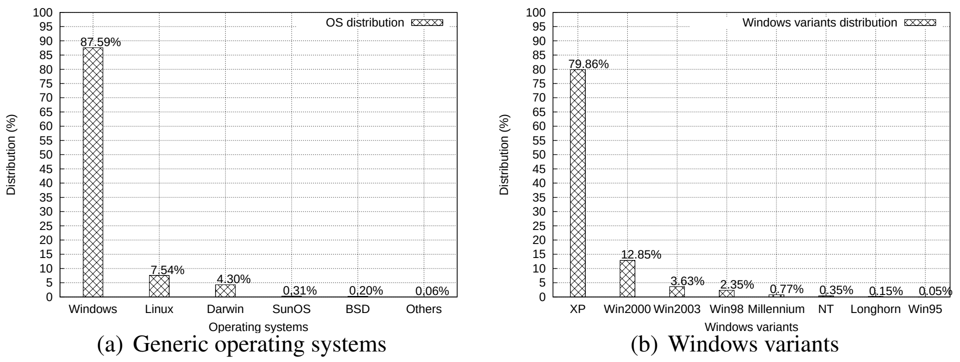


Figure 2.2: Distribution of hosts per CPU architecture in the SETI@home project. (source: SETI@home)

several versions (Figure 2.3(b)). From the plotted data, it emerges that heterogeneity is quite high at the OS level.



(a) Generic operating systems (b) Windows variants

Figure 2.3: Distribution of Machines per Operating Systems in the SETI@home. (source: SETI@home)

2.4 Components of a Desktop Grid

The main components required for setting up a desktop grid are (1) the desktop computers, (2) the underlying communication network, and (3) the middleware that glues everything together. Next, we succinctly describe each of these components.

2.4.1 Desktop Computers

In the context of desktop grids, the expression *desktop computers* designates the machines whose idle resources are to be harvested. The attractiveness of personal computers is strengthened by the continuous growth of their capabilities. Indeed, after more than 40 years, Moore's law still holds on [Schaller 1997], meaning that roughly every 18 months, the number of transistors available for CPU implementation doubles. Moreover, besides CPU, other core elements like network bandwidth (9 months to double) and space storage (12 months to double) continue to grow in size. Unfortunately, even if main memory have also become larger, the speed gap between CPU and the memory system continues to widen, although clever memory hierarchies with multiple in-chip caches and alike have allowed to partially mitigate the effects of the memory-CPU speed gap on performance[4]. Thus, not only are resources significantly idle, but they keep getting faster and faster. Note that, although performance improvement by raising the clock speeds of CPUs seems, at least for the time being, over or much limited [Sutter 2005], the trend towards multi-core CPUs appears to indicate that the level of unused computing power will continue to increase. Indeed, possibly entire cores will be available to exploit, further reinforcing the attractiveness of cycle scavenging schemes [Creeger 2005].

2.4.2 Communication Networks

Besides idle machines, another major component needed for proper implementation of desktop grids is a communication infrastructure. In fact, isolated individual machines, even powerful ones, are of low use if they can not communicate with each other. Thus, the emergence and fast evolution of affordable network technologies, which deliver bigger and bigger bandwidths over wide areas, has allowed the creation of powerful aggregates holding a large number of machines under a desktop grid infrastructure. In particular, the Internet with its fast and sustained growth has made possible the successful deployment of large volunteer desktop grid communities that support major public computing projects such as the pioneer SETI@home [seti 2007].

[4]In fact, a significant part of the transistors implemented in a CPU chip are devoted for fast on-chip memory caches. Moreover, a cache-level is added to the memory hierarchy roughly every decade.

It is important to note that most desktop grid environments do not provide communication among workers. Indeed, these environments only support applications where each worker solely communicates with the master worker. This type of applications is known as *embarrassingly parallel computations* or *pleasantly parallel computations* [Wilkinson & Allen 2004].

2.4.3 Middleware for Desktop Grid

The success of desktop grids over the last decade is not only due to hardware improvement in the area of computers and networks. Indeed, the emergence of some sound middleware solutions has contributed significantly for the deployment and exploitation of desktop grids, considerably facilitating the setup and maintenance of desktop grid environments. Examples include Condor [Litzkow *et al.* 1988], XtremWeb [Fedak *et al.* 2001], BOINC [Anderson 2004], United Devices [uniteddevices 2007] and DataSynapse [datasynapses 2007]. Next, we describe the main issues that a desktop grid middleware needs to address (we provide a detailed review of the major desktop grids frameworks in section 2.6).

- **Non-interference:** It is of paramount importance for the acceptance and success of a desktop grid environment that tasks running in volunteered machines do not interfere with the local workload. This means that interactive users of the machines must retain full priority over hosted tasks. Furthermore, the desktop grid system should allow the definition of an harvesting policy, allowing the local user to configure parameters related to the volunteering of the machines. Such parameters might include, amongst other things, the host operating system priority level to be used by scavenging tasks (usually hosted tasks are run at the lowest available priority), the allowed time frame (time of day and days of the weeks) for harvesting and what should occur to a hosted task when a machine is claimed by its local user (should the task be evicted, or on the contrary, kept in memory). In all cases, a desktop grid framework should minimize any interference it might cause on local resource users, even if this causes negative impact on the hosted task. Indeed, the acceptance of volunteer mechanisms is tightly linked with the perception of resource donors that they remain in full control of their machines.

- **Usability:** Usability is an important feature both for system installers and managers, and also for task submitters. Actually, deploying a desktop grid infrastructure should be painless and require minimal effort and resources from the coordinator side. The same should occur for normal management activities, which should not require excessive skills nor excessive work. Also significant, are the capabilities required for task submitters. Ideally, few skills should be required to task submitters, with the system adapting itself to submitters and their tasks. In this area, a much appreciated feature is the support of applications *as is*, without requiring burdensome adaptations or the mandatory use of a given programming language and related APIs.

  More importantly, volunteering resources should be an easy process. Indeed, solutions that demand too much effort will most certainly deter potential donors.

- **Scalability:** A desktop grid middleware should scale accordingly to the number of involved resources, in the sense that more resources should mean proportionally more available computing power. Additionally, management needs should not adversely suffer from the scale effect, in the sense that adding resources should not mean additional management effort.

- **Security:** As stated before, security in the context of desktop grids is a two-way issue, since both the resource donor and the submitter need to be considered. The former should be certain that the donated resource will not be tampered with. For instance, local private data will not be accessed nor the machine will be made more vulnerable to malicious software. On the other hand, the submitter should also receive guarantees over the integrity of the computations and over the inviolability of her data running on volunteered resources. As we shall see in section 2.6, security is a complex issue, with many desktop grid frameworks addressing security issues in a less than perfect way. Thus, the volunteering of resources and submission of tasks still require a fairly amount of implicit trust between donors and harvesters.

- **Fault Tolerance:** Any computer-based system is prone to fail. In a desktop grid environment, where the number of machines can quickly reach high figures, this is even more real, since for a constant probability of failures, having a large population of computers means that a significant number of machines will fail. Moreover, as stated before, since resources are donated under best effort with no guarantee of availability whatsoever, volatility of resources is significantly higher than it is with dedicated machines. Thus a proper desktop grid environment should be able to cope with the normal types of failures in a transparent way. Two main mechanisms for coping with failures are *checkpoint-and-restart* [Silva & Silva 1998] and task *replication* [Du *et al.* 2004]. Checkpoint-and-restart aims to limit the computation that needs to be redone when recovering from a failure, while task replication is also employed for validating results through majority voting.

- **Accountability of resource usage:** Although not a critical factor of success, the accountability of harvested resources can be important for assessing the real value of the harvesting system. Moreover, in public volunteered desktop grids, accountability of donated resources allows to establish donor rankings, where top positions are yielded to most active volunteers. Surprisingly, these rankings and all the dynamic surrounding them are known to substantially stimulate the enthusiasm of volunteers [Holohan & Garg 2005].

2.5 Types of Desktop Grids

Two main types of desktop grids can be considered: *institutional* and *Internet-based*. The former involves resources that are private to an institution and geographically located at a single site, for instance a corporation or an academic campus. On the contrary, as the name implies, the term *Internet-based* designates larger systems running over the Internet [Bohannon 2005a] like SETI@home, Einstein@home and Folding@home [Folding@home 2006], just to cite a few.

Institutional desktop grids present the advantage of a centralized management entity, who normally decides the sharing policy to be applied to the resources. For instance, if the management of a corporation determines that all machines must be available for harvesting purposes, then all users have to comply. However, for a better acceptability of the sharing paradigm and to foster cooperation and positive attitude, users should also benefit, even indirectly, from the resource sharing. Indeed, the human factor is important, and many individuals have ownership feelings about the machine that is allocated to them. Thus resource sharing needs to be properly introduced in such environments.

Additionally, in institutional environments, security issues might be easier to deal with, since core infrastructures are private and thus can be followed under a close control. Furthermore, institutional desktop grids are normally confined to a single geographic point, with all resources connected by local networks which are way faster than connections that link resources of public desktop grids. This allows to run applications with relatively high demands on networks, like for instance, applications that process massive amounts of data.

On the other hand, resources involved in public desktop grids are exclusively controlled by their respective owners and thus are much more unpredictable. For instance, a user might suddenly withdraw her machine from a public project to move it to another one, or to simply abandon resource volunteering altogether. Security measures need also to be stricter, since malicious resource donors under the cover of anonymity might try to sabotage the computation. Conversely, a dishonest submitter might try to submit a malicious application. Regarding connectivity, many workers might not be directly addressable because of firewalls and NAT schemes. Thus, connections need to be worker-initiated and over authorized ports (usually the HTTP port).

Although less stable and more prone to security problems, the number of resources involved in an appealing public computing projects might be orders of magnitude higher that what can be achieved at the scale of an institutional desktop grid. Gathering a large community of users requires creativity, like for instance, appealing screensavers and interesting problems to tackle, in order to persuade volunteers to join the public infrastructure. Furthermore, while institutional environments have to support their own additional costs, like electric power to run the machines and the associated cooling equipment, public-based desktop grids allow, at least from the point of view of submitters, a much more attractive economical model, since operating costs are distributed amongst volunteers, except for the expenses for supporting bandwidth, servers and personnel related to the supervisor side.

2.6 Major Desktop Grid Middleware

In this section, we review some of the most successful middleware platforms for desktop grids. We start by Condor and move on to examine the BOINC framework which is used by a large number of public computing projects. We also analyze the SZTAKI Desktop Grid extension to BOINC. We then review XtremWeb and its security-based approach to desktop grid computing. Finally, we analyze the commercial middleware GridMP.

To ease comparisons of the frameworks, our review focuses mostly on the parameters enumerated in section 2.4.3 (page 21), namely, *level of interference*, *usability*, *scalability*, *security*, *fault tolerance*, and *resource usage accountability*.

2.6.1 Condor

The Condor project was created in the mid-1980s at the University of Wisconsin-Madison by the Condor Research Project, and is considered as the pioneer framework in the area of harnessing CPU cycles [Litzkow *et al.* 1988]. The system has been continuously updated and adapted to new computing environments. It currently supports the major operating systems, such as some flavors of Unix, Linux, Windows and Mac OS. Condor is used to scavenge cycles in many computing sites throughout the world. Although it has some support for exploiting resources over wide areas and can be coupled with wide-area Internet-resource computing schemes like XtremWeb [Lodygensky *et al.* 2003b], Condor really excels in local area environments.

Condor is a workload management system oriented for high throughput computing, which provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their tasks (*jobs* in Condor's jargon), and Condor places them into a queue, chooses when and where to run based on resource availabilities and tasks demands, monitors their progress, and informs the submitter upon completion [Tannenbaum *et al.* 2001]. The Condor system aggregates resources in *pools*. Pools are usually delimited by logical and geographical boundaries. For instance, the resources at an university campus can be aggregated by departments, with a pool existing per department. To better use resources, pools can be linked and configured to cooperate with one another, with idle pools receiving tasks from pools that are too overloaded to deal with them. This mechanism is called *flocking* [Tannenbaum *et al.* 2001].

In Condor, management of resources and tasks revolve around *ClassAds*. This mechanism is used by the system to represent the characteristics and constraints of both machines and tasks. For instance, the characteristic of a machine are expressed by a ClassAd (for example, "720300 KFlops, 1024 MB of main memory"), jointly with the restrictions that might be imposed by the machine's owner regarding access for scavenging ("weekdays, 9pm-8am"). Similarly, the requirements of a task are specified via its own defined ClassAd[5]. The scheduler, which has a global knowledge of the ClassAds, tries to match the tasks related ClassAds with resources' ClassAds.

Architecture

A Condor pool is organized in a centralized way, with the *central manager* machine of the pool coordinating the activities of the workers and of the submitter machines. The main role of the central manager of a pool is to match demands (i.e., submitted jobs) with available resources. Since Condor services are organized through daemons (normally, a daemon per service), we briefly analyze the main supporting daemons of Condor [Tannenbaum *et al.* 2001].

- *condor_master*: this daemon runs at every machine that participates in Condor (independently of the machine's role). It acts as a meta-daemon, being responsible for keeping the rest of the local Condor daemons running on the machine, as well as updating them, whenever a new version is detected.

- *condor_startd*: this daemon represents a machine to the Condor pool, advertising the machine's capabilities and policies through ClassAd. It runs at workers.

- *condor_starter*: this service deals with executing a Condor task in the local machine and gets activated whenever the machine is selected to

[5]For example, the ClassAd `(OpSys="WINNT50" || OpSys="WINNT51") and (Memory >= 400)`, means that the task requires either Windows 2000 or Windows XP and at least 400 MB of memory to run.

run a job. It sets up the execution environment, monitors the job once it is running and cleans up after the execution has terminated.

- *condor_schedd*: this daemon allows the submission of jobs at the local machine, and thus, it only runs at machines configured for submissions of jobs.

- *condor_shadow*: the *shadow* daemon is activated at a submission machine whenever a job submitted at the machine is running. This daemon serves requests for files to transfer, logs the job's progress and reports statistics when the job completes.

- *condor_collector*: this daemon collects all the data regarding the status of a Condor pool. It does so by receiving the ClassAd which are periodically sent by the other Condor daemons. This daemon runs solely at the central manager.

- *condor_negotiator*: the negotiator daemon is responsible for all the matchmaking within the Condor system. Like *condor_collector*, it runs solely at the central manager.

Analysis

Non-interference: With its rich set of configurations, Condor allows resource donors to define the level of obtrusiveness they want to tolerate. For instance, it is possible to define that a resource should only be occupied if its local workload is below a given threshold, and that foreign tasks are only allowed to run after a certain time interval has elapsed with no keyboard or mouse activity detected at the local machine. For example, the default configuration only allows the execution of a foreign task on machines which have more than 15 consecutive minutes of keyboard and mouse idleness, and whose local workload is below 0.15. As soon as one of these conditions no longer holds, the task is withdrawn.

Usability: As the result of its long experience gathered over more than 20 years in the field, Condor is highly usable. The installation for resource donor is relatively straightforward, even for non-computer savvy, and as stated before, its default configurations protect resource owners. Regarding submitters, a major benefit is that regular binaries, providing they can be run unattended (that is, requiring no human intervention), can be executed with no changes. Additionally, Condor has support for running Java binaries, since it detects any Java virtual machine that exists at the available resources.

Although mastering the details for the submit files needed for submitting tasks might require time and a certain level of expertise, the submission process is trivial. Submissions can be made from any machine that was installed with this capability. An annoyance is the nonexistence of a graphical or web interface that would allow the monitoring and management of running tasks, although Condor has a rich command-line interface that can be scripted to automate procedures.

Scalability: Since it was originally developed for local area environments, a Condor pool is not suited for wide area networks, although several sites can have their respective pools cooperating with one another, as long as some adjustments are made to sites' firewall and NAT policies. Indeed, Condor supports the aggregation of pools through the so-called *flocking* mechanism, but the installation and setup is not trivial, at least when compared with the installation of a local pool [Tannenbaum *et al.* 2001]. Additionally, the centralized approach to resources and tasks management somewhat limits its scalability, although a single pool is said to be able to hold around hundreds of machines [Thain *et al.* 2005]. Due to limitations in the implementation of the *condor_schedd* daemon, a Condor pool requires a balanced set of submitting machines to perform efficiently, otherwise the whole system performance degrades quickly [Beckles 2005]. Moreover, a normal reaction of users when faced with a Condor performance degradation is to query the system about their submitted jobs, which causes even more performance degradation.

Security: Security has not been a top priority of Condor, in part due to its targeting of institutional environments, which are assumed to be trustworthy. Therefore, Condor security mechanisms are sufficient for so called *safe environments*, where users are supposed to cooperate with one another [Beckles 2005].

Fault Tolerance: Being a framework oriented for non-technical users, Condor has developed a fault tolerance oriented culture to allow smooth management of the system, incorporating some fault tolerance mechanisms. For instance, its Unix version implements automatic and transparent checkpointing of applications [Litzkow *et al.* 1997]. Condor can be set to periodically checkpoint a task either to local storage or to a pool-wide checkpoint server. The checkpointing is performed at the system level, with the whole image of the process that runs the task being saved to stable storage. Although the checkpoint mechanism comports some minor restrictions, like the need of the application to be linked against Condor special libraries, the impossibility to use certain system calls like *fork()* and its relatively large size of checkpoint files, it is one of the few solutions that provides usable and transparent checkpointing. At the harvested resource level, Condor's processes are hierarchically organized, such that upon detection of the failure of a process, an upper-level one can regenerate it. Only the failure of the master process (*condor_master*) renders the resource unusable, from a Condor point of view, at least until the next restart of the master process.

A problem, which arises from Condor's centralized architecture, lies in the consequences of a somewhat prolonged failure of a pool's master machine. Indeed, upon a failure at the pool's server (for example, the server crashes or is cut off from the network), the running tasks are lost, although they are automatically restarted whenever the server recovers.

Other examples of batch systems for scavenging cycles in local environments include Sun Grid Engine (SGE) [Gentzsch 2001] and Alchemi.Net [Luther *et al.* 2005]. Both exhibit strong similarities with Condor and for this reason, we will omit the details.

2.6.2 BOINC

The Berkeley Open Infrastructure for Network Computing (BOINC) is an Internet-wide distributed computing middleware which aims to harvest computing resources [Anderson 2004]. BOINC was developed by the SETI-@home's team to provide a generic solution for exploiting desktop grid resources. Indeed, before BOINC, public and Internet-wide computing-intensive scientific projects such as GIMPS [gimps 2007], distributed.net [Distributed.net 2007] and the first version of SETI@home, had to develop the whole infrastructure, instead of focusing solely on their applications. This was cumbersome, time consuming, and clearly inefficient. BOINC was created precisely to ease the setup and management of public computing projects, letting submitters to focus on their applications. Furthermore, the BOINC platform also eases the procedures for resource donors, allowing owners to share their resources among several projects, without much burden.

A major asset of the BOINC middleware is the fact that its development is being coordinated and pushed forward by real desktop grid users, namely the SETI@home's team. Moreover, its adoption by other major public computing projects have widen its base of users, and brought new real problems to solve. Indeed, due to its open source nature, other users are able to contribute. Thus, the resulting platform is highly usable, providing out-of-the-box solutions for many problems that arise on wide public desktop grids. BOINC provides the whole software needed for implementing a full desktop grid project.

The BOINC framework acts at the level of a *project*, where a project is setup by an organization and comprises not only the application(s)[6] whose execution is sought across the volunteer resources, but also the recruitment phase, where resource donors are recruited to participate in the project. Indeed, to participate into a BOINC project, a resource donor must install the client-side software and register the resources to volunteer so that they can ask for tasks to process from the selected project. A project is accessible at its master URL, which is the home page of the project's web site. This URL also serves as a directory of scheduling servers.

Before recruiting volunteers, a BOINC project requires a server-side infrastructure (*server complex* in the BOINC jargon) able to support the server processes, like management and distribution of tasks to workers, reception and validation of results, registering and unregistering of volunteers, just to name some of the operations that are the responsibility of the BOINC server-side. Specifically, the server complex involves hardware (server machines and network bandwidth), and all the software infrastructure (database, different processes, etc.) provided by the BOINC middleware, which needs to be configured for the project. In addition, the coordinators of a project need to provide the application that effectively executes tasks at the volunteer resources (possibly in several versions to support distinct platforms). This application needs to be integrated into the BOINC platform,

[6]Note that a project can involve one or more applications, and the set of applications can change over time.

requiring that it calls some of the BOINC's functions, like *boinc_init()* and *boinc_finalize()* and others if more than a basic integration is sought[7].

Note that although both Condor and BOINC aim to exploit idle resources, their targets are quite different. As stated before, Condor is oriented toward easy exploitation of local area resources. Conversely, BOINC focuses on public computing, and it is much more labor intensive for task submitters. For instance, deploying and maintaining a BOINC project requires important setup, maintenance and recruitment efforts. In fact, it is not unusual for a BOINC public project to have a somewhat long beta phase (several months) to iron out issues regarded to the worker application(s) and infrastructure. Moreover, BOINC creates an asymmetric scavenging system, since all donors which have they resources working for a project are themselves unable to profit from the computing power for running their own applications.

The advantage of BOINC stems from the potential of attaining a much wider audiences. Interestingly, Condor supports seamless integration with BOINC, meaning that a machine that integrates a Condor pool and has the BOINC client installed, will have its idle time harvested for any BOINC-based project the machine might be attached as long as no Condor task exists than can be run at the machine.

Architecture

The BOINC middleware is organized into two main modules: the *server-side* and the *client* part. BOINC follows a reverse client-server model, which as the name implies, is the traditional client-server with reversed roles. So, instead of having a client that requests services from a server, in the reverse client-server model, the client actually requests tasks (*workunits* in BOINC's jargon) to process, and thus performs work for the server. Next, we describe the server-side complex module and then the client module.

Server-side The BOINC server-side, also known as the BOINC *server complex*, is centered around a relational database that holds all the metadata

[7]BOINC also supports so-called *legacy* applications where no code modification can be performed – for instance, the source code is not available, solely the binary is – but a much better support is achieved if the application is integrated into the BOINC framework.

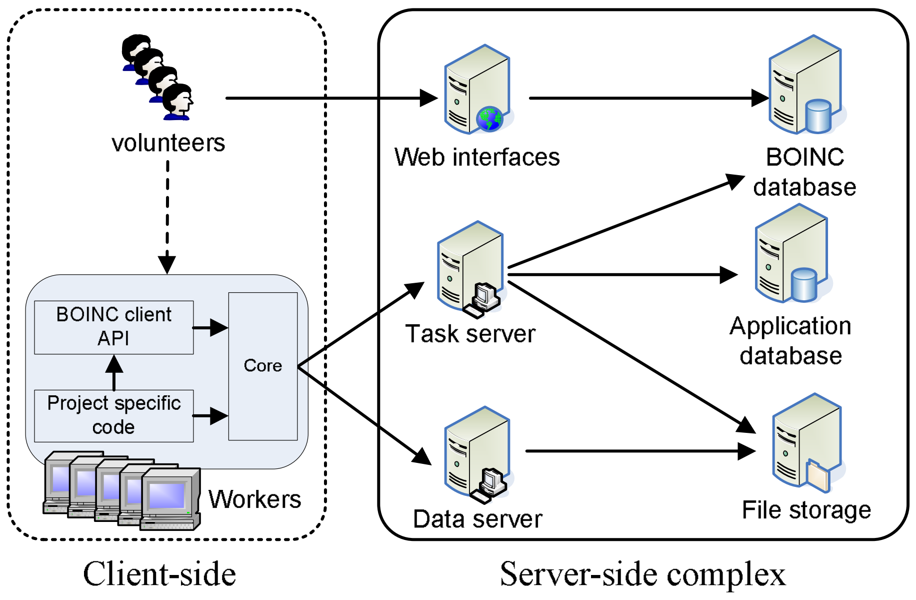


Figure 2.4: Generic representation of BOINC (adapted from [Anderson *et al.* 2005]).

associated with the project such as applications, platforms, versions, tasks, results, volunteer accounts, teams and so on [Anderson 2004].

A BOINC server-side also includes a *web interface*, a *task server* and a *data server* [Anderson *et al.* 2005]. Specifically, web interfaces exist for account and team management, message boards, and other features. The task server creates tasks, dispatches them to workers, and processes the returned results. It includes several components like the *work generator*, the *scheduler*, the *feeder*, the *transitioner*, the *validator*, the *assimilator*, the *file deleter* and the *database purger*. Finally, the data server makes available input files and executables to be downloaded by the workers, and allows the uploads of output files. The interaction of these server-side BOINC components with the client-side is depicted in Figure 2.4.

Client. In a BOINC project, volunteers participate by running the client software part on their computers. The client software of BOINC consists of several components [Anderson *et al.* 2006], namely, *applications*, *core client*, *manager* and *screensaver*, as illustrated in Figure 2.5. The applications are the reason for using BOINC, since they represent the real work that the projects managers want to have performed. As stated above, a resource owner might have her machine(s) attached to more than one project at once. BOINC handles the distribution of local resources through the applications, namely CPU, according to the user preferences (for example, it can be configured to split 20% CPU for application A, 30% for application B and 50% for application C).

The BOINC core client, which is also known as the *BOINC daemon*, handles the operations with the server-side, communicating with the schedulers and managing both download and upload operations. For instance, it is responsible for sharing local resources among applications if the host is attached to several BOINC projects. The core client interacts with the worker applications through the runtime system and by way of a library that needs to be linked with the application. Specifically, the communications between the core client and the applications are done via message passing of XML messages exchanged through shared memory. Note that having the core client dealing with all communications and management issues allows application programmers to focus mostly on their applications, without having to deal with the burden of interacting with the server-side. As stated by their authors, BOINC is designed to be used by scientists or other resource-hungry users that are not system programmers or IT professionals [Anderson 2004].

The BOINC manager provides a graphical user interface to view and control computation status. It communicates with the core client using remote procedure calls over TCP. In fact, BOINC manager handles management requests from any software that communicates via the BOINC GUI RPC protocol.

Finally, the screensaver module, if enabled by the volunteer, runs when the computer is idle. It does not generate screensaver graphics by itself, but rather communicates with the core client, requesting that one of the running applications display an appropriate screensaver. Note that although including support for a screensaver in the client might be seen as displaced, since it is not fundamental for the execution of the applications, it is a mechanism that can attract and retain volunteers. In fact, much of the initial success of SETI@home was due to its appealing screensaver[8].

[8]Some authors even use the term *Screen Saver Science* to describe *Public Resource Computing* [Andersen *et al.* 2006].

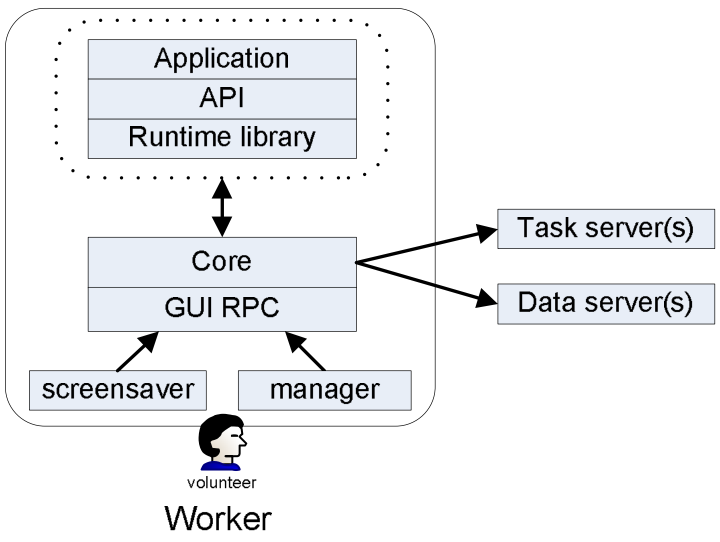


Figure 2.5: BOINC's client-side components (adapted from [Anderson *et al.* 2006]).

Analysis

Non-interference: The level of obtrusiveness of BOINC relatively to the volunteered machine can be configured by the resource donor, indicating, for instance, the time periods when the foreign applications should run, the maximum amount of storage space that can be used, and when the application can connect to the Internet. BOINC executes its applications under the lowest priority level provided by the operating system. In addition, a volunteered host can be configured under a more strict sharing policy, with foreign applications only allowed to run when no local user activity is taking place.

Usability: Since it targets *public resource computing*, BOINC strongly focuses on usability in order to recruit and retain thousands of volunteers. For instance, the graphical manager aims to be a simple interface for non-computers savvy donors to understand and configure their resources. Additionally, to ease the aggregation and maintenance of multiple machines, BOINC allows a donor to aggregate her machines under three base profiles: *work*, *school* and *home*. A change in a profile is propagated to all machines of this donor that are attached to the profile.

An interesting feature of the BOINC framework lies in its automated management of application's update. Whenever a project releases a new version of its application, the BOINC client of the attached machines downloads the new version, verifies its correct signature and then replaces the old one. However, this does not happen with the BOINC core client to preclude malicious updates.

Usability is not restricted to the client part. Indeed, although some computer knowledge are definitively required for setting up a server-side complex, much of the burdening work is facilitated by BOINC comprehensive set of tools and templates. For instance, the project web site, with support for user forums, volunteer statistics and even the socially rewarding feature *user of the day* is relatively straightforward to configure.

Scalability: Although based on a centralized architecture, BOINC currently supports large volunteer computing projects. For instance, the SETI@-home project has more than 170,000 active resource donors totaling more than 300,000 machines[9]. In fact, in its early version, SETI@home (non-BOINC based) endured a much more higher popularity than anticipated, forcing their developers to focus on the server-side scalability. BOINC has inherited much of these scalability-oriented features and has continued to be developed along this path. In part, the scalability is achieved by decoupling web functions from task functions and data related ones. Also, for the more solicited components, is it possible to distribute them to several machines to spread the load. Some protection also exists at the client side, like exponential backoff, upon which a client whose request could not be served at the server-side will exponentially augment its waiting time interval and add a random value, before performing a new request attempt. This serves to avoid workers saturating the server-side after, for example, a prolonged downtime occurred at the server-side.

Security: Security in BOINC is a very important issue. At the server complex level, security issues arise with the web server and database components (MySql, PHP, etc.), although eventual vulnerabilities in these components are not the responsibility of BOINC. However, by gaining unauthorized access to the volunteer database, a malicious user can, for instance, change the configuration settings of the volunteer machines. Note that replacing an application's binary by a malicious file, for instance a Trojan, re-

[9]http://www.boincstats.com/, June 2007.

quires the file to be signed with the project's private key, since the BOINC core client always checks the signature of a candidate application signature before replacing a binary. Thus, to protect against effective binary replacement at the server's, the project managers must securely keep the private key. Anyway, any security hazard that might occur in a high visible BOINC project would seriously undermine the credibility for resource volunteering and would probably mean the loss, at least temporarily, of many volunteers. However, it should be noted that security hazards involving popular components like web server and database are usually quickly fixed, and thus conscientious security procedures at the server side diminish the risks.

At the client side, few security provisions are taken. In fact, since BOINC does not yet implement sandboxing, a buggy, or even worse, a malicious application can cause serious damages. On Windows machines, this is further aggravated since BOINC tends to be executed as a Win32 service, which by default is installed under the Local System Account, the highest privilege level.

Another security menace at the client side relates with the possibility of a volunteer to replace the binary of an application to which the machine is attached to. Anecdotal episodes have shown that replacing the binary application is a rather straightforward operation for local users with administrator privileges[10]. One example was the Akos Fekete's episode. Akos Fekete was an Einstein@home contributor from Hungary, and used his knowledge of assembly programming to produce an optimized version based on the official binary application. His version more than doubled the performance of the original one, all of this done at the binary level, since he did not have access to the application source code [Knight 2006][11]. The patched version was rapidly spread through the project's forum, with enthusiastic volunteers replacing the official binary with the patched one, in order to have their resources processing more work per time unit, and thus earn more credits. All of this was done without the approval of the project coordinators. Although the outcome of this episode actually benefited the project – some of the improvements were incorpo-

[10]This also casts doubts about the soundness of the binary signature scheme, since the binary of the hosted application can locally be replaced by another one.

[11]Although BOINC is open source, public projects rarely release their applications source code, in part to avoid the surge of unwanted versions or for fear that a security issue can be exploited.

rated in the official version of the application – it illustrates the fragilities of the BOINC client security model. Indeed, a malicious user could follow a similar approach to spread a patched binary that would disrupt the computations, and in fact further patched versions from the well intentioned Akos Fekete were banned by the project coordinators fearing that an optimization could disrupt the integrity and soundness of the results. In summary, although BOINC provides some security features, volunteering resources for a BOINC project requires a certain amount of trust.

Fault tolerance: Since it targets highly volatile environments, BOINC provides some support for fault tolerance. Indeed, one of the goals of BOINC is autonomy in the sense that the software should recover without human intervention from problems, even those caused by unexpected volunteer actions [Anderson *et al.* 2006]. For instance, the framework supports redundancy at the level of database servers, thus potentially increasing availability with the added bonus of scalability. Likewise, the detection of *dead worker machines* is performed through a simple soft state mechanism, upon which a task is assigned a deadline when scheduled to a worker. Whenever the worker fails to return the results before the deadline expires, the supervisor reassigns the task to another worker. In addition, results that are received past the respective deadline are discarded by the supervisor.

At the worker level, the BOINC client supports some application-level checkpointing, meaning that applications themselves should incorporate checkpoint capabilities to tolerate local node failures. The application should notify the core client when it starts and terminates a checkpoint operation. This is needed in order to allow the BOINC client to update some state data, namely the CPU time used so far, which needs to be saved to persistent storage, so that it can be later restored. Additionally, knowledge of a recent checkpoint operation helps the core client scheduler to decide if it should or not replace a running task by another one belonging to another project, in the case the machine is participating in more than one project.

By default, the validation of results is achieved through replication with majority voting, where the number of replicated instances is set by the project coordinators. For projects extremely sensible to numeric variation, BOINC also supports homogeneous replication, upon which replica instances of a task are solely assigned to equivalent machines, that is, machines that have the same CPU vendor and the same operating system offering guarantees that their outcomes are identical if properly computed. Homogeneous replication allows result validation through strict equality comparison (i.e., bit-to-bit identical results) [Taufer *et al.* 2005a]. Although validation through replication has the advantage of being a generic mechanism, which can be used without adaption by any project, it is vulnerable to collusion. In fact, as a follow-up of the Akos Fekete's episode, right after the beginning of stage S5, the Einstein@home project coordinators were forced to ignore results from further Fekete's optimized versions since they feared that errors could exist and that the outcome of the project would not be scientifically accepted. Moreover, due to the relatively high level of adoption of the patched version, the probability of the instances of a same task being executed by a patched version was significant. Since the project was using a minimum quorum of 2 for validation[12], this meant that possible errors would not have been caught by the validation mechanism if two paired workers were using the same version.

Accountability of resource usage: The credit assignment mechanism of BOINC also relies on replication. Indeed, whenever an application sends back the result of the task it has just completed, it appends its claim for credit, asking for the number of credits corresponding to the produced computing effort[13]. The project coordinators are free to implement the policy they consider best suited for the project, taking in consideration the credit claims of the instances of a same task. For example, it is possible to define the computing credit of a task as the arithmetic average of the credit claims of all instances, or more drastically (from the point of view of resource donors), as the minimum credits claimed among all instances.

Fostering motivation: An interesting feature of BOINC lies in the supporting features oriented for motivation of volunteers. Indeed, besides the above cited support for graphical screensaver, BOINC's default installation includes message forums where volunteers and project coordinators can exchange tips and point of views, possibly fostering social ties. The project web site also allows a volunteer to consult the status of the computed workunits, and of her volunteered machines. Moreover, a BOINC project also includes web rankings, where volunteers are ranked accord-

[12]In the stage S5 the Einstein@home project reduced the minimum validation quorum from 3 to 2, in order to have more computing power available.

[13]The credit claim is computed by the application, but this value can be jeopardized by cheaters trying to boost their credits.

ingly to their credits. Volunteers can also be organized in teams. All of this fosters competition and rivalry among volunteers and teams, increasing the motivation for volunteers to bring additional resources to the public desktop grids [Holohan & Garg 2005]. Additionally, the server complex can produce, if configured to do so, a plethora of statistics related to the computations and volunteers, and export it in XML format. Several statistics aggregator web sites have emerged[14], which process the XML files and display the statistics for all major projects. Also, third party enthusiasts, have developed BOINC related tools, with functionalities that range from monitoring a local BOINC installation to control a full network of BOINC installed machines.

An extension to BOINC: SZTAKI Desktop Grid

The SZTAKI Desktop Grid (SZDG) is an extension to BOINC developed by the Hungarian MTA SZTAKI [Kacsuk *et al.* 2007; Balaton *et al.* 2007]. The extension is aimed at enabling local desktop grids (LDGs), that is, institutional and enterprise level desktop grids. In particular, SZDG allows for an easy setup and implementation of a private computing project supported solely by local computing resources. This can be important for desktop computing projects that can only resort to local resources, for instance due to data privacy or other similar reasons.

Another major feature of SZDG is the possibility of exploiting hierarchically organized resources in an almost transparent manner. In fact, under SZDG computing resources can be stacked in a hierarchical way, extending the classic server-client model of BOINC. In this way, SZDG allows aggregated resources such as clusters or local sets of computers to be transparently used, with such resources appearing as powerful but as a single computing element from the perspective of the BOINC server [Kacsuk *et al.* 2007]. For this purpose, SZDG implements a hybrid BOINC module (named *child LDG server*), which sits between the main server and the end-level resources that actually execute the tasks. On the one hand, this hybrid module requests tasks from the main server and thus appears as a regular client, yet a powerful one, to the main server. On the other hand, the hybrid module acts as a server for the resources that sit lower in the hierarchy, providing them the tasks it has itself obtained from the main server

[14]http://www.boincstats.com

and forwarding their results to the main server. The hybrid module handles specific issues such as the automatic deployment of applications and tasks at the workers', guaranteeing the integrity of the binary code and of the input data.

To ease the programming of applications to be executed on a BOINC-SZDG environment, SZDG provides an API, named the *Distributed Computing Application Programming Interface* (DC-API). As stated by the authors, the API aims to be simple and easy to use. Nonetheless, the DC-API also provides advanced features to support more demanding scenarios [Balaton *et al.* 2007], allowing programmers to have a finer control of SZDG. Besides easing the adaptation of applications to SZDG, the DC-API also isolates the applications from the executing back end. In this way, an application can be moved from an executing environment to another one, requiring just the recompilation of the code.

The SZDG framework also allows for the execution of parallel applications, providing support for parallel environments such as the Message Passing Interface (MPI) in clusters. In this mode, the application is handled by another modified BOINC client-server module that sits at the front-end machine of the cluster [Balaton *et al.* 2007]. This module acts as a wrapper to the parallel environment of the cluster, interacting with the appropriate components in order to submit tasks and to collect the results, forwarding them back to the upper layers of the hierarchy.

2.6.3 XtremWeb

XtremWeb is a volunteer open source platform middleware developed at the University of Paris-Sud, France [Fedak *et al.* 2001; Fedak 2003][15]. It was designed to study execution models in global computing, but has evolved to a rather complete desktop grid environment. Contrary to the BOINC platform, XtremWeb has a relatively discrete public exposure and focuses mainly on important research issues such as the support for multiple applications, security and portability, besides high computing performance. Despite its commitment to research, XtremWeb is a valuable desktop grid platform. In fact, it has already been used in several large scale desktop grid projects, as reported in [Lodygensky *et al.* 2003a].

[15]http://www.xtremweb.net

Architecture

The XtremWeb infrastructure is comprised of three main types of components: the *coordinator*, the *client* and the *worker* [Fedak *et al.* 2001]. The coordinator is responsible for hosting applications and tasks submitted by clients and for scheduling the tasks to the workers which express their willingness to donate some CPU time. The coordinator orchestrates the executions over the volunteer resources. On the current implementation of the architecture, there is one coordinator, although the architecture is prepared to support multiple coordinators for the sake of fault tolerance and load balancing.

Workers run at the volunteer machines, and are the entities that actually execute the tasks. A worker monitors the load of the resource, and whenever conditions are met (i.e., local load is below a given threshold and no interactive activity is observed for a while), the worker executes a task that might have been interrupted previously, or if no task exists, it requests one from the server. It then proceeds to execute the task. On completion, it sends back the results to the server.

The *client* term under XtremWeb designates the module from which an user can submit an application and the tasks she wants to have executed by way of the scavenging platform. Note that this departs from the BOINC nomenclature, where the client designates the module that actually executes the task/application (identified as *worker* under XtremWeb).

In XtremWeb, the client module acts as an intermediary between the user's application and the scavenging system [Cappello *et al.* 2005]. It is implemented as a library plus a daemon process. The daemon process runs at every machine that allows the submission of applications. The library provides an interface between the application and the coordinator. The basic actions ensured by the client are identification, submission of tasks and results retrieval. The existence of a client module in XtremWeb is related to the platform support for multi-application/multi-users, since XtremWeb allows for the coexistence of multiple executing applications possibly submitted by different users, departing from the BOINC single-application model. This way, XtremWeb can be seen as a generic harvesting system, much like Condor, but inherently designed for wide-scale environments. In fact, XtremWeb and Condor can be combined together for enabling resource sharing among firewall- and NAT-isolated clusters, forming a lightweight grid of Condor pools [Lodygensky *et al.* 2003b]. In this scheme, the workers of XtremWeb execute as Condor tasks, fetching their jobs from the central XtremWeb's coordinator, which can be located in a different administrative domain from the machines that run the XtremWeb's workers.

Analysis

Non-interference: Obtrusiveness in XtremWeb is controlled by the monitor thread of the worker and follows the standard approach of scavenging system: a resource is considered idle in the continued absence of local interactivity combined with low local load activity. Similarly to BOINC, communications are also worker-initiated. Therefore, firewall policies and NAT systems are not an impeding issue.

Usability: Usability of XtremWeb is enhanced by its support for multiple applications, which allows several users to concurrently exploit the computing resources. This allows to setup XtremWeb for harvesting private, yet geographically dispersed resources. Regarding platforms, XtremWeb supports Windows, Linux and Mac OS, and further support is facilitated by its Java-based infrastructure[16]

Scalability: Since the coordinator module can be distributed, XtremWeb scalability should be good. In this context, experimental results are limited to 1000 machines [Lodygensky *et al.* 2003b].

Security: As stated before, security is a main focus on XtremWeb, especially at the level of protecting volunteered resources. For that purpose, XtremWeb executes applications under sandboxing, either through an own implemented sandbox environment, or resorting to the Java sandbox, in the case of Java applications.

Fault Tolerance: XtremWeb does not provide explicit support for checkpoint and restart, thus forcing applications to resort to application-level checkpointing if such feature is needed. Moreover, submitters are also responsible for verifying the computed results, since no provision for replication exists. However, the coordinator periodically stores, on persistent storage, the data and metadata concerning tasks and workers. On restart, the coordinator reads the stored data for setting its own state, and then retrieves tasks that have already been scheduled [Lodygensky *et al.* 2003b].

[16]There is also a version of XtremWeb written in C++.

This centralized logging of state data by the coordinator also allows the system to tolerate client's faults and mobility, since the results from a task can be retrieved through any client machine.

2.6.4 GridMP

GridMP departs from the previously reviewed desktop grid middleware by the fact of being a commercial product, distributed by *UnitedDevice, Inc*. It is a closed platform, and thus scarce information exists relatively to its inner architecture and working.

Note that although the Entropia desktop grid middleware has been extensively studied in the literature[17](e.g. [Chien *et al.* 2003]) and thus would have been a natural choice for being reviewed here, the Entropia, Inc. company seems to have disappeared, taking along the commercial middleware.

GridMP is available in five versions, all based on the concept of harvesting resources [uniteddevices 2007]. The versions are: *GridMP Enterprise*, *GridMP for clusters*, *GridMP on demand*, *GridMP Global* and *GridMP Alliance*.

GridMP Enterprise targets enterprise desktop grids, aggregating computer resources across servers, end-users machines and clusters. *GridMP for clusters* is more specific, since it aims to simplify the management of clusters. The *GridMP on demand* is a pay-per-use compute service for users that need access to high performance compute power without having to invest on permanent resources. *GridMP Global* is a wide public grid for large institutions performing research and analysis projects. It gathers volunteer resources (which have installed the GridMP global's client) supporting large projects such as *fightcancer@home*. Finally, *GridMP Alliance* is a solution that allows owners of vast underutilized resources to commercialize their computing power by selling it to *GridMP on demand* customers. In this review, we focus on *GridMP Enterprise*, since it incorporates all the main features of a desktop grid middleware. GridMP targets institutions that aim to harvest their own resources' spare cycles with guarantees of a high level of security for both the applications and the associated data that are to exploit underutilized resources. Indeed, GridMP has a strong emphasis on security, resorting to sandboxing, and to the encryption of data and communications. According to United Devices, Inc., some of the clients of GridMP include pharmaceutical, life sciences, geosciences, finan-

[17]Some of the Entropia founders have had strong links to the academic world.

cial services and industrial engineering companies, to name just a few, most of them dealing with sensible and valuable data. GridMP was also used as the desktop grid middleware for the *World Community Grid*, which is a volunteer-based resource powering large-scale, public interest research projects [gridorg 2007]. However, the *World Community Grid* switched in 2005 to the BOINC middleware.

The GridMP platform is comprised of three primary components: (1) *tools and interfaces*, (2) a *server* and (3) local running *agents* [uniteddevices 2007]. As the name implies, the tools and interfaces allow users, application developers and resource administrators to interact with the system. Specifically, GridMP provides *application services* that are wrappers with simple interface abstractions to allow applications to be run without any code modification on GridMP's managed resources. In addition, a *software development kit* is provided for deeper integration of applications. For instance, this allows to implement pipeline-like dependencies on applications, or to exploit parallelism. Both resources and applications can be monitored and controlled through the *web management and reporting console*. Note, that according to its marketing brochure, GridMP can run on Windows and Linux machines.

The server acts as the core element in GridMP. It is responsible for matching the users' demand for resources and the availability of resources. Its main functions include resource management (which machines are authorized to join the pool), user management (which users can access the system for submitting and monitoring applications), application and workload management (scheduling applications to resources, and dealing with faults and resource overloads) and data management[18]. Finally, the agent is the software that connects a resource to GridMP. Its primary functions are to detect idle cycles on the computer where it runs, to request jobs from the GridMP server, to execute them and to send back the results.

Other commercial desktop grid systems include Frontier from Parabon Computation [parabon 2007], GridServer from Data Synapse [datasynapses 2007] and Digipede [digipede 2007].

[18]GridMP allows data requests to be treated separately from application to allow data reutilization, that is, allowing that applications requiring the same set of data can be preferentially scheduled on the same resources to allow data reuse.

2.6.5 P2P-based Architectures

To the best of our knowledge, no generic P2P-based middleware exists for cycle harvesting, contrary to file sharing, where a significant number of functional and quite popular frameworks are in broad usage.

The Cluster Computing On the Fly (CCOF) is a scalable and modular peer-to-peer cycle sharing architecture for open access and wide-scale environments. It aims at executing generic bag-of-tasks (BoT) applications over harvested resources. One of the interesting features of CCOF lies in its so-called *wave* scheduling policy, upon which resources are organized accordingly to their geographic timezones [Zhou & Lo 2005]. The goal is to schedule tasks in a way that executions occur mostly on the nighttime period of resources. For that purpose, tasks running on resources that are close to enter daytime (and probably about to be claimed back by owners) can be migrated by the scheduler to resources which are in nighttime. This way, tasks follow nighttime's located resources like a wave around the globe, hence the name given to the scheduler.

Regarding the submission of applications, CCOF fosters a symmetrical harvesting model, upon which peers donate resources but are also able to harvest other peers' resources, by submitting themselves applications. This departs from the single submitter model implemented by BOINC and other centralized middleware. However, note that this submission model opens up a whole lot of security and trust problems. Additionally, an economical model is needed to control submission of applications, specially to discourage the so-called *free riders* (i.e., application submitters that exploit the system giving much less on return) and other non-social behaviors. Economical incentives are also needed for resource owners, since without the existence of a cause like a challenging scientific project like SETI's search or Einstein's gravitational wave analysis, resource owners will have no incentives to aggregate their resources to the CCOF-based desktop grid. CCOF only seems to exist as an architecture proposal, with its main concepts studied through simulations. Indeed, no implementation nor prototype appears to exist.

The Personal Power Plant (P3) [Shudo *et al.* 2005] promotes CPU cycle harvesting. This research project resorts to the JXTA [Verbeke *et al.* 2002] P2P middleware for implementing high level services, such as master-worker and Message Passing Interface (MPI) parallel programming libraries. JXTA allows transparent firewall and NAT traversal at the cost of communication performance. Although, P3 already addresses some issues like sabotage-tolerance through replication, it currently only supports Java application. In fact, the project is still in an early stage of development, and a production version seems distant in time.

Han and Park propose a lightweight *Personal Grid* (PG) formed out of unstructured peer nodes [Han & Park 2003]. Contrary to the current wide-scale desktop grid projects, where only the coordinating entity can submit tasks, PG's main goal is to allow that any individual can have her own multi-task application(s) executed over the volunteered resources. PG has no central control, implementing a peer-to-peer model. Specifically, the proposed system explores a network of super nodes, calling *cluster* to a set of worker nodes that is connected to a same super node. The aggregation of a worker node to a given *cluster* (a worker node is only connected to a single super node) is determined by the network proximity: on the prototype implementation, two nodes are considered close if they can reach each other through a link level broadcast, and thus belong to a same local network. Since any node can submit an application to be executed over a network of workers, PG has a mechanism to match applications' demands to existing resources. Indeed, to submit a task, the submitter node releases an *advertisement* to the network. This advertisement, which holds a meta description of the task (URL of the needed files, message digest codes, etc.) is sent to the super node which then forwards the metadata through the network of super node and so on. To avoid flooding, the advertisement is limited by a *time to live* (TTL). When terminated, the results are sent back to the submit node. Contrary to the server-based model where only a central and credible entity can release tasks, PG is prone to malicious submitters and thus security of both resources and tasks' results are important open issues.

CompuP2P is a peer-to-peer based architecture which aims to allow peers to share computing resources such as CPU, memory, storage, etc. under an economical market model [Gupta & Somani 2004]. CompuP2P uses ideas from game theory and microeconomics to foster incentive-based schemes for peers to share their otherwise idle resources. CompuP2P resorts to the Chord DHT overlay for addressing and connecting nodes. To cope with node failures and departures, the system relies on *dynamic checkpoint-*

ing, upon which the unused memory of peers nodes is used for storing application-level checkpoints. In the true spirit of the CompuP2P's computing market, resources needed by dynamic checkpointing are also negotiated. Note, that no information is given by the authors regarding how the system tolerates failures of a machine holding dynamic checkpoints.

Other peer-to-peer based projects for exploiting idle cycles resort to structured overlay networks. This includes Flock of Condor [Butt *et al.* 2003], WaveGrid [Zhou & Lo 2006] and the *self-organizing Wide area Overlay of networks of virtual Workstations* (WOW) [Ganguly *et al.* 2006]. Flock of Condor aims to dynamically organize the cooperation (*flocking* in the Condor's jargon) of pools of Condor, while WaveGrid aims to exploit time zones, focusing on scavenging resources during nighttime, to benefit from resource idleness. WOW departs from the former two by resorting to virtualization to overcome network asymmetries and build an homogeneous platform.

2.7 Summary

In this chapter, we presented the main motivations for resorting to desktop grids, analyzing not only their main benefits, but also their major limitations. Additionally, two types of desktop grids – institutional and public – were discussed, and the main middleware for desktop grids were reviewed with emphasis on the main characteristics such as fault tolerance, security and scalability.

A common characteristic found in all reviewed desktop grids is their centralized organization, with a server (or a set of servers) coordinating the execution of tasks over volunteer resources. This centralization is mainly justified by the communication model, which is worker-initiated, and restricted to server-worker with no direct communications occurring among workers. This a consequence of the network asymmetry, with NAT and firewall schemes making difficult the direct communication between peers.

Regarding fault tolerance, it emerges that current desktop grid middleware resorts both on checkpoint-and-restart mechanisms and on replication for detecting and tolerating faults. As we shall see in the next chapters, both techniques can be further exploited to improve the makespan of applications executed over desktop grids.

3 Resource Usage in Desktop Grids

It is common sense that the effective usage level of computing resources is rather low, especially in environments where computers are mostly used interactively for office and communication activities like web browsing and email. Some studies focusing on Unix have shown that the vast majority of workstations, desktop computers and even servers remain idle most of the time [Heap 2003]. In this chapter, we aim to quantify the usage of main computer resources (CPU, memory, disk space and network bandwidth) based on traces collected from real desktop machines running the Windows 2000 operating system. First, we introduce the data collection methodology, presenting the Windows Distributed Data Collection framework (WindowsDDC) that was used to monitor the machines. Then, we characterize the average resource usage of classroom laboratories, based on a 77-day trace collected over 169 machines of 11 classrooms of an academic institution.

3.1 Introduction

Today, academic institutions frequently have large dozens of PCs in their classrooms and laboratories, with a large percentage of these PCs mostly devoted to teaching activities. A similar situation happens at corporations where computers are idle most of the time. Thus, most of the available computing power simply goes unused. In fact, considering that these PCs are only used during extended office work (from 8.00 am to 8.00 pm on weekdays), it means that more than half of the time these machines are left idle and could be used for running CPU demanding applications.

PCs of classrooms have an attractive characteristic for resource harvesting: no individual user owns the PCs, while office computers are generally affected to an individual user who "owns" or acts as the owner of the machine. Some of these individual owners do not tolerate schemes that exploit idle resources. Therefore, the use of individual PCs for distributed and parallel computing has to deal with social issues, besides engineering and technological ones [Anderson *et al.* 1995]. Computers in the classroom are centrally managed, have no individual owner, and thus the social issues regarding their use in resource harvesting are less cumbersome. Nonetheless, even without individual owner, care must be taken to avoid undermining regular user comfort, since resources are still primarily devoted to interactive users. Gupta et al. [Gupta *et al.* 2004] analyze the relation between resource borrowing and interactive usage comfort, concluding that resource stealing schemes can be quite aggressive without disturbing user comfort, particularly in the case of memory and disk.

Our main motivation for studying resource usage was to characterize the availability and the pattern usage of academic classroom computers, quantifying the portions of important resources such as CPU, RAM, disk space and network bandwidth which are left unused. In particular, we were interested in differentiating resource usage according to the existence or not of interactive login sessions, and their impact on resource usage.

3.2 The WindowsDDC Framework

The need to automate the periodic data collection over the machines to survey fostered us to develop a framework to support remote data collection in local area networked Windows machines. The framework was named Windows Distributed Data Collector (WindowsDDC). It aims to cover the needs that arise in distributed data collection at the scale of local area networks of Windows PCs [Domingues *et al.* 2005a].

Besides the purpose to collect machines usage data, we felt the need for a framework that would allow running a regular and unattended console application (termed as *probe*) across a set of machines. Such framework could be used, for example, for regularly assessing the system performance in the following manner: a benchmark can periodically be run over a set of machines, with the meaningful data extracted from the collected output of the benchmark. This would allow for an early detection of performance drops. Alternatively, we could use this framework to spot probable hard disk failures, resorting to the Self-Monitoring Analysis and Reporting Technology (SMART) [Allen 2004] values collected from the machines' hard disks, with suspicious disks being flagged as having a high probability of failing in a near future. These situations are just examples of what a remote execution framework allows to achieve in a set of networked machines. Therefore, we required a distributed collection platform that would follow these requirements:

- The framework should allow for easy collection, parsing and storage of selected data over sets of networked machines;
- The framework should be modular and easily extensible, allowing the easy integration of probes and associated post-collection filters in order to fulfill further needs and opportunities for data collection;
- No restriction should be placed on probe, besides the requirement of being a console application that can be run unattended;
- No software should be installed at remote machines in order to avoid administrative and technical burdens. However, it is acceptable to use a single dedicated machine (coordinator) for running the data collection system as long as no special hardware is required;
- The system should be as autonomous and adaptive as possible in order to minimize administrator interventions. For instance, the system should cope with the transient nature of machines availability. Ideally, the system should alert administrators only when abnormal events that might require human intervention occur;
- Due to budget restriction, the framework should be based on open source or freeware software;
- The data collector system should support Windows NT, 2000 and XP. Additionally, the framework should allow a smooth transition to Unix if needed;
- The system should be able to execute probes at remote nodes in one of two modes: single-shot or periodic. The former serves for probes that

should be executed only once (e.g., a benchmark) while the latter, as the name implies, periodically executes the probe (e.g., resource monitoring). Additionally, single-shot mode needs to deal with volatile machine availability, only terminating when execution of probe(s) has occurred in all the specified machines;

- To keep low intrusiveness, the framework should allow remote execution of processes at a low level of priority.

The remote probing solution was chosen in part because it avoids the installation of software in remote nodes, thus eliminating administrative and maintenance burdens that remote daemons and alike normally require. Another motivation for the remote probe approach is the possibility of tailoring the probe to our monitoring needs, capturing only the wanted metrics. The built-in remote monitoring capabilities like *perfmon* and Windows Management Interface (WMI) [Tunstall *et al.* 2002] were discarded for several reasons. First, both mechanisms have high timeout values (order of seconds) when the remote machine to be monitored is not available, a frequent situation in the considered environments. Furthermore, both *perfmon* and WMI impose a high overhead not solely on the remote machine but also on the network. *Perfmon*'s overhead is caused mostly by the need to transfer the remotely read data to the local machine, while WMI's overhead is a consequence of its dependence over Distributed COM (DCOM) for accessing a remote machine [Tunstall *et al.* 2002]. For these reasons, and since to the best of our knowledge no system complied with our requirements, we decided to develop WindowsDDC.

3.2.1 Experiments and iterations

WindowsDDC is based on a centralized architecture, with only one dedicated machine (*coordinator*) running the coordinator module of WindowsDDC. All the executions are performed in the context of what we term an *experiment*. Specifically, an experiment is composed by one or more binary probes and their associated post-collecting codes, plus the set of target machines where the execution of the probes should occur. A probe is simply a win32 console application that runs unattended and emits its results via the standard output (*stdout*) and the standard error (*stderr*) channels.

WindowsDDC organizes an experiment in successive iterations. An iteration consists of the execution attempt of every defined probes over the whole set of target machines. WindowsDDC uniquely identifies an iteration with the GMT time in Unix *epoch* format measured when the iteration starts (to cope with time asynchronism among the monitored machines, all time references are set by the coordinator machine).

An iteration is executed as follows: for every machine belonging to the execution pool, the user-defined probes and their respective post-collect codes are sequentially run. When the execution of all probes have been attempted at a given machine[1], WindowsDDC shifts to the next machine of the pool. Figure 3.1 shows the sequential steps of the execution of a WindowsDDC's probe over a remote machine. First, (a) the remote machine connectivity is checked through an ICMP ping. On success, (b) the probe is executed in the remote machine. Next (c), output results are returned to the coordinator machine. Finally, (d) these results are then post-processed at the coordinator's and stored.

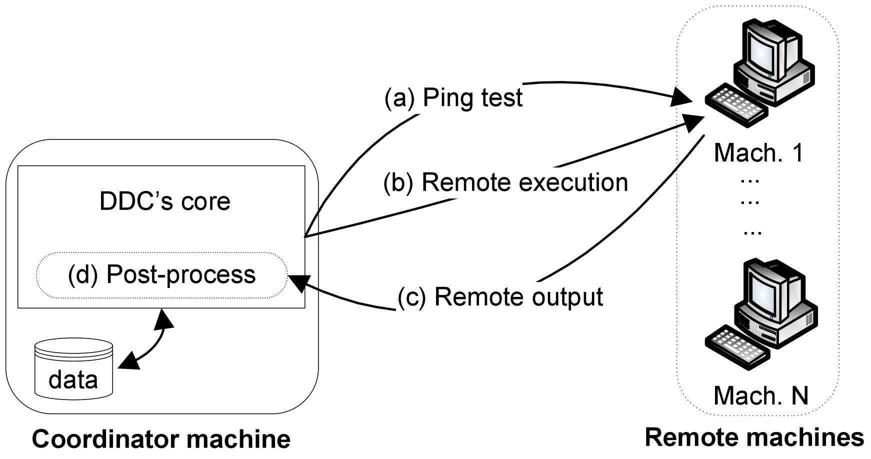


Figure 3.1: Overview of WindowsDDC architecture.

At the coordinator machine, the results of the execution of a probe are cumulatively redirected to text files (one file holds *stdout*, another one stores *stderr*) whose names are based on the probe's name. These files are stored under a directory hierarchy (one directory per remote machine) with the root directory named after the experiment name. This default logging

[1]An execution might not be successful, failing for several reasons – for instance the remote machine is down or without network connectivity.

behavior of the outputs can be replaced by user-defined code that gets executed on the coordinator machine right after remote execution occurs. This post-collecting code, which is specific to a probe, receives as input the results of the probe's execution and can implement actions that the user defines as deemed appropriate for the given context. For example, after parsing and processing the execution's output, post-collecting code can decide to report a particular event via email. Besides the output of the execution, the post-collecting code also inherits the execution context that contains information relatively to the machine where the probe was executed, the execution status (exit code from the probe), as well as the wall clock time spent on the remote execution, among other data.

One of the configurable parameter for an experiment is the interval time that should separate the start of two consecutive iterations and that effectively defines the data collection frequency. So, after an iteration has completed, WindowsDDC pauses the execution until time has come to start the next iteration. If this specified time interval cannot be respected, WindowsDDC waits for a minimum time gap before starting the next iteration. This prevents execution loops that spin in an uncontrolled manner.

WindowsDDC maintains a trace file for every probe of an experiment. At the end of an iteration, a text line summarizing the outcome of the iteration is appended to the file. This row starts with the time stamp identifier of the iteration and includes, among other items, a comma separated list with the machine names where the probe was successfully executed, a similar list for failed executions (executions aborted due to timeouts) and a third one that holds the names of the machines that were unavailable when the execution was attempted (the ping attempt failed). This row also contains the wall clock time that was needed to complete the iteration. A trimmed example of a trace row is shown in Listing 3.1. From the data, it can be extracted that the iteration began at time stamp 1100441460 (in Unix's epoch format, meaning 14-11-2004 14:11:00 GMT) with 120 successful executions (machine *m*01 and others), no failed executions and 49 unavailable machines (machine *m*12 and others) yielding 71.0% successful executions. The iteration took 381.9 seconds. The listing also displays the next two entries, 1100442360 and 1100443260.

Listing 3.1: Extract of an Iteration File

```
...
1100441460|120|0|49|71.0|381.9|m01,...| |m12,...|
1100442360|124|0|45|73.4|385.2|m01,...| |m14,...|
1100443260|124|0|45|73.4|386.0|m01,...| |m14,...|
...
```

The trace file of a probe can be used to drive an off line temporal analysis of the probe execution. This file can also be used as a log, and in fact, the periodic mail reports that are sent by a WindowsDDC experiment includes the trace file (due to its high redundancy – the file contains mostly machine names – the file is highly compressible and thus can be sent via email) as a source of information regarding the evolution of the experiment.

3.2.2 Remote Execution

The remote execution mechanism of WindowsDDC is based upon the set of freeware tools from SysInternals [Russinovich & Cogswell 2006], namely the versatile *psexec*. *Psexec* is a utility that allows the remote execution of an application given the proper access privileges. *Psexec* is a flexible tool, configurable through appropriate command line switches.

In WindowsDDC, *psexec* is used as follows: an appropriate command line that includes all the needed *psexec* switches and parameters, as well as the probe executable name with its own command line arguments is formatted and executed in the context of a separate thread within the WindowsDDC core. The need of a separate thread for the remote execution arises from the possibility of deadlock at the remote machine that would stall WindowsDDC execution. Therefore, after a given time interval (configurable for each probe), if the execution of *psexec* has not yet terminated, a timeout is triggered with the execution being aborted by way of canceling the execution thread. Under these circumstances, the remote machine is flagged as having failed the probe execution, with a new execution attempt scheduled for the next iteration.

Since a target machine can be unavailable at a given time (powered off, unplugged from the network, etc.), before attempting the execution of a probe, the connectivity of the target machine is assessed with ICMP pings. If no answer to the pings is received, the remote machine is assumed to be unavailable and thus no remote execution is attempted in the current iteration. The advantage of using ICMP pings over immediately attempting the remote execution is that ping's timeout can be controlled and thus set to a lower value (for instance, hundredths of milliseconds) than the time length it would take *psexec* to detect remote machine unavailability, which is in the range of seconds. This way, detection of an unavailable machine is much faster. Additionally, WindowsDDC's ping mechanism also allows to perform ping studies, where the machines are only tested for network connectivity[2].

3.2.3 Post-collecting code

An important element for WindowsDDC flexibility lies in its ability to execute user-defined code right after the execution of a probe, allowing the processing of the output channels (*stdout* and *stderr*). For that purpose, post-collecting code needs to be written in the form of a Python[3] class that extends the `DDC_cmd` class, implementing the method `ParseResult()`. This method is executed by the main core of WindowsDDC at the coordinator's machine right after the remote execution has terminated. It receives as parameters, the probe's *stdout* and *stderr* contents, an object representing the remote machine that actually executed the probe, the execution exit code of the probe, the iteration identifier, as well as other context data (e.g., the directory path at the coordinator's where the output files of the current execution are stored). The base class method for `ParseResult()` is shown in Listing 3.2.

WindowsDDC was a central piece in the monitoring infrastructure, allowing us to collect the whole trace usage presented in this chapter, without having to install any software at remote machines. In fact, for most of the machines, we never physically accessed them, even ignoring the exact physical location of some of them. WindowsDDC software is available under the GNU Public License (GPL)[4].

[2]A drawback occurs if remote machines are blocking ICMP pings.
[3]http://www.python.org/
[4]http://www.estg.ipleiria.pt/~patricio/DDC/

Listing 3.2: ParseResult() Method (generic version)

```
# Method to parse stdout/stderr (lists)
# @param machObj  [IN] machine obj.
# @param status   [IN] execution status
# @param outList  [IN] list with stdout (can be None)
# @param errList  [IN] list with stderr (can be None)
# @param execTime [IN] execution time
# @param macroDict[IN] macro dictionary (used to extract the
#              directory of output file)
# @param timestamp[IN] timestamp for "timestamping" events
# @return None
def ParseResult(self,machObj,status,outList,errList,
                              execTime,macroDict,timestamp):
    "Parse result of execution"
    # Convenience alias to m_CmdName
    CmdName = self.GetCmdName()
    if status != 0:
        print "%s - %s - exec. error -- STATUS=%s" % \
            (CmdName,machObj.m_MachName,status)
    else:
        print "%s executed at '%s' in %0.3f secs" % \
            (CmdName,machObj.m_MachName,execTime)
    if outList != None:
        # Debug
        if self.m_DbgLevel > 4:
            print "stdout: %d lines, stderr:%d lines" %\
                  (len(outList),len(errList))
        # Append STDOUT/STDERR lists to save files
        self.ListToSaveFile(outList,'OUT',machObj,macroDict)
        self.ListToSaveFile(errList,'ERR',machObj,macroDict)
```

3.3 Methodology and Monitored Metrics

In this section, we present the monitoring methodology and the gathered metrics.

3.3.1 Methodology

Our monitoring methodology resorted on periodically probing the remote machines through WindowsDDC. Specifically, every 15 minutes an attempt was made to perform a remote execution of a software probe (*W32probe*) sequentially over the whole set of machines. Next, we detail the collected metrics.

3.3.2 Monitored Metrics

For the purpose of this study we developed the *W32probe* probe. *W32probe* is a simple win32 console application that outputs, via standard output (*stdout*), several metrics aimed at characterizing the state of the machine that is being monitored. These metrics are grouped in two main categories: *static* and *dynamic*. Static metrics describe characteristics that typically remain constant over time. Examples of such metrics include CPU name and type, and the amount of installed main memory. Dynamic metrics are related to current computing activity, measuring the usage of main resources. Dynamic metrics include CPU idleness percentage, memory load, available free disk space and whether an interactive session exists at the machine. Next, a brief description of the two categories of metrics is given.

Static Metrics

Static metrics comprise the following elements:

- **Processor name, type and frequency**: this identifies the processor name and its frequency.
- **Operating system**: name, version and service pack version, if any.
- **Amount of main memory**: size of installed main memory.
- **Amount of virtual memory**: size of configured virtual memory.
- **Hard disks**: for every installed hard disk, *W32probe* returns a descriptive string, the serial identification number and the size of the drive.
- **Network interfaces**: display the MAC address and the associated description string for every installed network interface.

Dynamic Metrics

Dynamic metrics collected from *W32probe* include the following items:

- **Boot time and uptime**: indicates the system boot time and the respective uptime, both expressed relatively to the moment when the probe was run.
- **CPU idle time**: CPU time consumed by the idle thread of the operating system since the computer was booted. This metric can be used to compute the average CPU idleness between two consecutive samples.

- **CPU idle percentage**: CPU idle percentage since the machine was booted. This metric simply corresponds to the division of *CPU idle time* by the machine's uptime.

- **Main memory load**: main memory load (as returned by the field `dwMemoryLoad` filled by win32's `GetMemoryStatus()` API function).

- **Swap memory load**: analogue to the main memory load metric but for the swap area.

- **Free disk space**: returns free disk space.

- **Hard disk power cycle count**: SMART parameter that counts the number of power cycles of the disk, that is, the number of times the disk has been powered on/powered off since it was built [Allen 2004].

- **Hard disk power on hour counts**: SMART parameter that counts the number of hours that a hard disk has been powered on since it was built.

- **Network usage**: this metric comprises two main values and two derived ones. Main values are *total received bytes* and *total sent bytes*. Derived values are *received byte rate* and *sent byte rate* that are simply computed, respectively, from *total received bytes* and *total sent bytes*.

- **Interactive user login session**: if any user is interactively logged at the monitored machine, the *username* and domain name (if any), along with the session initialization time are returned.

3.4 Experiment

3.4.1 Computing Environment

Using WindowsDDC and *W32probe*, we conducted a 77-day monitoring experiment using 169 computers of 11 classrooms (L01 to L09) of an academic institution for a total of 11 complete weeks. The monitored classrooms were used for regular classes. Additionally, when no classes were being taught, students used the machines to perform practical assignments and homework, as well as for personal use (email, web browsing, etc.). To avoid any changes of behavior that could bias results, only system administrators were aware of the monitoring experiment.

Each classroom has 16 machines, except L08 which only has 9 machines. All machines run Windows 2000 professional edition (service pack 3) and are connected via a 100 Mbps Fast Ethernet link. The main characteristics of the computers, grouped by classrooms (from L01 to L11), are summarized in Table 3.1. The columns INT and FP refer, respectively, to the integer and floating-point performance of the NBench benchmark's [Mayer 2007]. NBench, which is derived from the well-known ByteMark [bytemark 2007] benchmark, was ported from Linux, with the C source code compiled under Visual Studio 2003 in release mode. Like its ancestor, NBench relies on well-known algorithms to summarize computer performance with two numerical indexes: INT for integer performance and FP to expose floating point performance. It is important to note that the presented values are not suitable for absolute comparisons with NBench original values, since the operating systems and the compilers are different. However, the indexes can be used to assess relative performance among the monitored machines, since the same benchmark binary was used to compute the values. The final column of Table 3.1 expresses FLOPS performance as given by the Linpack benchmark [Longbottom 2007] compiled with Visual Studio .Net in release mode. All performance indexes were gathered with the Windows-DDC framework using the corresponding benchmark probe (NBench for INT and FP, Linpack for MFlops). Figure 3.2 plots the cumulative distribution of the machines according to their INT (left) and FP (right) performance indexes. The plots present similar shape, indicating that a single index (either INT or FP, or a combination of both) is enough for comparing performances of a set of machines. Additionally, it can be seen from both plots that, from a performance perspective, the machines can be roughly ranked in four major sets.

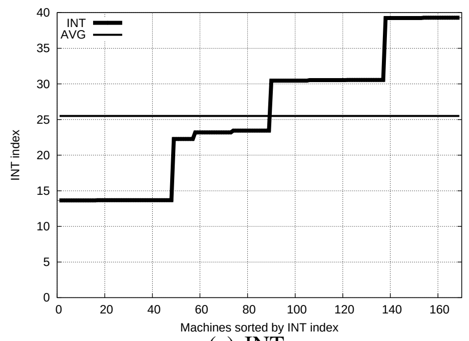
Combined together, the resources of the 169 machines are rather impressive: 56.62 GB of memory, 6.66 TB of disk and more than 98.6 GFlops of floating point performance.

3.4.2 Settings and limitations

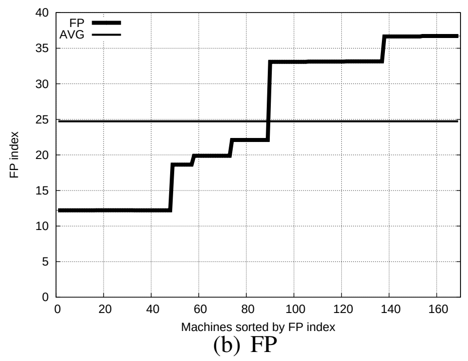
For the purpose of the monitoring experiment, the period for *W32probe* execution attempt over the set of machines was configured to 15 minutes. This

	Qty	CPU (GHz)	RAM (MB)	Disk size (GB)	INT	FP	Linpack (MFlops)
L01	16	P4 (2.4)	512	74.5	30.53	33.12	850.31
L02	16	P4 (2.4)	512	74.5	30.46	33.08	851.19
L03	16	P4 (2.6)	512	55.8	39.29	36.71	903.18
L04	16	P4 (2.4)	512	59.5	30.55	33.15	847.23
L05	16	PIII (1.1)	512	14.5	23.19	19.88	389.49
L06	16	P4 (2.6)	256	55.9	39.24	36.65	899.32
L07	16	P4 (1.5)	256	37.3	23.45	22.10	520.10
L08	9	PIII (1.1)	256	18.6	22.27	18.64	396.52
L09	16	PIII (0.65)	128	14.5	13.65	12.21	227.37
L10	16	PIII (0.65)	128	14.5	13.68	12.22	227.33
L11	16	PIII (0.65)	128	14.5	13.68	12.22	227.32
Total	**169**	**-**	**56.25 GB**	**6.66 TB**	**4315.69**	**4164.98**	**98654.12**
Avg.	**-**	**-**	**340.83 MB**	**40.33 GB**	**25.54**	**24.64**	**583.75**

Table 3.1: Main characteristics of the monitored machines.



(a) INT



(b) FP

Figure 3.2: Machines sorted by their relative computing power.

value was a compromise between the benefits of gathering frequent samples and the negative impact that this strategy might cause on resources (machines and network) and on the volume of monitoring data to process.

A 15-minute interval between samples means that captured dynamic metrics are coarse grained, with quick fluctuations of values escaping the monitoring system. For instance, a 5-minute memory activity burst using nearly 100% of main memory is indistinguishable from a 10-minute period with 50% memory usage, since samples comprising both memory usage bursts will report the same average memory space usage. However, this is seldom a problem, since all metrics are relatively stable, and thus not prone to fluctuate widely in a 15-minute interval. The only exception is the CPU idleness percentage, which is prone to quick changes. But, precisely to avoid misleading instantaneous values, CPU usage is returned as the average CPU idleness percentage observed since the machine was booted. Therefore, given the CPU idleness values for two consecutive samples, it is straightforward to compute the average CPU idleness between these two samples, given that no reboot occurred in the meantime. If a reboot occurred, then the sample taken after the reboot reports the average CPU idleness since the machine was booted.

A subtle and unexpected limitation of our methodology was due to user habits, particularly with users who forget to logout. In fact, over the original 277,513 samples captured on machines with an interactive session, we found out that 87,830 samples corresponded to users' interactive sessions lasting 10 hours or more. Since classrooms remain open 20 hours per day, closing from 4 am to 8 am, these abnormal lengthy sessions have to do with users that left their login session opened. To assert our hypothesis, we grouped the samples of interactive sessions upon their relative time occurrence since the start of the corresponding interactive session. For every time interval the average and standard deviation of CPU idleness was computed. Table 3.2 (page 63) presents these data, with the first column corresponding to the time intervals, the second one holding the count of samples, and the third displaying average CPU idleness jointly with the corresponding standard deviation. The data show that the time interval $[10-11]$ hour (i.e., samples collected during the 10th and 11th hour of any interactive session) is the first one that presents an average CPU idleness above 99% (99.27%). This very high value indicates that no interactive activity existed when the samples were collected. Therefore, in order to avoid results biased by that abnormal interactive user sessions, we consider samples reporting an interactive user session equal or above than 10 hours as being captured on non-occupied machines. Note that this threshold is a conservative approach, which means that real interactive usage is probably lower than reported in this study. Figure 3.3 (page 63) plots the number of samples (left y-axis) and the average percentage of CPU idleness (right y-axis) of data shown in Table 3.2.

An important conclusion to drawn from the forgotten sessions is that verification of user logins does not seem enough to assess machine interactive usage. Metrics like keyboard idleness and mouse usage should be used as a complementary diagnosis. However, in Windows environments,

Length of session (hour)	Number of samples	CPU idleness (stdev)
$[0-1[$	65521	91.93% (12.69)
$[1-2[$	47057	94.72% (11.08)
$[2-3[$	28374	94.54% (11.76)
$[3-4[$	13387	95.28% (12.57)
$[4-5[$	9514	96.24% (11.40)
$[5-6[$	7334	96.95% (10.28)
$[6-7[$	5654	97.43% (9.51)
$[7-8[$	4754	97.70% (9.24)
$[8-9[$	4181	98.03% (8.61)
$[9-10[$	3907	98.73% (6.09)
$[10-11[$	3637	99.27% (3.84)
$>= 11$	84193	99.61% (1.65)

Table 3.2: Samples from interactive sessions grouped by their relative time occurrence.

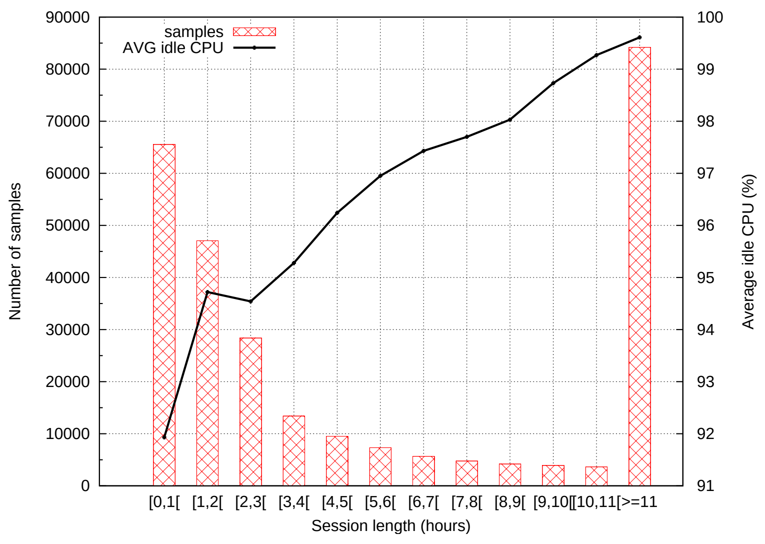


Figure 3.3: Samples from interactive sessions grouped by their relative time occurrence.
Left y-axis depicts number of samples, right y-axis plots average CPU idleness.

	No login	With login	Both
Samples (avg. uptime %)	393,970 (33.87%)	189,683 (16.31%)	583,653 (50.18%)
Avg. CPU idle	99.71% (1.99)	94.24% (11.20)	97.93% (4.99)
Avg. RAM load	54.81% (8.45)	67.53% (11.95)	58.94% (9.59)
Avg. SWAP load	25.74% (4.28)	32.83% (7.86)	28.04% (5.44)
Avg. disk used in GB	13.64 (3.30)	13.64 (4.31)	13.64 (3.63)
Avg. sent bytes in Bps	255.3 (7029.6)	2602 (31241.9)	1017.9 (14898.4)
Avg. received bytes in Bps	359.2 (5754.6)	8662.1 (47604.8)	3057.9 (19357.2)

Table 3.3: Global resource usage (values within parenthesis indicate the standard deviation σ).

the monitoring of keyboard and mouse require, to the best of our knowledge, usage of driver hooks, which not only forces software installation at remote machines, but also require the software to be run at a high privilege level. Interestingly, very high level of CPU idleness (99% or above) also seems to be a good indicator of non-interactive usage on a machine, even if an interactive session is opened. Finally, an aside conclusion to be drawn from forgotten user sessions is the need to configure classroom computers to detect long unused user sessions and to automatically logout.

3.5 Results

During the 77 days of the experiment, 6883 iterations were run with a total of 583,653 samples collected. The main results of the monitoring are summarized in Table 3.3. The column "No Login" shows the results captured when no interactive user session existed, while the column "With login" expresses samples gathered at user occupied machines. Both results are combined in the final column "Both". On rows that display average values, the standard deviation σ is given within parenthesis.

Machines responded to 50.18% of the sampling attempts over the 77 days, and in 393,970 samples (33.87%), the queried machine did not have an interactive login session. This means that during the 77 days, for slightly more than one third of the time, machines were completely available and free for resources harvesting. In fact, unoccupied machines presented 99.71% CPU idle time, expressing almost full idleness[5]. The presence of an interac-

[5]This is logical in a client machine that does not run anything, when no users are logged on, apart the OS and some services like anti-virus software and Windows Update.

tive session reduces the CPU idleness to an average of 94.24%. This means that an interactive session roughly consumes an average 5.5% of CPU. This CPU idleness confirms other studies performed in academic classrooms running Unix environments [Acharya *et al.* 1997], but with higher than the values found by Bolosky et al. [Bolosky *et al.* 2000], who reported an average CPU usage of about 15%. In fact, Bolosky et al. analyzed corporate machines from Microsoft stating that some of the machines presented an almost continuous 100% CPU usage, a fact that obviously raised mean CPU usage.

Main memory usage values are difficult to interpret recurring only to global averages, since the amount of main memory of the assessed machines ranged from 128 MB to 512 MB. However, main memory occupancy increases roughly 12% when interactive usage occurs at a machine. This is a natural behavior, since an interactive session obviously means that interactive applications will be opened and thus consuming memory. Even though, the broad conclusion is that a significant amount of memory goes unused. Again, as a consequence of higher main memory usage verified during interactive sessions, swap memory load raises by 5% when an interactive user is logged on the machine.

Used disk space is independent of the presence of interactive login sessions: average of 13.64 GB for both situations. The relatively low variability respecting used disk space, confirmed by the low standard deviation of 3.63, is a consequence of system usage policy: an interactive user is restricted to up to 300 MB of temporary local hard disk drive (the actual size depends on the capacity of the machine hard drive). Additionally, this temporary storage can be cleaned after an interactive session has terminated. This policy restricts users from cluttering disks, and also avoids that personal files can mistakenly be forgotten, or that Trojan programs and alike be maliciously dropped in shared machines. In fact, users are fostered to keep their files in a central file server with the benefit of being able to access their files independently of the desktop machine being used.

Regarding network usage, the values clearly show that the machines have mostly a client role in client-server interactions. Indeed, when used interactively the average incoming traffic is nearly four times bigger than the outgoing traffic. As importantly, the outgoing traffic under interactive usage increase 10-fold relatively to an idle machine and almost 25-fold for incoming traffic. This way, network traffic is another metric that can be used to detect interactive usage.

3.5.1 Machines Availability

Figure 3.4 (page 67) plots the count of accessible machines over the 11-week experiment. During this period, the average count of accessible machines was 84.87 (shown by the horizontal line in the plot), while the average count of occupied machines was 27.58. This means, that on average, roughly 70% of the powered on machines were free of users, and thus fully available for hosting foreign computation. Also, on average, slightly more than half of the set of 169 machines was powered on.

Figure 3.4 exhibits a sharp pattern with high frequency variations showing that machine counts fluctuate widely during a 24-hour period, with a relatively high count of accessible machines during daytime followed by a drop during the nighttime period. Weekends are recognizable by the flatness of the curve (note that the x-axis labels of the plots denote Mondays and thus weekends are on the left of these labels). Since classrooms are open on Saturdays, weekend slowdowns are more noticeable on Sundays, except for the Saturday 1st May since this was a holiday (Labor Day). The negative spike that can be found around 10th June corresponds to another holiday (Portuguese National Day), while the other negative spike on 21st June was motivated by a somewhat long maintenance period on the power circuits that feed the classrooms. The high frequency variations exhibited on weekdays mean that the count of available resources is somewhat volatile and thus harvesting the resources requires tolerant and adaptable mechanisms.

The left plot of Figure 3.5 (page 67) shows two metrics related to uptime. The double cross curve, which appears almost as a straight line, represents machine availability measured in units of *nines* [Douceur 2003]. The nine unit is defined as the $\log_{10}$ of the fraction of time a host is not available. The name of the unit comes from the number of nines in its availability ratio. For example, one nine means a 0.9 availability ratio (90%), that is, $log_{10}(1-0.9) = 1$ nine, two nines represents a 0.99 availability ratio (99%, that is $log_{10}(1-0.99) = 2$), and so on. The simple cross curve displays the fraction of time each machine is up. In both curves, machines are sorted in descending order by their cumulated uptimes.

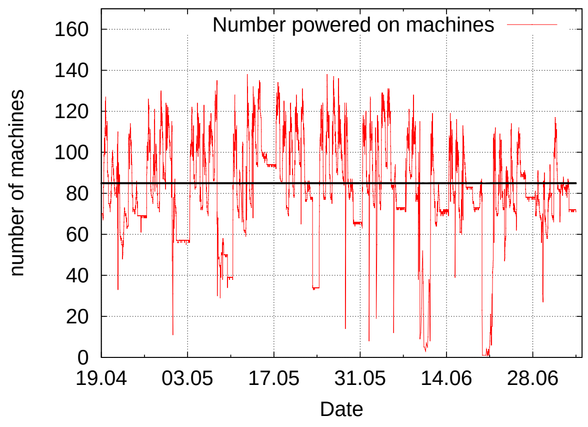
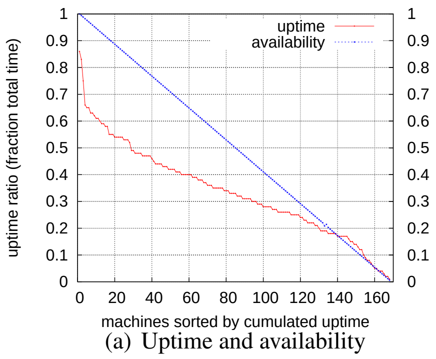
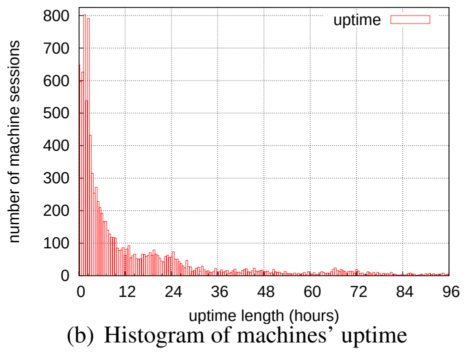


Figure 3.4: Count of powered on machines over the experiment period.



(a) Uptime and availability



(b) Histogram of machines' uptime

Figure 3.5: Machines' uptime and availability.

The ratio availability curve shows that only 30 machines have cumulated uptimes bigger than half the experiment period, that is, 37.5 days. Also, less than 10 machines have cumulated uptimes ratio higher than 0.8 and none was above 0.9. Comparatively to the Windows corporate environment described in Douceur [Douceur 2003] where more than 60% of the machines presented uptimes bigger than one nine, the analyzed classroom machines present much lower uptime ratios. This is a consequence of the possibility of the machines to be power off at the end of a class, since they have no steady users. On the contrary, they are two patterns for corporate machines: *daytime* and *24 hours*. Daytime are machines powered on during office hours, while 24 hours machines remain powered on for long periods.

3.5.2 Stability of Machines

An important fact in resource harvesting regards machine stability, that is, how long a given group of machines will be available for intensive computation. We define two levels of stability: a light level, where group stability is broken by a machine shutdown or reboot, and a more demanding level, which add to the non-reboot policy, the need for a machine to remain free of user's sessions.

Machines uptime. In this section, we analyze the machines' sessions, focusing on uptime length and reboot count. We define a machine's session as the activity comprised between a boot and its corresponding shutdown.

During the whole experiment 10,688 sessions of machines were captured by our sampling methodology. It is important to note that due to the 15-minute period between consecutive samples, some short machine sessions might have not been captured. In fact, between two samples, WindowsDDC can only detect one reboot, since its reboot detection is based upon the uptime of the machine.

The average duration of the length of sessions was 15 hours and 55 minutes. This value exhibits a high standard deviation of 26.65 hours indicating that session length fluctuates widely. Since multiple reboots that occur between two samples escape our monitoring setup (only the last one is detected), the above given average duration exceeds the real value. Figure 3.5(b) displays the distribution of the uptime length of the machines for sessions that lasted at most 96 hours (4 days). These sessions accounted for

98.7% of all machine sessions and 87.93% of cumulated uptime. These data allows us to conclude that most machine sessions are relatively short, lasting few hours, an indication of the somewhat high volatility of machines.

Machines power on cycles. Networked personal computers, especially Windows machines, have a reputation for instability, requiring frequent reboots to solve system crashes, complete software installations or simply to refresh system resources[6].

Since our sampling methodology has a coarse grained granularity of 15 minutes, some of the short machine sessions may go unnoticed. Thus, in order to have a detailed view of reboots, we recurred to SMART's parameters [Allen 2004]. Indeed, by resorting to the SMART *power cycle count* metric, it becomes possible to spot undetected machine sessions. For instance, if two consecutive samples of a machine have a difference in *power cycle count* parameters higher than one, this means that at least one short machine session, with its corresponding boot and shutdown sequence, occurred without being noticed by the monitoring mechanism.

An important issue regarding SMART is that parameters vary among disk manufacturers, not only in their availability, but also in the units used to represent the metrics [Hughes *et al.* 2002]. For instance, while almost all machines reported the power on hour in hours, the disks of two machines counted power on hour in minutes with the 16-bit value overflowing after roughly 1090 hours of disk usage. A third machine had a disk that expressed this metric in seconds, but using a 32-bit register (and thus with no risk of overflow). Finally, a fourth machine had a disk that simply did not provide the *power on hour count* parameter.

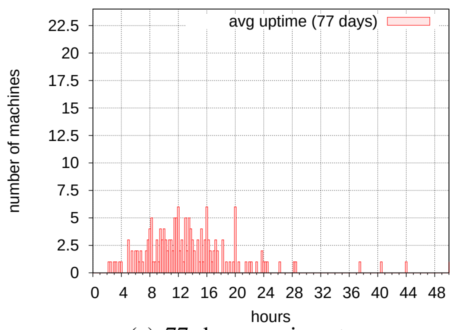
The cumulated count of *power on cycles* for the whole set of machines was 13,871, with an average of 82.57 power cycles per machine and a standard deviation of 37.05 over the 77 days. This represents 1.07 power on cycle per day. The number of power on cycles is 30% higher than the number of machine sessions counted by our monitoring analysis. This means that a significant percentage of power cycles are of very short duration (less than 15 minutes) escaping our sampling mechanism.

With the SMART's parameters *power on hour count* and *power on cycles*, we can compute the average *power on hours per power on cycle*, henceforth

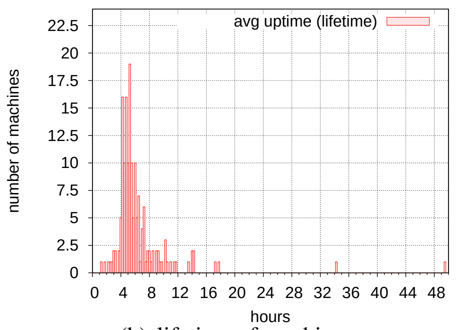
[6]To be fair, it is important to note that one of the main drivers of Windows 2000 development was to reduce the high rate of reboots of its predecessor [Murphy & Levidow 2000].

referred as *uptime per power cycle*. For the 77-day monitoring, the *uptime per power cycle* was 13 hours and 54 minutes with a standard deviation of nearly 8 hours. As stated before, the difference between the average uptime per power cycle and the average machine session length (see section 3.5.2) can be explained by the short lived sessions that are not caught by our sampling methodology. The histogram of average uptime length per power on cycle is shown in Figure 3.6(a), with the uptime counts being grouped by intervals multiple of 15 minutes.

Given the absolute count of power cycles and power on hours, it is possible to compute the uptime per power cycle for the whole disk life. Since machines are relatively new – the oldest are three years old, and the newest nine months old – the probability of machines still having their original disk is high and thus average uptime per power cycle serves as a good measure of average uptime. For our monitored system, the uptime per power cycle was 6.46 hours with a standard deviation of 4.78 hours. This value is surprisingly lower than the one we found during our 77-day monitoring. This might be due to the fact that machines are fully reinstalled at the beginning of each semester, an operation that requires many reboots. Figure 3.6(b) (right) plots the distribution of average uptime per power cycle considering the whole lifetime of disks.



(a) 77-day experiment



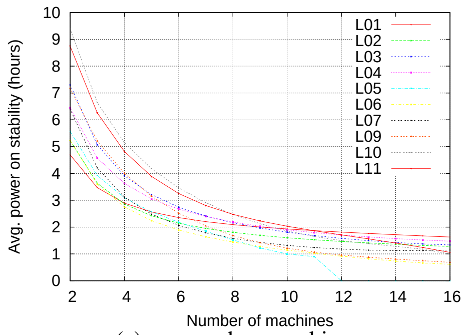
(b) lifetime of machines

Figure 3.6: Average uptime per power cycle.

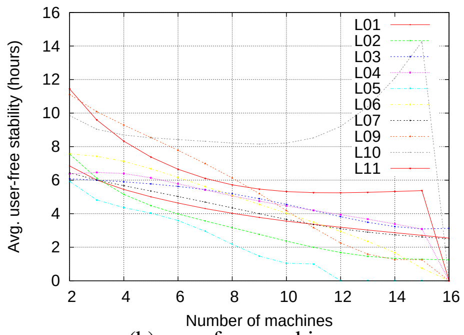
3.5.3 Group Stability

An important issue when executing parallel applications in network of non-dedicated personal computers is group stability, that is, how long a given set of machines will be available for foreign computation. In fact, some parallel environments are very sensitive to changes in the machine pool. For instance, initial implementations of the Message Passing Interface (MPI) would stop the whole computation when one or more machines failed [Al Geist *et al.* 1996].

Our definition of stability is similar to Acharya's [Acharya *et al.* 1997]: a set of K machines is said stable for a time interval T, if for T consecutive time units, all machines remain available for a joint parallel computation. This means that the failure of a machine ends up the stability for the set of K machines.

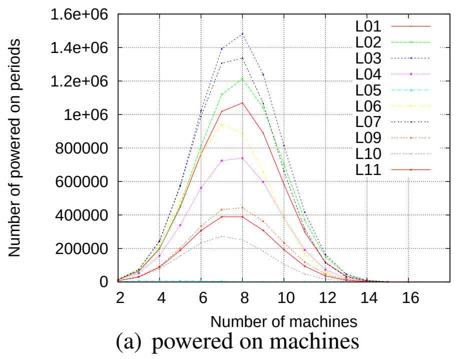


(a) powered on machines

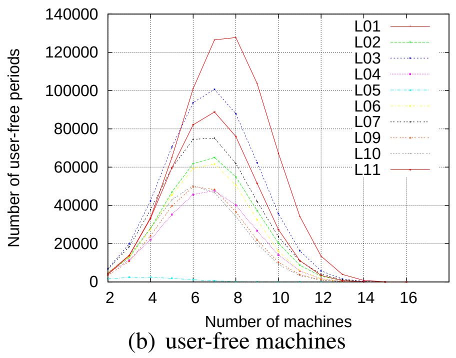


(b) user-free machines

Figure 3.7: Stability of machines per classroom.



(a) powered on machines



(b) user-free machines

Figure 3.8: Count of stability periods over N machines.

We limited the stability computation to the 10 classrooms that have 16 machines, thus excluding L08. Two views of the stability of the classrooms are shown in Figure 3.7. Both plots depict the average stability length of *power on machines*, with the number of machines ranging from 2 to 16. All curves present a similar shape, with the average stability decreasing as the number of machines in the set increases. Even so, on average, the powered on machines stability is quite interesting since that even with 16 machines, almost all classrooms presents an average stability above one hour. Figure 3.7(b) (right) presents the average stability for user-free machines. This metric differs from the average stability length of power on machines since it requires that a machine remains user-free. That is, if a user logs on a machine, this machine is no longer considered available and thus the stability of the set in which the machine was integrated terminates. The curves for the average stability of user-free machines are much more heterogeneous than the stability of power on machines. Surprisingly, the average stability periods are significantly bigger than the average length of power on stability periods (note that the plots have different *y*-scales). In reality, this is mostly a consequence of the number of stable periods, which are, quite logically, much less for user-free machines than for power on machines. For instance, the user-free spike for classroom L10 which presents a 14 hours average stable period for 15 machines is somewhat misleading since its corresponds to a single occurrence.

The number of stable periods for both situations is shown in Figure 3.8. The shape of the plots shown in Figure 3.8 reflects the fact that the number of machine combinations reaches its maximum around 8 machines. Since user-free machines are a subset of powered on machines, the number of stable periods is much higher for powered on machines than for user-free machines.

3.5.4 User Sessions

During the monitoring period, we recorded 23,224 interactive user-sessions from 1684 different logins. The average session length was 8942 seconds, that is, a little bit more than 2 hours and 29 minutes. This value is similar to the duration of most computer classes, which lasts around 3 hours. However, the duration of sessions fluctuates widely, with an observed standard deviation of 9083 seconds, that is, roughly 2 hours and 32 minutes. The histogram of the duration of the user sessions grouped by quarters of hour is plotted in Figure 3.9 (page 73). The high number of short sessions that exists – more than 2200 login sessions lasted less than 15 minutes and slightly less than 2000 lasted between 15 and 30 minutes – might be due to students login in for checking their email. Otherwise, the duration of sessions are distributed almost evenly between 30 minutes and 3 hours. For duration bigger than 3 hours, session counts drops smoothly and linearly. The spike of nearly 900 sessions that occurs at the right end of the plot is due to our truncation policy for user-session bigger than 11 hours (see Section 3.4.2, page 60).

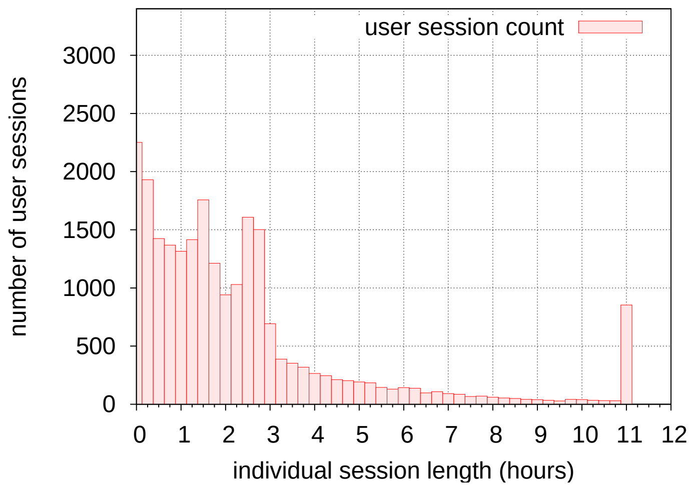
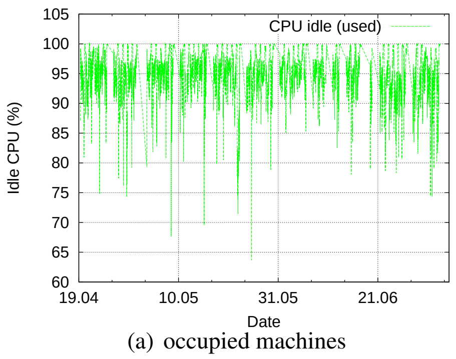


Figure 3.9: Distribution of the duration of user-sessions.

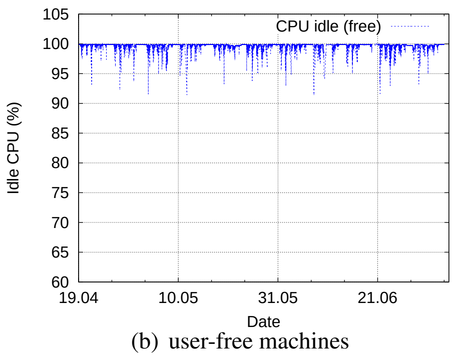
3.5.5 Global Resource Usage

Figure 3.10 plots the average percentage of CPU idleness over the whole 11 weeks. Specifically, Figure 3.10(a) displays the CPU idleness when an interactive user session exists, while Figure 3.10(b) represents the idleness of user-free machines. Both plots exhibit a similar trend: a convergence near 100% CPU idleness, obviously more noticeable in user-free machines. Once again, daytime, nighttime and weekend patterns are identifiable in the plots. This is especially true for weekends, where idleness stabilizes near 100%. In the user-occupied machines plot (Figure 3.10(a)), some drops in global average CPU idleness below 70% are visible. These drops are mostly caused by the existence of a relatively busy single machine when the number of occupied computers is very low, less than 10 units.

An interesting issue respects the CPU idleness of user-free machines, shown in Figure 3.10(b): numerous negative spikes occur, with idleness percentage dropping near 90%, a surprising condition for unused machines.



(a) occupied machines



(b) user-free machines

Figure 3.10: Average CPU idleness.

Seconds since boot	**Number of samples (no user session)**	**Avg. CPU idleness (%)**
$[0,60]$	19	77.70
$]60,120]$	154	83.89
$]120,180]$	65	86.60
$]180,240]$	49	90.44
$]240,300]$	43	91.60
Total	330	86.05

Table 3.4: Average CPU idleness on user-free machines right after boot up.

After some research, we found out that these negative spikes occur when a significant number of machines are powered on at the same time, for instance, in the first class of the day taught in a classroom. In fact, the boot up of a machine consumes a significant amount of resources and since a user can only log on after a certain amount of time, if the machine is sampled shortly after startup, but before a user has had the opportunity to log on, observed average CPU idleness will be relatively low, and since no user is yet logged on, the machine will be reported as being user-free. To support our hypothesis, we found that average CPU idleness was 86.05% in user-free machines with less than 5 minutes uptime. The average CPU idleness percentage ranging from 0 to 5 minutes uptime with no user sessions is given in Table 3.4 and plotted in Figure 3.11.

Interestingly, even for machines with interactive usage, the CPU idleness average seldom drops below 75%, and mostly fluctuates around 95%. This confirms the potentiality of CPU harvesting, even when interactive sessions exist as demonstrated by Ryu et al. [Ryu & Hollingsworth 2004].

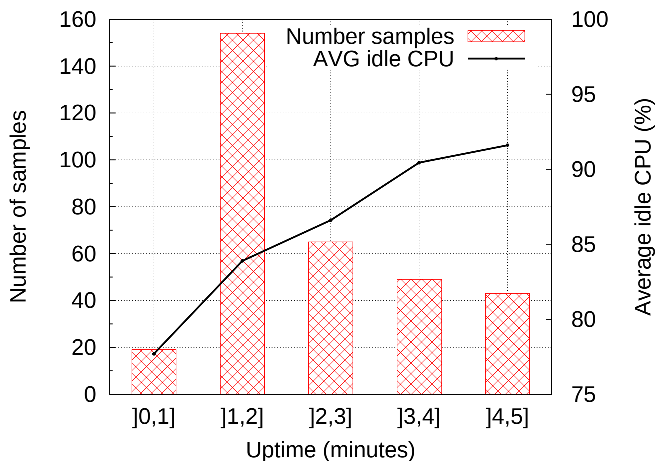
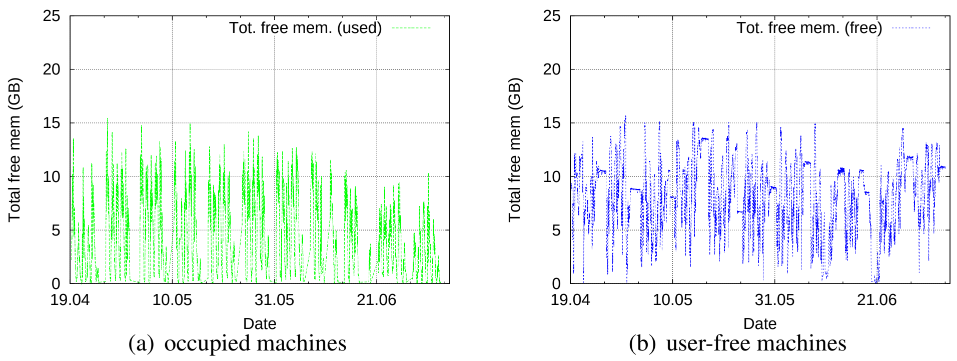


Figure 3.11: Average CPU idleness right after machine boot up.

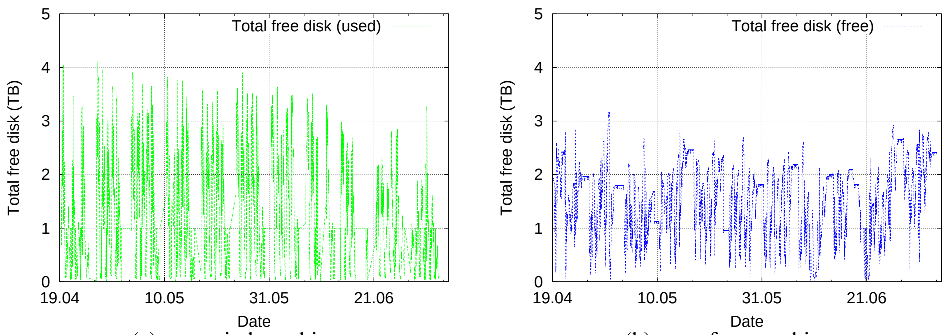
Figure 3.12 (page 75) shows the sum of free memory observed over the 11-week observation. The left plot focuses on the unused memory of occupied machines, while the right plot displays the same metric for user-free machines. Both plots mostly reflect the machines usage pattern, especially the number of powered on machines at a given time. In fact, the high frequency fluctuations in all plots derive essentially from the number of available machines, constituting another proof of the volatility of the resources.



(a) occupied machines (b) user-free machines

Figure 3.12: Sum of free memory of the machines.

Plots of free disk space cumulated from all machines, shown in Figure 3.13, exhibit the sharp high frequencies characteristics of high volatile environments. The differences of cumulated available space between used and unused machines are dependent of the ratio of machines in either condition. On weekdays, especially during work time, since most machines powered on are being interactively used, this set of machines presents normally more than 1.5 TB of available disk space. The cumulated available disk space, even if highly volatile, rarely drops under 1 TB. This is an impressive figure indicator of the dimension of resources that can be exploited in classroom machines.



(a) occupied machines (b) user-free machines

Figure 3.13: Cumulated free disk space of the machines.

3.5.6 Weekly Analysis

Figure 3.14 (page 77) combines two plots related to the weekly distribution of samples: the left plot characterizes the average number of powered on machines (top curve) and the standard deviation (bottom curve), while the right plot shows the weekly distribution of CPU idleness's average. Both plots reflect the weekly pattern with stable curves on weekends, especially Sundays, and during the nights. Besides following the night and weekend pattern, the weekly distribution of average CPU idleness presents a significant plunge on Tuesdays afternoons, dropping below 91%. Although we could trace back the Tuesdays' afternoon CPU usage spike to a practical class which was taught in one classroom and consumed an average of 50% of CPU, we could not determine what actually caused this abnormal CPU usage.

Confirming the high availability of CPU for resource scavenging, the average CPU idleness never drops below 90%, mostly ranging from 95% to 100%. The phases of near 100% average CPU idleness correspond to the periods when classrooms are closed: 4 am to 8 am during weekdays, and from Saturdays' 9 pm to Mondays' 8 am on weekends.

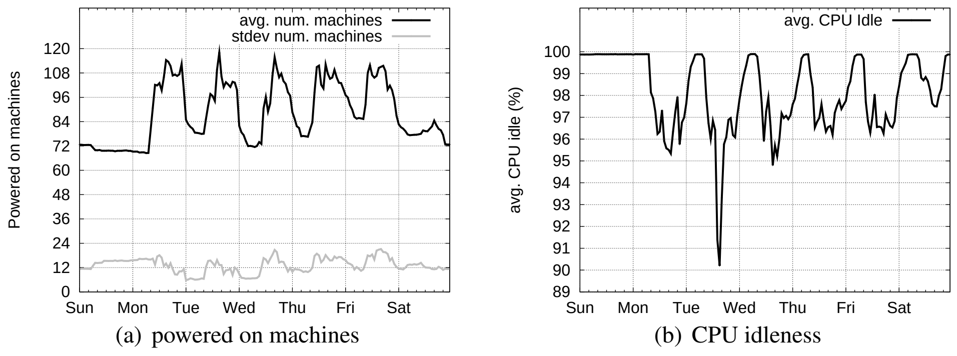


Figure 3.14: Weekly distribution of powered on machines and average CPU idleness.

Figure 3.15(a) shows the weekly average memory distribution with the top curve representing average RAM load and the bottom curve showing average swap load. The right plot (Figure 3.15(b)) depicts the average network rates for received (top curve) and sent traffic (bottom curve).

Both the RAM and swap load curves exhibit the weekly pattern, although in a smoothed way. Note that RAM load never falls below 50%, meaning that a significant amount of RAM is used by the operating system. Comparing RAM and swap usage, it can be observed that the swap curve roughly follows memory usage, although strongly attenuating high frequencies. This is a natural consequence of how memory systems are organized.

The weekly distribution of network rates (Figure 3.15(b)), is yet another example of the weekend/nighttime trend line. Since we are plotting a rate, the drops originated by nighttime periods and weekends appear as smoothed lines. The network client role of the classroom machines appears clearly visible, with received rates up to several times higher than sent rates.

From the observation of the weekly distribution of resource usage, we can conclude that even if a lower number of machines is powered on off-work periods (that is, from 4 am to 8 am on weekdays, and on weekends), these resources are much more stable than the more numerous workday-time ones. However, even on working hours, idleness levels are quite high.

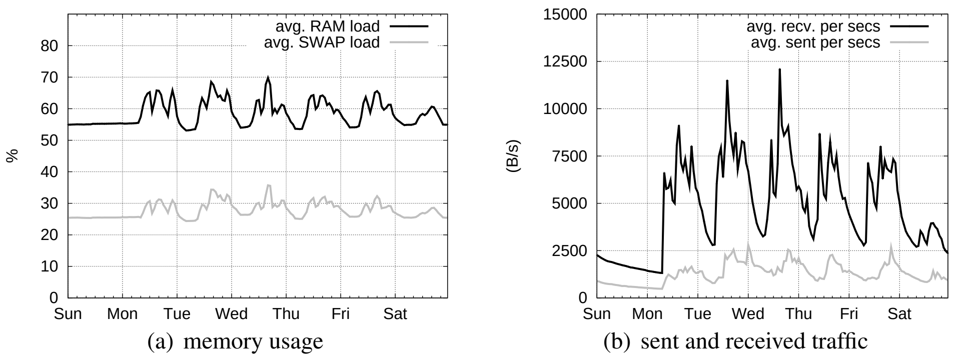
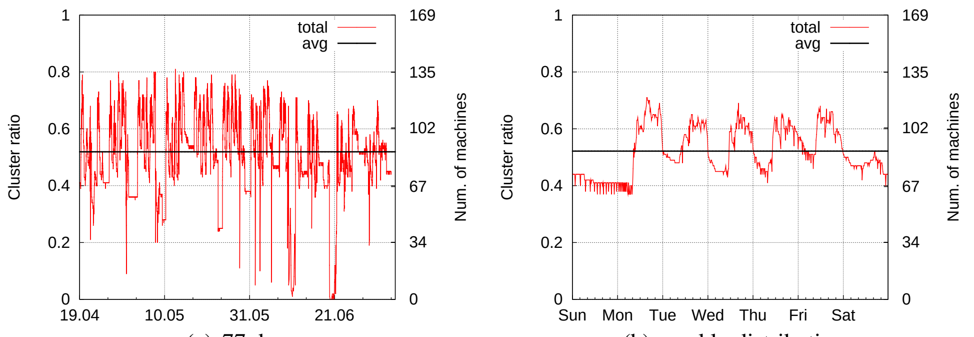


Figure 3.15: Weekly distribution of free memory and network traffic.

3.5.7 Equivalence ratio

Arpaci et al. [Arpaci *et al.* 1995] define the *equivalent parallel machine* as a metric to gauge the usable performance of non-dedicated machines relatively to a parallel machine. Kondo et al. [Kondo *et al.* 2004a] adopt an analogue metric, based upon clusters, and which they appropriately name the *cluster equivalence ratio* (CER). Specifically, the *cluster equivalence ratio* measures, for a given set of non-dedicated machines, the number of cluster machines (that is, dedicated machines) that would be needed for achieving an equivalent computing power. In summary, the *cluster equivalence ratio* aims to measure the fraction of a dedicated cluster that a set of non-dedicated machines is worth to an application, considering solely the CPU. In this study, we apply this definition by computing the CPU availability of a machine for a given period accordingly to its measured CPU idleness over the considered period. For instance, a machine with 90% of CPU idleness is viewed as a dedicated machine with 90% of its computing power. It is important to note that this methodology assumes that all idle CPU can be exploited. Thus, the obtained results should be regarded as an upper limit of CPU resources that can effectively be harvested.

To cope with the performance heterogeneity of machines, the computation of the *performance cluster ratio* was done resorting to a combined INT and FP index, named INTFP. Specifically, a 50% weight was given to each index to compute the combined machine index. For instance, a machine of L03 (INT:39.29, FP:36.71) is worth 1.19 of a L01's machine (INT:30.53, FP:33.12).



(a) 77 days (b) weekly distribution

Figure 3.16: Cluster equivalence ratio.

Figure 3.16 plots the cluster equivalence ratio for the 77-day experiment (left) and its weekly distribution (right). The average cluster ratio is 0.26 for occupied machines and 0.25 for user-free machines. Combining together occupied and unoccupied machines yields a 0.51 cluster equivalence ratio, meaning that the set of non-dedicated machines is roughly equivalent to a dedicated cluster with half the size. This follows the 1:2 rule found by [Arpaci *et al.* 1995].

3.6 Related Work

The evaluation of computer resources usage has been a research topic since the wide deployment of networked computers in the late 80's. In fact, soon it was noticed that computer resources, noticeably CPU, were frequently underused, especially in machines primarily used for tasks dependent on human interaction.

Several studies have observed the high level of resources idleness in networked computers, not only about CPU [Arpaci *et al.* 1995; Acharya *et al.* 1997], but also memory [Acharya & Setia 1999] and disk storage [Bolosky *et al.* 2000].

Through simulation, Arpaci et al. study the interaction of sequential and parallel workloads [Arpaci *et al.* 1995]. They conclude that for the considered workloads, a 2:1 rule applies, meaning that N non-dedicated machines are roughly equivalent to N/2 dedicated machines. We found a similar value for our set of machines.

The study presented in [Acharya *et al.* 1997] focuses on the potentiality of using non-dedicated Solaris machines to execute parallel tasks during idle periods. The authors evaluate the machines availability and stability based upon a 14-day trace of computers that are primarily assigned to undergraduate students. The authors observe that reasonably large idle clusters are available half the time noting, however, that such set of machines are not particularly stable, that is, machines frequently go down.

Acharya and Setia analyze the main memory idleness and assess its potential utility by using a two-week memory usage trace from two sets of Solaris workstations [Acharya & Setia 1999]. One set includes machines fitted with, at the time of the study, a high amount of main memory (total of 5.2 GB for 29 machines, averaging 183 MB per machine), while the other set is more modest (total of 1.4 GB for 23 machines, that is, an average of 62 MB per machine). The study shows that, on average, a large fraction of the total memory installed on a machine is available, with idle machines presenting around 50% of unused main memory.

Ryu et al. aim to harvest idle resources from what they define as non-idle machines, that is, machines that are lightly loaded by interactive usage [Ryu & Hollingsworth 2004]. They conclude that a vast set of idle resources can be harvested without much interference on interactive users. However, their methodology requires modification at the kernel level and therefore seems impractical for closed operating systems like Windows.

All of the aforementioned cited studies are focused on UNIX environments and rely on somewhat reduced traces to draw their conclusions. Our analysis targets Windows machines, monitoring a medium sized set of machines over a relatively long period of time.

Bolosky et al. have conducted a study in a vast set of nearly 30,000 Microsoft Corporate desktop machines, reporting availability, CPU load and file system usage in a corporate environment [Bolosky *et al.* 2000]. The study is oriented toward the demonstration of the potential to build a serverless distributed file system. Thus, issues discussed in this chapter such as main memory load, network usage, and interactive sessions and its impact over resource usage are not reported. In contrast, our work is focused on categorizing resources usage, and we have some results that are substantially different, especially respecting a much lower CPU load than observed in the corporate environment depicted in [Bolosky *et al.* 2000].

This is mostly due to the fact that the machines we monitored were essentially client computers employed for interactive usage, with no really demanding computing activity taking place.

Heap studies the resource usage of Unix and Windows servers, through 15-minute periodic gathering of monitoring data [Heap 2003]. He found out that Windows servers had a CPU idleness average near 95%, while Unix servers averaged 85% CPU idleness, showing that even server machines present high levels of CPU idleness.

Kondo et al. studied the CPU and host availability from the perspective of desktop grid applications in an Entropia environment [Kondo *et al.* 2004a]. They collected usage data for 220 machines of an academic campus for a total of 28 days. Most machines were assigned to researchers, and thus were effectively "owned" by an individual. The measurements were conducted by submitting actual tasks to the Entropia system [Chien *et al.* 2003]. These tasks performed computation and periodically reported their computation rates, thus allowing for the characterization of the availability of hosts and of CPUs. Note that this approach is interesting in the sense that it allows for a real characterization from a scavenging point-of-view. However, such characterization is only possible in a desktop grid environment fitted with an desktop grid middleware like the Entropia middleware, while our approach required no installation of software at the monitored nodes. The average host availability of the environment was 3 hours during weekdays and slightly less than 6 hours for weekends. The cluster equivalence ratio average was slightly under 0.70 for weekdays and near 0.90 for weekends. Relatively to the cluster equivalence ratio found in our study, the value are significantly higher. We believe this is mostly due to the fact that machines studied by Kondo et al. were assigned to individual users, thus having a much more restricted and stable community of users than the classrooms where we conducted our study.

In their assessment of around 330,000 BOINC/SETI@home's hosts, Anderson and Fedak report that an average of 89.9% of the machines' CPU time is effectively yielded to SETI@home [Anderson & Fedak 2006]. This means that roughly 90% of the CPU computing power is actually harvested, confirming not only the high idleness of CPU but also the effectiveness of scavenging schemes such as BOINC. Furthermore, the average PC participating in the project, as of February 2006, had an impressive configuration:

1.568 GFlops of processing power, 819 MB of main memory (2.03 GB swap) and 63 GB of disk space, with more than half of the disk space (36 GB) left unused. This attests the wealth of resources that can potentially be harvested for the purpose of public volunteer computing.

Our approach is distinct from previous work by focusing on academic classrooms fitted with Windows desktop machines. The workload traces were collected for 11 weeks over a medium-sized set of 169 desktop machines (Pentium III and Pentium 4). An analysis of main resource usage is conducted with emphasis in differentiating resource usage between machines occupied with interactive users and free machines, assessing the effect of interactive user over resources consumption. Furthermore, we also present a novel approach to assess machines availability, combining collected samples with data extracted from SMART counters of machines' hard disks [Allen 2004], namely the *power on hour counts* and *power on cycle* counters. Coupled with the collected traces, these SMART values allow to infer about machines *power on pattern*, giving a rough estimation of the machines' availability since hard disk was installed, which is, for most of the computers, the date they were built.

3.7 Summary and Discussion

This chapter presented the main results of a 77-day monitoring usage study of 169 Windows 2000 machines. The results show that resources idleness in the studied academic classrooms comprised of Windows machines is very high, confirming previous works such as [Heap 2003] and [Bolosky *et al.* 2000].

The main conclusions that can be drawn from this chapter are:

- CPU idleness is impressively high, with an average of 97.93%. Likewise, the 94.24% average CPU idleness measured in user-occupied machines indicates that CPU harvest schemes should be profitable not only when a machine is unoccupied but also when interactive usage of the machine exists.

- Another interesting result associated to CPU is that interactive usage of the machine can be sensed by the level of CPU idleness: CPU idleness above 98% almost certainly means that the machine is not being interactively used, even if a login session exists.

- The cluster equivalence ratio is 0.26 for interactively used machines, and 0.51 when user-free machines are also considered. This means that the 169 non-dedicated machines are roughly worth 86 dedicated machines.

- Memory idleness is also noticeable, with respectable amounts of unused memory, especially in machines fitted with 512 MB of main memory. Coupled with a relatively fast network technology such as 100 Mbps switched LAN or gigabit Ethernet, such resources might be used for network RAM schemes such as networked temporary RAM drives.

- Due to the fact that the single hard disk of all machines only contains the operating system installation plus specific software needed for classes, the free space storage among monitored machines is high. And with the trend for exponential growth of hard disks, the tendency is that more and more unused disk space becomes available, at least in sites whose management policy limits the use of local machines' disks to temporary storage. A possible application for such disk space relates to distributed backups, or to the implementation of a local data grid.

- Usage of the SMART hard disk accountability metrics can yield some average indications about the number of power on hours and the number of reboots of a given machines.

We believe our results can be generalized to other academic classrooms that resort to the Windows operating systems and which follow a similar classroom usage policy: shared machines with minimal disk space for interactive user, and student off-class access to computers for work assignment and e-communication use.

Overall, classrooms comprised of Windows machines seem appropriate for desktop grid computing. The attractiveness of such environments for resources harvesting is strengthened by the fact that machines have no real personal *owner*, being managed by a central authority. This eases the deployment of resource harvesting schemes, since an authorization from the central authority gives access to the shared machines. Obviously, it is still important for harvesting environments to respect interactive users, guaranteeing that interactive usage is not degraded by harvesting schemes.

In conclusion, this chapter confirms that resource idleness observed in classroom environments has great potentiality. Indeed, if properly channeled, these resources can yield good opportunities for grid desktop computing. In the next two chapters, we propose and analyze some scheduling strategies to overcome the volatility of resources and to speed up turnaround times of applications.

4 Fault-Tolerant Scheduling

In this chapter, we focus on fault-tolerant scheduling for institutional desktop grids. The goal is to devise fault-tolerant aware scheduling policies in order to provide for fast execution of bag-of-tasks applications over non-dedicated institutional desktop grids. For this purpose, we propose several scheduling policies supported by *shared checkpointing* methodologies, where checkpoint images can be shared among nodes. Under this approach, a task interrupted in one host can be resumed from the last stable checkpoint as soon as another host is available. In addition, the shared checkpoint methodology also allows the replication of tasks, which can be important not only for fault tolerance purposes, but also for speeding up the turnaround time of the application. We conclude the chapter with the presentation of `DGSchedSim` simulator, a simulator that we developed to evaluate the scheduling policies herein presented.

4.1 Introduction

4.1.1 Fault Tolerance and Checkpointing

A common solution to cope with the limitations imposed by volatility of desktop grids is checkpointing [Elnozahy *et al.* 2002]. It consists in periodically saving the state of the executing process to stable storage. Whenever a computation is interrupted, the application can be resumed from the last available checkpoint as soon as the original resource or an equivalent one is available. This way, the loss caused by a failure is reduced to the computing time elapsed since the last usable checkpoint was saved plus the overhead of restoring the computation.

Broadly, two main types of checkpoints exist: system-level and application-level [Silva & Silva 1998]. The former involves saving the whole state of the process that executes the application to be preserved. Although transparent to programmers and users, system-level checkpointing usually generates big chunk of data to be saved (in the order of hundreds of megabytes, if not more), since the whole process's memory image needs to be preserved. This significantly increases the costs of the checkpointing operations. Furthermore, a system-level checkpoint is strongly tied to the operating system where it was created, and thus can only be used for restoring execution in a compatible system, that is, a machine with a matching hardware and with the same operating system version. This seriously hinders the portability of system-level checkpointing schemes, especially in heterogeneous environments like desktop grids. On the contrary, checkpointing at the application-level requires the direct intervention of the application programmer to point out and explicitly code the meaningful data that need to be preserved. However, since only the necessary state is saved, application-level checkpoints are way lighter than system-level. Moreover, if saved in a transportable format (e.g., XML), checkpoints can be used to restore the execution in another machine, possibly with a different hardware and operating system (as long as a version of the application exists for the restore machine), thus effectively allowing task migration. As stated previously in section 2.6.1 (page 28), apart from Condor, which relies on system-level checkpointing [Litzkow *et al.* 1997] all major desktop grid middleware like BOINC and XtremWeb resort to application-level checkpointing. System-level checkpointing is impractical for such systems due to OS limitations (to the best of our knowledge, system-level checkpointing is not available in Windows) and due to the disk space requirements. Therefore in this thesis, we only consider application-level checkpointing.

An important issue regarding checkpointing lies in the storage location. Indeed, a usual limitation of desktop grid-based computing is that checkpoints are private, in the sense that a checkpoint saved in a given machine will only be used to resume the application in that machine. Moreover, if the machine has a large downtime period (for example, it is power down for the weekend), any checkpoint kept there is inaccessible. In this chapter, we explore the advantages of sharing checkpoints in a desktop grid environment for the purpose of speeding up turnaround time. Under the shared checkpoint approach, portable checkpoints are saved in a network shared storage and can be used for restoring, moving or replicating tasks to another machines.

4.1.2 Institutional Desktop Grids

As stated before in section 2.2, the usefulness of desktop grids is not limited to public projects. In fact, it is frequent for institutions like corporations and academia to own a significant number of personal computers, primarily devoted to low demanding computing activities like e-office and other similar interactive tasks. These machines can be used by local users to execute demanding e-science applications like simulations or image processing. The attractiveness of institutional desktop grids is reinforced by their network infrastructures which can benefit from local and metropolitan fast network technologies (100 Mbps Fast Ethernet is standard in such environments, often complemented with a Gigabit Ethernet backbone). Also, comparatively to the resources volunteered to public projects, institutional desktop grids are usually comprised of more homogeneous computing infrastructures, with, for instance, a substantial percentage of machines matching a unique operating system and software profile[1]. Also, security can be more effectively controlled, with incidents like malicious usage and anomalies being potentially traced back to the source in a more efficient manner. Finally, the usual existence of a central entity charged of managing and coordinating the computing resources and their usage, allows the adoption and enforcement of policies regarding shared and cooperative usage of resources, limiting uncooperative behavior from users unwilling to share the computing resources made available to them by the institution.

This chapter focuses on institutional desktop grids comprised of ownerless machines, as found in academic classrooms. The ownerless designation comprehends machines that are not assigned to any individual in particular, contrary to office machines that are normally under the control of a given user.

[1]Hardware might be more variable, with newer machines delivering better performance and having bigger resources, namely memory and disk, than older ones.

4.1.3 Bag-of-tasks Applications

Mostly due to the communication limitations, desktop grid applications are mostly restricted to sets of independent tasks executed under a supervisor-worker paradigm, upon which a central supervisor entity coordinates the whole computation. Under this model, a worker requests tasks from the supervisor, processing the tasks it receives in background mode and under minimal local system priority, since precedence to access resource is always given to local users. Upon the completion of the task, results are sent back to the supervisor and a new task is requested by the worker. In order to achieve high resource efficiency in such environments, communications must be kept to a minimum, with tasks being essentially CPU bound, thus having a high computation to communication ratio. This type of application is commonly referred as bag-of-tasks applications [Cirne *et al.* 2003].

4.1.4 Turnaround Time of Bag-of-Tasks Applications

The *turnaround time* of an application is defined as the wall clock time elapsed between the start of the application and its termination, when it is executed over a given set of resources. Since a bag-of-tasks application is comprised of several tasks, the turnaround time corresponds to the elapsed time between the start of the first task and the completion of the last task.

We consider only applications comprised of independent tasks, that is, no dependency exist among the tasks, so that the tasks can be freely scheduled. This way, the *first* task is simply the first task to be scheduled, and the *last* task is the last one to be completed.

Turnaround time is especially relevant in iterative and/or speculative research based on computing, like for instance computer-based simulations [Petrou *et al.* 2004]. Indeed, in iterative/speculative based research, the next steps of the work are dependent on the outcome of the current execution. Moreover, speeding up the turnaround time allows to explore more hypotheses, discarding the bad ones, or alternatively, to faster reach a solution. It is important to note that a turnaround time oriented scheduling inevitably hinders the overall throughput of the desktop grid system. So, whereas in public volunteer projects the emphasis is normally on the number of tasks carried out per time unit (throughput), our approach privilege a fast execution of applications possibly at the cost of the whole system throughput, inclusively by sacrificing resources with replication of tasks, if this might contribute for a shorter turnaround time.

4.2 Scheduling Policies

In this chapter, we propose several scheduling policies focused on delivering fast turnaround time for typical bag-of-tasks applications executed over institutional desktop grids. Specifically, we evaluate the turnaround time delivered by several heuristic-oriented scheduling policies. Besides sharing checkpoints, the heuristics include adaptive execution timeouts, task replication with and without checkpointing on demand, and short-term prediction of resource availability.

For the purpose of resource harvesting, we consider that the non-dedicated machines are partitioned in sets, with a given set assigned to a harvesting user for a fixed period of time. This model corresponds to a time partitioned management of the machines for resource harvesting. For instance, considering an academic environment, a representative example would be a single user having permission to harvest, for a whole day, the machines of a given number of classrooms. It is important to note that under our assumed scenario, the machines are shared between their regular users and the desktop grid user, who is, at least for the assigned time frame, the sole user of low priority non-dedicated cycles. Furthermore, the emphasis for the use of harvesting resources is clearly on pursuing a fast turnaround time, so that the scheduling methodology might trade resources for achieving a faster execution time. For instance, if deemed appropriate, a task can be replicated among multiple machines, even if this means that when the first replicated instance of a task terminates, the computation performed by the other replicas will be discarded and, thus the computing power will have been effectively wasted. Given these conditions, our goal is to devise and assess scheduling strategies based on checkpoints that promote faster turnaround times.

4.2.1 Scheduler Knowledge

We assume a medium knowledge-base scheduler [Anglano *et al.* 2006], upon which the scheduler is periodically (for instance, every 2 minutes) fed with data about the availability of the machines. This is a realistic assumption in institutional desktop grids, and can be achieved resorting through a simple heartbeat mechanism or through a monitoring environment like Ganglia [Massie *et al.* 2004]. The periodic knowledge of the availability status of the machines is useful for scheduling purposes. This is especially true for a checkpoint-sharing aware scheduler, which can react much more effectively to a sudden machine unavailability (whether the machine was voluntarily power off by its owner, or a crash occurred), since it can reschedule the task formerly assigned to the now unavailable machine to another machine that can resume it from the last stable checkpoint. On the contrary, a scheduler based on private checkpointing, would only be able to restart the task from scratch, since the checkpoint file would be inaccessible. Thus, for shared checkpoint policies, we assume that the checkpoint server is co-located with the scheduler.

Next, we present each of the proposed scheduling policies, starting by the First-Come-First-Served (FCFS) methodology, which we use as a reference.

4.2.2 FCFS

First-Come-First-Served (FCFS) is the classical eager scheduling algorithm for bag-of-tasks applications, where a task is simply delivered to the first worker that requests it. This scheduling policy is particularly appropriate when high throughput computing is sought and thus it is commonly implemented by major desktop grid middleware like BOINC, Condor and XtremWeb. This way, it is used by the main volunteer grid desktop projects. However, FCFS is normally inefficient if applied unchanged in environments where fast turnaround times are sought, especially if the number of tasks and resources are in the same order of magnitude, as it is often the case for local users' bag-of-tasks [Kondo *et al.* 2004b]. Indeed, a FCFS-based scheduler assigns tasks following the order of the workers' requests. This might create undesirable schedules, especially in the last stage of the application, where the number of tasks to compute is less than the number of available resources. For instance, consider the extreme situation of the last task being assigned to the slowest machine of the pool, simply because the slowest machine requested a task.

In order to adapt the FCFS policy to fast turnaround-oriented environments, we applied several changes. First, we added support for sharing checkpoints to promote checkpoint mobility, and consequently for an improved usage of resources relatively to private checkpointing. Then, based on FCFS, several scheduling policies were devised, namely FCFS-AT (AT stands for *adaptive timeout*), FCFS-TR (*task replication*), FCFS-TR-DMD (*task replication with checkpoint on demand*) and FCFS-PRDCT-DMD (*prediction with checkpoint on demand*). Next, we describe each of the proposed scheduling policies.

4.2.3 FCFS-AT

FCFS-AT improves the base FCFS with the inclusion of an adaptive timeout that defines a maximum time for the machine to complete the execution of the assigned task. Should the timeout expire before the task has completed, the scheduler considers the task as non-terminated and will therefore reassign it to another machine, where it will be restarted, possibly from the last stable checkpoint, if shared checkpointing is enabled. However, under a private checkpoint model, the task needs to be restarted from scratch in an available machine, unless the original executing machine rapidly returns to availability, which allows the task to be resumed from the private checkpoint.

The timeout is computed each time the task is assigned to a requesting machine. The timeout computation takes into account the needed CPU time to complete the task (measured relatively to a reference CPU), as well as, the machine performance as given by the *Bytemark* benchmark indexes [bytemark 2007]. Both values are used to estimate the minimum time needed by the candidate machine to complete the task assuming an ideal execution (i.e., a fully dedicated – 100% CPU for the task – and a flawless execution without interruptions). An heuristic tolerance time is added to the base timeout to provision for overhead and the possible existence of interactive usage of the machine. This tolerance time depends on the expected running time of the task under the ideal conditions and also on the time of the day. Table 4.1 shows the factor by which the estimated completion time under ideal conditions is multiplied to obtain the total timeout time. We introduced different factors for the nighttime periods and the weekends to take into account the likely stability of the desktop grid resources during period of low or non-existence of human presence as it was observed in Chapter 3.

CPU reference time (secs) or time of day	**Timeout factor**
≤ 1800	1.500
$]1800, 3600]$	1.325
$]3600, 7200]$	1.250
> 7200	1.150
Nighttime (0.00-8.00 am)	1.100
Weekend (Sat,Sun)	1.050

Table 4.1: Timeout tolerance factors for FCFS-AT and derived policies.

4.2.4 FCFS-TR

FCFS-TR adds task replication on top of the FCFS-AT scheduling policy. The strategy is to resort to task replication at the terminal stage of the computation under the condition that all uncompleted tasks are already assigned and there is at least one free machine. Replicating the task augments the probability of a faster completion of the task, especially if the replica is scheduled to a faster machine than the current one. Even if a replica is assigned to a slower or to an equal performance machine, it can still be useful by acting as a backup in case the primary machine fails or gets delayed.

In all replica-based policies, the number of replicas-per-task (henceforth *replica count*) is limited by a predefined threshold. The purpose of the replica count limitation is to promote fairness in the replica functionality, avoiding that some excessively replicated tasks clutter the resources. Therefore, the tasks whose replica count has already reached the limit can only be further replicated when one of the current replicas is interrupted. Moreover, when a task is terminated all other results produced by replicas that might exist are simply discarded.

4.2.5 FCFS-TR-DMD

FCFS-TR-DMD adds the *checkpointing on demand* feature to task replication implemented by FCFS-TR. As the name suggests, checkpointing on demand allows the scheduler to order a worker to perform a checkpoint operation on the task it is currently executing. This fresh checkpoint is useful immediately before the creation of a replica, since it allows the replica to start with an up-to-date state of the source task. In fact, resorting only on regular checkpoints would mean that a valid but possibly aged state of the task would be used to create the replica, and thus computation already performed would need to be redone by the computer hosting the replica.

It is important to note that checkpointing on demand poses two requirements: (1) the capability for the supervisor to initiate communication with workers (to demand the checkpointing operation) and (2) the ability of a task to be checkpointed at any time. In fact, supervisor-initiated communication might be avoided if the desktop grid system allows the supervisor to send information and commands in response to a worker's periodic heartbeat. However, this approach would induce a delay on the supervisor demand to checkpoint, in average equal to half the heartbeat's periodicity. Furthermore, depending on the granularity of the task and how the application checkpointing was programmed, the ability to perform an immediate checkpoint might not be feasible. For instance, the execution of a task that implements an iterative execution is comprised of successive iterations. It is reasonable to suppose that checkpoints can be saved between but not during iterations. Therefore, if an iteration requires T time units, in average, saving a checkpoint can only occur $T/2$ time units after it has been ordered. In this work, we do not consider the delays that may be imposed by an imperfect checkpointing on demand mechanism.

4.2.6 FCFS-PRDCT-DMD

The FCFS-PRDCT-DMD scheduling policy resorts to short-term prediction regarding machines' availability on top of FCFS-TR-DMD. When a prediction indicates that a currently requested machine might fail in the next scheduling interval, the scheduler requests a checkpointing on demand operation and promotes the creation of a replica if conditions are met, that is, if at least a free machine exists and the maximum number of replicas for the considered task has not yet been reached. The rationale behind this policy is to anticipate the potential unavailability of machines, taking the proper measures to reduce or even eliminate the effect of the machine unavailability on the execution of the application.

The used prediction method was the Sequential Minimal Optimization (SMO) algorithm. This algorithm was selected since it yielded the best prediction results for machine availability, as shown by Andrzejak et al. [Andrzejak *et al.* 2005].

4.3 Segmented Execution with Shared Checkpoints

In this section, we detail the concept of *segmented execution* that aims, under ideal execution conditions, to achieve an optimal turnaround time of an application comprised of T homogeneous tasks when executed over a fixed set of M homogeneous machines. Note that we solely address homogeneous tasks, since scheduling heterogeneous tasks, even over homogeneous resources is NP-hard [Pineau *et al.* 2005].

The goal of the segmented execution approach is to split the execution in equal-sized segments and schedule them so that full usage of the computing resources is achieved, yielding an ideal execution time. The main idea is to segment the execution of tasks, splitting a task into several temporal segments. These segments are then executed sequentially (for instance, segment one needs to be completed before the execution of segment two can start, and so on), with the execution possibly scheduled over different machines. To allow a segmented execution, a shared checkpoint/restart mechanism is assumed, allowing not only temporal segmentation of the execution of tasks but also that the segments of a task can be executed at any machine.

The rationale behind the segmented execution approach comes from the fact that when considering execution under ideal conditions, namely failure-free and fully dedicated machines, it is uncommon to have all machines fully occupied by tasks during the whole execution, and thus computing power is lost, lengthening turnaround time. In fact, depending on the ratio T/M and considering a non-segmented execution, some machines will be left for some periods with no tasks to process. Specifically, this happens in the last stage of execution, when all tasks have at least been already started and if the number of tasks, T, is not a multiple of the number of machines. For instance, consider the simplest case, where M homogeneous machines are set to execute T tasks, each one requiring a single CPU time unit to complete. The time needed to carry out the execution of the tasks is given by $\lceil T/M \rceil$. Unless T is a multiple of M, the last stage of executions (when the machines receive their last task to execute) will only involve $T \bmod M$ machines (considering that the previous execution stages fully occupied all M machines), and thus $M - (T \bmod M)$ machines will be left idle without tasks to execute. The goal of the segmented execution is precisely to schedule tasks in such a way that in every execution stage, all M

machines are fully occupied. For this purpose, execution of all T tasks need to be done in T/M segments, each task being split in M segments. Since, at any given time, at most M tasks can be executing (one per machine) and since the execution of the segments of a task imposes sequentiality (the second segment can only be executed after the first one and so on), the execution needs to be carefully scheduled in order to yield a turnaround time of T/M time units. As stated before, the segmented scheduling requires the interruption of tasks since the segments of a given task are not necessarily executed consecutively.

4.3.1 Example of a segmented execution

To make matters clear, we give an example of an ideal segmented execution considering an homogeneous set of machines and an homogeneous set of tasks. Specifically, we consider a set of 3 homogeneous machines ($M = 3$, with M_1, M_2, M_3) and an application comprised of 5 homogeneous tasks ($T = 5$, with individual tasks labeled T_1 to T_5). For the sake of simplicity, we assume that every single task requires one unit of CPU time for complete execution.

The optimal turnaround time considering segmented execution is T/M, that is, 5/3 time units, while a non-segmented execution would require $\lceil 5/3 \rceil$, i.e., 2 entire time units to complete. The segmented execution of the whole application is to be completed in five steps (one time unit per step), with every task being split in 3 segments. A possible scheduling of the execution is outlined in Figure 4.1, where each cell represents the task indicated along the y-axis during the step pointed by the x-axis. A gray cell indicates that the task is stopped for the given step. The pseudo-code for the algorithm used to compute this solution is listed in Algorithm 4.1. Note that other scheduling layouts exist, possibly more optimized. For instance, a segmented execution scheduling may minimize the interruption of tasks (an interruption occurs when after completing a non-terminal segment of the task, the execution is either stopped or moved to another machine). Indeed, relatively to the next segment of the task, two situations might occur: either the segment is processed on the same machine and thus a private checkpoint is enough for resuming the execution, or the next segment is scheduled to another machine. This latter case forces a checkpoint transfer, between the shared storage point and the newly scheduled machine.

Algorithm 4.1 Segmented execution algorithm.

```
{Init stage}
$tasks\_D \leftarrow [(T_1 : 0), ..., (T_i : 0), ..., (T_t : 0)]$
{Iterate step-by-step (the number of steps is equal to $T$)}
for every $step\, i\, in$ $\{i = 1, \ldots, T\}$ do
    for every $task_j\, in$ $\{j = 1, \ldots, T\}$ do
        if $(M - execstep(T_j) \geq (T - i + 1))$ then
            {$Task_j$ needs to be scheduled in this step $i$}
            $execstep(T_j) \leftarrow execstep(T_j) + 1$
        end if
        {Assigns a task for every vacant machine (if any)}
        if $NumVacantMachines > 0$ then
            for every vacant machine $Mach_k$ do
                {Lookup for an unassigned task}
                for every $Task_j\, in$ $\{j = 1, \ldots, T,\}$ do
                    if $Unassigned(Task_j)$ then
                        $Mach_K \leftarrow Task_J$
                    end if
                end for
            end for
        end if
    end for
end for
```

Therefore, a possible goal would be to find the schedule that minimizes the interruption of tasks, and for the unavoidable interruptions, to minimize the number of checkpoint transfers. For instance, considering the example depicted in Figure 4.1, two transfers of tasks occur: one for task T_2 (from M_2 to M_1 in step 4) and the second one for T_3 (from M_3 to M_1 in step 5).

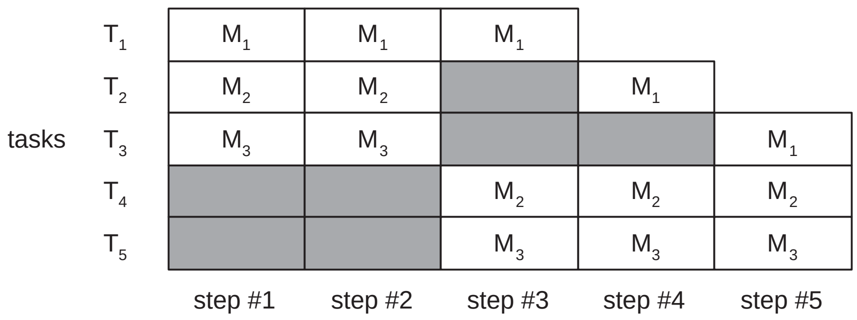


Figure 4.1: Optimal segmented execution of 5 tasks over 3 machines.

4.4 The DGSchedSim Simulator

To assess the performance of the scheduling policies presented in this chapter, we developed the simulator `DGSchedSim`. The main goal of the simulator is to provide accurate execution models of bag-of-tasks applications executed over a set of machines which is characterized by the individual performance of the machines. To support models close to real environments, the simulations are trace-driven by traces collected from real desktop grid systems. Our choice for trace-driven simulation is due to the fact that such simulations are credited as being more reliable than assessments based on analytical load distribution models [Dinda 1999]. In the context of non-dedicated resources, one of the key benefits of simulation over real testbed experiments is to allow reproducible and controlled experiments. In particular, real desktop grid systems are prone to various external uncontrollable factors such as the interactive load induced by users, variable network load and failures, just to name a few. These random external factors results in unpredictable fluctuations of resource availability, rendering difficult, if not impossible, to have repeatable execution conditions for successive experiments. Additionally, compared to real testbeds, simulated scenarios are much easier to setup and change since no real resources (hardware and human), besides the ones that effectively run the simulations, are actually needed. Indeed, simulation makes possible to study environments not available or possibly inexistent in the real world. For example, increasing the number of machines of a simulated scenario may be a simple question of editing a resource file, while to achieve the same effect in a real testbed requires much more effort, if at all doable.

4.4.1 Requirements

The requirements that guided the development of the simulator `DGSchedSim` were the following:

- Capacity to simulate the execution of bag-of-tasks applications in desktop grid environments;
- Ability to support the simulation of predefined and user-defined scheduling algorithms;

- Support for simulating heterogeneous resources with variable performance;
- Capability for recreating real load scenarios based on trace load collected from real environments;
- Ability to provide relevant information about the temporal evolution of a simulation so that results can be better understood, allowing the refinement of the simulated scheduling algorithms;
- Support for a predefined set of parameters that are relevant to scheduling in desktop grids, namely checkpoint policies and associated parameters.

4.4.2 Input

For carrying out a simulation, `DGSchedSim` requires four main items: (1) the application requirements, (2) the characteristics of desktop machines that represent the grid to simulate, (3) the load traces needed to drive the simulation, and (4) the user-defined scheduling algorithms.

The description of an application includes, besides the number of tasks, the computing requirements of an individual task. The computing needs of a task are expressed by the required CPU time, which is the number of dedicated CPU time units necessary for the complete execution of the task. This metric is given relatively to a reference machine. To extrapolate the required CPU time for a task, the computing capabilities of the involved machines need to be quantified. For this purpose, `DGSchedSim` resorts to the INTFP index, which corresponds to the arithmetic mean of the two numerical performance indexes, INT and FP, of the NBench benchmark [Mayer 2007], as previously introduced in Chapter 3. Specifically, to compute the CPU time needed for the execution of a task in a given machine, the simulator uses the ratio between the machine's INTFP index and the reference machine's index. For example, a ratio of 3 means that the machine is credited as being as three times faster than the reference machine, and thus the execution of a task will consume 1/3 of the CPU time that would have been needed if the execution was performed by the reference machine. Other characterizing elements of a task include the maximum required memory, the needed disk space, the input data size (size of data used as input) and the individual checkpoint size, in case checkpointing is enabled.

Every simulated machine is defined by a single entry in a so called *desktop grid configuration file*. An entry holds the machine name, its INTFP performance index, its static attributes like CPU model and speed, amount of main memory, disk space and the speed of the network interface. The entry also defines the thresholds for volunteer computing, namely the maximum main memory and maximum disk space available for running volunteer tasks. Note that tasks requiring resources above the thresholds of a machine cannot be run on the machine.

The traces are organized by time stamps: a time stamp corresponds to the chronological time, in UNIX's epoch format, when the data was collected. At every time stamp, the collected data aggregates various metrics for all the monitored machines, like uptime, CPU idleness, presence of interactive user, load of main memory, and so on. A trace is comprised of data captured at successive time stamps. For the purpose of a simulation, the time interval between consecutive time stamps may influence the accuracy of the results of a simulation. A wide interval between successive time stamps, even if it speeds up the time needed to execute the simulation, might worsen the precision of the simulation since events occurring between two time stamps can not be reproduced.

To perform a simulation with a given trace, it is necessary to specify the time stamp of the trace to be used as the starting point. However, different starting points might yield substantially different results, since different load patterns can be crossed by the simulation. For instance, a Friday afternoon's starting point will most certainly yield a much different execution pattern and turnaround time than a Monday morning's. This is especially relevant for short-lived applications. Therefore, to prevent results from being biased by the unpredictability of the starting point, `DGSchedSim` supports the possibility of multi-run, meaning that several simulations are executed from different starting time stamps with the final results corresponding to the average of all the executed simulations. Under the multi-run mode, the starting points may be user-selected (to allow for reproducible results) or randomly generated by the simulator. To drive a simulation, `DGSchedSim` can use the traces produced by WindowsDDC (see section 3.2, page 50).

`DGSchedSim` updates the status of the simulation at every time stamp. Based on the trace information of the time stamp being processed, the simulator updates the status of machines and tasks. If the machine assigned to a given task is still available, the simulator updates the execution progress percentage of the task accordingly to the idle CPU time of the executing machine, weighted by the computing capabilities relatively to the reference machine.

To support experimentation of scheduling policies, `DGSchedSim` supports the addition of user-defined scheduling algorithms. To add a scheduling algorithm the user only needs to implement a Python class[2] derived from the base class `dgs_sched`. Specifically, the class needs to override the method `DoSchedule()`, which gets called at every time stamp. This method receives as parameters the current time stamp and the list of non-executing tasks (i.e., tasks that have not yet been started or which are currently stopped). Through its base class, the method can also access the core data of the simulation like the list of machines and their associated statuses.

4.4.3 Output

Besides turnaround time, `DGSchedSim` can be set to produce other types of results. The goal behind the diversity of outputs is to allow a better comprehension of the outcome of the simulation.

For every simulation executed, several statistics are generated such as the turnaround times, the number of saved and loaded checkpoints, among other items. These statistics are saved in a file in CSV-like format allowing the use of generic tools like spreadsheets for further analysis and processing. If instructed to do so, `DGSchedSim` can also produce a file containing the evolution over time of the counts of tasks completed, stopped and under execution. This information can be plotted to survey the temporal evolution of the execution and the consequent behavior of the scheduling policy. Furthermore, to enable a better understanding of the execution, important for perceiving strategies to minimize turnaround time, `DGSchedSim` can also produce a set of images depicting the physical distribution of tasks over machines as well as their states of execution. Since one image is generated per time stamp, the whole set can be combined to compose an animation displaying the application execution driven by the traces of the desktop grid. An example of `DGSchedSim`'s output is shown on Figure 4.2. Specifically, the image represents the activity at time stamp 1101380656 of the

[2]The simulator was written in Python.

32 machines defined for the shown simulation. As indicated by the filled boxes, five of the machines were processing tasks.

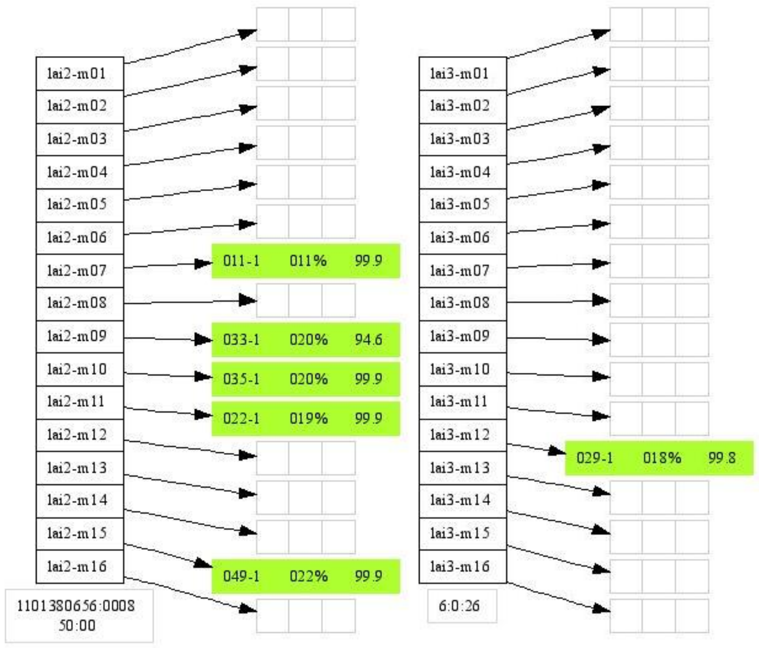


Figure 4.2: Example of a `DGSchedSim`'s graphical output.

As an aside feature, `DGSchedSim` can also compute the cluster equivalence ratio (CER) yielded by a given load trace. The simulator applies the CER definition (section 3.5.7, page 78), computing the CPU availability of a machine for a given period correspondingly to its measured CPU idleness over the analyzed period of time. For instance, a machine with 70% CPU idleness is accounted as a 0.70 dedicated machine. This methodology assumes that all idle CPU can be harvested, and thus the obtained results should be regarded as an upper limit of CPU resources that can effectively be exploited.

Finally, to speed up simulations, `DGSchedSim` can itself create a bunch of tasks suitable for execution under Condor. This feature was heavily used to simulate the herein proposed scheduling policies and allowed us to gather an inner knowledge of scavenging desktop grid cycles from the perspective of the user.

4.5 Summary

In this chapter, we presented the case for the importance of turnaround time for bag-of-tasks executed over desktop grids. We also defined a set of scheduling policies targeted for improving turnaround times of the classical FCFS scheduling methodology. The proposed scheduling policies exploit the concept of shared checkpoints, upon which, application-level checkpoints can be shared among workers, providing mobility to tasks. Finally, we summarily presented the `DGSchedSim` simulator, which was used for simulating the aforementioned scheduling policies. In the next chapter, we analyze the simulation results for the scheduling policies that were proposed in this chapter.

5 Evaluation of Fault-Tolerant Scheduling

In this chapter, we present the main results for the scheduling policies described in the previous chapter. We first describe the simulated computing infrastructure, namely the set of machines and the desktop grid trace which was used to drive the simulations. We then present and analyze the main results.

5.1 Computing Environment

5.1.1 Machines

To assess the behavior of the scheduling policies relatively to the speed of machines and of the heterogeneity of resources, two machine sets, henceforth identified as $M01$ and $M04$, were simulated. The $M01$ set holds 32 identical fast machines of type D (see Table 5.1), while the $M04$ set, as the name suggests, is a mix of four different groups of machines: 8 machines of type A, 8 of type B, 8 of type C, and 8 of type D. The $M01$ group delivers a higher computational power than $M04$, since only 8 machines of $M04$ are as fastest as the machines of the $M01$ set. Concretely, and based on the reference ratios of the machines that constitute the sets, $M01$ is equivalent to 54.14 reference machines, while $M04$ yields 36.66 reference machines, meaning that $M01$ is approximately 1.5 times faster than $M04$.

The four types of simulated machines are summarized in Table 5.1. The column *CPU* describes the machine's CPU type and clock speed, while the next column, *Perf. index*, indicates the combined INTFP performance index

Type	CPU	Perf. index	Ratio to reference
A	PIII@650 MHz	12.952	0.518
B	PIII@1.1 GHz	21.533	0.861
C	P4@2.4 GHz	37.791	1.511
D	P4@3.0 GHz	42.320	1.692
Avg.	–	28.649	1.146
Reference	P4@1.6GHz	25.008	1.000

Table 5.1: Sets of simulated machines and their performances.

of the respective machine group. Finally, the fourth column corresponds to the reference performance ratio of machines relatively to the reference machine. The reference machine, shown in the last row of Table 5.1, is used as reference for calibrating the execution of tasks. For instance, a task requiring 1800 seconds of CPU on the reference machine, would take nearly 3475 seconds on a type *A* machine, and slightly more than 1063 seconds on a type *D* machine.

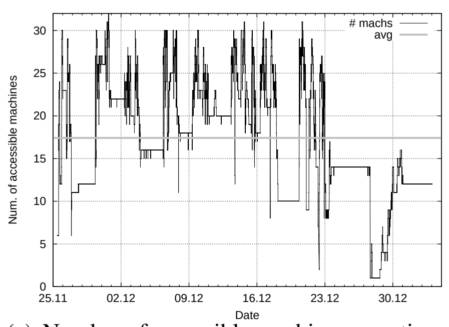
5.2 Trace

Due to the difficulty of conducting repeatable and configurable experiments in real desktop grid platforms, we resorted to simulation to carry out performance analysis of the proposed algorithms. All simulations were driven by a trace collected from two academic classrooms of 16 machines each, totaling 32 machines. The trace is comprised of samples collected, through WindowsDDC, at a two-minute cadence at the available machines. Each sample aggregates several metrics, like uptime, CPU idleness and memory load to name just the metrics that are relevant to the simulations. All the machines run the Windows 2000 (service pack 3) operating system. As reported in Chapter 3, no shutdown policy exists for the machines: when leaving a machine, a user is advised but not obligated to shut it down. Therefore, machines may remain powered on for several days. On weekdays, the classrooms remain open 20 hours per day, closing from 4 am to 8 am. On Saturdays, the classrooms open at 8 am and close at 9 pm, remaining closed until the following Monday.

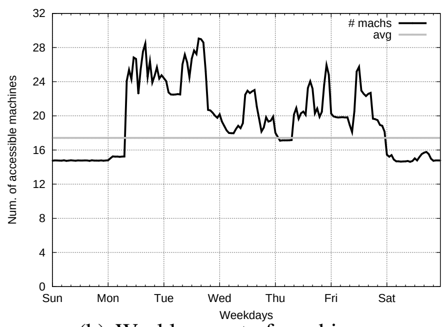
5.2.1 Characterization of the Trace

The trace represents 39 consecutive days (from the 25th November of 2004 to the 3rd January of 2005) and contains 27,193 sampling periods, for a total of 473,556 samples. The samples were collected during a period with classes, except for the last ten days which corresponded to the Christmas holidays. When no classes were taught, the machines were used by the students for their practical assignments, web browsing and email activities.

Figure 5.1 plots the number of accessible machines for the trace (Figure 5.1(a)), as well as the weekly distribution of accessible machines (Figure 5.1(b)). The average number of accessible machines for the trace was 17.42 (shown by the horizontal line *avg*) with a standard deviation of 6.78.



(a) Number of accessible machines over time

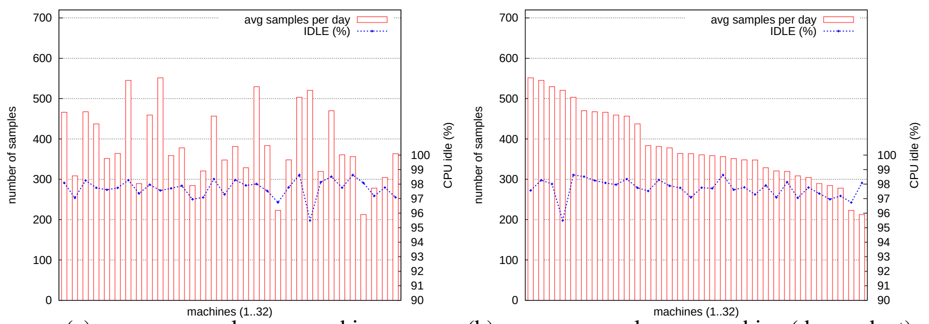


(b) Weekly count of machines

Figure 5.1: Count of accessible machines.

The flat portions of the plot displaying the count of accessible machines indicate periods of resource stability. These periods correspond mostly to weekends and also, although in a lesser degree, to nighttimes, as can be seen on the weekly distribution. This behavior is explained by the fact that when the classrooms are closed, like on weekends and during nighttimes, the machines maintain the state (powered on/powered off) that they had at classroom's close time. Indeed, since only power users can remotely access the resources, the physical inaccessibility of the machines means that their state remains unaltered apart an unexpected and rare event occurs, like a crash or a power outage.

The average daily count of samples collected per machine is shown in the plots of Figure 5.2. The left *y*-axis is scaled to the maximum number of samples per machine in a day (720 samples, corresponding to a sample every 2 minutes). The machines (x-axis) are displayed by their identifying number (1 to 32) in the left plot, and reversely sorted by the average count of samples in the right plot. The number of samples of a machine measure the availability of the machine, since only an accessible machine (powered on and with network connectivity) can be sampled. The plot indicates that machine availability varies roughly between 550 samples per day (corresponding to 76.39% availability) for the most available and 220 (30.55% availability) for the least available. The plot also displays the average CPU idleness percentage per machine (right y-axis), with values ranging from slightly less than 96% to 99%. Once again, this confirms the high CPU idleness of machines.

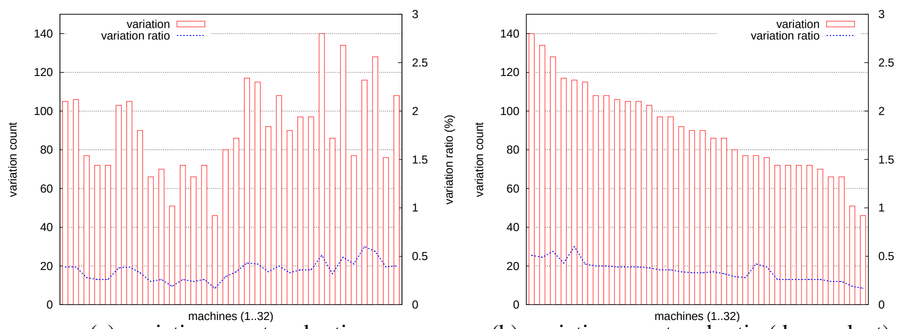


(a) average samples per machine (b) average samples per machine (descendant)

Figure 5.2: Samples per machine per day (left y-axis) and CPU idleness per machine (right y-axis).

To assess the volatility of the trace, we computed the *variation count* of every machine of the trace. A variation is defined as the change of state in a machine's availability, either from available to unavailable (for instance, the machine was powered off), or vice-versa (the machine was booted, or regained connectivity to the network). The variation count of a trace is then a set which holds, for every machine of the trace, the machines' variation count. Furthermore, to facilitate comparison, we defined the *variation ratio* of a machine as the ratio between its actual variation count and the potential maximum number of variations, which corresponds to the number of samples minus one (that is, 719).

Figure 5.3 combines the variation count per machine (plotted against the left y-axis), and the variation ratio (right y-axis) for the trace class. (Plot 5.3(b) displays the machines reversely sorted by their respective variation count.) It is important to note that we use the broad definition of availability, upon which a machine is considered available as long as it is powered on, independently of the existence or not of an interactive user. This definition follows what can be configured for some desktop grid workers like BOINC or even Condor (by default, Condor does not allow the coexistence of interactive usage and volunteer computation, but this can easily be changed). The main metrics of the trace are summarized in Table 5.2 (page 108).



(a) variation count and ratio (b) variation count and ratio (descendant)

Figure 5.3: Variation count per machine (left Y-axis) and the corresponding variation ratio (right Y-axis).

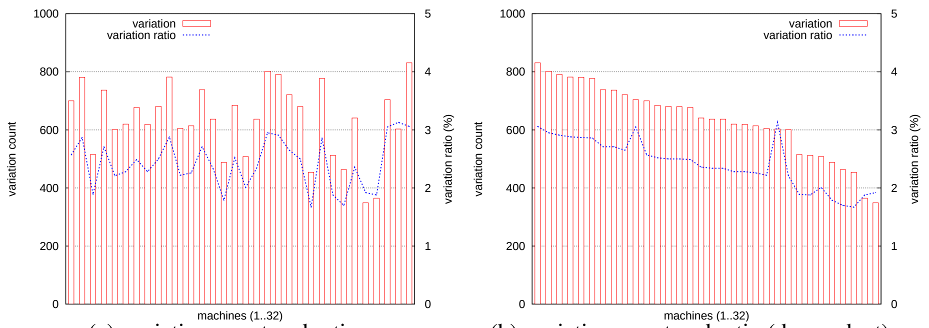
Although a set of 32 machines may appear as somewhat limited and not representative of existing institutional settings, it should be noted that we consider that a pool of machines to harvest is split among several users, with the set of 32 machines representing the machines assigned to a given user for a limited period of time, for instance, one day. Furthermore, much of the criticality of scheduling oriented for fast turnaround time occurs in the last stage, when the number of resources is bigger than the number of tasks to be completed. Therefore, under such circumstances, for assessing scheduling policies, the ratio *number of tasks / number of machines* is more significant than the total number of machines.

5.2.2 CPU Idle Threshold

As can be observed in Table 5.2, the machines have high percentages of CPU idleness, indicating the high availability of the studied computing environment. Regarding variability, as measured by the *variation ratio per ma-*

Metric	***weekday***	***weekend***	**Total**
Trace length (days)	27	12	39
No of samples	353618	119938	473556
Avg. count of machines	18.9 ($\sigma = 7.4$)	16.1 ($\sigma = 3.4$)	17.4 ($\sigma = 6.8$)
Avg. CPU idleness	95.7% ($\sigma = 7.7\%$)	98.7% ($\sigma = 2.7\%$)	96.5% ($\sigma = 7.2\%$)

Table 5.2: Main metrics of trace *class*

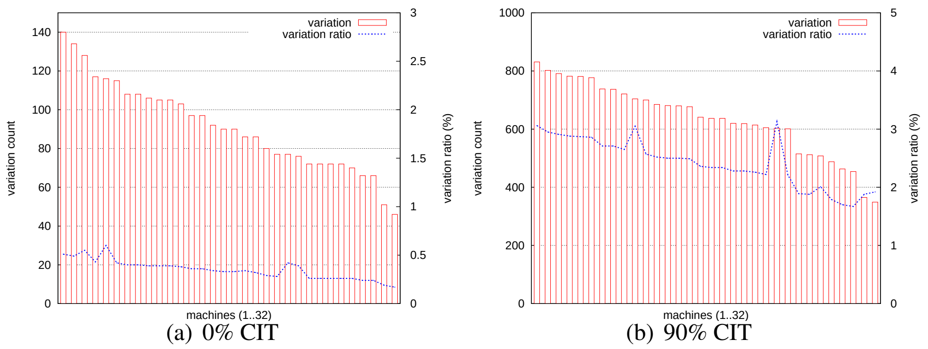


(a) variation count and ratio (b) variation count and ratio (descendant)

Figure 5.4: Variation count per machine (left y-axis) and the corresponding variation ratio (right y-axis) for a 90% CIT.

chine, the trace has very low values (0.35%, with $\sigma = 0.10\%$), meaning that fault tolerance mechanisms by themselves might not yield major improvement on performance, since the stability of resources is high. Therefore, for the purpose of evaluating the proposed scheduling policies under a more stringent environment, namely with a higher level of volatility, we defined a threshold condition regarding the minimum level of idle CPU required for a machine to be targeted for scavenging at a given period. Specifically, under the *CPU Idle Threshold* condition (CIT), a machine is defined as candidate to scavenging at a time t, only if its CPU idleness is above the defined threshold.

In practical terms, the CIT condition is equivalent to a resource owner defining that the machine should only be considered for opportunistic computing when CPU idleness is above a predefined threshold. In this study, besides the 0% CIT, which corresponds to the original trace, simulations were also carried out for a 90% CIT condition. From the point of view of the resources, this corresponds to a much more demanding scenario, where



(a) 0% CIT (b) 90% CIT

Figure 5.5: Comparison of 0% CIT and 90% CIT variation count (left y-axis) and variation ratio (right y-axis).

only periods that have less than 10% of CPU consumed by the resource owner can be harvested by hosted tasks.

Figure 5.4 plots the variation count and ratio for the trace when the CIT is set to 90%. As expected, the variability of the resources is much more pronounced than for the 0% CIT case. The mean variation count is 707.78 (with $\sigma = 447.7$), and the average variation ratio is 2.69 ($\sigma = 1.62$). Thus, relatively to the 0% CIT, the resources measured under the 90% CIT are nearly 8 times more volatile. To allow for a better comparison of the effect of a 90% CIT relatively to a 0% CIT, the variation count and ratio plots of both CITs are plotted next to each other in Figure 5.5, page 109 (note that the plots have different y-axis scales).

5.3 Main Results

In this section, we present the most relevant results for the proposed scheduling policies. Before examining the main results for the studied cases, we present the concept of *Ideal Execution Time*, and then we describe the application scenarios that were simulated.

5.3.1 Ideal Execution Time

For a better assessment of the results, the obtained turnaround times are given relatively to the *Ideal Execution Time* (IET). Specifically, the reported results correspond to the *slowdown ratio*, that is, the ratio of the application

Number of tasks	**Task (seconds)**	**turnaround (minutes)**	
		M01	M04
25	1800	17.73 (17.73)	35.46
25	7200	70.91 (70.91)	141.83
75	1800	53.19 (41.49)	70.91
75	7200	212.74 (165.93)	283.66

Table 5.3: *Ideal Execution Time (IET)* for the considered task/machine pairs. For the $M01$ set, the *segmented ideal execution time* is shown within parenthesis.

turnaround time relatively to the IET for the given characteristics of the application (number of tasks and the amount of reference CPU time required per task), and the machine set (either $M01$ or $M04$). The IET measures the theoretical turnaround time that would be required for the execution of the application if performed under ideal conditions, that is, with fully dedicated and failure free machines, and with no overhead whatsoever. The fully dedicated requirement means that all CPU (and other resources) are devoted to the application. Thus, IET is independent of the usage trace of the resources, being determined by the characteristics of the application (number of tasks and individual size of the tasks) and the performance of the executing machines. Table 5.3 reports the IETs for the scenarios analyzed in this study, that is, the $M01$ and $M04$ machine sets, and applications comprised of either 25 or 75 tasks of 1800 or 7200 seconds each. The table also includes the execution time considering an ideal segmented execution based on shared checkpoints as computed by Algorithm 4.1 (page 96). The values are shown within parenthesis and solely for the $M01$ machine set, since mapping tasks to heterogeneous machines is, as reported before, NP-hard [Pineau *et al.* 2005]. Note that for the 25-task cases, the ideal segmented execution time concurs with IET. This is due to the fact that the number of tasks is less than the number of machines.

5.3.2 Simulated Tasks

Simulations were carried out with applications comprised of either 25 or 75 tasks, with individual tasks requiring either 1800 or 7200 seconds of CPU time when executed on the reference machine. The number of tasks per application was chosen in order to assess a scenario where the number of tasks would be slightly smaller than the number of machines (25 tasks), and another one, where the number of tasks would be moderately higher than the number of machines (75 tasks).

The impact of the checkpoint policies over the turnaround time was measured by varying the number of saved checkpoints during the execution of a task. For this purpose, simulations were performed with no checkpoints, one checkpoint and nine-checkpoint per single-task execution. In single-checkpointed executions, the checkpoint file is saved when the task reached half of its execution, while for the nine checkpoints executions a checkpointing operation is performed every 10% of the execution (i.e., at 10%, 20%, 30% and so forth). The size of the individual checkpoints was set to 1 MB for all simulated scenarios. In fact, bigger sizes (up to 10 MB) were also simulated and they did not yield significant differences, mostly due to the fast communication links of the targeted LAN environments, and thus, the network data movement is not a determinant fact in terms of overhead. Note that network costs (latency and the transfer time) and disk costs (read and write operations) were considered for modeling the checkpointing operations. Specifically, latency was set to 1 millisecond, and the network speed considered to be 92 Mb/s as measured by the IPerf utility [Iperf 2007]. Regarding disk operations, read and write speeds were experimentally determined at the reference machine to be, respectively, 31.263 MB/s and 28.081 MB/s. These values are conservative, since the reference machine was a laptop, which usually have relatively slow I/O subsystems, at least when compared with desktop machines.

For the scheduling policies that rely on replication, that is, TR, TR-DMD and TR-PRDCT-DMD, the replication count threshold (Section 4.2.4, page 92) was set to 3, since this proved to be the value that delivered the best results for the considered setting.

Our simulated model assumes that the presence of a local user, although holding priority over resources, would not suspend the execution of the running task in that machine. Instead, the quality of service for local users is assumed to be preserved via the process priority mechanism of the host's operating system, with guest processes running at the lowest priority level. As stated before, this behavior is similar to the one that can be configured in the BOINC client, as well as in other desktop grid middleware like Condor and XtremWeb.

To assess the effects of the weekday/weekend variations of the trace over the workloads – as seen earlier on, weekends present much lower volatility of resources – separated simulations were carried out for weekdays and weekends. Furthermore, to prevent results from being biased by using only specific portions of the trace, all simulations were run multiple times from different starting points, with the reported turnaround times corresponding to the mean average of the multiple executions. Specifically, the number of runs for every simulated case was 12 for weekday periods, and 10 for weekend ones.

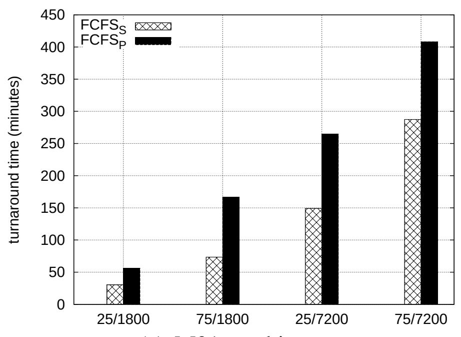
5.4 Presentation of Results

The presentation of results is split in two parts. First, to assess the effect of sharing checkpoints, the turnaround time from the shared version of a scheduling policy is compared to the corresponding private version for each one of the four *number of tasks/individual task length* scenarios (i.e., 25/1800, 25/7200, 75/1800, 75/7200). Note that the shared versions of scheduling policies have their name terminated with an *_S*, while private versions are identified with an ending *_P* in their names.

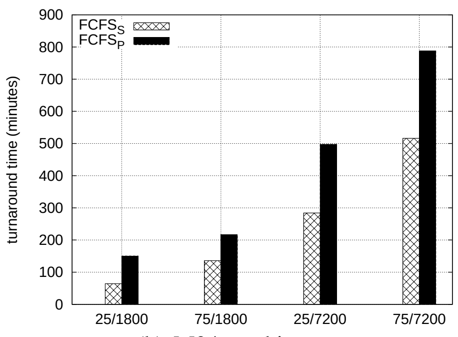
In the second part, and since scheduling based on shared checkpoints consistently outperforms private-based schemes, we compare the shared checkpointing versions of all the proposed scheduling policies. Specifically, every plot aggregates the slowdown ratios for the studied scheduling methodologies, that is, adaptive timeout (AT), first-come-first-served (FCFS), transfer replicate (TR), transfer replicate with checkpoint on demand (TR-DMD), and transfer replicate with prediction and checkpoint on demand (TR-PRDCT-DMD). An important note is that the plots display slowdown ratios, with lower values meaning faster, and thus better, turnaround times. To assess the effects of volatility over the scheduling policies, results are shown for 0% and 90% CIT.

5.4.1 Shared versus Private Checkpointing

0% CIT over weekdays. Figure 5.6 (page 113) displays the turnaround times, in minutes, for the shared and private checkpoint versions of the FCFS scheduling policy, when simulated over weekday periods, considering a single-checkpoint per task and CIT sets to 0%. Specifically, Figure 5.6(a) reports the results for the M01 machine set, while Figure 5.6(b) presents the turnaround times for the M04 machine set. Independently of the scenario, the shared version markedly outperforms the private version. This occurs for almost all the other scheduling policies as shown in Figure 5.7 (AT, single-checkpoint per task, page 114), Figure 5.8 (TR, single-checkpoint per task, page 114), Figure 5.9 (TR-DMD, with nine checkpoints per task, page 115) and Figure 5.10 (TR-PRDCT-DMD, with nine checkpoints per task, page 115). All plots correspond to executions performed over weekday periods considering a 0% CIT.



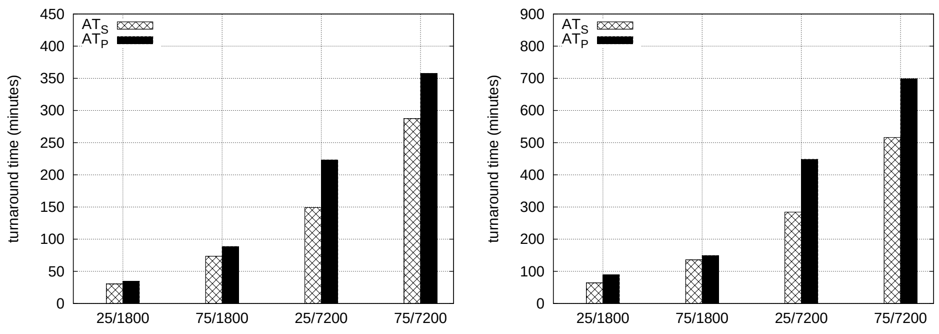
(a) M01 machine set



(b) M04 machine set

Figure 5.6: Turnaround execution time for the shared and private FCFS (one checkpoint per execution) with 0% CIT over weekdays.

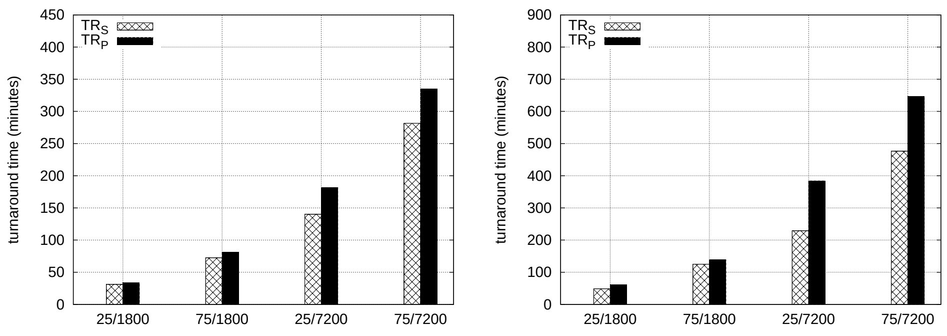
The superior performance delivered by the shared-based policies is mostly due to the rapid detection of interrupted executions, allowing for a fast rescheduling of the interrupted tasks on other machines. On the contrary, private schemes only detect an interruption when the associated timeout expires, thus losing precious time between the failure detection and the consequent restart operation on another machine. Indeed, the importance of timeouts is demonstrated by the fact that the adaptive timeout policy (AT) greatly improves the performance of the private versions of FCFS but without bringing any benefit to the shared checkpoint methodologies (see Figure 5.6 and Figure 5.7). Another observation that reinforces the early failure detection as an important factor for achieving faster turnaround time is that execution times are not significantly influenced by the checkpoint frequency, since checkpointless executions perform closely to one- and nine-checkpoint executions, at least when considering a 0% CIT.



(a) M01 machine set (b) M04 machine set

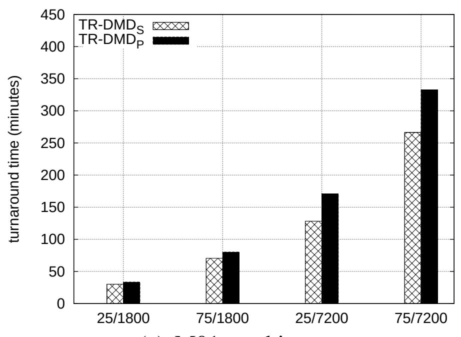
Figure 5.7: Turnaround execution times for shared and private AT (one checkpoint per execution) with 0% CIT over weekdays.

For a same scenario, and independently of the scheduling policy, the M01 machine set yields faster turnaround times than the ones achieved through the M04 machine set (note that the plots for representing the M01′s turnaround times have different *y*-axis scales than the ones showing the M04′s). The superior performance of the M01 machine set relatively to M04 is due to the fact that the M01 set, as seen in section 5.1.1 (page 103) aggregates approximately 1.5 times more computing power than M04.

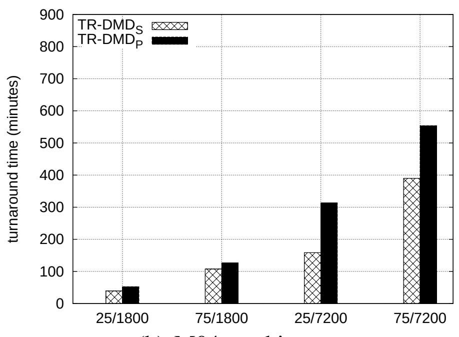


(a) M01 machine set (b) M04 machine set

Figure 5.8: Turnaround execution times for shared and private TR (one checkpoint per execution) with 0% CIT over weekdays.

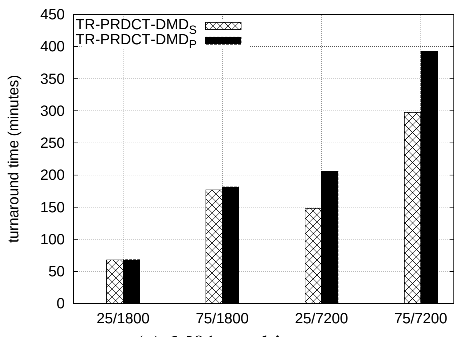


(a) M01 machine set

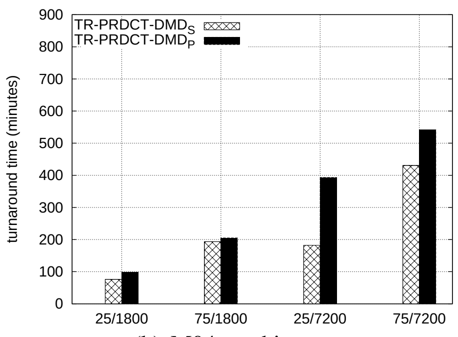


(b) M04 machine set

Figure 5.9: Turnaround execution times for shared and private TR-DMD (nine checkpoints per execution) with 0% CIT over weekdays.



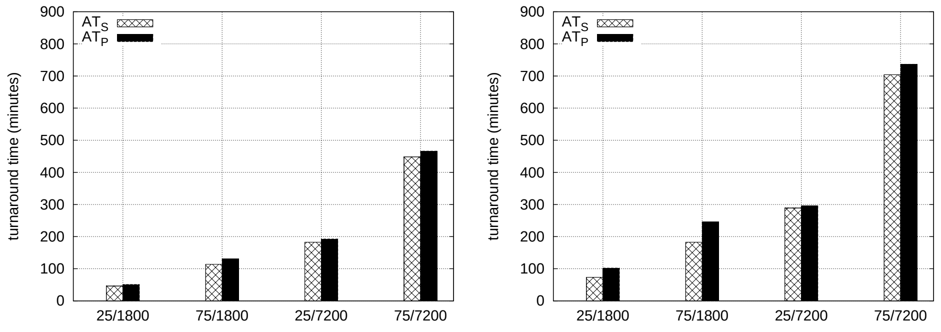
(a) M01 machine set



(b) M04 machine set

Figure 5.10: Turnaround execution times for shared and private TR-PRDCT-DMD (nine checkpoints per execution) with 0% CIT over weekdays.

0% CIT over weekends. Figure 5.11 (page 116) plots the turnaround times for the AT scheduling methodology when executed over weekend periods. As it clearly emerges from the plots, the private-based methodologies perform similarly to shared ones. This is a consequence of the high stability of resources on weekends, caused by the fact that classrooms are locked from Saturday 9 pm to the following Monday 8 am, with users having no physical access to the machines. Thus, every machine maintains the state it has at 9 pm on Saturday until the end of this no access period. Since practically no failure occurs over weekends, the early detection advantage of the shared approach yields no benefit.

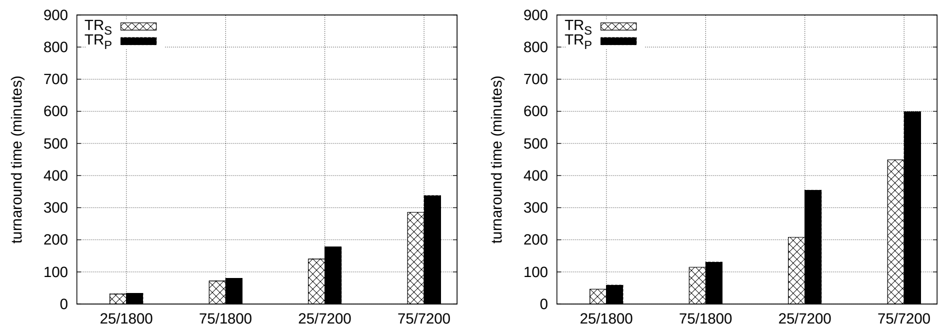


(a) M01 machine set (b) M04 machine set

Figure 5.11: Turnaround execution times for shared and private AT (one checkpoint per execution) periods with CIT=0% over weekends.

Comparatively to the equivalent AT executions over weekdays (see Figure 5.7), executions over weekends produce lengthier turnaround times. This is due to the number of accessible machines, which diminishes on weekends relatively to weekday periods (see Figure 5.1(b), page 105), thus impacting the whole available computing power.

Although over weekends, private and shared schemes perform very similarly, for the simple TR policy (Figure 5.12, page 117), shared schemes provide significant benefits relatively to private-based methodologies. This is especially true for applications with 75 tasks executed over the M04 machine set. Furthermore, as replication is only activated in the last stage of the application execution, that is, when all tasks have already been completed or at least been assigned to a machine, the performance improvements achieved by simple replication are due to replicas being scheduled to faster machines. This means that the task location yielded by FCFS (and the FCFS-based policies, like AT) is far from optimal and can be improved by simply reassigning tasks to faster machines. Therefore, this suggests that TR is appropriate for applications executed over stable heterogeneous resources.

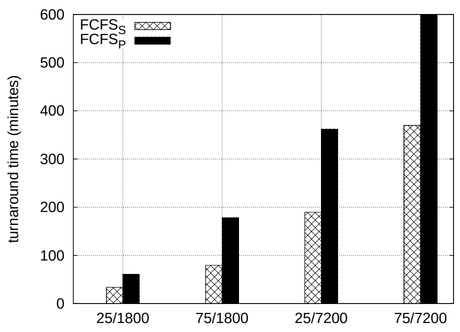


(a) M01 machine set (b) M04 machine set

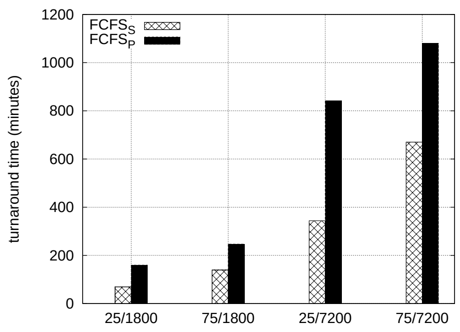
Figure 5.12: Turnaround execution times for the shared and private TR (one checkpoints per execution) periods with CIT=0% over weekends.

90% CIT over weekdays. Figure 5.13 compares the private and shared checkpointing turnaround times under the FCFS policy executed over weekdays, with CIT sets to 90%. The pattern of results is similar to what was observed for the 0% CIT, although, as expected, with lengthier turnaround times than the ones obtained for a 0% CIT (see Figure 5.6, page 113). This is due to the fact that under a 90% CIT the opportunities for exploiting idle resources diminish, since the 90% CPU idleness threshold must be met for a task to be scheduled. An interesting issue is that the degradation of performance relatively to the 0% CIT is minor for the shared-based policies, confirming that shared checkpointing overcomes, at least partially, the effects of volatility induced by a 90% CIT requirement. A similar trend occurs for the TR, TR-DMD and TR-PRDCT-DMD policies, shown on Figure 5.14 (page 118), Figure 5.15 (page 119) and Figure 5.16 (page 119), respectively.

90% CIT over weekends. The turnaround times for weekends under a 90% CIT closely follow the ones observed for a 0% CIT, with no major differences existing between the shared and private versions, a consequence of

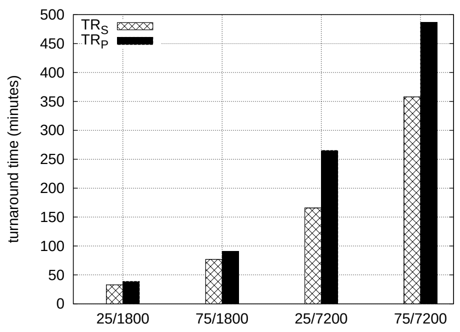


(a) M01 machine set

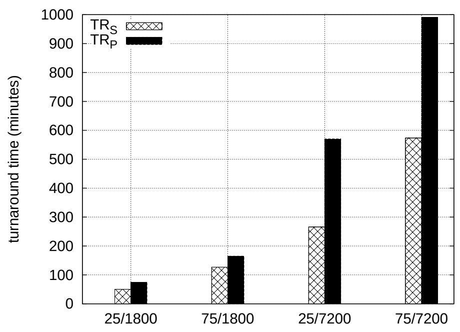


(b) M04 machine set

Figure 5.13: Turnaround execution times for shared and private FCFS (one checkpoint per execution) with a 90% CIT over weekdays.

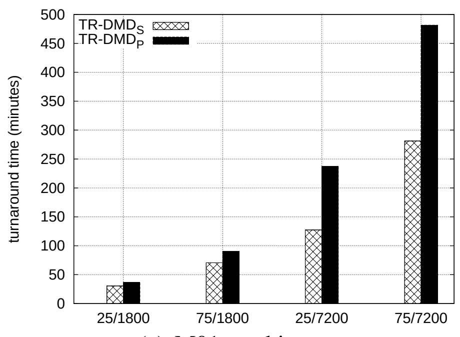


(a) M01 machine set

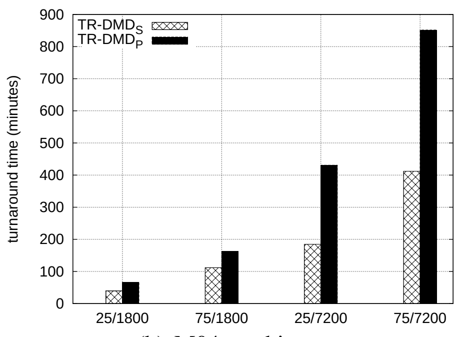


(b) M04 machine set

Figure 5.14: Turnaround execution times for the shared and private TR (one checkpoint per execution) with a 90% CIT over weekdays.

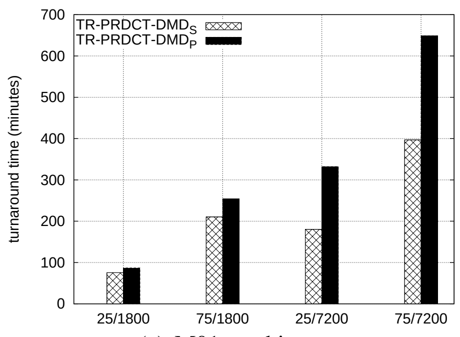


(a) M01 machine set

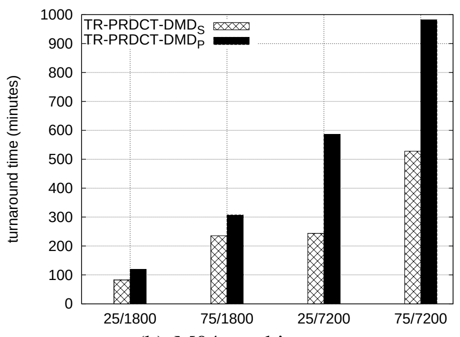


(b) M04 machine set

Figure 5.15: Turnaround times for the shared and private TR-DMD (nine checkpoints per execution) with a 90% CIT over weekdays.



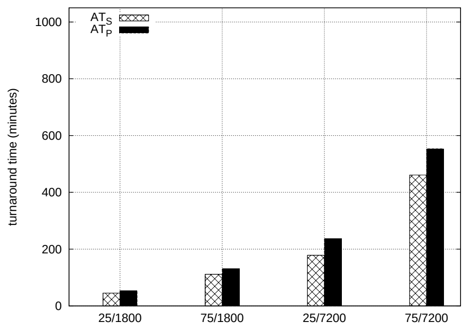
(a) M01 machine set



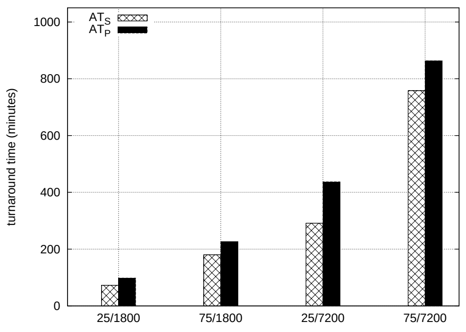
(b) M04 machine set

Figure 5.16: Turnaround execution times for shared and private TR-PRDCT-DMD (nine checkpoints per execution) with a 90% CIT over weekdays.

the relative stability of resources over weekends. Moreover, the only effect of a 90% CIT is to lengthen the turnaround times, independently of the used checkpointing type. Plots for the AT and the TR cases are shown in Figure 5.17 and in Figure 5.18, respectively.

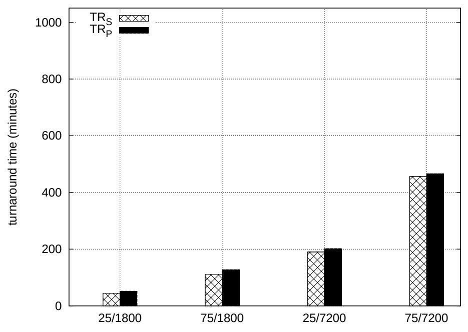


(a) M01 machine set

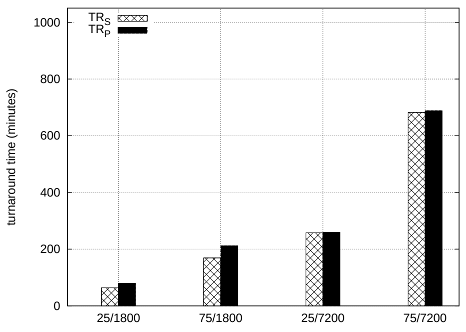


(b) M04 machine set

Figure 5.17: Turnaround execution times for shared and private AT (one checkpoint per execution) with a 90% CIT over weekends.



(a) M01 machine set



(b) M04 machine set

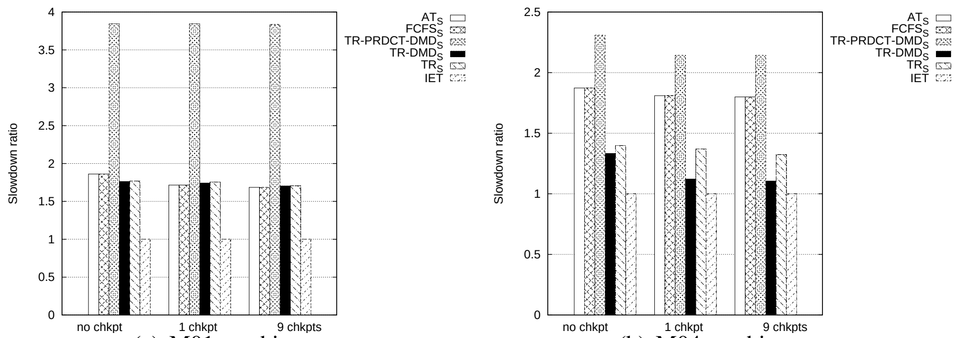
Figure 5.18: Turnaround execution times for shared and private TR (one checkpoint per execution) with a 90% CIT over weekends.

5.4.2 Shared-based Policies

We now analyze the slowdown ratios yielded by the shared-based scheduling policies when executed over the trace. We first detail the behavior of the policies over weekdays for a 0% CIT, and then for weekends. We then follow the same approach for the 90% CIT case.

0% CIT over weekdays. Figure 5.19 aggregates the slowdown ratio plots of the machine sets M01 (left) and M04 (right) for the execution of 25 tasks of 1800-second CPU time on weekdays. For the M01 set, all scheduling methodologies perform on a same level, apart from the prediction-based policy which produced the worse slowdown ratios. As expected in a homogeneous set of machines, replication yields no benefits. Regarding the weak performance of the prediction-based scheduling, we hypothesize that the prediction methodology, while trying to replicate tasks executing on machine predicted as unavailable on the next scheduling round, might occupy other machines that would otherwise run more useful replications (from the point of view of turnaround time). As we should see later on, the prediction-based method yields more positive results for larger tasks.

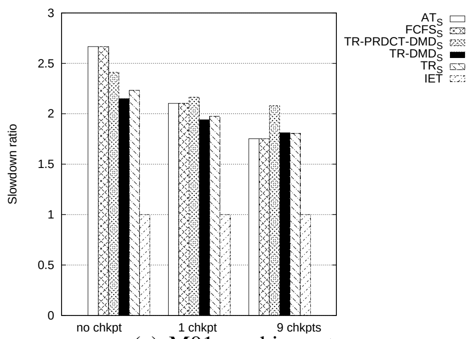
For the M04 machine set, the replication-based policies delivered the best turnaround times, with the checkpointing on demand mechanism yielding benefits relatively to the simple replication policy. The positive results of the replication-based scheduling can be explained by the heterogeneity of the M04 set, which creates opportunities for replications, namely when a task is replicated to a faster machine, something that obviously can not happen under M01.



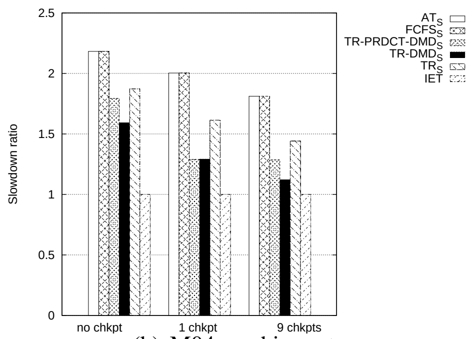
(a) M01 machine set (b) M04 machine set

Figure 5.19: Slowdown ratios for 25/1800 tasks on weekdays (0% CIT).

For the 25/7200 case (Figure 5.20, page 122), the prediction-based policy performs practically on pair with the simple replication policy that still delivers the best turnaround time on the M01 set. For the M04 set, replication-based policies coupled with checkpointing on demand achieve the fastest turnaround times. In fact, the prediction-based scheduling generates much better results for 7200-second tasks than it does for 1800-second ones, confirming that this policy is better suited for longer tasks, especially in heterogeneous machine environments like M04. Once again, while checkpoint benefits on turnaround times are seen with low checkpoint frequency (i.e., one checkpoint), increasing the checkpoint frequency to nine checkpoints per execution only marginally improves performance. This a consequence of the low variability of resources when a 0% CIT is considered.



(a) M01 machine set



(b) M04 machine set

Figure 5.20: Slowdown ratios for 25/7200 tasks on weekdays (0% CIT).

The relative shape of the 75/1800 plots (Figure 5.21, page 123) are somewhat similar to the 25/1800 case (Figure 5.19, page 121) indicating that the number of tasks does not seem to influence the behavior of the scheduling policies. In this case, the replication-based policy with checkpointing on demand is consistently the fastest policy regardless of the machine set. Again, results demonstrate that the prediction-based policy is not well suited for short tasks.

Slowdown ratios for the 75/7200 case, shown in Figure 5.22(b) (page 123), further confirm our previous indication regarding the appropriateness of the prediction-based policy to longer tasks in heterogeneous environments. In fact, independently of the checkpoint frequency, the prediction-based policy outperforms the other scheduling methods for the M04 machine set. For the homogeneous set M01, shown in Figure 5.22(a), both basic task replication (TR) and replication with checkpoint on demand (TR-DMD)

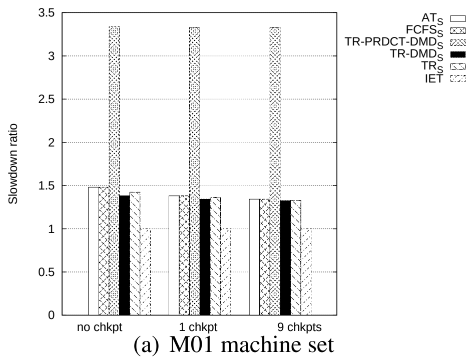
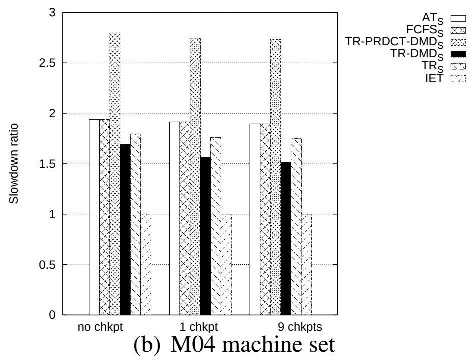


Figure 5.21: Slowdown ratios for 75/1800 tasks on weekdays (0% CIT).

yield the best results, although without much benefit relatively to the basic FCFS and AT. As stated before, this can be explained by the limited role of replication in a homogeneous set of machines, which is only meaningful when a task is interrupted.

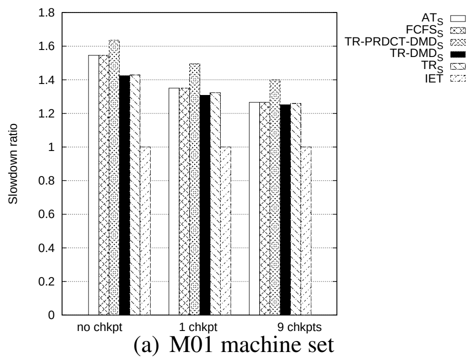
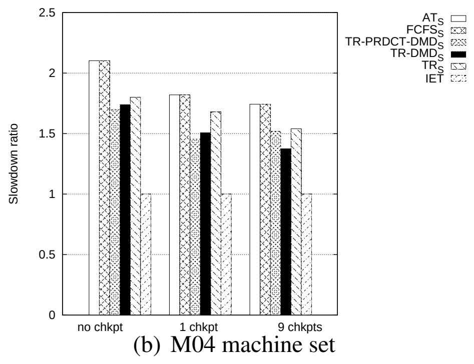
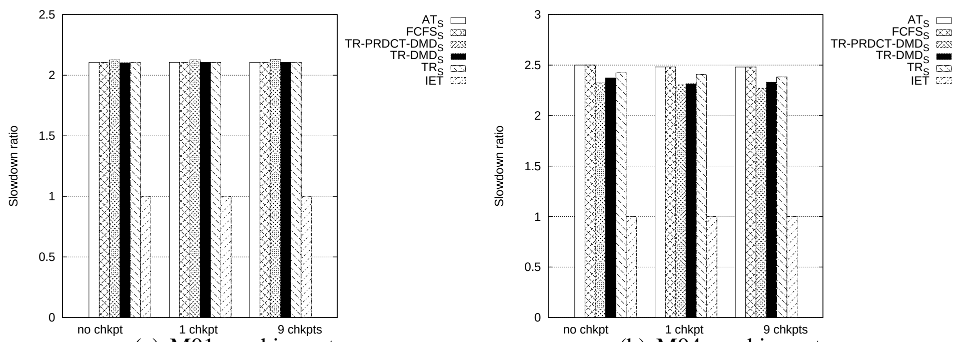


Figure 5.22: Slowdown ratios for 75/7200 tasks on weekdays (0% CIT).

0% CIT over weekends. For weekend executions, only the 75/7200 case is shown (Figure 5.23) since the results for all the other cases follow a similar trend. With the M01 homogeneous set of machines, all scheduling policies perform roughly at the same level. The same occurs for the M04 set of machines, although replication-based policies such as TR and TR-DMD yield barely perceptible faster turnaround times.

As previously observed (see Figure 5.11, page 116), and contrary to what occurs on the weekday period, the shared checkpoint versions present no real advantage over the private checkpoint ones, especially with the homogeneous machine set (M01). This is due to the higher stability of resources on weekends.

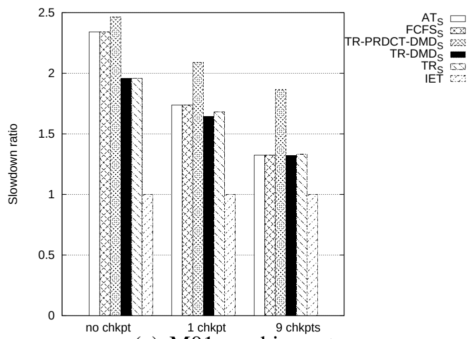


(a) M01 machine set (b) M04 machine set

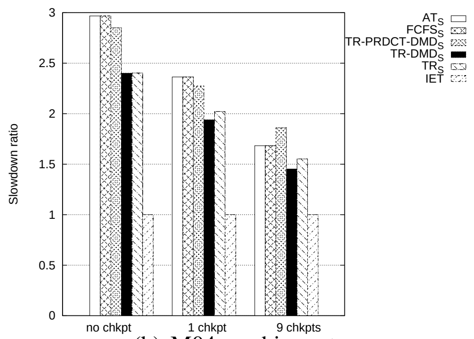
Figure 5.23: Slowdown ratios for 75/7200 tasks on weekends (0% CIT).

90% CIT over weekdays. Results regarding 90% CIT present similar shapes that the ones observed for a 0% CIT. This is noticeable on the comparison of Figure 5.24 (page 125), which presents the slowdown ratio for the 75/7200 case for a 90% CIT, and Figure 5.22 (page 123), which depicts the same case for a 0% CIT. As expected, slowdown ratios are bigger for the 90% CIT. This is a consequence of the higher volatility induced by the higher CIT. Furthermore, the comparison of the plots highlights that higher checkpoint frequency are effective in masquerading the negative impact of volatility on performance, proving the usefulness of checkpoints in volatile environments. For instance, in the depicted 75/7200 case, the 9-checkpoint executions are much faster than the checkpointless ones.

90% CIT over weekends. Setting the CPU idleness threshold to 90% bears minimal impact on resources volatility on weekend periods, and thus it is practically unnoticeable on the results. This can be observed from Figure 5.25 which displays the slowdown ratio for the 75/7200 case executed for a 90% CIT over weekends. All policies perform closely, except for the



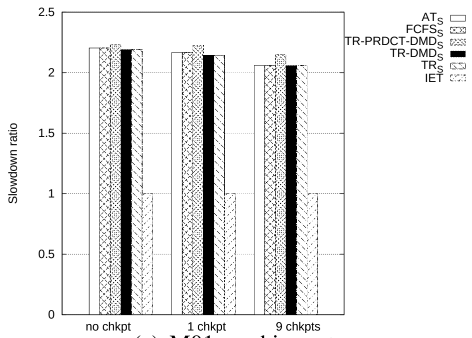
(a) M01 machine set



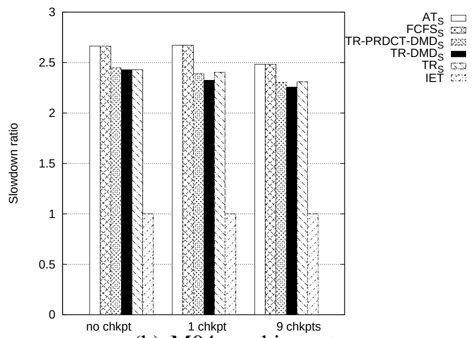
(b) M04 machine set

Figure 5.24: Slowdown ratios for 75/7200 tasks on weekdays (90% CIT).

replica-based ones which yield minor benefits for the heterogeneous set of machines M04.



(a) M01 machine set



(b) M04 machine set

Figure 5.25: Slowdown ratios for 75/7200 tasks on weekends (90% CIT).

5.5 Related Work

In this section, we review the main work related to scheduling for fast turnaround time over desktop grids.

Scheduling methodologies for reducing turnaround times of task-parallel applications in desktop grids were thoroughly studied by Kondo et al. [Kondo *et al.* 2004b]. The authors analyzed several strategies such as resource exclusion, resource prioritization, and task duplication. The study also resorted to trace based simulations, using traces collected from the Entropia desktop grid environment [Kondo *et al.* 2004a]. However, the study targeted only small sized-tasks, with the length of tasks being 5, 15 and 35 minutes of CPU time. Moreover, the work did not consider migration nor checkpointing, assuming that interrupted tasks would be restarted from scratch. Although perfectly acceptable for small tasks, this assumption is not appropriate for longer tasks, especially in volatile environments.

The OurGrid project implements the workqueue-with-replication scheduling policy (WQR) [Cirne *et al.* 2006]. Under this scheme, tasks are assigned in a FCFS-like manner, regardless of the metrics related with the performance of machines. When all tasks have been distributed to workers, and if there are enough free resources, the system creates replicas from randomly chosen tasks. WQR acts as a best-effort scheduler, with no guarantee of executing all tasks. This differs from our work, which is based on the assumption that an application must be completely executed, thus requiring that the scheduler enforces the execution of all tasks. Cirne et al. conclude that task replication significantly augments the probability of an application being terminated.

Anglano and Canonico [Anglano & Canonico 2005] propose WQR-FT that extends the basic WQR methodology with replication and checkpointing. They recommend the usage of checkpointing for environments with unknown availability, since it yields significant improvements when volatility of resources is high. Likewise WQR, WQR-FT is also limited by its best effort approach, with no guarantee that all tasks comprising a given application are effectively executed.

Weng and Lu [Weng & Lu 2005] study the scheduling of bag-of-tasks with associated input data over grid environments (LAN and WAN) of heterogeneous machines. They propose the Qsufferage algorithm that considers the influence of the location of input data repositories on scheduling. The study confirms that the size of the input data of tasks has an impact in the performance of the heuristic-based algorithms.

Zhou and Lo [Zhou & Lo 2005] propose the Wave scheduler for running tasks in volunteer peer-to-peer systems. Wave scheduler uses time zones to organize peers, so that tasks are preferentially run during workers' nighttime periods. At the end of a nighttime period, unfinished tasks are migrated to a new nighttime zone. In this way, tasks ride a wave of idle cycles around the world, which reduces the turnaround time. However, the exploitation of moving night zones is only feasible in a wide-scale system, distributed all over the globe.

Taufer et al. [Taufer *et al.* 2005b] define a scheduling methodology based on availability and reliability of workers. Specifically, workers are classified based on their availability and reliability. The scheduler deploys the tasks based on this classification, assigning tasks to workers accordingly to the priority of tasks and to the reliability and availability of workers. The proposed scheduling policy targets BOINC-based projects, taking advantage of the fact that this middleware already collects enough information to classify individual workers from the availability and reliability point of view.

5.6 Summary and Discussion

As demonstrated by the results presented in this chapter, sharing checkpoints appears as an effective strategy for lowering the turnaround time of bag-of-tasks applications executed over an institutional desktop grid.

The main results of this chapter are:

- Improvements of up to 60% were obtained relatively to the private checkpoint methodology. This way, the usage of shared checkpoints should be promoted for users with soft real time deadlines who resort to institutional desktop grid resources to run their applications.

- An interesting result is that turnaround time for short duration tasks[1] are only marginally dependent on checkpoint frequency. This suggests that it is viable to run checkpointless applications over institutional desktop grids, provided that the tasks are short. This confirms the results of Kondo et al. [Kondo *et al.* 2004b].

- For highly volatile scenarios, such as the ones yielded by a 90% CIT, sharing checkpoints become mandatory if the goal is to have fast turnaround times.

- Our results also point out that replication-based policies can be effective for speeding up turnaround times, especially in heterogeneous

[1]Less than 1-hour of CPU time, considering the average machine of the available resources.

environments. Indeed, for settings comprised of homogeneous machines, a replica can only act as a backup, and thus no major gain should be expected.

- Regarding replication, care should be taken in setting an appropriate replication factor count, in order to allow that a maximum of uncompleted tasks can be replicated to faster resources, especially during the critical last stage of the bag-of-tasks execution, which frequently determines the turnaround time. For the studied environment, three was the replication factor that yielded the best results. Additionally, the effectiveness of replications improves with checkpointing on demand, although implementing such a scheme might break the traditional reverse-client model implemented by most desktop grid middleware.

- Although the prediction-based techniques seem inappropriate for short tasks, they yield substantial benefits with long-running tasks.

- An important issue regarding scheduling for fast turnaround time relates to the day of the week. Indeed, the considered resources exhibit the traditional weekday/weekend use pattern, meaning that on weekends the resources are highly stable, and thus, a simple replication scheme such as TR is enough. Although the studied resources exhibited a somewhat short nighttime period, such periods are also exploitable. This means that scheduling strategies should adapt accordingly to the *time of the day/day of the week* pair. For instance, applications comprised of lengthy tasks (up to two days), and which do not resort to checkpointing should preferentially be run over weekends to profit from the stability of the resources.

- We believe that the results herein presented can be extrapolated to other institutional environments, since most of them should exhibit similar patterns to those found in our environment.

In the next chapter, we focus on sharing checkpoints over wide-scale desktop grids.

6 Sharing Checkpoints over Wide-Scale Desktop Grids

In this chapter, we extend the concept of shared checkpoints to wide-scale desktop grid environments like the Internet. Specifically, we target desktop computing projects with applications that are comprised of long-running independent tasks, executed on thousands of computers on the Internet.

6.1 Introduction

We present the *chkpt2chkpt* system that resorts to a distributed hash table-based (DHT) infrastructure for sharing checkpoint among the workers of a desktop grid project. The main idea is to organize the worker nodes of a desktop grid into a peer-to-peer (P2P) DHT. Worker nodes can take advantage of this P2P network to track, share, manage and claim the space of the checkpoint files. Moreover, the system introduces the concept of *guardian* to allow for a finer grain control over the execution of tasks, speeding up the detection of interrupted tasks, and consequently, the recovery process.

We use simulation to validate our system and we show that remotely storing replicas of checkpoints can considerably reduce the turnaround time of the tasks, when compared to the traditional approaches where nodes manage their own checkpoints locally. These results make us conclude that the application of P2P techniques seems to be quite helpful for managing and sharing checkpoints in wide-scale desktop grid environments.

6.2 Motivation

Given the high volatility of desktop grid resources, platforms like BOINC and XtremWeb resort to application-level checkpointing. However, as seen previously in Chapter 4, this approach presents a clear limitation, because all the checkpoint files of a node are only stored at the local nodes, and thus are private. If this node fails, the local checkpoint files will not be available, and thereby they turn out to be useless. Martin et al. [Martin *et al.* 2005] reported that the wide-scale public computing project *climateprediction.net* in which they are involved, would greatly improve its efficiency with the existence of a mechanism to support the sharing of checkpoint files among worker nodes allowing the recovery of tasks in different machines. It should be noted that the *climateprediction.net*'s individual tasks (*workunits*) are computationally very demanding (up to three months of computation on an average machine), and thus a significant percentage of workers drop the project before completing their first workunit. For wide-scale volunteer computing projects, the alternative, proposed in Chapter 4, of storing checkpoints in the central supervisor is not feasible, since the central supervisor would become a clear bottleneck.

In this context, we propose to use a P2P infrastructure for sharing checkpoint files that should be tightly integrated with the desktop grid environment. Under this approach, the worker nodes act as peers of a P2P Chord [Stoica *et al.* 2001] distributed hash table, which they use to track the checkpoint files. If the replicated checkpoint files are available in the P2P overlay, recovering from a failed task can be much more effective than the private checkpointing model which is used in the traditional desktop grid middleware.

It is important to note that the P2P overlay network should be regarded as a complement to the desktop grid infrastructure: it does not replace the centralized approach followed by all major volunteer based desktop computing projects. It is a fact that the central model has its limitations, namely being a single point of failure and a bottleneck. However, it allows for a tight control over the progress of the computation and has been used successfully in all major projects. For these reasons, we keep untouched some of the paradigms that have made the centralized approach a successful option.

Although there are many recent examples of peer-to-peer file-sharing and backup applications, e.g. [Cox *et al.* 2002; Annapureddy *et al.* 2005; Sit *et al.* 2003], just to mention a few non-commercial systems, sharing checkpoints requires a different type of solution. To start, checkpoints become garbage in a deterministic way. For instance, as soon as some task is finished (or definitively canceled), all of its checkpoints should be discarded. Another difference concerns the usefulness of the checkpoints. While the motivation for volunteering storage space for storing music or other multimedia files in one's disk is obvious, the same is not true with checkpoints. Thus, a system that replicates checkpoints needs to explicitly reward users that concede their space. On the other hand, in our solution, we can take advantage of the strict control that we have on the creation and placement of checkpoints. This sort of reasons make us think that creating a checkpoint replication system goes beyond a simple adaptation of existing file-sharing solutions. Moreover, we add a fine-grain control, allowing for early discovery and consequent recovery of interrupted tasks. In this way, we promote faster turnaround times.

In this chapter, we propose and validate through simulation a desktop computing architecture called *chkpt2chkpt*, which couples the traditional central supervisor based approach with a P2P distributed hash table that enables checkpoint sharing. The purpose of *chkpt2chkpt* is to reduce the turnaround time of bag-of-tasks applications executed over wide-area volunteer environments. By reusing the checkpoints of interrupted tasks, nodes need not to recompute those tasks from the beginning. This considerably reduces the average turnaround time of tasks in any realistic scenario where nodes often departs with their tasks unfinished. Moreover, by promoting the utilization of previous computations (by way of task's states restored from checkpoints), our replication system also increases the throughput of the entire desktop grid system.

6.3 Overview

Our system is built upon the traditional model of public computing projects, with a central supervisor coordinating the global execution of an application. Specifically, the central supervisor replies to a volunteer worker node (henceforth *worker*) request for work by assigning it a processing task. The worker then computes the assigned task, sending back to the central supervisor the results when it has completed the task. The targeted applications are bag-of-tasks, each one comprised of a large number of independent tasks, with an application only terminating when all of its tasks are completed. Every task T is uniquely identified by a number, with the application A being represented by the set A_T such as $A_T = \{T_1, \ldots, T_i, \ldots\}$. Furthermore, we only consider sequential tasks that can be individually broken into multiple temporal segments $\{S_{t_1}, \ldots, S_{t_i}, \ldots, S_{t_n}\}$ and whose intermediate computational states can be saved in a checkpoint when a transition between temporal segments occurs. Whenever a task is interrupted, its execution can be resumed from the last stable checkpoint, either by the same node (if it recovers) or by some other worker. Our main goal is to promote the availability of checkpoints to increase the recoverability of the interrupted tasks, thereby improving the turnaround time of the applications.

Workers self-organize to form a DHT[1], which they use to maintain the distributed checkpointing scheme and to keep track of the execution of tasks, in such a way that requires a minimal intervention of the central supervisor.

Checkpoints are identified by a sequential number starting at 1. Note, that for the purpose of our system, not all checkpoints taken by a worker node are exported to the DHT, with some checkpoints kept private. For example, a task might be checkpointed every 5 minutes to local storage, but to avoid some overhead and network traffic, only one out of six checkpoints is exported to the DHT, meaning that, from the point of view of the global system, the checkpoint period is 30 minutes. To simplify, in the reminder of this chapter, we use the designation *checkpoint* to refer to checkpoints effectively exported to the DHT.

To describe the basic idea of the proposed system, we first expose the simple case of a single worker executing a task from the start to end (see Figure 6.1, page 133). In this case, interaction occurs as follows:

1. The worker requests the central supervisor for a computing task, receiving a task ready to be processed.

2. The worker registers the task in the DHT, by selecting a regular peer-worker of the DHT to store a tuple called "worker-task info". This

[1]It is not strictly necessary that all the nodes participate in the DHT, but only that they can access and write data on the DHT. To simplify description, we assume that all nodes belong to the DHT.

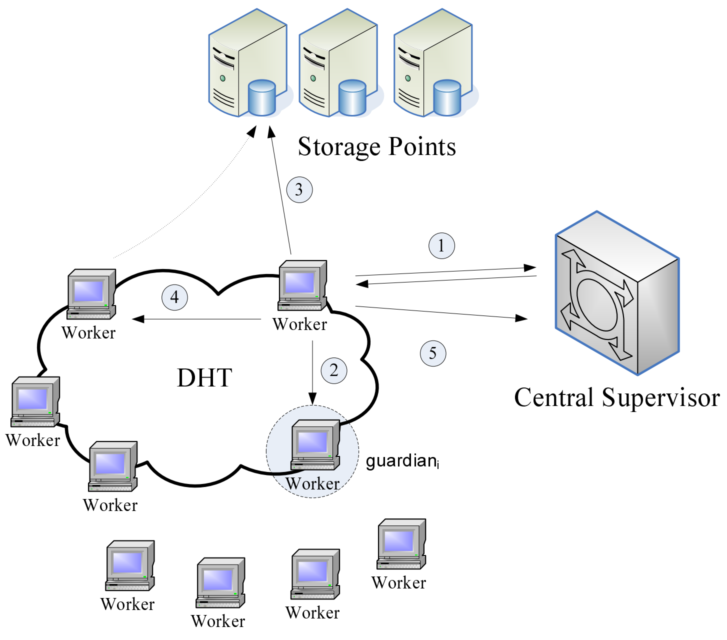


Figure 6.1: A summary of *chkpt2chkpt*'s components and interactions.

tuple keeps information about the task (we describe it in Section 6.4). We call this peer-worker the "*guardian* of *i*" and represent it as $guardian_i$.

3. Each time the worker has to perform a checkpointing operation, it writes the checkpoint in its own disk and replicates it in some storage point. The storage point is selected accordingly to a given metric, like for instance, the network distance.

4. The worker uses the DHT to store a pointer to that storage point. This pointer is accessible by any other node of the DHT, using as key the pair formed by the task identifier and the checkpoint number.

5. Finally, when the worker node has completed the whole task, it sends back the results to the central supervisor.

If a worker fails two things may happen: (1) if the worker recovers after a short period of time, it resumes its previous task execution from the last checkpoint file maintained in its local disk. However, (2) if the worker fails for a period longer than a specified timeout, then the central supervisor may redistribute that task to some other worker node. In this case, the new worker performs step 2 as before, trying to register the checkpoint in the

DHT, but it may get in response from $guardian_i$ the number of the checkpoint where the task was left, an indication that at least one checkpoint from a previous execution attempt exists. In this case, the worker tries to fetch the checkpoint to resume the task. However, the worker may not get any reply with the number of the checkpoint, if the previous $guardian_i$ has meanwhile departed the network. In this scenario, the worker starts looking for the best checkpoint, that is, the one with highest number, using a procedure that we describe in Section 6.4.5 (page 140). After having found such checkpoint, the worker proceeds as explained before.

In our failure model, we assume that the central supervisor is protected by some replication mechanism and is always available despite the occurrence of some transient failures – this corresponds to what happens with some major public computing projects like, for examples, *SETI@home* and *Einstein@home*. On the contrary, we assume that worker nodes may fail frequently under a failure model that is either *crash-stop* or *crash-recovery*. Nodes that fail and then recover but lose their previous task can be seen as new nodes. Due to the high volatility of volunteer resources [Godfrey *et al.* 2006], nodes are very prone to fail, and thus can deprive, at least temporarily, the DHT from some state information and checkpoints. Thus, we consider as a typical case, the possibility of a node not finding some information that it is looking for in the DHT. We inherit from existing solutions, which are current practice in volunteer desktop grid systems, that use replication of computation to overcome some of these failures.

An important issue regarding *chkpt2chkpt*, as well as many peer-to-peer file sharing systems, is the garbage collection of shared files that are no longer useful. This is relevant with checkpoint files, whose usefulness expires as soon as the task is completed or abandoned[2]. Thus, it is necessary to remove the no longer needed checkpoints, as well as the related management information (*metadata*) for these checkpoints. We propose two approaches to erase the useless files, that we identify as *pull* and *push*. Under the *pull* approach, the storage nodes query the $guardian_i$ for task i before deleting the stored information. Conversely, in the push mode, the worker node that finishes a task sends messages to remove all the administrative information related to the task. This is further detailed in Section 6.5.

[2]A task may be abandoned if for whatsoever reasons it fails to complete. For instance, the task may systematically crash due to a software bug of the task's code.

Parameter	**Definition**
P_i	Worker node processing task i
task i	Task processed by worker node p_i
key $i:j$	Key for j^{th} checkpoint of task i
hash(i:j)	Hash of key $i:j$
$guardian_i$	Timeout watchdog for worker p_i while processing task i
WTI_i	"worker-task info of i" (tuple kept by $guardian_i$)
t_p	Timeout to detect an abandoned task (watchdog: *guardian*)
t_t	Timeout to detect an abandoned task, with $t_t >> t_p$ (watchdog: *supervisor*)

Table 6.1: Parameter definitions.

6.4 Description of *chkpt2chkpt*

6.4.1 Basic Components

We assume that task i is identified by i and that n sequential checkpoints are produced along the execution of the task. We identify the node that is working on task i as p_i, the j-th checkpoint of task i with the key $i:j$, and the hash of this key as $hash(i:j)$, where $hash()$ is the hash function of the DHT. The DHT mechanism ensures that there is a single node responsible for holding key $i:j$, typically a node whose identifier is close to $hash(i:j)$ (to simplify, we refer to this as the node $hash(i:j)$, although the real identifier of the node will usually be close but different). Any node in the DHT can reach the node $hash(i:j)$ in a deterministic way.

One of the principles that guided the design of the *chkpt2chkpt* system was to keep low the burden on the central supervisor, as well as to respect the traditional worker-initiated communication model. Hence, the checkpointing service must be supported, as a best effort, by the nodes of the DHT. In particular, *chkpt2chkpt* strongly relies on $guardian_i$, on the storage points, and on the indirection pointers to ensure proper operation of task i. The node $guardian_i$ serves to indicate that worker p_i is processing the segment that leads to checkpoint j of task i. Nodes can determine the location of $guardian_i$ by computing the hash of key $i:0$, that is, $hash(i:0)$. The *guardian* of i stores a tuple that we call "worker-task info of i", represented as WTI_i. The format of this tuple is (p_i, j, t_p), where p_i is the processor node of task i, j is the checkpoint being computed, and t_p is a timeout information used to detect workers that abandon a task. Specifically, when the timeout t_p expires, $guardian_i$ sends a message to the central supervisor announcing a possible failure of p_i. This allows the *chkpt2chkpt* system to maintain timeouts with a much finer granularity than it is usually possible with minimal effort from the central supervisor. The advantage is that the system can recover from failures of worker nodes in a much faster way, since failures are detected much quicker than what would have been possible by relying solely on the per-task global timeout.

Ideally, we would like to maintain the invariant INV in *chkpt2chkpt*, which we define as follows: WTI_i exists in the node $guardian_i$ if and only if task i is being processed. From this, it follows that if there is no WTI_i in the node $guardian_i$, the nodes can assume that the task is already finished (or yet to start). For performance reasons, we allow this invariant to be violated once in a while, as node $guardian_i$ can be down and task i can still be active.

At this point, we can use a concrete example to better illustrate the interaction with the DHT: consider that node Worker$_A$ wants to fetch the last checkpoint available of task 43 (assume that it is in fact available). (1) It issues a $get(43:0)$ operation. (2) Assume that $43:0$ hashes to 578. The DHT will forward the request to the owner of key 578, which may be node 581. (3) Node 581 will reply with the number of the requested checkpoint, e.g., 3. (4) Now, to get checkpoint #3, the requesting node issues a $get(43:3)$ operation that hashes to, for instance, (5) to node 1411 and that node 1411 is accessible. (6) Node 1411 will then reply with the address (either an IP address or a machine name) of the storage node SN_k that it is known to hold checkpoint #3 of task 43. Finally, (7) Worker$_A$ contacts SN_k so (8) it can download the requested checkpoint. A graphical illustration of the described example is given in Figure 6.2 (page 137).

6.4.2 Processing a Task

While processing a task i, a worker node p_i needs to perform two operations at $guardian_i$: increase the number of the checkpoint when it has completed the previous one and send periodic *heartbeat* message to let $guardian_i$ know that it is alive, before t_p expires. As long as this timeout does not expire (or the timeout of the central supervisor, as we explain in Section 6.4.3), only processor p_i can change values in the WTI_i tuple (this is one of the

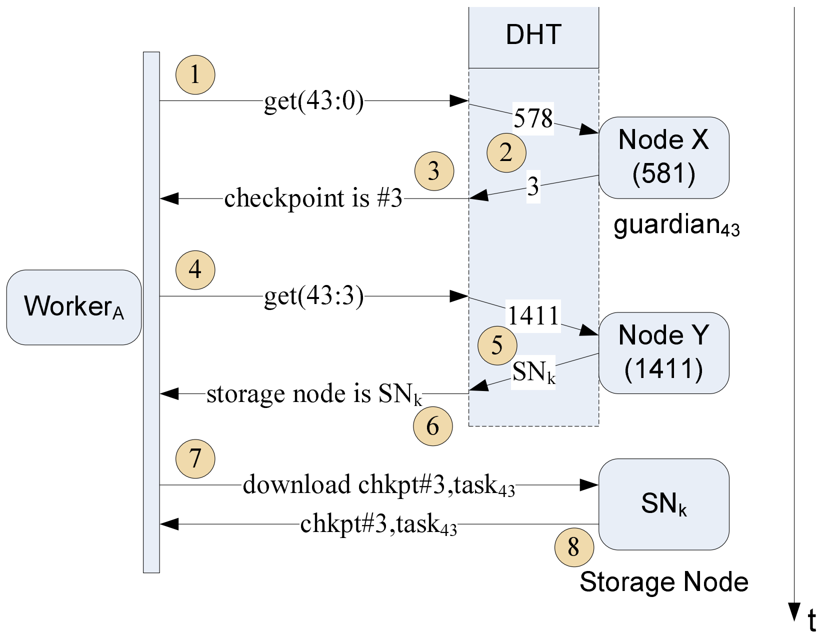


Figure 6.2: Locating and retrieving a checkpoint under *chkpt2chkpt* (example).

small twists that we do to the normal operation of the DHT). However, it is possible that two workers try to process the same task i simultaneously: this occurs when $guardian_i$ or the central supervisor wrongly consider the first worker as failed and the central supervisor reassigns the task to the other. If this occurs, it is up to node $guardian_i$ to decide which node owns the task. It is either the first node that tries to take control of the task, if t_p is expired, or it is the initial worker, if t_p is not expired. One interesting way of providing the periodical heartbeat is to slightly modify the normal $put()$ operation of the DHT, such that rewriting the WTI_i tuple serves as a heartbeat. In this case, this operation will also serve without any modification as a watchdog for $guardian_i$, because it periodically rewrites the value j of the current checkpoint in the $guardian_i$ (possibly a new one if the previous $guardian_i$ failed).

6.4.3 Starting and Resuming a Task

The central supervisor assigns three types of tasks: (1) tasks that are being delivered for the first time and tasks whose previous execution attempts have exceeded the task timeout, either (2) t_p (given by $guardian_i$) or (3) t_t

(given by the central supervisor). To improve the turnaround time, we need to set $t_p << t_t$ so that a failure can be early detected, and dealt with. We aim to minimize the number of messages and the number of bytes exchanged between node p_i and the central supervisor, as this latter one is a potential communication bottleneck. If the central supervisor is delivering task i for the first time, it can immediately send all the data to the worker and the worker can start to process the data as shown in Figure 6.1 (page 133).

However, it may happen that a worker departs the network without delivering its task. In this case, upon expiration of the timeout t_p, $guardian_i$ sends a message informing the central supervisor of the likely interruption of task i, which puts task i in the redistribution list[3]. In the redistribution of a task, the central supervisor only sends the identifier of the task, say i, and flags the repetition to the newly assigned worker p_i, which must check at $guardian_i$ whether some other node is still processing this task, to avoid concurrent processing of the same data. If worker p_i finds that the previous owner of task i is still holding the task (because it came back to life, for instance), it gives up the task and tries to get a new one from the central supervisor.

Finally, we have the case where the central supervisor redistributes a task for which t_t has expired. Here, the new worker, say p_i, will have the task regardless of the situation of the previous worker. To this end, p_i must instruct $guardian_i$ to check the new owner of the task in the central supervisor. When a task is redistributed by the central supervisor (regardless of the timer, t_p or t_t, that has expired), the new worker node always tries to fetch the last available checkpoint, as we describe in Section 6.4.5.

6.4.4 Separation of Processing and Storage

To ensure the availability of checkpoints, the worker node processing a task should store replicas of its checkpoints in other nodes. This way, we keep the sequential checkpoints of a running task available in case the worker that holds the task fails and its task gets redistributed. Given this constraint, we consider the following two additional assumptions to build our checkpoint replication system:

[3]To avoid concurrent processing of the same task, $guardian_i$ sends another message to the central supervisor if a timed out worker ever recovers (due, for instance, to a machine with transient network access that gets reconnected to the network).

Assumption 1: nodes offering processing time may not have space available for storage of checkpoints. The system should explicitly manage a separation between processing and storage nodes;

Assumption 2: it is not feasible to maintain a constant number of replicas of each checkpoint in the DHT. When nodes enter and leave the DHT, they cannot transfer big checkpoints from one node to another as the DHT changes, because nodes may be distant from each other and this would clog the network. We base this assumption on the work of Blake and Rodrigues, which states that only two features out of *high availability*, *scalable storage*, and *dynamic peer networks* can be obtained [Blake & Rodrigues 2003].

To cope with Assumption 1, *chkpt2chkpt* extends the contribution model of traditional public-computing projects. Besides donating CPU cycles, nodes can contribute to the P2P infrastructure with storage space and bandwidth. The system separates CPU donation from storage space and bandwidth volunteering. In fact, a node can provide CPU cycles (executing tasks), or volunteer storage and bandwidth (integrating the DHT or being a storage point), or donate resources to both causes. To foster motivation for donors to volunteer space storage and bandwidth for the P2P infrastructure, a rewarding credit mechanism and associated ranking system, similar to the one employed to recompense CPU donation in public-computing projects, can be devised [Anderson 2004]. Under this scheme, a resource donor receives credits for the space storage effectively devoted to shared checkpoints. Furthermore, to foster motivation for checkpoint storage donors, an added bonus can be provided whenever a locally-stored checkpoint is used to resume a task in another machine.

To cope with Assumption 2, we do as we explained before: the DHT does not hold the checkpoints, but only pointers to the checkpoints. The purpose of this indirection is to avoid unneeded network traffic, because checkpoints can be very large, like in the case of the *climateprediction.net* project where each checkpoint file has about 20 MB [Martin *et al.* 2005]. To access checkpoints, nodes use the standard $get()$ functionality of the DHT. For instance, to access checkpoint j of task i, a node needs to issue a $get(i:j)$. Since there is yet another level of separation, this $get()$ operation returns an indirection pointer to the storage, instead of the storage itself.

6.4.5 Managing the Checkpoints

To retrieve checkpoint j of task i from a storage point, the interested nodes get the value of the key $i : j$, which is a pointer holding all the needed information to reach the checkpoint. However, the checkpoint may be unreachable (for example, the machine holding it is down or simply disconnected from the network). In this case, the node starts a procedure with a logarithmic number of steps to find the highest number available checkpoint.

The worker node successively divides the space of keys $i : 1$, $i : 2$, $i : \ldots$, $i : n$ (assuming that n is the number of checkpoints) in approximately equal parts. First, it looks for checkpoint $\lceil n/2 \rceil$. If this checkpoint exists it will consider the interval $[\lceil n/2 \rceil, n]$, otherwise it will consider the interval $[1, \lceil n/2 \rceil)$. In either case, it will split this second interval in two and repeat the procedure until it finds the highest available checkpoint. For example, if $n = 10$, the node will look for checkpoint 5. If checkpoint 5 exists, it will now look for checkpoint $\lceil (10-5)/2+5 \rceil = 8$ and so on, until it may find out that 7 is the highest available checkpoint. When setting limits for these intervals, the worker must also try some checkpoints beyond the limits it previously found to make sure that a negative answer is not due to a disappeared checkpoint. For instance, checkpoint 5 might have been missing, which would make the node restrict its search to the interval $[1, 5)$. However, it could be the case that checkpoints 6 and 7 were still there and the node would wrongly get checkpoint 4 as the last one.

An aspect that we evaluate experimentally and which is crucial to the performance of our scheme is the availability of the checkpoints. As referred before, we use indirection pointers to separate storage from the DHT. A consequence of this is that checkpoints may be lost due to the disappearance of the indirection pointers stored in the DHT. To overcome this problem, the processing node periodically refreshes the pointers to old checkpoints. It may also occur that indirection pointers are left hanging, either because the storage point left the network or because it deleted the checkpoint. In this case, the node looking for the checkpoint must try to fetch checkpoints with lower numbers.

6.5 Garbage Collection

As introduced earlier in Section 6.3, under the *push* mode, garbage collection is performed by the worker that has finished a task. Specifically, after having completed the task, the worker sends a message to every node that store information of the task: WTI_i, pointers to checkpoints, and to the storage points. In fact, as we show ahead, deletion of the WTI_i tuple requires more than a simple deletion message. Alternatively, we also define a *pull* mode, if for some reason a storage node is left with state of task i hanging (for example, the finishing worker cannot complete the send operation of the deletion message). Additionally, nodes that store replicas of large checkpoints also use the *pull* approach if they need to recover space before the task ends. Indeed, although the storage point can immediately delete any replicas, it can also use a more graceful approach of fetching the WTI_i to know if the task is over or the checkpoint is old.

It may happen that when the worker node tries to delete the WTI_i, this tuple is temporarily unreachable, just to come back later and violate the invariant INV, leaving an orphaned WTI_i (Section 6.4.1). To avoid this inconsistency, when the worker node finishes task i, it stores n as the last checkpoint written, which means that the task ended. For garbage collecting purposes, reading n as the current checkpoint is the same as not finding the task — it just means that the task is not running. The $guardian_i$ must store this tuple for some time before deleting it, to ensure that a finished task cannot come back to life, due to some transient misbehavior of the DHT, capable of bringing an old WTI_i back. Finally, $guardian_i$ can only delete WTI_i with a checkpoint value lower than n after asking the central supervisor whether task i has already finished.

6.6 Evaluation

In this section, we evaluate the advantages of replicating checkpoints to recover from failures. We compare, through simulation, the turnaround time of *chkpt2chkpt*, where each checkpoint is replicated exactly once, versus a typical private solution, where each worker locally stores its own checkpoints. As in previous chapters, we use the traditional definition of turnaround time, corresponding to the time that goes from the moment when the central supervisor distributes the task up to when it receives the last result. Furthermore, we assume a homogeneous set of workers, with individual nodes prone to crash-stop and crash-recovery failures, both of them following a random geometric distribution. At discrete time intervals, we randomly decide whether the worker changes state with a probability that is fixed throughout the computation of the task. In the crash-stop model, a node can change from working to crashed without ever recovering, corresponding, in the context of a volunteer project, to a worker that has simply abandoned the project. Under the crash-stop model, the task will only restart when it is rescheduled to another worker. Finally, in the crash-recovery model, the worker node can change from working to crashed state and vice-versa with the same probability. In the private checkpointing solution, a new worker must restart a reassigned task from the beginning, while in *chkpt2chkpt* it can be resumed from the highest available checkpoint.

Under ideal execution conditions, that is, if run uninterrupted and with full machine dedication, a task requires t_{exec} time units to complete, with checkpointing occurring every $t_{checkpoint}$ time units, for a total of n checkpoints ($t_{exec} = n \cdot t_{checkpoint}$)[4]. Additionally, we consider that the execution pace of the tasks is dictated by two timeouts: the timeout of the entire task t_t and the timeout t_p. We set $t_t = 3 \cdot t_{exec}$ (only for private checkpoints) and $t_p = 3 \cdot t_{checkpoint}$ (only for distributed checkpoints). If these timeouts expire, the entire task is *immediately* reassigned and restarted. In all cases, we set $t_{checkpoint}$ to be 10 time units and fixed the number of checkpoints per task (n) to 5.[5] Hence, we have $t_{exec} = 50$, $t_t = 150$, and $t_p = 30$ (see Table 6.2). When a task is reassigned to another worker, the private approach restarts the computation from scratch, that is, from checkpoint 1, while in the distributed checkpoint solution, the new worker tries to fetch a previous checkpoint. Unless otherwise stated, the probability of recovering each of the previously saved checkpoints is set to 50% (we take the most recent one, that is, the one with the highest index).

The simulation results, corresponding to the average of at least 50 random trial points, are plotted in figures 6.3 to 6.5. Specifically, figures 6.3 and 6.4 compare the execution times of private versus distributed solu-

[4]We assume that the n^{th} checkpoint is saved at the end of the task (in fact, it can be regarded as the result of the task).

[5]As expected, when we keep the failure ratio constant and increase the number of checkpoints for the same task, the turnaround time clearly improves until some point and then becomes nearly constant.

$t_{checkpoint}$	n	t_{exec}	t_p	t_t
10	5	50	30	150

Table 6.2: Settings of the experiment.

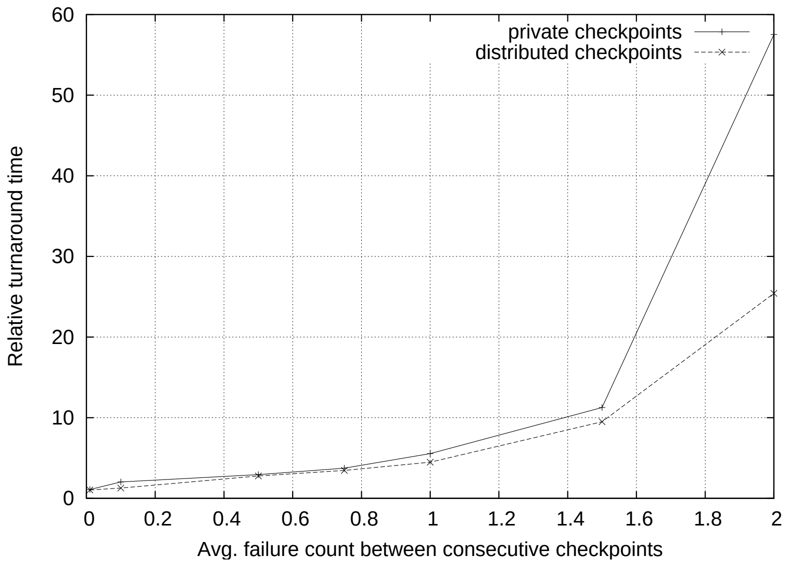


Figure 6.3: Turnaround time with crash-recovery failures.

tions when the failure rate increases, for the *crash-recovery* and *crash-stop* models, respectively. These execution times are relative to the minimum possible execution time, i.e., t_{exec}. The average count of failures that occur between consecutive checkpoint operations ($t_{checkpoint}$) is represented in the x-axis. The curves show that our scheme performs better for higher failure rates. This makes sense, because if failures are rare, i.e., if the environment is only lowly volatile or not volatile at all, there is no real need to share checkpoints, as we have already observed in Chapter 5 with the turnaround times obtained over weekends. Under the crash-stop model, where a worker never returns to its task after failure, the distributed approach yields even better results when compared to the private checkpointing approach. This is a consequence of the fact that the failure state lasts longer than the time needed to execute the task, a situation that degrades performance in the private approach. On the contrary, the shorter inter-checkpoint timeouts of the distributed approach, t_p, enables a faster reac-

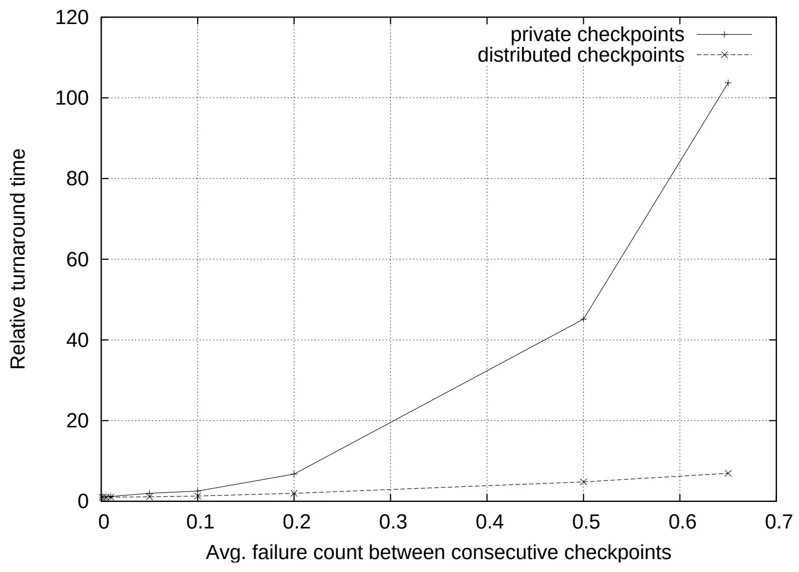


Figure 6.4: Turnaround time with crash-stop failures.

tion. This comparison is fair, because no dependency exists on the central supervisor to manage these per-checkpoint or per-process timeouts (except when they cause a redistribution). Finally, in Figure 6.5 (page 145), we evaluate the impact of the probability of checkpoint availability (for a fixed crash-recovery probability). It is quite clear that the availability of checkpoints is crucial to the performance of our system: if availability is too small, like 40% or less, *chkpt2chkpt* is of low utility. Hence, we believe that these results show the validity of the *chkpt2chkpt* approach and enable us to derive some conclusions about the benefits of using P2P structured techniques in wide-scale desktop grids.

6.7 Related Work

In this section we discuss related work. We focus on two major areas related to *chkpt2chkpt*: use and management of distributed checkpointing, and scavenging systems that resort to structured DHT overlay networks.

In Chapters 4 and 5, we analyzed the effects of sharing checkpoints in local area environments, resorting to a centralized checkpoint server [Domingues *et al.* 2006d]. Likewise, Condor [Thain *et al.* 2005] relies on a central server for sharing checkpoint files, and allows the migration of tasks

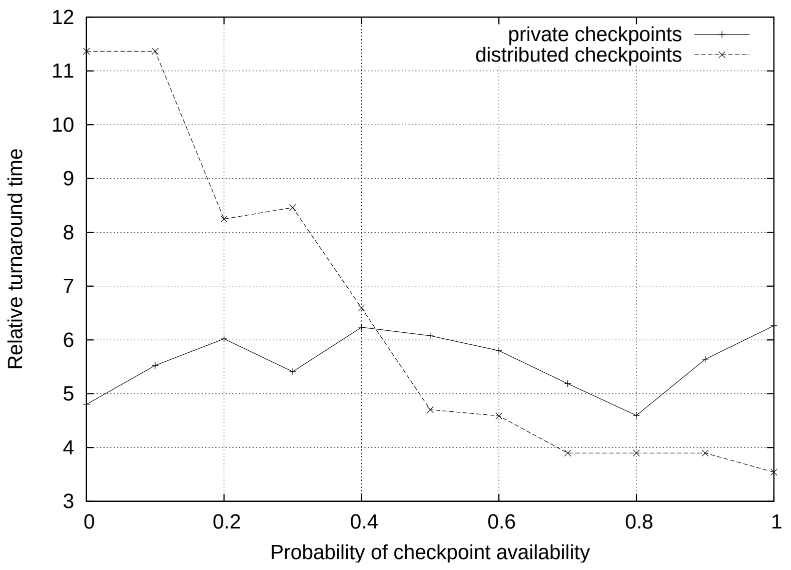


Figure 6.5: Turnaround time for varying checkpoint availability.

for fault tolerance and faster turnaround time. However, these approaches are limited to LAN environments, while *chkpt2chkpt* targets Internet-based desktop grids with possibly thousands of nodes.

Tritrakan and Muangsin [Tritrakan & Muangsin 2005] simulate the benefits of direct communication between a submitter machine that proposes a work and worker nodes in a desktop grid environment. Under their approach, scheduling is still done centrally, but the transfer of the needed files (input data and/or results) occurs directly between the submitter machine and the selected worker machine. A central file repository is still needed to cope with situations when a worker machine is not promptly available at submit time, and also, to deal with submitter machine unavailability when results are ready and need to be transferred from the worker machine. When compared to our work, this approach uses a different semantic in the access to resources, since submission of tasks is no longer the sole exclusivity of a central machine. This freedom, although interesting, might pose some serious security risks, especially in untrusted environments like openly accessible desktop grids.

Wei et al. [Wei *et al.* 2005] explore the use of BitTorrent [Cohen 2003] to solve the scalability issues that arise with large data files. The authors compare the performance of FTP-based solutions to BitTorrent-based solutions, concluding that BitTorrent is effective for deploying large files required by a significant number of workers. For relatively small files, the high overhead of BitTorrent renders its use counterproductive. Our approach is different, since we aim to promote file sharing directly between worker nodes.

Several works resort to structured DHT overlay networks for achieving different purposes. For instance, WaveGrid is a peer-to-peer desktop grid system aimed to achieve fast turnaround execution times [Zhou & Lo 2006]. It resorts to the peer based model, where all peers can submit applications to be executed over the desktop grid system. To enable communication within peers, WaveGrid makes use of a CAN DHT overlay network [Ratnasamy *et al.* 2001].

Another DHT-based infrastructure is proposed by Butt et al. [Butt *et al.* 2003]. They present a technique which uses a Pastry DHT [Rowstron & Druschel 2001] for resource discovery in distributed Condor pools [Thain *et al.* 2005] spread over several administrative domains, to overcome the restrictions of the statically defined flocking mechanism supported by Condor.

FreeLoader [Vazhkudai *et al.* 2005] and Squirrel [Iyer *et al.* 2002] are two Pastry-based systems that scavenge resources in local area environments. FreeLoader resorts to unused storage of regular desktop machines to provide disk space for immutable scientific datasets. Squirrel targets web caching and exploits locality in web data object references. The key idea is to enable web browsers on desktop machines to share their local caches, without the need for dedicated hardware and the associated administrative costs.

Similarly to the work we present here, there are many other systems that use DHTs to manage data, from file systems to replicas of entire systems. For instance, Venti-DHash is a cooperative backup system, which couples the Venti backup system with an Internet peer infrastructure for archiving snapshots of file systems [Sit *et al.* 2003]. Venti-DHash uses DHash, which is a Chord-based distributed hash table (DHT). Pastiche [Cox *et al.* 2002] is another peer-to-peer backup system that resorts to a Pastry [Rowstron & Druschel 2001] DHT for the identification and organization of redundant data for saving space storage. Unlike our application-level checkpointing and unlike Venti-DHash, which acts at the block level, Pastiche makes replicas at the machine level.

Other interesting approaches to create file systems are Shark [Annapureddy *et al.* 2005] and Kosha [Butt *et al.* 2004]. The main asset of Shark lies in its cooperative-caching mechanism, in which mutually distrustful clients use a DHT to exploit their associated file caches to reduce load on a file server. Finally, Kosha aims to harvest unused storage of desktop machines within a LAN environment. It uses a structured overlay network to provide location and mobility transparency, load balancing and file replication.

6.8 Summary

In this chapter, we presented an extension to the traditional desktop grid architecture by using a DHT to maintain decentralized replicas of checkpoint files, thus promoting sharing of checkpoints over a wide-scale environment. With the use of this technique, any node of the grid can resume a failed task provided that a checkpoint file is available in the P2P infrastructure. Almost all interactions needed to replicate checkpoints are decentralized among the DHT, thus containing the load on the central supervisor. Moreover, we keep all the interactions involving the central supervisor strictly worker-initiated, without disrupting the basic assumptions of existing architectures, and thus easing the adoption of the proposed checkpoint infrastructure. Simulation results show that our proposed scheme can considerably reduce the turnaround time of tasks when there is a significant probability of node failures. In this way, the use of P2P techniques in desktop grids seems to be a promising approach, although several issues such as scalability and sabotage-tolerance of the whole system need to be addressed. We examine some of these issues in the next chapters.

7 Desktop Grid Topologies for Sharing Input Data and Checkpoints

In this chapter, we propose some extensions to the traditional desktop grid model, focusing on the dependability aspects of these extensions. Specifically, we concentrate our attention on hybrid topologies that introduce intermediary nodes between the central server and the workers. We first suggest a local proxy server (LPS) for institutional desktop grid resources that acts as a liaison machine between the local resources of the institution and the central server. Then, we widen the topology to accommodate unrelated home volunteer resources, thereby exploiting dependability in a more unreliable context. For this purpose, we resort to a network of super nodes, focusing on the fault tolerant methodologies that can be used in such instable environments.

7.1 Introduction

As stated in previous chapters, the current approaches for wide-area desktop grid computing are based on a centralized model. Although this model provides for central control, easing the management and supervision of projects, it also creates a single point-of-failure and a potential performance bottleneck. Coping with this performance constriction point usually requires costly resources such as capable hardware and substantial bandwidth for the server-side.

In this chapter, we look at federated and peer-to-peer approaches for the management of volunteer desktop grid resources with the goal to reduce the bandwidth demand and the computational load at the global servers. We then focus on how some of the fault tolerant-based techniques that were covered in previous chapters can be adapted to these new topologies. It is important to note that the extension to move decentralized architecture should be achieved without disrupting the current server-based model of volunteer computing, so that existing middleware can easily be adapted to these paradigms, allowing workers and volunteers to swiftly migrate to the new model. This is relevant for existing projects that have a large community of users.

This chapter is organized as follows. We first lay out the levels of cooperation that we deem achievable in desktop grids. We then present two topologies, one that targets structured organizations, namely institutional desktop grids, and another one that encompasses P2P overlays in environments such as typical home volunteers. The distinction between institutional environments and home volunteers is due to the fact that the former can be more tightly controlled while the latter is comprised of individual and mostly anonymous nodes. Our presentation is oriented toward the benefits that dependability mechanisms can bring to such environments. We finish by reviewing related work and outlining the main conclusions.

7.2 Levels of Cooperation

In this section, we briefly identify several levels of cooperation among desktop grid nodes, ranging from the simplest one – caching of input data sets – to the most demanding one – communication between cooperating workers.

1. **Caching of input data sets**. The designation of *input data sets* encompasses the input data that are needed to carry out the execution of a task. The rationale for caching input data sets comes from the fact that some applications perform multiple runs, possibly with different parameters or algorithms, on the same input data sets. In the case of applications executed over desktop grids, this corresponds to several tasks requiring the same input data. Since input data sets for individual tasks can attain several tens of megabytes or more, a clever caching of input data set can significantly save bandwidth both at the server's and at the worker's side. A concrete example is the public

project Einstein@home [einstein 2007]. This project resorts to parameter sweep, exploring a wide range of signal parameters (frequency and spin-down rate), with a given input data set originating several tasks. For instance, the execution of a reference workunit[1] of the Einstein@home project requires the download of 18 files, for a total of 53 MB of data. In our approach, we aim to extend the caching of an input data set from the worker realm – where only one worker node benefits – to an intermediate level, upon which several nearby workers might share a data set that was downloaded from the supervisor by only one of them.

To prevent tempering of input data sets, the server-side that produces the data sets can sign them, resorting to its private key, while the complementary public key is made available to workers. Each time a worker receives an input data set from a non-trusted source, it can verify the data set through the data set's signature. This way, it is straightforward to verify the integrity of shared input data sets.

2. **Sharing of checkpoints**. A further level in cooperation is to have workers to share checkpointed states. As seen in Chapter 4 and Chapter 5, the sharing of checkpoints can significantly reduce execution times, especially in volatile environments. Moreover, sharing checkpoints also increases the rate of successful executions, thus reducing the needs for rescheduling as well as the traffic and load at the server-side. Note that sharing checkpoints departs from the caching of input data sets since the data to be shared (checkpointed states) are produced by the workers, while input data are made available by the server-side, and thus from the point of view of workers, input data are *read-only*.

3. **Communication among workers**. Communication among workers represents the highest level of cooperation, allowing to support applications with dependent tasks, thus significantly broadening the class of applications that can be executed over desktop grid resources.

   Communication can be *indirect*, for instance through distributed tuple spaces [Gelernter 1985]. Indeed, a tuple space can be used for exchanging information among local cooperating worker nodes follow-

[1] http://www.aei.mpg.de/~bema/einsteinathome/refwu.zip

ing, for instance, the *blackboard system metaphor* [Corkill 1991]. This approach can allow the execution over desktop grids of loosely coupled applications, i.e., applications that can be broken into independent sub-tasks, each one requiring heavy processing with light communication. Examples include brute force search like the *ChessBrain* project [chessbrain 2007; Frayn & Justiniano 2004], Monte-Carlo simulations and distributed evolutionary algorithms such as genetic programming and simulated annealing [Kirkpatrick *et al.* 1983]. All of these applications involve the parallel generation of a batch of solutions and then using these solutions to come up with an answer for the initial problem [Gupta & Sekhri 2006]. Conversely, communications between workers can be *direct*, with a point-to-point channel, or with a node relay between them. This requires either NAT traversal mechanisms such as *Simple Traversal of UDP NAT* STUN [Rosenberg *et al.* 2003], *Traversal Using Relay NAT* TURN [Rosenberg *et al.* 2004] and *Interactivity Connectivity Establishment* ICE [Rosenberg 2006]. A good survey of NAT punching techniques is given by Ford et al. [Ford *et al.* 2005]. Providing for communication among workers is a vast subject, whose details are out of scope of this thesis.

7.2.1 Assessing the Benefits of Cooperation

In this section, we briefly assess the benefits of cooperation focusing on the savings that can be achieved at the server-side level. For this purpose, we consider the success and error rates observed from the execution attempt of several thousand tasks from various BOINC-based projects.

Table 7.1 (page 153) aggregates the success and failure rates for the following BOINC-based projects: Einstein@home, Rosetta@home, QMC@home and SETI@home projects. For each project, the data were gathered by accessing the status of workunits publicly available at the site of each project through specially crafted URLs. Specifically, the URLs were *PROJECT/workunit.php?wuid=ID*, where *PROJECT* is the URL of the project and ID is the identifier of the sought workunit[2].

The column *Quorum* of Table 7.1 indicates the quorum level of the project. For instance, the Einstein@home project requires a *quorum of 2* for the vali-

[2]For instance, the URL for accessing the data regarding an Einstein@home's workunit is http://einstein.phys.uwm.edu/workunit.php?wuid=ID.

Project	**Quorum**	**Attempts**	**Success**	**Failure**
Einstein@home	2	13502	8032 (59.49%)	5470 (40.51%)
SETI@home	2	5924	3660 (61.78%)	2264 (38.22%)
QMC@home	1	12402	9210 (74.26%)	3192 (25.74%)
Rosetta@home	1	10129	9410 (92.90%)	719 (7.10%)
Total	-	41957	30312 (72.25%)	11645 (27.75%)

Table 7.1: Execution statistics regarding several BOINC-based projects.

dation of a workunit. This means that a workunit is only finished when the execution of its two tasks yield the same result[3]. So, whenever a task fails to complete or the results of a task do not match, another task is scheduled for execution. This process is repeated until a majority of tasks produces the same result or when the threshold for the maximum number of executions is reached (20 for the case of Einstein@home). In Table 7.1, the column *Attempts* counts the number of execution attempts of tasks, with the columns *Success* and *Failure* reporting how many of these attempts were successful and how many failed, respectively.

Analyzing Table 7.1, it can be seen that, for the Einstein@home project, 13502 execution attempts yielded 8032 correct executions and 5470 did not complete or were erroneous. This means that 40.51% of the execution attempts ended up with no results and needed to be rescheduled by the server-side for repetition. SETI@home presents similar results with a failure rate of 38.22%. On the other hand, the Rosetta@home project yielded a success execution rate above 92%. This is mostly due to the shorter duration of tasks of Rosetta@home comparatively to the other projects. Conversely, Einstein@home is the most resource demanding of all the studied volunteer projects. Although it would be possible for the Einstein@home to distribute smaller, less demanding tasks to achieve higher rates of successful executions, this would increases the load on the server-side since more tasks and input data sets would need to be downloaded. Thus, a delicate balance needs to be achieved between the duration of a task and the amount of input data that its execution requires.

Across the observed projects, 27.75% of execution attempts failed, meaning that slightly more than a quarter of tasks needs to be rescheduled and consequently requires a redistribution to workers. Therefore, as much as

[3]Under BOINC, a task is an instance of a workunit.

27.75% of resources end up wasted at the server-side, especially bandwidth, the same happening at the workers'.

Some projects, like Einstein@home, promote the reuse of input data by attempting to schedule tasks to workers that already have, from previous executions of other tasks, the required input data sets. This feature is called *locality scheduling* [Anderson *et al.* 2005]. To assess the effect of locality scheduling in Einstein@home, we analyzed the publicly available scheduling logs for seven consecutive days[4]. From the logs, it was observed that there was 547,532 requests for workunits, with 390,827 (71.38%) of these requests receiving workunits requiring input data already present at the worker's. Thus, only 156,705 requests (28.62%) required the downloading of the associated input data set, making locality scheduling an effective bandwidth saver for the Einstein@home project.

Both the first and the second level of cooperation can mitigate the negative impact of failures at both server-side and at the worker's by reusing input data and previously checkpointed computation. Specifically, through sharing of input data, level one has the potential of reducing the data traffic between the server-side and workers. Likewise, by promoting computation reuse through sharing of checkpoints, level two increases the success rate of executions, lessening the amount of tasks that needs to be redone and therefore the resources demand at both server and workers.

To quantify the effects of reducing the failure rate P_ε, we analyze the outcome considering that a Δ_ε% reduction is achieved on just 20% of the execution attempts. Specifically, Table 7.2 shows the effects on the failure rate P_ε of several BOINC-based projects, when the failure rate P_ε is reduced by Δ_ε% on just 20% of the execution attempts. The column Δ_ε=0% displays the failure rates previously shown in Table 7.1 (page 153), while Δ_ε=100% represents what would happen to the overal failure rate of each volunteer project if 20% of the execution attempts would be successful, while the other 80% would maintain the original failure rate. For instance, for the Einstein@home project, cutting the failure rate by 50% (Δ_ε=50%) for solely 20% of the execution attempts, yields a failure rate of 36.46%. This way, if cooperation techniques such as reuse of input data and sharing of checkpoints can be applied to 20% of the workers and cut their failure rate in half, the overall failure rate of the project drops from 40.51% to 36.46%, that is a

[4]http://einstein.phys.uwm.edu/sched_logs/, May 2007.

Project	Δ_ε=0%	Δ_ε=10%	Δ_ε=25%	Δ_ε=50%	Δ_ε=75%	Δ_ε=100%
Einstein@home	40.51%	39.70%	38.48%	36.46%	34.43%	32.41%
SETI@home	38.22%	37.46%	36.31%	34.40%	32.49%	30.58%
QMC@home	25.74%	25.23%	24.45%	23.17%	21.88%	20.59%
Rosetta@home	7.10%	6.96%	6.75%	6.39%	6.04%	5.68%
Total	27.75%	27.20%	26.36%	24.98%	23.59%	22.20%

Table 7.2: Effect on reducing by Δ_ε the failure rate P_ε on 20% of the execution attempts.

drop of 4.05%. This way, by simply halving the failure rate on 20% of the execution attempts, the Einstein@home project would see its exploitable computing power grow by 4.05%.

7.3 Federating Institutional Desktop Grids

A first level for hierarchical organization of resources of a volunteer infrastructure is to complement the resources of a local institutional desktop grid with the addition of a *local proxy server* (LPS). A local proxy server is a machine that holds services responsible for aggregating the resources of a single geographical institutional site. It interacts with the global server-side, taking advantage of its collective view and knowledge of the local resources to reduce the interactions between local resources and the global server-side.

As seen in Chapter 4, the designation *local institutional desktop grid* (LIDG, for short) refers to the volunteer resources of an institution (for instance, an academic campus or a corporation) possibly located at a single geographical site so that network connections among the volunteer nodes of a single site are provided by local area technology. Furthermore, we assume that local institutional environments are coordinated by a (local) central management, who authoritatively decides which and how resources are to be volunteered, overcoming any resistance that personnel might have toward volunteering machines.

The local proxy server infrastructure exploits two main characteristics of institutional environments: tight control of resources and fast internal communications. Indeed, under a central management who can enforce and control a given volunteering policy, a cooperative behavior is easier to achieve. Regarding communications, a single geographical site brings the benefits of faster inter-worker connections plus network symmetry. In fact, profiting from local area network technologies, worker nodes are connected to each other and to the local proxy server by fast communication links. Actually, communication links within the worker nodes are significantly faster than the ones that connect the whole institution to the outside world, namely to the global server-side. Additionally, communications among the worker nodes of an institution can usually flow quite freely, since they are not hampered by NAT schemes, and individual nodes' firewall policy can collectively be adjusted by the intervention of the management authorities of the institution.

7.3.1 Overview

A local proxy server mediates the interaction between the client workers running on the institution's machines and the global master server-side. Specifically, any request to the master server-side performed by a worker is routed through the local proxy server, which forwards it to the global master server. Likewise, when the global master server answers back, the proxy local server reroutes the answer back to the worker. This communication routing allows the local proxy server to maintain state information about the local workers, and to effectively cooperate with the global master server. For instance, the request of a worker for tasks can be complemented by the local proxy server with the list of input data sets that already exist at the institution. This allows the global master server to assign a task whose data set already exists locally, thus avoiding the download of a possibly lengthy data set. Therefore, having the local proxy server intermediating the exchanges amongst workers and the global server-side allows to implement mechanisms such as caching of input data that can benefit the community of local worker nodes and reduce the traffic to the global servers.

As reported in Chapter 2, the *SZTAKI Desktop Grid* (SZDG) [Kacsuk *et al.* 2005; Balaton *et al.* 2007] extension to BOINC is a good example of a LPS-based system. Specifically, SZDG implements an hybrid BOINC module that acts as a client relatively to the BOINC master, and acts as a server to the BOINC clients that sit lower in the hierarchy and that actually execute the tasks. Therefore, SZDG can be used as a Local Proxy Server, with checkpoint sharing easily integrated into the framework.

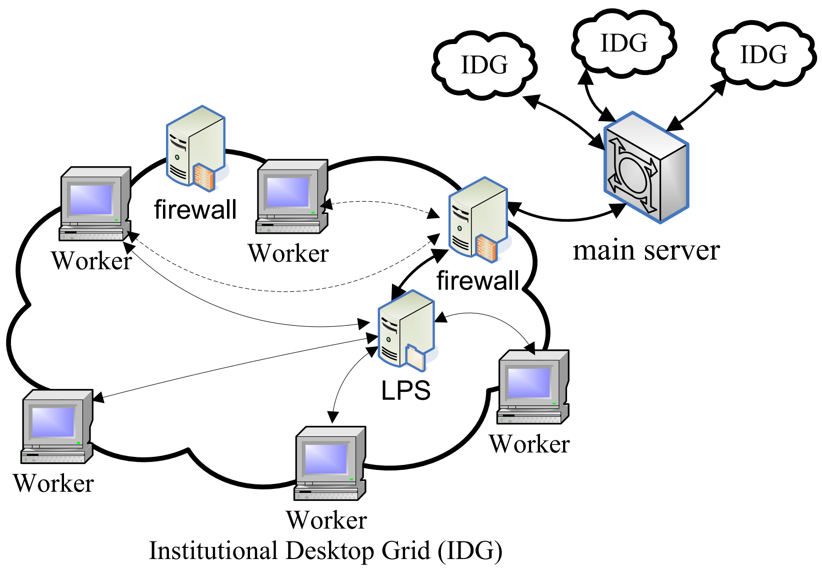


Figure 7.1: A Local Proxy Server in a IDG.

7.3.2 Functions of the LPS

A *local proxy server* can have three main functions: (1) to cache input data sets, (2) to act as an institutional checkpoint repository, and (3) to support communication and synchronization operations among the local nodes of the institutional environment. Next, we detail each of these functions in the context of a local desktop grid environment. Figure 7.1 represents an institutional desktop grid fitted with a local proxy server. In the depicted environment, any connection to the Internet needs to be done through the institution's firewall, including the connections managed by the LPS. The dashed line represents the ability of the local machines to bypass the LPS to connect to the master server-side (for instance, to circumvent an unresponsive LPS). Indeed, local proxy services are provided as a best effort. When such services are unavailable – for instance, a failure occurred at the local proxy server – local worker nodes can still directly interact with the global servers, as it happens in the classic server-based model. This way, the usual disadvantages of a central point of failures such as the LPS are somewhat circumvented.

Cache of input data sets. A local proxy server is a perfect fit for acting as a cache of input data sets. When requesting tasks to the master server-side, the LPS can send along the list of input data sets that it already has, so that the master side can preferentially assign tasks related to those data sets. A further benefit of empowering the LPS with the responsibility of locally scheduling tasks is that it allows for an easy implementation of the shared checkpointing mechanism, as shown next.

Institutional checkpoint repository. Besides caching input data sets, a local proxy server can also act as a checkpoint repository, storing copies of the checkpoints saved by the local workers. Then, whenever a node goes down and remains unavailable for a period of time longer than a predefined threshold, the local proxy server assumes the node is on a *long downtime period*, and thus assigns the partially executed task to an idle node as soon as one is detected. The identification of idle nodes is simplified by the fact that the workers' traffic to the project master server is routed through the local proxy server, since idle nodes will be the ones requesting tasks. Thus, a requesting node will receive the uncompleted local task, jointly with the last stable checkpoint, from where the task can be resumed. This way, the execution time of tasks can be significantly reduced. Sharing of checkpoints contributes for a higher success rate and thus further reduce the number of tasks that needs to be rescheduled by the server-side.

7.4 Desktop Grids for Unrelated Peers

Similarly to institutional desktop grids, aggregating unrelated peers like home users who have volunteered their machines might yield substantial benefits. Aggregating unrelated peers allows to implement collective schemes, such as caching of input data files and sharing of checkpoints. In addition, if resources are properly aggregated at the network level, organizing peers into groups allows for the implementation of loosely coupled communication amongst workers, making possible the execution of cooperative applications.

Contrary to LAN-based federated desktop grids, aggregating unrelated peers is much more challenging. Firstly, an environment comprised of unrelated peers, or at best, loosely coupled peers poses some serious security and trust issues. In fact, while control and central management is rather easy to enforce in LAN-based institutional desktop grids, unrelated peers environments are much more unreliable, and thus workers' security, as well as the integrity and accuracy of results should be a priority. Moreover, connection management needs to be addressed, since a substantial percentage of volunteer machines are behind a firewall and/or a NAT server and thus are not directly addressable from the "outside". Finally, volunteer machines have high churn [Anderson & Fedak 2006], and a variable and often unpredictable uptime, making cooperating schemes more difficult to implement.

7.4.1 A Model for Grouping Unrelated Peers

In this section, we propose a super node based solution, that we name *Super Node-Based Desktop Grid* (SNBDG), upon which so-called *super nodes* assume a special coordinating role. Note that we still follow the traditional reverse client-server desktop grid model, relying on the existence of a master-side which ultimately coordinates the whole computation. Our approach focuses on empowering the edge nodes, with networks of *super nodes*[5] connected together and providing scheduling and storage services to attached worker nodes.

Super nodes in peer-based environments serve to accommodate the heterogeneous set of machines that are volunteered to public computing project. In fact, it has been shown that the efficiency of peer-to-peer schemes can be hampered if all nodes are required to perform the same demanding network functions [Chawathe *et al.* 2003]. By splitting nodes in super and regular ones, the whole network services can perform smoothly, benefiting all the enrolled volunteers.

Architectural organization. SNBDG follows an unstructured peer-to-peer approach similar to the ones laid out by the successful FastTrack protocol [Liang *et al.* 2006] (KaZaA) and Skype [Guha *et al.* 2006], where nodes are differentiated into *super node* and *regular*, accordingly to their functions.

A rough layout of the SNBDG architecture is shown in Figure 7.2 (page 160). Apart the server-side, nodes are split in super nodes and regular worker

[5]Other designations of super nodes found in the literature include *super peer* and *ultrapeer*.

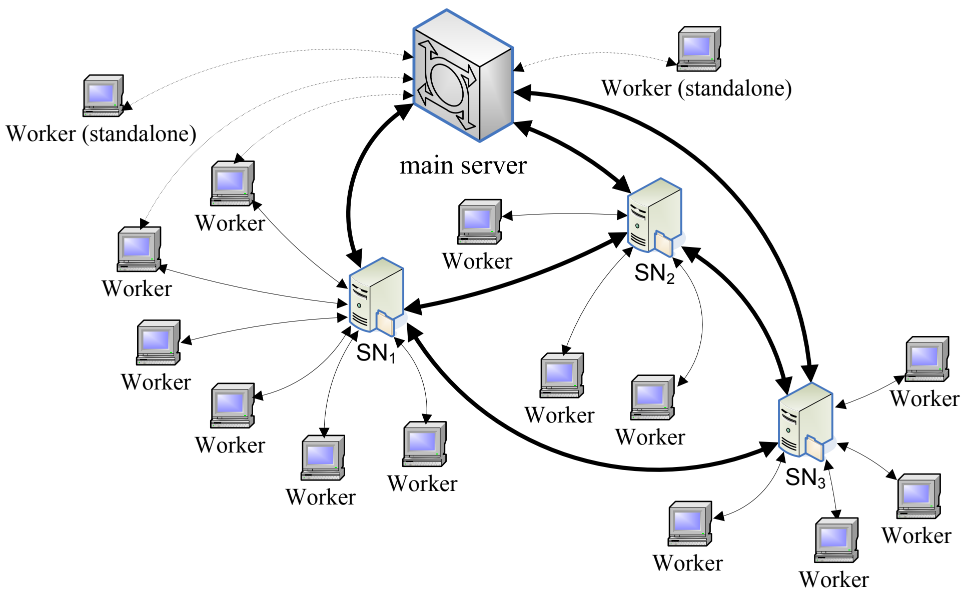


Figure 7.2: Architectural organization of SNBDG.

nodes. The formers are connected with each other, forming a partially connected graph, while worker nodes are connected to super nodes. Note, that worker nodes which are not connected to any super node – independent nodes – can still exist, but those nodes will not benefit from the SNBDG infrastructure. Furthermore, the independent state of a worker node might be permanent or transient, when, for instance, the super node to which the node was attached disappears and the node has not yet attached to another super node.

Regular worker nodes. A regular worker node is simply a worker node that is connected to a super node. Like the worker nodes of current desktop grid models, under SNBDG, regular nodes handle the core activity of the desktop grid, that is, they process the tasks that are assigned to them. In addition, a regular node can interact with its super node to access services such as tasks scheduling and the sharing of input data sets and of checkpoints.

Super nodes. The main role of a super node is to promote and coordinate the cooperation among the regular nodes which are connected to it. Specifically, a super node has the following main functions:

- Manage tasks, requesting them from the master server, and distributing them to requesting workers. The super node might also upload the results to the master server side, although this operation might be done directly by the worker node if no cost nor speed savings are effectively achieved by proxying the results through the super node.

- Cache input data sets needed for processing the tasks. This makes way for an input data set to be reused among several workers, possibly avoiding several downloads of the same input data set from the project's master side, thus preserving bandwidth both at the worker and at the server-side. Note that this feature can be regarded as an extension of the LPS's locality scheduler, as seen in section 7.3.2.

- Provide storage to hold checkpoints of partially executed tasks. This feature needs to be balanced between the checkpoints size and both the storage space and the bandwidth required for uploading and downloading checkpoints between regular nodes and the super node.

- Support the resiliency of the system, tolerating failures of other super nodes of the network. Indeed, whenever a super node departs from the network, its functions and responsibilities should be taken over by neighbor super nodes.

In order to perform the functions of a super node, a node needs adequate technical characteristics, namely appropriate bandwidth, a public IP address, no restricting firewall, sufficient storage space, and, preferentially, long uptime. This means that not all volunteer nodes can perform as a super node. In fact, due to the resource requirement that the duties of a super node might impose, namely at the bandwidth level, only nodes that have explicitly being authorized by their owners can be assigned a super node role. A major critic raised over the Skype's VoIP (*Voice over IP*) infrastructure [Guha *et al.* 2006] lies in the fact that the owner of a node has no control over the possibility of her machine becoming a super node. Indeed, in the Skype's network, the conversion to super node is determined and performed automatically by the Skype software running at the node. And all of these critics are made to an application that provides an added value VoIP service to its users, which is the opportunity to talk and to text for free to other Skype users. Thus, in a desktop grid system, where resources are volunteered by users for a common cause, and many times without direct benefits for the resource providers, the extra demand that the role of super node imposes need to be explicitly authorized by the resource owner, otherwise the system is most probably deemed to failure, since potential volunteers would not enroll in the system.

We now outline the network dynamics of the SNBDG topology. We first examine how the nodes can find and connect to a network of super nodes. We then overview the main operations of a super node and proceed to present how the network can deal with departure of nodes, either worker or super node ones.

Connecting to a network of super nodes. To connect to a network of super nodes (NSN), a node – regular or super – first needs to obtain the address of a super node. The way this is achieved depends on whether the node has previously been connected to a network of super nodes[6], or on the contrary, if this is the node first contact with a network of super node (*first-time node*). Previously connected nodes can simply resort to their persistent list of IP addresses of super nodes, and try to contact each one of the nodes until a successful connection is obtained. The worker node can retrofit its own list of super nodes, based on past sessions. For instance, the first entries of the list should contain the super nodes which have been, at the same time, highly available and close from a network point of view.

For first-time nodes, or similarly, for nodes whose list of past-connected super nodes has no currently available super node, a request can be made to the global server, so that a list of super nodes can be obtained. In fact, the global server can easily keep an up-to-date list of super nodes by having nodes to report their status whenever they contact the global server. Note, that such a global server-based scheme should be seen as a last resort, otherwise the global server-side can be flooded with queries for super node addresses.

[6]Due to the unstructured way that super nodes aggregates themselves, it is possible that more than one network of super nodes exist at a given time. In such case, two super nodes associated to different networks will not able to contact each other via the super node's network.

After having connected to a super node network, a regular node may try to seek a better connection or a more suited super node (for instance, the current one might be overloaded) by trying out other super nodes. This is made possible by experimenting other locally saved addresses of super nodes, or fresh ones obtained from the current super node. Likewise, a freshly connected super node will also seek other connections, but contrarily to a regular node, not only for the purpose of obtaining a more appropriate connection, but also for seeking out other super nodes to which it can connect in order to fully integrate the network of super nodes. Indeed, a super node needs to maintain connections with several super nodes in order to more softly tolerate the departure of a super node to which it is attached to.

Operations over a network of super nodes. The type of operations in which a node can engage on a network of super nodes obviously depends on its role: *worker* or *super* node. A worker node will mostly request tasks from its super node (tasks can also be requested directly from the master server-side), and, if needed, access existing data like input data sets or shared checkpoints.

Super nodes have a wider involvement than regular nodes on the super node infrastructure. For instance, a super node needs to maintain a list of the input data sets it stores locally, and another one with the input data sets which are easily accessible to it, that is, they are stored at other super nodes to whom it has direct contact with. The super node also needs to assume scheduling responsibilities, by distributing tasks, new and partially executed ones, to requesting workers. Another possible function of a super node might be to host checkpointed states of partial executions, and distribute them to other requesting worker nodes whenever the worker node that was processing a task is no longer executing it. On top of this, a super node is also involved in other tolerance-related mechanisms such as the ones needed to support the departure of a worker node or, more importantly, of another node to which it is connected. Next, we deal with departures of nodes.

Departing from the network of super nodes. As expected in non-dedicated resource environments such as desktop grids, a node might depart the super node network at any time, in either a *soft* or a *hard* manner. In the former, the node is softly shutdown, and it might still have time to communicate its departing status, although this may not happen: for instance, the network message(s) signaling its departure might not reach its destination(s), and since the node is departing the network, retransmission might not be feasible. In the case of a unexpected failure, the node disappears without notifying the network.

Departure of a worker. A departing worker node has few impact over the network of super nodes if it is processing an independent task, that is, without being involved with other worker nodes. Thus, if a worker node gets interrupted and departs the network, the corresponding super node will wait until a given timeout expires, before integrating the presumably interrupted task in the pool of *ready to schedule* tasks. The timeout can be the *Time To Live* (TTL) defined for the task, but to achieve a finer granularity in the detection of lost tasks, a *Time To Report* (TTR) can be defined, with TTR significantly lower than TTL. The TTR, which is associated to the task, defines the periodicity for the worker to report to its super node the status of the task that it is currently processing. The TTR can therefore be seen as a worker/task heartbeat, allowing for early detection of abandoned tasks, that is, tasks whose workers no longer appear active.

Tolerating the departure of super nodes. To preserve the state and data kept at a super node, a periodic checkpoint needs to be taken by the super node itself. To avoid confusion with the checkpoints taken at the workers', we identify the checkpoint of the super node as $SN_{checkpoint}$. This checkpoint holds the state of the super node, namely the list and state of the tasks that are being processed by worker nodes that are attached to the super node. Note that the state of an individual task can easily be known if checkpoint sharing of tasks is enabled, since on receiving a checkpoint taken at a worker's, the super node is updated about the current state of the task. Since $SN_{checkpoint}$ only contains some states of the super node, the checkpointing mechanism can be implemented at the application-level.

To tolerate the departure of a super node from the network, the checkpoints taken at the super node's are replicated to, at least, a nearby super node. Over time, several successive versions (say 1,..., N) of checkpoints holding the super node's state and data will exist at neighbor nodes. An important note is that at the super node level, checkpoints are organized per super nodes. For instance, the super node SN_A might be checkpointed at time t_{SN_1} with the checkpoint replicated to the super node SN_B, while SN_B's checkpoint and the consequent replication might occur at time t_{SN_2} to another super node, say SN_C.

To avoid cluttering disk space with checkpoints which are no longer useful, every super node's checkpoint should be timestamped with a TTL. Whenever this TTL expires, the checkpoint should be deleted by the holding super node and any global state referring the location of the checkpoint should be appropriately updated. Note that a checkpoint stored at a super node might be deleted before its TTL expires, for instance if the task is signaled as completed, or if the storing super node needs to claim back storage space.

When the departure of a super node is detected, the workers that were attached should try to connect to the super node that holds the most recent checkpoint of the now unreachable super node (say version N). This way, and if the connection is successful[7] the recovery procedure can be bootstrapped, in order to allow for on line restoration of state and data, and to resume regular operations. However, an important issue arises: how can a worker node locate the most recent and stable super node's checkpoint? To address the *last stable checkpoint location* issue, we propose two solutions: a *distributed hash table*-based (DHT) one and a *neighbor-based* one. We detail both solutions in the next sections.

A DHT-based network of super nodes

Under this approach, a DHT is formed exclusively by super nodes, with each node holding part of the key space. Since the DHT holds the location of super node's checkpoints, the keys are obtained by hashing the checkpoint IDs, which are themselves comprised of the super node's ID – each super node has a unique ID – concatenated with the checkpoint's sequential number (as seen before, checkpoints are numbered sequentially, from 1 to N).

[7]The super node holding checkpoint N might also be unreachable and thus checkpoint $N-1$ needs to be reached, and so forth.

When a super node exports its current checkpoint to a neighbor node, it inserts in the DHT, under the key yielded by the hash of the checkpoint ID, the location of the checkpoint replica (indicating the host's name or its IP address), a message digest of the checkpoint, and the date/time stamp. In addition, the super node also updates a DHT's key which holds the sequential number of the just saved checkpoint. This key, which results from the hashing of the string formed by the super node ID suffixed with a meaningful string (for instance, "LAST") allows any node to request the sequential number of the last checkpoint saved by the super node, and consequently to retrieve the DHT's key that holds the location of the super node's last saved checkpoint.

A key from the DHT is removed when the corresponding checkpoint is no longer valid (its TTL has expired) or the corresponding super node has ceased its functions. In this latter case, all keys pointing to checkpoints of the super node need to be removed. Note that the management of keys – insertion, update and removal – is performed by super nodes, while worker nodes can solely lookup the DHT. Moreover, instead of a single big-sized DHT, several small ones can exist, each one grouping a set of super nodes and worker nodes, all being independent of one another. This DHT-based approach is similar to the path followed in Chapter 6, except that the DHT only involves super nodes and thus is way smaller.

Whenever a super node departs the network for a period of time longer than a given threshold, the worker nodes that are connected to it are left without super node. Thus they need to reconnect to a super node, ideally to the one that holds the last saved checkpoint. We call this stage *recovery mode*. The recovery mode protocol is detailed in Figure 7.3 and works as follows[8]:

1. The worker nodes consult the DHT to obtain the location of the super node that holds the last saved checkpoint. To avoid a burst effect to the DHT that would occur if all workers entered simultaneously in recovery mode, each individual worker needs to pause for a random variable amount of time before starting its recovery mode.

2. A worker tries to connect to the super node that holds the last checkpoint. If the connection is successful, the worker requests a *restore*

[8]The circled numbers of Figure 7.3 correspond to the stages of the protocol.

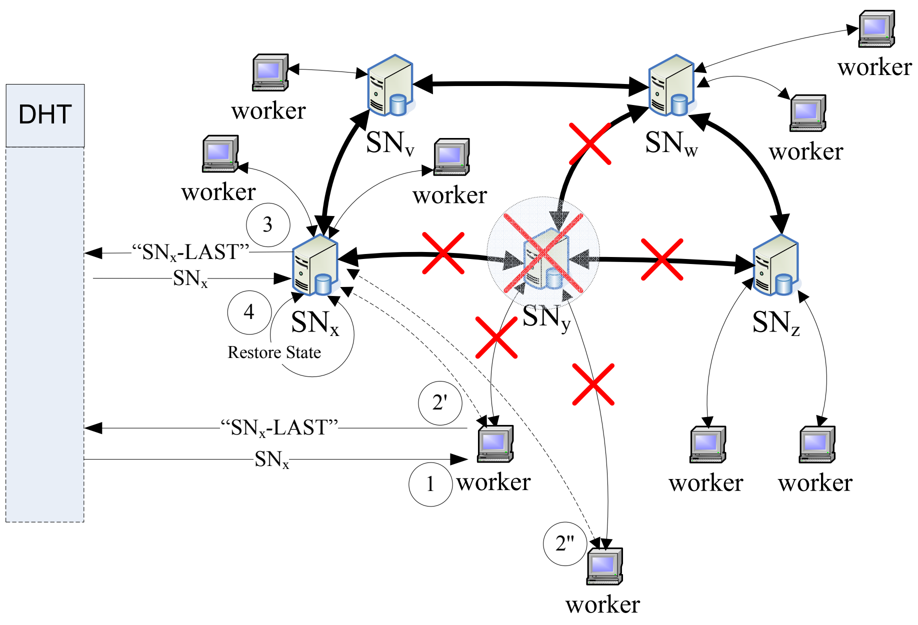


Figure 7.3: Recovery Protocol for the DHT-based Approach.

operation, that is, the restoration of the state of the departed super node. This is done through the checkpoint that is stored at the now contacted super node.

3. Before restoring the requested state, the super node checks whether it really holds the most recent checkpoint of the departed super node. It performs this check by looking up the appropriate key in the DHT. This verification is done to rule out improper restore operations that might be (1) requested by workers that somehow obtained outdated information from the DHT and (2) to discard malicious worker nodes possibly interested in triggering a denial of service.

4. If the super node finds out that it really holds the last available version of the checkpointed state of the departed super node, it will restore it. Requests of further worker nodes will then be accepted. On the contrary, if the super node finds out that it does not hold the most recent version of the checkpoint, it can either ignore subsequent *restore requests*, or redirect them to the super node, which according to its knowledge, hosts the most recent checkpoint.

An issue arises when the super node registered at the DHT as storing the most recent checkpointed state of a departed node has itself left the network or has disposed of the checkpoint without properly synchronizing its status within the DHT. Under these circumstances, and after having failed to reach the super node which should be holding the most recent checkpoint version (say version N), the worker nodes will then attempt to contact the super node holding the $N-1$ checkpoint version. If available, the super node for $N-1$ erases the DHT entry regarding the N^{th} checkpoint version and proceeds for restoration as explained before. Likewise, if the super node for the $N-1$ version is itself unreachable, the recovery should proceed with $N-2$ and so on, until either a proper restore operation can be performed or no checkpoint can be found.

A neighbor-based network of super nodes

We describe an alternative to the DHT-based scheme for locating the most recent and available checkpointed state of a departed super node. Specifically, we resort to a message- and neighbor-based scheme that we call *Message and Neighbor Scheme* (MNS).

Similarly to the DHT-based scheme previously described, under MNS, a super node checkpoints its state when it deems it as necessary and exports the resulting checkpoint (say version I), via the network of super nodes, to a neighbor super node. However, instead of registering the checkpoint's location under a DHT maintained by the super nodes, the super node sends to the holder of its checkpoint I, the list of super nodes that hold the former checkpoint versions that are still active, that is, those checkpoints whose TTL have not yet expired (for instance, version $I-1, I-2, \ldots$). This list – *checkpoint location list* (CLL) – is also forwarded to any node (worker or super node) that contacts with the super node. This way, the location of the most recent checkpoint is slowly diffused to the interesting parties, that is, the worker nodes that are connected to this super node, and to the neighbor super nodes that hold former versions of the super node's state.

When a given super node goes down, the recovery protocol works as follow (see Figure 7.4, where the circled numbers correspond to the different stages of the protocol):

1. Every worker node that was attached to the now departed super node will seek to connect to the super node that has the most recent check-

pointed state of the departed super node. For this purpose, each worker resorts to its checkpoint locations' list[9] to try to contact the super node listed as holding the most recent checkpoint and asking it to initiate recovery mode for the departed super node.

2. When it receives the initiate recovery mode request, the super node replacement candidate will first check if the original super node is really unavailable. This verification is needed to rule out not only malicious workers, but also workers whose network might be fragmented leaving them unavailable to reach the super node[10].

3. If the original super node is in fact unreachable, the super node replacement candidate will itself try to contact the other super nodes which appear on its version of the CLL, notifying them that it will initiate recovery mode of the departed super node, unless one of these super nodes has a more recent version of the checkpoint.

4. If a more recent version exists at one of these peer super nodes, the super node replacement candidate drops its intention of restoring the checkpointed state, and acts as a *forwarder*, that is, it will forward every *requests for recovery* to the super node that has a more recent checkpoint. On the contrary, if no more recent version can be found, the super node waits for a given period of time, and then restores the state of the departed super node. In this case, the other super nodes that were contacted in the previous step will themselves forward any worker node's recovering mode request they might receive to the super node used for restoration.

7.4.2 Costs of Replicating Checkpoints

We briefly discuss the factors that influence the costs caused by the methodologies that provide fault tolerance to the super nodes. Indeed, replicating checkpoints represents an overhead that needs to be balanced against the profits of recovering a super node's state. We focus on the (1) content of checkpoints and on (2) the frequency of checkpointing.

[9]Note that workers can have different versions of the CLL, since a worker only receives an update when it contacts its super node. Thus some nodes maybe aware of version I, while other ones may not.

[10]Under such situation, the super node itself will see the worker node as having departed.

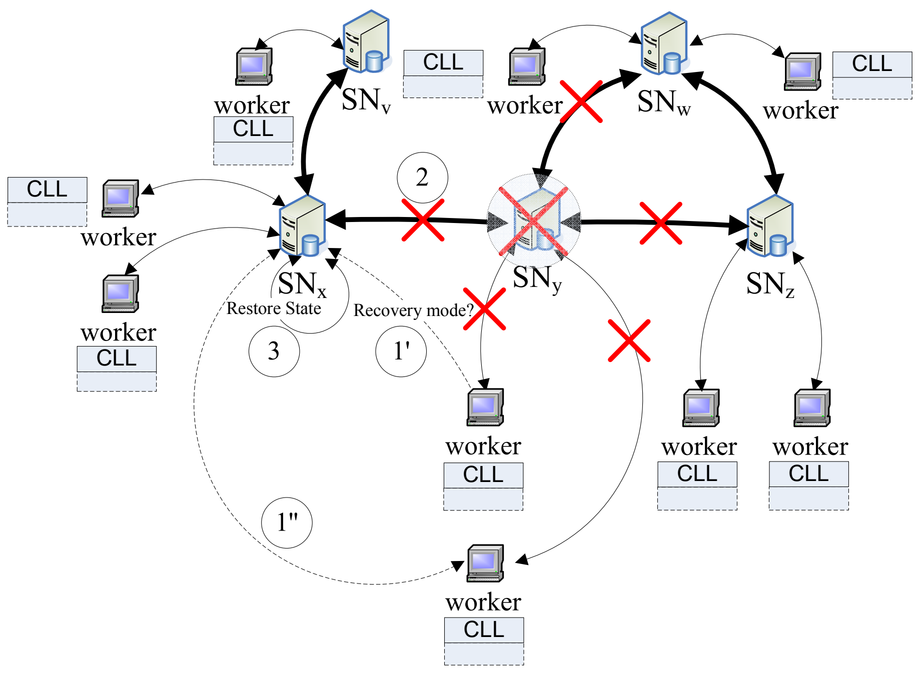


Figure 7.4: Recovery Protocol for the NMS-based Approach.

Content of a Super Node's Checkpoint

An important factor relates to the content of a super node's checkpoint, which should be minimized to lighten storage space and to preserve network bandwidth when checkpoints are replicated. Therefore, a super node's checkpoint will be comprised of the individual checkpoints of the tasks that are being computed by worker nodes which are attached to the super node. The checkpoint will also include some *metadata*, such as the checkpoint ID, the super node's ID and the time-to-live. Note that to further preserve bandwidth and storage space, each checkpoint of an individual task should be compressed by the worker before it is replicated to its attached super node. The task's checkpoint will only get decompressed when and if the task is resumed at another worker node. This way, storage space and CPU are preserved at the associated super node.

It should be pointed out that the input data sets needed for a given task are not included in the individual checkpoint of the task, and thus are not part of a super node's checkpoint. The rationale behind this approach is that input data sets can be obtained either from other super nodes or, as a last resort, from the master server. Instead, the list of input data sets held at a given super node can be sent along the replica of a checkpoint. This eases the lookup of a given input data set, making possible that a super node obtains input data sets from another super node.

Frequency of Replications

The frequency of replication of checkpoints impacts the overhead induced by checkpointing: a high frequency loads the network, while a low checkpoint frequency might reduce the usefulness of checkpointing. This is valid for both task's checkpoints and for super node's checkpoints. However, it should be pointed out that a super node is more important than a worker node, since the failure of a super node impacts all worker nodes that are attached to it, while the failure of a regular node only hinders the task being executed at the worker.

Regardless of the role of a node, the replication frequency for checkpoints should be linked to two factors: (1) the failure rate of the node and (2) the speed of change of the node's state. Indeed, nodes with low failure rates can be checkpointed at lower rate relatively to more instable nodes. This means that the checkpoint frequency should be set per node, depending on its average availability and on the rate of change of the node's state. Moreover, as super nodes are mostly selected on their availability, this means that the replication rate of the checkpoint of a super node can be low. In fact, if a super node's availability declines, the super node will be demoted to the rank of a regular node. In this way, if super nodes are properly selected, the replication mechanism should cause few impact on resources.

7.5 Related Work

We now review related work focusing on P2P-based and derived systems that exploit wide-scale volunteering resources.

The *Chessbrain* project [chessbrain 2007; Frayn & Justiniano 2004] is an example of a distributed application that departs from the independent task model. As the name implies, this project implements a distributed chess application, upon which workers analyze possible solution sets of a chess play, returning their results to the master node as it is usual in volunteering projects. However, contrary to other projects, Chessbrain has a soft real time constraint, since the time for a whole game of chess is accounted for and limited. Thus the workers' results need to be sent back within a short time frame, else they are useless. A key achievement of Chessbrain was performed when the project managed a draw in a game against an human grand master chess in 2004. However, during this attempt, it was found out that the centralized approach was reducing scalability, with only around 2000 workers having effectively contributed to the game, while many others where not allowed to contribute due to the communication clogging on the main server [Frayn *et al.* 2006]. Therefore, the project has since taken a federated approach to cope with its soft real time demands, seeking to aggregate resources which are near to one another in what the authors call *cluster nodes*. For that purpose, a framework termed *MsgCourier* is being developed. MsgCourier requires every participant peer node to communicate with a cluster node, instead of directly interacting with the master node[11]. No details are available on the way how peer nodes can locate and contact peer nodes, except that the cluster nodes should be chosen from trustworthy ones [Frayn *et al.* 2006], possibly from a list of known volunteers. Our proposal for institutional desktop grid based on *local proxy servers* is similar to the MsgCourier's approach, since both aims to reduce bandwidth demand at the server-side, although the LPS approach is more generic.

Although KaZaA [Liang *et al.* 2006] and Skype [Guha *et al.* 2006] are not computing-oriented (that is, they do not aim to exploit CPUs cycles), both of these platforms are pioneer and successful examples of the usage of unrelated volunteer peer-to-peer nodes over the Internet. The former is a file-sharing applications, while Skype is a well-know VoIP solution. Both exploit network of super nodes, although in KaZaA, the transition of a node to a super node role is only performed if the node's owner authorizes it. On the contrary, in the Skype network, the change to super node requires no prior consent from a node's owner. This feature has draw critics and fears, mostly due to the network bandwidth and computing resources a super node role might impose, but has the merit of avoiding non-cooperative behavior, namely selfish conduct that could affect the overall network strength[12]. As noted before (section 7.4.1, page 159), in a network

[11]The documentation for MsgCourier designates the *master node* as the *super node*. We do not use this term to avoid further confusion.

[12]Users that benefit from P2P systems, but that do not contribute to them are called *free riders*.

of volunteer workers such as SNBDG, the conversion to a super node can only be done with the consent of its owner, otherwise most resource owners will simply stop volunteering their resources. Both KaZaA and Skype rely on strong communication and data encryption to minimize the possibility of interference from outside applications in their networks[13].

The OurGrid project [Cirne *et al.* 2006], whose scheduling algorithm *workqueue-with-replication* we mentioned in Chapter 5, aims to attract computing laboratories from any place to create a global community of shared resources. For this purpose, institutions that join OurGrid grant access to their own resources, receiving in turn the permission to use the idle time of other laboratories' resources. Regarding the interaction of the machines, OurGrid is based on a peer-to-peer network, where each participating institution/laboratory corresponds to a peer in the system. To cope with free riders, OurGrid resorts to the *network of favors*, a decentralized resource allocation and accounting scheme, not based on money. In the network of favors, a *favor* corresponds to the allocation of a computing resource to a peer that requests it, and the value of that *favor* is the amount of work done for the requesting worker. Specifically, each peer keeps locally track of the total value of favors if has given to and received from each peer it has interacted with. Whenever an idle node is requested by more than one peer, the requested node calculates a local reputation value for each of the resource seekers, selecting the one whom it owns most favors. To prevent malicious behavior from resource owners or/and application submitters, OurGrid isolates resources from applications through sandboxing via the hypervisor-based virtual machine Xen system [Barham *et al.* 2003]. Regarding fault tolerance, and as reported in Chapter 5, OurGrid only supports bag-of-tasks applications, and no provision, besides overscheduling, is done to cope with node failures.

As reported earlier in Chapter 2, the Personal Power Plant (P3) is another peer-to-peer-based framework to scavenge idle cycles [Shudo *et al.* 2005]. Similarly to the OurGrid project, the framework, which is still in development, allows any node to submit tasks for execution. However, contrary to other P2P environments, P3 offers support for communication between tasks through both an object passing and a message passing programming libraries, effectively supporting parallel tasks. For communi-

[13]Note that the KaZaA network is completely independent of the Skype's network.

cation and organization of the nodes, P3 resorts to JXTA [Verbeke *et al.* 2002], which provides for the network overlay, organizing the nodes in JXTA-supported peer groups called *job groups*. P3 is built around two main daemon programs: *host* and *controller*. The *host* daemon is to be run by resource providers, and its main function is to execute the tasks it receives from its job group's controller. Conversely, the *controller* daemon, besides being used for submitting jobs, also controls the hosting machines. Indeed, at each host machine, a so called *host daemon* connects the machine to its group's controller daemon. A major issue with P3 lies in the overhead that is imposed by JXTA on communications. Additionally, no provision seems to exist relatively to fault tolerance when a parallel-enabled application is executed, while result verification for independent tasks is provided through redundancy.

Other peer-to-peer based systems include the *Personal Grid* (PG) [Han & Park 2003], which we briefly presented in Chapter 2. PG's main goal is to allow that any individual can have her own multi-task application(s) executed over the volunteered resources. PG has no central control, implementing a peer-to-peer model. Specifically, the proposed system explores a network of super nodes, calling *cluster* to a set of worker nodes that is connected to a same super node. The aggregation of a worker node to a given *cluster* (a worker node is only connected to a single super node) is determined by the network proximity: on the prototype implementation, two nodes are considered close if they are reachable through a link level broadcast, and thus belonging to a same local network. Since any node can submit an application to be executed over a network of workers, PG has a mechanism to match the needs of the applications to existing resources. Indeed, to submit a task, the submitter node releases an *advertisement* to the network. This advertisement, which holds a meta description of the task (URL of the needed files, message digest codes, etc.) is sent to the super node which then forwards the metadata through the network of super nodes and so on. To avoid flooding, the advertisement is limited by a TTL. When terminated, the results are sent back to the submit node. PG only aims to allow the execution of independent tasks, providing no support for communication, nor synchronization among workers, even if they are located in the same cluster. In addition, and contrary to the server-based model where only a central and credible entity can release tasks, PG is prone to malicious submitters and thus the security of both resources and of results are important open issues.

7.6 Summary

In this chapter, we analyzed two hierarchical desktop grid models oriented toward a better exploitation of resources: federation of servers and P2P overlay of workers. The former model organizes the workers behind a *local proxy server* that acts as the representation proxy of the institutional resources. The *P2P overlay of workers* model targets peer nodes which have no special affinity with each other, apart being close in network terms. For the P2P environment, we propose a super node-based infrastructure, where more reliable, available and resource rich nodes act as super nodes, coordinating the data and task distribution of the worker nodes that are connected to them.

As soon as the fault tolerance issues are solved, P2P-based infrastructures will play an important role in desktop grids, as they already do in the area of file sharing. P2P-based computing will allow not only for the sharing of data and checkpoints, but also for the execution of tasks that require communications among workers.

8 Sabotage Tolerance through Comparisons of Checkpoints

Redundancy is commonly used by public resource computing middleware for the validation of final results. Simultaneously, checkpointing is used for fault tolerance purposes. In this chapter, we merge the redundancy model with checkpointing of intermediate execution points. Specifically, the signature of checkpoints from intermediate execution points of redundant tasks are compared with each other to detect possible errors. With this methodology, errors can be caught earlier. This yields faster execution and higher confidence in the application results.

8.1 Introduction

The correctness of computations that are performed over volunteer resources is a major issue in desktop grids. Indeed, if a worker sends back some wrong results, it may undermine the whole computation.

A possible source for incorrect results is faulty hardware. Anderson cites *overclocking* as a significant cause of faulty computations in projects that resort to the BOINC framework [Anderson 2004]. Another cause for erroneous results is motivated by the somewhat fierce competition among volunteers who dispute the top places of the contribution ranks elaborated by the project managers [Bohannon 2005b]. In fact, some volunteers try to increase their credits resorting to dishonest tricks to collect undue credits. For instance, they might have their volunteered machines return fabricated results that require no or minimal computation, or even resend results instead of performing the honest computations [Molnar 2000]. This type of users is known as *lazy cheaters*. A *saboteur* is another type of users who acts maliciously for the sole purpose of ruining the computation, without concern for credits, nor any other direct benefices [Sarmenta 2002]. In contrast to lazy cheaters, saboteurs may be difficult to spot since they may be resourceful and committed to perform everything they can to disrupt the computation.

Commonly, desktop grid projects resort to redundancy as a sabotage-tolerance technique [Du *et al.* 2004]. Under this approach, the same task is distributed to $2M-1$ distinct and independent worker machines to avoid collusion. When completed, results are compared and there is a majority vote. If a result has majority, that is, at least M tasks return a same result or an equivalent one[1], it is interpreted as the correct one and the task is flagged as completed. On the contrary, if no consensus can be found, all results are discarded and the task is marked for rescheduling. The requirement for executing $2M-1$ replicas of a task over distinct and independent machines stems from the need to avoid the tainting of results. This can occur if a faulty or malicious worker executes more than one replica. For example, for $M=2$ (i.e., 3 replicas per task), if two replicas are executed by the same faulty worker, then the majority result will be the one reported by the faulty machine, that is, the incorrect one.

In this chapter, we present a checkpoint and replication-based error detection technique that simultaneously exploits checkpointing and redundancy. The technique compares intermediate checkpoint digests of redundant instances of a same task. If differences are found, we conclude that at least one execution went wrong. In contrast to the simple redundancy mechanism, where diverging computations can only be detected after a majority of tasks have completed (we call this approach "compare-at-end"), the comparison of intermediate equivalent checkpoints allows for earlier detection of errors, since divergences among the replicated executions can be spotted at the first checkpointing operation that occurs after an error. This allows one to take proactive and corrective measures without having to wait for the completion of the tasks, therefore allowing for a faster task completion, since tasks spotted as faulty can immediately be rescheduled.

[1]Some projects dependent on floating-point operations might have slightly different results when executed in different platforms, but yet equivalent from the project point of view [Taufer *et al.* 2005a].

Furthermore, the checkpoint-based comparison technique makes feasible the setup of correctness tests, upon which the correctness of a worker is evaluated without its knowledge. For that purpose, a task whose intermediate checkpoint digests are known by the supervisor (having being validated by previous executions) is dispatched to the worker under assessment. The worker is certified if the returned checkpoint digests are correct. On the contrary, if it fails the test, it can be placed in a blacklist, or, at least, the results that it produces will need to be more thoroughly checked.

To complement the error detection methodology based on comparison of equivalent checkpoints, we propose a checkpoint-based replication technique whose goal is to promote the fast completion of redundant replicas of a same task, in order to speed up the validation of results. Specifically, under the proposed technique, the replication of a redundant replica is scheduled as soon as the replica is determined to be erroneous or lagging behind, comparatively to other replicas. To minimize the computation to be re-executed, the technique tries to initialize the replica from an already validated intermediate checkpoint. The technique extends the checkpoint-based verification, promoting a balanced execution of redundant replicas, since validation can only occur when a majority of results have been completed. Moreover, since execution credits are given to workers only after results have been validated, this also accelerates validation and proper credit assignment, which is an important issue for a vast percentage of volunteers [Holohan & Garg 2005].

The organization of this chapter is as follows. First, we review validation methods for results computed over volunteer desktop grid resources. Second, we construct a theoretical model that estimates the benefit of comparing intermediate checkpoints as a function of the probability of task error and checkpoint frequency. Third, we propose the use of immediate replacement of erroneous or slowly executing tasks to prevent further delays in the execution of tasks and in the validation of results. Fourth, we conduct simulations and analysis of results using our novel approach, which confirms the benefits estimated by our theoretical model. Finally, we compare our approach with related work.

8.2 Results Validation Techniques

In this section, we review the most common techniques for validation of the results computed by volunteer desktop grid resources. Specifically, we analyze *replication with majority voting*, *spot-checking* and the generic *credibility-based* approach.

8.2.1 Majority Voting

The *majority voting* method detects erroneous results by sending identical tasks to multiple workers. After the results are returned, they are compared. If they are identical, or at least a majority of results are, the result is assumed as being the correct one. Sarmenta determines the amount of redundancy for majority voting needed to achieve a bound on the frequency of voting errors given the probability that a worker returns an erroneous results [Sarmenta 2002]. Considering φ as the probability that a worker is erroneous and returns an incorrect result, and that ε is the percentage of final results (after voting) that are incorrect. Let r be the number of identical results out of $2r-1$ required before a vote is considered complete and a result is decided upon. Then the probability of a incorrect result being accepted after a majority vote is given by:

$$\varepsilon_{majv}(\varphi, r) = \sum_{j=r}^{2r-1} \binom{2r-1}{j} \varphi^j (1-\varphi)^{2r-1-j} \quad (8.1)$$

From Equation 8.1, Sarmenta shows that the error rate decreases exponentially as long as φ is kept under 50%. So voting is especially effective when the error rate is small. However, when the fault rate is relatively large, increasing redundancy does not significantly reduce the error rate. For example, when $\varphi = 20\%$, the error rate is still more than 1% when $r = 6$. Figure 8.1 plots the frequency of voting errors in function of φ, for the r replication factor sets, respectively, to 1, 2, 4, 6 and 8. Note that $r = 1$ means that no replication is done, and thus the probability of error is solely dependent on the worker. Furthermore, while increasing the redundancy level lowers the probability of an erroneous result being accepted, with φ above 50% the reverse occurs, that is, the probability of erroneous results escaping the majority voting detection system increases exponentially.

Parameter	**Definition**
φ	Probability that a worker is erroneous and returns an erroneous result
ε	Fraction of results after voting that will be erroneous
r	Number of identical results out of $2r-1$
f	Fraction of hosts that commit at least one error
q	Frequency of spot-checking
m	Number of temporal segments
p_e	Probability of having a computational error in any of the checkpoints
c	Number of segments, or equivalently, checkpoints per task ($c = m$)
T	Total time of the computation
W	Time elapsed from the occurrence of an error up to its detection (random variable)

Table 8.1: Parameter definitions.

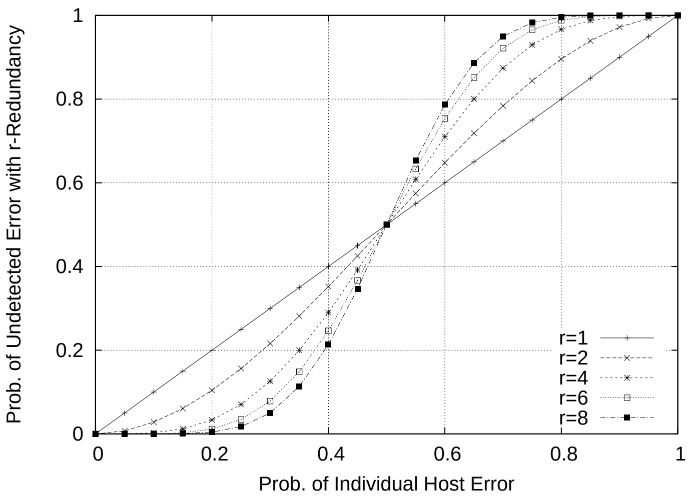


Figure 8.1: Probability of undetected errors on majority voting.

The redundancy of majority voting is $\frac{r}{1-f}$, where f is the fraction of hosts that can commit at least one error. The performance impact of replication can be huge. Indeed, even the lightest replication scheme, that is, $r = 2$, cuts the performance of the entire system in half. Another potential drawback to this method is that it is susceptible to correlated failures, as the bounds computed for ε_{majv} assume that error occur independently among hosts. However, if hosts collude together often and conduct a coordinated attack, majority voting may not be efficient. In conclusion, majority voting is effective when the host error rate is relatively small, below 1% and the error behavior of hosts is independent from each other, that is, no collusion exists among workers. Despite its high resource consumption, n-replication is widely used in Internet-based projects, such as SETI@home and Einstein@home[2], since it is a straightforward method to implement, and generic enough to support any applications, as long as an appropriate results comparator is provided. In fact, n-replication is supported natively in the BOINC framework [Anderson 2004].

8.2.2 Spot-checking

Spot-checking is another method for error detection [Sarmenta 2002]. It consists in the random distribution of tasks with a known result to worker, as a way of testing them. The returned results are compared with the ones that are known to be correct, and any discrepancies cause the corresponding worker to blacklisted, i.e., any past or future results returned from the erroneous host are discarded (perhaps unknowingly to the worker).

The main advantage of spot-checking is that the amount of redundancy computation can be set to be negligible, specially when compared to the majority voting method. In particular, the amount of redundancy of spot-checking is given by $\frac{1}{1-q}$, with q representing the frequency of spot-checking.

The disadvantage of spot-checking is the difficulty of effectively blacklisting an erroneous host, when it can register under new anonymous identities at will (as we shall see in chapter 9), or if hosts have high churn rate as reported by Anderson and Fedak [Anderson & Fedak 2006]. Moreover, blacklisting may be harmful if it removes from the project workers that un-

[2]As previously stated in Chapter 2, in its stage S5, the Einstein@home project reduced the minimum validation quorum from 3 to 2 in order to diminish the computing power lost to replication.

intentionally and infrequently return invalid results. Nonetheless, without blacklisting, the spot-checking technique is practically ineffective.

8.2.3 Credibility-based Validation

Another way of reducing the acceptation of erroneous results as correct ones is to use conditional probabilities of errors. This approach analyzes the ratio of correct results computed by a given host in the past to assess the credibility of the host. A system based on this principle is called a *credibility-based* system [Sarmenta & Hirano 1999]. The idea is based on the assumption that hosts that have computed many results with relatively few errors have a higher probability of producing errorless computation than hosts with a history of returning erroneous results. Thus, the credibility of an host increases with correct results. Tasks are assigned to hosts such that more attention is given to the tasks distributed to higher risk hosts. To determine the credibility of each host, any error detection method such as majority voting, spot-checking, or various combinations of the two can be used. The credibility values are then used to compute the conditional probability of a result's correctness. As such, this method, like spot-checking, assumes that the error rate per host remains consistent over time. Sarmenta and Hirano conclude that a method that combines voting and spot-checking (using voting also for spot-checking) is the most effective way of using credibility.

A credibility-based approach is also taken by Taufer et al. [Taufer *et al.* 2005b] with workers being classified by their availability and reliability according to their past behavior, that is based on the tasks they have previously computed (or attempted to compute). Specifically, workers are dynamically classified into four classes: *High Available/High Reliable* (HA/HR), *Low Available/High Reliable* (LA/HR), *High Available/Low Reliable* (HA/LR) and *Low Available/Low Reliable* (LA/LR). A *high available* worker is one that has a high probability of being available to process tasks. Conversely, a *low available* worker is seldom available for desktop grid computing. A *high reliable* worker is one whose computations are dependable, that is, there is a high probability of its results being correct. However, the purpose of the author's approach is to properly schedule tasks in order to improve turn-around time, and it is not expressly related to sabotage-tolerance.

8.3 Assumptions and Definitions

We assume a wide-scale computing project, where a central supervisor coordinates the whole computation by distributing tasks to requesting volunteer worker machines. The tasks that comprise an application are sequential and independent from each others. Furthermore, we assume that all communications occur exclusively between workers and the supervisor, and more precisely, communications are worker-initiated in order to circumvent Internet asymmetries caused by NAT and firewall schemes [Son & Livny 2003]. Thus, the supervisor is passive in the sense that it can only answer to worker requests. As seen in earlier chapters, this communication model is the one adopted by several desktop grid frameworks such as BOINC and XtremWeb [Fedak *et al.* 2001; Anderson 2004].

At the worker level, fault tolerance is achieved through application-level checkpointing [Silva & Silva 1998]. We only consider tasks which can individually be broken into m temporal segments $S_t = \{S_{t_1}, \ldots, S_{t_m}\}$. The intermediate computational states can be checkpointed at the end of each temporal segment, yielding the checkpoint set $C = \{C_1, \ldots, C_m\}$, with C_m taken at the end of the computation. Like in current desktop grid middleware platforms, whenever a task is interrupted, its execution can be resumed from the last stable checkpoint C_j. Note, that in this chapter, we do not consider task migration nor checkpoint sharing.

Depending on the application, checkpoints can get quite large, in the range of tens to hundreds of megabytes in size, and thus it might be inefficient to transfer and compare them. (For the purpose of comparison, all checkpoints need to be on the machine that effectively performs the comparison; thus at least one of them has to be transferred.) Thus, for comparison purposes, we assume that message digests of checkpoints (provided by the MD5 [Rivest 1992] and the SHA-family [Eastlake & Jones 2001] algorithms, for example) can be used. Due to their reduced and predictable dimensions, message digests can be easily exchanged and compared. Furthermore, an application-specific preprocessing function might be deployed to normalize checkpoints (for instance, for removing task-dependent identifiers) prior to the use of a generic digest algorithm. For the purpose of comparison, checkpoint C_j is represented by the message digest $\mathrm{MD}(C_j)$. Additionally, the comparison of checkpoints needs to be executed between what we term as *equivalent checkpoints*, that is, checkpoints from

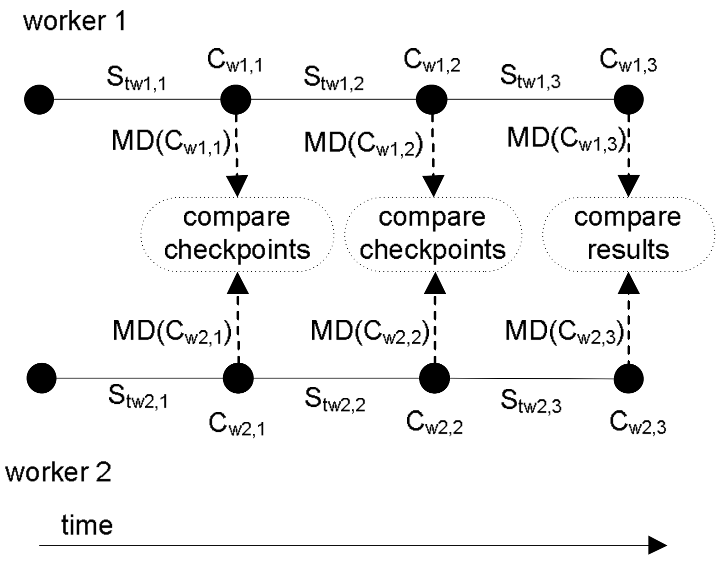


Figure 8.2: Three-segmented execution of a task by two workers with comparison of intermediate checkpoints and results.

different replicas of a task that represent a same execution point of the task. Figure 8.2 exemplifies two workers executing a three-segment task ($m = 3$), with comparisons of intermediate checkpoints and of the final results.

Regarding redundancy, we assume that the system executes each task $2r - 1$ times, resorting to $2r - 1$ independent workers, with the supervisor applying majority voting to validate results, electing the so-called *canonical result* [Anderson 2004]. Afterward, when the result verification is completed, the system assigns the proper credits to the workers that have returned correct results.

8.4 Comparison of Equivalent Checkpoint Digests

For the comparison of equivalent checkpoint digests, a worker is requested by the supervisor to return, along with the results of the task that it computed, a selected set of message digests of the checkpoints saved during the computation of the task. The list of checkpoints whose message digests are requested is defined during the task creation, so that redundant replicas of a task share the same set of requested checkpoint digests.

When a majority of redundant executions are completed, and the supervisor holds enough results for meaningful comparisons, the checkpoint digests from equivalent execution points are compared to one another. If the digests are different, the execution point with the divergent digests is marked as suspicious. Comparatively to the *compare-at-end* methodology, the selective digests technique allows for a finer grain detection level, since an erroneous computation can be located right after the first divergent checkpoint, enabling the identification of the execution segment where the divergence occurred.

8.4.1 Reducing the Time to Detect an Error

Although the selective digests strategy allows for a more precise location of error occurrence, since the segment where the error occurred can be identified, comparison of checkpoints by itself does not speed up the detection of incorrect computations, since error detection can only occur after, at least, two redundant replicas have terminated.

A more proactive variant is to have workers returning available checkpoint digests during the computation. Ideally, from a detection point of view, the worker should send back to the supervisor a checkpoint digest immediately after its computation. This way, an error can be spotted by the supervisor as soon as a majority of checkpoint digests is available for the considered execution point. Thus, upon detection of a divergent computation, corrective measures can immediately be triggered by the supervisor. For instance, the execution of another replica of the task can be scheduled. Additionally, the apparent faulty worker can be marked as a suspect and further probed to assess its computational validity, or if repeating a faulty behavior, it can be blacklisted altogether [Sarmenta 2002].

Additionally, on failure detection, the supervisor can notify the worker to cancel its current task computation. Note that, although commendable for a honest worker victim of a faulty hardware, the cancellation order might also alert smart malicious workers that their dishonesty have been spotted. This way, malicious users might gain valuable insights about the sabotage detection capabilities of the supervisor and thus have access to an effective way to probe the project detection mechanisms. Furthermore, even if the supervisor issues a cancellation order to the worker, this order can only reach the worker in the next communication phase (which is initiated by the worker). This is a consequence of the worker-initiated communication model. Thus, we assume that detected failures are dealt silently by the supervisor, with no notification issued to faulty workers.

8.4.2 Theoretical Analysis

In this section, we conduct an initial analysis of the advantage of detecting erroneous computations at intermediate checkpoints. The goal of this analysis is to estimate the potential benefits of our approach.

As stated earlier, we assume that the execution of a task is segmented into m fragments. Additionally, we make the following simplifying assumptions:

Assumption 1: machines are homogeneous, as well as segments. A task always requires T time units to complete, and each segment takes T/m time units to execute. The probability of obtaining a wrongly computed segment is the same for all the workers and for all segments of the same task;

Assumption 2: all replicas of a task start at the same time across all workers;

Assumption 3: the errors are independent of each others, and thus, no contamination of replicas occur, meaning that comparison of replicas is enough to catch all the errors.

Although these assumptions may seem too restrictive, we show experimentally in Section 8.6 that our analysis also holds for other more heterogeneous scenarios. We will focus on two variables that affect the system: the probability, p_e, of having a computational error in any of the checkpoints[3] (either due to a computational mishap or malicious behavior) and the number of checkpoints of the task. We consider that results are validated through r-replication, with all replicas having to compute the same equivalent checkpoint digests. However, comparison of equivalent intermediate checkpoint digests allows partial validation at point j as soon as the r replicas of a task have sent back their respective message digests of checkpoint j, that is, $\mathrm{MD}(C_j)$. We compare this new and improved approach against the state-of-the-art method, which can only detect errors at the end of the execution.

[3]We use the designation *segment* and *checkpoint* interchangeably, referring to the segment and to the subsequent checkpoint which is taken at the end of the segment.

When the computational error occurs before the first validation checkpoint (C_1), the checkpoint comparison method will anticipate the detection $T \cdot \frac{m-1}{m}$ time units sooner than the regular methodology. This case occurs when there is one or more errors in the computation of all the r replicas. The probability of this event is $1-(1-p_e)^r$, which we denote as p to simplify. For the next checkpoint (C_2), the comparison of equivalent checkpoints saves $T \cdot \frac{m-2}{m}$ time units, relatively to the normal validation method. This happens with probability $p \cdot (1-p)$. Extending this reasoning to checkpoint i yields a saving of $T \cdot \frac{m-i}{m}$ with probability $p \cdot (1-p)^{i-1}$. (In the last segment, when $i=m$, or if there is no error for the whole computation, our approach brings no benefit since in both cases the error detection only occurs at the end.) We let W be a random variable to represent the error detection time, that is, the time elapsed from the occurrence of an error up to its detection. In other words, if we reschedule the task as soon as the error in the checkpoint is detected, W represents the maximum time that we can save, relatively to the compare-at-end approach, with a single error detection. However, in the regular strategy, the computation time can be even worse than $T+W$, because other errors can delay the task even further. Hence, if we are able to calculate W, we can have a measure of the advantage of detecting errors by comparing intermediate checkpoints. To calculate the expected value of W, we proceed as follows (we omit the probability of not having any error, as there is no gain in that case):

$$E[W] = \sum_{i=1}^{m} \left(pq^{i-1} \cdot \frac{m-i}{m} T \right) = T \cdot p \cdot \left(\sum_{i=1}^{m} q^{i-1} - \frac{1}{m} \sum_{i=1}^{m} i \cdot q^{i-1} \right) \tag{8.2}$$

Where $q = 1-p$. Since $\sum_{i=1}^{m} q^{i-1}$ is a sum of terms of a geometric sequence, its sum is $S_{m-1} = \frac{1-q^m}{1-q}$. We can use standard techniques to compute the second term of the difference.

Consider that $S'_{m-1} = \sum_{i=1}^{m} i \cdot q^{i-1}$. By multiplying S'_{m-1} by q and taking the difference $(1-q) \cdot S'_{m-1}$, we get $S'_{m-1} = \frac{S_{m-1} - mq^m}{1-q}$. Since $p = 1-q$, this yields:

$$E[W] = T \cdot p \cdot S_{m-1} - \frac{T \cdot p}{m} S'_{m-1} = T \left(1 - \frac{1-q^m}{mp} \right) \tag{8.3}$$

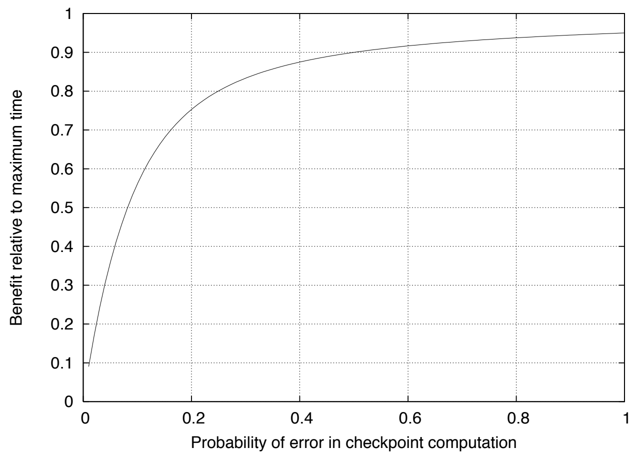


Figure 8.3: Benefit (W) relative to expected maximum time (T) as a function of the probability of error (p).

In Figures 8.3 (page 189) and 8.4 (page 190), we depict the time that we can save relative to T ($E[W]/T$), considering Equation 8.3. Specifically, in Figure 8.3, we set $m = 20$, while in the other figure we set $p = 0.05$. From Eq. 8.3 we conclude that the maximum time that a checkpoint comparison can save converges to T, when $m \to \infty$. When $p \to 1$, the time that we can save approaches $T \cdot \frac{m-1}{m}$ as we would expect. For example, we find that with an error rate of 5% and checkpoint frequency of 20 times per task, the gain is as high as 35% compared to the case where error detection is done only at the end of task execution. Note that this is a conservative estimate of the benefit as the worker software of many projects (such as Einstein@home [einstein 2007] and SIMAP [simap 2007]) have higher checkpointing frequencies (as high as one checkpointing operation per minute). In particular, in the BOINC-based project climateprediction.net [Christensen *et al.* 2005], a task requires around 3 months of CPU time in a fast PCs, being checkpointed 72 times during the whole execution. In conclusion, for even conservative estimates of error rates and checkpoint frequencies, the benefit of comparing digests of intermediate checkpoints is significant, and is even greater for higher probabilities of error or for longer computations with checkpoints.

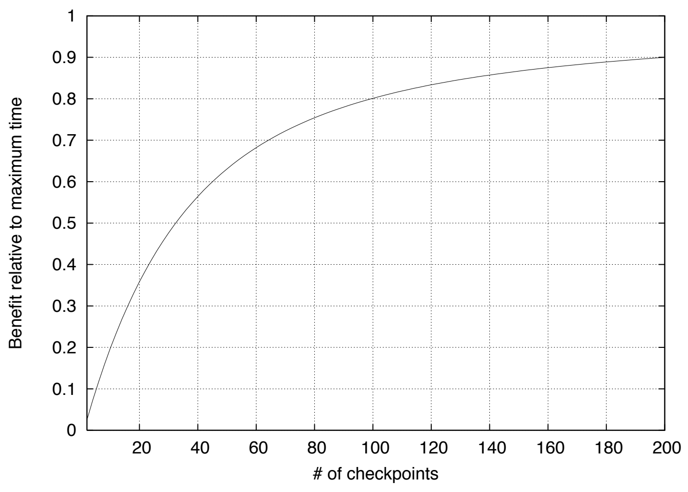


Figure 8.4: Benefit (W) relative to expected maximum time (T) as a function of the checkpointing frequency (m).

8.5 Checkpoint-based Task Replication

Some BOINC-based desktop projects increase, at least for specific period of times, the redundancy level to foster the chance of fast completion of tasks. Surprisingly, one of the main motivation for this important decision, which has important implications on the available computing power of the system as seen in Section 8.2 (page 180), is not directly related to the gain of an higher confidence level for the results, but the need to quickly rewards workers with the proper amount of credits. In fact, credits are only committed to workers after validation of the results. These credits are determined by the supervisor, based on the credit claims made by the intervening workers: jointly with the completed results, the worker sends a claim with the amount of credits it believes it deserves. This claim is computed by the project application running at the client. To circumvent the high volatility of volunteers, a number of replicas higher than what is required for majority voting is scheduled for execution[4]. This provides

[4]That is 3, although a $2+1$ scheme can also be employed – a $2+m$ means that initially 2 tasks are executed, and only if the results are different, than a third one is scheduled for execution. This scheme proceeds until consensus is found or a limit threshold of execution attempts is reached.

timely assignment of credits even in the presence of sluggish and drop-out workers. However, this approach wastes resources, and thus slows down the whole computation.

To speed up completion and validation of individual tasks, thereby promoting fast credit assignment, we combine the comparison of intermediate checkpoint digests with task replication. Furthermore, to prevent lengthy re-computations due to the replication of task, we resort to already validated checkpoints to load execution state in tasks to replicate, avoiding to restart from scratch.

The task replication works by loosely coupling the execution of the redundant replicas of a same task, which are configured for reporting selective checkpoint digests. Note that workers processing replicas are not aware of each other, otherwise the risk of collusion would increase. The supervisor follows the progress of the coupled replicas of a task through the messages holding the checkpoint digests sent back by these replicas, validating the received checkpoint digests of the selected execution point through comparison as soon as a majority of results has been received.

Whenever a worker lags behind its replica partners by more than a specified threshold – the threshold takes into account the relative speed of the workers – the supervisor initiates a replace operation, with the goal of substituting the behind-schedule worker. The rationale behind the task substitution decision lies in the fact that a significant delay is a strong indication that the task got delayed and possibly interrupted, and thus a fast replacement is needed if a quick and balanced execution is sought. To further speed up substitution, the substitute task should start from the last validated checkpoint, if available. To prepare for the replica substitution, the supervisor requests, upon the next communication of a paired-worker, the last validated checkpoint from this worker (not the digest, the entire checkpoint file). Upon receiving it, it checks its validity through message digest comparison, and creates a task which integrates the validated checkpoint file. This *replace task* is then scheduled to a requesting worker, which starts the computation from the checkpoint execution point, thus skipping the computation up to this point. From the point of view of the supervisor, the newly scheduled task replaces the lost/delayed one, and thus the monitoring of execution proceeds as previously explained. Note that, in order to prevent excessive replicas, replication should only be performed if the number of replicas is below a predefined threshold.

8.6 Experimental Results

In this section, we confirm and extend the theoretical results obtained in Section 8.4.2 through simulation. Specifically, we assign a number of tasks to a set of workers, setting the duration of these simulated tasks beforehand. Whenever a worker computes a segment, it randomly determines whether that computation is wrong or correct (once a segment is wrong, all the remaining segments from that worker are also considered as erroneous). The total time of the computation, T, is the time at which the last replica finishes its last checkpoint, regardless of whether it is correct or wrong. Assume that checkpoint C_j was the first one to be wrong and that the last replica finished C_j at time T_W. We are interested in the random variable $W = T - T_W$, which represents the benefit of using intermediate checkpoints relative to the state-of-the-art *compare-at-end*. In particular, the metric we use to quantify the gain compared to the compare-at-end is the relative value W/T.

We started by considering the same parameter settings that were used to generate Figure 8.4. So, we set an uniform $p_e \approx 0.0253$ for all execution segments, considering homogeneous segments, and a two-replica scheme, which corresponds to $p = 0.05$. As expected, in Figure 8.5 (page 193) we got a curve that closely follows the theoretical prediction. Then, we studied the impact of considering different durations for the segments and different error probabilities for each of the computed segments. We used two different random distributions for this: uniform and truncated Gaussian. To maintain consistency, the average values for the error probability and for the segment duration were the same as for the fixed case, p_e and T, respectively. In the uniform distribution, the actual error probability was chosen uniformly from the interval $[0.5p_e, 1.5p_e)$ (which is always inside the interval $[0, 1]$), while the duration was chosen using the same distribution in the interval $[0.5T, 1.5T)$. For the Gaussian distribution, we considered averages of p_e and T, and standard deviations of 30% of the average. Additionally, we truncated the values of p_e and T to be inside the ranges $[0.5p_e, 1.5p_e]$ and $[0.5T, 1.5T]$, respectively. In Figure 8.5, we show the average result of

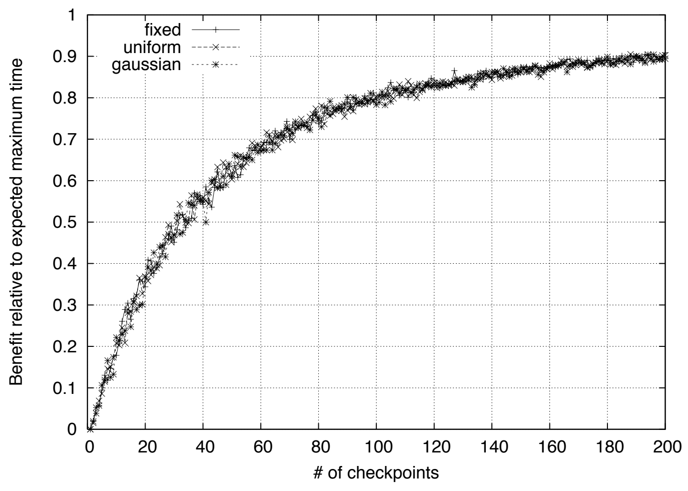


Figure 8.5: Benefit (W) relative to expected maximum time (T) (obtained experimentally).

varying the number of checkpoints for 300 different trials. As we can see, the curves overlap.

The most interesting conclusion from these results is that the particular random distribution that controls the duration and the errors of the segments does not seem to make any significant difference, at least for the same averages. This would not be true if, for instance, the average duration of segments i and j was different for segments i and j[5]. We believe on a simple and intuitive reason for this: on average the slowest replica should finish checkpoint i around time $i \cdot \frac{T}{m}$, where T is the time at which the slowest replica finishes the task. Although some particular cases may not follow this trend, our experimental results confirm this intuition for the average case.

[5]However, note that it would not make much sense to consider different average durations for different segments, unless we were targeting a particular application with a well-known behavior.

8.7 Related Work

In this section, we review related work, namely checkpoint-based approaches to error detection.

Antonelli et al. [Antonelli *et al.* 2004] proposed a distributed checkpoint-based technique for sabotage-tolerance. Similarly to our approach, the scheme addresses sequential computation that can be split in multiple consecutive temporal segments. To certify a given checkpoint C_j, the supervisor creates a verification task. This task references the checkpoint to verify and holds the network contact details of the worker which performed the computation. The task is then assigned to a worker node (*verifier*), which uses the network contact details to request the checkpoint from the worker being scrutinized. After receiving the checkpoint, the verifier loads it and executes the task up to the next checkpoint, that is, C_{j+1}. It then computes the message digest of the checkpoint and sends it as a result to the supervisor. Finally, the supervisor compares the digest to the other equivalent digests. This distributed scheme is appealing since it distributes the computation needed for verification of checkpoints through the workers. However, some major issues are not addressed by the authors. For instance, the scheme requires worker nodes to be directly addressable for checkpoint transfer, thus restricting many nodes that are behind firewalls and NATs to participate. Furthermore, workers need to keep some of the checkpoints of the computed tasks and transfer them when requested, a demand that might require meaningful space storage and network bandwidth, especially with large individual checkpoints. Additionally, the direct checkpoint transfer assumes that the worker is always available, which is seldom the case in a volatile environment like desktop grid. Finally, promoting direct contact between workers may foster collusion.

Agbaria and Friedman [Agbaria & Friedman 2005] proposed a replication and checkpoint-based scheme to detect intrusions through anomaly spotting. They resort to checkpoint comparison for the purpose of identifying intrusions in a Byzantine environment. Similarly to our approach, the execution is split in n sequential phases, with a checkpoint being taken by every worker node at the end of every phase. To support a maximum of t intruded nodes (each node executes a replica), the proposed scheme requires $t+1$ replicas when no intruded node exists. However, when intrusion exists, the protocol needs additional stages, involving more than the $3t + 1$ replicas which would be required by a straight Byzantine agreement protocol. The unbalance is justified by the fact that intrusions are rare and thus it compensates to have a lightweight scheme which is only penalized when intrusions do occur. The protocol distinguishes between *workers* (nodes that perform the computation and which can get intruded) and *auditors*, which are responsible for assessing the integrity of the workers. Specifically, the auditors are used to agree that all the $t + 1$ replicas match. A major demand of the protocol lies in the required synchronization, with workers having to send their checkpoints to the auditors within a given time frame. This requires that the replica execution occurs simultaneously, a premise that might be hard to fulfill in a volatile environment such as desktop grids. Furthermore, the checkpoints (or equivalently, a message digest) need to be sent to the auditors at the end of every stage, an operation that requires communication resources and might be difficult if auditors are not directly addressable [Son & Livny 2003]. Relatively to the solution that we propose, our emphasis is more on the practicality of the error detections schemes and its integration with current desktop grid frameworks.

Wang et al. [Wang *et al.* 2003] resorted to the comparisons of system checkpoints for narrowing down configuration failures in the management of Windows-based systems. Specifically, when a suspected to be configuration failure occurs, a comparison between a pre-failure checkpoint and the current state is performed to try to locate the cause of failure.

8.8 Summary

In this chapter, we proposed a strategy for early detection of errors by comparing equivalent checkpoints of redundant tasks executed over unreliable desktop grid resources. We developed a theoretical model that estimates the benefit of using intermediate checkpoints given a task length and task segment error rate.

The main results of this chapter are the following:

- Through simulations, we found that for a 5% error rate and a checkpointing frequency of 20 times per task, the gain for the execution time is as high as 35% relatively to the traditional approach where error detection is only performed at the end of the execution of a task.

- The benefits increase with higher checkpointing frequencies and/or if the computing environment has an higher error rate. On the other hand, smaller checkpointing frequencies and/or less error prone environments yield smaller benefits, if any.

In conclusion, the checkpoint compared methodology presented in this chapter is appropriate for error-prone environments. In the next chapter, we address the issue of attracting reliable and good performing resources for volunteer computing.

9 Reputation and Trust Management in Volunteer Computing

In this chapter, we focus on reputation and trust management for volunteer computing. We start by briefly analyzing key issues surrounding unique identification of users over the Internet, namely under which conditions such identification is possible, and how these conditions fit into public computing projects and their resource donors. Then, we focus on reputation systems for public computing, proposing a volunteer invitation-based mechanism for recruiting trusted resource donors. Specifically, we propose and analyze an invitation-based reputation system for public computing that we call *Invitation System*. This system mimics social behaviors to build up a reputation network, with participation in a given public computing project requiring an invitation from an already active donor. To promote responsible invitations, inviters are linked to the behavior of their invitees, being rewarded with credits proportional to their invitees' computing effort – the more the invitees produce, the bigger is the inviter's bonus.

To complement the Invitation System, we outline a simple reputation certification, upon which a volunteer can apply for an invitation to a desktop grid project by presenting credentials from public projects where the volunteer has contributed to. In this way, reputation credentials can be shared across projects.

9.1 Introduction

In trust relationships developed through the Internet, like the ones that occur in public computing – both project owners and resource donors should trust each other – reputation systems can be important for setting initial trust level [Resnick *et al.* 2000]. For instance, the *FeedBack Score* rating mechanism of the popular auction site *eBay* [eBay 2007] has already proved that a reputation system, even as simple as the +1/0/-1 evaluation system of transactions[1], is important to promote deals between persons that do not know each other [Resnick *et al.* 2000].

Reputation systems play an important role because they collect, distribute and aggregate feedback about the behavior of participants and help to decide whom to trust, implicitly encouraging trustworthy conducts. Regarding fault tolerance of volunteer desktop grid systems involved in public computing projects, the existence of reputation systems can diminish the requirements of the fault tolerance mechanisms, especially regarding detection and control of malicious users. For instance, supported by appropriate reputation systems, fault detection systems can focus on resource donors with no or low reputation, instead of spreading uniformly its efforts over the whole population of resource donors [Sarmenta 2002].

9.2 The Problem of Identity

In his seminal work, Douceur proved that environments where users (labeled as *identities*) can assume multiple identities are prone to *Sybil attacks*, unless a logically centralized authority exists [Douceur 2002]. Indeed, by impersonating multiple identities, it become feasible for malicious users to gain control of a substantial part of the system[2] [Hu *et al.* 2006]. Therefore, uniquely identifying a volunteer participant is one of the most serious challenges faced by trust management systems for public computing.

Commonly used attributes like email addresses and IP addresses offer few, if any, guarantees of persistent identification. For instance, a malicious

[1]An eBay's user can classify the transaction she had with another user assigning a +1 (satisfied), 0 (so-so), and -1 (unsatisfied). The rank of a user is given by the arithmetic sum of all the users that have classified her.

[2]The *sybil* designation stems from the 1973's book "Sybil" written by R. Schreiber. In this book, which is based on the author's life, the main character suffers from *multiple personality disorder*, impersonating multiple identities.

user can easily and quickly create an email account in one of the many free email providers for the sole purpose of anonymously engage into a volunteer computing project. When, and if the malicious user behavior is caught by the sabotage-tolerance system and the corresponding email address is placed in a blacklist, the malicious volunteer can quickly create a new email account, and rejoin the project under a new and unsuspected identity.

Likewise, IP addresses are not suited for unique and persistent identification of users, since most hosts are not directly connected to the Internet. Rather, Internet access is performed through ISP's or corporate's firewalls, possibly with a masqueraded and private IP address that can vary periodically. Therefore, under such dynamic conditions, IP addresses are meaningless for identification purposes. Furthermore, mobile computing devices like laptops allow their owners to easily connect from any place they might be, further invalidating reliable identification through IP address, since IP addresses change accordingly to the geographical location and to the Internet provider. On top of that, resourceful users can also resort to Internet anonymity services such as the *The Onion Router* (TOR) [Dingledine *et al.* 2004], which allows access under non-traceable IP addresses. Similarly, *Medium Access Control* (MAC) addresses that could be used by ISP to uniquely identify the networking hardware can also be spoofed. Concluding, both IP and MAC address cannot serve as a meaningful identification mechanism, at least on systems where users cannot be trusted.

OpenID[3] has recently emerged as distributed identity framework aiming to act as a single point of authentication for services accessible over the web [Recordon & Reed 2006]. OpenID relies on the concept that anyone can identify themselves on the Internet through an URL. Thus, to login to an OpenID-enabled website, the user only needs to type her OpenID URL. The website will then query the user's OpenID provider to checkout the validity of the given authentication. If the authentication credentials are accepted, the user will then be allowed in the website, being properly identified. This way, OpenID allows for single-sign-on, upon which the user only needs to authenticate with an *OpenID identity provider*. Although OpenID is still a work in progress, the fact that it has been endorsed by ma-

[3]http://openid.net/

jor computer players like VeriSign, Microsoft and IBM, to name just a few, gives some positive prospects about its future.

Ironically, although an important issue, unique and reliable identification of users, if at all possible, also raises major privacy issues as the Pentium III's unique identifier flaw demonstrated some years ago [Schneier 1999; McCullagh 2000]. Indeed, when Intel announced the inclusion of an unique identification ID on every Pentium III processors for the purpose of promoting more responsible transactions over the Internet, a chorus of protests emerged from end user associations worried with the privacy and freedom implications of such identification scheme. In consequence, the ID feature was dropped[4]. Another example of an universal identification system that has often be criticized and failed to gain acceptance is the Microsoft Passport [Passport 2007][5].

Unique identification schemes might discourage honest volunteers, not only because of the burden that unique identification schemes would probably impose, but also for the loss of privacy they might represent for volunteers, or at least be perceived as such. After all, volunteers are providing added value to the system, with practically no tangible reward in exchange. Therefore, should a valid and verifiable identification be required for volunteering resources, it would probably demote many potential volunteers to donate their resources. Thus, in this work, we rely on the traditional identification system, based on email addresses, knowing that it does not guarantee uniqueness of users. However, apart from saboteurs who aim at disrupting the volunteer projects without gaining any direct rewards, all other volunteers, including lazy cheaters, have strong interests in keeping their identity throughout the project, in order to keep adding up the credits they gain by processing workunits.

9.3 The Invitation System

In this section, we present, to the best of our knowledge, a novel approach based on invitations to build network of honest and devoted volunteer nodes for public computing. The system resorts to participating volunteers

[4]In practice, the processor identifier is still available as a BIOS option, which is off by default.

[5]The failure reasons for the Passport system were not solely due to low adhesion of users, but they were also caused by e-commerce web sites refusing to adopt Microsoft technology over the concern that Microsoft could create a huge database of users.

to recruit other workers, rewarding or penalizing the recruiters accordingly to the performance and accuracy of their invitees. We name the system, Invitation System, or IVS for short.

9.3.1 Overview

The Invitation System mimics human social relationships to create a trustworthy community of volunteers, with volunteers inviting other users to volunteer resources, implicitly vouching for their guests' trustworthiness. In IVS, a user can only enroll as a volunteer in a desktop grid project through an invitation sent by another volunteer who is already contributing to the project.

Invitation System resorts to email addresses for identifying users and communicating with them. For IVS, an invitation is merely a specially crafted communication string sent to an invitee's email address, and which allows the invitee to register in the public computing system. Although it is weakened by the aforementioned email identification fragilities, this scheme eases the setup procedure reducing it to a few emails exchange and some input from the invitee[6]. Note that this email-based registration approach is similar with what currently happens in the registration process of the major public computing projects.

To insure that invitations are made in a rationale way, inviters are rewarded or penalized accordingly to the behavior of their guests. The goal is to make the inviters, up to a certain level, responsible for the behavior and performance of the participants that have joined through their invitations. Under the IVS approach, a volunteer participant who has proved her honesty, worthiness and commitment to the public computing project by having properly computed the tasks it was assigned to, is granted with a certain number of *invitations cards* that she can distribute to known users who want to join the volunteering network.

The contribution of a participant is evaluated through the amount of work performed for the project. This is measured in credits, while the accuracy of results can be assessed through sabotage tolerance measures such as replication, as previously seen in Chapter 8. To motivate volunteers to recruit participants via their invitation cards, inviters receive a bonus given

[6]The input is needed to prevent automatized registration schemes associated with self-invitation, as we describe in Section 9.3.7.

by a profit function $W(x, n)$, where x is related to the computing contribution achieved by participants that have enrolled through their invitations, and n corresponds to the link depth between inviter and invitee, as explained later on Section 9.3.3. Reciprocally, when an invitee returns an incorrectly computed results, the inviter is penalized by the withdrawal of $L(x, n)$ amount of credits. The goal of the reward/penalty mechanism is to motivate volunteers to carefully choose the users they invite to join the volunteering network: a good invitation yields credits, while a badly chosen invitee provokes loss of credits.

A basic outline of a new worker joining the volunteer project through an invitation is given in Figure 9.1. Specifically:

1. The invitee receives the invitation.
2. The invitee requests its activation to the project supervisor.
3. The supervisor registers the new worker.
4. Next, a regular work cycle follows, with the new worker requesting a task.
5. The worker receives the task.
6. The worker processes the task.
7. Having completed the task, the worker sends the results back to the supervisor.

After having properly processed some tasks and earned the corresponding credits, the worker might receive some invitation cards, while the original inviter is awarded with $W(x, 1)$ bonus credits, with x corresponding to the credits earned by the invitee node, and $n = 1$ indicating a direct link between the inviter and the invitee.

Although the reliance of IVS over credit rewards and penalties for motivating responsible behaviors might seem fragile, the importance of the credit-based rewarding system for public computing cannot be understated. In fact, although in most projects credits are merely virtual, and do not translate in any tangible asset, a significant number of volunteers donate their resources primarily for the thrill of earning credits and to compete with other volunteers [Holohan & Garg 2005]. To spur the competition

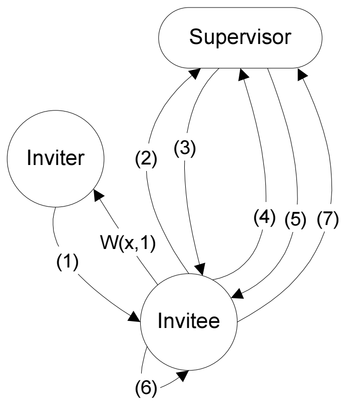


Figure 9.1: Workflow of the Inviter-Invitee Relationship.

spirit, most volunteer computing projects maintain a publicly accessible classification of donors, with volunteers ranked by the amount of credits they earned[7]. Indeed, many volunteers are more focused on the classification competition than on the scientific goal that is being tackled by the computing project. Anecdotal evidences have been reported of volunteers engaging in freshly created projects having as main motivation the fact that an early participation strengthen their chances of toping the project's credit ranks. These credit-driven volunteers are sometimes referred as *crunchers* [Bohannon 2005b]. Further evidence of the credit importance is demonstrated by the vigorous complains expressed in the user forums which are linked to the most popular public projects (like *SETI@home,Einstein@home* and alike), about credit related issues such as accountability and delay in the attribution of credits. An additional confirmation of the influence of credit rewarding mechanism is given by the efforts performed by skilled participants to optimize the binary executables and the subsequent rapid adoption of such binaries by volunteers eager to boost their credit earnings, as reported in Chapter 2 (see Section 2.6.2).

Regarding credits, invitation-based systems allow an additional credit-based classification: sorting inviters by the amount of credit bonuses they earned through invited nodes. This might spur the competition among credit-oriented participants, further fostering invitations and the consequent participation in resource volunteering.

[7]See for instance, http://boincstats.com/

9.3.2 Invitation Cards

In IVS, the *invitation cards*, which allow already contributing volunteers to invite other users, are distributed by the system in accordance to the volunteer's performed computation, measured by the number of correctly computed tasks. Specifically, whenever a volunteer[8] completes a given amount of credits (say, $\text{invitation}_{credits}$), the system assigns $\text{invitation}_{cards}$ to the volunteer. The volunteer is then free to send invitation cards to resource owners (henceforth *invitee*) as long as the invitees are not already registered in the public computing project[9].

It should be noted that an invitation card has no physical existence, it is merely a specially crafted URL, with a limited time-validity, which is recognized by the project as an invitation of a member to an invitee. This invitation is sent through email to the invitee. When the invitee accesses the invitation's URL for the first time, a registration procedure is initialized, which if successful, yields a user ID to the invitee, and the corresponding right to volunteer resources to the project. The registration step is similar to what is usual in web-based registration systems, with protection against automated registration and alike (for instance, resorting to a CAPTCHA-based system to defer automated registering [von Ahn *et al.* 2004]).

9.3.3 Relationship Threshold Distance

The relationship distance between two related nodes is the number of generations that separate the two nodes, considering that one is ascendant of the other and thus has either directly invited the second one (direct link) or another ascendant. We designate this metric as the *relationship distance* (RD). Over time, the chain of volunteers evolves, with participants that were once invited, receiving invitation cards to distribute and so on. An interesting issue of the system relates to the link strength, if any, that should exist between inviters and non-directly invited descendants. As the name implies, a non-directly invited descendant is a participant that has received

[8]In this context, a volunteer designates the human who can have several machines engaged in the project under a same *user ID*.

[9]Obviously, an already registered user can always enroll under a newly created ID, using a different email address to receive another invitation. However, from the point of view of the system the two addresses correspond to two different users. Moreover, the user is not able to merge the credits of her two registrations and thus yields no practical benefits from it. We tackle this issue in section 9.3.7.

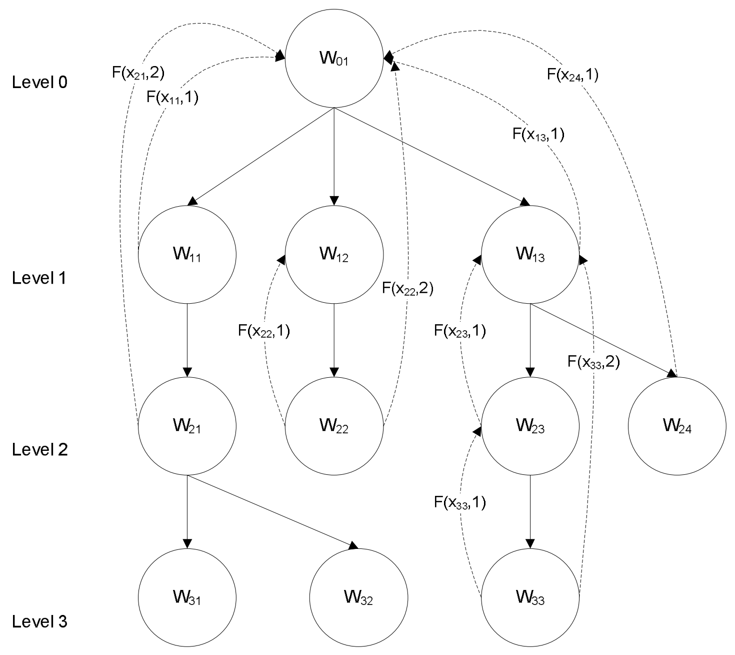


Figure 9.2: An Example of an Invitation System Tree.

an invitation through a former invitee of an inviter, and therefore was not directly invited by the participant. Formalizing, a n^{th}-generation descendant is a user that was invited by a participant that was herself invited by a $(n-1)^{th}$ generation descendant and so forth. The relationship distance is given by n.

The *Relationship Threshold Distance* (RTD) designates the relationship distance to consider for penalty/rewarding. Beyond RTD, invitee-inviters links are not considered. For instance, RTD$=1$ means that relationship is only considered between inviter and direct invitees (i.e., between nodes with single-unit relationship distance). A RTD$=2$ extends the penalty/rewarding relationship to 2-level invitees and so forth.

Figure 9.2 represents three levels of a simple inviter-invitee relationship tree, with a relationship threshold distance sets to 2. For the sake of clarity not all ascendant links are shown. Function $F(x,n)$ designates the reward/penalty function between inviter and invitee. Note that the cardinality of the inviter-invitee relationship is $1:N$, since, while an invitee can only have one inviter, an inviter might have many invitees.

9.3.4 Bootstrapping the Invitation System

To bootstrap the invitation system, that is, to recruit the beginner nodes when a volunteer project is launched, initial invitations need to be sent by project coordinators and alike to credible and willing to participate users. Thereafter, over time, a list of inviters will emerge, as a way to reward the most dedicated participants for their efforts and honesty to the volunteer project. These inviters will then recruit other participants and so on.

9.3.5 Management Overhead

A potential limitation of the invitation system is the overhead that it might induce on the supervisor side of the volunteer project. Indeed, inviter-invitee relationships need to be kept by the supervisor, and the relationship-tree might, especially in a wide-scale project, become unmanageable or, at least, absorb significant resources. Dependent on the RTD, updating the credits of participants might require expensive resources from the supervisor of the project. However, records of nodes' relationships can be limited depending on the RTD defined for the project. Thus a balance is needed, between the overhead to support for managing and updating the relationship tree, and the depth of information that the computing project wants to retain relatively to its volunteers.

9.3.6 Collusion Avoidance

By its knowledge regarding the relationships of worker nodes (*who has invited who*), the invitation system can diminish the possibility of collusions, especially in systems that resorts to redundancy for result verification. Indeed, to preclude collusion, the system scheduler should only distribute instances of a same tasks to non-related worker nodes. This way, the project avoids having related participants in the same voting group.

Although more accurate than the random approach followed by current public computing systems[10], the strength of this collusion avoidance feature is limited by the RTD sets by the system. Furthermore, nodes can always collude by hiding their relationships, resorting to different and non-

[10]For instance, BOINC distributes redundant task randomly, only checking that a worker is not assigned with a redundant instance of a same task that it has previously computed.

related inviters, although this is certainly harder to achieve than it is in open projects (i.e., non-invitation dependent).

9.3.7 Preventing Misuse

A possible misuse of the system would be for a participant who holds several machines to invite herself under a new identifier, trying to benefit for the bonuses conceded to inviters for well-behaved participants. For instance, a user with three machines could, in a first instance only register one machine, and when granted invitation cards, use them to enroll her other two machines under a newly created identities. However, even if the sum of credits achieved by the multiple identifiers of the same user are superior to the credit granted to a single identifier with multiple machines, these credits are spread across multiple identifiers and might not be much fruitful in terms of ranking, except if identifiers are allowed to group as teams. Furthermore, self-invitation of multiple-machine volunteer could be further discouraged by rewarding the volunteering of multiple machines in such a way that self-invitation would not yield additional earnings relatively to multiple machine owners. For instance, an owner with n machines engaged on the desktop grid project would receive an extra amount of credits proportional to n and to the number of computed credits.

A more subtle problem with self-invitation regards saboteurs, that is, resourceful cheaters that are determined to undermine the results of the computation without yielding any real benefit, except for the denial of service they provoke. Saboteurs can resort to successive self-invitation until a registered worker is beyond the relationship threshold distance, and thus no longer connected to the first inviter. Then, this worker can be used to inject fake results. This can be somewhat mitigated by keeping, for the sole purpose of audition, the whole registration link level between inviters and invitees. This means using a somewhat limited RTD for credits rewards and penalties, but an unlimited RTD in order to have a proper registration history. Note, that in order to shorten the registration history, users who have been inactive for a somewhat long period (for instance, one year) could have their registration information deleted from the system. Another issue that might render sabotage through self-invitation less appealing is the fact that the invitation and registration processes require human presence and, thus a saboteur would need to go over a reasonable amounts of tedious self-invitations and registrations to create anonymous accounts for sabotage.

9.4 Relationship Between Inviter-Invitees

In this section, we study the relationship model between inviters and invitees. Specifically, we analyze how the error rate of workers, which we model as a random variable, and the rewarding factors set by the project managers influence the relationship among nodes, namely between (a) inviter and potential invitees, and between (b) inviter and invitees. For this purpose, we change the error rate and the rewarding factors and assess the effects on the recruiter nodes, using as metric the amount of credits that a recruiter can expect to earn. Our approach is based on the assumption that an inviter aims to maximize her earnings, accommodating her behavior to achieve this goal. For this purpose, we propose a theoretical model aggregating several parameters, such as the probability of error of a recruit, governed by the random variable E, the n-RTD between inviter and invitee, and the error penalty function $\theta(i)$. With this model, we aim to quantify the expected credit earning $E[W]$ for inviters and the factors that shape it.

9.4.1 Theoretical Analysis

In an Invitation System, the credits earned by a volunteer node are given by the sum of (a) the self-computed tasks, (b) the bonuses originated by the credits earned by invitees, and (c) the penalties suffered due to erroneous computations of linked invitees. This is represented by Equation 9.1. Since $C_{self\_computed}$ is the sole responsibility of the node and thus it is not influenced by any inviter-invitee relationship the worker might be in, we focus on the other two terms, C_{bonus} and $C_{penalty}$.

C_{bonus} corresponds to the sum of all the bonuses awarded to the node due to the positive participation of its invitees. Specifically, the bonus yielded by an invitee to a n-relationship distance (RD) inviter is given by $W(x,n)$, where x represents the amount of credit earned by the invitee's computation, and n stands for the RD between the node and the recruited worker. Conversely, the penalty induced by a inviter-invitee relationship is given by $L(x,n)$. Defining $\phi(n)$ as the function that returns the counts of

Parameter	Meaning
C_{node}	Total of credits belonging to *node*
$C_{self\_computed}$	Credits earn by computing tasks
C_{bonus}	Credits earn through invitees' bonus
$C_{penalty}$	Credits lost due to invitees' errors
P_{ε}	Probability for the occurrence of an error
RD	Relationship Distance
RTD	Relationship Threshold Distance
$\phi(n)$	Function that returns the counts of workers located at distance n of their inviter
$\theta(i)$	Penalty factors function (i is the number of errors)
$W(x,n,\theta(i))$	Bonus function of a n^{th} level invitee
$L(x,n,\theta(i))$	Penalty function of a n^{th} level invitee

Table 9.1: Parameters, functions and constant definitions.

workers located at a $n-$RD distance of their inviter node, and considering the system-wide constant RTD_{max} as the maximum RD taken into account for inviter-invitee's relationships, Equation 9.1 can be rewritten as Equation 9.2.

$$C_{node} = C_{self\_computed} + C_{bonus} - C_{penalty} \tag{9.1}$$

$$C_{node} = C_{self\_computed} + \sum_{j=1}^{RTD_{max}} \sum_{i=1}^{\phi(j)} W(x_{i,j}, j) - L(x_{i,j}, j) \tag{9.2}$$

To manage the vulnerability of the invitation system to erroneous workers that produce several wrong results in a time frame T, we resort to blacklisting. Specifically, after E_{max} errors for results computed within a time frame T, the faulty worker is blacklisted. Moreover, the penalty function, $L(x,n)$, that affects the inviter's credit when one of its n-RD invitee produces an erroneous result, should be proportionate to the so far contributed gain of the invitee, that is, $W(x,n)$, and should increase with consecutive errors of the worker. Therefore, we model host computing errors in the time frame T through the random variable E, and consider P_{ε} as the probability of the occurrence of an error (i.e., $P_{\varepsilon} = P(E > 0)$), and P_{ε_i} as the probability of an invitee to erroneously compute i results in a T time frame (i.e., $P_{\varepsilon_i} = P(E = i)$). The expected penalty $L(x,n,\theta(i))$ for a recruiter node is given by Equation 9.3, with $\theta(i)$ being a function that yields penalty factors that increase accordingly to the number i of erroneous results. Considering Equation 9.3, the expected credit gain $E[W(x,n,\theta(i))]$ of an inviter with a n-RD invitee is shown in Equation 9.4.

$$L(x,n,\theta) = \sum_{i=1}^{\varepsilon_{max}} P_{\varepsilon_i}.W(x_i,n).\theta(i) \tag{9.3}$$

$$E[W(x,n,\theta(i))] = (1-P_\varepsilon).W(x,n) - \sum_{i=1}^{\varepsilon_{max}} P_{\varepsilon_i}.W(x_i,n).\theta(i) \tag{9.4}$$

We now study the effects of P_ε and the associated P_{ε_i}, as well as the importance of the penalty function $\theta(i)$ over the expected credit gain of an inviter. Specifically, we determine the P_ε threshold of a potential worker relatively to $\theta(i)$, above which it becomes unprofitable to invite a worker.

We set ε_{max} to 3, meaning that at the third error within a time frame T the invited worker is blacklisted, with all her results being discarded. Additionally, we assume that $P_{\varepsilon_i} = (P_{\varepsilon_1})^i$, and that $P_\varepsilon = \sum_{i=1}^{\varepsilon_{max}} P_{\varepsilon_i}$. Regarding $\theta(i)$, we consider it to be i times the integer constant α, that is, $\theta(i) = i.\alpha$. This way, $\theta(i)$ controls the penalty level for erroneous results. For instance, considering $\alpha = 1/2$ (and consequently, $\theta(i) = i/2$), means that the first error costs the inviter half of the bonus she has earned so far with the erroneous invitee (i.e., $W(x,n)/2$), that a second error induces an additional penalty of $W(x,n)$ and a third one (and final one, since we set ε_{max} to 3) costs further $W(x,n).3/2$ credits. This means that an invitee delivering 3 erroneous results within the considered time frame T will cost its inviter $3.W(x,n)$ credits before getting blacklisted[11]. The cumulated penalties relatively to various α penalty factor for the linear model are given in Table 9.2 (page 211).

The ratio $E(x,n,\theta)/W(x,n)$, which corresponds to the gain that an inviter can expect from an invitee, is plotted in Figure 9.3 (page 211) with P_ε varying between 0.0 and 1.0, and α sets to 1/4, 1/2, 1, 2 and 4, respectively. It can be seen that even in the most penalizing scenario, which occurs for $\alpha = 4$, the expected gain turns negative solely for probability of errors near 15%. This error rate is high, corresponding roughly to an average of one

[11]It should be noted that $W(x,n)$ increases between errors, that is, the worker produces valid results between errors. Thus, to be more precise, we should consider $W_i(x,n)$ as the gain yielded by the worker until the i-th error.

α	**1st error**	**2nd error**	**3rd error**	**Cumulated**
$\alpha = 1/4$	$W(x,n)/4$	$W(x,n)/2$	$3W(x,n)/4$	$3W(x,n)/2$
$\alpha = 1/2$	$W(x,n)/2$	$W(x,n)$	$3W(x,n)/2$	$3W(x,n)$
$\alpha = 1$	$W(x,n)$	$2W(x,n)$	$3W(x,n)$	$6W(x,n)$
$\alpha = 2$	$2W(x,n)$	$4W(x,n)$	$6W(x,n)$	$12W(x,n)$
$\alpha = 4$	$4W(x,n)$	$8W(x,n)$	$12W(x,n)$	$24W(x,n)$

Table 9.2: Penalty factors yield by varying α in the linear model.

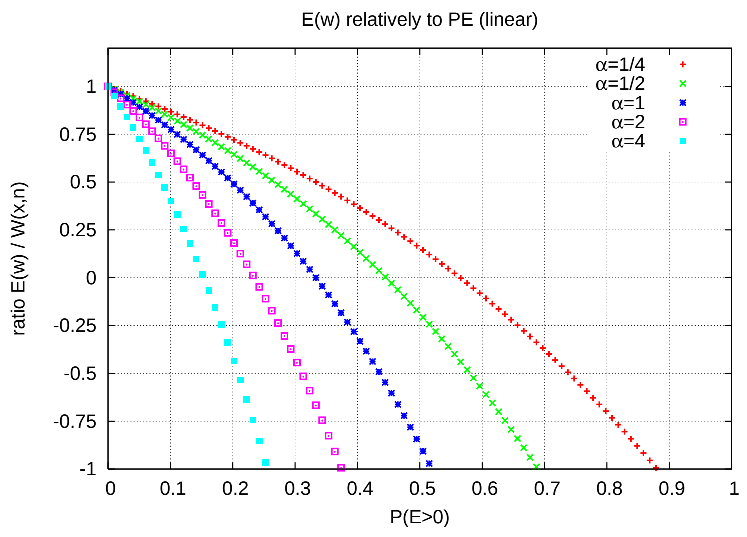


Figure 9.3: $E(x,n,\theta)/W(x,n)$ ratio for the linear penalty model.

bad result out of seven. This might indicate that either the machine is defective or the user is maliciously messing up with the results. Anyway, in Invitation System the worker should be quickly blacklisted for committing ε_{max} errors, with the corresponding inviter harshly penalized in $24.W(x,n)$ credits. The plot also shows that smaller values of α result in less penalty. For instance, with $\alpha = 1/4$ the expected gain for an inviter only becomes zero for error rates bigger than 55%. This can be seen with more accuracy on Figure 9.4 that plots the $E(x,n,\theta)/W(x,n)$ ratio yielded by varying α from 0 to 4, for P_ε =1%, 5%, 10% and 20%. From the plot, we can observe that, as expected, higher error rates yield higher penalties.

To further assess the effects of $\theta(i)$ over the expected gain, we set a cubic penalty model, upon which $\theta = i^3.\alpha$. Comparatively to the linear model, this penalty scheme is interesting in the sense that the penalty for first error

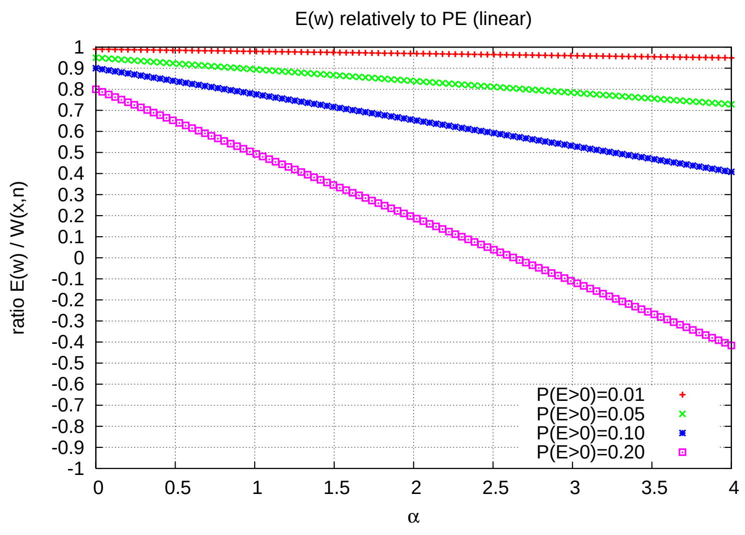


Figure 9.4: Effects of varying rates on the penalty (linear model).

α	**1st error**	**2nd error**	**3rd error**	**Cumulated**
$\alpha = 1/4$	$W(x,n)/4$	$2W(x,n)$	$27W(x,n)/4$	$9W(x,n)$
$\alpha = 1/2$	$W(x,n)/2$	$4W(x,n)$	$27W(x,n)/2$	$18W(x,n)$
$\alpha = 1$	$W(x,n)$	$8W(x,n)$	$27W(x,n)$	$36W(x,n)$
$\alpha = 2$	$2W(x,n)$	$16W(x,n)$	$54W(x,n)$	$72W(x,n)$
$\alpha = 4$	$4W(x,n)$	$32W(x,n)$	$108W(x,n)$	$154W(x,n)$

Table 9.3: Penalty factors yield by varying α in the cubic penalty model.

remains the same (that is, $i.\alpha$), but increases sharply for subsequent errors, thus penalizing inviters whose workers commit more than one error. The cumulated penalties relatively to various α penalty factors for the cubic model are given in Table 9.3. Figure 9.5 plots the $E(x,n,\theta)/W(x,n)$ ratio for the cubic penalty model.

In conclusion, adopting $\theta(i) = i^k.\alpha, k \in \aleph$ yields smooth penalties, with the negative impact to the inviter's gains being more noticeable when the corresponding invitee commits multiple errors. This fosters inviters to seek reliable invitees in order to profit from the inviter-invitee partnership.

Relatively to invitees, the fact that a single casual error is lowly penalized means that a single error will not cause a major hurdle to the inviter-invitee relationship. However, abnormally high error rates (for instance, above 10%) augments the probability of multiple errors and thus trigger

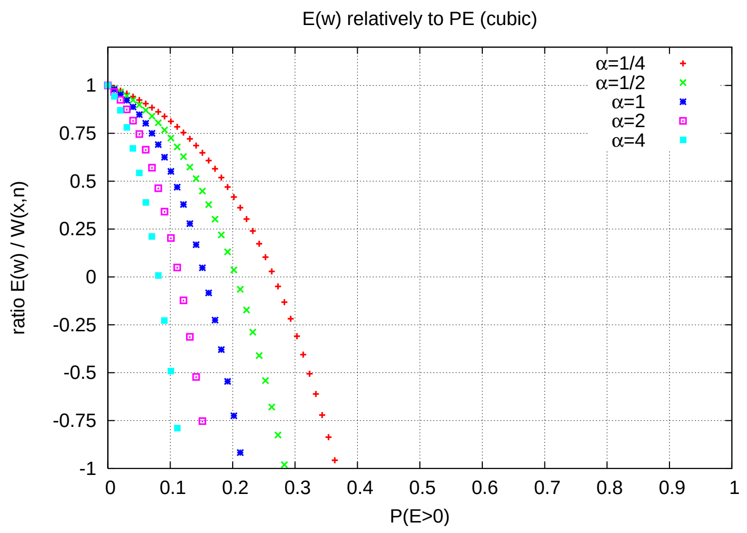


Figure 9.5: $E(x,n,\theta)/W(x,n)$ ratio for the cubic penalty model.

harsher penalties. In fact, high error rates mean that either the machine or its configuration is faulty or that the computation is being sabotaged by the machine's owner. Either way, the invitee is providing no added value to the project and thus should be withdrawn from the computation, with the inviter accordingly made responsible. Note that if the invitee has malicious intentions of perturbing the computations by faking or tampering with results, she will be able to forge a new email identity and seek a new invitation. However, it is expectable that the malicious worker will soon run out of inviters, exhausting her contacts and thus being forced out of the project.

9.5 Sharing Reputation Across Volunteer Projects

The invitation-based system can be extended so that it supports commendation of participants across multiple volunteer projects. The basic goal is to allow a volunteer who is already participating in a public project (or has participated in the past), to apply for an invitation in another project presenting as references a virtual certificate provided by the project(s) where she is currently participating (or had participated). This virtual certificate holds the applying worker's performance and trustworthiness metrics,

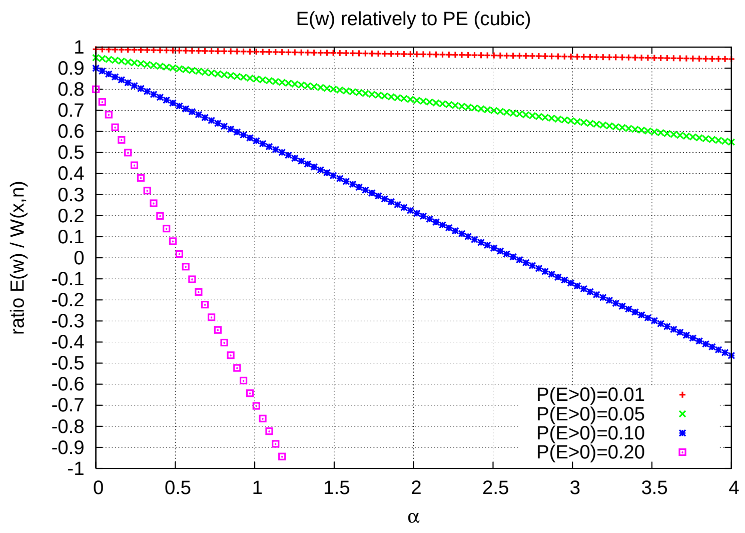


Figure 9.6: Effects of varying rates on the penalty (cubic model).

such as the ratio of successful tasks completed, position in performance rankings, and so on.

Note that the participation of a volunteer in multiple projects is not a novelty, and its actually promoted by the BOINC platform, which allows for a volunteer to donate resources to several projects, specifying the available CPU time distribution to be allocated to each project. For example, a resource involved in three projects, can be set up so that 50% of the CPU time goes for project A, 30% for project B, and the remaining 20% for project C. The rationale for promoting the support of multiple projects, which from the individual point of view of a project might seem counterproductive since the project risk losing exclusivity of resources, lies in the fact that many projects have downtime (for hardware and software maintenance, and reparation of the server infrastructure), and shortage of tasks (for instance, when transitioning from one experiment to another). Thus, participation in multiple projects helps to cope with a particular project downtime, besides allowing volunteers to donate resources for several causes they might find worthy.

9.5.1 Implementation

In terms of implementation, the virtual recommendation certificate could be an URL, unique to the pair participant/project, hosted by the project from which the participant is requesting references. The virtual certificate would be sent to the volunteer's registered email, on request, and would have a limited time validity. Thus, when applying for an invitation to another project, the volunteer participant could attach its reference certificate(s) (the volunteer might already be participating in more than one project). Then, the project from which the volunteer is seeking an invitation could consult the reference certificate(s), analyze the metrics provided there, and decide accordingly whether it should or not deliver an invitation to the requesting volunteer.

The project-based reference certificate has the advantage of being simple, since it only requires a project to setup a secure web services capable of providing participation metrics of a given participant. In fact, the BOINC framework already gives free web access to the work records of volunteer computers. This way, reference certificates should be straightforward to implement. A further benefit of the reference certificate would be to promote that a volunteer uses the same identification (i.e., email address) across all the projects in which the volunteer already participates or has participated. To further stimulate adoption of single identifications across projects, a credit boost (or any other form of reward) could be assigned to volunteers signing up with the same identification across projects.

9.6 Related Work

Invitation-based systems are not a novelty in the Internet. Indeed, the social networking site Orkut [orkut 2007] and Google's email system, Gmail [gmail 2007], were initially only accessible through invitations, requiring an introduction from an already register member. The invitation system allowed to maintain a certain knowledge over the relationships between users, and principally to limit the number of users during the beta testing stage[12]. Contrary to the above mentioned invitation schemes, Invitation

[12]Soon after it was launched, in April 2004, Gmail's invitations were highly praised, and approximately 50,000 of them were actually auctioned and traded in eBay, with prices rising up to 100 US dollars [Musgrove 2004]. This was mostly motivated by the fact that early users of the Gmail service

System strongly connects the inviter with the invitee behavior. In fact, by having the inviter rewarded or penalized accordingly to the invitee's behavior and dedication, our proposed invitation system promotes responsible use of invitations.

Danezis et al. proposed a social-based scheme for managing access to open and partially anonymous publishing, filtering out inappropriate posters by having other users reporting on unacceptable behavior they might spot (for instance, inappropriate or provoking comments, spam, etc.) [Danezis & Laurie 2007]. Similarly to our approach, the system is built-up upon invitations. Whenever an user wants to post, she needs to identify the inviter that introduced her to the system. However, users that want to object a post need to identify their whole system path. This whole path is comprised of all the inviter-invitee chain upon *root*, and allows to uniquely identify an objector at the system-level (the scheme does not require true identity), implicitly making them responsible for their objections. Additionally, if any other user wants to contest the objection, the contester can take responsibility of the post, having her full system path appended to the post jointly with the objector's path. A possible problem that might arise with this methodology is related to the size of the user-root path, which will increase over time. However, this potential problem should rarely happen mostly due to the properties of scale free networks, which are usually exhibited by social-based networks. In fact, in such environments a small percentage of users are responsible for a large percentage of invitations, thereby limiting the size of the user-root path [Watts & Strogatz 1998].

Tribler is a peer-to-peer system proposed by Pouwelse et al [Pouwelse *et al.* 2006]. The system aims to integrate social relationships into peer-to-peer networks. The goal is to explore social groups to promote cooperation over peer-to-peer file sharing, avoiding the pitfalls of today's peer-to-peer file sharing like fake files, virus and malware-infected files. Moreover, Tribler includes support for sharing of preference lists among members ("buddies" in Tribler's jargon). Specifically, preference lists are spread over the members by way of the *BuddyCast* algorithm that resorts to epidemic protocols for efficient data sharing. By default, the preference list of a peer is filled by her last downloaded files. Additionally, Tribler incorporates a

had much higher probability of being able to pick their desired user name, since email addresses were assigned in a first come first served basis.

recommendation engine, which based on a user preference list is able to suggest a list of files that might interest the user. Tribler also promotes faster Bittorrent-like downloads by allowing cooperative download, upon which the idle upload bandwidth of idle online buddies can be used. In fact, to discourage the so-called *free riders*, the Bittorrent protocol requires a balanced upload/download, a demand that might impede users with asymmetrical Internet connections in which the allowed upload traffic is much smaller and slower than the download one. By requesting the non-used upload capability of buddies, Tribler allow users to cooperatively download chunks of a file, and thus promote faster downloads. Under Tribler's model, the main reward for resource donors is social recognition.

9.7 Summary

In this chapter, we exploited the possibilities of strengthening the robustness of results by way of a wiser selection of resource donors, resorting to volunteers' social relationships. Specifically, we propose the Invitation System upon which potential volunteers can only integrate a computing infrastructure by way of invitations addressed by workers that have significantly and positively contributed to the project. To motivate workers to invite other resource donors, inviters are rewarded by receiving a bonus proportional to the credits earned by the invited workers. However, to promote responsible invitations, inviters are penalized whenever an invitee produces an erroneous results by losing credits, with the penalty increasing with consecutive errors of an invitee. To further foster responsibility and motivation of inviters, bonuses and penalties can go beyond the inviter-invitee direct link, with the performance and behavior of n-level descendants of an invitee also impacting, although in a lesser degree, the n-level ascendants' rewards and penalties. Note that the Invitation System still requires a result verification mechanism (such as replication). In fact, the system itself strongly relies on the accuracy of the result validation subsystem for enforcing proper bonuses and penalties.

To complement the Invitation System, namely for inviters to assess potential candidates to invitation, we outline a simple mechanism upon which a want-to-be resource donor can present credentials from her contribution to other public computing projects. This involves public computing projects to make accessible the credentials of their donors, detailing their main positive (tasks successfully completed) and negative contributions (abandoned and erroneous tasks) to the project in question. Thus, whenever a want-to-be resource donor applies for an invitation, she can present her credentials allowing the inviter to form a more precise assessment of the pros and cons of an invitation.

10 Conclusion and Future Work

In this final chapter, we outline the major contributions of the thesis, and we trace venues for future work.

10.1 Conclusions

In this thesis, we have addressed some issues related to a more efficient exploitation of desktop grid resources. This was mostly done through the adaption of dependable and fault-tolerant mechanisms such as *checkpointing* and *redundant computing*, and the proposal of new techniques for sabotage tolerance. A common goal in our work was to speed up the execution of applications, namely to improve turnaround time for submitters. For this purpose, we focused on issues like the reduction of wasted computation, faster detection of faulty computations and on diminishing the level of verification and/or redundancy needed for certifying results.

As confirmed by the 77-day long study presented in Chapter 3, a vast amount of resources goes unused in institutional environments, where many computers that are primarily dedicated to interactive usage like e-office and communication applications have a low level of resources utilization (CPU, memory and disk). In particular, the average CPU idleness tops 97%, while memory usage averages 60%. Jointly with studies that focus on Internet-wide resources, these values emphasize what can be gained from harvesting institutional resources.

In Chapter 4 and Chapter 5 we proposed some methodologies based on shared checkpoints to improve turnaround time over private checkpointing policies, by making checkpoints more accessible. Additionally, when coupled with replication, sharing checkpoints can yield major performance improvements, depending on the volatility of resources and on the length of individual tasks. Specifically, more volatile environments or/and lengthier tasks increase the benefits of shared methodologies over private checkpointing schemes. The studied prediction-based policies also performed well with medium- and long-sized tasks, and were more suited to stable environments.

Another conclusion drawn from this work is that sharing checkpoints over the Internet can significantly improve turnaround time of executions. In fact, as reported in Chapter 6, the *chkpt2chkpt* scheme that promotes checkpoint sharing over the Internet yields good benefits when the rate of tasks interrupted for long period of times (relatively to task's duration) is above 5%. This means that on volatile environments with fail-crash failures and with relatively long tasks (several hours or days) a DHT-based scheme can bring significant performance benefits.

While a DHT-based shared checkpointing scheme yields strong advantages, resilient and scalable implementations of open DHT are practically non-existent, thus rendering difficult a real implementation of *chkpt2chkpt*. Nonetheless, as seen in Chapter 7, desktop grids can organize themselves in federations such as institutional and peer-to-peer ones, with appropriate checkpointing methodologies adapted to these topologies.

Sharing checkpoints are not the sole manner to speed up executions in desktop grid environments. Checkpointed states of partial execution points can be combined with redundant executions – another widely used dependability mechanism in Internet-wide desktop grids – to promote faster detection of errors. Indeed, as seen in Chapter 8, checkpoint-based comparison mechanisms can speed up the spotting of errors, allowing repairing measures (rescheduling of faulty executions) to be activated sooner than it would have been possible with usual schemes, which can only detect errors when a majority of workers has finished their tasks.

Checkpointing is not the sole dependability mechanism whose wise usage can enhance the performance of soft deadline applications. In fact, the *Invitation System* introduced in Chapter 9, fosters the cautious recruitment of volunteers for desktop grid projects by rewarding recruiters with a bonus indexed to the productivity of the recruited hosts. However, to counter invitations to bad performers, recruiters who invite workers that produce erroneous results are penalized with the withdrawal of virtual credits. This simple scheme raises the average quality of volunteers.

10.2 Main Contributions

We now summarize the main contributions drawn from the work presented in this thesis:

1. In Chapter 3, we showed that the idleness level of desktop grid resources is very high, especially for CPU, which for the academic environment that we studied, had an average idleness as high as 97.94%. This confirms the 95% CPU idleness empirical rule. Likewise, the *equivalence ratio* was 0.51, similar to the values found by Arpaci et al. [Arpaci *et al.* 1995] but smaller than those found by Kondo et al. [Kondo *et al.* 2004a].

2. Proper management of checkpoints, especially sharing of checkpoints amongst desktop grid nodes can speed up the execution of applications comprised of independent tasks, by reducing the level of computation to be redone when failures occur and interrupt the computation. This is shown in Chapter 5 for institutional desktop grids confined to local area environments.

3. By resorting to a DHT-based watchdog (*guardian*) and to a checkpoint sharing mechanism that form the *chkpt2chkpt* system (Chapter 6), we have also improved turnaround performances of applications executed over wide-area desktop grids, showing the benefits of Internet-wide sharing schemes.

4. In Chapter 7, we proposed some extensions to the standard model for organizing desktop grid nodes. The proposed model aims to deliver a better usage of resources and broaden the type of applications to be run over such environments. For instance, for institutional desktop grids and for unstructured set of workers spread over the Internet, a peer-to-peer organization with two-level nodes – super nodes and regular ones – can allow for the reuse of input data sets and for checkpoint sharing among worker nodes.

5. In Chapter 8, we showed that dependability techniques like redundant computing can be coupled with checkpointing to allow for a faster detection of erroneous or/and of interrupted executions of individual tasks that comprise a distributed application. In this way, through comparison of checkpoints from equivalent intermediate execution points, errors can be detected at these partial execution points, avoiding the need to wait for full completion or for full timeout, as it is the case in schemes based on the comparisons of completed results. Our technique allows for a speedier reaction (for instance, rescheduling the computation), and therefore for a faster completion of the whole execution. This is especially important in faulty environments.

6. Invitation-based schemes, such as *Invitation System* that we proposed in chapter 9, can potentially establish a more well-behaved and more dedicated workforce. Additionally, to allow the enrollment of resource owners who do not know anyone that can address them an invitation, it should be possible to share reputation among volunteer projects. This way, a worker who has already positively participated in other projects will have references that will ease her admission to an invitation-based project. All of these contribute for a more sabotage-tolerant desktop grid environment.

10.3 Future Work

Much work can still be done in the area of dependability for desktop grids. In this section, we briefly summarize some of the open issues in this research field:

- We studied some prediction-based scheduling schemes coupled with shared checkpointing. We believe that this study can be further refined, adding new prediction methodologies and broadening the analysis to Internet-wide resources. The goal would be to exploit the computer usage habits that might exist in institutional and wide-scale environments, adapting the dependability mechanisms to such environments. For instance, the checkpointing frequency of tasks executed at a given machine could be fine-tuned in accordance with the availability pattern of the machine, increasing during *instable* periods and decreasing during more *stable* periods (e.g., weekends).

- Another venue for future work relates to the development of dependability schemes targeting the federated and peer-to-peer topologies that were introduced in chapter 7. By integrating the controlling features of a centralized server-side and the resiliency and scalable features of a peer-to-peer environment, this field of research seems promising.

- Task mobility is still limited in desktop grids. As shown in this thesis, this mechanism is a valid solution for overcoming machine failures, but its adoption is still hindered by the difficulty of moving data among nodes of desktop grids. Thus, smart and efficient data moving and copying schemes are needed. In this area, some promising results have been achieved with the Bittorrent protocol by Wei et al. [Wei *et al.* 2005].

- System-level virtual machines such as *VMware*, *VirtualBox* and *QEMU*, to name just a few, are changing computing. Several characteristics make virtualization appealing for public resource computing, both from the developers and volunteers point of view. For instance, virtualization provides for an easy deployment of the same computing environment across all participating machines. This includes the operating system and all the software stack that might be required by the desktop grid application. Furthermore, having a unique and well-known environment across all volunteers considerably eases the task of the developers, since they only have to deal with a single platform. The use of virtual machines for desktop grid computing also brings an enhanced security for both volunteers and for desktop grid applications. Indeed, on the one hand, the sandboxing isolation offered by system-level virtual machines makes the execution of a foreign application by a volunteer machine much safer. In fact, virtual machines are often used in security-oriented environments for testing potentially malicious software. On the other hand, virtual machines can also contain volunteers from tampering with the virtual environment, although this cannot be totally prevented, since savvy users will always be able to access the virtual environment. However, the most appealing feature from the point of view of dependability, lies in the possibility of saving the state of the virtualized OS to persistent storage. This is done in a transparent manner, requiring no intervention nor modification of the operating system. This checkpointing feature allows simultaneously for fault tolerance and migration, making possible the exportation of a virtual environment to another physical machine, with the execution being resumed at the remote machine. Thus, with all these features, virtual machines will most certainly play an important, if not decisive, role in desktop grid computing.

- The precise accounting of computing resources is another area that needs to be tackled for a wide adoption, not only of desktop grids, but of general purpose grids. Indeed, even for current volunteer projects were credits are merely virtual and bear no true economic value besides prestige, significant complains are voiced. This can be seen in the forums that are associated with the main public volunteer systems. Many volunteers question the fairness of the currently existing credit accounting systems. Moreover, for wide-scale adoption of grid computing, a way to fairly and properly reward resource contributors is needed to gather interest of resource owners. Likewise, a market for on-demand computing power is only viable if buyers or borrowers trust the accounting scheme, i.e. that they are being accurately charged for what their applications effectively consume. Therefore, proper accountability of resources is an important open issue that is part of the wider area of *quality of services*.

- The climate problems associated to carbon emissions and the rising costs of energy have fostered a more rationale use of energy. As the number of computer and related infrastructures (routers, etc.) continues to grow, a significant effort is being made in rationalizing the use of energy by computers. Thus, desktop grids face an additional challenge: produce fast results but in a energy-wise manner. Under such circumstances, fault tolerance assumes an important role, specially in the area of preserving computing results against failures like, for instance, crashes of machines and alike. Likewise, dependability techniques for certifying results will certainly need a major overhaul with energy-expensive schemes such as replication being avoided for more energy-wise approaches. Thus, energy-aware scheduling and fault tolerant methodologies will be important elements for future desktop grids.

All in all, finding proper solutions for these problems (and many others) will certainly make desktop grids, and more broadly, grids, closer to attain the commodity status that was forecasted in the seminal work of Foster et al. [Foster & Kesselman 1998]. The goal is to allow users to obtain computing power in the same simple way as we, today, plug electric appliances to obtain electricity from the power grid. We strongly believe that this vision of grid computing is viable and that computing power as a large scale utility will be a reality in the future. With this thesis, we hope to have contributed to make this future closer and expect to proceed, in the next years, to the coming of age of this reality.

Bibliography

ACHARYA, ANURAG, & SETIA, SANJEEV. 1999. Availability and utility of idle memory in workstation clusters. *Pages 35–46 of: SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press. 79, 80

ACHARYA, ANURAG, EDJLALI, GUY, & SALTZ, JOEL. 1997. The utility of exploiting idle workstations for parallel computation. *Pages 225–234 of: SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press. 65, 71, 79, 80

AGBARIA, A., & FRIEDMAN, R. 2005. A replication-and checkpoint-based approach for anomaly-based intrusion detection and recovery. *Pages 137–143 of: Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*. Los Alamitos, CA, USA: IEEE Computer Society. 194

AL GEIST, W.G., HUSS-LEDERMAN, S., LUMSDAINE, A., LUSK, E., SAPHIR, W., SKJELLUM, T., & SNIR, M. 1996. MPI-2: Extending the Message-Passing Interface. *Pages 128–135 of: Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing, Lyon, France, August 1996*. London, UK: Springer-Verlag. 71

ALLEN, B. 2004. Monitoring hard disks with SMART. *Linux Journal*, **2004**(117). 51, 59, 69, 82

ANDERSEN, R., COPENHAGEN, D., & VINTER, B. 2006. Harvesting Idle Windows CPU Cycles for Grid Computing. *In: Proceedings of the 2006 International Conference on Grid Computing and Applications (GCA-2006)*. 33

ANDERSON, DAVID. 2004. BOINC: A System for Public-Resource Computing and Storage. *Pages 4–10 of: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, 2004, Pittsburgh, USA*. 2, 3, 12, 13, 14, 15, 21, 29, 32, 33, 139, 177, 182, 184, 185

ANDERSON, DAVID, CHRISTENSEN, CARL, & ALLEN, BRUCE. 2006. Designing a Runtime System for Volunteer Computing. *Pages 126–135 of: Proceedings of the ACM/IEEE SC2006 Conference, November 2006, Tampa, Florida, USA*. New York, NY, USA: ACM. 32, 34, 37

ANDERSON, DAVID P., & FEDAK, GILLES. 2006. The Computational and Storage Potential of Volunteer Computing. *Pages 73–80 of: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. Los Alamitos, CA, USA: IEEE Computer Society. 1, 81, 159, 182

ANDERSON, D.P., COBB, J., KORPELA, E., LEBOFSKY, M., & WERTHIMER, D. 2002. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, **45**(11), 56–61. 12, 16

ANDERSON, D.P., KORPELA, E., & WALTON, R. 2005. High-Performance Task Distribution for Volunteer Computing. *Pages 196–203 of: Proceedings of 1st International Conference on e-Science and Grid Computing. Melbourne, Australia*. 32, 154

ANDERSON, T.E, CULLER, D.E, & PATTERSON, D.A. 1995. A case for NOW (Networks of Workstations). *Micro, IEEE*, **15**(1), 54–64. 50

ANDRZEJAK, ARTUR, DOMINGUES, PATRICIO, & SILVA, LUIS. 2005. Classifier-Based Capacity Prediction for Desktop Grids. *In: Integrated Research in Grid Computing - CoreGRID Workshop, November 2005, Pisa, Italy*. 8, 93

ANDRZEJAK, ARTUR, DOMINGUES, PATRICIO, & SILVA, LUIS. 2006 (April). Predicting Machine Availabilities in Desktop Pools. *In: 10th IEEE/IFIP Network Operations and Management Symposium (NOMS'2006), 1-4 April 2006, Vancouver, Canada*. 9

ANGLANO, C., & CANONICO, M. 2005. Fault-Tolerant Scheduling for Bag-of-Tasks Grid Applications. *Pages 630–639 of: Proceedings of the 2005 Eu-*

ropean Grid Conference (EuroGrid 2005). Lecture Notes in Computer Science, vol. 3470. 126

ANGLANO, C., BREVIK, J., CANONICO, M., NURMI, D., & WOLSKI, R. 2006 (Sept.). Fault-aware Scheduling for Bag-of-Tasks Applications on Desktop Grids. *Pages 56–63 of: Proceedings of 7th IEEE/ACM International Conference on Grid Computing, Barcelona, Spain.* 89

ANNAPUREDDY, S., FREEDMAN, M.J., & MAZIERES, D. 2005. Shark: Scaling File Servers via Cooperative Caching. *Pages 129–142 of: Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI). June 2005, Boston, USA.* 131, 147

ANTONELLI, DOMINIC, CORDERO, AREL, & METTLER, ADRIAN. 2004. *Securing Distributed Computation with Untrusted Participants.* Tech. rept. University of California at Berkeley. 194

ARAUJO, FILIPE, DOMINGUES, PATRICIO, KONDO, DERRICK, & SILVA, LUIS. 2006. Validating Desktop Grid Results by Comparing Intermediate Checkpoints. *In: 3th Integration Workshop, CoreGRID - Network of Excellence. October 2006, Krakow, Poland.* 10

ARPACI, REMZI H., DUSSEAU, ANDREA C., VAHDAT, AMIN M., LIU, LOK T., ANDERSON, THOMAS E., & PATTERSON, DAVID A. 1995. The interaction of parallel and sequential workloads on a network of workstations. *Pages 267–278 of: SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint International Conference on Measurement and Modeling of Computer Systems.* New York, NY, USA: ACM Press. 1, 78, 79, 221

BALATON, ZOLTAN, GOMBAS, GABOR, KACSUK, PETER, KORNAFELD, ADAM, KOVACS, JOZSEF, MAROSI, ATTILA CSABA, VIDA, GABOR, PODHORSZKI, NORBERT, & KISS, TAMAS. 2007. SZTAKI Desktop Grid: a Modular and Scalable Way of Building Large Computing Grids. *Page 475 of: International Parallel and Distributed Processing Symposium.* Los Alamitos, CA, USA: IEEE Computer Society. 12, 39, 40, 156

BARHAM, PAUL, DRAGOVIC, BORIS, FRASER, KEIR, HAND, STEVEN, HARRIS, TIM, HO, ALEX, NEUGEBAUER, ROLF, PRATT, IAN, & WARFIELD, ANDREW. 2003. Xen and the art of virtualization. *Pages 164–177 of: SOSP '03:*

Proceedings of the nineteenth ACM symposium on Operating systems principles. New York, NY, USA: ACM Press. 18, 173

BECKLES, B. 2005. Building a secure Condor pool in an open academic environment. *Pages 19–22 of: Proceedings of the UK e-Science All Hands Meeting 2005*. 28

BELLARD, F. 2005 (april). QEMU, a Fast and Portable Dynamic Translator. *Pages 41–46 of: Proceedings of the USENIX Annual Technical Conference, FREENIX Track. April 2005, Anaheim, CA, USA*. 18

BLAKE, CHUCK, & RODRIGUES, RODRIGO. 2003. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. *Pages 1–6 of: Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX). May 2003, Lihue, Hawaii, USA*. 139

BOHANNON, JOHN. 2005a. DISTRIBUTED COMPUTING: Grassroots Supercomputing. *Science*, **308**(5723), 810–813. 11, 17, 23

BOHANNON, JOHN. 2005b. DISTRIBUTED COMPUTING: Grid Sport: Competitive Crunching. *Science*, **308**(5723), 812–812. 177, 203

BOLOSKY, WILLIAM J., DOUCEUR, JOHN R., ELY, DAVID, & THEIMER, MARVIN. 2000. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *Pages 34–43 of: SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM Press. 65, 79, 80, 82

BUTT, ALI RAZA, JOHNSON, TROY A., ZHENG, YILI, & HU, Y. CHARLIE. 2004. Kosha: A Peer-to-Peer Enhancement for the Network File System. *Pages 51–62 of: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing. November 2004, Pittsburgh,PA,USA*. Washington, DC, USA: IEEE Computer Society. 147

BUTT, R., ZHANG, R., & HU, Y. C. 2003. A Self-Organizing Flock of Condors. *In: Supercomputing 2003*. 47, 146

BYTEMARK. 2007. *ByteMark's Benchmark (http://www.byte.com/bmark)*. 60, 91

CAPPELLO, F., DJILALI, S., FEDAK, G., HERAULT, T., MAGNIETTE, F., NERI, V., & LODYGENSKY, O. 2005. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, **21**(3), 417–437. 16, 41

CHAWATHE, Y., RATNASAMY, S., BRESLAU, L., LANHAM, N., & SHENKER, S. 2003. Making gnutella-like P2P systems scalable. *Pages 407–418 of: Proceedings of the 2003 conference on Applications, Technologies, Architectures and Protocols for Computer Communications, 2003*. ACM Press New York, NY, USA. 159

CHESSBRAIN. 2007. *The ChessBrain Project (http://www.chessbrain.net/)*. 152, 171

CHIEN, ANDREW, CALDER, BRAD, ELBERT, STEPHEN, & BHATIA, KARAN. 2003. Entropia: architecture and performance of an enterprise desktop grid system. *J. Parallel Distrib. Comput.*, **63**(5), 597–610. 2, 43, 81

CHOI, S., BAIK, M., HWANG, C., GIL, J., & YU, H. 2004. Volunteer availability based fault tolerant scheduling mechanism in DG computing environment. *Pages 366–371 of: Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA'04), Cambridge, MA, USA*. 15

CHRISTENSEN, CARL, AINA, TOLU, & STAINFORTH, DAVID. 2005. The Challenge of Volunteer Computing with Lengthy Climate Model Simulations. *Pages 8–15 of: Proceedings of 1st International Conference on e-Science and Grid Computing. December 2005, Melbourne, Australia*. Los Alamitos, CA, USA: IEEE Computer Society. 189

CIRNE, W., PARANHOS, D., COSTA, L., SANTOS-NETO, E., BRASILEIRO, F., SAUVE, J., SILVA, F., BARROS, C., & SILVEIRA, C. 2003. Running Bag-of-Tasks applications on computational grids: the MyGrid approach. *Pages 407–416 of: Proceedings of International Conference on Parallel Processing, October 2003*. 5, 88

CIRNE, W., BRASILEIRO, F., ANDRADE, N., COSTA, L.B., ANDRADE, A., NOVAES, R., & MOWBRAY, M. 2006. Labs of the World, Unite!!! *Journal of Grid Computing*, **4**(3), 225–246. 126, 173

COHEN, B. 2003. Incentives Build Robustness in BitTorrent. *In: First Workshop on the Economics of Peer-to-Peer Systems. June 2003, Berkeley, CA, USA*. 145

CORKILL, D.D. 1991. Blackboard systems. *AI Expert*, **6**(9), 40–47. 152

COX, L.P., MURRAY, C.D., & NOBLE, B.D. 2002. Pastiche: making backup cheap and easy. *ACM SIGOPS Operating Systems Review*, **36**, 285–298. 131, 146

CREEGER, MACHE. 2005. Multicore CPUs for the Masses. *ACM Queue*, **3**(7). 20

DANEZIS, G., & LAURIE, B. 2007 (april). Private Yet Abuse Resistant Open Publishing. *In: 15th International Workshop on Security Protocols*. 216

DATASYNAPSES. 2007. *DataSynapse (http://www.datasynapse.com)*. 21, 44

DIGIPEDE. 2007. *Digipede (http://www.digipede.com/)*. 44

DINDA, P. 1999. *The Statistical Properties of Host Load (Extended Version)*. Technical Report CMU-CS-98-175. School of computer Science - Carnegie Mellon University. 97

DINGLEDINE, R., MATHEWSON, N., & SYVERSON, P. 2004 (August). TOR: The second-generation onion router. *In: 13th USENIX Security Symposium*. 199

DISTRIBUTED.NET. 2007. *Distributed.net (http://www.distributed.net)*. 29

DOMINGUES, PATRICIO, MARQUES, PAULO, & SILVA, LUIS. 2005a. Distributed Data Collection through Remote Probing in Windows Environments. *Pages 59–65 of: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05), Lugano, Switzerland*. IEEE Computer Society. 8, 50

DOMINGUES, PATRICIO, MARQUES, PAULO, & SILVA, LUIS. 2005b. Resource Usage of Windows Computer Laboratories. *Pages 469–476 of: International Conference Parallel Processing (ICPP 2005)/Workshop PEN-PCGCS, Oslo, Norway*. 1, 8, 11

Domingues, Patricio, Araujo, Filipe, & Silva, Luis Moura. 2006a. A DHT-Based Infrastructure for Sharing Checkpoints in Desktop Grid Computing. *Pages 126–133 of: e-Science'06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, Amsterdam, Netherlands*. IEEE Computer Society. 10

Domingues, Patricio, Marques, Paulo, & Silva, Luis. 2006b. DGSchedSim: A Trace-Driven Simulator to Evaluate Scheduling Algorithms for Desktop Grid Environments. *Pages 83–90 of: PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06), Montbéliard, France*. IEEE Computer Society. 9

Domingues, Patricio, Andrzejak, Artur, & Silva, Luis. 2006c. Scheduling for Fast Turnaround Time on Institutional Desktop grid. *In: Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture, Paris, France*. 8

Domingues, Patricio, Silva, Joao Gabriel, & Silva, Luis. 2006d. Sharing Checkpoints to Improve Turnaround Time in Desktop Grid. *Pages 301–306 of: 20th IEEE International Conference on Advanced Information Networking and Applications (AINA 2006), 18-20 April 2006, Vienna, Austria*. IEEE Computer Society. 9, 144

Domingues, Patricio, Andrzejak, Artur, & Silva, Luis Moura. 2006e (december). Using Checkpointing to Enhance Turnaround Time on Institutional Desktop Grids. *Pages 73–81 of: e-Science'06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, Amsterdam, Netherlands*. 9

Domingues, Patricio, Sousa, Bruno, & Silva, Luis Moura. 2007. Sabotage-Tolerance and Trust Management in Desktop Grid Computing. *Future Generation Computation Systems*, **23**(7), 904–912. 10

Douceur, John R. 2002. The Sybil Attack. *Pages 251–260 of: IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag. 198

DOUCEUR, J.R. 2003. Is remote host availability governed by a universal law? *ACM SIGMETRICS Performance Evaluation Review*, **31**(3), 25–29. 66, 68

DU, WENLIANG, JIA, JING, MANGAL, MANISH, & MURUGESAN, MUMMOORTHY. 2004. Uncheatable Grid Computing. *Pages 4–11 of: ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. Washington, DC, USA: IEEE Computer Society. 23, 178

EASTLAKE, D., & JONES, P. 2001. *US Secure Hash Algorithm 1 (SHA1)*. RFC 3174 (Informational). 184

EBAY. 2007. *eBay auction site (http://www.ebay.com/)*. 198

EINSTEIN. 2007. *Einstein@home (http://einstein.phys.uwm.edu/)*. 6, 12, 151, 189

ELNOZAHY, E. N. (MOOTAZ), ALVISI, LORENZO, WANG, YI-MIN, & JOHNSON, DAVID B. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Survey*, **34**(3), 375–408. 85

FAN, ZHE, QIU, FENG, KAUFMAN, ARIE, & YOAKUM-STOVER, SUZANNE. 2004. GPU Cluster for High Performance Computing. *Pages 47–58 of: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 6-12 November 2004, Pittsburgh, USA*. IEEE Computer Society. 12

FEDAK, GILLES. 2003 (June). *XtremWeb: une plate-forme pour l'étude expérimentale du calcul global pair-à-pair*. Ph.D. thesis, Université Paris XI. 40

FEDAK, GILLES, GERMAIN, C., NERI, V., & CAPPELLO, FRANCK. 2001. XtremWeb: A Generic Global Computing System. *Pages 582–587 of: 1st Int'l Symposium on Cluster Computing and the Grid (CCGRID'01), Brisbane, Australia*. IEEE Computer Society. 2, 3, 21, 40, 41, 184

FIGUEIREDO, RJ, DINDA, PA, & FORTES, JAB. 2003. A Case for Grid Computing on Virtual Machines. *Pages 550–559 of: 23rd International Conference on Distributed Computing Systems*. 18

FOLDING@HOME. 2006. *Folding@home (http://folding.stanford.edu/)*. 23

FORD, B., SRISURESH, P., & KEGEL, D. 2005. Peer-to-peer communication across network address translators. *In: Proceedings of the 2005 USENIX Annual Technical Conference*. 152

FOSTER, I., & KESSELMAN, C. 1998. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA. 225

FRAYN, CM, & JUSTINIANO, C. 2004. The ChessBrain Project - Massively Distributed Speed-critical Computation. *Pages 10–13 of: Proceedings ICSEC Workshop on Grid Computing and Applications, Singapore*. 152, 171

FRAYN, COLIN, JUSTINIANO, CARLOS, & LEW, KEVIN. 2006. ChessBrain II - A Hierarchical Infrastructure for Distributed Inhomogeneous Speed-Critical Computation. *Pages 13–18 of:* LOUIS, SUSHIL J., & KENDALL, GRAHAM (eds), *IEEE Symposium on Computational Intelligence and Games (CIG'06), 2006*. IEEE. 172

GANGULY, A., AGRAWAL, A., BOYKIN, PO, & FIGUEIREDO, R. 2006. WOW: Self-Organizing Wide Area Overlay Networks of Virtual Workstations. *Pages 30–42 of: 15th IEEE International Symposium on High Performance Distributed Computing, 2006*. 47

GELERNTER, D. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, **7**(1), 80–112. 151

GENTZSCH, WOLFGANG. 2001. Sun Grid Engine: Towards Creating a Compute Power Grid. *Pages 35–39 of: Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid*. Los Alamitos, CA, USA: IEEE Computer Society. 29

GIMPS. 2007. *GIMPS (http://www.mersenne.org/prime.htm)*. 29

GMAIL. 2007. *gmail (http://gmail.google.com/)*. 215

GODFREY, P. BRIGHTEN, SHENKER, SCOTT, & STOICA, ION. 2006 (September). Minimizing Churn in Distributed Systems. *Pages 147–158 of: ACM SIGCOMM'06*. 134

GRIDORG. 2007. *grid.org (http://www.grid.org)*. 44

GUHA, S., DASWANI, N., & JAIN, R. 2006. An experimental study of the Skype peer-to-peer VoIP system. *In: Proceedings of 5th Int'l Workshop on Peer-to-Peer Systems (IPTPS), 2006, Santa Barbara, CA, USA*. 159, 161, 172

GUPTA, ASHISH, LIN, BIN, & DINDA, PETER A. 2004. Measuring and Understanding User Comfort With Resource Borrowing. *Pages 214–224 of: HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society. 50

GUPTA, R., & SEKHRI, V. 2006. CompuP2P: An Architecture for Internet Computing Using Peer-to-Peer Networks. *IEEE Transactions on Parallel and Distributed Systems*, **17**(11), 1306–1320. 152

GUPTA, R., & SOMANI, A. 2004. CompuP2P: An Architecture for Sharing of Computing Resources in Peer-to-peer Networks with Selfish Nodes. *In: Proceedings of second Workshop on the Economics of peer-to-peer Systems*. 46

HAN, JAESUN, & PARK, DAEYEON. 2003 (September). A Lightweight Personal Grid Using a Supernode Network. *Pages 168–175 of: P2P 2003 – Proceedings of 3rd International Conference on Peer-to-Peer Computing, 2003.(P2P 2003)*. 46, 174

HEAP, DG. 2003. *Taurus - A Taxonomy of Actual Utilization of Real UNIX and Windows Servers*. Tech. rept. GM12-0191. IBM White Paper. 1, 11, 49, 81, 82

HOLOHAN, A., & GARG, A. 2005. Collaboration Online: The Example of Distributed Computing. *Journal of Computer-Mediated Communication*, **10**(4). 23, 39, 179, 202

HU, HAIFENG, KAMINSKY, MICHAEL, GIBBONS, PHILLIP, & FLAXMAN, ABRAHAM. 2006 (September). SybilGuard: Defending Against Sybil Attacks via Social Networks. *Pages 265–276 of: ACM SIGCOMM'06*. 198

HUGHES, GF, MURRAY, JF, KREUTZ-DELGADO, K., & ELKAN, C. 2002. Improved disk-drive failure warnings. *Reliability, IEEE Transactions on*, **51**(3), 350–357. 69

IPERF. 2007. *Iperf - The TCP/UDP Bandwidth Measurement Tool (http://dast.nlanr.net/Projects/Iperf/)*. 111

IYER, S., ROWSTRON, & DRUSCHEL, P. 2002. SQUIRREL: A decentralized, peer-to-peer web cache". *In: Principles of Distributed Computing (PODC 2002)*. 146

KACSUK, PETER, PODHORSZKI, NORBERT, & KISS, TAMAS. 2005 (May). *Scalable Desktop Grid System*. Tech. rept. TR-0006. Institute on System Architecture, CoreGRID - Network of Excellence. 156

KACSUK, PETER, PODHORSZKI, NORBERT, & KISS, TAMAS. 2007. Scalable Desktop Grid System. *Pages 27–38 of: High Performance Computing for Computational Science - VECPAR 2006*. Lecture Notes in Computer Science, vol. 4395/2007. Springer Berlin / Heidelberg. 39

KIRKPATRICK, S., GELATT JR, CD, & VECCHI, MP. 1983. Optimization by Simulated Annealing. *Science*, **220**(4598), 671. 152

KNIGHT, WILL. 2006 (May). *Programmer speeds search for gravitational waves*. http://www.newscientisttech.com/article.ns?id=dn9180. 36

KONDO, DERRICK, ARAUJO, FILIPE, DOMINGUES, PATRICIO, & SILVA, LUIS. 2007. Result Error Detection on Heterogeneous and Volatile Resources via Intermediate Checkpointing. *In: CoreGRID Workshop on Grid Programming Model Grid and P2P Systems Architecture Grid Systems, Tools and Environments. June 2007, Hellas Heraklion, Crete, Greece*. 10

KONDO, D., TAUFER, M., BROOKS, CL, CASANOVA, H., & CHIEN, AA. 2004a (April). Characterizing and Evaluating Desktop Grids: an Empirical Study. *In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPSŠ04)*. 12, 78, 81, 126, 221

KONDO, DERRICK, CHIEN, ANDREW A., & CASANOVA, HENRI. 2004b. Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids. *In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society. 90, 125, 127

LARSON, S.M., SNOW, C.D., SHIRTS, M., & PANDE, V.S. 2002. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*. 12

LIANG, J., KUMAR, R., & ROSS, K.W. 2006. The FastTrack overlay: A measurement study. *Computer Networks*, **50**(6), 842–858. 159, 172

Litzkow, M., Livny, M., & Mutka, M. 1988. Condor - A Hunter of Idle Workstations. *Pages 104–111 of: 8th International Conference on Distributed Computing Systems (ICDCS)*. Washington, DC: IEEE Computer Society. 21, 25

Litzkow, M., Tannenbaum, T., Basney, J., & Livny, M. 1997. *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*. Technical Report 1346. University of Wisconsin-Madison Computer Sciences. 29, 86

Lodygensky, O., Fedak, G., Néri, V., Cordier, A., & Cappello, F. 2003a (March). Auger & XtremWeb : Monte Carlo computation on a global computing platform. *In: Proceedings of Computing in High Energy and Nuclear Physics (CHEP2003)*. 40

Lodygensky, O., Fedak, G., Cappello, F., Neri, V., Livny, M., & Thain, D. 2003b (May). XtremWeb & Condor: sharing resources between Internet connected Condor pool. *Pages 382–389 of: 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*. 25, 42

Longbottom, Roy. 2007. *Roy Longbottom's PC Benchmark Collection (http://homepage.virgin.net/roy.longbottom/)*. 60

Luther, A., Buyya, R., Ranjan, R., & Venugopal, S. 2005 (June). Alchemi: A .NET-Based Enterprise Grid Computing System. *Pages 27–30 of: Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*. 29

Martin, A., Aina, T., Christensen, C., Kettleborough, J., & Stainforth, D. 2005. On Two Kinds of public-resource Distributed Computing. *Pages 931–936 of: Proceedings of the Fourth UK e-Science All Hands Meeting. September 2005, Nottingham, UK*. 12, 16, 130, 139

Massie, M.L., Chun, B.N., & Culler, D.E. 2004. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, **30**(7), 817–840. 90

Mayer, Uwe F. 2007. *NBench Project (http://www.tux.org/ mayer/linux/bmark.html/)*. 60, 98

McCarty, B. 2003. Botnets: big and bigger. *Security & Privacy Magazine, IEEE*, **1**(4), 87–90. 16

McCullagh, Dellan. 2000. *Intel Nixes Chip Tracking ID*. http://www.wired.com/news/politics/0,1283,35950,00.html . 200

Molnar, D. 2000. The SETI@Home Problem. *ACM Crossroads Student Magazine*, September. 16, 178

Murphy, B., & Levidow, B. 2000. Windows 2000 Dependability. *In: Proceedings of the IEEE International Conference on Dependable Systems and Networks*. 69

Musgrove, Mike. 2004. Trader, 15, Riding a Gmail Boom. *Washington Post, 6th June 2004*, June, F01. 215

orkut. 2007. *Orkut*. http://orkut.com/. 215

parabon. 2007. *Frontier - Parabon Computation (http://www.parabon.com/)*. 44

Passport. 2007. *Microsoft Passport (http://www.passport.net/)*. 200

Petrou, D., Ganger, G.R., & Gibson, G.A. 2004. Cluster Scheduling for Explicitly-speculative Tasks. *Pages 336–345 of: Proceedings of the 18th annual international conference on Supercomputing*. ACM Press New York, NY, USA. 88

Pineau, J.F., Robert, Y., & Vivien, F. 2005. Off-line and on-line scheduling on heterogeneous master-slave platforms. *Research Report*, **31**. 94, 110

Pouwelse, J.A., Garbacki, P., Bakker, J. Wangand A., Yang, J., Iosup, A., Epema, D., M.Reinders, van Steen, M.R., & Sips, H.J. 2006 (Feb). Tribler: A social-based peer to peer system. *In: Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS)*. 216

qmc. 2007. *QMC@home (http://qah.uni-muenster.de/)*. 12

Ratnasamy, Sylvia, Francis, Paul, Handley, Mark, Karp, Richard, & Schenker, Scott. 2001. A Scalable Content-addressable Network. *Pages 161–172 of: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*. ACM Press. 146

RECORDON, DAVID, & REED, DRUMMOND. 2006. OpenID 2.0: a platform for user-centric identity management. *Pages 11–16 of: DIM '06: Proceedings of the second ACM workshop on Digital identity management*. New York, NY, USA: ACM. 199

RESNICK, P., ZECKHAUSER, R., FRIEDMAN, E., & KUWABARA, K. 2000. Reputation Systems: Facilitating Trust in Internet Interactions. *Communications of the ACM*, **43**(12), 45–48. 198

RIVEST, R. 1992. *The MD5 Message-Digest Algorithm*. RFC 1321 (Informational). 184

ROSENBERG, J. 2006. Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. *draft-ietf-mmusic-ice-08 (work in progress), March*. 152

ROSENBERG, J., WEINBERGER, J., HUITEMA, C., & MAHY, R. 2003. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. RFC 3489 (Proposed Standard). 152

ROSENBERG, J., *et al.* 2004. Traversal Using Relay NAT (TURN). *draft-rosenberg-midcom-turn-05 (work in progress), July*. 152

ROSETTA. 2006. *Rosetta@home (http://boinc.bakerlab.org/rosetta/)*. 12

ROWSTRON, ANTONY, & DRUSCHEL, PETER. 2001 (Nov). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *In: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*. 146

RUSSINOVICH, M., & COGSWELL, B. 2006. *Sysinternals - PsTools (http://www.sysinternals.com/)*. 55

RYU, KD, & HOLLINGSWORTH, JK. 2004. Unobtrusiveness and Efficiency in Idle Cycle Stealing for PC Grids. *In: Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe, New Mexico, USA*. 74, 80

SARMENTA, L.F.G., & HIRANO, S. 1999. Bayanihan: Building and studying web-based volunteer computing systems using Java. *Future Generation Computer Systems*, **15**(5), 675–686. 183

SARMENTA, LUIS F. G. 2002. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Gener. Comput. Syst.*, **18**(4), 561–572. 178, 180, 182, 186, 198

SCHALLER, RR. 1997. Moore's law: past, present and future. *Spectrum, IEEE*, **34**(6), 52–59. 20

SCHNEIER, BRUCE. 1999. *Why Intel's ID tracker won't work.* http://news.zdnet.com/2100-9595_22-513519.html . 200

SETI. 2007. *SETI@home (http://setiathome.berkeley.edu/)*. 6, 12, 20

SHUDO, K., TANAKA, Y., & SEKIGUCHI, S. 2005. P3: P2P-based middleware enabling transfer and aggregation of computational resources. *In: Proceedings of IEEE International Symposium on Cluster Computing and the Grid, 2005 (CGrid 2005)*. 45, 173

SILVA, L. M., & SILVA, J. G. 1998. System-Level Versus User-Defined Checkpointing. *Pages 68–74 of: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*. Washington, DC, USA: IEEE Computer Society. 2, 23, 86, 184

SIMAP. 2007. *SIMAP@home (http://boinc.bio.wzw.tum.de/boincsimap/)*. 15, 189

SIT, E., CATES, J., & COX, R. 2003. A DHT-based Backup System. *In: Proceedings of the 1st IRIS Student Workshop*. 131, 146

SON, SECHANG, & LIVNY, MIRON. 2003. Recovering Internet Symmetry in Distributed Computing. *Pages 542–549 of: Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, May 2003, Tokyo, Japan*. Los Alamitos, CA, USA: IEEE Computer Society. 17, 184, 195

STAINFORTH, D., KETTLEBOROUGH, J., MARTIN, A., SIMPSON, A., GILLIS, R., AKKAS, A., GAULT, R., COLLINS, M., GAVAGHAN, D., & ALLEN, M. 2002. Climateprediction.net: Design Principles for Public-Resource Modeling Research. *Pages 32–38 of: Proceedings of the 14th IASTED International Conference: Parallel And Distributed Computing And Systems*. 12

STOICA, ION, MORRIS, ROBERT, KARGER, DAVID, KAASHOEK, FRANS, & BALAKRISHNAN, HARI. 2001. Chord: A Scalable Peer-to-Peer Lookup

Service for Internet Applications. *Proceedings of the 2001 SIGCOMM conference*, **31**(4), 149–160. 130

Sutter, Herb. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, **30**(3). 20

sztaki. 2007. *Sztaki Desktop Grid. (http://szdg.lpds.sztaki.hu/szdg/)*. 6, 12

Tannenbaum, T., Wright, D., Miller, K., & Livny, M. 2001. Condor - A Distributed Job Scheduler. *Beowulf Cluster Computing with Linux. MIT Press*, October. 25, 26, 28

Taufer, M., Anderson, D., Cicotti, P., & Brooks III, CL. 2005a. Homogeneous Redundancy: a Technique to Ensure Integrity of Molecular Simulation Results Using Public Computing. *Pages 119–127 of: Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), 2005*. 18, 38, 178

Taufer, Michela, Teller, Patricia J., Anderson, David P., & Charles L. Brooks, III. 2005b. Metrics for Effective Resource Management in Global Computing Environments. *Pages 204–211 of: Proceedings of 1st International Conference on e-Science and Grid Computing. Melbourne, Australia*. Los Alamitos, CA, USA: IEEE Computer Society. 127, 183

Thain, D., Tannenbaum, T., & Livny, M. 2005. Distributed computing in practice: the Condor experience. *Concurrency and Computation Practice and Experience*, **17**(2-4), 323–356. 15, 28, 144, 146

Tritrakan, K., & Muangsin, V. 2005 (March). Using Peer-to-Peer Communication to Improve the Performance of Distributed Computing on the Internet. *Pages 295–298 of: 19th International Conference on Advanced Information Networking and Applications (AINA 2005)*, vol. 2. 145

Tunstall, C., McDonald, R.L., & Cole, G. 2002. *Developing WMI Solutions: A Guide to Windows Management Instrumentation*. Addison-Wesley Professional. 52

uniteddevices. 2007. *United Devices, Inc. (http://www.ud.com)*. 2, 21, 43, 44

Vazhkudai, S., Ma, X., Freeh, V., Strickland, J., Tammineedi, N., & Scott, S. 2005 (nov). FreeLoader:Scavenging Desktop Storage Resources for Scientific Data. *In: Supercomputing 2005 (SC'05): Int'l Conference on High Performance Computing, Networking and Storage, Seattle, Washington, USA*. 146

VERBEKE, J., NADGIR, N., RUETSCH, G., & SHARAPOV, I. 2002. Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment. *In: Proceedings of the 3rd International Workshop on Grid Computing*. Springer. 45, 174

VIRTUALBOX. 2007. *VirtualBox (http://www.virtualbox.org)*. 18

VMWARE. 2007. *VmWare, Inc. (http://www.vmware.com)*. 18

VON AHN, LUIS, BLUM, MANUEL, & LANGFORD, JOHN. 2004. Telling humans and computers apart automatically. *Communications of ACM*, **47**(2), 56–60. 204

WANG, YIN-MIN, VERBOWSKI, CHAD, & SIMON, DANIEL R. 2003. Persistent-state Checkpoint Comparison for Troubleshooting Configuration Failures. *Pages 311–316 of: Proceedings of International Conference on Dependable Systems and Networks, 2003*. IEEE Computer Society. 195

WATTS, DJ, & STROGATZ, SH. 1998. Collective dynamics of 'small-world' networks. *Nature*, **393**(6684), 409–10. 216

WEI, BAOHUA, FEDAK, GILLES, & CAPPELLO, FRANCK. 2005. Collaborative Data Distribution with BitTorrent for Computational Desktop Grids. *Pages 250–257 of: ISPDC '05: Proceedings of the 4th International Symposium on Parallel and Distributed Computing*. Washington, DC, USA: IEEE Computer Society. 145, 223

WENG, C., & LU, X. 2005. Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid. *Future Generation Computer Systems*, **21**(2), 271–280. 126

WILKINSON, BARRY, & ALLEN, MICHAEL. 2004. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. 2 edn. Prentice Hall. 21

ZHOU, D., & LO, V. 2005. Wave Scheduler: Scheduling for Faster Turnaround Time in Peer-based Desktop Grid Systems. *In: 11th Workshop on*

Job Scheduling Strategies for Parallel Processing (ICS 2005), Cambridge, MA. 45, 126

ZHOU, D., & LO, VIRGINA. 2006. WaveGrid: A Scalable Fast-Turnaround Heterogeneous Peer-Based Desktop Grid System. *In: 20th International Parallel & Distributed Processing Symposium (IPDPS), April 2006, Rhodes, Greece*. 47, 146