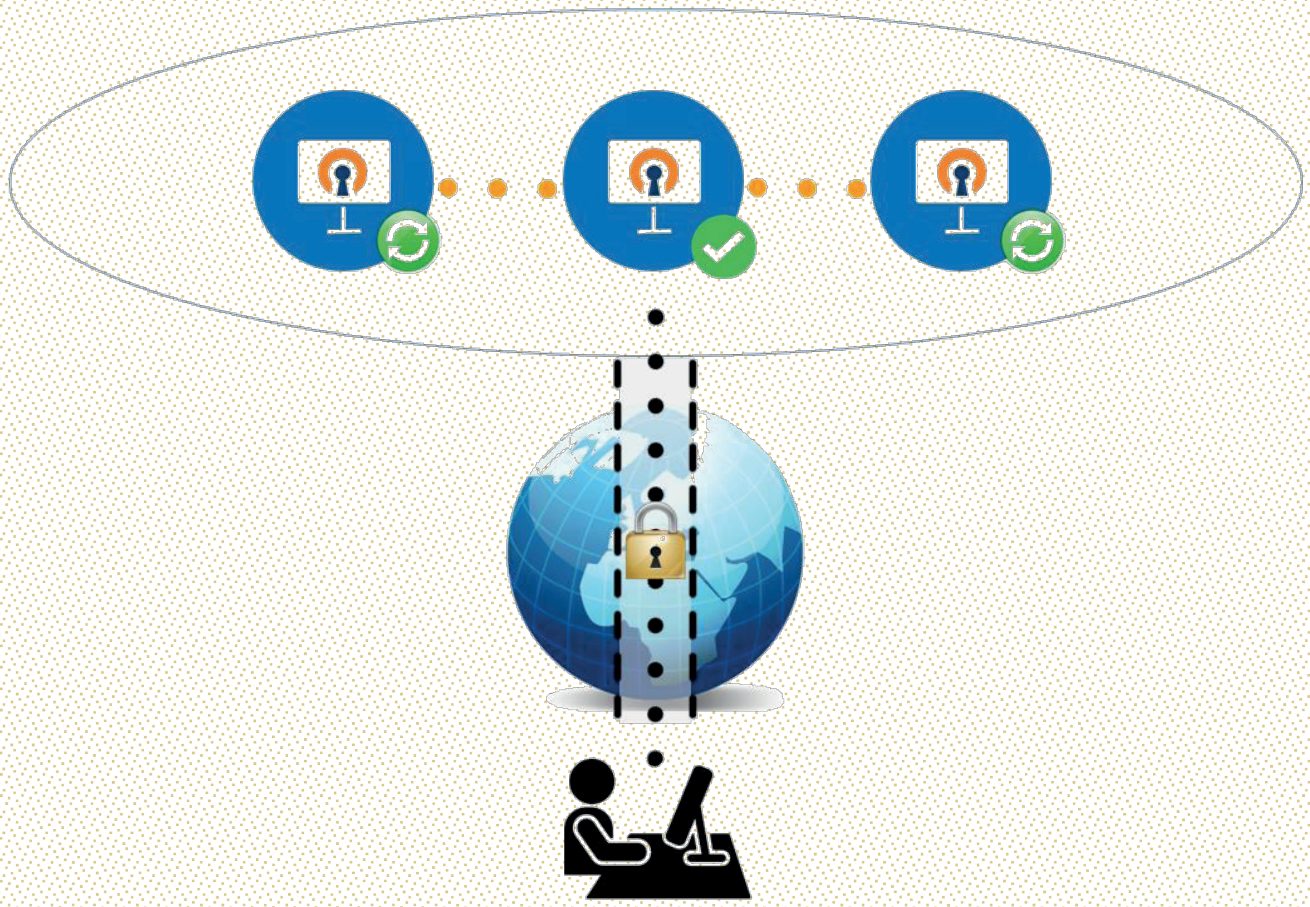


Daniel Coelho Bastos

# REDUNDÂNCIA COM SINCRONIZAÇÃO DE ESTADO EM SERVIDORES SSL VPN

Dissertação de Mestrado na área científica de Engenharia Informática, especialidade Engenharia de Software, orientada pelo Professor Doutor Fernando Boavida e por Sérgio Alves e Pedro Almeida da Dognaedis e apresentada ao Departamento de Engenharia de Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Julho de 2017

· U C ·

Universidade de Coimbra

**Mestrado em Engenharia Informática**

**Dissertação/Estágio**

Relatório Final

# Redundância com Sincronização de Estado em Servidores SSL VPN

## (SSL VPN Server Side Failover)


Daniel Coelho Bastos

dbastos@student.dei.uc.pt

Entidade Acolhedora:

Dognaedis

Orientadores:

Fernando Boavida boavida@dei.uc.pt

Sérgio Alves – Dognaedis salves@dognaedis.com

Pedro Almeida – Dognaedis palmeida@dognaedis.com



Data: 3 Julho de 2017



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

*“When something is important enough, you do it even if the odds are not in your favor.”*

Elon Musk

# Agradecimentos

Em primeiro lugar gostaria de agradecer à minha família, especialmente aos meus pais, por todo o apoio durante o meu percurso académico e por sempre acreditarem em mim. O caminho foi difícil mas o resultado está à vista.

Agradecer também ao Tiago Andrade, o amigo sempre presente nos bons e maus momentos, o amigo que nunca me deixou abrandar o ritmo, o amigo que fez de mim um melhor aluno e uma melhor pessoa. Nunca deixes de ser como és.

Agradecer à Rita Monteiro, a pessoa que me leva a atingir sempre mais do que aquilo que eu acho que sou capaz. A pessoa que faz de mim mais forte e preparado para vencer as maiores adversidades. És a pessoa mais competente e determinada que conheço.

Agradecer ao Pedro Almeida, ao Sérgio Alves e à equipa da Dognaedis pela oportunidade que me deram em abraçar um projeto tão desafiante. O voto de confiança que me deram de que eu seria capaz de realizar este projeto é algo que nunca vou esquecer. Além disso, agradecer a forma como me acolheram e me deram a liberdade de desenvolver este projeto sem qualquer tipo de restrições. Cresci muito convosco.

Agradecer ao Professor Doutor Fernando Boavida, por quem eu ganhei uma clara afinidade no decorrer desta dissertação. A sua disponibilidade, simpatia e boa disposição foram constantes durante todo o projeto e tranquilizaram-me nos momentos mais difíceis. Os seus conselhos e as suas dicas sempre acertadas não serão esquecidos.

Por fim agradecer aos meus amigos de toda uma vida: Filipa Bastos, Fábio Veríssimo, Mafalda Correia, Diana Lemos e Bárbara Londreira vocês são o melhor que a amizade tem para oferecer. A vossa amizade conforta-me todos os dias e sem a vossa influência eu não seria o que sou hoje.

Um muito obrigado a todos,

Daniel Bastos

# Resumo

No mundo atual a Internet é percepcionada como um dado adquirido. Disponível 24 horas por dia, 7 dias por semana, 365 dias por ano. A disponibilidade de um serviço é um fator crítico para o seu sucesso. No entanto, para conseguir este feito, é necessário um grande trabalho de engenharia.

Quando um *website* ou serviço online pretende comunicar de forma segura com os seus clientes existe uma solução óbvia, a utilização do protocolo SSL/TLS. Este acrescenta uma camada de proteção às comunicações efetuadas ao utilizar mecanismos seguros e algoritmos comprovados matematicamente. No entanto, esta camada extra aumenta os desafios ao nível da disponibilidade da ligação. Uma vez que as comunicações passam a estar cifradas, estas tornam-se mais difíceis de manipular por mecanismos de *failover* e *load-balacing*.

Para organizações e utilizadores preocupados com a sua privacidade online existe uma solução mais abrangente: o uso de uma Rede Privada Virtual (VPN). Esta rede possibilita uma ligação segura entre um utilizador e um dado servidor remoto, que irá servir de intermediário entre a Internet e o utilizador. Tudo aquilo a que o utilizador aceder será pedido ao servidor para aceder por ele e, posteriormente, entregar-lhe de forma segura através da ligação, conhecida como túnel, criada entre eles. Este túnel é encriptado utilizando um protocolo de segurança como o SSL/TLS. Como resultado, o *website* ou serviço online não consegue determinar a origem do pedido visto que o servidor VPN oculta qual o utilizador com quem este está a trocar informação.

Esta dissertação tem como objetivo aumentar a disponibilidade de uma ligação VPN baseada no protocolo SSL/TLS. Para isso procura implementar um mecanismo de mitigação de falhas do lado do servidor. Assim sendo, é criado um agregado de servidores que partilha o mesmo IP, sendo que só um está ativo em cada momento. Os servidores que estão em reserva recebem o estado das ligações entre os clientes e o servidor principal. Como consequência, existem várias máquinas prontas a tomar o lugar de servidor principal, de forma transparente, no caso de este falhar. Contudo, o maior desafio é a replicação do estado da ligação, isto devido à criptografia envolvida no processo. Nesta dissertação é apresentada uma solução capaz de sincronizar o estado da ligação entre o servidor principal e um ou mais clientes com um ou mais servidores de reserva.

**Palavras-chave:** VPN, SSL, TLS, Disponibilidade, Failover, Segurança, OpenBSD, OpenVPN.

# Abstract

Nowadays the Internet is perceived as a common good. Available 24 hours a day, 7 days a week, 365 days a year. Availability online is a key factor for any business in today's world. However, to achieve high availability online, a great deal of engineering work is required.

When a website or online service has the objective of comunicating safely with its customers, an obvious solution exists: the use of the SSL/TLS protocol. This protocol adds a layer of protection to the information travelling between the client and the server. It is also based on proven mathematical algorithms. On the other hand, this layer also adds new challenges regarding availability. Due to the fact that the information travelling is now encrypted, operations like failover or load-balancing are now harder to implement because it is difficult to manipulate encrypted information.

For organizations and users who are more concerned about online privacy there is a broader solution: the use of a Virtual Private Network (VPN). This private network enables an always encrypted connection between a client and a given remote server. The remote server acts as a gateway to the Internet, thus all the information accessed by the client goes through the "tunnel" between him and the server before reaching the Internet and vice-versa. To protect the "tunnel" a protocol like SSL/TLS is used. As a result, all the communication is encrypted and a website or online service is unable to determine the identity of the user being communicating with.

This thesis has the aim of increasing the availability of a VPN connection based on the SSL/TLS protocol. The proposal is to develop a server side failover procedure. To achieve that, a cluster of servers is created and configured not only to share a IP address, but also to have one server acting as master and the others acting as backups. The backup servers receive information from the master regarding connected clients and clients tunnel state. As a consequence, if the master server stops working for any particular reason, a backup server can take its place as master and be able to maintain the ongoing connections. On the other hand, it can be considered a significant challenge to implement a transparent failover procedure when dealing with encrypted data. This thesis offers a solution capable of synchronizing VPN encrypted tunnel state information between a cluster of OpenVPN servers.

**Keywords:** VPN, SSL, TLS, Security, Availability, Failover, Backup, OpenBSD, OpenVPN.

# Índice

# Lista de Figuras

# Lista de Tabelas

# Lista de Acrónimos

**AES** – Advanced Encryption Standard.

**BSD** – Berkeley Software Distribution

**HMAC** – Hash-based Message Authentication Code

**HTTPS** – Hyper Text Transport Protocol Secure ou Hyper Text Transport Protocol over SSL/TLS.

**IP** – Internet Protocol ou Protocolo da Internet

**MAC** – Media Access Control

**OSI** – Open Systems Interconnection ou Interconexão de Sistemas Abertos

**PPTP** – Point-to-Point Protocol ou Protocolo de Túneis Ponto-a-Ponto

**PRF** – Pseudo Random Function

**RSA** – Rivest, Shamir, Adleman.

**SHA** – Secure Hash Algorithm

**SSL** – Secure Socket Layer ou Camada de Socket Seguro

**TCP** – Transmission Control Protocol ou Protocolo de Controlo da Transmissão

**TLS** – Transport Layer Security ou Segurança da Camada de Transporte

**UDP** – User Datagram Protocol

**VPN** – Virtual Private Network ou Rede Privada Virtual

# Glossário

**Advanced Encryption Standard** – Algoritmo de encriptação de chave simétrica.

**Backdoor** – Meio secreto de aceder a informação priveligiada de forma ilegítima.

**Berkeley Software Distribution** – Família de sistema operativos do tipo Unix com origem na Universidade de Berkeley.

**Cluster** – Agregado de duas ou mais máquinas que partilham o mesmo endereço de rede.

**Clustering** – Ação de criação do cluster.

**Denial of Service (DoS)** – Ataque de rede com intuito de tornar um site ou serviço inacessível através do congestionamento do mesmo com pedidos.

**Failover** – Método de mitigação de falhas.

**Firewall** – Sistema que controla o tráfego que entra e sai de uma rede de computadores.

**Flag** – Designação para um atributo dado na inicialização de um software que introduz uma alteração no seu comportamento.

**Handshake** – Processo de inicialização de uma ligação SSL/TLS entre cliente e servidor.

**Hash** – Função matemática aplicada a um conjunto de dados para gerar um número único difícil de reverter.

**Kernel** – Camada do sistema operativo conhecida como núcleo. Encontra-se entre a camada aplicacional e o equipamento físico.

**Load-Balancing** – Técnica de distribuição de trabalho por entre várias máquinas. Conhecida em português por "Balanceamento de Carga".

**Multicast** – Transmissão de informação para vários destinatários.

**Open Source** – Código aberto ou software livre.

**Protocolo** – Conjunto de regras que gerem a comunicação entre duas entidades.

**Pseudo Random Function** – Função utilizada para gerar pedaços aleatórios de informação.

**RSA (Rivest, Shamir, Adleman)** – Algoritmo de encriptação de chave assimétrica

**SHA** – Conjunto de funções criptográficas de *hash*.

**Unicast** – Transmissão de informação para um único destinatário.

**X.509** – Standard da International Telecommunication Union (ITU) para Infraestruturas de Chaves Públicas (PKI) de gestão de certificados digitais e criptografia de chaves públicas.

# 1. Introdução

Esta dissertação é realizada no âmbito do estágio/dissertação do Mestrado em Engenharia Informática e visa ser o seu produto final. Reunirá não só os conhecimentos adquiridos durante o Mestrado e a Licenciatura em Eng. Informática, mas também os conhecimentos adquiridos durante o estágio na empresa Dognaedis. Esta decorreu ao longo do ano letivo de 2016/17, sob a orientação do Professor Doutor Fernando Boavida da FCTUC e do Sérgio Alves e Pedro Almeida da Dognaedis.

Neste primeiro capítulo são apresentadas as motivações em realizar este projeto e é efetuado um enquadramento do tema do estágio/dissertação, expondo a sua importância no plano atual. De seguida é apresentada a entidade que acolheu este estágio, a Dognaedis, detalhando a sua história, missão, produtos e objetivos. No capítulo Objetivos são enunciadas as metas a atingir como resultado da dissertação, e no capítulo seguinte são expressas as contribuições deste documento. Por fim, é dada uma visão geral sobre a organização deste documento.

## 1.1. Motivação Pessoal

Durante o meu percurso na faculdade, a área que mais me despertou interesse foi a segurança. A facilidade com que é possível descobrir online informação sobre tudo e todos é um misto de fascinante e alarmante. Sendo eu uma pessoa que valoriza a sua privacidade tanto online como na vida em geral, tornou-se para mim imperativo perceber como funcionam os mecanismos que nos protegem quando navegamos na Internet. Além disso, tornou-se um passatempo manter-me sempre atualizado em relação a notícias deste tema.

Quando surgiu a oportunidade de realizar esta dissertação e de poder estagiar numa empresa especializada em segurança a decisão de candidatar-me pareceu-me óbvia. A possibilidade de aprender com pessoas que lidam com clientes reais e desenvolvem soluções próprias de segurança não podia ser mais apelativa para mim.

Poder compreender o interior de um *software* tão complexo quanto o OpenVPN e desenvolver um módulo inovador que lida com uma das questões fulcrais neste serviço (a sua disponibilidade) foi uma experiência desafiante, instigando-me a dar o melhor.

## 1.2. Enquadramento e Motivações do Projeto

Hoje em dia tudo acontece na Internet. A importância da rede digital é maior do que nunca e está enraizada em todos os quadrantes da nossa sociedade. Sistemas bancários, hospitais, serviços de luz, água e gás, escolas e serviços do Estado como as finanças ou a segurança social estão todos ligados à Internet. Todos os dias são transmitidos dados sensíveis sobre cidadãos e empresas através da rede. Tal facto pressupõe um risco de que informação sensível possa parar nas mãos erradas, via falhas de sistema ou ataques informáticos.

Dado que a Internet não possui mecanismos de segurança de raiz, para proteger a informação transferida e os seus intervenientes foram sendo criados vários protocolos e mecanismos de segurança ao longo do tempo. Um dos mais conhecidos é o Secure Socket Layer (SSL) e a sua evolução – o Transport Layer Security (TLS), que está presente em toda a internet e é o principal mecanismo de segurança utilizado por páginas web, identificado pelo prefixo *https*. Existem ainda as Virtual Private Networks (VPNs), as chamadas Redes Privadas Virtuais, que possibilitam confidencialidade nas comunicações efetuadas entre máquinas na mesma rede. Estas redes asseguram a maioria das transmissões de informação sensível que acontecem hoje em dia em empresas e organizações governamentais de todo o mundo. Desta forma, estas assumem grande relevância no panorama da segurança actual.

No âmbito das VPNs, os atributos principais são: os protocolos utilizados, a autenticação e a encriptação utilizadas para proteger as comunicações. Além disso, é também importante a disponibilidade do serviço. Qualquer serviço está sujeito a sofrer períodos de indisponibilidade, seja por falhas fortuitas de componentes ou por ataques informáticos com esse propósito. Em serviços críticos, como a banca, bolsa ou telemedicina a indisponibilidade é altamente penalizadora e pode ter graves consequências monetárias. Como a segurança em serviços críticos é um atributo com grande importância, o uso de VPNs é peremptório. Por consequência, a disponibilidade de um serviço VPN é crucial.

Como as VPNs operam maioritariamente no formato Cliente – Servidor é essencial garantir a disponibilidade do servidor para que o serviço não pare de funcionar. Qualquer falha nas ligações entre os clientes e o servidor tem como efeito prejuízos significativos.

## 1.3. Entidade Acolhedora

A Dognaedis é uma empresa focada na segurança de informação. Fundada em 2010, nasceu de um projecto de investigadores da Faculdade de Ciência e Tecnologias da Universidade de Coimbra que esteve em incubação no Instituto Pedro Nunes. Após 5 anos passou a ser uma empresa privada e o seu mote é estar na vanguarda das tecnologias de segurança.

A empresa oferece serviços de consultoria na área da segurança, auditorias de redes e *software* e ainda monitorização e design de redes. O seu principal produto é o CodeV, uma solução que detecta erros em código e valida a segurança do mesmo. Em Março de 2016 a Dognaedis foi adquirida pela multinacional Prosegur, uma das maiores empresas de segurança do mundo.


*Figura 1 - Logotipo da Dognaedis*

## 1.4. Objetivos

O objetivo desta dissertação prende-se com a disponibilidade de um serviço VPN. Uma das falhas apontadas ao método atual é o facto de o cliente no momento em que um servidor falha, desconectar-se e perder o estado das operações que estava a fazer no momento. A sua resposta é tentar iniciar uma nova ligação com outro servidor disponível. Assim, o cliente perde sempre o estado da sua ligação com o servidor original e ainda tem o peso acrescido de ter de lidar com a mudança para outro servidor. Este é um processo que pode obviamente ser melhorado.

Deste modo, a solução passa por implementar redundância no lado do servidor. Quando o servidor principal falha outro toma o seu lugar, e assim em diante. Com esta solução é alcançado o *failover* do serviço, ou seja, um mecanismo de mitigação de falhas.

Contudo, o problema da perda do estado da ligação mantém-se quando acontece a troca do servidor em falha para o servidor de reserva.

O objetivo desta dissertação é também o de implementar um mecanismo de transferência do estado das ligações entre os clientes e o servidor principal, para os servidores de reserva. Isto possibilitará a transição de um servidor para outro de forma transparente e sem quebrar quaisquer operações que estivessem em curso quando o servidor principal falhou. Esta solução permitirá aumentar a qualidade dos sistemas de *failover* atuais.

## 1.5. Contribuições

Tendo em conta todo o trabalho realizado durante os dois semestres, é possível afirmar que como resultado deste foram alcançadas as seguintes contribuições:

- Código resultante do desenvolvimento da solução
- Documentação da solução desenvolvida
- Testes de validação da solução desenvolvida
- Identificação de possíveis desenvolvimentos futuros da solução

A solução desenvolvida é assim, e como esperado, a principal contribuição desta dissertação. O código, a documentação e os testes de validação perfazem o conjunto de artefactos resultantes desta dissertação. A identificação de desenvolvimentos futuros da solução garante a possível continuidade do projeto. É importante realçar, por fim, todo o estudo efetuado para conseguir formular a solução de forma a acrescentar valor aos métodos já existentes.

## 1.6. Organização do Documento

Este documento está organizado em 6 capítulos. Pretende dar uma visão completa e detalhada do trabalho desenvolvido durante o estágio relativo à dissertação *SSL VPN Server Side Failover*.

No Capítulo 1 introduz-se a motivação e o enquadramento do estágio/dissertação, enunciando os seus objetivos, as suas contribuições finais e a entidade que o acolheu.

No Capítulo 2 encontra-se uma análise do estado da arte, em que se analisam algoritmos e protocolos de segurança, VPNs, protocolos de redundância e ferramentas de *clustering*. Faz-se ainda um breve estudo de soluções comerciais de alta disponibilidade e abordam-se *softwares* com semelhanças relativamente à solução pretendida. Finalmente, são ainda fundamentadas as escolhas efetuadas ao nível das ferramentas utilizadas durante o projeto.

No Capítulo 3 é enunciada a metodologia adotada durante durante o estágio para a realização do projeto. Posteriormente, são também detalhadas as etapas definidas para o primeiro e segundo semestre bem como a sua calendarização.

Na Capítulo 4 é efetuada uma análise dos objetivos gerais do projeto, para depois definir os requisitos deste. Os requisitos são organizados como funcionais e não-funcionais.

No Capítulo 5 é especificada a solução desenvolvida. Primeiro aborda-se o seu propósito, o seu ambiente e configurações e depois apresenta-se uma representação de alto nível. Adicionalmente, é detalhado o funcionamento da solução e são documentados os testes de validação realizados.

No Capítulo 6 são retiradas conclusões e efetuadas apreciações sobre o trabalho alcançado. Também são definidos os objetivos para uma futura continuação do desenvolvimento do projecto.

Por fim, são incluídos os Anexos A, B, C e D com uma explicação do funcionamento do OpenVPN e tutoriais sobre a instalação e configuração do OpenVPN no OpenBSD bem como a configuração do CARP. E ainda os Anexos E e F com detalhes da implementação da solução e o Anexo G com a explicação dos scripts de automatização utilizados nos testes de validação. Por fim, apresenta-se a documentação do código desenvolvido redigida em formato Doxygen no Anexo H.

# 2. Estado da Arte

Neste capítulo são abordados os temas relevantes ao enquadramento e âmbito do tema deste estágio/dissertação.

Inicialmente introduzem-se os atributos relativos à segurança nas tecnologias de informação e os algoritmos criptográficos utilizados para garantir cada um deles. No terceiro subcapítulo aborda-se o protocolo SSL/TLS, dando uma explicação detalhada do seu funcionamento.

De seguida, apresentam-se as VPNs e comparam-se as várias soluções disponíveis no mercado, fazendo depois uma análise mais aprofundada das SSL VPNs e do OpenVPN. No quinto subcapítulo são estudados os vários protocolos de redundância existentes. No sexto subcapítulo é efetuado um estudo de soluções VPN com *clustering* e partilha de estado já existentes no mercado. No sétimo subcapítulo são abordadas soluções comerciais de alta disponibilidade.

No penúltimo subcapítulo são analisados protocolos de sincronização de estados com características semelhantes às da solução que se pretende implementar. Por fim, no último subcapítulo são argumentadas as decisões dos protocolos e das ferramentas a utilizar no projeto.

## 2.1. Atributos de Segurança

Com o progressivo aumento da monitorização e armazenamento de informações pessoais por parte de aplicações e serviços online, a segurança é mais do que nunca uma característica de vital importância. O risco de informação sensível ser interceptada por partes não autorizadas, ou o local onde esta é guardada estar desprotegido, são preocupações reais que os utilizadores enfrentam quando escolhem utilizar um serviço.

Nas tecnologias de informação a segurança pode ser avaliada de acordo com 6 atributos:

- Confidencialidade,
- Integridade,
- Disponibilidade,
- Não-repúdio,
- Autenticação
- Autorização [1].

**Confidencialidade**

A informação é confidencial quando não é divulgada publicamente, isto é, mantém-se secreta. Exemplo: O número do cartão de crédito utilizado numa compra online deve ser mantido secreto.

**Integridade**

A informação mantém a sua integridade quando não é alterado o seu conteúdo sem autorização, ou seja, quando esta se mantém intacta. Exemplo: Um contrato enviado por e-mail não deve ser alterado após o e-mail ter sido enviado.

**Disponibilidade**

A informação mantém a sua disponibilidade enquanto o seu acesso for possível. Caso o acesso à informação seja impedido existe uma quebra de disponibilidade. Exemplo: um *web site* deve manter-se disponível quando recebe um ataque de *Denial of Service*.

Um dos aspectos mais importantes para garantir a disponibilidade de um serviço é eliminar a possibilidade de uma falha individual de um componente tornar todo o serviço indisponível. Como consequência, o desafio passa por replicar o máximo possível da infraestrutura, isto é, criar redundância, para que exista sempre um mecanismo de substituição de qualquer componente em falha por outro saudável (*failover*).

Outra técnica é a de repartir trabalho por várias máquinas. A esta técnica dá-se o nome de *load-balancing*. Ao utilizar várias máquinas em simultâneo é possível lidar com a falha de uma máquina simplesmente redirecionando o tráfego que seria para essa máquina para outras máquinas saudáveis. No entanto, esta técnica assenta frequentemente numa só máquina, denominada *load-balancer*, que gere o tráfego antes deste chegar aos servidores. A possibilidade de falha desta máquina é, assim, também um risco porque torna todo o serviço indisponível, contrariando o seu próposito. Não obstante, existem também formas de contornar esta limitação [2].

Conclui-se assim que não existe uma forma infalível de alcançar disponibilidade total. Porém, existem diversas técnicas e protocolos que garantem um aumento considerável da disponibilidade de um serviço.

Para sistemas críticos é ainda utilizado o conceito de alta disponibilidade, onde a disponibilidade alcançada é superior a 99% (3,65 dias de indisponibilidade durante um ano) [3] [4].

No Capítulo 2.5 são apresentados vários protocolos de redundância capazes de garantir o *failover* e o consequente aumento de disponibilidade de um serviço.

**Não-repúdio**

O não-repúdio é a incapacidade de uma entidade negar a sua participação em determinada ação. O seu objetivo é o de aumentar a confiança entre os participantes de uma operação. Exemplo: Uma pessoa não pode negar a sua participação na assinatura de um contrato quando esta foi presenciada por um notário.

**Autenticação**

A autenticação é o ato de confirmar uma dada informação ou entidade como verídico. Exemplo: A utilização uma impressão digital para iniciar sessão num telemóvel.

**Autorização**

A autorização é a capacidade de apenas um utilizador autorizado aceder a determinada informação. O utilizador está autorizado quando a entidade responsável pela informação permite de forma deliberada o seu acesso a esta. Exemplo: Bloqueio do acesso a funcionalidades ou programas de um sistema operativo. Só os utilizadores autorizados terão acesso a esses programas e funcionalidades.

## 2.2. Algoritmos Criptográficos

Para alcançar os atributos de segurança acima referidos foram desenvolvidos algoritmos criptográficos que se baseiam em cifras matemáticas extensivamente estudadas e validadas. Entre estas destacam-se as chaves simétricas, as chaves assimétricas e as funções *hash*.

**Chaves Simétricas**

Chaves simétricas é o nome que se dá a um par de chaves idênticas utilizadas por diferentes atores para trocar informação codificada/encriptada. Para que esta técnica seja eficaz, uma chave secreta é criada e partilhada entre os intervenientes que querem comunicar de forma segura, através de um meio secreto (em pessoa por exemplo).

Após ambos possuírem a chave, estes podem começar a trocar mensagens num meio não seguro (Internet por exemplo), utilizando a chave para encriptar/codificar e desencriptar/descodificar as mensagens.

Entre os algoritmos de chave simétrica mais utilizados estão o AES, o Blowfish e o 3DES.

Na figura seguinte ilustra-se uma mensagem encriptada e desencriptada de forma segura, utilizando chaves simétricas:

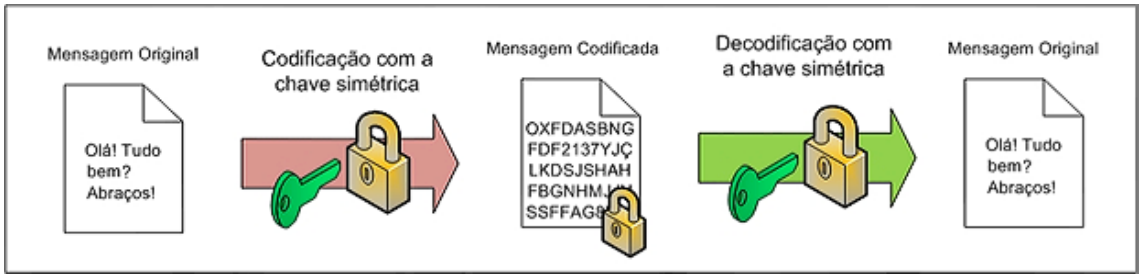



*Figura 2 - Exemplo da encriptação e desencriptação de uma mensagem utilizando chaves simétricas*

Como se pode verificar na figura 2, a mesma chave (a verde) é capaz de encriptar e desencriptar a mensagem. Este facto reitera a importância de manter a chave secreta apenas conhecida entre os atores autorizados, uma vez que quem tiver esta chave consegue não só ler a mensagem como também enviar novas mensagens.

Uma propriedade importante da criptografia de chaves simétricas é o facto de o processo de encriptação e desencriptação ser bastante leve a nível computacional.

As chaves simétricas são utilizadas para garantir confidencialidade.

**Chaves Assimétricas**

Chaves assimétricas é o nome que se dá a um par de chaves distintas. Uma das chaves designa-se por chave pública e a outra por chave privada. Ambas estão diretamente associadas a apenas um ator e considera-se que a relação entre elas é muito forte.

Quer isto dizer que, para uma mensagem encriptada com a chave pública apenas se consegue desencriptar essa mensagem com a chave privada e vice-versa.

Em contraste com as chaves simétricas, nas chaves assimétricas não existe necessidade de trocar nenhuma chave de forma secreta. A chave pública pode ser tornada pública sem qualquer problema e a chave privada mantém-se privada, garantindo assim que só o seu possuidor poderá desencriptar mensagens encriptadas com a sua chave pública.

Entre os algoritmos de chave assimétrica mais utilizados estão o RSA, o DSA e o ECDSA. O mecanismo de troca de chaves Diffie-Hellman também utiliza chaves assimétricas.

Na figura seguinte ilustra-se uma mensagem encriptada e desencriptada de forma segura, utilizando chaves assimétricas:

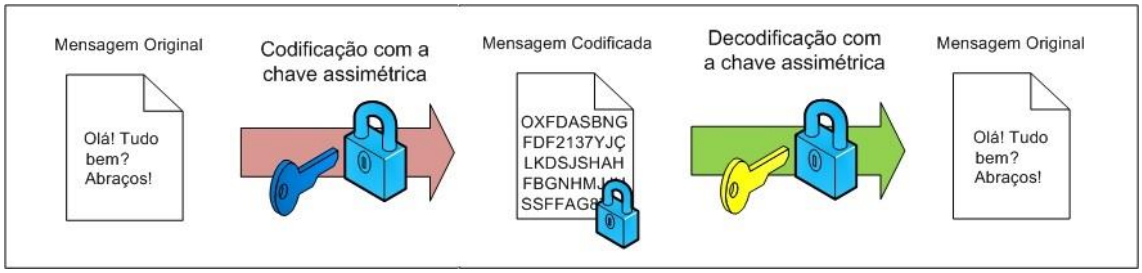



*Figura 3 - Exemplo da encriptação e desencriptação de uma mensagem utilizando chaves assimétricas*

Vamos supor, neste caso, que a chave azul é a chave pública e a chave amarela é a chave privada de um ator. É possível verificar que para uma mensagem encriptada com a chave pública apenas a outra chave, a privada, é capaz de a desencriptar. O contrário também se verifica.

Este método oferece assim vantagens em relação ao método anterior, uma vez que é possível encriptar informação especificamente para um só destinatário utilizando uma chave que é pública e conhecida por todos.

As chaves assimétricas são utilizadas para garantir confidencialidade, não-repúdio e autenticação.

**Função *Hash***

As funções *hash* são algoritmos que geram um valor único para uma dada informação. Ao contrário das chaves simétricas e assimétricas, estas são desenhadas de modo a que seja impossível reverter uma mensagem codificada com uma função *hash*, ou seja, descodificar a mensagem. Contudo, a título exemplificativo, para o algoritmo MD5 já foram descobertas técnicas de ataque que possibilitam quebrar a integridade de uma mensagem codificada por este.

Entre os algoritmos de função *hash* mais utilizados estão o SHA1 e o SHA2.

A figura seguinte ilustra o funcionamento de uma função *hash*:

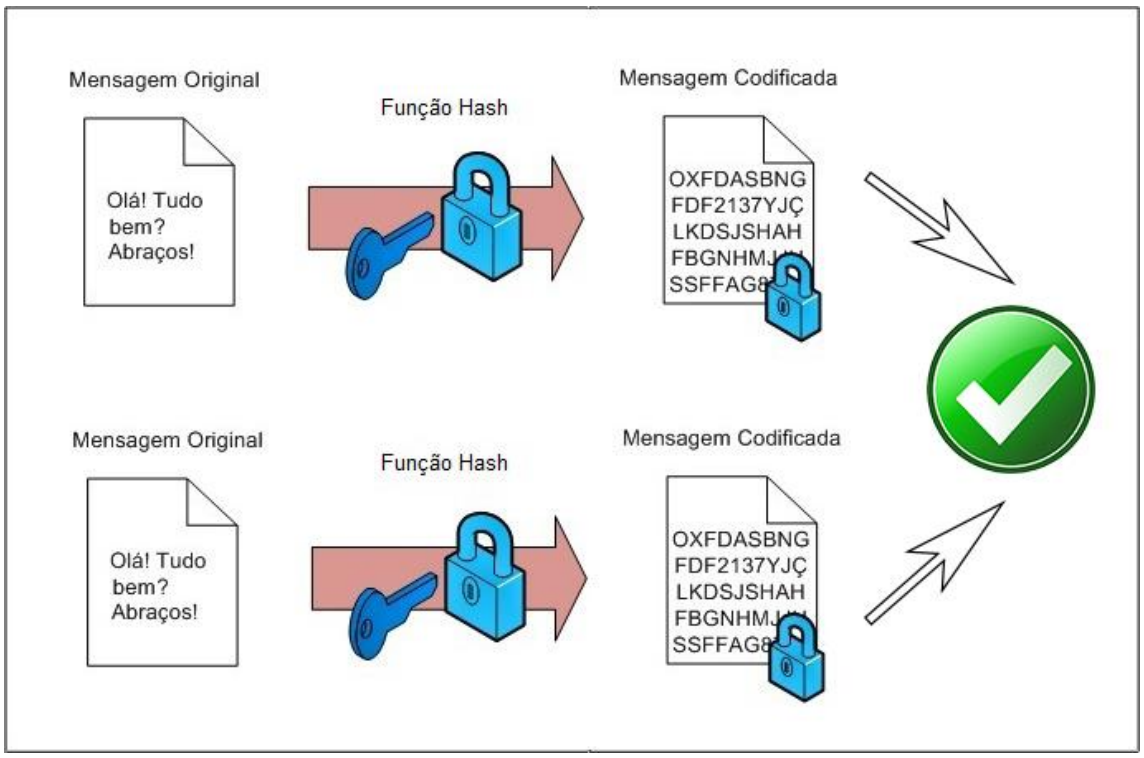



*Figura 4 - Exemplo de uma verificação bem sucedida utilizando uma função hash*

Ao valor único resultante da codificação da mensagem através de uma função *hash* dá-se normalmente o nome de *hash*. Este valor é geralmente enviado em conjunto com a mensagem. O recetor para validar a integridade da informação apenas necessita de repetir a operação de aplicar a função de *hash* na mensagem e depois comparar o valor obtido com o valor recebido. Caso os valores coincidam comprova-se que a mensagem não foi adulterada. Caso contrário, há a confirmação de que a mensagem foi alterada.

Como conclusão percebe-se que a melhor forma de garantir os diferentes atributos é conjugar diferentes algoritmos. É exactamente isso que o protocolo SSL/TLS faz e que está descrito no próximo capítulo.

## 2.3. Secure Socket Layer (SSL) / Transport Security Layer (TLS)

O SSL ou a sua evolução, o TLS, é um protocolo criptográfico criado com o intuito de possibilitar comunicações seguras através da Internet. As motivações que levaram à criação destes protocolos são óbvias, uma vez que a Internet foi desenhada sem quaisquer preocupações de segurança em mente. Por consequência, qualquer comunicação poderia ser escutada e alterada por terceiros.

Com o surgimento destes protocolos tornou-se possível comunicar de forma segura através da Internet, garantindo a autenticidade do servidor com que estamos a comunicar e que a informação transmitida não é alterada por terceiros. Lojas online ou qualquer *website* que trabalhe com informação pessoal ou privilegiada usa o protocolo SSL/TLS para proteger a comunicação com os seus utilizadores.

O objectivo principal do SSL/TLS é assegurar a segurança criptográfica das comunicações. Outros objectivos incluem garantir a interoperabilidade com outros programas e bibliotecas, a extensibilidade em relação a novas cifras ou funções de *hash* e, por fim, a eficiência relativamente ao desempenho das operações criptográficas.

Vamos analisar em maior detalhe a versão mais recente suportada pelo OpenVPN (na sua versão 2.4.1) deste protocolo, a TLS 1.2 [5].

Uma suite de cifras utilizando o protocolo SSL/TLS é definida por [6]:

- Método de troca de chaves
- Método de autenticação
- Algoritmo de encriptação
- Tamanho da chave de encriptação
- Modo de Cifra (quando aplicável)
- Algoritmo MAC (quando aplicável)
- Função Pseudo-Aleatória (PRF - *Pseudo Random Function*)
- Função *hash* utilizada

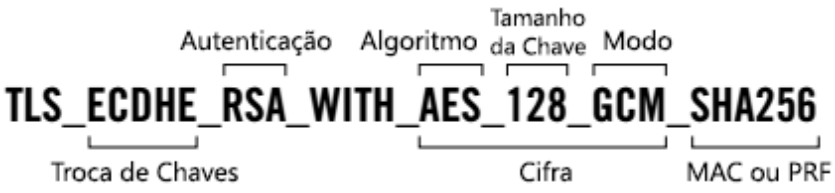



*Figura 5 - Construção do nome da suite de cifras utilizadas*

A essência do protocolo SSL/TLS está no chamado *handshake*, o aperto de mão dado entre o cliente e o servidor. No entanto, existem vários protocolos dentro do SSL/TLS que especificam a sua implementação e funcionamento.

**Protocolo Record**

O protocolo Record é responsável pela implementação de alto nível do SSL/TLS. Este é responsável por vários aspetos importantes na comunicação cliente-servidor: transporte, encriptação, validação da integridade e compressão de mensagens. Além disso é extensível. A extensibilidade do protocolo Record é uma característica importante, porque às suas funcionalidades podem ser acrescentadas outras através de subprotocolos.

A especificação SSL/TLS define 4 subprotocolos principais [6]:

- Protocolo Handshake
- Protocolo Change Cipher Spec
- Protocolo Application Data
- Protocolo Alert

**Protocolo Handshake**

O aperto-de-mão, ou *handshake*, é como todas as ligações SSL/TLS começam. Existem 3 tipos de *handshake*: total, abreviado ou com autenticação de cliente e servidor.

O *handshake* total acontece quando é feita pela primeira vez uma ligação entre cliente e servidor. Este é composto por 4 fases: troca de parâmetros de ligação, validação do certificado ou autenticação por outro meio, geração do *master secret*, verificação que as mensagens do *handshake* não foram alteradas por terceiros [7].

O *handshake* abreviado é o resumo de uma sessão previamente iniciada. Isto é possível na medida em que tanto o servidor como o cliente guardam parâmetros de segurança durante algum tempo após o término de uma ligação. Sendo assim, quando o cliente inicia o *handshake,* envia um ID previamente definido na ligação anterior para que o servidor saiba que este quer resumir a ligação. Caso o servidor concorde em resumir a sessão reutiliza o *master secret* da ligação anterior e inicia a nova ligação de imediato.

O *handshake* com autenticação de cliente e servidor é igual ao *handshake* total com a diferença de que o servidor requisita um certificado ao cliente para autenticar a sua permissão para estabelecer uma ligação.

- Troca de Chaves

A segurança de uma sessão SSL/TLS é assegurada por uma chave partilhada de 48-bytes chamada *master secret*. A troca de chaves gera o *premaster secret*, o qual serve de base para a construção do *master secret*.

A troca de chaves acontece de forma diferente consoante o algoritmo utilizado. O principal algoritmo suportado é o RSA. Este é utilizado pelos métodos Diffie-Hellman Efémero (DHE) e Diffie-Hellman de Curva Elíptica (ECDHE).

- Autenticação

A autenticação é aplicada como consequência da troca de chaves. O algoritmo RSA é normalmente utilizado para fazer a autenticação do cliente. Devido à utilização de uma estrutura de chaves públicas, a autenticação está implícita no processo de troca de chaves.

- Encriptação

A informação transmitida no protocolo SSL/TLS pode ser encriptada utilizando um de vários algoritmos, sendo o AES o mais popular. Há que considerar também o tamanho da chave de encriptação, utilizando-se 128 bits quando a velocidade é um requisito ou 256 bits quando se pretende proteção máxima.

- Renegociação

A renegociação é um processo que pode acontecer a meio de uma sessão. A sua utilização é útil em situações em que seja necessário alterar parâmetros de segurança da ligação. As razões podem ser a adição/remoção de certificados ou a mudança do algoritmo de encriptação para maior segurança ou maior velocidade.

**Protocolo Change Cipher Spec**

O protocolo Change Cipher Spec consiste numa mensagem normalmente enviada no final do *handshake* pelo cliente para o servidor e, no sentido inverso, para notificar quais as definições de encriptação em uso. Pode servir para alterar a suite de cifras em uso na ligação.

**Protocolo Application Data**

Este protocolo é responsável pela transferência de mensagens da aplicação que, para o SSL/TLS, são apenas pedaços de informação. O protocolo Record é o que tem a responsabilidade de organizar os pedaços de informação em pacotes e depois aplicar a encriptação.

**Protocolo Alert**

O protocolo Alert implementa um mecanismo de notificações simples que tem como propósito informar um dos lados da conversação de que algo de excecional aconteceu no outro lado. Este mecanismo é utilizado geralmente para notificações de erros ou avisos.

A figura seguinte mostra o processo que acontece durante um aperto de mão (*handshake*) SSL/TLS:

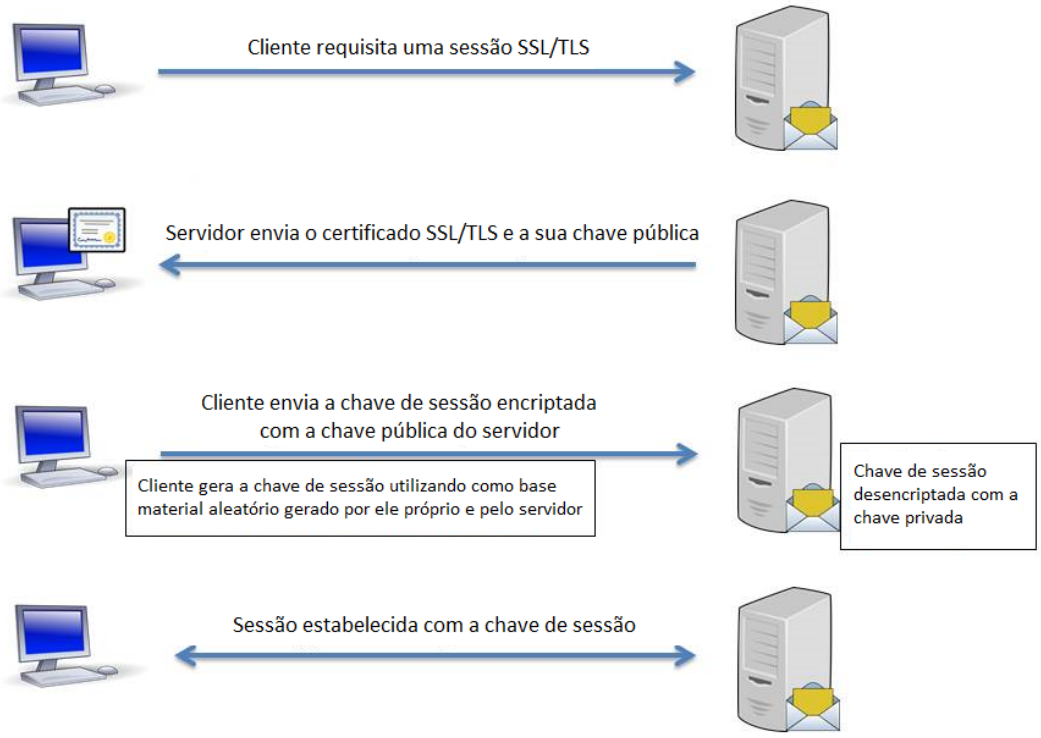



*Figura 6 – Representação das ações envolvidas durante um aperto de mão SSL/TLS*

Pode verificar-se na figura acima que o aperto de mão SSL/TLS é constituído por 3 passos. No primeiro passo o cliente faz um pedido de início de uma sessão SSL/TLS ao servidor de um serviço. No segundo passo o servidor envia o seu certificado SSL/TLS, que valida a sua identidade, e também a sua chave pública, que faz parte de um algoritmo de chaves assimétricas tal como explicado no Capítulo 2.2. No terceiro passo o cliente gera uma chave simétrica, abordada também no Capítulo 2.2, que será utilizada como chave de sessão.

Para conseguir partilhar a chave de sessão de forma segura com o servidor o cliente encripta a chave de sessão com a chave pública do servidor. Logo, só é possível ao servidor obter a chave de sessão através da desencriptação da comunicação com a sua chave privada.

A partir do momento em que ambos possuem a chave de sessão, todas as comunicações são efetuadas encriptando o conteúdo com a chave de sessão.

## 2.4. Virtual Private Network (VPN)

VPN significa Virtual Private Network e o seu objectivo é proteger a privacidade da informação trocada entre duas (ou mais) máquinas na Internet.

Como a Internet não oferece qualquer mecanismo de base para encriptação da informação trocada, ao longo do tempo foram sendo criados mecanismos de autenticação e proteção de dados como o *https* e o SSL/TLS. No entanto, todo o tráfego transmitido continua a poder ser intercetado e a ser possível pelo menos identificar a sua origem e destino.

As VPNs oferecem uma solução para esse problema. Criando uma rede "local" privada em que as ligações entre as várias máquinas são efetuadas através de túneis, a informação transmitida passa a estar encriptada e deixa de ser possível ser acedida e analisada por terceiros. Além disso, suporta mecanismos de autenticação para que só as máquinas autorizadas possam aceder à sua rede privada. A figura seguinte mostra o funcionamento de uma ligação VPN:

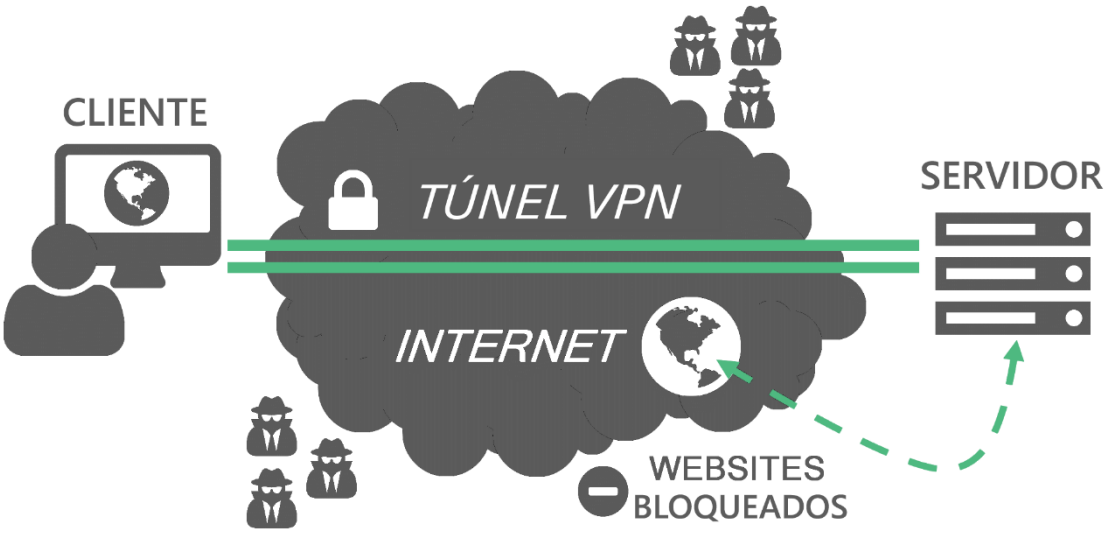



*Figura 7 - Representação do funcionamento de uma VPN*

Os casos de uso para a utilização de uma VPN são variados, podendo ir desde o foro pessoal até ao empresarial [8] [9]. Aqui constam alguns exemplos:

- Comunicação de forma segura por um utilizador numa Wi-Fi pública
- Comunicação de máquinas de uma empresa com filiais em vários pontos do mundo
- Proteção da identidade de um jornalista que escreve uma história com implicações políticas

- Acesso a conteúdos geograficamente bloqueados através da ligação a servidor presente em território não bloqueado
- Acesso a informação censurada por governos ou organizações políticas.

Ao usar uma VPN é necessário configurar uma ligação ao servidor VPN. Este servidor pode ser contratado a uma empresa que forneça serviços VPN, que terá os recursos para oferecer vários servidores em vários países por um preço fixo mensal/anual [10]. Ou, por outro lado, é possível também construir um servidor VPN caseiro, o que proporciona maior controlo sobre o software e as configurações utilizadas. O software utilizado e os detalhes da configuração dependem do tipo de VPN utilizada pelo serviço/servidor doméstico. Sendo assim existem várias opções, mais fáceis ou difíceis de configurar, que oferecem diferentes níveis de proteção. Como tipos de VPNs existem [9]:

- VPNs baseadas no protocolo PPTP
- VPNs baseadas no protocolo IPSec
- VPNs baseadas em SSL/TLS
- OpenVPN (que também é baseado em SSL/TLS mas é bastante diferente das VPNs baseadas em SSL)

Através da recolha de informação de várias fontes, foram compiladas as características e qualidades dos vários tipos de VPNs [9] [11] [12] [13] [14] [15] [16]. Para comparar as diferentes VPNs temos a seguinte tabela:

*Tabela 1 - Comparativo entre diferentes soluções VPN*

| VPNs / Características | PPTP | IPSec | SSL/TLS | OpenVPN |
|---|---|---|---|---|
| Facilidade de Configuração | Fácil | Difícil | Fácil | Média |
| Segurança (Resistência a ataques) | Fraca | Forte | Média | Forte |
| Handshake | PAP, CHAP, MS-CHAP v1/v2 | Diffie-Hellman | RSA-2048, RSA-4096 | RSA-2048, RSA-4096 |
| Autenticação de Dados | - | MD5, SHA1, SHA2 | SHA1, SHA2 | SHA1, SHA2 |
| Criptografia de Dados | MPPE (128-bit) | 3DES, AES-128, AES-256 | 3DES, Blowfish, AES-128, AES-256 | 3DES, Blowfish, AES-128, AES-256 |
| Protocolos/Bibliotecas Utilizados | GRE | ESP, AH, IPComp, IKE | OpenSSL | OpenSSL, LibreSSL, PolarSSL |
| Portos | TCP 1723 e GRE (Protocol 47) | UDP 500 para a troca inicial de chaves, ESP (Protocol 50), UDP 1701 para configuração inicial L2TP e UDP 4500 para NAT traversal. | TCP 443 | TCP 443, UDP 1194 (por defeito). Podem ser configurados outros portos. |
| Suporte por SOs | Windows, Linux, BSD, OSX, iOS, Android, DD-WRT | Windows, Linux, BSD, OSX, iOS, Android | Windows, Linux, BSD, OSX, iOS, Android | Windows, Linux, BSD, OSX, iOS, Android, DD-WRT |
| Software Cliente | Não Necessário | Não Necessário | Necessário | Necessário |

Com base na tabela anterior é possível formular a seguinte tabela:

*Tabela 2 - Comparativo dos atributos de segurança de diferentes soluções VPN*

| VPNs / Características | PPTP | IPSec | SSL/TLS | OpenVPN |
|---|---|---|---|---|
| Garante Autenticação dos Dados | × | √ | √ | √ |
| Garante Integridade dos Dados | × | √ | √ | √ |
| Garante Confidencialidade dos Dados | √ | √ | √ | √ |

É possível verificar que o PPTP é claramente o protocolo mais inseguro dos 4. Isto deve-se às suas graves falhas de segurança, que impedem a garantia de autenticação dos intervenientes e integridade dos dados transferidos.

Devido às vulnerabilidades do PPTP, iremos focar-nos apenas nas VPNs baseadas nos protocolos IPSec, SSL/TLS e na OpenVPN.

**VPNs baseadas no protocolo IPSec**

O IPSec vem incorporado no standard IPv6 e é o protocolo standard da IEEE/IETF para segurança de redes baseadas em IP. Este protocolo funciona nas camadas 2 e 3 do modelo OSI. Uma característica diferenciadora deste protocolo é a possibilidade de o administrador criar e configurar políticas de segurança para cada ligação. É possível encriptar apenas o tráfego entre duas máquinas baseando-se no IP de origem/destino ou nos portos utilizados. Isto aumenta a flexibilidade, mas também a complexidade de configurar e dar suporte a este tipo de redes.

Tal como o PPTP, o IPSec usa também dois canais de comunicação: um canal de controlo, no porto UDP 500 ou 4500 para iniciar a ligação; e um canal de informação que utiliza Encapsulated Security Payload (ESP), que é o protocolo 50 de IP. A integridade dos pacotes é garantida através do Hash-based Message Authentication Code (HMAC).

O IPSec pode funcionar em Tunneling Mode ou em Transport Mode sendo que este modo é normalmente usado em combinação com o Level 2 Tunneling Protocol (L2TP).

Vantagens:

- Forte autenticação e encriptação
- Bom suporte por parte de diferentes operadores de serviços
- Implementação de políticas de segurança detalhadas

Desvantagens:

- Difícil de configurar
- Não se integra bem em redes NAT
- Diferentes implementações de diferentes operadores por vezes não funcionam em conjunto. [17]

**VPNs baseadas no protocolo SSL/TLS**

As VPNs baseadas em SSL são as mais populares hoje em dia. Muitas vezes estas VPNs são denominadas de *Web-based* VPNs, devido a usarem o mesmo protocolo dos websites seguros (*https*). Apesar de não existir um standard de configuração, na maioria dos casos são utilizados um nome de utilizador e palavra-passe para autenticar os clientes e certificados X.509 para encriptar a ligação. O porto de comunicação normalmente utilizado é o TCP 443, o mesmo do *https*.

A característica diferenciadora deste tipo de VPNs é o facto de o utilizador para se ligar apenas necessitar de um navegador web (*browser*).

Vantagens:

- Utilização diretamente no *browser*
- Fácil de utilizar

Desvantagens:

- Pouco robusto (dificuldade em partilhar ficheiros locais ou ligar-se a vários servidores)

**OpenVPN**

Segundo o seu próprio código fonte, o OpenVPN é uma aplicação que transmite dados de redes IP através de um túnel seguro utilizando um só porto TCP/UDP. Além disso suporta encriptação, autenticação e compressão de pacotes, bem como autenticação de sessão e troca de chaves baseada no protocolo SSL/TLS.

O OpenVPN é uma solução *open-source* baseada em SSL/TLS, mas também em HMAC, daí não ser catalogada como fazendo parte das VPNs baseadas no protocolo SSL/TLS. A sua flexibilidade possibilita a configuração de chaves pré partilhadas, certificados X.509 e HMAC.

Esta solução utiliza também dois canais: um de controlo e outro de informação. No entanto, todo o tráfego passa por um único porto por TCP ou UDP. Por defeito, é utilizado o porto UDP 1194.

Para gerir as ligações é criada uma placa de rede virtual (TUN ou TAP) que serve de interface para o OpenVPN. No TUN apenas o tráfego IP é transmitido pelo túnel VPN e o encapsulamento é efetuado na camada 3 do modelo OSI, enquanto que no TAP todo o tráfego de rede é transmitido através do túnel VPN uma vez que o encapsulamento é efetuado na camada 2 do modelo OSI. Uma das diferenças visíveis entre TUN e TAP é o seu desempenho, na medida em que o TUN como lida com menor número de pacotes encriptados vai ter um maior desempenho.

Por último, é importante referir que todos, ou quase todos, os sistemas operativos suportam OpenVPN, apesar da necessidade de instalação de software adicional (ao contrário das VPNs baseadas em SSL). O facto deste protocolo ser *open source* é uma vantagem dado que o código é continuamente inspecionado por peritos em segurança. As possibilidades de existência de uma *backdoor* são, por consequência, bastante remotas [18] [9] [19].

Vantagens:

- Flexibilidade de configuração
- Forte autenticação e encriptação
- Suporte por parte de todos os principais sistemas operativos

Desvantagens:

- Necessita instalação de software no lado do cliente

## 2.5. Protocolos de Redundância

Depois de abordar o que é uma VPN, e quais os diferentes tipos de VPNs que existem, vamos focar-nos em disponibilidade. A alta disponibilidade na grande maioria das soluções é garantida através de redundância espacial, ou seja, cópias do mesmo componente que atuam em caso de falha do componente principal. Neste caso a redundância/*failover* será implementada ao nível dos servidores VPN, o que resulta em dois ou mais servidores com as configurações iguais à do original.

Os protocolos de redundância principais são [20]:

- Hot Standby Router Protocol (HSRP)
- Virtual Router Redundancy Protocol (VRRP)
- Common Address Redundancy Protocol (CARP)

**Hot Standby Router Protocol**

O HRSP foi concebido com a necessidade de providenciar *failover* de routers em redes Ethernet LAN.

O conceito é agrupar dois routers num router virtual com um endereço MAC e IP próprio diferente de cada um dos routers físicos. O HSRP possibilita ainda a criação de vários routers virtuais dentro da mesma rede. Apesar de cada um dos routers físicos poder receber pacotes destinados ao router virtual apenas um deles - o router ativo - transmite os pacotes recebidos.

O router ativo é escolhido através de um processo de eleição, ficando o outro em reserva. Após a escolha do router ativo, este e o de reserva transmitem periodicamente uma mensagem entre si (opcionalmente encriptada com uma password de 8 caracteres) para detetarem possíveis falhas de um ou outro. Caso o router ativo falhe o router em reserva toma o seu lugar e um novo router de reserva é eleito. Caso aconteça o oposto, o router ativo continua ativo e é eleito um novo router em reserva [20] [21].

**Virtual Router Redundancy Protocol**

O VRRP é bastante semelhante ao HSRP, apenas com algumas diferenças. Continua a existir um processo de eleição de router ativo e em standby, mas no VRRP são eleitos vários routers de standby cada um com grau diferente de prioridade. O que tiver maior grau de prioridade naquele momento é o que substitui o router ativo em caso de falha e assim sucessivamente.

Além disso no VRRP são utilizados ICMP *redirects*, um mecanismo que possibilita aos routers enviar informação de encaminhamento para as máquinas de destino. Este mecanismo é útil em redes assimétricas cujos pacotes transmitidos numa direção diferem daqueles que são transmitidos na direção contrária. Convém ainda referir que o HSRP não permite a utilização de ICMP *redirects* porque estes revelam os endereços MAC dos routers físicos [20] [22].

**Common Address Redundancy Protocol**

O CARP é um protocolo de redundância de *routers*, *firewalls* ou até de servidores *web*. Tal como o VRRP, o CARP é um protocolo *multicast* que agrupa várias máquinas num grupo virtual chamado CARP *group*. Este grupo possui um único endereço MAC e IP partilhado entre as suas várias máquinas.

Nesse grupo, uma máquina é eleita como ativa/*master*, enquanto as restantes máquinas ficam como *backup*, tendo cada uma prioridade definida para substituir o *master* caso este falhe. As prioridades podem ser definidas manualmente para que se possa dar a uma determinada máquina maior probabilidade de ser *master*. O CARP obriga todas as máquinas a terem o seu próprio endereço MAC e IP além dos endereços virtuais.

Além disso, todos os endereços IP do mesmo grupo têm de estar na mesma sub-rede [20] [23].

Quando a máquina que está como *master* num dado momento deixa de funcionar por qualquer razão, o CARP elege uma nova máquina como *master* e começa a redirecionar o tráfego para a nova máquina. Visto que o CARP não guarda qualquer tipo de estado da ligação, a informação que estava a ser trocada como o *master* original é perdida.

Para replicação de informação entre o grupo é necessário utilizar outras ferramentas como o *rsync* para replicação de ficheiros e o *pfsync* para replicação de regras/estado da *firewall* [24] [25].

## 2.6. Ferramentas de Clustering

Como ferramentas de *clustering* os softwares mais conhecidos são o Pacemaker, o Corosync/Heartbeat e o DRBD. Estes softwares têm propósitos diferentes e podem trabalhar todos em conjunto. Além destes existe o CARP, descrito na secção anterior, que existe enquanto protocolo e ferramenta de *clustering* para distribuições BSD.

O Pacemaker é um gestor de recursos em *cluster* disponível para Linux. Para ser utilizado em distribuições BSD tem de ser compilado a partir do código fonte, o que não é aconselhado. As suas funcionalidades incluem o arranque e terminação de serviços, bem como a monitorização do *cluster* para garantir um funcionamento sem falhas e evitar corrupção da memória.

O Heartbeat é um motor de *cluster* cujo intuito é providenciar uma infraestrutura (comunicação e sociedade) entre máquinas no mesmo *cluster*. Este motor foi desenvolvido para Linux mas encontra-se praticamente descontinuado, tendo sido substituído pelo Corosync [26]. O Corosync é um motor de *cluster* mais avançado e desenvolvido em estreita relação com o Pacemaker, oferecendo funcionalidade extra.

O Distributed Replicated Block Device (DRBD) é um sistema de replicação distribuída de armazenamento desenvolvido para Linux [27]. Este é geralmente utilizado em *clusters* de alta disponibilidade para acesso a informação armazenada em memória. O que este faz é replicar informação pelos vários nós do *cluster*. Outra ferramenta de replicação de armazenamento é o *rsync*, também desenvolvido para Linux mas não tão popular [24].

Tanto o Pacemaker como o Corosync têm origens na Red Hat e atualmente estão sob a alçada da ClusterLabs [28]. A empresa Linbit desenvolve o DRBD e providencia também suporte aos outros produtos.

## 2.7. Soluções Comerciais de Alta Disponibilidade

No mercado existem várias implementações de alta disponibilidade com opções de clustering. Empresas como a Cisco, Barracuda, SonicWall, NetScaler ou Watchguard desenvolveram soluções próprias para funcionarem em conjunto com o seu *hardware*.

No entanto, para partilha de estados relativos a VPNs, apenas a Cisco apresenta soluções integradas. Através da sua plataforma AnyConnect, oferece partilha de estados para as VPNs baseadas em protocolos SSL/TLS e IPSec [29] [30]. Em todas as outras soluções de empresas concorrentes a partilha de estados não contempla túneis VPN.

Vantagens:

- Integração *hardware/software*
- Bom suporte

Desvantagens:

- Custo
- Código fechado

## 2.8. Protocolos de Sincronização de Estados

Para melhor compreender o desafio de sincronizar estados de aplicações impõe-se o estudo de soluções de sincronização de estados. Foram selecionados o *pfsync* e o *dhcpd*, que são ambas soluções com funcionalidade semelhante àquela que é pretendida alcançar nesta dissertação.

**pfsync**

O *pfsync* é uma interface de sincronização de regras de firewall do OpenBSD, a pf (Packet Filter). Quando configurado, o *pfsync* envia atualizações de alterações nas regras da firewall do nó principal para os outros nós de um *cluster* através de *multicast*. Todas as atualizações são enviadas sem qualquer autenticação.

Esta solução é implementada ao nível do Kernel e cria uma interface própria para fazer a comunicação entre máquinas. Por consequência, compreender o seu código fonte é deveras complicado [25].

**dhcpd**

O *dhcpd* é o servidor de Dynamic Host Configuration Protocol (DHCP). Este gere a atribuição de endereços IP a máquinas de uma rede de forma dinâmica. Quando um cliente pede um endereço IP o dhcpd atribui-lhe um IP e um *lease time*, que é o tempo em que esse endereço é válido. Os registos dos utilizadores e respetivos *lease times* são guardados no ficheiro *dhcpd.leases*. Quando o *lease time* termina, o IP expira e deixa de ser válido. Assumindo este funcionamento, para um *cluster* de servidores DHCP existe a necessidade de sincronizar os endereços de IP que estão atribuídos e qual o seu *lease time*.

Através da configuração das *flags* -y e -Y é possível ativar a sua funcionalidade de sincronização do *dhcp*. Esta funcionalidade permite sincronizar os endereços IP atribuídos e os respectivos *lease times*. A sincronização é feita através do envio e receção de mensagens através de uma interface de rede normal de sistema, definida na inicialização do *dhcpd* em conjunto com a *flag*.

Utilizando a *flag* -Y define-se uma interface local para o envio de mensagens *multicast*, ou um endereço IP/domínio para o qual enviar mensagens *unicast*. Com a *flag* -y define-se uma interface de receção de mensagens, que pode ser *unicast* ou *multicast*.

Com a funcionalidade de sincronização ativa, os servidores *dhcpd* atualizam o seu ficheiro *dhcpd.leases* consoante as mensagens que receberem dos outros servidores. Este é um processo simples, já que apenas envolve a escrita ou remoção de dados contidos num ficheiro. Deste modo, todos os servidores terão o ficheiro com os mesmos dados e não existirão conflitos. [31]

O código do *dhcpd* e, por conseguinte, da funcionalidade de sincronização, é escrito na linguagem de programação C. A sua análise, comparativamente ao *pfsync*, resulta como mais acessível e fácil compreender. Como resultado, alguns dos seus conceitos foram utilizados na solução desenvolvida.

## 2.9. Ferramentas Escolhidas

Nesta secção enunciam-se as ferramentas utilizadas no decorrer deste estágio/dissertação. Para cada ferramenta expõem-se ainda as razões que motivaram a sua escolha.

**VirtualBox VM**

De modo a facilitar a configuração de várias máquinas e ter flexibilidade para efetuar os mais variados testes, foi tomada a decisão de utilizar máquinas virtuais. Desta forma, a escolha do *software* de virtualização recaiu sobre a VirtualBox VM, uma vez que já me encontrava previamente familiarizado com este *software*. Isso permitiu rentabilizar tempo e recursos no conhecimento e na utilização de outros *softwares* necessários ao projeto.

**OpenBSD**

O OpenBSD é um sistema operativo *open source*, livre e Unix-like focado em segurança [32]. Um dos seus objetivos é ser o sistema operativo mais seguro do mundo. Além disso, é conhecido pela sua boa documentação e pelo facto de ter o seu código auditado com regularidade. A sua escolha deve-se ao facto de ser um sistema leve, altamente seguro, ter suporte ao OpenVPN e uma solução incorporada e fácil de configurar de *clustering* - o CARP. [33]

**OpenVPN**

O OpenVPN é o *software* VPN mais revisto e auditado pela comunidade de segurança. Duas auditorias recentes revelaram apenas vulnerabilidades de baixo impacto e que foram prontamente corrigidas [34] [35]. A sua natureza *open source* possibilita uma maior extensibilidade do seu código e a capacidade de introdução de novas funcionalidades. Sendo assim, e como o objetivo desta dissertação é a introdução de uma nova funcionalidade nos servidores, foi neste *software* que recaiu a escolha.

**CARP**

O CARP é o protocolo de *clustering* escolhido. Tem a vantagem de ser o protocolo de *clustering* já incluído no OpenBSD e incluir funcionalidades avançadas de configuração e segurança (utiliza criptografia). Dada a facilidade de configuração é também o mais indicado para fazer *cluster* dos servidores OpenVPN. Salientar também que é um protocolo sem patentes associadas, em contraste com os protocolos HSRP e VRRP.

**Wireshark**

O Wireshark é uma ferramenta de análise de tráfego que possibilita uma monitorização e compreensão do tráfego de rede. A sua utilização prende-se com a necessidade de analisar o tráfego entre cliente e servidor OpenVPN. Serve também para monitorizar e testar a solução a implementar de partilha de estado dos clientes entre servidores OpenVPN.

**Netcat**

O Netcat é o canivete suíço das ferramentas de rede. As suas capacidades incluem descoberta e escuta de portos, transferência de ficheiros, troca de mensagens, entre outros. Esta ferramenta utiliza os protocolos TCP e UDP e é útil na geração de tráfego entre cliente e servidor ou entre servidores.

Neste Capítulo 2 foram apresentadas e analisadas as tecnologias relevantes para o desenvolvimento da dissertação. A partir desta análise foram escolhidas as ferramentas a utilizar durante o projeto bem como recolhidas informações importantes para a formulação da solução desenvolvida. De seguida expõem-se as metodologias e o planeamento utilizado nesta dissertação.

# 3. Metodologias e Planeamento

Neste capítulo é expressa a metodologia utilizada, detalhando as várias etapas que constituíram o acompanhamento do projecto. São também apresentadas as tarefas executadas durante o primeiro e segundo semestres do projecto, incluindo a sua calendarização.

## 3.1. Metodologia Adotada

A metodologia adotada foi a metodologia SCRUM, devido à sua agilidade e à sua natureza iterativa. O SCRUM consiste em definir um conjunto de tarefas a executar no início do projeto (*product backlog*) que irão ser executadas em *sprints* individuais. Durante o decorrer do projeto são escolhidas um conjunto de tarefas a abordar de cada vez (*sprint backlog*) e é planeada a sua execução. À execução propriamente dita de cada tarefa dá-se o nome de *sprint*, ocupando normalmente um tempo entre 2 a 4 semanas [36].

Cada fase do projeto foi seguida por reuniões de acompanhamento. No final de cada *sprint* a reunião possibilitou uma revisão dos resultados obtidos e tomada de decisões sobre os *sprints* seguintes.

## 3.2. Primeiro Semestre

A planificação do calendário para o primeiro semestre foi efetuada com base no seguinte plano de trabalhos: pesquisa e análise do estado da arte, implementação de cenários com OpenVPN e CARP, formulação e estruturação de uma solução para o problema e, por fim, a escrita do relatório intermédio.

O diagrama Gantt da figura seguinte ilustra o plano delineado:

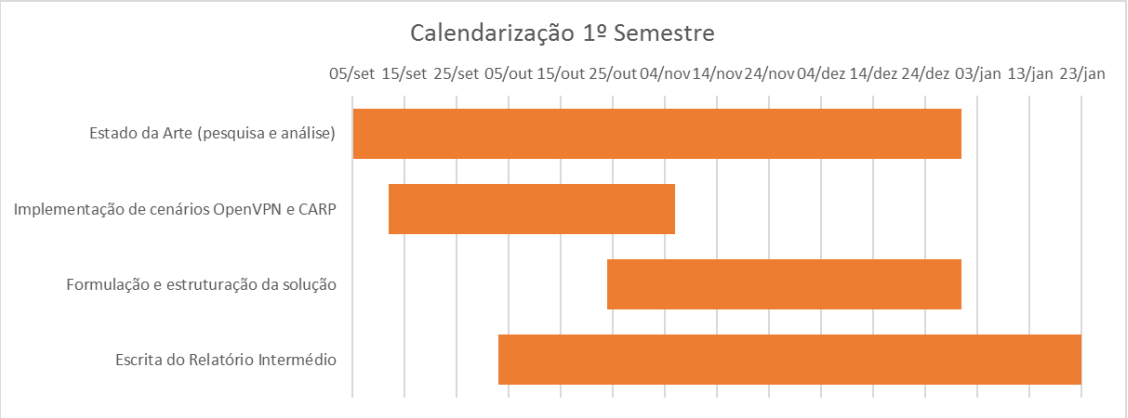



*Figura 8 – Calendarização para o 1º Semestre*

No estado da arte foram efetuados os seguintes *sprints*:

- Analisar o protocolo de segurança SSL/TLS
- Analisar vários tipos de VPNs, com especial enfâse no OpenVPN
- Analisar vários tipos de protocolos de redundância, com especial enfâse no CARP
- Analisar diferentes sistemas operativos focados em segurança
- Analisar ferramentas de *clustering*
- Analisar alternativas de implementação de *failover*
- Analisar soluções de sincronização de estados (pfsync e dhcpd com *flag* -y e -Y)
- Analisar código do pfsync para perceber qual o mecanismo que ele utiliza para sincronizar estados das *firewalls*
- Analisar código do DHCPd para perceber qual o mecanismo utilizado para sincronizar clientes e os seus tempos de *lease*

Na implementação em ambiente virtual de cenários OpenVPN e CARP foram efetuados os seguintes *sprints*:

- Instalação e configuração de máquinas virtuais através do software VirtualBox VM com o sistema operativo OpenBSD
- Instalação e configuração de servidores OpenVPN em máquinas virtuais
- Instalação e configuração de clientes OpenVPN em máquinas virtuais
- Ligação de clientes a servidores OpenVPN na mesma rede
- Ligação de clientes a servidores OpenVPN em redes diferentes
- Configuração do cluster de servidores utilizando CARP
- Configuração de clientes a ligarem-se ao cluster de servidores
- Teste de envio e receção de mensagens e ficheiros através do túnel VPN
- Teste de diferentes configurações de segurança e características do túnel VPN

Na formulação e estruturação da proposta de solução foi primeiramente analisado o código do OpenVPN, investigando qual a informação que é guardada sobre os túneis VPN entre cliente e servidor a cada momento. De seguida, e em consonância com os orientadores da empresa, foi detalhada a proposta de solução.

A elaboração do relatório intermédio de estágio foi realizada de forma progressiva, começando pela redação dos capítulos introdução e estado de arte e terminando no capítulo conclusão. O mês de janeiro foi o mês em que maior intervalo de tempo foi dispensado para esta atividade.

## 3.3. Segundo Semestre

Na segunda parte do estágio o objetivo foi o de implementar a solução formulada. Nesse sentido, o foco foi desenvolver o *software* que permite a transferência de informação do estado das ligações dos clientes ligados pelos vários servidores de forma sincronizada.

A figura seguinte ilustra a calendarização definida para o segundo semestre:

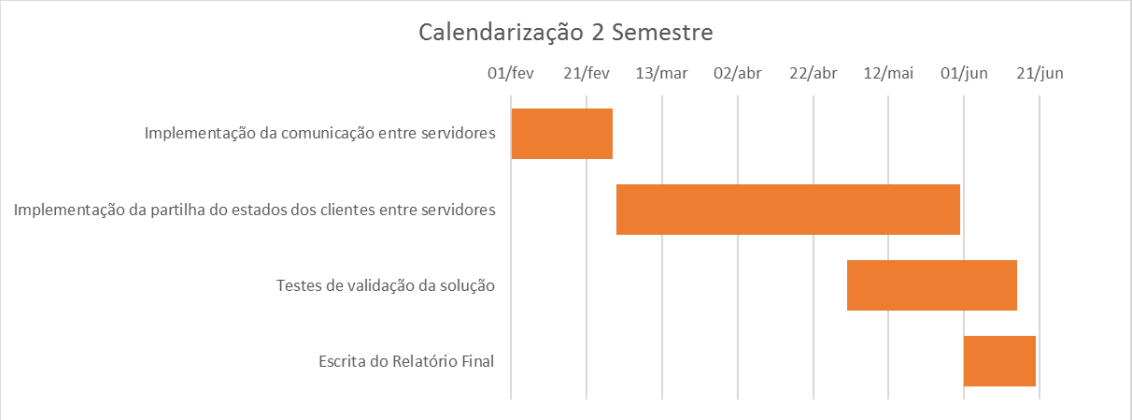



*Figura 9 - Calendarização para o 2º Semestre*

Na implementação da comunicação entre servidores foram efetuados os seguintes *sprints*:

- Implementação da *flag* de ativação da funcionalidade VPNSync
- Implementação de comunicação *unicast* entre servidor principal e servidores de reserva
- Implementação de comunicação *broadcast* entre servidor principal e servidores de reserva
- Implementação de comunicação *multicast* entre servidor principal e servidores de reserva

Na implementação da transferência dos dados relevantes (estado do cliente) entre servidores foram efetuados os seguintes *sprints*:

- Criação do cliente nos servidores de reserva
- Sincronização de valores da sessão SSL/TLS
- Sincronização das chaves de sessão AES e SHA
- Sincronização de valores relativos aos pacotes trocados entre servidor principal e cliente e renegociações.

Nos testes de validação da solução foram efetuados os seguintes *sprints*:

1. Ligar um cliente, desligar o servidor principal e fazer a transição do cliente para um servidor de reserva (que é o novo servidor principal).
2. Ligar um cliente, desligar o servidor principal e fazer a transição do cliente para um servidor de reserva, e depois desligar de novo o servidor principal e fazer a transição para o servidor de reserva seguinte.
3. Ligar dois clientes, desligar o servidor principal e fazer a transição de ambos para um servidor de reserva.
4. Desligar o servidor principal e fazer a transição para um servidor de reserva, enquanto um *ping* é efetuado para uma máquina na rede interna dos servidores.
5. Desligar o servidor principal e fazer a transição para um servidor de reserva, enquanto é utilizado o Netcat para trocar mensagens entre o cliente e uma máquina na rede interna dos servidores.

A redação do relatório final começou na primeira semana de Junho e prolongou-se até ao final de Junho. Começou pela correção dos problemas levantados na avaliação intermédia e seguiu com a escrita do funcionamento da solução e testes. Terminou com a exposição das conclusões atingidas e com uma visão sobre possíveis futuros desenvolvimentos.

# 4. Análise de Requisitos

Na análise de requisitos são identificadas as condições a que deve estar sujeita a solução a implementar. Para isso, foi essencial o estudo efetuado no estado da arte, nomeadamente o estudo do OpenVPN, do CARP, bem como dos protocolos de sincronização *pfsync* e *dhcpd* (com a *flag* -y e -Y).

O objetivo principal desta dissertação é o de criar uma solução de *failover* do lado do servidor para um serviço SSL VPN. A solução atual no software escolhido, o OpenVPN, passa por definir no cliente diferentes IPs de servidores ao qual este deve ligar. Sendo que este só pode estar ligado a um de cada vez, quando um servidor deixa de responder por qualquer motivo, o cliente efetua uma nova ligação para o próximo IP da lista.

Esta solução apresenta dois problemas: o cliente ter o peso computacional de efetuar uma nova ligação cada vez que um servidor falha, e o cliente perder todo o estado das acções que estava a efetuar no momento em que a ligação falhou.

A solução para os dois problemas apresentados é desenvolver uma alternativa que permita ao cliente não ter de efetuar qualquer tipo de ação quando um servidor falha. Isto é, implementar um mecanismo do lado do servidor que permita uma troca de um servidor em falha por um servidor saudável, sem que o cliente se aperceba desta falha. É importante também que o servidor saudável que vai substituir o que falhou tenha a capacidade de continuar as ações que o cliente está a fazer.

Como consequência, é agora possível especificar os requisitos da solução.

## 4.1. Requisitos Funcionais

Os requisitos funcionais são os requisitos que especificam o comportamento do sistema, detalhando as suas funcionalidades e ações. Nos requisitos funcionais principais são definidos os requisitos mais relevantes e essenciais para que a solução seja considerada válida. Os requisitos funcionais principais são:

- O *cluster* de servidores é o único responsável pelo *failover*. O cliente não aplica mecanismo de *failover*.
- O cliente deve ligar-se ao *cluster* de servidores através de um único IP.
- O cliente não deve perder a ligação quando o servidor principal falha e um de reserva toma o seu lugar.
- O *cluster* de servidores deve possuir um funcionalidade de sincronização do estado de clientes ligados.
- A funcionalidade de sincronização do estado deve utilizar um endereço e porto exclusivamente dedicados para a comunicação entre servidores.

Nos requisitos funcionais secundários são definidos os requisitos que adicionam funcionalidade extra à solução. Estes são abordados caso os principais sejam atingidos e caso exista tempo disponível para o desenvolvimento do código necessário. Os requisitos funcionais secundários são:

- A solução deve funcionar para 2 ou mais servidores em *cluster*.
- A solução deve funcionar para 2 ou mais clientes.

Com estes requisitos funcionais é possível aplicar um mecanismo de *failover* transparente ao cliente. Isto porque o cliente apenas tem na sua configuração um único IP que vai representar o *cluster* de servidores, e os servidores possuem um mecanismo de *failover* que permite que o endereço IP aponte sempre para um servidor saudável. Além disso, os servidores têm uma funcionalidade de sincronização da informação relativa ao cliente ligado que mantém a sua informação atualizada em todos.

## 4.2. Requisitos Não-Funcionais

Os requisitos não funcionais especificam o comportamento do sistema ao nível do seu desempenho e usabilidade. Os requisitos não-funcionais relativos à solução são os seguintes:

- A solução deve ser desenvolvida utilizando a mesma linguagem de programação do OpenVPN (linguagem C)
- A funcionalidade de sincronização do estado deve ser ativada na inicialização dos servidores OpenVPN através de uma *flag* opcional.
- O processo de sincronização do estado do cliente entre dois servidores não deve demorar mais de 5 segundos
- O servidor principal deve enviar mensagens de sincronização de cada cliente cada vez que é necessário atualizar a sua informação de estado.

A partir destes requisitos não-funcionais é possível desenvolver uma solução adequada e fácil de utilizar. Para que a solução seja o mais integrada possível no *software* OpenVPN, só faria sentido desenvolvê-la na mesma linguagem de programação do mesmo. A *flag* na configuração é a forma por defeito utilizada pelo OpenVPN para ativar/desativar funcionalidades. Os 5 segundos para o processo de sincronização são definidos tendo em conta que o processo inicial de uma ligação demora cerca de 60 segundos e é pouco provável que a ligação caia logo a seguir. Portanto, 5 segundos é um valor que se define como sendo aceitável para a performance da solução. Por fim, o servidor principal deve manter os servidores de reserva atualizados para que estes o possam substituir de forma transparente.

Com todos os requisitos definidos é agora possível avançar para a especificação da solução, detalhando o seu desenvolvimento e funcionamento.

# 5. Especificação da Solução

Neste capítulo é especificado o sistema implementado durante o segundo semestre e que é o produto final deste estágio/dissertação. Inicialmente, é explicado o seu propósito e dada uma visão geral do funcionamento. Posteriormente, é apresentado o ambiente e as configurações do mesmo. Estas servem de base para na secção seguinte ser detalhado o funcionamento do código desenvolvido. Por último, são discutidos os resultados dos testes de validação.

O Anexo A contém uma análise do funcionamento do OpenVPN que serviu de base para a formulação da solução desenvolvida. Daqui em diante a solução será referida pelo nome VPNSync.

## 5.1. Propósito

O propósito deste sistema é servir áreas em que a disponibilidade seja um atributo crítico de negócio, tal como a bolsa ou a banca. No entanto, serviços como a telemedicina ou serviços de monitorização de ameaças informáticas também poderão beneficiar desta solução. Uma das vantagens desta implementação é a diminuição de custos, uma vez que com apenas duas máquinas como servidores se conseguirá aumentar a disponibilidade do serviço de forma significativa.

## 5.2. Representação de Alto Nível

O sistema desenvolvido tem por objetivo criar um mecanismo de *failover* de servidores OpenVPN com sincronização dos estados das ligações dos clientes entre servidores. Isto significa que todos os servidores do *cluster* mantêm informação sobre o estado actual das ligações.

Como consequência, no momento em que o servidor principal sofra uma quebra e um dos servidores de reserva tome o seu lugar, o estado das ligações mantém-se.

O sistema está ilustrado na figura seguinte:

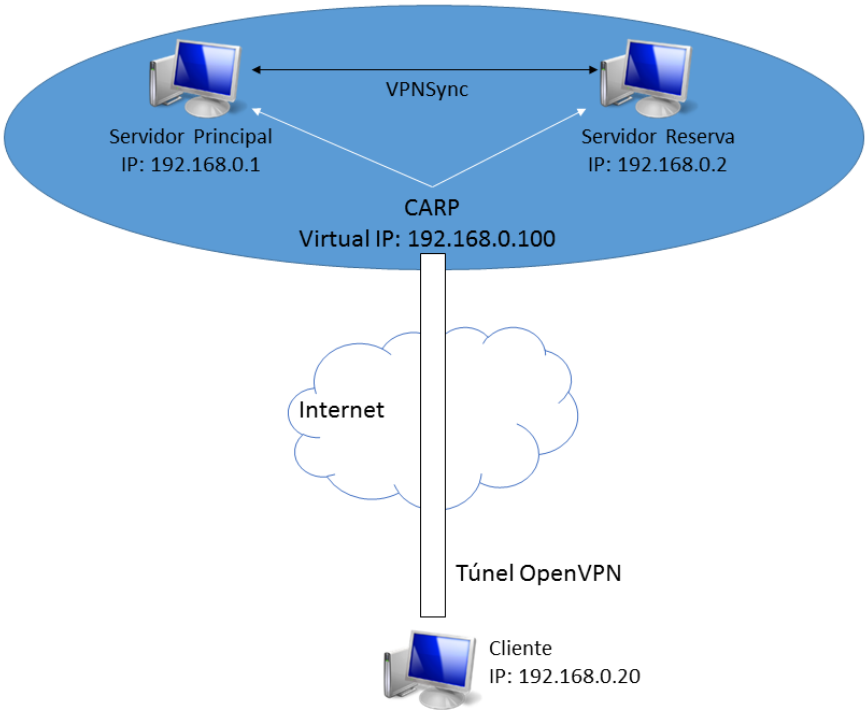



*Figura 10 - Representação do sistema de failover*

Na figura seguinte apresenta-se o diagrama de sequência que retrata o funcionamento do OpenVPN com a funcionalidade VPNSync ativa:

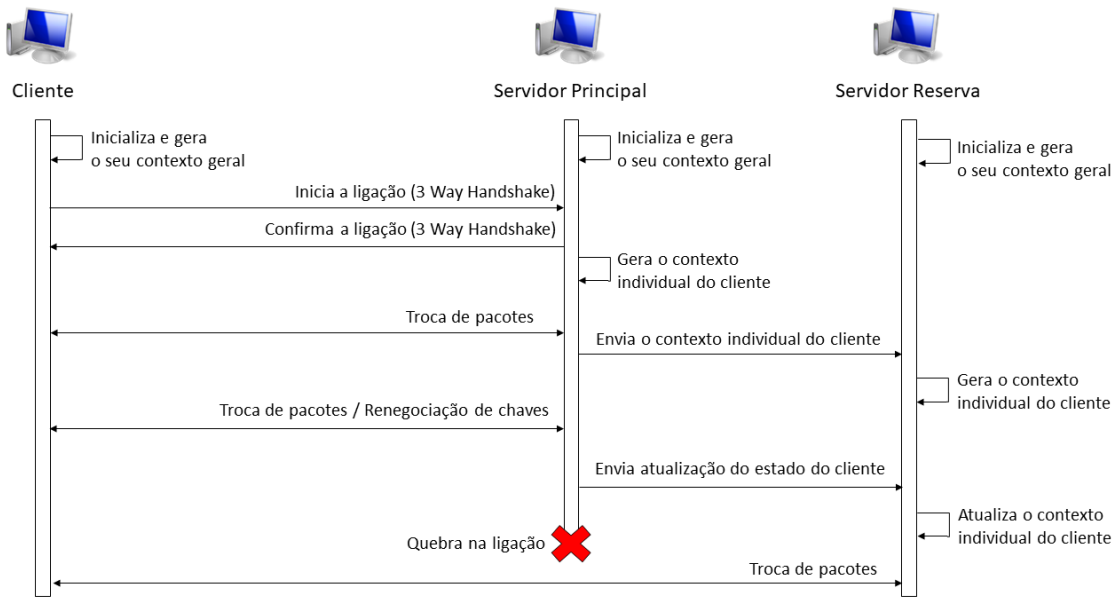



*Figura 11 - Diagrama de Sequência da funcionalidade VPNSync*

A solução funciona da seguinte forma:

1. Os servidores ligam-se e abrem uma ligação UDP para comunicação do servidor principal para o(s) servidor(es) de reserva.
2. Um cliente liga-se. O servidor que estiver a principal (*master*) no CARP estabelece a ligação e cria o contexto para esse cliente e guarda no seu contexto geral. Envia de seguida o contexto do cliente pela ligação UDP para o(s) servidor(es) de reserva. O(s) servidor(es) de reserva cria(m) um novo cliente com o contexto recebido guarda(m) no seu contexto geral.
3. Sempre que as configurações de sessão do cliente são alteradas, o servidor principal atualiza o servidor de reserva com as novas configurações.
4. Na eventualidade de o servidor principal falhar, um servidor de reserva toma o seu lugar e, como já tem o contexto das ligações em andamento, consegue continuar as sessões dos clientes sem necessidade de renegociação.

## 5.3. Ambiente e Configurações

A funcionalidade VPNSync para funcionar correctamente necessita de um ambiente com as seguintes condições:

- OpenBSD versão 6.1.
- OpenVPN versão 2.4.1.
- Servidores OpenVPN parte da mesma rede interna.
- Interface CARP configurado em todos os servidores.
- Interface TUN configurado no OpenVPN em todos os clientes e servidores.
- Protocolo UDP configurado no OpenVPN em todos os clientes e servidores.
- Cifra AES-256-CBC configurada no OpenVPN em todos os clientes e servidores.
- Autenticação SHA256 ou SHA512 configurada no OpenVPN em todos os clientes e servidores.
- *Flag* 'vpnsync' configurada no OpenVPN em todos os servidores.
- Endereço *multicast* “239.0.2.0” disponível.
- Porto 1196 aberto em todos os servidores.

É importante referir que o servidor principal OpenVPN deve ser configurado de acordo com o tutorial do Anexo B, sendo os servidores de reserva clones deste. Os clientes OpenVPN devem ser configurados de acordo com o tutorial do Anexo C.

**Versões OpenBSD e OpenVPN**

No âmbito da criação do produto final, começou por ser utilizado o sistema operativo OpenBSD versão 6.0 e a versão incluída do protocolo CARP, bem como o software OpenVPN versão 2.3.11. No entanto, após ter sido disponibilizado o OpenBSD 6.1 e a versão 2.4.1 do OpenVPN, o código já desenvolvido foi adaptado com sucesso para as novas versões.

**CARP**

O protocolo CARP possibilita a ligação do cliente a um IP único que identifica o *cluster* de servidores OpenVPN. Dentro do *cluster* este gere também qual dos servidores é o servidor principal e quais são os de reserva. Quando o servidor principal falha, este efetua a transição do tráfego para um servidor de reserva, que entretanto foi eleito como o novo servidor principal. Os servidores OpenVPN devem ter o protocolo CARP configurado de acordo com o Anexo D.

Em relação à redundância implementada pelo CARP é de referir que não é aplicado qualquer mecanismo de *load-balancing*, isto é, balanceamento de carga entre os servidores. Apenas um servidor está "ativo" a cada momento.

**TUN e UDP**

Os túneis OpenVPN utilizam como protocolo de transporte o UDP e a interface TUN. A razão destas escolhas prende-se com o facto de se garantir rapidez e maior facilidade de implementação da solução preservando a segurança das ligações. O UDP é também o protocolo predefinido para uso no OpenVPN [9].

**AES-256-CBC e SHA256**

Como protocolo de autenticação e encriptação utiliza-se o TLS 1.2, que é atualmente o mais avançado (a versão 1.3 ainda não se encontra terminada). A cifra utilizada é a AES-256-CBC e o método de autenticação é o SHA256, combinação que oferece uma proteção elevada [15].

A escolha do AES-256-CBC limita a possibilidade de serem utilizadas outras cifras, como o Blowfish ou o AES-256-GCM, devido ao funcionamento específico de cada uma. Já em relação ao SHA256, é possível também utilizar o SHA512 devido ao seu funcionamento ser semelhante.

**Flag 'vpnsync'**

Para ativar a funcionalidade VPNSync é necessário adicionar ao ficheiro de configurações do OpenVPN uma linha com o texto "vpnsync". Isto irá ativar por defeito o endereço *multicast* "239.0.2.0" e o porto 1196. No Anexo E está a explicação de como foi implementada a *flag*.

**Endereço "239.0.2.0" e Porto 1196**

O endereço "239.0.2.0" e o porto 1196 são as configurações utilizadas por defeito para as comunicações *multicast*. Estas comunicações funcionam apenas no sentido servidor principal -> servidores de reserva. O seu propósito é o de enviar o estado da ligação cliente-servidor principal para todos os servidores de reserva mas não para todas as máquinas em rede, algo que o *multicast* possibilita.

O protocolo de transporte utilizado é o UDP. Como os servidores estão na mesma rede interna é assumido que existirá uma ligação rápida e sem quebras entre estes. Consequentemente, a fiabilidade da entrega dos pacotes trocados não será um problema. Outro motivo para o uso do UDP é a própria arquitetura do VPNSync, uma vez que esta não exige uma ligação individual e permanente entre servidor principal e cada servidor de reserva.

A razão da utilização do endereço "239.0.2.0" por defeito prende-se apenas com o facto de este ser um endereço privado de *multicast* que à partida estará disponível na maior parte das máquinas. O motivo da utilização do porto 1196 prende-se com o facto de os portos utilizados pelo OpenVPN por defeito serem: o 1194 para ligações VPN e o 1195 para a funcionalidade de controlo remoto de um servidor. Sendo assim, o candidato lógico a utilizar é o porto seguinte, ou seja, o 1196.

**Outras considerações**

É utilizada a funcionalidade *keepalive* que permite uma monitorização entre servidor e cliente com *pings* para verificar se ambos estão ativos. As configurações definem uma troca de *pings* a cada 30 segundos e caso não haja resposta por 90 segundos a ligação é terminada e o cliente tenta renegociar uma nova ligação.

A solução desenvolvida visa sincronizar apenas o estado gerado pelo OpenVPN no momento da ligação do cliente, sendo que alterações relativas à camada aplicacional não fazem parte do âmbito desta dissertação. Por consequência, o tráfego com destino para o *endpoint* (servidor principal) é assumido como perdido quando o servidor falha.

Não se considera o uso de *http-proxy*, *socks-proxy*, *float*, *ipv6*, *routes*, *custom environment variables*, *multihome* ou outros *plugins* na configuração do OpenVPN.

## 5.4. Funcionamento

A funcionalidade VPNSync foi implementada, ao nível do código, seguindo o exemplo de como está implementada a funcionalidade *management* [37], que possibilita o controlo remoto de um servidor OpenVPN. Isto acontece porque esta funcionalidade é a que possui maiores semelhanças com aquilo que é pretendido para o VPNSync, nomeadamente:

- Funcionar somente em modo servidor e não em modo cliente,
- Estar desativada por defeito,
- A sua ativação acontecer através da inserção de uma *flag* no ficheiro de configurações do servidor.

Como resultado, torna-se assim mais fácil e intuitivo a continuação do desenvolvimento do VPNSync por outros programadores, uma vez que segue as mesmas práticas utilizadas pela equipa de desenvolvimento do OpenVPN.

Como instrumento adicional, foi produzida ainda documentação do código desenvolvido utilizando a popular ferramenta Doxygen [38]. A partir desta ferramenta foi possível gerar um conjunto de páginas em formato *html* com descrições e imagens sobre o funcionamento da funcionalidade VPNSync. Para cada função desenvolvida existe:

- Uma descrição breve
- Uma descrição detalhada
- Um diagrama que ilustra por que outras funções esta é chamada
- Um diagrama que ilustra que outras funções é que esta chama.

A documentação foi redigida em língua inglesa para que possa ser interpretada a nível internacional, tomando por base a própria documentação Doxygen do OpenVPN [39]. O resultado final pode ser consultado no Anexo H.

### 5.4.1. Arquitectura e Funções

Ao formular a solução foi necessário tomar decisões acerca do design e arquitectura a utilizar. Entre as decisões tomadas destacam-se duas, relacionadas com as seguintes questões:

1. Como implementar a comunicação entre os servidores?
2. Como desenvolver as ações do VPNSync de modo a que atuem nos servidores sem interferir com o seu funcionamento normal?

Para a primeira questão foi utilizado um processo incremental. Primeiramente foi desenvolvida uma solução de comunicação "um para um", utilizando *sockets unicast*. Esta solução estava limitada à utilização de apenas dois servidores (um principal e outro de reserva). De seguida foi desenvolvida uma solução "um para todos", utilizando *sockets broadcast*. No entanto esta solução tinha também uma limitação inerente: o facto de todas as máquinas da rede receberem a informação, mesmo não sendo servidores OpenVPN.

Por fim, foi desenvolvida uma solução "todos para todos", utilizando *sockets multicast*, que garante a comunicação está restrita apenas aos servidores OpenVPN e que pode ser efetuada de "um para todos". Esta solução também tem a vantagem de possibilitar uma configuração mais simples, uma vez que cada servidor apenas necessita de ter a informação de qual é o endereço *multicast* e o porto, seja ele principal ou reserva. Além disso, não é necessário alterar qualquer tipo de configurações durante a troca de um servidor de reserva para principal.

Para a segunda questão foram definidas duas possibilidades: utilizar *threads* ou utilizar processos. As *threads* são tarefas que podem ser criadas por um processo e ser executadas de forma concorrente. Estas têm a vantagem de partilhar a memória do processo, uma vez que são executadas como parte deste. Já um processo pode ser criado por outro processo, denominando-se por processo filho. No entanto, um processo tem sempre memória própria. Como consequência, o que acontece quando um processo cria um processo filho é que o filho recebe uma cópia da memória atual do processo pai. Para trocar informação entre processo filho e processo pai é necessário usar um canal de comunicação entre processos denominado *pipe*.

Depois da análise duas possibilidades utilizar threads foi considerada a melhor opção, no entanto o OpenVPN possui uma arquitectura monolítica [40]. Isto significa que o OpenVPN não suporta *threads*. Sendo assim a solução recaiu sobre utilizar processos.

Para implementar a solução foram criados os ficheiros *vpnsync.c* e *vpnsync.h*. Neles foram desenvolvidas todas as funções relativas à funcionalidade VPNSync. No entanto, para poder utilizar a funcionalidade foram necessárias realizar 4 tarefas:

- Adicionar os ficheiros *vpnsync.c* e *vpnsync.h* à compilação do OpenVPN
- Adicionar os ficheiros *binn.h* e *binn.c* à compilação do OpenVPN
- Alterar o código do OpenVPN para implementar a *flag* 'vpnsync'
- Alterar o código do OpenVPN para incluir as chamadas às funções do VPNSync.

O Anexo F detalha como os ficheiros *vpnsync.c*, *vpnsync.h*, *binn.h* e *binn.c* foram adicionados à compilação do OpenVPN.

As funções contidas nos ficheiros *vpnsync(.c* e *.h)* estão organizadas da seguinte forma:

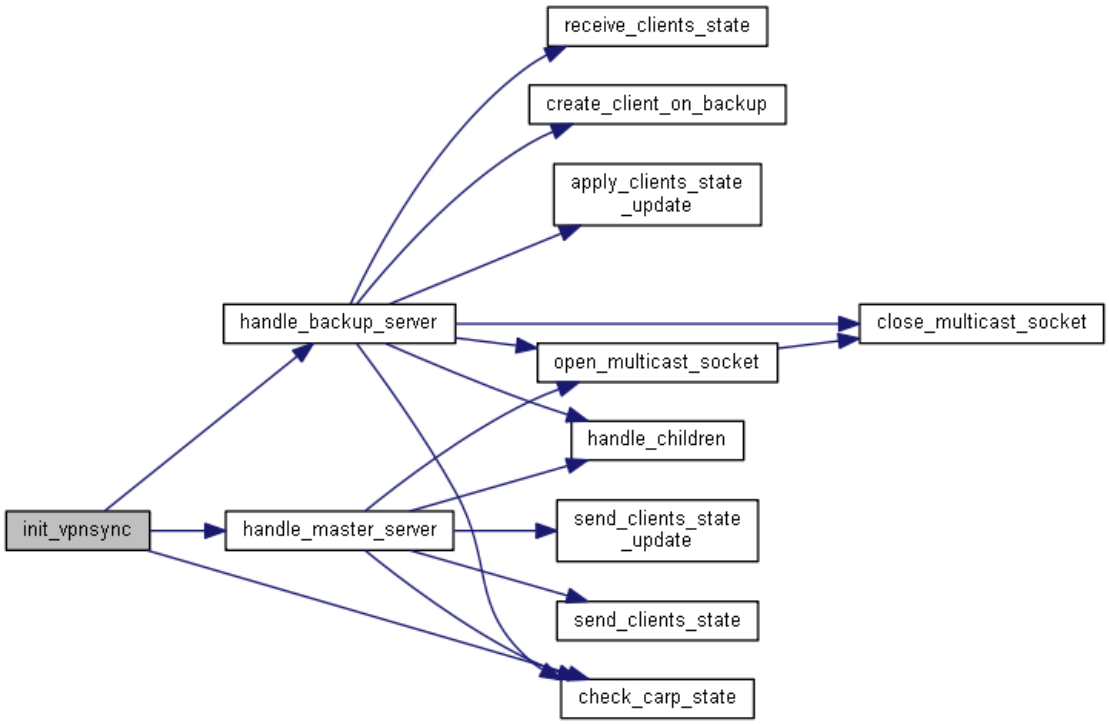



*Figura 12 - Diagrama de chamada de funções do VPNSync*

A figura anterior foi obtida graças à ferramenta Doxygen e representa a visão mais global do VPNSync, contendo todas as funções e como elas se inter-relacionam.

A função *init_vpnsync()* é o ponto de partida para as operações do VPNSync. Esta é chamada apenas se a *flag* 'vpnsync' estiver definida no ficheiro de configurações do servidor. No momento de inicialização do servidor OpenVPN (UDP), mais especificamente na função *tunnel_server_udp_single_threaded()*, é efetuada a verificação se a *flag* está definida e a consequente chamada da função *init_vpnsync()* caso o resultado seja positivo.

A partir da *init_vpnsync()* é chamada em primeiro lugar a função *check_carp_state()* que verifica se o interface CARP está ativo na máquina. Caso esteja o seu estado é verificado para averiguar se o servidor é o principal (*master*) ou um de reserva (*backup*). Dependendo do resultado obtido é chamada a função *handle_master_server()* ou a *handle_backup_server()*.

A função *handle_master_server()* é responsável por todas as operações que ocorrem num servidor que seja principal. Já a função *handle_backup_server()* é responsável por todas as operações que ocorrem num servidor que seja de reserva.

A figura seguinte ilustra o funcionamento da função *handle_master_server()*:

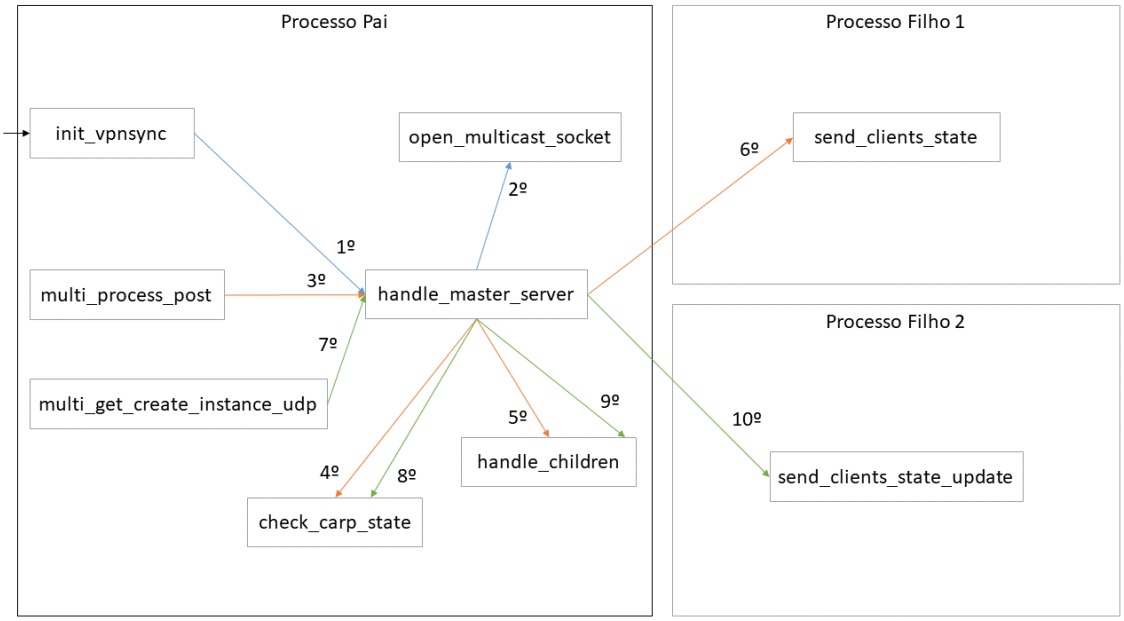



*Figura 13 - Funcionamento da função handle_master_server*

Na função *handle_master_server()* são chamadas 5 funções: *open_multicast_socket()*, *check_carp_state()*, *send_clients_state()*, *send_clients_state_update()* e *handle_children()*.

No momento de inicialização do servidor principal, na função *tunnel_server_udp_single_threaded()* presente no ficheiro *mudp.c*, a função *init_vpnysnc()* é chamada. Neste momento é aberto o canal de comunicação entre servidor principal e servidores de reserva pela função *open_multicast_socket()*. Nesta função é aberto um *socket* UDP de comunicação *multicast* no endereço "239.0.2.0" e porto 1196.

Quando um cliente se liga ao servidor principal o servidor chama a função *handle_master_server()* após a conclusão do aperto de mão. Isto acontece na função *multi_process_post()*, presente no ficheiro *multi.c*, após a chamada da função *multi_connection_established()* que finaliza um aperto de mão bem sucedido.

O objetivo é o de chegar à função *send_clients_state()*, que é iniciada num processo filho gerado apenas para esse propósito. O *send_clients_state()* envia em *multicast* a informação necessária para todos os servidores de reserva de modo a que consigam recriar a instância do cliente, com todas as informações necessárias, para garantir uma transição transparente caso o servidor principal falhe. Esta informação é enviada em dois pacotes dado que o seu tamanho excede o máximo permitido por um pacote.

Para evitar que um servidor de reserva retrasmita a informação recebida quando cria o contexto do cliente, antes de ser chamada a função *send_clients_state()* é feita uma verificação do estado do CARP através do *check_carp_state()*. Quando a função *send_clients_state()* termina a função *handle_children()* encarrega-se de terminar o processo filho.

É importante referir que o envio do estado do cliente só acontece 60 segundos após a ligação ter sido iniciada. Isto acontece pela necessidade de dar tempo para que o *handshake* termine com sucesso.

Quando é efetuada uma troca de pacotes entre servidor e cliente, é chamada a função *multi_get_create_instance_udp()*, que por sua vez chama a *handle_master_server()* e por fim a *send_clients_state_update()* que envia informações sobre o número de pacotes (ID) enviados e recebidos e a hora para os servidores de reserva. Isto é necessário para evitar que, no evento de uma mudança de servidores, o cliente comece a receber pacotes com IDs repetidos que podem levar à terminação da ligação.

Por fim, salientar que o socket *multicast* é fechado, através da função *close_multicast_socket()* chamada pela função *tunnel_server_udp_single_threaded()*, quando o programa recebe a indicação para terminar.

A figura seguinte ilustra o funcionamento da função *handle_backup_server()*:

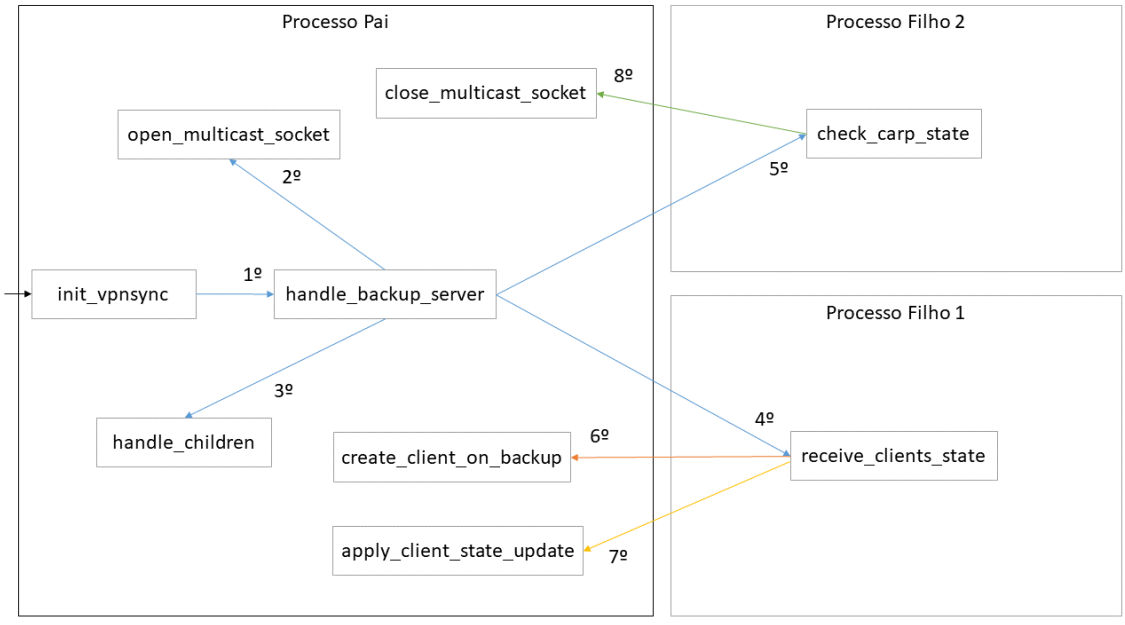



*Figura 14 - Funcionamento da função handle_backup_server*

Na função *handle_backup_server()* são chamadas 7 funções: *open_multicast_socket()*, *handle_children(), receive_clients_state()*, *check_carp_state()*, *create_client_on_backup()*, *apply_client_state_update()* e *close_multicast_socket()*.

Na função *open_multicast_socket()* é aberto um *socket* UDP para receber comunicações *multicast* do endereço "239.0.2.0" e porto 1196. Após a abertura do *socket* dois processos filho são criados: um dedicado a monitorizar o estado do CARP através da função *check_carp_state()* e outro dedicado a escutar por comunicações vindas do servidor principal através da função *receive_clients_state()*.

Quando é recebido o estado de um cliente a função *receive_clients_state()* envia a informação para o processo pai e neste a função *create_client_on_backup()* é chamada. Porém, visto que o tamanho da informação é superior ao que é possível transmitir num único pacote, a restante informação é enviada num segundo pacote. Na função *create_client_on_backup()* é recebido o segundo pacote com a restante informação e criado o cliente "fantasma" com todas as informações recebidas do servidor principal. Como consequência da necessidade de enviar a informação em dois pacotes, é necessário garantir que o servidor de reserva não está à escuta de novos clientes ao mesmo tempo que está à escuta do segundo pacote. Foi tomada então a decisão de interditar a recepção de novos clientes nos 10 segundos seguintes à recepção de um novo cliente (diga-se o primeiro pacote) de modo a garantir a correta criação do cliente no servidor de reserva.

Já no evento da recepção de uma atualização de estado (que só utiliza um pacote), a função *receive_clients_state()* envia a informação para o processo pai e neste é chamada a função *apply_client_state_update()* que atualiza o estado do cliente "fantasma". Aqui não existe qualquer restrição temporal.

Caso o estado do CARP passe para principal (*master*) a função *check_carp_state()* retorna e o processo filho é terminado pela função *handle_children()*. Como consequência o outro processo filho é também terminado e o *socket multicast* fechado. A função *handle_backup_server()* retorna com a informação de que deve ser chamada a função *handle_master_server()* uma vez que o servidor passou a principal.

### 5.4.2. Informação Sincronizada entre Servidores

Para que seja possível transitar um cliente entre dois servidores de forma transparente é necessário que um conjunto vasto de informações esteja sincronizado entre os servidores. Aqui convém realçar que os servidores de reserva foram criados a partir de clones do servidor principal, após a configuração do OpenVPN neste.

Este passo garante também que as seguintes informações são idênticas em todos os servidores:

- Certificado da Autoridade de Certificação (local neste caso)
- Certificado do servidor OpenVPN
- Chave Privada do servidor OpenVPN
- Parâmetros Diffie-Hellman
- Chave tls-auth (método adicional de autenticação dos pacotes)

As informações restantes necessárias são relativas à ligação propriamente dita entre o cliente e o servidor. Deste modo, foi tomada a opção de sincronizar toda a informação do cliente apenas após o aperto-de-mão entre cliente e servidor. Isto deve-se ao material aleatório utilizado durante o aperto-de-mão. Este material tem como objetivo gerar as chaves simétricas de encriptação e desencriptação que são utilizadas para as comunicações. Sendo assim, a decisão passou por sincronizar as chaves resultantes do aperto-de-mão visto que o material aleatório é descartado após o mesmo.

A função *send_clients_state()* é responsável pelo envio da informação relativa ao cliente para os servidores de reserva. Para que os servidores de reserva sejam capazes de construir a instância do cliente com as mesmas características são necessárias as seguintes informações do cliente:

- Data de criação
- Endereço IP
- Porto
- Common name (nome associado ao certificado SSL)
- IP virtual (do túnel OpenVPN)
- Número de sessões negociadas
- ID da sessão atual
- ID remoto da sessão atual
- Chave AES de encriptação
- Chave AES de desencriptação
- Chave SHA de encriptação
- Chave SHA de desencriptação

Estas informações são extraídas da estrutura *multi_instance* que guarda informação sobre o estado do cliente, tal como analisado no Anexo A. O servidor recebe estas informações através da função *receive_clients_state()* e gera o cliente na função *create_client_on_backup()*. Após o cliente estar criado, é necessário manter as informações da ligação atualizadas para que seja possível uma mudança transparente de servidores a qualquer momento.

Para manter as informações do cliente atualizadas é necessário ir sincronizando os seguintes campos nos servidores:

- Número de pacotes recebidos
- Data do último pacote recebido
- Número de pacotes enviados
- Data do último pacote enviado

São estes os valores que são enviados pela função *send_clients_state_update()* no servidor principal. Sempre que o servidor recebe um novo pacote do cliente este chama esta função. O servidor de reserva atualiza no cliente "fantasma" as informações recebidas através função *apply_client_state_update()*.

No caso de uma atualização das chaves de sessão, que por defeito acontece uma vez por hora, a função *send_clients_state()* é chamada e é enviada a informação como se de um novo cliente se tratasse. No entanto, o servidor de reserva reconhece cliente pelo IP e apenas atualiza as informações deste.

Para auxiliar na transmissão da informação entre os servidores foi necessário encontrar uma forma de serializá-la. Foi selecionada a biblioteca Binn para serializar a informação.

O Binn define-se como sendo um formato de serialização de informação binária compacto, rápido e fácil de utilizar [41]. Foram estas as características que pesaram no momento da sua escolha. Para integrar a biblioteca do formato Binn em qualquer projeto apenas é necessário adicionar dois ficheiros à compilação do programa. O Anexo F detalha como a biblioteca Binn foi adicionada à compilação do OpenVPN.

## 5.5. Testes de Validação

Para validar a solução desenvolvida foram compostos testes de modo a provar se os requisitos definidos no Capítulo 4 estão a ser atingidos. Os testes de validação foram efetuados no seguinte ambiente de testes em VirtualBox:

- 2 clientes
- 3 servidores
- 1 máquina na rede interna dos servidores OpenVPN

Para cada cliente o cenário de testes é apresentado na figura seguinte:

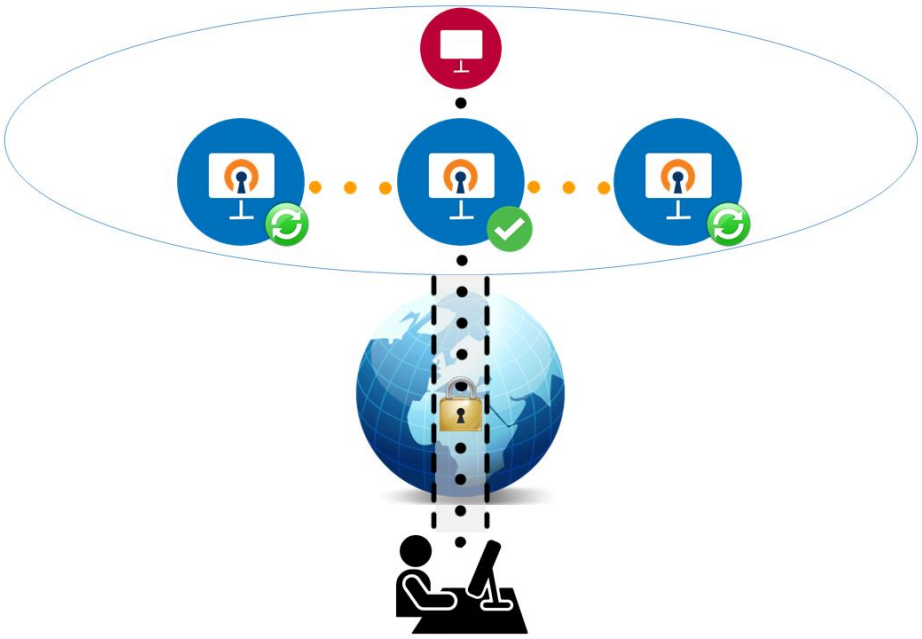


*Figura 15 - Cenário de testes para cada cliente OpenVPN*

Tal como indicado, existem 3 servidores OpenVPN e uma quarta máquina que está presente na rede interna dos servidores. Existem ainda dois clientes e cada cliente liga-se ao mesmo IP, definido na interface CARP em todos os servidores. As configurações do OpenVPN em servidores e clientes são as indicadas nos Anexos B, C, e D que recordo aqui:

- IP do *cluster* de servidores OpenVPN: 192.168.0.100
- Porto do *cluster* de servidores OpenVPN: 1194
- IP do VPNSync (endereço *multicast*): 239.0.2.0
- Porto do VPNSync: 1196
- IP do Cliente 1: 192.168.0.20
- IP do Cliente 2: 192.168.0.21
- Rede Interna dos servidores OpenVPN: 10.66.0.0
- IP da Máquina na rede interna dos servidores OpenVPN: 10.66.0.3

Para conseguir automatizar os testes de forma mais articulada foram desenvolvidos os seguintes *scripts* de configuração para utilizar nos servidores:

- *script_setsysflags*
- *script_build*
- *script_carp*
- *script_openvpn*
- *script_fail*

Inicialmente, o *script_setsysflags* define as variavéis de sistema necessárias para que o VPNSync possa funcionar, como o encaminhamento de pacotes IP e *multicast*. Este *script* só necessita de ser executado uma vez por máquina.

De seguida, o *script_build* dá início à compilação do OpenVPN e deixa-o pronto a executar. O *script_carp* inicia o interface CARP com as seguintes configurações: o servidor fica como principal (*master*) se não existir já um principal, caso contrário fica como reserva (*backup*). O *script_openvpn* inicia a execução do OpenVPN carregando consigo o ficheiro de configurações *server_tun0.conf*, formulado no Anexo B.

Por fim, o *script_fail* termina a execução do OpenVPN e elimina o interface CARP, simulando assim uma falha no servidor. Os *scripts* completos podem ser visualizados no Anexo G.

Os seguintes testes foram efetuados:

1. Ligar um cliente, desligar o servidor principal e fazer a transição do cliente para um servidor de reserva (que é o novo servidor principal).
2. Ligar um cliente, desligar o servidor principal e fazer a transição do cliente para um servidor de reserva, e depois desligar de novo o servidor principal e fazer a transição para o servidor de reserva seguinte.
3. Ligar dois clientes, desligar o servidor principal e fazer a transição de ambos para um servidor de reserva.
4. Desligar o servidor principal e fazer a transição para um servidor de reserva, enquanto um *ping* é efetuado para uma máquina na rede interna dos servidores.
5. Desligar o servidor principal e fazer a transição para um servidor de reserva, enquanto é utilizado o Netcat para trocar mensagens entre o cliente e uma máquina na rede interna dos servidores.

**Teste 1**

No primeiro teste ficou de imediato comprovado que ambos os servidores de reserva conseguem de forma bem sucedida receber as informações do servidor principal sobre o cliente ligado (que tal como referido anteriormente só acontece 60 segundos após a ligação ter sido iniciada), posteriormente criando o cliente "fantasma" com essas informações. A transição do cliente entre o servidor principal e o primeiro servidor de reserva aconteceu de forma transparente e imediata (inferior ao requisito de 5 segundos), utilizando o script *script_fail* para fingir uma falha no servidor principal. Como o CARP reconhece instantaneamente uma falha no servidor principal, a mudança de servidor principal e posterior redirecção de novos pacotes para o novo servidor principal.

**Teste 2**

No segundo teste a primeira transição aconteceu de forma transparente, tal como no primeiro teste. A segunda transição também aconteceu de forma transparente, no entanto é necessário referir que em algumas situações pode acontecer que um pacote possa ser enviado do servidor para o cliente com um ID desatualizado e gerar um aviso no cliente. Isto acontece caso a transição ocorra antes de o servidor de reserva conseguir receber e aplicar a última atualização, o que é todavia pouco provável.

As figuras seguintes ilustram uma mudança de servidor principal:

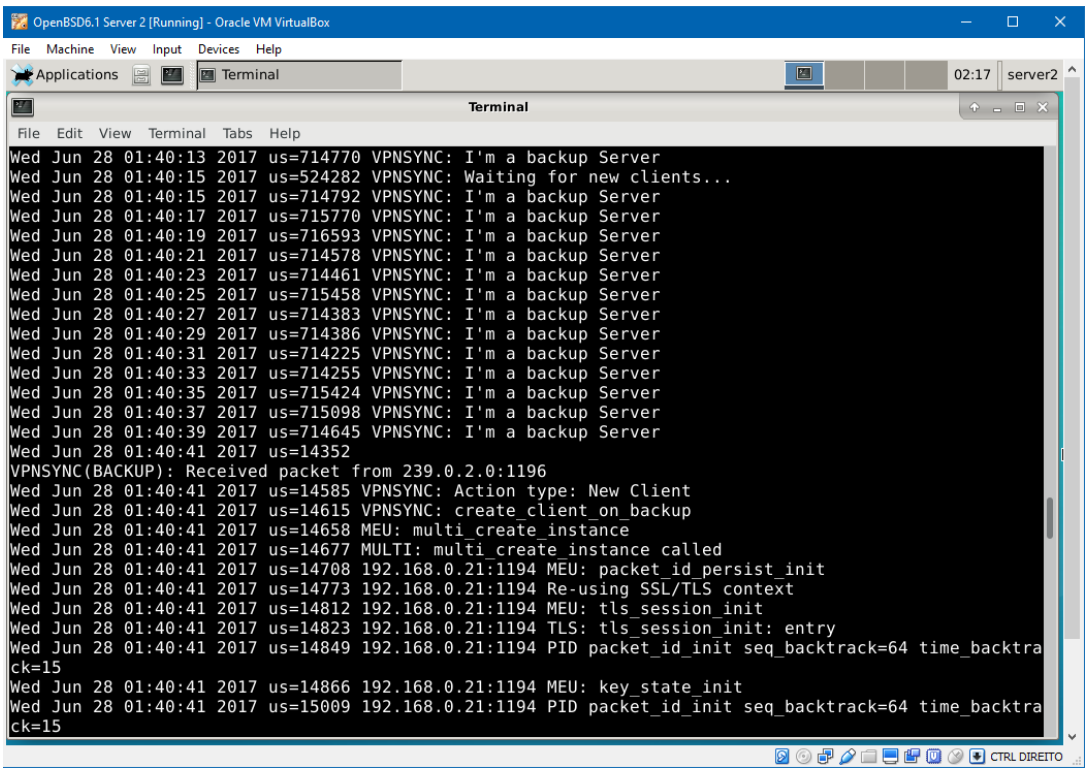



*Figura 16 - Servidor de Reserva a receber um novo cliente*

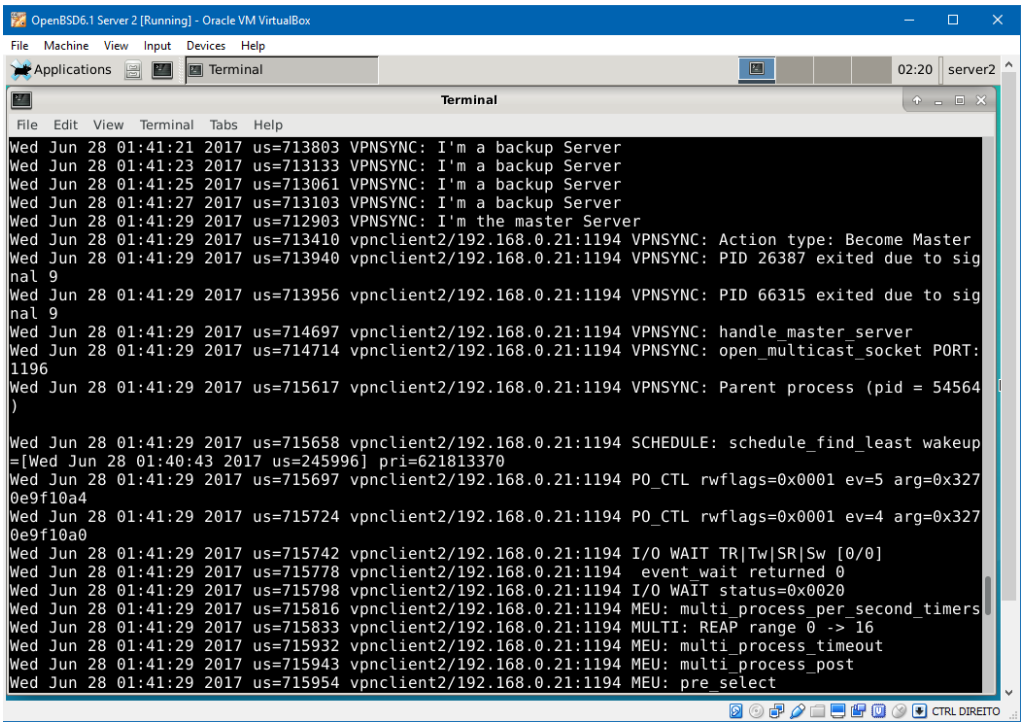



*Figura 17 - Servidor de Reserva que passa a Principal*

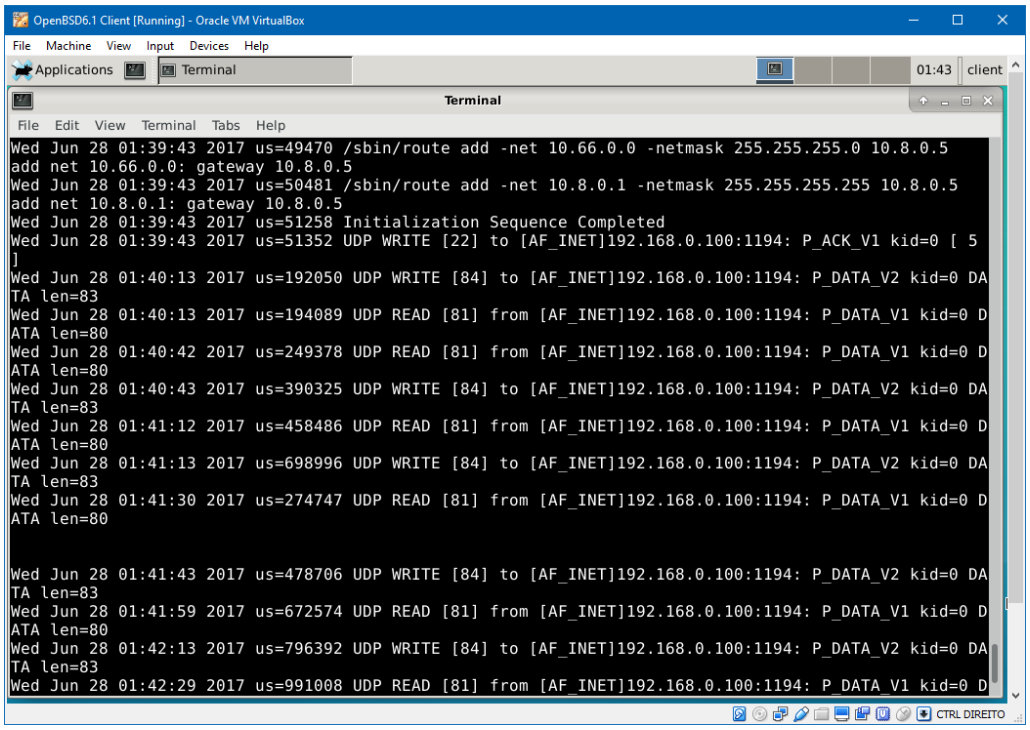



*Figura 18 - Cliente durante o processo de mudança de servidores*

É possível verificar que o cliente não se apercebe da mudança de servidores que aconteceu na hora 01:41. Por esta razão, conclui-se que esta aconteceu de forma transparente e sem quebrar a sessão estabelecida.

**Teste 3**

No terceiro teste ligaram-se dois clientes ao *cluster* OpenVPN. Após a conclusão dos apertos-de-mão de ambos e troca de alguns *pings* específicos do OpenVPN (configurados pela funcionalidade *keepalive* anteriormente mencionada), o servidor principal foi desligado. A transição de um servidor de reserva para principal foi imediata e ambos os clientes continuaram a sessão de forma transparente.

**Teste 4**

No quarto teste foi efetuado um *ping* do cliente para a máquina na rede interna. Foi depois simulada uma falha no servidor principal e respectiva transição de um servidor de reserva para servidor principal a meio do *ping*. Os resultados demonstraram que existe uma perda de *pings* pouco significativa durante a transição e que resulta da referida sincronização do ID e hora dos pacotes entre o cliente e o novo servidor principal.

**Teste 5**

Na figura acima pode verificar-se o que acontece depois do cliente utilizar o Netcat para enviar uma mensagem, para a máquina na rede interna dos servidores OpenVPN com endereço “10.66.0.3”, e entretanto o servidor principal falhar.

Quando o servidor de reserva, que entretanto se tornou o principal, envia um pacote para o cliente este reporta a recepção de um pacote com um ID diferente do que estava à espera. Como resultado, este emite um aviso de que este pacote pode ser repetido, porque o pacote com o ID indicado já foi recebido no passado.

O que acontece de seguida é o envio de um pacote do cliente para o servidor, que o servidor regista como tendo ID superior ao número de pacotes recebidos que ele possui registado em relação aquele cliente. No entanto, o servidor tenta sincronizar os IDs de modo a prosseguir com a ligação desde que os valores não sejam muito dispares. Caso a sincronização seja bem sucedida, o cliente passa a receber pacotes com os IDs que este espera. Caso não seja bem sucedida e o número de pacotes repetidos seja elevado (o que pode ser definido através da funcionalidade *replay-window* ), o cliente termina a sessão em curso e tenta iniciar uma nova sessão.

A figura seguinte mostra o histórico do cliente quando é efetuado o teste 4:

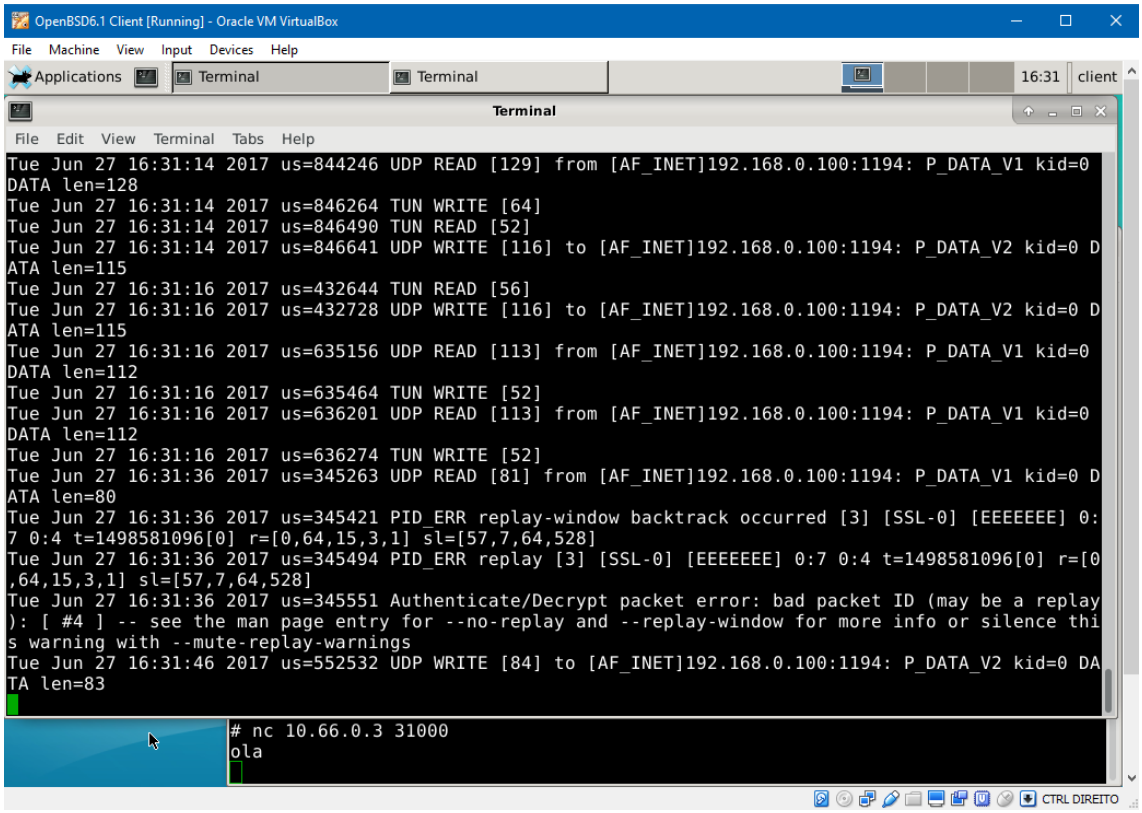



*Figura 19 - Cliente quando recebe um pacote com um ID inesperado*

# 6. Conclusão

Neste capítulo apresentam-se as conclusões retiradas sobre o projeto desenvolvido, começando por abordar os resultados alcançados e as ilações que se podem retirar destes. Por fim, detalham-se possíveis futuros desenvolvimentos para a solução desenvolvida.

## 6.1. Resultados Alcançados

Em conclusão, é possível afirmar que os objetivos da dissertação foram em grande parte atingidos. Os resultados alcançados incluem:

1. Levantamento do estado da arte
2. Análise do funcionamento interno do OpenVPN
3. Planeamento da metologia e das tarefas a realizar
4. Redação dos requisitos funcionais e não-funcionais
5. Formulação e estruturação de um cenário de testes e proposta de solução
6. Desenvolvimento da solução
7. Testes de validação da solução

**Levantamento do Estado da Arte**

Numa primeira fase foi efetuado um levantamento do estado da arte, que consistiu numa pesquisa intensiva dos mais variados temas relacionados com o projeto. Entre estes incluem-se: atributos e algoritmos de segurança; protocolos SSL/TLS, IPSec, PPTP, VPNs (incluindo OpenVPN); protocolos de redundância e ferramentas de *clustering*. Adicionalmente, foram ainda revistas soluções comerciais de alta disponibilidade e as suas características ao nível do *failover*, bem como protocolos de sincronização de estados para outras aplicações (que não VPNs). O resultado deste levantamento encontra-se no Capítulo 2.

**Análise do funcionamento interno do OpenVPN**

Depois de tomada a decisão de utilizar o OpenVPN como software de cliente e servidor de VPN, iniciou-se uma investigação para compreender o seu funcionamento. Através da análise do seu código fonte e da leitura de vários artigos foi possível identificar as suas estruturas e operações principais importantes para o tema desta dissertação. Como resultado desta análise foi redigido o documento Anexo A incluído neste relatório.

**Planeamento da metologia e das tarefas a realizar**

Em conjunto com os orientadores da empresa e com o Professor Boavida, foi definida uma metodologia a utilizar, o SCRUM. Como parte da metologia foram planeadas as tarefas a realizar e a sua calendarização. Adicionalmente, foram realizadas reuniões de acompanhamento do projeto.

**Redação dos requisitos funcionais e não-funcionais**

Após analisar o funcionamento do OpenVPN, e com base no estado da arte, foram definidos os requisitos funcionais e não-funcionais para a solução a desenvolver.

**Formulação e estruturação de um cenário de testes e proposta de solução**

Com a ajuda da ferramenta de criação de máquinas virtuais, a VirtualBox VM, foi possível implementar um cenário para o desenvolvimento e teste da solução.

Com esta implementação foi possível não só comprovar a vantagem de utilizar o CARP para aplicar *failover* do lado do servidor, como também confirmar que o problema da perda do estado se mantém mesmo quando este mecanismo é aplicado.

Por fim, e com base em todas as etapas anteriores, foi formulada e estruturada uma proposta de solução para manter o estado da ligação cliente-servidor quando o servidor principal falha e é substituído por um de reserva.

**Desenvolvimento da solução**

A solução desenvolvida foi apresentada no Capítulo 5. Inicialmente, foi definido o propósito da solução e representada uma visão de alto nível da mesma de forma a facilitar a sua compreensão. De seguida, foram apresentadas as condições do cenário de desenvolvimento e as configurações do cliente e servidores.

Passando à explicação do funcionamento da solução, foi enunciada a arquitectura e explicadas as decisões de design tomadas no desenvolvimento da mesma. O mecanismo de sincronização das informações do estado do cliente entre o servidor principal e os servidores de reserva foi explicado em detalhe. Como suporte à explicação da implementação da solução foram redigidos os Anexos E, F, G e H.

**Testes de validação da solução**

Por fim, para validar a solução desenvolvida foram efetuados testes de modo a acessar se os requisitos definidos estavam a ser atingidos pela solução desenvolvida. Os testes revelaram que a transparência é atingida no caso de não existir nenhuma ligação em andamento entre o cliente e uma terceira máquina. No caso de existir, alguns pacotes

podem conter informações desatualizadas e o cliente regista o facto de estar a receber pacotes incorretos. No entanto, a renegociação apenas existe se o número de pacotes incorretos e o tempo passado ultrapassarem um certo limite definido pelas funcionalidades *keepalive* e *replay-window*.

Conclui-se assim que os objetivos foram atingidos, embora a solução possua uma margem de evolução que não foi possível concretizar por ultrapassar o tempo definido para esta dissertação. Desde o primeiro momento foi assumido que a solução teria desenvolvimento posterior a esta dissertação, razão pela qual existiu um foco em criar documentação detalhada da mesma. Adicionalmente, houve uma preocupação em implementar a solução de modo a que esta fosse o máximo compatível com possíveis atualizações do OpenVPN. De seguida apresentam-se propostas para a continuação do desenvolvimento da solução.

## 6.2. Futuros Desenvolvimentos

Dada a natureza exploratória desta solução, é imperativo considerar futuros desenvolvimentos da mesma. Sendo assim detalham-se os possíveis trabalhos futuros:

- Implementação de *dump* do estado de todos os clientes
- Implementação de encriptação na informação transmitida entre os servidores
- Implementação de maior controlo das configurações do VPNSync através da *flag*
- Implementação de *threads* em vez de processos caso o OpenVPN mude a sua arquitectura
- Possíveis alterações futuras do OpenVPN que obriguem a alterações da solução desenvolvida.

**Implementação de *dump* do estado de todos os clientes**

No evento de um servidor principal falhar, este perde todas as informações sobre os clientes ligados. No entanto, caso volte a funcionar este passa a servidor de reserva. O que acontece atualmente é que este apenas recebe informação sobre os clientes que iniciarem ligação depois de este ter voltado a iniciar.

A solução para atualizar o servidor com a informação sobre os clientes já ligados seria este fazer um pedido ao servidor principal para que este envie a totalidade da informação sobre todos os clientes ligados. Este *dump* ou descarga, possibilitaria a sincronização deste servidor com todos os outros.

Começou-se o desenvolvimento desta funcionalidade, porém este não ficou terminado devido a complicações na arquitetura a utilizar. A necessidade de utilizar processos filho e comunicação entre processos faz com que a implementação desta funcionalidade tenha impacto no funcionamento normal do servidor principal, o que não é aceitável.

**Implementação de encriptação na informação transmitida entre os servidores**

A informação que é transmitida entre servidores neste momento não contém qualquer tipo de encriptação. Por consequência, um invasor que conseguisse escutar a rede interna dos servidores OpenVPN poderia interceptar a informação. Considerando que a informação partilhada é extremamente sensível, se esta caísse nas mãos erradas toda a segurança da rede ficaria em risco. Para garantir a confidencialidade da informação a proposta seria implementar uma chave simétrica pré-partilhada conhecida apenas pelo administrador dos servidores OpenVPN.

**Implementação de maior controlo das configurações do VPNSync através da *flag***

Atualmente a solução assume de forma pré-determinada um endereço *multicast* e um porto para comunicação entre servidores. Estes valores podem ser passíveis de ser alterados através da *flag* 'vpnsync' caso esta seja preparada para isso. A razão para esta funcionalidade não estar já implementada prende-se apenas com a falta de disponibilidade de calendário para tal.

**Implementação de *threads* em vez de processos**

A utilização de *threads*, ao invés de processos traria significativas vantagens de implementação da funcionalidade VPNSync. Porém, para que isto fosse possível seria imperativo a mudança da arquitetura do OpenVPN para suportar *threads*. Essa mudança parece estar programada para a versão 3 [40].

**Possíveis alterações futuras do OpenVPN que obriguem a alterações da solução desenvolvida**

Por último, assinalar o risco de possíveis alterações do OpenVPN poderem interferir com a solução desenvolvida e impedir o seu funcionamento correto. Portanto, poderá ser necessário fazer alterações ao código desenvolvido para garantir que a solução continua a funcionar de forma correta.

# Anexo A – Análise do Funcionamento do OpenVPN

Para descrever a arquitetura interna do OpenVPN utilizam-se módulos. Os módulos são estruturas conceptuais que ajudam a compreender a forma como o software funciona e como o código fonte está organizado. Cada módulo representa uma parte do software e está preparado para interagir com outros módulos.

A composição do OpenVPN é representada na seguinte figura:

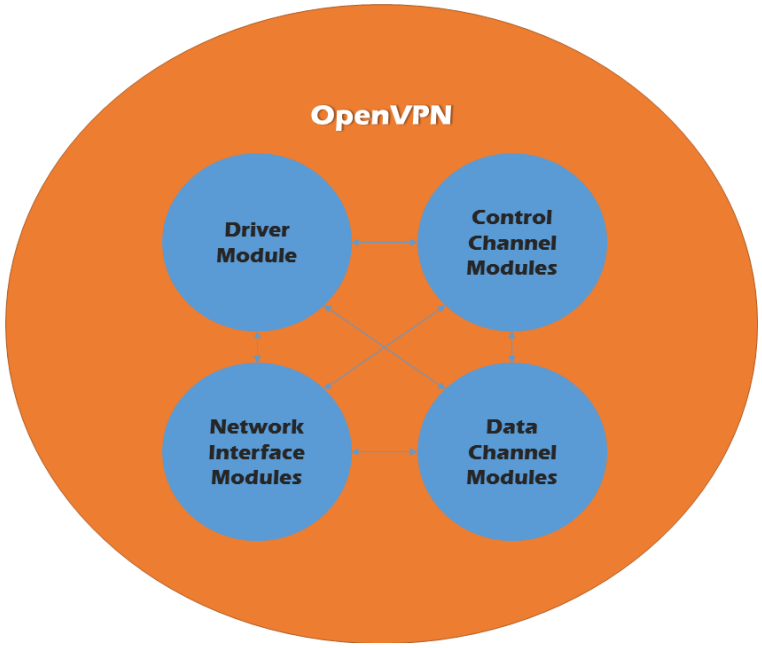



*Figura 20 - Representação dos módulos que formam o OpenVPN*

Cada módulo é descrito da seguinte forma:

- **Módulo Driver**

A rotina principal do Sistema. Este módulo está responsável pela gestão de eventos do OpenVPN. Para cada evento tem definido qual o caminho a tomar e comunica com os outros módulos para lhes delegar a resolução de eventos.

- **Módulos de Interface de Rede (Network Interface)**
  - Multiplexer Externo

Este módulo é responsável pela receção e envio de pacotes para máquinas remotas através da interface de rede externa. Na receção de pacotes faz ainda o *demultiplexing* (separação de sinais e encaminhamento) para o túnel VPN correspondente e a separação dos pacotes de controlo e de informação.

- Multiplexer Interno

Este módulo é responsável pelo envio e receção de pacotes de máquinas acessíveis dentro da rede através da interface de rede TUN/TAP. Na receção de pacotes determina ainda qual o túnel VPN que lhe corresponde para chegar ao destino correcto.

- **Módulos Control Channel**
  - The Reliability Layer

Este módulo define uma camada fiável e sequencial de transporte para mensagens de controlo (*control channel messages*).

- The Control Channel TLS module

Este módulo oferece um encapsulamento seguro das mensagens de controlo utilizando o protocolo TLS.

- The Control Channel Processor

Este módulo gere a criação, gestão e encerramento de tuneis VPN. Contém as estruturas *tls_multi* e *key_state* importantes.

- **Módulos Data Channel**
  - The Data Channel Control module

Este módulo controla o processamento de pacotes de informação (*data channel packets*). Dependendo das regras definidas no túnel VPN envia os pacotes para um ou mais dos três seguintes módulos.

- The Data Channel Crypto module

Este módulo aplica operações de segurança nos pacotes de informação.

- The Data Channel Fragmentation module

Este módulo proporciona a fragmentação dos pacotes de informação.

- The Data Channel Compression module

Este módulo proporciona a compressão dos pacotes de informação.

## Contexto Geral (multi-context)

O OpenVPN quando se inicia em modo servidor cria a estrutura *multi_context* que representa o seu contexto geral, isto é, todas as suas configurações de túneis VPN e dos seus processos, bem como o estado das ligações em curso.

Por cada cliente que se liga ao servidor uma estrutura do tipo *multi_instance* é criada e registada na estrutura *multi_context*. A estrutura *multi_instance* contém as informações de estado e configurações do túnel VPN que liga o servidor ao cliente em questão.

Além do registo das estruturas *multi_instance*, o *multi_context* faz ainda indexação de cada cliente ligado numa hash table.

Durante a ligação, o Multiplexer Interno e o Externo acedem à estrutura *multi_context* para saberem qual a *multi_instance* que corresponde a um endereço para que possam fazer o envio e receção de pacotes.

Data Fields

| | |
|---|---|
| int | thread_mode |
| struct multi_instance ** | instances |
| | Array of multi_instances. |
| struct hash * | hash |
| | VPN tunnel instances indexed by real address of the remote peer. |
| struct hash * | vhash |
| | VPN tunnel instances indexed by virtual address of remote hosts. |
| struct hash * | iter |
| | VPN tunnel instances indexed by real address of the remote peer, optimized for iteration. |
| struct schedule * | schedule |
| struct mbuf_set * | mbuf |
| | Set of buffers for passing data channel packets between VPN tunnel instances. |
| struct multi_tcp * | mtcp |
| | State specific to OpenVPN using TCP as external transport. |
| struct ifconfig_pool * | ifconfig_pool |
| struct frequency_limit * | new_connection_limiter |
| struct mroute_helper * | route_helper |
| struct multi_reap * | reaper |
| struct mroute_addr | local |
| bool | enable_c2c |
| int | max_clients |
| int | tcp_queue_limit |
| int | status_file_version |
| int | n_clients |
| struct multi_instance * | pending |
| struct multi_instance * | earliest_wakeup |
| struct multi_instance ** | mpp_touched |
| struct context_buffers * | context_buffers |
| time_t | per_second_trigger |
| struct context | top |
| | Storage structure for process-wide configuration. |
| struct event_timeout | stale_routes_check_et |
| struct deferred_signal_schedule_entry | deferred_shutdown_signal |

*Figura 21 - Composição da estrutura multi_context*

## Contexto Individual de cada Cliente ligado (multi_instance)

Tal como já referido, para cada cliente é gerada uma estrutura *multi_instance* que é armazenada no servidor e registada no seu contexto geral (*multi_context*). Esta contém informações de estado e configurações do túnel VPN incluindo a criptografia aplicada.

### Data Fields

| | |
|---|---|
| struct schedule_entry | se |
| struct gc_arena | gc |
| bool | defined |
| bool | halt |
| int | refcount |
| int | route_count |
| time_t | created |
| | Time at which a VPN tunnel instance was created. |
| struct timeval | wakeup |
| struct mroute_addr | real |
| | External network address of the remote peer. |
| ifconfig_pool_handle | vaddr_handle |
| char | msg_prefix [MULTI_PREFIX_MAX_LENGTH] |
| unsigned int | tcp_rwflags |
| struct mbuf_set * | tcp_link_out_deferred |
| bool | socket_set_called |
| in_addr_t | reporting_addr |
| struct in6_addr | reporting_addr_ipv6 |
| bool | did_open_context |
| bool | did_real_hash |
| bool | did_iter |
| bool | connection_established_flag |
| bool | did_iroutes |
| int | n_clients_delta |
| struct context | context |
| | The context structure storing state for this VPN tunnel. |

*Figura 22 - Composição da estrutura multi_instance*

## Inicialização de um túnel VPN

Quando um cliente se liga a um servidor VPN a primeira ação que é o aperto de mão ou *handshake*. Este aperto de mão é divido em 3 etapas:

- Primeira etapa
  - O cliente gera o seguinte material aleatório (random):
    - Pre-master secret: 48 bytes
    - PRF seed do Cliente para o master secret: 32 bytes
    - PRF seed do Cliente para a key expansion: 32 bytes
  - O cliente envia o material aleatório que gerou para o servidor.
- Segunda etapa:
  - O servidor gera o seguinte material aleatório:
    - PRF seed do Servidor para o master secret: 32 bytes
    - PRF seed do Servidor para a *expansion key*: 32 bytes
  - O servidor calcula a *expansion key* utilizando em conjunto o seu material aleatório e o que recebeu do cliente.
  - O servidor envia o material aleatório que gerou para o cliente.
- Terceira etapa:
  - O cliente calcula a *expansion key* utilizando em conjunto o seu material aleatório e o que recebeu do servidor.

Durante este processo é criada uma estrutura *key_state* que faz parte da estrutura *tls_multi*, que por sua vez faz parte da estrutura *multi_instance*. Quando existe uma renegociação dos parâmetros de segurança, devido a um *timeout* ou falha na ligação, este processo é repetido.

Data Fields

| | |
|---|---|
| int | state |
| int | key_id |
| | Key id for this key_state, inherited from struct tls_session. |
| struct key_state_ssl | ks_ssl |
| time_t | established |
| time_t | must_negotiate |
| time_t | must_die |
| int | initial_opcode |
| struct session_id | session_id_remote |
| struct link_socket_actual | remote_addr |
| struct crypto_options | crypto_options |
| struct key_source2 * | key_src |
| struct buffer | plaintext_read_buf |
| struct buffer | plaintext_write_buf |
| struct buffer | ack_write_buf |
| struct reliable * | send_reliable |
| struct reliable * | rec_reliable |
| struct reliable_ack * | rec_ack |
| struct buffer_list * | paybuf |
| counter_type | n_bytes |
| counter_type | n_packets |
| bool | authenticated |
| time_t | auth_deferred_expire |

*Figura 23 - Composição da estrutura key_state*

## Sessão SSL/TLS (tls_multi)

Uma sessão SSL/TLS é guardada na estrutura *tls_multi*, mais especificamente na estrutura *tls_session*. A estrutura *tls_session* contém informação de estado relativa à estrutura *key_state* ativa e à inativa, que é a antiga estrutura ativa, mais conhecida por *lame duck*. O ciclo de vida da estrutura *tls_session* é definido da seguinte forma:

1. Inicialização:
    - Inicializa uma estrutura *tls_session*
    - Inicializa a estrutura *key_state* primária
2. Renegociação:
    - Faz a limpeza do antigo lame duck *key_state*
    - Transfere o antigo *key_state* primário para o novo *key_state lame duck*
    - Inicializa um novo *key_state* primário
3. Limpeza:
    - Faz a limpeza da estrutura *tls_session*
    - Faz a limpeza de todos os objetos *key_state* associados com a sessão

Data Fields

| | |
|---|---|
| struct tls_options | opt |
| struct key_state * | key_scan [KEY_SCAN_SIZE] |
| | List of `key_state` objects in the order they should be scanned by data channel modules. |
| struct key_state * | save_ks |
| struct link_socket_actual | to_link_addr |
| int | n_sessions |
| | Number of sessions negotiated thus far. |
| int | n_hard_errors |
| int | n_soft_errors |
| char * | locked_cn |
| char * | locked_username |
| struct cert_hash_set * | locked_cert_hash_set |
| char * | peer_info |
| uint32_t | peer_id |
| bool | use_peer_id |
| char * | remote_ciphername |
| | cipher specified in peer's config file |
| char * | auth_token |
| | If server sends a generated auth-token, this is the token to use for future user/pass authentications in this session. |
| time_t | auth_token_tstamp |
| | timestamp of the generated token |
| bool | auth_token_sent |
| | If server uses –auth-gen-token and token has been sent to client. |
| struct tls_session | session [TM_SIZE] |
| | Array of `tls_session` objects representing control channel sessions with the remote peer. |

*Figura 24 - Composição da estrutura tls_multi*

Esta análise foi baseada na interpretação do código fonte do OpenVPN bem como na informação disponibilizada pela documentação Doxygen [39].

# Anexo B – Tutorial de Instalação do Servidor OpenVPN em OpenBSD

1. **Instalar o OpenBSD (download install61.iso)**
2. **Indicar onde estão os programas possíveis de instalar:**
    a. Escolher uma das fontes desta lista https://www.openbsd.org/ftp.html e copiar o endereço até ao sitio dos packages.
    b. Introduzir os seguintes comandos na *shell*:

*#echo "PKG_PATH= http://ftp.heanet.ie/pub/OpenBSD/6.1/packages/amd64/" >> .profile*

*#echo "export PKG_PATH" >> .profile*

*#cat .profile*

3. **Instalar o editor nano**

*#pkg_add nano*

4. **Definir Interface em0 como Internal Network e atribuir-lhe um IP estático:**
    a. Alterar ficheiro /etc/hostname.em0 para:

*inet 192.168.0.1 255.255.255.0*

5. **Instalar package openvpn:**

*#pkg_add openvpn*

6. **Preparar as diretorias para os ficheiros de configuração e log do openvpn:**

*#install -m 700 -d /etc/openvpn/private*

*#install -m 700 -d /etc/openvpn/private-client-conf*

*#install -m 755 -d /etc/openvpn/certs*

*#install -m 755 -d /var/log/openvpn*

7. **Instalar EASYRSA 3 para criação de certificados e chaves:**

*#pkg_add easy-rsa*

*#cd /usr/local/share/easy-rsa/*

*#ls -alpd easyrsa vars**

*#less vars.example*

8. **Definir parâmetros e diretorias para o easy-rsa:**

*#pkiDir="/etc/openvpn/easy-rsa-pki/"*

*#mkdir ${pkiDir}*

*#easyrsaDir="/usr/local/share/easy-rsa/"*

*#cd ${easyrsaDir}*

9. **Gerar a ta.key para segurança adicional (protege contra UDP *flood*):**

*#ls -alp /etc/openvpn/private/vpn-ta.key || openvpn --genkey --secret*

*/etc/openvpn/private/vpn-ta.key*

10. **Inicializar o pki (cria as diretorias para o pki):**

*#./easyrsa --batch=0 --pki-dir=${pkiDir} init-pki*

*#ls -alpd ${pkiDir} ${pkiDir}/**

11. **Criar parâmetros Diffie-Hellman:**

*#./easy-rsa --batch=1 --pki-dir=${pkiDir} gen-dh*

*#ls -alpd ${pkiDir}/dh.pem*

12. **Criar o certificado de autoridade (CA) e verificá-lo:**

*#./easyrsa --batch=1 --pki-dir=${pkiDir} --req-cn=vpn-ca build-ca nopass*

*#ls -alpd ${pkiDir}/ca.crt ${pkiDir}/private/ca.key ${pkiDir}/index.txt ${pkiDir}/serial*

*#openssl x509 -in ${pkiDir}/ca.crt -text -noout*

*#openssl rsa -in ${pkiDir}/private/ca.key -check -noout*

13. **Gerar o server request:**

*#./easyrsa --batch=1 --pki-dir=${pkiDir} --req-cn=vpnserver gen-req vpnserver nopass*

*#ls -alpd ${pkiDir}/reqs/vpnserver.req ${pkiDir}/private/vpnserver.key*

*#openssl req -in ${pkiDir}/reqs/vpnserver.req -text -noout*

*#openssl rsa -in ${pkiDir}/private/vpnserver.key -check -noout*

14. **Verificar e assinar o server request:**

*#./easyrsa --batch=1 --pki-dir=${pkiDir} show-req vpnserver*

*#./easyrsa --batch=1 --pki-dir=${pkiDir} sign server vpnserver*

*#ls -alpd ${pkiDir}/issued/vpnserver.crt ${pkiDir}/certs_by_serial/01.pem*

*#openssl x509 -in ${pkiDir}/issued/vpnserver.crt -text -noout*

*#echo "Last added cert in db: 'cat ${pkiDir}/index.txt| -1'"*

*#echo "Next added cert will have number: 'cat ${pkiDir}/serial'"*

**15. Criar a Lista de Revocação de Certificados (CRL):**

*#./easyrsa --batch=1 --pki-dir=${pkiDir} gen-crl*

*#chown :_openvpn ${pkiDir}/crl.pem; chmod g+r ${pkiDir}/crl.pem*

*#ls -alF ${pkiDir}/crl.pem*

*#openssl crl -in ${pkiDir}/crl.pem -text -noout*

**16. Copiar as chaves e os certificados para as pastas de configuração do openvpn:**

*#cp -p ${pkiDir}/ca.crt /etc/openvpn/certs/vpn-ca.crt*

*#cp -p ${pkiDir}/private/ca.key /etc/openvpn/private/vpn-ca.key*

*#cp -p ${pkiDir}/issued/vpnserver.crt /etc/openvpn/certs/vpnserver.crt*

*#cp -p ${pkiDir}/private/vpnserver.key /etc/openvpn/private/vpnserver.key*

*#cp -p ${pkiDir}/dh.pem /etc/openvpn/dh.pem*

*#cp -p ${pkiDir}/crl.pem /etc/openvpn/crl.pem*

**17. Verificar todos os certificados:**

*#openssl x509 -in /etc/openvpn/certs/vpn-ca.crt -text -noout*

*#openssl x509 -in /etc/openvpn/certs/vpnserver.crt -text -noout*

*#openssl crl -in /etc/openvpn/crl.pem -text -noout*

*#openssl rsa -in /etc/openvpn/private/vpn-ca.key -check -noout*

*#openssl rsa -in /etc/openvpn/private/vpnserver.key -check -noout*

**18. Interface de Gestão (opcional):**

*#test -f /etc/openvpn/private/mgmt.pwd || touch*

*/etc/openvpn/private/mgmt.pwd*

*#chown root:wheel /etc/openvpn/private/mgmt.pwd; chmod 600*

*/etc/openvpn/private/mgmt.pwd*

*#nano /etc/openvpn/private/mgmt.pwd*

*Inserir a password no document e guardar*

*Ligar à openvpn management console: #telnet localhost 1195*

**19. Fazer o setup da configuração do servidor OpenVPN:**

*#test -f /etc/openvpn/server_tun0.conf || touch /etc/openvpn/server_tun0.conf*

*#chown root:_openvpn /etc/openvpn/server_tun0.conf; chmod 640 /etc/openvpn/server_tun0.conf*

*#nano /etc/openvpn/server_tun0.conf*

a. Inserir as seguintes linhas no ficheiro:

```
dev tun0
port 1194
proto udp
cipher AES-256-CBC
auth SHA256
ncp-disable
ca /etc/openvpn/certs/vpn-ca.crt
cert /etc/openvpn/certs/vpnserver.crt
key /etc/openvpn/private/vpnserver.key
dh /etc/openvpn/dh.pem
tls-auth /etc/openvpn/private/vpn-ta.key 0
#
local 192.168.0.100
server 10.8.0.0 255.255.255.0
status /var/log/openvpn/openvpn-status.log
log-append /var/log/openvpn/openvpn.log
management 127.0.0.1 1195 /etc/openvpn/private/mgmt.pwd
crl-verify /etc/openvpn/crl.pem
persist-key
persist-tun
#
keepalive 30 90
user _openvpn
group _openvpn
verb 3
```

**20. Editar regras da firewall (opcional)**

a. Editar ficheiro /etc/pf.conf

**21. Ligar o IP forwarding:**

*#nano /etc/sysctl.conf*

*Alterar a linha para: net.inet.ip.forwarding=1*

**22. Ligar o OpenVPN:**

*#/usr/local/sbin/openvpn --daemon --config /etc/openvpn/server_tun0.conf*

**23. Verificar se o OpenVPN está a funcionar correctamente:**

*#ifconfig tun0*

*#pgrep -fl openvpn.*server*

*#netstat -na|grep '\.1194'*

*#ls -al /var/log/openvpn/**

*#tail /var/log/openvpn/openvpn-status.log*

*#tail /var/log/openvpn/openvpn.log*

**24. Parar OpenVPN:**

*#pgrep -fl openvpn pgrep -xl openvpn*

*#pkill -x openvpn*

**25. Começar OpenVPN no arranque do sistema operativo:**

*#nano /etc/hostname.tun0*

a. Adicionar as linhas seguintes ao ficheiro:

up
group _openvpn
description "OpenVPN to local net 192.168.0.x"
!/usr/local/sbin/openvpn --config /etc/openvpn/server_$if.conf & false
!echo "Starting OpenVPN on $if"

**26. Adicionar um cliente (gerar certificado e chave):**

*#vpnclientuser=vpnclient1*

*#./easyrsa --batch=1 --pki-dir=${pkiDir} --req-cn=${vpnclientuser} gen-req ${vpnclientuser} nopass*

*#ls -alpd ${pkiDir}/reqs/${vpnclientuser}.req*

*${pkiDir}/private/${vpnclientuser}.key*

*#openssl req -in ${pkiDir}/reqs/${vpnclientuser}.req -text -noout*

*#openssl rsa -in ${pkiDir}/private/${vpnclientuser}.key -check -noout*

a. Assinar o certificado:

*#./easyrsa --batch=1 --pki-dir=${pkiDir} show-req ${vpnclientuser}*

*#./easyrsa --batch=1 --pki-dir=${pkiDir} sign client ${vpnclientuser}*

*#openssl x509 -in ${pkiDir}/issued/${vpnclientuser}.crt -text -noout*

**27. Criar ficheiro ovpn para enviar ao cliente para que este se possa ligar:**

*#nano /etc/openvpn/vpnclient1.conf.ovpn*

a. Adicionar as seguintes linhas ao ficheiro:

```
client
dev tun0
proto udp
cipher AES-256-CBC
auth SHA256
remote 192.168.0.100 1194
persist-key
persist-tun
tls-auth ta.key 1
user _openvpn
group _openvpn
verb 3
key-direction 1
<ca>
Inserir conteúdo do ficheiro vpn-ca.crt
</ca>
<cert>
Inserir conteúdo do ficheiro vpnclient1.crt
</cert>
<key>
Inserir conteúdo do ficheiro vpnclient1.key
</key>
<tls-auth>
Inserir conteúdo do ficheiro vpn-ta.key
</tls-auth>
```

**28. Enviar ficheiro vpnclient1.conf.ovpn para o cliente.**

**29. Para testar a ligação usar a ferramenta *Netcat*:**

*#nc -l 31000*

a. Esperar pela ligação do cliente e pela receção de mensagens

**30. Para receber um ficheiro:**

*#nc -l 31000 > filename.out*

a. Esperar pela ligação do cliente e pela receção do ficheiro

**31. Para verificar informação de clientes ligados:**

#tail /var/log/openvpn/openvpn-status.log

# Anexo C – Tutorial de Instalação do Cliente OpenVPN em OpenBSD

1. **Instalar o OpenBSD**

2. **Definir Interface em0 como Internal Network e atribuir-lhe um IP estático:**
   a. Alterar ficheiro /etc/hostname.em0 para:

   *inet 192.168.0.20 255.255.255.0*

3. **Instalar package openvpn:**

   *#pkg_add openvpn*

4. **Guardar ficheiro ovpn recebido do servidor OpenVPN em /etc/openvpn/.**

5. **Começar OpenVPN no arranque do sistema operativo:**

   *#nano /etc/hostname.tun0*

   a. Adicionar as seguintes linhas ao ficheiro:
   up
   group openvpnclient
   description “OpenVPN client to local openvpn server”
   !/usr/local/sbin/openvpn --config /etc/openvpn/vpnclient.conf.ovpn -dev $if &false
   !echo “Starting OpenVPN client on $if”

6. **Verificar se a ligação está ativa:**

   *#ifconfig tun0*
   *#pgrep -fl openvpn.*client*
   *#less /var/log/messages*

7. **Testar a ligação enviando mensagens para o servidor:**

   *#nc 192.168.0.100 -p 31000*
   *#texto a enviar*

   a. Esperar resposta por parte do servidor

8. **Testar a ligação enviando um ficheiro:**

   *#nc 192.168.0.1 31000 < nomedoficheiro.in*

   a. Esperar pela conclusão da transferência

# Anexo D – Configuração de servidores em cluster CARP

Para configurar um *cluster* de servidores utilizando o CARP introduz-se as seguintes configurações em cada um dos servidores.

## Servidor Principal

No servidor principal criar um script com os seguintes comandos:

*#sysctl net.inet.carp.preempt=1*

*#ifconfig carp0 create*

*#ifconfig carp0 vhid 1 carpdev em0 pass lanpasswd 192.168.0.100 netmask 255.255.255.0*

## Servidor(es) de Reserva

No(s) servidor(es) de reserva criar um script com os seguintes comandos:

*#sysctl net.inet.carp.preempt=1*

*#ifconfig carp0 create*

*#ifconfig carp0 vhid 1 carpdev em0 pass lanpasswd advskew 128 192.168.0.100 netmask 255.255.255.0*

A diferença entre a configuração do servidor principal e os servidores de reserva está no atributo *advskew*. Este atributo define o grau de prioridade que a máquina tem de ser a máquina principal, quanto menor o valor maior preferência terá para substituir o principal. Quando omitido o seu valor é zero.

Após a execução do script para verificar o estado do *cluster* basta executar o comando:

*#ifconfig carp0*

# Anexo E – Implementação da *flag* vpnsync

Esta implementação possibilita a inicialização do vpnsync através de uma *flag* configurada no OpenVPN (na linha de comandos ou no ficheiro server_tun0.conf).

**Exemplo (na linha de comandos):**

*#/usr/local/sbin/openvpn --daemon --vpnsync --config /…/server_tun0.conf*

No ficheiro server_tun0.conf:

*vpnsync*

**Para implementar esta funcionalidade foi necessário alterar os seguintes ficheiros:**

- config-msvc.h
- configure
- configure.ac
- config.h

na pasta /openvpn-2.4.1.

**E os ficheiros:**

- options.h
- options.c

na pasta /openvpn-2.4.1/src/openvpn.

**No ficheiro config-msvc.h:**

*//linha 23*
*#define ENABLE_VPNSYNC 1*

**No ficheiro configure:**

*//linha 836*
*enable_vpnsync*
*//linha 1541*
*--disable-vpnsync disable tunnel sync server support [default=yes]*
*// linha 4987*
*# Check whether --enable-vpnsync was given.*
*if test "${enable_vpnsync+set}" = set; then :*

```
  enableval=$enable_vpnsync;
else
  enable_vpnsync="yes"
fi
// linha 16747
test "${enable_vpnsync}" = "yes" &&
$as_echo "#define ENABLE_VPNSYNC 1" >>confdefs.h
```

**No ficheiro configure.ac:**

```
//linha 130
AC_ARG_ENABLE(
        [vpnsync],
        [AS_HELP_STRING([--disable-vpnsync], [disable tunnel sync server support
@<:@default=yes@:>@])],
        ,
        [enable_vpnsync="yes"]
)
//linha 1174
test "${enable_vpnsync}" = "yes" && AC_DEFINE([ENABLE_VPNSYNC], [1], [Enable
tunnel sync server capability])
```

**No ficheiro config.h:**

```
// linha 5
/* Configuration settings */
#define CONFIGURE_DEFINES "enable_crypto=yes enable_crypto_ofb_cfb=yes
enable_debug=yes enable_def_auth=yes enable_dlopen=unknown
enable_dlopen_self=unknown enable_dlopen_self_static=unknown
enable_fast_install=needless enable_fragment=yes enable_http_proxy=yes
enable_iproute2=no enable_libtool_lock=yes enable_lzo=yes enable_lzo_stub=no
enable_management=yes enable_multi=yes enable_multihome=yes
enable_pam_dlopen=no enable_pedantic=no enable_pf=yes enable_pkcs11=no
enable_plugin_auth_pam=no enable_plugin_down_root=yes enable_plugins=yes
enable_port_share=yes enable_selinux=no enable_server=yes enable_shared=yes
enable_shared_with_static_runtimes=no enable_small=no enable_socks=yes
enable_ssl=yes enable_static=yes enable_strict=no enable_strict_options=no
enable_systemd=no enable_vpnsync=yes enable_win32_dll=yes
enable_x509_alt_username=no with_crypto_library=openssl with_gnu_ld=no
with_mem_check=no with_plugindir='$(libdir)/openvpn/plugins' with_sysroot=no"
// linha 98
/* Enable tunnel sync server capability */
#define ENABLE_VPNSYNC 1
```

**No ficheiro options.h:**

```
// linha 168
/* VPN SYNC COMMAND LINE OPTION */
#ifdef ENABLE_VPNSYNC
  bool vpnsync_boolean;
  const char *master_ip;
  int vpnsync_port;
#endif
};
```

**No ficheiro options.c:**

```
// linha 768
#ifdef ENABLE_VPNSYNC
  "\n"
  "VPN Tunnel State Synchronization Between Servers:\n"
  "--vpnsync ip port    : Activate Tunnel State Synchronization Between Servers\n"
  "                This must be used with CARP protocol activated\n"
  "                ip is master server ip and port is master server port\n"
  "                ip and port only have to be set on backup servers.\n"
#endif
// linha 911
/* VPNSYNC INIT OPTION */
#ifdef ENABLE_VPNSYNC
  o->vpnsync_boolean=false;
#endif
// linha 1565
#ifdef ENABLE_VPNSYNC
  SHOW_BOOL(vpnsync_boolean);
  SHOW_STR (master_ip);
  SHOW_INT (vpnsync_port);
#endif
// linha 8107
#ifdef ENABLE_VPNSYNC
  else if (streq (p[0], "vpnsync")) {
     options->vpnsync_boolean = true;
     options->vpnsync_port = 1196;
     if (p[1]) {
       options->master_ip = p[1];
       if(p[2]) {
         options->vpnsync_port = atoi (p[2]);
       }
     }
   }
#endif
```

# Anexo F – Implementação dos ficheiros VPNSync e Binn

Para adicionar os ficheiros vpnsync.h, vpnsync.c, binn.h e binn.c à compilação do OpenVPN foi necessário alterar os seguintes ficheiros:

- Makefile
- Makefile.am
- Makefile.in

dentro da pasta /openvpn-2.4.1/src/openvpn.

**No ficheiro Makefile:**

*// linha 87*
*am__openvpn_SOURCES_DIST = binn.c binn.h vpnsync.c vpnsync.h argv.c* \
*// linha 119*
*am_openvpn_OBJECTS = binn.$(OBJEXT) vpnsync.$(OBJEXT) argv.$(OBJEXT)* \
*// linha 374*
*openvpn_SOURCES = binn.c binn.h vpnsync.c vpnsync.h argv.c argv.h* \
*// linha 482*
*include ./$(DEPDIR)/binn.Po*
*// linha 556*
*include ./$(DEPDIR)/vpnsync.Po*

**No ficheiro Makefile.am:**

```
// linha 40
openvpn_SOURCES = \
        binn.c binn.h \
        vpnsync.c vpnsync.h \
```

**No ficheiro Makefile.in:**

```
// linha 87
am__openvpn_SOURCES_DIST = binn.c binn.h vpnsync.c vpnsync.h argv.c \
// linha 119
am_openvpn_OBJECTS = binn.$(OBJEXT) vpnsync.$(OBJEXT) argv.$(OBJEXT) \
// linha 374
openvpn_SOURCES = binn.c binn.h vpnsync.c vpnsync.h argv.c argv.h \
// linha 482
@AMDEP_TRUE@@am__include@
@am__quote@./$(DEPDIR)/binn.Po@am__quote@
// linha 556
@AMDEP_TRUE@@am__include@
@am__quote@./$(DEPDIR)/vpnsync.Po@am__quote@
```

# Anexo G – Scripts de automatização dos testes de validação

Nos servidores os seguintes scripts ajudaram a automatizar os testes de validação:

- *script_setsysflags*
- *script_build*
- *script_carp*
- *script_openvpn*
- *script_fail*

**script_setsysflags**

```
#!/bin/sh
echo "Setting System Flags"
sysctl net.inet.carp.preempt=1
sysctl net.inet.ip.forwarding=1
sysctl net.inet.ip.mforwarding=1
echo 'net.inet.carp.preempt=1' >> /etc/sysctl.conf
echo 'net.inet.ip.forwarding=1' >> /etc/sysctl.conf
echo 'net.inet.ip.mforwarding=1' >> /etc/sysctl.conf
```

**script_build**

```
#!/bin/sh
alias proj="cd ./openvpn-2.4.1"
proj
./configure LDFLAGS="-L/usr/local/lib" CFLAGS="-I/usr/local/include"
export AUTOCONF_VERSION=2.69 AUTOMAKE_VERSION=1.9
autoreconf -f -i
make
make clean
make install
```

**script_carp**

```
#!/bin/sh
echo "Starting CARP"
ifconfig carp0 create
ifconfig carp0 carpdev em1 vhid 1 pass lanpasswd 192.168.0.100 netmask
255.255.255.0
```

Todos os servidores usam o mesmo script em relação ao CARP para garantir que um servidor quando inicia e já existe um outro definido como principal este passa sempre a servidor de reserva. Isto é relevante no caso de um servidor principal que falha e que depois volta a iniciar. Com esta configuração ele passa para servidor de reserva e não de volta para servidor principal.

**script_openvpn**

```
#!/bin/sh
echo "Starting OpenVPN Server"
/usr/local/sbin/openvpn --daemon --config ./configs/server_tun0.conf & false
```

**script_fail**

```
#!/bin/sh
pkill -SIGTERM -x openvpn
ifconfig carp0 destroy
echo 'OpenVPN Server terminated'
echo 'CARP Interface terminated'
```

No cliente apenas foi necessário desenvolver um *script* com o inuito de iniciar o OpenVPN carregando o ficheiro de configuração:

- *script_client*

**script_client**

```
#!/bin/sh
echo "Starting OpenVPN Client"
/usr/local/sbin/openvpn --daemon --config /etc/openvpn/vpnclient.conf & false
```

# Anexo H – Documentação Doxygen do VPNSync

# VPNSync 1.0

OpenVPN Server Side Failover with Synchronization of Clients State

Main Page | Modules | Data Structures | Files | Search

## VPNSync Source Code Documentation

## Introduction

The VPNSync feature allows for a cluster of OpenVPN servers to synchronize connected clients state. The Master OpenVPN server has the responsability to mantain the Backup Servers updated on the client state info.

## Requirements

The VPNSync module to work correctly needs:

- **2.4.1**, OpenVPN version that VPNSync is implemented.
- **Servers** must be on the same network.
- **CARP** interface configured on all servers, so that a single IP points to the cluster of servers.
- **UDP**, OpenVPN tunnel has to use UDP (TCP is not supported).
- **Port 1196** open, used for multicast communications.
- **Port 1197** open, used for unicast communications (not available yet).
- **AES-256-CBC** cipher, (other ciphers are not supported as of yet).
- **SHA256** or **SHA512** authentication.
- **Flag** in conf file, to activate VPNSync functionality the flag 'vpnsync' must be included in the servers config file.

## Files

The VPNSync Module consists of 4 files:

- vpnsync.h, the header for the VPNSync functions
- vpnsync.c, the C file containing the VPNSync functions
- binn.h, the header file for the Binn serialization library functions
- binn.c, the C file containing the Binn serialization library functions

Plus minor changes to these OpenVPN files:

- **mudp.c**, the C file responsible for UDP server operations
- **multi.c**, the C file responsible for incoming and outgoing packet management

## How it Works

When a Client connects to the IP defined by CARP the connection is made with the master server defined at the moment. After the 3-Way-Handshake ends between the client and Master Server the Server sends the information stored about the client to all Backup Servers on the cluster. The Backup Servers initialize a 'fake' client with the information received from the Master and prepare to exchange packets with the client if the Master Server fails/crashes.
If/when the master server stops working, one backup server takes his place and starts exchanging packets with the client. This change is most of the times transparent to the client. It could occasionally show warnings about replay packets being received, but that is to be expected due to synchronization timing constraints.

Generated by doxygen 1.8.13

# VPNSync 1.0

OpenVPN Server Side Failover with Synchronization of Clients State

Main Page | Modules | Data Structures | Files | Search

## Master Server VPNSync Module

**Features:**

- Send Client state info to all Backup Servers when he connects.
- Send updates from Client state to all Backup Servers (related to packets exchanged and renegotiations).
- Send all Clients state info to a specific Backup Server who was added to the cluster (not yet available).

**Description**

The VPNSync functionality on the Master Server is activated by adding the flag 'vpnsync' to the OpenVPN config file. When VPNSync is active the following happens:

- On initialization, the Server opens a multicast socket to communicate with Backup Servers.
- When a client connects, the Server completes the 3-Way-Handshake with the client and after sends two multicast packets with client details and tunnel crypto to the Backup Servers.
- When a client renegotiates its connection, the Server sends the updated information to Backup Servers through multicast.

**In Development**

- On initialization, the Server opens a unicast socket to communicate with Backup Servers.
- When a Backup Server joins the CARP cluster and asks for a dump of all client states through a unicast message, the Server replies with the information of all the clients it has connected at the moment.

Generated by doxygen 1.8.13

# VPNSync 1.0

OpenVPN Server Side Failover with Synchronization of Clients State

Main Page | Modules | Data Structures | Files | Search

## Backup Servers VPNSync Module

**Features:**

- Receive Client state info from Master Server when client connects.
- Receive updates to Client state from Master Server (related to packets exchanged and renegotiations).
- Ask for all Clients state info dump to the Master Server (not yet available).

**Description**

The VPNSync functionality on the Backup Servers is activated by adding the flag 'vpnsync' to the OpenVPN config file. When Unicast Dump feature is available the Master IP and Port have to be defined after the flag, like 'vpnsync 192.168.0.1 1196'. When VPNSync is active the following happens:

- On initialization, the Server opens a multicast socket to receive packets from the Master Server.
- On initialization, the Server also creates a child process to continuously monitor the CARP state.
- When the Server receives a multicast packet, it deserializes the Binn object and creates a new 'fake' client with the information received. If it is an update, it updates the respective client in the system.
- If the CARP state transitions to Master the Server terminates the child processes and starts the VPNSync feature as a Master Server.

**In Development**

- On initialization, the Server opens a unicast socket to communicate with the Master Server.
- When the Server joins the CARP cluster and asks for a dump of all client states through a unicast message, the Master Server replies with the information of all the clients it has connected at the moment.

Generated by doxygen 1.8.13

# VPNSync 1.0

OpenVPN Server Side Failover with Synchronization of Clients State

Main Page | Modules | Data Structures | Files | Search

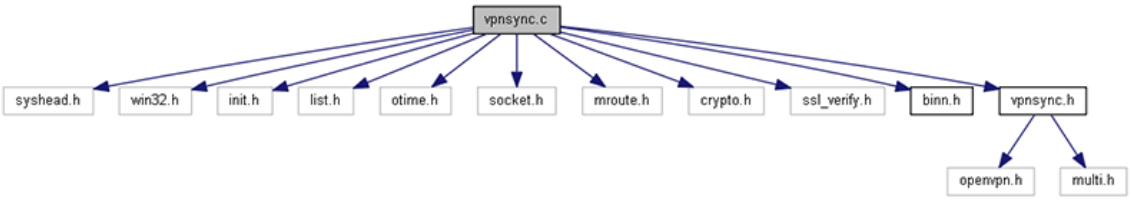
Macros | Functions | Variables

## vpnsync.c File Reference

C File containing the definition of VPNSync functions. More...

```
#include "syshead.h"
#include "win32.h"
#include "init.h"
#include "list.h"
#include "otime.h"
#include "socket.h"
#include "mroute.h"
#include "crypto.h"
#include "ssl_verify.h"
#include "binn.h"
#include "vpnsync.h"
```

Include dependency graph for vpnsync.c:




### Macros

| | |
|---|---|
| #define | **BUFLEN** 512 |
| #define | **MULTICAST_ADDR** "239.0.2.0" |

### Functions

| | |
|---|---|
| const char * | **print_session_id** (const struct session_id *sid, struct gc_arena *gc)<br>Master/Backup: Print Session ID Info. More... |
| void | **set_keys** (struct key2 *k, const struct key_type *kt, const char *prefix0, const char *prefix1)<br>Master: Get Cipher and HMAC Keys and store them on global variables k2 and kt2. More... |
| struct multi_instance * | **create_client_on_backup** (void *buf, struct multi_context *m)<br>Backup: Create 'Fake' Client with Info received from Master Server. More... |
| int | **send_clients_state_update** (struct multi_instance *mi, int socketfd, struct sockaddr_in send_socket)<br>Master: Send a Client State Update to Backup Servers. More... |
| int | **send_clients_state** (struct multi_instance *mi, int socketfd, struct sockaddr_in send_socket)<br>Master: Send a Client State to Backup Servers. More... |
| void | **apply_client_state_update** (void *buf, struct multi_context *m)<br>Backup: Apply a Client State update coming from the Master Server. More... |
| void | **receive_clients_state** (struct context *c, struct multi_context *m)<br>Backup: Receive a Client State from the Master Server. More... |
| int | **ask_for_clients_state_dump** (struct multi_context *m)<br>Backup Server: Ask Master Server for Clients State Dump. More... |
| int | **send_all_clients_state** (struct multi_context *m)<br>Master: Send All Clients State Dump to a specific Backup Server. More... |
| void | **wait_request_for_dump** (struct multi_context *m) |

| | |
|---|---|
| | Master: Wait for Backup Servers to ask for Clients State Dump. More... |
| void | **handle_children** (int signum)<br>Master/Backup: Handler to terminate Child Processes (SIGCHLD). More... |
| void | **close_unicast_socket** ()<br>Master/Backup: Close Unicast Socket. More... |
| int | **open_unicast_socket** (int multicast_port, int flag)<br>Master/Backup: Open Unicast Socket for Clients State Dump. More... |
| void | **close_multicast_socket** ()<br>Master/Backup: Close Multicast Socket. More... |
| int | **open_multicast_socket** (int multicast_port)<br>Master/Backup: Open Multicast Socket for new Client State and Updates. More... |
| int | **set_master_struct** (const char *master_server, int master_port)<br>Backup: Tries to translate Server IP - If successful means Master Server is reachable. More... |
| int | **check_carp_state** (int repeat)<br>Backup: Check CARP Interface to find if Server is Master or Backup (Loops if needed). More... |
| int | **handle_master_server** (struct context *c, struct multi_context *m, struct multi_instance *mi, int action)<br>Master: Handle Master Server operations. More... |
| int | **handle_backup_server** (struct context *c, struct multi_context *m, int flag)<br>Backup: Handle Backup Server operations. More... |
| void | **init_vpnsync** (struct context *c, struct multi_context *m)<br>Master/Backup: Main Function - Initialize VPNSync operations. More... |

## Variables

| | |
|---|---|
| struct sockaddr_in | **master_sock_struct** |
| struct sockaddr_in | **backup_sock_struct** |
| struct sockaddr_in | **multicast_sock_struct** |
| int | **sockfd_multi** |
| int | **sockfd_uni** |
| struct ip_mreq | **group** |
| int | **pipefd** [2] |
| struct key2 * | **k2** |
| struct key_type * | **kt2** |

## Detailed Description

C File containing the definition of VPNSync functions.

**Author**

dbastos

**Date**

Sep 2016 to June 2017

Definition in file **vpnsync.c**.

## Macro Definition Documentation

### ◆ BUFLEN

```
#define BUFLEN   512
```

Max length of buffer

Definition at line **26** of file **vpnsync.c**.

### ◆ MULTICAST_ADDR

```
#define MULTICAST_ADDR   "239.0.2.0"
```

Multicast Address

Definition at line **28** of file **vpnsync.c**.

## Function Documentation

### ◆ apply_client_state_update()

```
void apply_client_state_update ( void *                 buf,
                                 struct multi_context *  m
                               )
```

Backup: Apply a Client State update coming from the Master Server.

This function applies received information from Master Server multicast packets containing updated client state. It is called by **handle_backup_server()** after receiving a multicast packet with the "Client Update" information from the function **receive_clients_state()**. The information applied is the packet ID counter and timestamp for a specific client.

**Parameters**

**buf** Buffer with info received from Master Server

**m** Backup Server multi_context structure

**Returns**

Returns the resulting multi_instance structure with client information.

Definition at line **457** of file **vpnsync.c**.

### ◆ ask_for_clients_state_dump()

```
int ask_for_clients_state_dump ( struct multi_context *  m )
```

Backup Server: Ask Master Server for Clients State Dump.

This function asks Master Server for a dump of all clients state information. It is called by **handle_backup_server()** once at the beginning. Its purpose is to synchronize clients who previously connected to the Master Server when the Backup Server didn't existed yet. Note: Unicast dump feature is not available yet.

**Parameters**

**m** multi_context Server structure

**Returns**

Returns 0 if request was successful. -1 if request failed.

Definition at line **516** of file **vpnsync.c**.

Here is the call graph for this function:


## ◆ check_carp_state()

int check_carp_state ( int repeat )

Backup: Check CARP Interface to find if Server is Master or Backup (Loops if needed).

This function checks the current state of the CARP interface. It's called by **init_vpnsync()** for a initial check and by **handle_backup_server()** for a continuous check on the state of the Backup Server (on a child process). It is also called by **handle_master_server()** to ensure that only the Master Server is able to send client states. If `repeat` input is set to zero (0) then it checks once and returns the result, if it's set to one (1) then it loops until the state is Master. If the state is Master and repeat is set to one then it writes on a pipe the information that the Backup Server just transitioned to Master. The child process is terminated when the Backup Server transitions to Master.

**Parameters**

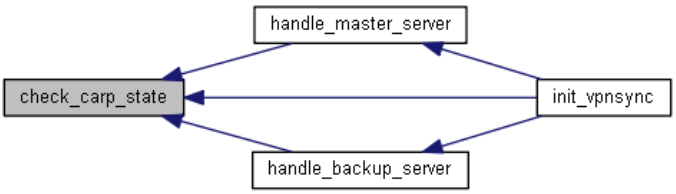
**repeat** 0 if is to check once. 1 if is to loop until Server is Master.

**Returns**

Returns 1 if Server is Master. 2 if Server is Backup.

Definition at line **726** of file **vpnsync.c**.

Here is the caller graph for this function:




## ◆ close_multicast_socket()

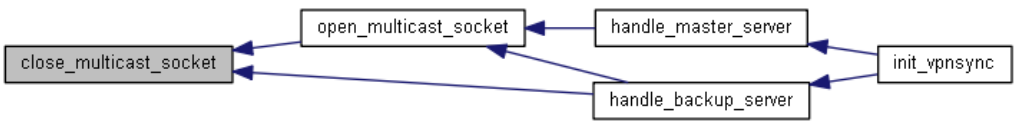
void close_multicast_socket ( )

Master/Backup: Close Multicast Socket.

This function closes the multicast socket. It is called by **handle_backup_server()** in the case of Backup Server and by tunnel_server_udp_single_threaded() in the case of Master Server.

Definition at line **654** of file **vpnsync.c**.

Here is the caller graph for this function:




## ◆ close_unicast_socket()

void close_unicast_socket ( )

Master/Backup: Close Unicast Socket.

This function closes the unicast socket. It is called by **handle_backup_server()** in the case of Backup Server and by tunnel_server_udp_single_threaded() in the case of Master Server. Note: Unicast dump feature is not available yet.

Definition at line **618** of file **vpnsync.c**.

Here is the caller graph for this function:


## ◆ create_client_on_backup()

```
struct multi_instance* create_client_on_backup ( void *                 buf,
                                                 struct multi_context * m
                                               )
```

Backup: Create 'Fake' Client with Info received from Master Server.

This function efectively creates a client instance on the Backup Server. It is called by handle_backup_client() after **receive_clients_state()** receives a packet from the Master Server. It deserializes the Binn object and uses the information to call multi_create_instance() and add the client to the server's client hashtable. In addition, the tls_session and key_state structures are alse set with the information received. Finally, the multi_process_post() function is called to finalize the client's initialization process.

**Parameters**

**buf** Buffer with info received from Master Server

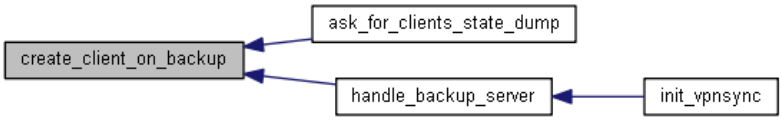
**m** Backup Server multi_context structure

**Returns**

Returns the resulting multi_instance structure with client information.

Definition at line **111** of file **vpnsync.c**.

Here is the caller graph for this function:




## ◆ handle_backup_server()

```
int handle_backup_server ( struct context *       c,
                           struct multi_context * m,
                           int                    flag
                         )
```

Backup: Handle Backup Server operations.

This function handles Backup Server operations. It opens the multicast socket and creates two child processes: one for receiving client states from the Master Server and other to check its current CARP state. The two processes communicate with the parent through a pipe. The parent is waiting (using select) for communication from the pipe and when one of the children writes to the pipe the parent decides what to do. If the communication is from **receive_clients_state()** then it means that parent has just received information to create a new client, so it calls **create_client_on_backup()**. If the communication is from **check_carp_state()**, then the Server is no longer a Backup Server because it became Master, so the function terminates its children and exits gracefully. (Unicast implementation for receiving a dump of all clients state from Master is not fully implemented yet.)

**Parameters**

**c** Top-level Server context.

**m** multi_context Server structure.

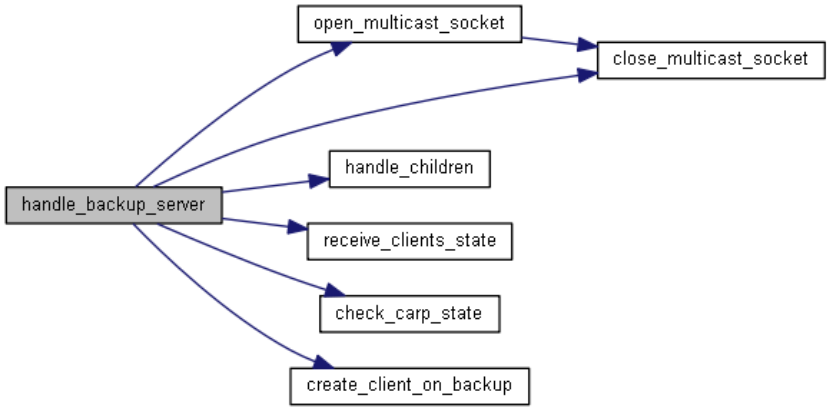
**flag** 0 to check CARP state to find if Server is really Master. 1 to ignore.

**Returns**

Returns the success (0) or failure (-1) of the operation.

Definition at line **823** of file **vpnsync.c**.

Here is the call graph for this function:




Here is the caller graph for this function:


## ◆ handle_children()

void handle_children ( int signum )

Master/Backup: Handler to terminate Child Processes (SIGCHLD).

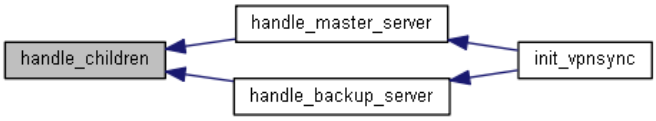
This function handles the termination of child processes. It is called by **handle_master_server()** and **handle_backup_server()**.

**Parameters**

**signum** Signal received from child process

Definition at line **595** of file **vpnsync.c**.

Here is the caller graph for this function:




## ◆ handle_master_server()

```
int handle_master_server ( struct context *        c,
                           struct multi_context *  m,
                           struct multi_instance * mi,
                           int                     action
                         )
```

Master: Handle Master Server operations.

This function handles Master Server operations. It opens the multicast socket and continues to exit gracefully when called by **init_vpnsync()**. It is also called by multi_process_post() right after multi_connection_established(), which is the last step of a client-server initialization process. This time it creates a child process and calls **send_clients_state()** to send a multicast packet to all Backup Servers with the information about the client that has just connected. Note: This funtion also calls **check_carp_state()** to ensure that only the Master Server is able to send multicast packets.

**Parameters**

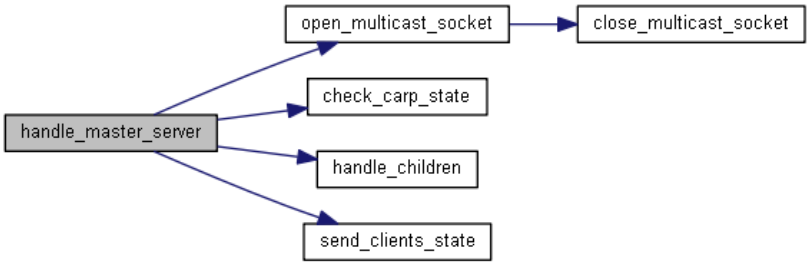
| | |
|---|---|
| **c** | Top-level Server context. |
| **m** | multi_context Server structure. |
| **client_state** | Client multi_instance structure. |
| **flag** | 0 to check if Server is really Master. 1 to ignore. |

**Returns**

Returns the success (0) or failure (-1) of the operation.

Definition at line **771** of file **vpnsync.c**.

Here is the call graph for this function:




Here is the caller graph for this function:


## ◆ init_vpnsync()

```
void init_vpnsync ( struct context *       c,
                    struct multi_context * m
                  )
```

Master/Backup: Main Function - Initialize VPNSync operations.

This function initializes the VPNSync functionality. First checks if Server is Master or Backup by calling **check_carp_state()** and afterwards calls **handle_master_server()** or **handle_backup_server()** accordingly. If Server is Backup when **handle_backup_server()** returns 0 it means that it is now Master, so the **handle_master_server()** is called. If **handle_master_server()** or handle_backup_server returns -1 the function prints error message and exits. If **handle_master_server()** returns 0 the function exits gracefully.

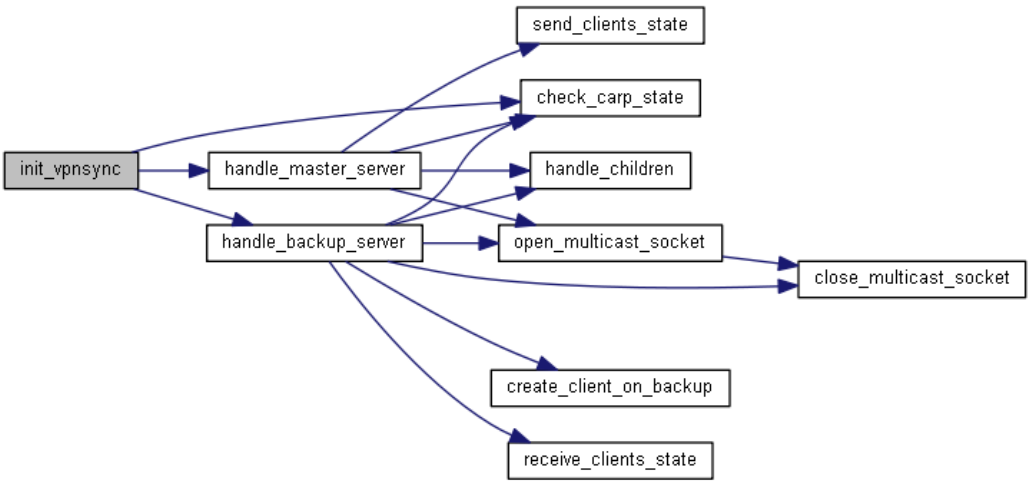
**Parameters**

**c** Top-level Server context.

**m** multi_context Server structure.

Definition at line **930** of file **vpnsync.c**.

Here is the call graph for this function:




## ◆ open_multicast_socket()

int open_multicast_socket ( int multicast_port )

Master/Backup: Open Multicast Socket for new Client State and Updates.

This function opens the multicast socket. It is called once by **handle_master_server()** and once by **handle_backup_server()**. Packets containing client state are sent by multicast from the Master Server to all Backup Servers that join the multicast group. The default configuration is address 239.0.2.0 and port 1196.

**Parameters**

**multicast_port** Multicast port (by default 1196)

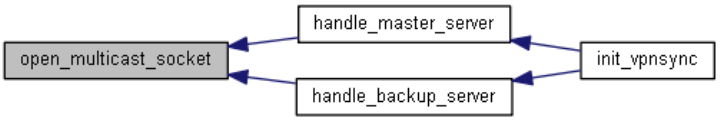
**Returns**

Returns the success (0) or failure (-1) of the operation.

Definition at line **660** of file **vpnsync.c**.

Here is the call graph for this function:


Here is the caller graph for this function:




## ◆ open_unicast_socket()

int open_unicast_socket ( int multicast_port,
int flag
)

Master/Backup: Open Unicast Socket for Clients State Dump.

This function opens the unicast socket. It is called once by **handle_master_server()** and once by **handle_backup_server()**. The unicast socket is reserved only for the event of a client state dump between the Master Server and a new Backup Server added to the cluster that asks for it. Note: Unicast dump feature is not available yet.

**Parameters**

**multicast_port** Unicast port is multicast port + 1 (by default 1197)

**flag** Flag to specify is its Master or Backup Server

**Returns**

Returns the success (0) or failure (-1) of the operation.

Definition at line **624** of file **vpnsync.c**.

Here is the call graph for this function:


## ◆ print_session_id()

```
const char* print_session_id ( const struct session_id *  sid,
                               struct gc_arena *          gc
                             )
```

Master/Backup: Print Session ID Info.

This function receives a session_id structure with a int array and returns the session id in string format. It is called by **create_client_on_backup()** and **send_clients_state()**.

**Parameters**

**sid** Session ID

**gc** Garbage collector

**Returns**

Returns the Session ID in human readable format.

Definition at line **51** of file **vpnsync.c**.

## ◆ receive_clients_state()

```
void receive_clients_state ( struct context *        c,
                             struct multi_context *  m
                           )
```

Backup: Receive a Client State from the Master Server.

This function receives from Master Server multicast packets containing client state information. It is called by **handle_backup_server()** in a new dedicated process. It waits in loop for incoming packets and forwards them through a pipe to the parent process. The child process is terminated when the Backup Server transitions to Master.

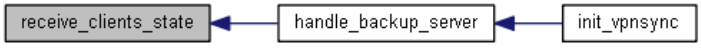
**Parameters**

**c** Top-level Server context structure.

**m** multi_context Server structure

Definition at line **491** of file **vpnsync.c**.

Here is the caller graph for this function:




## ◆ send_all_clients_state()

```
int send_all_clients_state ( struct multi_context *  m )
```

Master: Send All Clients State Dump to a specific Backup Server.

This function sends all clients state to a specific Backup Server. It is called by **wait_request_for_dump()** when a request is received from a Backup Server. For each client it calls **send_clients_state()**. Note: Unicast dump feature is not available yet.

**Parameters**

**m** multi_context Server structure

**Returns**

Returns the success (0) or failure (-1) of the operation.

Definition at line **541** of file **vpnsync.c**.

Here is the call graph for this function:


Here is the caller graph for this function:


## ◆ send_clients_state()

```
int send_clients_state ( struct multi_instance *  mi,
                         int                      socketfd,
                         struct sockaddr_in       send_socket
                       )
```

Master: Send a Client State to Backup Servers.

This function sends client state information to Backup Servers using multicast packets. It is called by **handle_master_server()** and ran in a dedicated child process. The information sent includes the address, port, common name and session id of the client. In addition, the cipher keys and hmac keys of the tunnel are also sent. Using the Binn library, the information is serialized into a Binn object and the object is sent in a multicast packet. Note: There is a wait time of 60 seconds before the information is sent to give time for the connection client-server to be fully established.

**Parameters**

**mi** multi_instance structure with client state information

**socketfd** Socket file descriptor used to send clients state

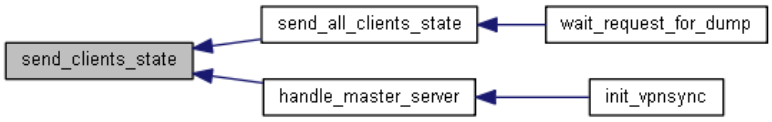
**send_socket** Socket structure used to send clients state

**Returns**

Returns the success (0) and failure (-1) of the operation.

Definition at line **325** of file **vpnsync.c**.

Here is the caller graph for this function:




## ◆ send_clients_state_update()

```
int send_clients_state_update ( struct multi_instance *  client_state,
                                int                      socketfd,
                                struct sockaddr_in       send_socket
                              )
```

Master: Send a Client State Update to Backup Servers.

This function sends updated client state information to Backup Servers using multicast packets. It is called by **handle_master_server()** and ran in a dedicated child process. The information sent includes the current packet id counter and timestamp for both received and sent packets. Using the Binn library, the information is serialized into a Binn object and the object is sent in a multicast packet.
Note: The wait time between updates is 60 seconds.

**Parameters**

| | |
|---|---|
| **mi** | multi_instance structure with client state information |
| **socketfd** | Socket file descriptor used to send clients state |
| **send_socket** | Socket structure used to send clients state |

**Returns**

Returns the success (0) and failure (-1) of the operation.

Definition at line **289** of file **vpnsync.c**.

## ◆ set_keys()

```
void set_keys ( struct key2 *             k,
                const struct key_type *   kt,
                const char *              prefix0,
                const char *              prefix1
              )
```

Master: Get Cipher and HMAC Keys and store them on global variables k2 and kt2.

This function sets the structures key2 and key_type with the keys negotiated during the 3-Way-Handshake between the client and the server. It's called by generate_key_expansion() on ssl.c. Its purpose is to extract the keys to a local structure, which enables its subsequent use in the **send_clients_state()** function.

**Parameters**

| | |
|---|---|
| **k** | key2 structure with cipher and auth keys |
| **kt** | key_type structure with cipher and auth info |
| **prefix0** | "Master Encrypt" |
| **prefix1** | "Master Decrypt" |

Definition at line **85** of file **vpnsync.c**.

## ◆ set_master_struct()

```
int set_master_struct ( const char *  master_server,
                        int           master_port
                      )
```

Backup: Tries to translate Server IP - If successful means Master Server is reachable.

This function sets the structure master_sock_struct with the Master Server IP and Port. It is called by **handle_backup_server()**. The aim is to prepare the information to communicate with the Master Server through a unicast socket to ask for a clients state dump. Note: Unicast dump feature is not available yet.

**Parameters**

| | |
|---|---|
| **master_server** | Master Server IP address |
| **master_port** | Master port for Unicast communications (by default 1197) |

**Returns**

Returns the success (0) or failure (-1) of the operation.

Definition at line **713** of file **vpnsync.c**.

### ◆ wait_request_for_dump()

void wait_request_for_dump ( struct multi_context * m )

Master: Wait for Backup Servers to ask for Clients State Dump.

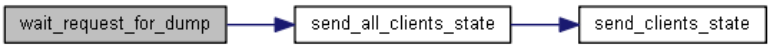
This function waits for a request from a Backup Server to receive a dump of all clients in the system. It is called by **handle_master_server()** in a new dedicated child process. Note: Unicast dump feature is not available yet.

**Parameters**

**m** multi_context Server structure

Definition at line **573** of file **vpnsync.c**.

Here is the call graph for this function:




## Variable Documentation

### ◆ backup_sock_struct

struct sockaddr_in backup_sock_struct

Backup socket structure unicast communication

Definition at line **33** of file **vpnsync.c**.

### ◆ group

struct ip_mreq group

Group structure for multicast communication

Definition at line **41** of file **vpnsync.c**.

### ◆ k2

struct key2* k2

Key structure to save cipher and hmac keys

Definition at line **45** of file **vpnsync.c**.

### ◆ kt2

struct key_type* kt2

Key-type structure to save cipher and hmac context

Definition at line 47 of file vpnsync.c.

### ◆ master_sock_struct

struct sockaddr_in master_sock_struct

Master socket structure unicast communication

Definition at line 31 of file vpnsync.c.

### ◆ multicast_sock_struct

struct sockaddr_in multicast_sock_struct

Socket structure for multicast communication

Definition at line 35 of file vpnsync.c.

### ◆ pipefd

int pipefd[2]

Pipe for comunications between children and parent processes

Definition at line 43 of file vpnsync.c.

### ◆ sockfd_multi

int sockfd_multi

Socket file descriptor for multicast communication

Definition at line 37 of file vpnsync.c.

### ◆ sockfd_uni

int sockfd_uni

Socket file descriptor for unicast communication

Definition at line 39 of file vpnsync.c.

# Bibliografia

[1] M. S. Merkow e J. Breithaupt, Information Security: Principles and Practices, Pearson Education, 2014.

[2] C. Kopparapu, Load Balancing Servers, Firewalls and Caches, 2002.

[3] "Digital Ocean," [Online]. Available: https://www.digitalocean.com/community/tutorials/what-is-high-availability. [Acedido em 6 12 2016].

[4] "IT Pro Portal," [Online]. Available: http://www.itproportal.com/2014/11/19/5-minutes-unplanned-downtime-year-really-possible/. [Acedido em 9 01 2017].

[5] "IETF - TLS 1.2 RFC," [Online]. Available: https://tools.ietf.org/html/rfc5246. [Acedido em 09 06 2017].

[6] I. Ristić, Bulletproof SSL and TLS - Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications, Feisty Duck Limited, 2015.

[7] "Moserware," [Online]. Available: http://www.moserware.com/2009/06/first-few-milliseconds-of-https.html. [Acedido em 11 11 2016].

[8] "PC World," [Online]. Available: http://www.pcworld.com/article/2030763/networking/how-and-why-to-set-up-a-vpn-today.html. [Acedido em 24 10 2016].

[9] E. F. Crist and J. J. Keijser, Mastering OpenVPN, Packt Publishing, 2015.

[10] "Best VPN Services," [Online]. Available: https://www.bestvpn.com/best-vpn-services/. [Acedido em 9 01 2017].

[11] "Best VPN," [Online]. Available: https://www.bestvpn.com/blog/29990/vpn-encryption-terms-explained-aes-vs-rsa-vs-sha-etc/. [Acedido em 18 10 2016].

[12] "iVPN," [Online]. Available: https://www.ivpn.net/pptp-vs-l2tp-vs-openvpn. [Acedido em 19 10 2016].

[13] "Microsoft Technet VPN," [Online]. Available: https://technet.microsoft.com/pt-br/library/dd469653(v=ws.11).aspx. [Acedido em 24 10 2016].

[14] “Microsoft Technet IPSec,” [Online]. Available: https://technet.microsoft.com/en-us/library/cc959523.aspx. [Acedido em 1 11 2016].

[15] “Private Internet Access,” [Online]. Available: https://bra.privateinternetaccess.com/pages/vpn-encryption. [Acedido em 29 11 2016].

[16] “Best VPN Comparison,” [Online]. Available: https://www.bestvpn.com/blog/4147/pptp-vs-l2tp-vs-openvpn-vs-sstp-vs-ikev2/. [Acedido em 9 12 2016].

[17] “Network World,” [Online]. Available: http://www.networkworld.com/article/2220464/cisco-subnet/cisco-subnet-stronger-ipsec-vpn-configurations-needed.html. [Acedido em 16 12 2016].

[18] M. Feilner, OpenVPN - Building and Integrating Virtual Private Networks, Packt Publishing, 2006.

[19] J. J. Keijser, OpenVPN 2 Cookbook, Packt Publishing, 2011.

[20] G. Attebury and B. Ramamurthy, "Router and Firewall Redundancy with OpenBSD," in *CSE Conference and Workshop Papers*, 2006.

[21] “Cisco HSRP,” [Online]. Available: http://www.cisco.com/c/en/us/support/docs/ip/hot-standby-router-protocol-hsrp/9234-hsrpguidetoc.html. [Acedido em 10 11 2016].

[22] “Pearson IT Certification,” [Online]. Available: http://www.pearsonitcertification.com/articles/article.aspx?p=2141274. [Acedido em 10 11 2016].

[23] P. Danhieux, “CARP - The Free Fail-over Protocol,” em *SANS Institute*, 2004.

[24] “Everything Linux,” [Online]. Available: http://everythinglinux.org/rsync/. [Acedido em 10 11 2016].

[25] “OpenBSD Man Pages PFsync,” [Online]. Available: http://man.openbsd.org/pfsync. [Acedido em 1 12 2016].

[26] “Linux-HA,” [Online]. Available: http://www.linux-ha.org/wiki/Site_news. [Acedido em 2 12 2016].

[27] “DRBD,” [Online]. Available: https://www.drbd.org/en/. [Acedido em 2 12 2016].

[28] “ClusterLabs,” [Online]. Available: http://clusterlabs.org/. [Acedido em 1 12 2016].

[29] “Cisco IPsec,” [Online]. Available: http://www.cisco.com/c/en/us/td/docs/ios/12_2/12_2y/12_2yx11/feature/guide/ft_vpnha.html. [Acedido em 3 01 2017].

[30] “Cisco AnyConnect,” [Online]. Available: http://www.cisco.com/c/en/us/support/docs/security/anyconnect-secure-mobility-client/116312-qanda-anyconnect-00.html#anc4. [Acedido em 4 01 2017].

[31] “OpenBSD Man Pages dhcpd,” [Online]. Available: http://man.openbsd.org/dhcpd.8. [Acedido em 6 12 2016].

[32] M. W. Lucas, Absolute OpenBSD, No Starch Press, 2013.

[33] “Secpoint,” [Online]. Available: https://www.secpoint.com/top-10-most-secure-operating-systems.html. [Acedido em 31 10 2016].

[34] “ostif,” 11 05 2017. [Online]. Available: https://ostif.org/the-openvpn-2-4-0-audit-by-ostif-and-quarkslab-results/. [Acedido em 22 05 2017].

[35] “privateinternetaccess,” 11 05 2017. [Online]. Available: https://www.privateinternetaccess.com/blog/2017/05/openvpn-2-4-evaluation-summary-report/. [Acedido em 22 05 2017].

[36] “SCRUM Alliance,” [Online]. Available: https://www.scrumalliance.org/why-scrum. [Acedido em 16 01 2017].

[37] “OpenVPN Management Feature,” [Online]. Available: https://openvpn.net/index.php/open-source/documentation/miscellaneous/79-management-interface.html. [Acedido em 13 02 2017].

[38] “Doxygen,” [Online]. Available: http://www.stack.nl/~dimitri/doxygen/. [Acedido em 25 05 2017].

[39] “OpenVPN Doxygen Documentation,” [Online]. Available: https://build.openvpn.net/doxygen/html/index.html. [Acedido em 26 09 2016].

[40] “OpenVPN Architecture,” [Online]. Available: https://community.openvpn.net/openvpn/wiki/RoadMap. [Acedido em 13 02 2017].

[41] “Binn Library,” [Online]. Available: https://github.com/liteserver/binn. [Acedido em 20 03 2017].