University of Coimbra

Faculty of **S**ciences and **T**echnology

Department of **I**nformatics **E**ngineering

Operating Middleware and Timing Guarantees for Heterogeneous Sensor Networks

José Manuel da Silva Cecílio

Coimbra, 2013

Department of Informatics Engineering

Operating Middleware and Timing Guarantees for Heterogeneous Sensor Networks

by

José Manuel da Silva Cecílio

A dissertation submitted to the UNIVERSITY OF COIMBRA

in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Informatics Engineering

Adviser:

Prof. Dr. Pedro Nuno San-Bento Furtado

Assistant Professor at University of Coimbra

Coimbra, 2013

Financial support by:

1- EU Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 224282, Project **GINSENG**.

2 - Portuguese Foundation for Science and Technology (Fundação para a Ciência e a Tecnologia – co-financiamento do Programa Operacional Potencial Humano/POPH e da União Europeia) through the PhD grant **SFRH / BD / 71008 / 2010.**

Operating Middleware and Timing Guarantees for Heterogeneous Sensor Networks

ISBN 978-989-96001-7-1

Abstract

In many distributed contexts, the software infrastructure needs to incorporate data coming from nodes that include computers, wireless computational devices and wireless sensor networks (WSNs). The inclusion of pervasive devices in distributed systems provides flexibility and cost savings when compared to entirely cabled deployments. For example, in real industrial setups there will typically coexist wired sensors, wireless sensor and actuator networks (WSAN) and wired backbone nodes, forming a heterogeneous programmable distributed system.

Existing wireless embedded systems for distributed applications are programmed separately from the rest of the network. In this thesis we propose a modular approach to hide heterogeneity and offer a single common configuration and processing component for all nodes of the heterogeneous system. The main contribution consists on a middleware architecture that configures and processes data uniformly over heterogeneous networks. A set of mechanisms are proposed, with a single uniform component to work with heterogeneous distributed systems. This advances the current state-of-the-art in middleware for distributed control systems, by providing a single component that abstracts the underlying differences in all devices such as computers and WSN nodes, using a data stream processing model.

Timing related issues must be brought forward when designing such a middleware architecture for heterogeneous distributed control systems. In this line, the thesis also investigates and proposes an approach for planning operations with timing guarantees.

Timing guarantees in WSN sub-networks are enforced using real-time algorithms and protocols. In what concerns network protocols, pre-planned synchronous time-division algorithms are frequently used to enforce timing. But at the same time, operations timing requirements must be met over the whole heterogeneous system, regardless of what protocols and software is running in each part. We discuss how to plan monitoring and closed-loop operations with restricted time boundaries in the distributed heterogeneous system.

The mechanisms and approaches proposed in the thesis were successfully applied in an embodiment of the concept, as middleware component in an industrial refinery setting within EU project GINSENG, and all components were evaluated.

Resumo Alargado em Português

Em muitos contextos distribuídos, a infra-estrutura de software necessita incorporar dados provenientes de nós que podem ser computadores ou sistemas embebidos sem fios. A inclusão de dispositivos embebidos sem fios em sistemas distribuídos oferece flexibilidade e economia de custos quando comparado com implementações totalmente cabladas. Por exemplo, numa instalação industrial real podem existir sensores com fios, sensores sem fios e computadores cablados, formando um sistema programável heterogéneo distribuído.

As soluções actuais de desenvolvimento de sistemas distribuídos com sistemas embebidos são concebidas em separado, onde é necessário programar cada parte com código específico. No entanto, esta abordagem acarreta problemas no contexto de aplicações interactivas (por exemplo, (re)configuração de controlo de malha fechada, em qualquer parte da rede), em que a rede deve ser vista como um sistema heterogéneo único distribuído, oferecendo uma maior uniformidade, simplicidade e flexibilidade.

Nesta tese é proposta uma abordagem modular para desenhar esses sistemas distribuídos com dispositivos heterogéneos. Propõe-se um conjunto de mecanismos e uma arquitectura de middleware capaz de lidar com as diferenças de hardware e software provenientes das características dos nós. Ao mesmo tempo é requerido um interface único de configuração para todos os nós (computadores ou sistemas embebidos). Isso avança o estado-da-arte em abordagens de middleware para sistemas distribuídos com dispositivos embebidos, porque oferece um único componente de middleware que abstrai as diferenças subjacentes aos dispositivos e permite configura-los da mesma forma, utilizando um modelo de fluxo de dados e processamento.

Outra questão que surge relativamente à concepção de um sistema distribuído com dispositivos embebidos prende-se com o desempenho resultante de todo sistema. Surge então a questão se é possível oferecer garantias de tempos de operação dentro de toda a rede.

Nesta tese também é proposta uma abordagem para planeamento de garantias de tempo sobre o sistema heterogéneo distribuído compreendendo todas as partes.

Os mecanismos e abordagens propostas nesta tese foram aplicados com sucesso no conceito de componente de middleware no âmbito do projecto Europeu GINSENG que tinha como cenário de aplicação parte da rede de sensores existente na refinaria da Petrogal em Sines, Portugal.

Dado que a tese foi escrita em Inglês, na restante parte desta Secção faz-se um resumo em Português do conteúdo da tese.

1. Introdução

No capítulo 1 introduzimos os conceitos da tese, incluindo a estratégia utilizada para resolver questões de heterogeneidade resultantes da introdução de dispositivos embebidos sem fios em sistemas distribuídos computacionais de monitorização e controlo.

Existem duas soluções actuais para a implantação dos sistemas heterogéneos que incluem redes de sensores sem fios e o resto do mundo. Uma delas envolve a programação de todos os detalhes de processamento e comunicação à mão, dentro da rede de sensores sem fios e fora dela. O outro é baseado em abordagens de middleware. No entanto, as soluções de middleware existentes servem apenas uma parte do sistema distribuído. Por exemplo, há abordagens para redes de sensores sem fios e há outras soluções de middleware para sistemas distribuídos baseados em computadores, mas nenhuma deles é capaz de configurar ambas as partes de forma única. Dependendo do middleware, ou a rede de sensores sem fios ou a parte baseada em computador tem que ser programada separadamente.

A proposta desta tese gira em torno de uma arquitectura de middleware que configura e processa dados de maneira uniforme em sistemas heterogéneos distribuídos. O sistema distribuído pode ser visto como uma rede de processamento uniforme onde existe um componente único de nó, que pode ser instalado em qualquer nó, incluindo nós fora da rede de sensores sem fios. Cada nó terá o mesmo interface de configuração remota e as mesmas capacidades de processamento, sem qualquer programação adicional.

Mais genericamente, num sistema heterogéneo, com diferentes tipos de dispositivos sensores, estações de controlo e sub-redes, a abordagem oferece homogeneidade fácil e imediata sobre toda a infra-estrutura heterogénea.

O problema desta tese pode ser descrito como:

Como oferecer interoperabilidade num sistema heterogéneo distribuído com configuração igual e um modelo de processamento de dados igual em todos os nós (computadores ou sistemas embebidos), onde as mesmas operações (por exemplo, condições complexas de malha fechada ou alarme) podem ser executadas sem qualquer programação personalizada?

O objectivo principal deste trabalho é então responder à pergunta acima formulada.

Nenhuma das soluções actuais para sistemas distribuídos com redes de sensores sem fios aborda a reconfiguração sobre todo o sistema. Além disso, o estado-da-arte em reconfiguração de redes de sensores sem fios é focado em middleware para plataformas específicas, bem como em questões de baixo nível de reconfiguração, e carece de um modelo de reconfiguração a nível de recolha e processamento de dados.

Em cenários de aplicação como os sistemas distribuídos de controlo, utilizados em aplicações industriais, os tempos de operação tornam-se muito importantes. Surge então a questão, se é possível oferecer garantias temporais no sistema heterogéneo distribuído. Uma maneira de oferecer garantias de tempo é planear e implantar redes de sensores sem fios com protocolos de execução pré-planeados. Neste contexto, a proposta de middleware deve prever e controlar o tempo de execução das operações.

Esta tese propõe então uma arquitectura de middleware para lidar com a heterogeneidade e fornecer garantias de tempo em sistemas distribuídos com dispositivos heterogéneos. As contribuições da tese resumem-se a:

- Investigar metodologias de middleware e métodos para a construção de sistemas com base em não-codificação para (re)configuração remota e operação em ambientes heterogéneos.
- Investigar metodologias de planeamento para garantias de tempo em sistemas heterogéneos distribuídos.
- Avaliar a adequação dos mecanismos propostos na forma de middleware e de planeamento em suporte a aplicações industriais, utilizando a plataforma de testes do projecto Europeu GINSENG como um caso de estudo.

2. Background e Estado da Arte em Middleware para Redes de Sensores Sem Fios

Para abordar as questões da tese, precisamos investigar middleware e métodos para a construção de sistemas heterogéneos distribuídos com base em não-codificação e (re)configuração remota. O capítulo 2 aborda o estado-da-arte relacionado com middlewares. Primeiro são descritas características típicas de sistemas embebidos (software e hardware) que ajudam a entender o problema de heterogeneidade que pode ocorrer quando criamos um sistema distribuído com redes de sensores sem fios.

Na secção 2.2 deste capítulo, revemos o trabalho mais importante que está relacionado com o problema abordado nesta tese. São discutidas duas áreas principais: reconfiguração remota e arquitecturas de middleware.

3. Background e Estado da Arte em Planeamento de Redes com Garantias de Tempo

O capítulo 3 aborda o estado-da-arte relacionado com o planeamento de redes sem fios com garantias de tempo. Numa primeira parte, servindo de background é fornecida alguma informação sobre o controle de acesso ao meio (MAC), abordagens de protocolo de comunicação e mecanismos de escalonamento utilizados em protocolos de múltiplo acesso com divisão de tempo (protocolos TDMA).

Os mecanismos de planeamento utilizados para planear sistemas distribuídos de controlo com garantias de tempo são analisados de seguida, na seção 3.3. A discussão é focada em programação temporal (scheduling) e dimensionamento da rede, e em modelos de latência para operações com garantias temporais.

4. Definição de Requisitos para Redes de Sensores Heterogéneas que incluem Sensores Sem Fios

O capítulo 4 analisa os requisitos de cenários de aplicação para sistemas baseados em não-codificação, com configuração remota e operação em ambientes heterogéneos.

O capítulo analisa alguns cenários de aplicação e em seguida explora os requisitos de middleware que podem ser extraídos a partir deste conjunto de cenários de aplicação. Para a concepção do middleware proposto nesta tese, tiveram-se em consideração os seguintes requisitos:

- Aquisição de dados e processamento
- Heterogeneidade
- Interoperabilidade
- Flexibilidade
- Configuração do sistema e Adaptabilidade
- Utilizadores
- Desempenho

Todos esses requisitos foram considerados na arquitectura geral proposta, o que a torna capaz de suportar diferentes cenários de aplicação.

5. Mecanismos de Middleware para Redes de Sensores Heterogéneas que incluem Sensores Sem Fios

No capítulo 5 descrevem-se os mecanismos do middleware que permitem acolher os requisitos levantados no capítulo 4. Este capítulo começa por definir a arquitectura geral do sistema distribuído com sistemas embebidos, e depois descreve as metodologias e mecanismos utilizados para garantir a independência da plataforma (hardware e software) e do protocolo de comunicação, bem como os mecanismos para endereçamento e referenciação de cada nó no sistema heterogéneo distribuído. Para além destes mecanismos, são descritos os mecanismos utilizados para o modelo de dados e seu processamento.

6. Arquitectura dos Componentes de Nó e de Configuração Remota

O capítulo 6 propõe uma abordagem para arquitectura do componente nó (MidSN-NC) e do componente de configuração remota (MidSN-RConfig).

O componente de nó (MidSN-NC) oferece a capacidade de processamento (configurável) para sistemas heterogéneos distribuídos com sistemas embebidos com restrições de capacidade, bem como para outros dispositivos computacionais. A arquitectura proposta constrói uma camada intermediária de computação que irá servir como uma abstracção escondendo o hardware, os sistemas operativos e os protocolos de rede.

O MidSN-NC corre a nível da aplicação e é composto por duas partes principais:

1. Um kernel (NC-kernel) que é responsável pela troca de mensagens com qualquer outro nó do sistema e pelo gerenciamento de agentes. Um agente é um código específico desenvolvido pelos utilizadores e que serve para executar novas funcionalidades no nó;

2. Uma pequena máquina operacional (NC-GinApp) que fornece:

- Gestão de configuração – capacidade de um nó se configurar com base em comandos fornecidos por outros nós ou servidores;
- Gestão de dados e capacidades de processamento – capacidade de um nó gerir dados vindos dos sensores ou de outros nós e processa-los para tomar decisões ou para encaminha-los para outros nós;
- Aquisição e capacidades de actuação – capacidade de adquirir periodicamente valores dos sensores e processar comandos de actuação;

Este componente do nó (MidSN-NC) deve ser desenvolvido apenas uma vez para cada sistema operativo.

Neste capítulo é também introduzido e descrito o componente de (re)configuração remota (MidSN-RConfig).

O MidSN-RConfig é construído como um conjunto de módulos que permitem lidar com configurações provenientes dos utilizadores através de chamadas de API e os traduz em comandos. Os comandos são então enviados como mensagens a qualquer nó de destino.

O MIDSN-RConfig é composto por quatro módulos principais e um catálogo:

- O interface de programação (API) – fornece funções para enviar comandos de configuração para aplicações externas;
- O módulo de configuração – é responsável por tratar as chamadas da API e configurar os nós na rede;
- O adaptador de rede – faz o interface entre o componente MidSN-RConfig e a infra-estrutura de comunicação em rede;
- O módulo Plug&Play – configura novos nós para operar na rede;
- O catálogo – serve para guardar endereços, configurações e informações sobre os nós.

7. Planeamento da Rede e Operações com garantias de Tempo

O capítulo 7 propõe uma abordagem para planeamento de garantias de tempo em sistemas heterogéneos distribuídos compreendendo sistemas embebidos. É discutido como planear operações de monitorização e controlo em malha fechada com limites de tempo.

Neste capítulo é descrita uma organização típica do sistema de controlo distribuído, são descritas operações e requisitos que podem ser definidos sobre esses sistemas. É proposto também um modelo de latência utilizado para planear e estimar a latência de operações. A latência global é modelada por partes e cada parte é descrita, o que permite entender que latências estão envolvidas. Algumas dessas latências são introduzidas pelas redes de sensores sem fios com recursos limitados, enquanto outras são introduzidas por redes formadas por computadores e servidores.

Depois de compreendidas as latências associadas a cada parte, é discutido um algoritmo de planeamento da rede que tem em consideração requisitos de tempo fornecidos pelos utilizadores. A abordagem é baseada no planeamento de intervalos de tempo para cada nó, onde estes são capazes de enviar os seus dados para os nós de destino. A parte da rede cablada, i.e., a rede formada por computadores e servidores, também é modelada através de estatísticas temporais recolhidas através de experimentação. A abordagem permite dimensionar a rede para atender os requisitos temporais. Ela permite estimar a latência de monitorização e a latência de comandos (configuração ou actuação).

Neste capítulo propõe-se igualmente uma abordagem para reduzir a latência de comandos de configuração ou actuação. Esta abordagem consiste na determinação do número de slots de transmissão de comandos necessários para garantir um determinado requisito de tempo.

8. Definição de Limites Temporais e sua Verificação

No capítulo 8 propõe-se a definição e verificação dos limites de tempo expectáveis para avaliar garantias de desempenho do sistema. Estes limites de tempo podem ser definidos e utilizados em qualquer sistema distribuído para verificação do seu desempenho.

Neste capítulo são definidas medidas e métricas que permitem a criação de um relatório de desempenho do sistema distribuído, para ajudar os utilizadores a ajustarem o funcionamento daquele. Estas medidas e métricas são aplicadas a cada mensagem de forma a classificá-las, e são complementadas com as respectivas estatísticas. Cada mensagem será classificada de acordo com cada limite pré-definido como “in-time”, “out-of-time”, “waiting for” ou “loss”.

Por fim são descritos os mecanismos de verificação e um monitor que permitem aos utilizadores avaliar o desempenho das diversas partes do sistema distribuído e detectar qualquer anomalia existente no mesmo.

9. Avaliação Experimental do MidSN e seus Mecanismos

O capítulo 9 apresenta os resultados da avaliação experimental da abordagem MidSN e seus mecanismos, propostos nesta tese. O objectivo é mostrar que a proposta desta tese resulta num middleware capaz de ser executado em sistemas embebidos e em computadores ao mesmo tempo que fornece garantias temporais. Para salientar a capacidade de resolução do problema de heterogeneidade, mostra-se também a avaliação da sua execução em diferentes plataformas de hardware. Neste capítulo são avaliadas características tais como: memória, tempo de execução, consumo de energia e reconfiguração.

10. Avaliação Experimental da Estratégia de Planeamento de Operações e sua Monitorização

O capítulo 10 apresenta os resultados da avaliação experimental da abordagem de planeamento e verificação de tempo de operação, de forma a fornecer garantias de tempo. São reportados resultados sobre o algoritmo de planeamento descrito no capítulo 7, nos quais comprovamos experimentalmente que as estimativas temporais resultantes do planeamento do sistema correspondem aos tempos verificados.

Por fim, é criado um ambiente de simulação onde vamos introduzir alguns atrasos aleatórios nas mensagens para demonstrar como funcionam os mecanismos de avaliação dos requisitos temporais pré-definidos e a usabilidade da ferramenta de análise de desempenho.

11. Conclusões e Trabalho Futuro

O capítulo 11 apresenta um resumo das principais contribuições desta tese, e aponta algumas questões interessantes, em aberto, que requerem investigação adicional.

Acknowledgement

To begin and foremost I want to thank my advisor professor Pedro. He has taught me, both consciously and unconsciously, how good experimental research is done. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. productive and stimulating experience. I am also thankful for the excellent example he has provided as a successful researcher and professor.

My thanks additionally go to the many friends I have made during the GINSENG project, who have provided me with inspiration and assistance throughout my studies. Whilst each of my friends have made an impact in their own special way, I would like to thank Ricardo and José do Ó for their companionship and helpful contributions in the work done at the refinery.

Next I would like to thanks to my colleagues (João Pedro Costa and Pedro Martins) for their patience during weekly meetings. I am grateful to my advisor who kept us organized and was always ready to help.

I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was funded by the GINSENG project fellowship for my first year and was funded by the Portuguese Foundation for Science and Technology (SFRH/ BD/71008/2010) in subsequent years.

Lastly, I would like to thank my family for all their love and encouragement. For my parents, who supported me in all my pursuits. And to my girlfriend Ana who makes my life worth living. Her love and support has never faulted and I will be eternally in her debt for the sacrifices and compromises she has made during my Ph.D. study.

Thank you all.

Publications

José Cecílio, Pedro Furtado, "Wireless Sensors in Industrial Time-Critical Environments", a book of Computer Communications and Networks Series, Springer-Verlag, London, 2013.

José Cecílio, Pedro Furtado, "Architecture for Uniform (Re)Configuration and Processing over Embedded Sensor and Actuator Networks", in the journal of IEEE Transactions on Industrial Informatics, 2013.

Pedro Furtado, José Cecílio, "Configuration and Operation of Networked Control Systems over Heterogeneous WSANs", in the journal of ACM Transactions on Embedded Computing Systems, 2013.

Tony O'Donovan, James Brown, Felix Busching, Alberto Cardoso, José Cecílio, José do Ó, Pedro Furtado, Paulo Gil, Anja Jugel, Wolf-Bastian Pottner, Utz Roedig, Jorge Sá Silva, Ricardo Silva, Cormarc J. Sreenan, Vasos Vassiliou, Thiemo Voigt, Lars Wolf, Zinon Zinonos, "The GINSENG System for Wireless Monitoring and Control: Design and Deployment Experiences", in the journal of ACM Transactions on Sensor Networks, 2013.

José Cecílio, Pedro Furtado, "MidSN – A middleware for Uniform Configuration and Processing over Heterogeneous Sensor and Actuator Networks", in proceedings of the 12th International Conference on Ad Hoc Networks and Wireless (ADHOC-NOW 2013), Wrocław, Poland, July 8 – 10, 2013.

José Cecílio, Pedro Furtado, "Evaluating and Bounding Operations Performance in Heterogeneous Sensor and Actuator Networks with Wireless Components", in proceedings of the 12th International Conference on Ad Hoc Networks and Wireless (ADHOC-NOW 2013), Wrocław, Poland, July 8 – 10, 2013.

José Cecílio, Pedro Furtado, "Providing Timely Actuation Guarantees with heterogeneous SAN for Industrial Control Process", in proceedings of the 12th International Conference on Control, Automation, Robotics and Vision (ICARCV 2012), Guangzhou, China, December 5 – 7, 2012.

José Cecílio, João Costa, Pedro Martins and Pedro Furtado, "A Modular Architecture for Reconfigurable Heterogeneous Networks with Embedded Devices", in proceedings of the 4th International Conference on Ad Hoc Networks, Paris, France, October 16 – 17, 2012.

José Cecílio, Pedro Furtado, "Distributed Configuration and Processing for Industrial Sensor Networks", in proceedings of the sixth international workshop on Middleware Tools, Services and Run-time Support for Networked Embedded Systems (MidSens'11), Lisbon, Portugal, 12-16 December 2011.

W-B. Pottner, L. Wolf, J. Cecílio, P. Furtado, R. Silva, J. Sa Silva, A. Santos, P. Gil, A. Cardoso, Z. Zinonos, J. do Ó, B. McCarthy, J. Brown, U. Roedig, T. O'Donovan, C. J. Sreenan, Z. He, T. Voigt, A. Kleiny, "WSN Evaluation in Industrial Environments - First results and lessons learned", in proceedings of 3rd International Workshop on Performance Control in Wireless Sensor Networks (PWSN 2011), Barcelona, Spain, 29 June 2011.

José Cecílio, Filipe Monteiro, Pedro Furtado, "Configuration and Data Processing over a Heterogeneous Wireless Sensor Networks", in proceedings of 3rd International Workshop on Performance Control in Wireless Sensor Networks (PWSN 2011), Barcelona, Spain, 29 June 2011.

José Cecílio, João Costa, Pedro Martins, Pedro Furtado, "A Framework to (re)Configure a Heterogeneous Distributed Sensor Network for Closed-loop Control", in proceedings of 3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2011), Chicago, United States, 11 April 2011.

Other Publications

José Cecílio, Pedro Furtado, "A State-Machine Model for Reliability Eliciting over Wireless Sensor and Actuator Networks", in proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies, Niagara Falls, Ontario, Canada, August 27 – 29, 2012.

José Cecílio, Pedro Martins, João Costa, Pedro Furtado, "State machine model-based Middleware for Control and Processing in industrial Wireless Sensor and Actuator Networks", in proceedings of 10th IEEE International Conference on Industrial Informatics, Beijing, China, July 25 – 27, 2012.

José Cecílio, Pedro Martins, João Costa, Pedro Furtado, "A Configurable Middleware for Processing in heterogeneous industrial Intelligent Sensors", in proceedings of 16th IEEE International Conference on Intelligent Engineering Systems, Lisbon, Portugal, June 13 – 15, 2012.

José Cecílio, Pedro Furtado, "Network Planning Tool with Traffic-Adaptive Processing for Wireless Sensor Networks", in proceedings of International Conference on Sensor Networks (Sensornets 2012), Rome, Italy, 24 - 26 February 2012.

José Cecílio, Pedro Furtado, "Reconfigurable middleware for heterogeneous embedded devices", in proceedings of 12th International Middleware Conference (Middleware’11), Lisbon, Portugal, 12-16 December 2011.

José Cecílio, João Costa, Pedro Martins, Pedro Furtado, "Device-Independent Middleware for Industrial Wireless Sensor Networks", in proceedings of the 9th International Symposium on Parallel and Distributed Processing with Applications (ISPA 2011), Busan, Korea, 16-28 May 2011.

José Cecílio, João Costa, Pedro Martins, Pedro Furtado, "Sampling Rate and Data Quality Issues: Experiments from Ginseng Industrial Deployment", in proceedings of the International Workshop on Smart Grid and Home (SGH 2011), Busan, Korea, 16-28 May 2011.

José Cecílio, João Costa, Pedro Martins, Pedro Furtado, "Interoperability for Data Processing in Distributed Sensor Networks", in proceedings of International Conference on Information Processing in Sensor Networks (IPSN), Chicago, United States, 12-14 April 2011.

José Cecílio, João Costa, Pedro Martins, Pedro Furtado, "Configuration Interface for Industrial Wireless Sensor Networks", in proceedings of International Workshop on

Advanced Information Networking and Applications (WAINA-2011), Biopolis, Singapore, 22-25 March 2011.

José Cecílio, João Costa, Pedro Martins, Pedro Furtado, "Providing Alarm Delivery Guarantees in High-Rate Industrial Wireless Sensor Network Deployments", in proceedings of International Conference on Pervasive and Embedded Systems (PECCS 2011), Vilamoura, Portugal, 5-7 March 2011.

José Cecílio, João Costa, Pedro Martins, Pedro Furtado, "GinConf: A configuration and execution interface for Wireless Sensor Network in industrial context", in Proceedings of RealWSN 2010, Colombo, Sri-Lanka, 14-16 December 2010.

José Cecílio, João Costa, Pedro Furtado, "Survey on Data Routing in Wireless Sensor Networks", a full chapter of the book "Wireless Sensor Network Technologies for Information Explosion Era" in Springer book series "Studies in Computational Intelligence", 2010.

Table of Contents

List of Acronyms

ACK	Acknowledgment
ADCs	Analog-to-Digital Converters
API	Application Programming Interface
CoAP	Constrained Application Protocol
CPLD	Complex Programmable Logic
CPU	Central Processing Unit
CSMA	Carrier Sense Multiple Access
DACs	Digital-to-Analog Converters
DCS	Distributed Control System
ECA	Event-Condition-Action
ELF	Executable and Linkable Format
FDMA	Frequency Division Multiple Access
FP7	The Seventh Framework Programme
FPGA	Field Programmable Gate Array
GSM	Global System for Mobile
HTTP	HyperText Transfer Protocol
I/O	Input/Output
ID	Identification
IP	Internet Protocol
IPv6	Internet Protocol version 6
LAN	Local Area Network
LDR	Light Dependent Resistor
LPL	Low-Power Listening
MAC	Medium Access Control
MCU	Microcontroller Unit
OS	Operating System
PC	Personal computer
PID	Proportional–Integral–Derivative controller
PLC	Programmable Logic Controller
PLCs	Programmable Logic Controllers
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RFID	Radio-frequency identification
RLE	Run-length Encoding

ROM	Read Only Memory
SCADA	Supervisory Control and Data Acquisition
SOAP	Simple Object Access Protocol
SPI	Serial Peripheral Interface
SQL	Structured Query Language
SYNC	Synchronisation
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
uIP	Micro IP
WPx	Work package x
WSAN	Wireless Sensor and Actuator Network
WSN	Wireless Sensor Network
WSNs	Wireless Sensor Networks
XML	Extensible Markup Language

List of Tables

List of Figures

Chapter 1

Introduction

Many industrial premises, such as a refinery, have hundreds or thousands of sensors and actuators, which automate monitoring and control functionalities. About a decade ago, industry suppliers have started deploying wireless sensor and actuation solutions, which are easier to deploy and less costly than totally cabled ones. These solutions are based on small embedded devices, with sensing and actuation functionalities, as well as communication and computation capabilities. They revolutionize critical applications by allowing sensing and actuation at significantly lower cost.

Typically, multiple wireless sensor devices will be organized into some kind of Wireless Sensor Network (WSN) spanning a whole sensing region. That distributed system is different from a conventional distributed system built with computer nodes in many aspects. Resource scarceness is the primary concern, which should be carefully taken into account when designing software for those networks. Sensor nodes are often equipped with a limited energy source and a processing unit with a small memory capacity. Additionally, the network bandwidth is much lower than for wired communications, and radio operations are very expensive compared to pure computation in terms of battery consumption.

The enterprise software infrastructure in those industries needs to connect to the Distributed Control System (DCS) (the monitoring and control network). The DCS includes many heterogeneous devices, such as Programmable Logic Controllers (PLCs), computers and wireless embedded devices. The inclusion of WSN in industrial control and monitoring applications contributes to the heterogeneity of the distributed system.

Heterogeneous sensor networks are also useful in other application contexts such as: environmental monitoring, precise agriculture monitoring and control, warehouse tracking, transport logistics, surveillance and health case.

1.1. Definitions

Before discussing the problem statement and proposal of this thesis, we introduce some terms used to help the reader understand the thesis proposal. This thesis revolves on sensor and actuator networks. These networks can be built with one single platform (hardware and software) or can include different types of platforms. So, they can be homogeneous or heterogeneous (Figure 1.1).

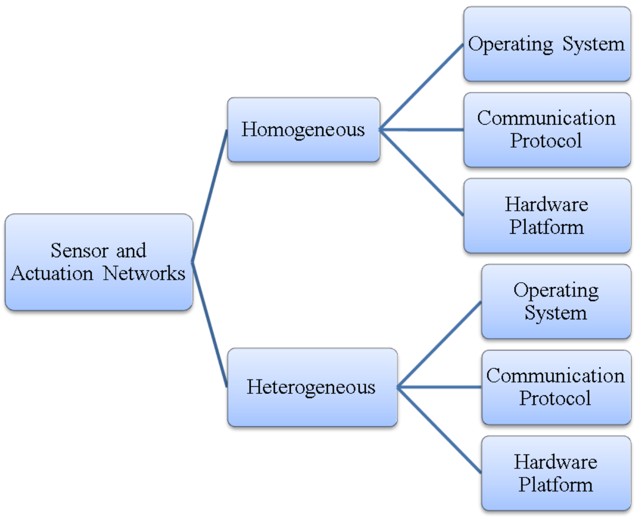


Figure 1.1 – Types of networks

In this thesis we will deal with heterogeneous sensor and actuator networks, referred as heterogeneous sensor networks. The heterogeneous concept that we deal in this thesis is related with operating systems, communication protocols and hardware diversity.

These networks can be found in different application scenarios. They can be found in distributed control systems (DCS), typically used in industrial monitoring and control systems to manage industrial processes, or in other application contexts. In any of these application scenarios, the network may include different classes of hardware and software. It may include embedded devices (constrained devices) or computer nodes (Figure 1.2), making a heterogeneous network.

In addition, the nodes can be connected using cable infrastructures or wireless links. When wireless links are assumed, we can have pervasive or mote devices. These devices are resources constrained, with computation, communication and programming capabilities.

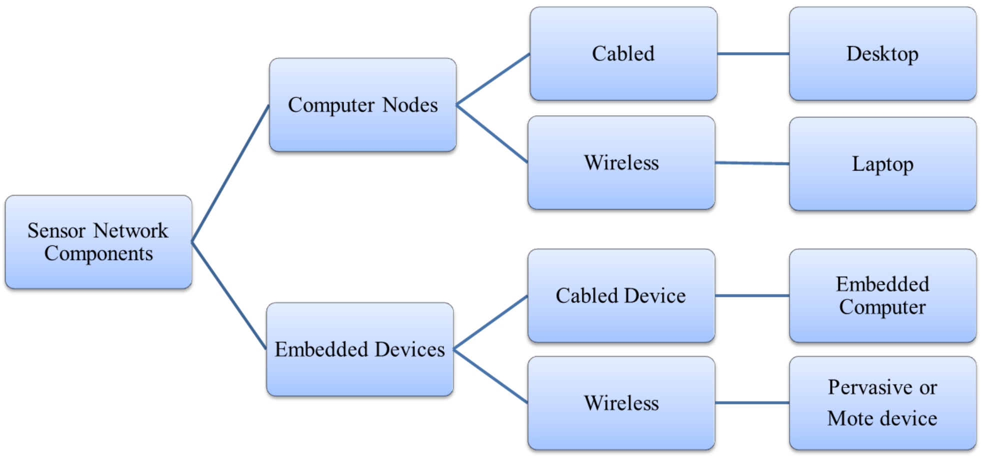


Figure 1.2 – Sensor network components

The heterogeneous sensor network organization assumed in this thesis may include cabled IP parts to computers and other devices, cabled sensors that provide analogue signals, wireless sensors and communication links, and wireless sensor networks composed by mote devices and specific communication and routing protocols. Figure 1.3 shows various components that may be included in a heterogeneous sensor network.

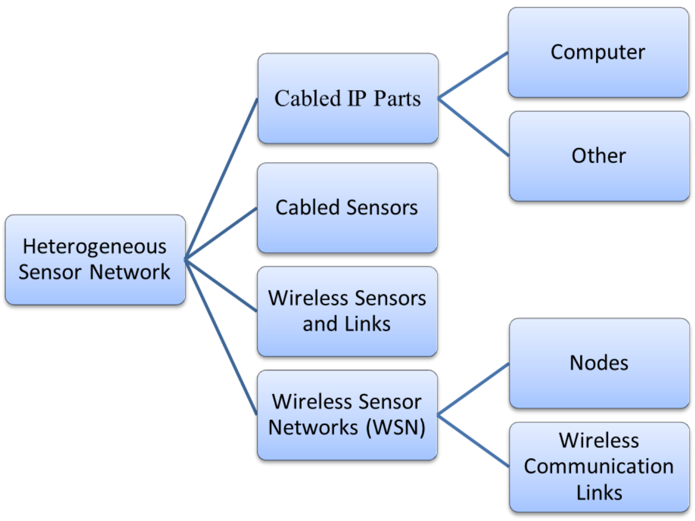

Heterogeneous Sensor Network
Cabled IP Parts
Cabled Sensors
Wireless Sensors and Links
Wireless Sensor Networks (WSN)
Computer
Other
Nodes
Wireless Communication Links

Figure 1.3 – Components of a heterogeneous sensor networks

Lastly and to conclude this introductory section, we assume that pervasive devices which operate a non-IP communication protocol (e.g. IEEE 812.15.4, ZigBee, Rime), are organized in sub-networks (WSNs), and each WSN is headed by a gateway. This gateway does the interface between non-IP protocols and IP protocols.

Computers, embedded or pervasive devices that support IP protocols may be connected directly to the heterogeneous sensor network or may be organized in sub-networks.

Figure 1.4 shows an example of a heterogeneous sensor network. In this example we have two sub-networks composed by pervasive devices running, for instance, the ZigBee communication protocol. They are connected to the distributed system through a gateway. Each gateway interfaces the sub-network with the rest of the distributed system that runs IP protocol.

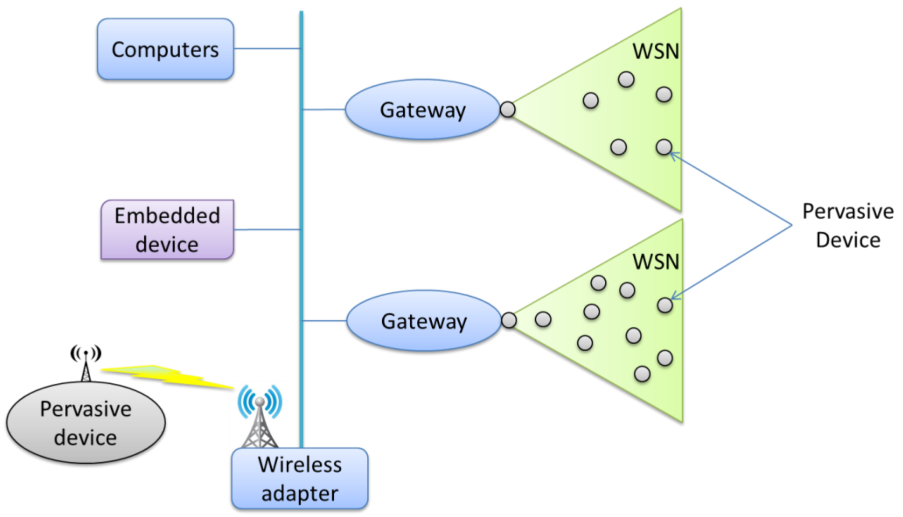


Figure 1.4 – Heterogeneous sensor network

1.2. *Problem Statement and Thesis Proposal*

There are two main current solutions for deploying the heterogeneous systems that include WSN nodes and the rest of the world. One involves programming every detail of processing and communication by hand, both within the WSN and outside of it. The other one is based on middleware. However, existing middleware solutions cover only part of the distributed system. For instance, there are several middleware approaches for WSN operating systems and there are other middleware solutions for computer-based distributed systems, but none is able to configure both parts in a unified manner. Depending on the middleware, either the WSN or the computer-based part of the distributed system has to be coded separately.

This thesis proposal revolves around a middleware architecture that allows a unified configuration and processes data uniformly over heterogeneous networks, composed by computers, constrained embedded devices and wireless sensor sub-network nodes. A distributed system can be seen as a uniform configuration and processing network by considering a single node component that can be installed in any node, including nodes outside of the WSN. Each node will have at least a uniform configuration interface (API), remote configuration and processing capabilities, without any further programming or gluing together. As an immediate advantage of this approach, a control station and indeed any node outside of the WSN will have at least the same configuration and processing capabilities and the same interface as the remaining nodes, without any custom programming needs. More generically, the approach aims to provide immediate homogeneity over heterogeneous deployments with different types of sensor devices, control stations and sub-networks.

Several related issues have to be solved in order to achieve this goal. The mechanisms proposed include: how to perform remote configuration, how to provide flexibility and extensibility for the whole system, without extensive coding, and how provide planning and monitoring operation timing guarantees.

The proposed middleware should provide easy remote configuration or reconfiguration by engineers with no expertise in programming of distributed systems. In industrial settings, monitor and closed-loop tasks are required to control physical processes. Remote configuration mechanisms are needed to configure those tasks. For instance, a control engineer (a person configuring operations) configuring a closed-loop should be able to specify which nodes will participate in the decision, where the decision is taken, which thresholds and actions must be taken and which node(s) will perform the action. All these tasks should be done in the same way, regardless of the underlying devices, so that even when devices are replaced by different ones, the same configurations can be applied.

The monitoring data should also be traffic configurable, concerning which data to process and the acquisition frequency. In-network computation may be included, to process data as close to the source as possible and to avoid network congestion. The middleware architecture has to provide flexibility and adaptability to a wide span of applications.

Operation timings are very important for many DCS scenarios, such as those used in industrial applications. This thesis also addresses how to provide timing guarantees in the heterogeneous distributed system. One way to provide those guarantees is to plan the WSN sub-networks with pre-planned time-division multiple access (TDMA) protocols, estimate and control operations timings, and decide whether to partition the sub-networks. Therefore, we propose an algorithm to plan and control operation timings. The algorithm takes as input the operation timings constraints and a base topology for the WSN sensor nodes. It uses a latency model to estimate operation timings, and if necessary partitions the WSN sub-network in the system until the timings requirements are guaranteed.

The problem statement of this thesis is therefore:

How to provide interoperability between different nodes and provide a single configuration and data processing model in that kind of distributed system that handles different realizations, where the same operations (e.g. complex closed-loop or alarm conditions) can be configured and run without any custom programming over different hardware or on different components of the system (sensors, sink nodes or controlling computers)? How to provide timing guarantees in such environments?

The main goal of this thesis is to answer the above questions from a distributed system configuration and timing perspective.

The state-of-the-art in this area is divided into two main parts: WSN middleware and internet middleware. None of the current solutions addresses uniform configuration and operation over a whole heterogeneous distributed system including WSN and cabled nodes. Moreover, the state-of-art in WSN configuration is focused on middleware for specific platforms and low-level configuration issues. The state-of-art in middeware outside of the WSN consists on middleware platforms that handle sensor data sources and create uniform data formats to be consumed by clients. However those middleware approaches do not allow configuring functionality within WSNs, since they are only focused on wrapping data coming from sensor sources for sharing and processing over the internet.

To address the thesis question, we need to investigate middleware and methods for building systems based on no-coding remote (re)configuration and operation in heterogeneous environments, and how to plan the middleware operations for timing execution guarantees.

1.3. GINSENG Project

This research thesis was done within the scope of European FP7 Specific Targeted Research Project (Strep) GINSENG [1]. The FP7 European project GINSENG, which ran from 2008 to 2012, investigated performance control in WSNs for critical application scenarios. As part of the objective, GINSENG-developed architectures should integrate into existing industry resource management systems, while still providing performance guarantees. In order to achieve this goal, GINSENG needed approaches to provide application-level operations with assured performance. The project had the following work packages:

WP1 – Design and Algorithms for Performance Controlled Wireless Sensor Networks: in this work package mechanisms and approaches were developed to monitor performance in WSNs. Topology control and power control strategies were also studied to save energy, prolong network lifetime, increase network capacity, maximize network coverage, and enhance the overall performance of the network.

WP2 – Network elements and debugging tools for performance controlled WSNs: a predictable and controllable sensor node medium access control protocol was developed. In addition, a debugging tool was also developed. It allows verifying if components are operating within the required performance bounds. In case the required performance bounds cannot be met, it provides mechanisms to identify the error cause.

WP3 – Middleware and system integration: in WP3, the goal was to bridge the gap between the field and the enterprise information systems, with end-to-end performance assurances. This work package comprises approaches to map the WSN properties on stream processing algorithms, management of the algorithms and the logic in all components (backend, middleware and WSN).

Application-layer software to interact with the WSN through a declarative application query interface was developed, where queries and commands can be issued.

WP4 – System demonstration and evaluation: WP4 showed the feasibility of the methods developed in WP1, WP2 and WP3 within a realistic industrial application context. It consisted of software integration, testbed creation and evaluation of the approaches and the integrated system.

Part of the work done in this thesis was supported by GINSENG project under WP3. The main partners working directly in this work package were SAP-research and ourselves (University of Coimbra). Among other functionality, the middleware developed within the project implemented some of the findings and proposals presented in this thesis, providing remote configuration capabilities and, at the same time, operations timings monitoring. It runs in both embedded nodes within WSN and more powerful computer nodes, and provides uniform configuration for the whole network, and full integration of the WSN into the enterprise information system.

Within the GINSENG-developed Middleware, we developed the System Configurator and the node application (GinApp) components, which implemented the proposed mechanisms for uniform configuration and operation over the heterogeneous system.

The node component (GinApp) is an application layer component used as conceptual driver of the GINSENG system. It receives user configurations and generates the digitised data that must be transmitted over the GINSENG deployment.

In order to provide timing execution guarantees with strict bounds over the heterogeneous distributed system, planning mechanisms were studied to take into account sense and send rates, maximum delay and number of nodes. GINSENG results in a planned and careful deployment of the sensor nodes to achieve the desired performance guarantees.

This project included an industrial testbed, where the different approaches were tested in an integrated fashion. The software developed under WP1, WP2 and WP3 was fully integrated and demonstrated within WP4.

Evaluation was very important for the GINSENG project to prove that the envisioned solutions work in real industry settings and that application-specific performance targets can be met. A testbed with two WSN was installed in the Sines oil refinery. The WSNs in the testbed were planned in terms of layout and schedules (GINSENG focuses on totally planned networks, offering performance guarantees). The testbed had 26 TelosB nodes organized hierarchically in two trees, two gateways and a control station receiving the sensor data (Figure 1.5). The control station and gateway computers were the cabled part of the DCS (Distributed Control System), and they were placed in a portable office.

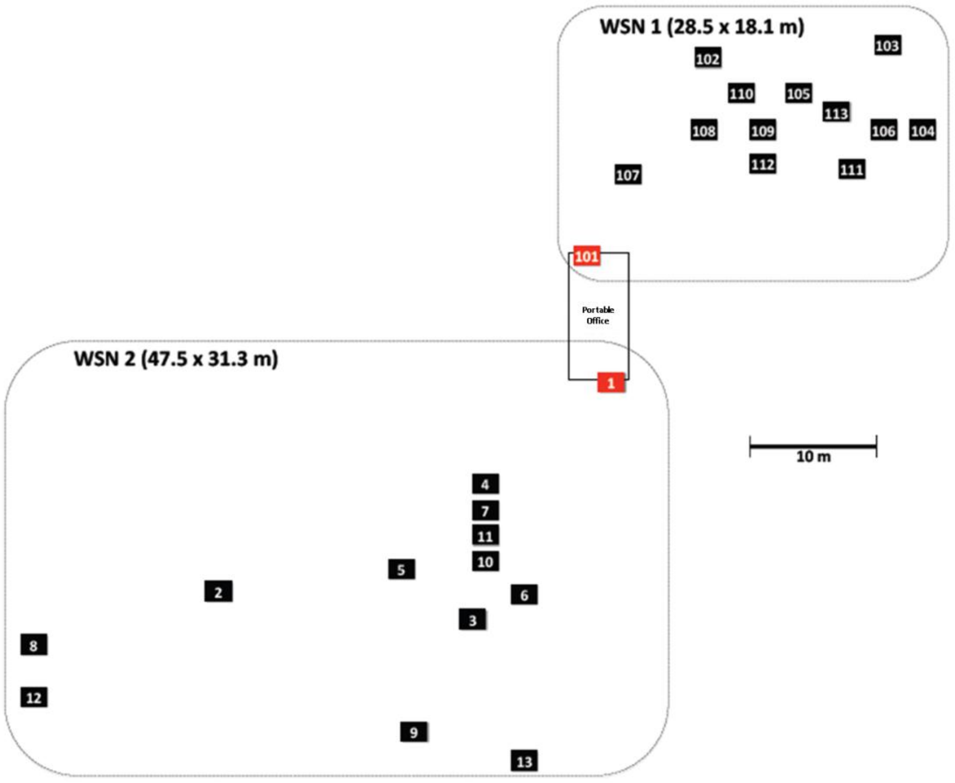


Figure 1.5 – Distributed control system of GINSENG – Sines testbed

Around the portable office, nodes were installed as shown in the example of Figure 1.6. Each node was attached to a sensor in the refinery using the analogue to digital converter of the TelosB nodes.

The WSN nodes ran the Contiki operating system with a TDMA network protocol (GinMac [2]) to provide precise schedule-based communication. The TDMA schedule had an epoch of 1 second where all nodes are available to send and receive data. Nodes within a WSN were time-synchronized and awake for their predefined time slot.

The GINSENG testbed allowed extensive evaluation of all components, both one-by-one and in a fully integrated experiment. Both WP1 and WP2 wireless sensor network-related approaches and WP3 middleware solutions were tested and demonstrated in that testbed, and resulted in a deliverable and a journal paper submission by the project team. The middleware parts that implemented findings presented in this thesis - the GinApp component that is the application-level code running in every node, and the System Configurator that is the interface allowing interaction and configuration of GinApp – were used, tested and demonstrated in the testbed, including also the evaluation of closed-loop control alternatives.

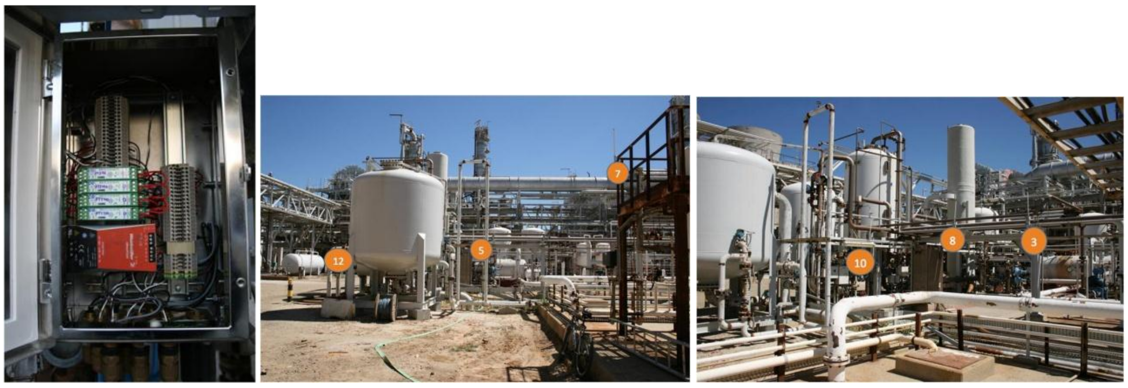


Figure 1.6 – Node deployment

1.4. *Thesis Objectives*

As discussed before, this thesis is around a middleware architecture for handling heterogeneity and providing timing guarantees in networked control systems with heterogeneous devices. We summarize the thesis objectives as:

- **To investigate middleware and methods for building systems based on no-coding remote (re)configuration and operation in heterogeneous environments.** The approaches must be able to deal with heterogeneity and with WSN sub-networks.
    - Modular, API-based architecture;
    - Flexible SQL-like operations management structure;
    - API-based interface plus drivers for heterogeneity handling;

    - Small (GinApp) operating machine fitting both constrained and more powerful devices;
    - Dynamic extension-capable features for the configuration and operation middleware;
- **To investigate planning for timing guarantees of systems running the configuration and operation middleware.**
    - Network dimensioning;
    - Number of sub-networks and gateways needed to provide timing requirements;
    - Monitoring latency forecast and bounding;
    - Actuation latency forecast;
- **To evaluate suitability of middleware and planning in supporting industrial applications, using the GINSENG testbed as a case study.**

1.5. Thesis Contributions

The thesis contributions are essentially described in Chapters 5, 7, 8, 9 and 10. In Chapter 5 we propose mechanisms to handle heterogeneity and distributed operations. Mechanisms to handle different hardware, software and communication protocols are discussed and proposed. We describe how node referencing and homogenization of heterogeneous underlying systems (hardware and software) are achieved and, the data and processing model that provides flexibility in configuration and processing over the heterogeneous sensor network.

In order to provide timing guarantees, in Chapter 7 we propose mechanisms to plan timing operation over heterogeneous distributed systems, and in Chapter 8 we propose the definition and evaluation of bounds to analyse timing requirements. The approach proposes schedules operations, predicts latencies and subdivides the wireless sensor network until the predicted latencies meet operation end-to-end latency requirements.

Lastly, Chapters 9 and 10 evaluate an implementation of the proposed mechanisms.

1.6. Thesis Outline

The remaining of this thesis is divided into ten chapters:

Chapter Two: Background and State-of-the-Art in Middleware Approaches for WSN. This chapter provides some background concerning embedded systems. Hardware and software characteristics and operating systems are introduced. The second part of this chapter examines the related work concerning middleware architectures and remote configuration approaches.

Chapter Three: Background and State-of-the-Art in Scheduling and Network Planning. This chapter provides some background on communication protocols, focused in the medium access control. Then, the related work concerning network planning and scheduling approaches are reviewed.

Chapter Four: Middleware Requirements for Heterogeneous Sensor Networks with WSN nodes. This chapter discusses the application scenario requirements for building systems based on no-coding remote (re)configuration and operation in heterogeneous environments. The chapter analyses application scenarios and explores the middleware requirements that can be extracted from the application scenarios.

Chapter Five: Middleware Mechanisms for Heterogeneous Sensor Networks with WSN nodes. An architecture capable of handing the requirements and application scenarios raised in Chapter 4 should be a module-based, node-adaptable middleware. In this chapter we propose the design of such a middleware and its mechanisms.

Chapter Six: Node and Configuration Components. This chapter proposes an architecture approach for the node component of the architecture (MidSN-NC), which provides uniform stream-based configuration and processing over heterogeneous distributed systems with constrained embedded devices as well as other computing devices. The chapter also describes how the MidSN architecture achieves the (re)configuration of nodes. The remote configuration component and a dynamic agent uploading mechanism are presented.

Chapter Seven: Network and Operations Planning. This chapter proposes an approach to plan for time guarantees over the middleware-ran heterogeneous distributed system comprising all parts of a distributed system with WSN sub-networks. It shows how to plan monitoring and closed-loop operations with restricted time boundaries in the distributed heterogeneous system.

Chapter Eight: Performance and Debugging. This chapter proposes the definition and surveillance of expectable time bounds, to assess system performance compliance in any distributed system. Assuming that we have monitoring or closed-loop operations with timing requirements, this allows a constantly monitoring of timing conformity. We define measures and metrics for reporting the performance to users and for helping users adjust their deployment factors. Those set of measures and metrics are used in debugging, which is given by tools and mechanisms to explore and report problems and system health.

Chapter Nine: Evaluation of MidSN. This chapter reports the evaluation results of an experimental implementation of the MidSN approach and mechanisms proposed in this thesis. It shows that MidSN middleware has a small footprint, is able to run over different hardware and software platforms and to evaluate performance.

Chapter Ten: Evaluation of Planning and Monitoring Approaches. This chapter reports the results of the experimental evaluation of the planning and debugging approaches proposed in this thesis. The objective is to show that MidSN can run over heterogeneous distributed systems with time guarantees.

Chapter Eleven: Conclusions and Future Work. This chapter presents a summary of the key contributions of this thesis, and points out some open interesting research issues that require further investigation.

Chapter 2

Background and State-of-the-Art in Middleware Platforms for WSN

This chapter discusses the state-of-the-art related to middleware. It first provides some background information to help understanding the heterogeneity problem that occurs when a heterogeneous sensor network with WSN sub-networks is built. In Section 2.2, the most relevant works related to remote reconfiguration and middleware architectures are reviewed.

2.1. Background

In this section, an overview of the characteristics of hardware and software which may be used in distributed systems with WSN sub-networks is provided. A heterogeneous sensor network includes a set of sensor acquisition and processing nodes, where each may be a computer node or an embedded device. The network may include wired and wireless technologies and sub-networks.

Firstly, in Section 2.1.1, hardware and software diversity are introduced. It is shown that a modular approach is needed to cover node's heterogeneity, allowing increased flexibility and integration into enterprise systems. To complement the software diversity, in Section 2.1.2, a brief review of wireless sensor operating software is presented.

2.1.1. Hardware and software diversity

Sensor, actuation and computation nodes are the fundamental components of an industrial distributed control system. To enable WSN-based applications, nodes have to provide the following basic functionality:

- signal conditioning and data acquisition for different sensors;
- storage of data (sample data and configurations);
- processing capabilities;
- analysis of the processed data for alert generation;
- actuation;
- scheduling and execution of the measurement tasks;
- management of node configuration (e.g., changing the sampling rate and reprogramming of data processing algorithms);
- reception, transmission, and forwarding of data packets;
- scheduling and execution of communication and networking tasks.

A node can be an embedded device or a more powerful PLC, computer server or workstation and it may need to provide any of the functionalities described above.

PLC or computer based platforms run mainly on Windows, Linux, or other operating systems developed for computer hardware. These platforms are predominantly equipped with standard LAN communication (IEEE 802.11). Because of the high processing ability and high communication bandwidth, these platforms offer the opportunity to use higher level programming languages (e.g. Java, C++), which make it easier to develop and implement software components. Additionally, they support networking protocols like Internet Protocol (IP), which simplifies the integration into enterprise systems.

But although those platforms are very flexible in terms of configuration and computing power, they are not adequate to deploy in each sensing and actuation location, since they are too expensive, big and requiring external power. Embedded devices are more suitable for those scenarios. Typically, they have limited resources, small size and sometimes they are battery operated (e.g. some wireless devices).

Various wireless devices are available today for building WSNs (e.g. MICAz [3], TelosB [4] motes, Waspmote [5], Econotag [6]), and new ones emerge regularly. This diversity offers the possibility to choose a platform that best fits the needs of specific applications.

Typically, processor, radio and memory capabilities of wireless devices are very constrained, making them cheap. The microcontroller unit (MCU) is most frequently programmed in C. This enables the development of a tight code that fits the limited memory size. Application developers have full access to hardware, but at the same time need to take care of resource constraints.

Unlike operating systems for standard computers, such as Windows or Linux, WSN software platforms are highly tailored to the limited node hardware. These are not full-blown operating systems, since they lack a powerful scheduler, memory management, and elaborate file system support.

2.1.2. Wireless sensor operating systems

TinyOS [7] and Contiki [8] are the most widespread operating systems. Other operating systems developed for WSNs include Mantis [9], SOS [10], SensorOS [11], MagnetOS [12], Nano-RK [13] and ERIKA [14]. In the next sub-sections TinyOS [7], Contiki [8], Nano-RK [13] and ERIKA [14] are briefly described.

2.1.2.1. TinyOS

TinyOS [7] is written in nesC [15], an extension to the C language, which supports event-driven component-based programming. The basic concept of component-based programming is to decompose the program into functionally self-contained components. These components interact by exchanging messages through interfaces. The components are event-driven. Events can originate from the environment (a certain sensor reading exceeds a threshold) or from other components, triggering a specific action. The main advantage of this component-based approach is the reusability of components.

The nesC language extension introduces several additional keywords to describe a TinyOS component and its interfaces. NesC and TinyOS are both Open Source projects supported by research community.

TinyOS is the native operating system of the Tmote, but it has been ported to other WSN hardware platforms. TinyOS cannot, however, dynamically load a new executable without a complete image replacement and reboot. Nonetheless, it is the de facto standard tool for WSN programming.

2.1.2.2. Contiki

Contiki [8] is a memory-efficient open source operating system for networked embedded devices. Contiki provides standard OS features like threads, timers, random number generator, clock and a file system support. It includes an IPv6 stack with support for TCP and UDP connections, as well as the Rime radio communication stack.

Contiki is supported by an event-driven Kernel with small footprint. The Contiki kernel consists of a lightweight event scheduler that dispatches events to running processes and periodically calls processes polling handlers. All program execution is triggered either by events dispatched by the kernel or through the polling mechanism. The kernel does not preempt an event handler once it has been scheduled. It supports two kinds of events: asynchronous and synchronous.

2.1.2.3. Nano-RK

Nano-RK is a real-time operating system (RTOS) with multi-hop networking support for use in wireless sensor networks. Nano-RK supports fixed-priority preemptive multitasking for guaranteeing that task deadlines are met, along with support for CPU and network bandwidth reservations. Tasks can specify their resource demands and the operating system provides timely, guaranteed and controlled access to CPU cycles and network packets in resource-constrained embedded sensor environments.

Nano-RK includes a lightweight wireless networking stack for packet forwarding, routing and TDMA-based network scheduling.

2.1.2.4. ERIKA

ERIKA Enterprise RTOS is a multi-processor real-time operating system kernel, implementing a collection of Application Programming Interfaces (APIs) similar to those of OSEK/VDX standard for automotive embedded controllers. ERIKA features a real-time scheduler and resource managers, allowing the full exploitation of the power of new generation micro-controllers and multi-core platforms.

Tasks in ERIKA are scheduled according to fixed and dynamic priorities, and share resources using the Immediate Priority Ceiling protocol. Interrupts always preempt the running task to execute urgent operations required by peripherals.

ERIKA Enterprise includes a RT-Druid Eclipse-based development environment, which allows writing, compiling, and analysing an application. RT-Druid is composed by a set of plug-ins such as schedulability analysis plug-in, which implements algorithms like scheduling acceptance tests, sensitivity analysis, task offset calculation. It also includes a set of design tools for modelling, analysing, and simulating the timing behaviour of embedded real-time systems.

Of these four operating systems for wireless sensor networks, TinyOS and Contiki are the most familiar to the WSN programmer, offering high-level programming languages and all components and capabilities needed to create a WSN.

2.2. State-of-the-Art

This section examines the most prominent work that is related to the problem of configuring and operating over heterogeneous sensor networks, namely, remote configuration and middleware approaches. The section examines two main areas: remote configuration and middleware architectures for WSNs.

Section 2.2.1 examines remote configuration approaches. In the literature there exist several works that address reconfiguration. Section 2.2.2 examines middleware architectures, showing also that they are typically targeted at a single platform. For instance, there are several middleware approaches for WSN operating systems, and other ones for distributed computer-based systems, but those approaches are not configuration and processing middleware running in all devices while providing the same functionalities to all nodes irrespective of their type.

2.2.1. Remote (re)configuration approaches

Several research groups have explored the benefits of HW reconfiguration by designing ad-hoc reconfigurable devices prepared to be adapted to a set of pre-recorded applications [16], [17], [18], [19], [20], [21], [22]. Traditionally, reconfigurable devices are based on Complex Programmable Logic (CPLD) or Field-programmable gate array (FPGA) that have capabilities to adapt to hardware changes. But those do not offer the application configuration flexibility that software approaches do.

Many articles proposed approaches to deal with the reconfiguration of wireless sensor networks. There are several aspects that should be kept in mind when (re)configuration mechanisms are studied. The first one involves the scarceness of resources (memory aspects and processing power need to be taken into account). Nodes in a WSN don’t have copious amounts of RAM such as a computer. The second one involves heterogeneity, or the ability of the network to have multiple node hardware.

The easiest way of reconfiguring a sensor node is to reprogram the node with updated firmware code (over-the-air programming). This is useful especially during the debugging of a sensor network. However, in most cases, reprogramming is too expensive in terms of resource consumption (e.g. memory, energy). Over-the-Air programming and dynamic software updating over WSNs was surveyed in [23], [24]. Such mechanisms allow reconfiguring the nodes without physically removing them from the deployment site, programming them and putting them back into the site. We next review some recent works on the subject, and then we reach conclusions on the comparison with our approach.

Typically, programs for wireless sensors are around a few tens of kilobytes (a single image with OS and application code is built and sent to the node). This relatively large amount of data has to be forwarded to nodes, occupying bandwidth during large periods of time and draining energy supplies as well. At its destination, the updated program must be written to (flash) memory, which requires large amounts of memory.

Some recent works on dynamic uploading and over-the-air programming include [25], [26], [27], [28], [29], [30], [31]. In [26], [28] and [30] the authors consider over-the-air approaches based on rateless codes, which significantly improve over-the-air programming by drastically reducing the need for packet rebroadcasting. The authors of [26] propose a design and implementation based on two rateless protocols, rateless Deluge and ACKless Deluge. They tested the approach and show that it saves significantly communication over regular Deluge. The work in [28] proposes an approach called SYNAPSE, which was designed to improve the efficiency of the error recovery phase. The work described in [30] refers to a new design of a boot loader which allows, at runtime, to switch between SYNAPSE++ and any of the disseminated applications.

In [29] the authors propose a protocol called Freshet for optimizing the energy required for code upload and speeding up the dissemination if multiple sources of code are available. The energy optimization is achieved by equipping each node with limited nonlocal topology information, which it uses to determine the time when it can go to sleep since code is not being distributed in its vicinity.

The authors of [31] discuss the dissemination time. They proposed to reduce the time and energy consumption through compression. Several compression algorithms are studied and compared in the paper.

Over-the-air approaches related with Macro-programming and middleware are proposed in [32], [33], [34]. These approaches typically use a middleware to reprogram the network. Most consist of mobile agents which run over virtual machines. They receive agents over-the-air, and can put them to run over the middleware. Typically, the agent code is generated by specific frameworks. Specific communication protocols are also developed to upload the code. Agent-based approaches are reviewed in more detail in the next section on middleware.

Analysis: Over-the-air programming approaches offer code flexibility, but they have relevant disadvantages when compared with our proposal. Typically, the dynamic upload approaches are targeted at configuring a single WSN, while our approach configures heterogeneous mixed WSN-non-WSN environments. Many of the reviewed works are concentrated only on the technical uploading optimization issues, but the system designer needs to develop the code for the nodes. Therefore, this requires expertise in the programming languages of the platforms involved, plus developing the code by hand for the portion outside of the WSN and the interconnections. It is also a lengthy and buggy process (since the programmer will be coding multiple nodes in a distributed system that needs to interact correctly). In comparison, our approach only requires users to specify operation configuration commands with no further programming, and the API is available for external applications to use directly. Nonetheless, we also added the capability to upload user-programmed agents in our proposed architecture.

Concerning performance and simplicity – there is a significant time overhead associated with dynamically loading the code or code fragments, and there are usually specific dynamic upload protocol requirements, while our approach is very fast. For instance, in our testbed it is possible to (re)configure one or many nodes by sending simple commands through three 10 ms downstream slots (one per tree level) that were made available in the pre-planned schedule based TDMA; the fact that our approach fits nicely into any runtime environment and requires no complex specific extra code updating protocols and related structures is a positive point for simplicity.

2.2.2. Middleware architectures

A middleware layer can be used on top of the operating system to program and execute over the system. The use of a middleware raises the level of abstraction with which users develop applications.

In the literature, most middleware proposals fall under two main classes: those that run only inside the wireless sensor network and under a specific tiny operating system; and those that integrate and process sensor data but work only outside the WSN. The later work on full-blown (non-embedded) computer devices, over IP networks and over non-embedded operating systems.

2.2.2.1. Middleware architectures inside the WSN

There are many proposals of middleware inside WSNs with different architecture approaches such as database abstraction, mobile agents, virtual machines, application driven and message-oriented middleware.

Database abstractions

Database approaches (TinyDB [35], Cougar [36], SINA [37] and DsWare [38]) treat the sensor network as a virtual database, queried through an SQL-like language. They view the whole network as a virtual database system and provide an easy-to-use interface that lets the user issue queries to the sensor network to extract the data of interest.

TinyDB [35] presents a query processing system for WSNs that includes acquisition techniques. It aims to provide a uniform way of accessing the data gathered by the WSN while also minimizing energy consumption. Each sensor node contains a tiny database that is queried using an SQL-like dialect named Tiny SQL. TinyDB requires and runs only on TinyOS, being installed as a single hex code image for each node.

SINA (System Information Networking Architecture) [37] models the network as massively distributed objects. SINA is a cluster-based middleware, and its kernel is based on a spread sheet database for querying and monitoring. Each cell represents a sensor node attribute (in the form of a single value, such as power level and location, or multiple values, such as temperature changes history). SINA incorporates two robust mechanisms: hierarchical clustering allowing scalability and energy savings, and an attribute-based naming scheme based on an associative broadcast to manage the spread sheets.

DsWare (Data service Middleware) [38] provides flexibility by supporting group – based decision, reliable data-centric storage, and implementing a mix of approaches to improve real-time execution performance, reliability of aggregated results and reduce network communication (overhead). DsWare provides applications with services supported by its architecture modules such as data storage, data caching, group management, event detection, data subscription, and scheduling. DsWare supplies applications with a convenient interface so that they do not have to implement their own application data-service. But in the other hand, DsWare does not provide solutions for heterogeneity.

TinyLime [39] is another database approach, designed with different programming paradigm. TinyLime is a database middleware built over TinyOS which is based on LIME [40]. It follows an abstraction model based on shared tuple space and extends LIME by adding features specialized for sensor networks which are not supported by LIME. TinyLime is designed for environments in which clients only need to query data from local sensors. It does not provide multihop propagation of data through the sensor network, which limits the kind of applications for which it is suitable

Mobile agents

The mobile agent concept for sensor networks has been explored fairly extensively [33], [41], [42], [43], [44], [45], [46], [47], [48], [49]. The key to this approach is that applications are modular, and each module can be distributed through the network (mobile code). Transmitting small modules consumes considerably less energy than a whole application. These approaches increase usability and reconfiguration capability. However, the nature of this programming approach means that it only supports one platform (no support for heterogeneity), which makes it unsuitable for devices with limited resources.

Agilla [33] is a middleware with stack-based architecture, which reduces the code size. Agilla allows agents to move from one node to another using clone and move instructions. Up to four agents are supported on a single sensor node. Agilla does not have any policy monitoring agent activities. Also, its assembly-like and stack-based programming model makes programs difficult to read and maintain. Agilla runs only on top of TinyOS.

Impala [41] also proposes autonomic behaviour that increases fault tolerance and network self-organization capabilities. The insight for Impala stems from the observation that sensor networks are long running and autonomous. Impala was specially designed as part of the project ZebraNet. It proposes an asynchronous, event-based middleware layer that uses program modules (mobile agents) compiled into binary instructions and then injected into the network. The approach ensures application adaptation and can automatically discern needed parameter settings or software uses. Programmers can plug in new protocols at any time and switch between protocols at will.

SensorWare [42] defines and supports lightweight mobile control scripts that allow an efficient use of computation, communication and sensing resources at the sensor nodes. This is achieved by means of service abstractions that can change at runtime by dynamically defining new services. Scripts are described as state machines influenced by external events.

COMiS [43] is part of the TinyMaCLaS project. Applications are written in the DCL (Distributed Compositional Language). Compiled component binaries are deployed and installed (registered) at deployment nodes. On receiving an update, the COMiS listener component checks that the version number is greater before installing and re-linking the new component, which is then restarted.

In Scatterweb [45], software is divided into a firmware core and modifiable tasks to provide an update environment. Tasks can register callbacks with the firmware to handle sensor events or received packet types. Software updates are first copied into EEPROM, and then written into flash. It allows checking the received code, and allows synchronization of updates across the network. A host tool is also presented. It supports over-the-air reprogramming via a USB/radio stick (ScatterFlasher) or via a www gateway (Embedded Web Server).

RUNES [46] is a middleware solution which provides publish/subscribe support developed for context-aware systems. It includes a language-independent component model which is supported by a minimal runtime API. Runes is composed by components and its interfaces that are customised for particular networked embedded systems. Nevertheless, the design and implementation of individual software components in RUNES are tasks meant for experts, which may not be easily carried out by an end-user.

MiSense [47] provides a well-defined content-based publish/subscribe service, but also allows the application designer to adapt the service. The middleware is divided in three layers: the communication layer, the common Services layer and the domain layer. It provides services to different kinds of applications, such as data aggregation, event detection, topology management, and routing. The programming interface provides a set of functions that will allow the user to control and program the sensor network as a whole network with different functional characteristics without worrying about the detailed placement of computation and communication.

TeenyLIME [48] is a new middleware for sensor networks based on the tuple space model made popular by Linda [50]. TeenyLIME operates by distributing the tuple space among the devices, transiently sharing the tuple spaces contents as connectivity allows, and introducing reactive operations that fire when data matching a template appears in the tuple space.

TeenyLIME restricts transient sharing only to the tuple spaces of one-hop neighbours. The control of the one-hop neighbourhood around a device, augmented with the powerful and expressive primitives provided by TeenyLIME, is versatile enough to enable a number of application-level uses.

EMMON [49] is a middleware architecture for large-scale, dense, real-time embedded monitoring. It includes a hierarchical communication architecture together with integrated middleware services, command and control software. EMMON was designed to use standard commercially-available technologies, while maintaining as much flexibility as possible to meet specific applications requirements.

Virtual machines

Similar to the mobile agent concept, where arbitrary code can be run, virtual machine middleware is more general because it does not associate code updates with specific structure. But virtual machine execution involves code interpretation, there is a significant run-time overhead cost compared to native binary code.

Maté [32] and SwissQM [34] are virtual machines for sensor networks which were implemented on top of TinyOS. Maté has a stack-based architecture with three execution contexts – clock, send, and receive. Mate breaks down the program into small self-replicating capsules consisting of 24 instructions. These capsules are self-forwarding or self-propagating. Although Maté has a small, concise, resilient, and simple programming model, its energy consumption is high for long running programs.

SwissQM [34] shows a small footprint and it is specializes in data acquisition and data processing. It follows a multi-source to single-sink communication pattern that is materialized by means of spanning-tree whose root is the sink. SwissQM provides functionality that permits data aggregation, program dissemination and topology management. Program fragments in the dissemination protocol are handled in a reliable way by means of timeouts and message snooping.

Squawk [51] is a small Java virtual machine written mostly in Java that runs without an operating system on a wireless sensor platform. Its architecture is based on two components: a class file pre-processor (usually called the translator) and the execution engine. The class file pre-processor translate standard class file into an internal pre-linked file that is compact and allows efficient execution of byte-code. Combining that file with the object serialization mechanism, the Squawk virtual machine can save a set of loaded and translated classes. The result byte-code is sent over a radio link to the devices in the network and, on arrival, it is de-serialized to be interpreted by the execution engine.

Squawk provides an API that allows developers to write and deploy applications for WSNs. It was developed for the Sun Small Programmable Object Technology wireless devices (sunSPOTs).

Application driven and message-oriented middleware

When designing real-time systems, the time triggered approach is expensive in the case where the expected rate of primitive event occurrence is low. An alternative is to use an event triggered approach, where the execution is driven by the events.

Event-driven communication is an asynchronous paradigm that decouples senders and receivers. Its clients are event publishers and event subscribers among which one-to-many and many-to-many communication is supported by a message transmission and notification service. To explore this concept, approaches such as Mires [52], ATaG [53], TinySOA [54], USEME [55] and MiLAN [56] were proposed.

Mires [52] proposes an adaptation of a message-oriented middleware for traditional fixed distributed systems. Mires provides an asynchronous communication model that is suitable for WSN applications, which are event driven. Mires is built on TinyOS using NesC. It adopts a component-based programming model using active messages to implement its publish/subscribe-based communication infrastructure.

ATaG [53] proposes an architecture independent macro-programming model. Applications are specified by means of a graph that is composed of abstract tasks and data items. Abstract tasks model processing functions and data items represent the information that can be exchanged between tasks. Abstract channels are used to connect abstract tasks to data items to indicate whether the former are consumed or produced by the latter. Once the abstract graph has been defined a translator generates a set of templates from it. The behaviour of each abstract task in ATaG needs to be implemented in a template.

TinySOA [54] presents a service oriented middleware. The main entity in TinySOA is the service, which is considered as a computational component that has a unique identifier and it is invoked asynchronously. Users can access the information via services by querying base stations or directly querying individual nodes in the network. TinySOA also allows users to specify a period in the queries to indicate that a query needs to be periodically executed.

In USEME [55], each service is a composition of a number of ports, each of which is a bi-directional interface comprising synchronous and asynchronous communication channels. The application general behaviour is specified using the USEME abstract language and is later automatically translated into a set of templates. These templates need to be filled with platform-dependent code to specify the behaviour of each operation defined in the previous step. Real-time communication constraints can be associated with each command or event helping to ensure timely network operation.

MiLAN (Middleware Linking Applications and Network) [56] is an adaptable middleware that explores the concept of proactive adaptation in order to respond to the needs in terms of QoS. MiLAN allows sensor network applications to specify their quality needs, and adjusts the network characteristics to increase application lifetime while still meeting those quality needs. It provides an abstraction layer that allows network specific plug-ins to convert MiLAN commands to protocol-specific commands that are passed through the usual network protocol stack.

Analysis: These middleware approaches are platform – and operating system – specific (mainly TinyOS). Database approaches, such as TinyDB, treat the whole sensor network as a large "virtual" database. Interactions with the sensors are done in the form of system queries using SQL-like language. It is easy to use, but it does not support heterogeneity. They only support TinyOS and a single TinyOS network. For instance, a heterogeneous network with cabled embedded systems, sensors, computers and multiple WSN sub-networks covering regions of the deployment site cannot be deployed and configured with a single non-programming approach when using TinyDB.

These approaches require a fixed global network structure which is not suitable for large networks with timing guarantees. For instance, create a global schedule for the MAC layer may originate a high delay between samples which may not feasible for some applications.

Approaches such as Agent-based and application driven also have serious problems concerning heterogeneity. For instance, the agent-based approach provides efficient mechanisms for network updates, in order to support dynamic applications. However, the nature of its code does not allow hardware heterogeneity, which makes it unsuitable for different devices with limited resources.

Lastly, virtual machine approaches provide a flexible programming paradigm. They allow the development of distributed algorithms and hide the heterogeneity of the run time environments and the hardware resources. However, the virtual machine approaches add a considerable code size and performance overhead, and applications need to be programmed in detail. It is not possible to simply configure operations.

Many of those approaches were developed for specific platforms. They do not offer heterogeneity support and interoperability between heterogeneous parts (cabled and wireless components was not dealt with).

Nonetheless, all of these middleware approaches lack the capacity for integrating embedded devices into a generic heterogeneous sensor network. Consequently, developing and deploying end-to-end applications for sensor networks in contexts of industrial or other environments remain highly complex.

2.2.2.2. Internet-based integration of sensor data

Traditionally, sensor networks are not IP-based. Their integration into IP-based WAN infrastructures requires the deployment of proxies at the edge of both networking domains that transform between non-IP communication in the sensor network and IP communication in the Internet. For this problem a generic approach to connect all the devices has to be developed. On the higher application level, Web services could be used to mesh and wire all different kinds of sensor information together, but on the lower connection layer, all different devices must be connected to a homogenous access layer. For this reason, several research projects have established to abstract from the hardware to a generic interface.

The Global Sensor Network (GSN) [57] is a middleware that, despite higher level functions, abstracts from the underlying, heterogeneous sensor technologies. GSN-specific Wrappers are used to connect different types of sensor data sources. The authors use the ”Virtual sensors” and Borealis concepts to abstract the sensors from the physical implementations and provide a homogeneous view of sensor data. They build a middleware to operate in computer nodes and gateways, which allows transforming data to a homogeneous format and viewing the whole network as a homogeneous one.

Borealis [58] is a distributed data stream management system. Sensor networks are interfaced with Borealis nodes by intermediary proxies. Each sensor network provides an adapter in order to provide common information to the Borealis node.

IrisNet [59] proposes a two-tier architecture consisting of sensing agents (SA), which collect and pre-process sensor data, and organizing agents (OA) which store sensor data in a hierarchical, distributed XML database. This database is modeled after the design of the Internet DNS, and supports XPath queries.

Hourglass [60] proposes, similarly to IrisNet, a service infrastructure to interconnect sensors and applications via services. Based on publish/subscribe mechanism, producers publish their services and consumers subscribe to interesting services.

HiFi [61] is based on a hierarchical, location-based organization for sensor data processing. Sensors form the leaves of the hierarchical tree and the intermediary nodes are relatively powerful entities performing different operations such as filtering, data cleaning, aggregation and join.

SStreaMWare [62] is a service-oriented middleware for heterogeneous sensor data management. SStreaMWare is a wrapper that uses data schemas to represent data of various types of sensors in a common generic way. Declarative queries can then be formulated according to these schemas. SStreaMWare includes a proxy to hide heterogeneity of sensor software and translate it in generic query services, which can be discovered and used dynamically.

The authors of [63] propose an intelligent bridge for messages exchange between heterogeneous Wireless Sensor Networks (WSNs). They define a general messages exchange mechanism that uses XML as message style and SOAP as transmission protocol.

EdgeServers [64] was designed to integrate sensor networks into enterprise networks. EdgeServers filter and aggregate raw sensor data (using application specific code) to reduce the amount of data forwarded to application servers. The system uses publish/subscribe style communication and also includes specialized protocols for the integration of sensor networks.

ESP framework [65] enables sensor systems to be queried without having to deal with the low level implementation of specific access methods. It provides a mechanism to describe and model sensor systems using ESPml, an XML-based language, by which information regarding the sensor deployment can be specified.

Analysis: These middleware were developed to integrate sensor data into the Internet. They are specific for computer nodes and do not allow configuring operation functionality within sensor networks. Consequently, they are only focused on wrapping data coming from sensor sources for sharing and processing over the internet.

MidSN is a distributed middleware which is able to run over different hardware realizations. None of the reviewed works handles the remote operation configuration and execution on a distributed system with heterogeneity that is addressed in this thesis. Those works provide heterogeneity out-of-WSN. Communication and application-level code needs to be hand-programmed for each WSN node.

2.2.2.3. IP-based homogeneous middleware

There have been several efforts to implement the Internet Protocol Stack on small constrained devices. The 6LoWPAN [66] protocol ports the IPv4 and IPv6 protocols to small devices. This enables running services on the application layer directly on sensor nodes. Web Service technology is often used [67], [68], [69], [70], [71], [72], [73] to connect and access sensors and actuators through the Internet.

Two standards in this area have emerged, the REST-based approach and SOAP-based Web services. Rest utilizes the common http methods (get, post) to transfer data and SOAP uses messages to communicate between services. The structure of the messages and the way to handle those is predefined in the web-service specification. SOAP is less flexible than REST.

The work in [67] describes an implementation of a SOAP based service running on the node. XML Parsing is required to get attributes from requests and to build the response.

The authors of [68] propose a lightweight web server and the Rest engine that runs directly on the sensor node. They used the uIPv4 and uIPv6 protocol to get IP connectivity which exploits techniques from 6lowpan to compress header size. They also compared the Rest implementation with SOAP based solutions, and concluded that the overhead of SOAP based services is ten times higher than their REST approach.

sMAP [69] proposes a simple representation of measurement information and actuation events based on modern REST web service techniques that allows for arbitrary architectural composition of data sources, freeing application designers from tight frameworks and enabling widespread exploration of the sensor application.

AutoWoT [71] proposes a toolkit which allows the user to create web services provided by a specific device and to automatically expose them via a REST API. AutoWoT offers a generic way of modeling Web resources and automatically builds web server components which expose the functionality.

The authors of [72] show how different applications can be built on top of REST WSNs. They describe the best-practices based on the REST principles that have already contributed to the popular success, scalability, and modularity of the traditional Web. The work in [73] illustrates a real world implementation of a REST WSN. The network is deployed across various university buildings and it is designed for the development of applications and services for the university community.

Recently, the CoRE Working Group has defined a REST based web transfer protocol called Constrained Application Protocol (CoAP) [74]. CoAP includes the HTTP functionalities which have been re-designed taking into account the low processing power and energy consumption constraints of small embedded.

The work described in [75] presents a REST WSN based on CoAP. The authors discuss the major differences between CoAP and HTTP and compare the two protocols in terms of power consumption and overhead. Their results show that the power consumption is lower when using CoAP compared to HTTP. The authors also propose and develop an end-to-end IP based architecture integrating a CoAP over 6LowPAN Contiki based WSN with an HTTP over IP based application.

The authors of [76] describe the implementation of the IETF Constrained Application Protocol (CoAP) for ContikiOS, which enables interoperability at the application layer through REST Web services. They demonstrate the power-efficiency of CoAP operation through radio power consumption.

Analysis: These works assume that sensor nodes are powerful enough to run a uIP protocol. Depending of the hardware platform, nodes may not have enough resources to support the overhead introduced by the IP protocol stack.

IPv6, REST and CoAP are important advances in WSN. They provide an infrastructure, but without further software, operations and interactions between nodes must still be hand-programmed.

On the other hand, MidSN is a generic software that is designed to allow users to configure application operations over the whole heterogeneous sensor network without any programming, and to be implementable on a wide range of nodes (embedded devices, PLCs or computers). It allows the implementation of different operation primitives to provide easy and simple (re)configuration. MidSN is based on drivers, which allow abstracting the hardware infrastructure and communication protocols. It is able to run over uIP stack, as well as over non-IP protocols. A correct driver must be developed to handle a specific protocol. As such, MidSN can be developed on top of REST or REST+CoAP.

In terms of application scenarios, MidSN aims to offer a solution to distributed systems with WSN sub-networks. It offers support for dynamic applications with easy configuration mechanisms to increase the usability by non-expert persons.

Chapter 3

Background and State-of-the-Art in Scheduling and Network Planning

This chapter discusses the state-of-the-art related to scheduling and network planning. It first provides some background information concerning medium access control (MAC), communication protocol approaches, and scheduling mechanisms used by time-division multiple access protocols (TDMA protocols) (Sections 3.1 and 3.2).

Planning mechanisms used to plan distributed control systems with timing guarantees are then examined in Section 3.3. The review is focused on network scheduling and dimensioning, and latency models to dimension operations with timing guarantees.

3.1. Medium access control (MAC)

One key issue in WSNs that influences whether the deployed system will be able to provide timing guarantees is the MAC protocol and its configurations. The MAC protocols for wireless sensor networks can be classified broadly into three categories: Contention based, Schedule based or hybrid. The contention based protocols can easily adjust to the topology changes as new nodes may join and others may die after deployment. These protocols are based on Carrier Sense Multiple Access (CSMA) mechanisms and have higher costs concerning message collisions, overhearing and idle listening. In contrast the schedule based protocol can avoid interferences, collisions, overhearing and idle listening, by scheduling transmit and listen periods, but has strict time synchronization requirements.

Both alternatives follows the IEEE 802.15.4 [77] standard, which designates the physical and the data link layer or the media access control layer of the OSI 7-layer model for ultra low-power and low data rate wireless Personal Area Networks (PANs). Many communications protocols where applications demand low power and doesn’t require high speed use this standard in their PHY and MAC layer.

3.1.1. Contention-based MAC protocols

In contention-based MAC protocols, the receiver wakes up periodically for a short time to sample the medium. When a sender has data, it transmits a series of short preamble packets, each containing the ID of the target node, until it either receives an acknowledgement (ACK) packet from the receiver or a maximum sleep time is exceeded. Following the transmission of each preamble packet, the transmitter node waits for a timeout. If the receiver is not the target, it returns to sleep immediately. If the receiver is the target, it sends an ACK during the pause between the preamble packets. Upon reception of the ACK, the sender transmits the data packet to the destination.

These protocols are implemented using units of time called backoff periods. The expected number of times random backoff is repeated is a function of the probability of sensing the channel busy, which depends on the channel traffic. Since these do not provide a precise schedule to send data and use random backoff, they are not useful for applications requiring strict timing guarantees. On the other hand, they can easily adjust to topology changes, such as when new nodes join and others leave after deployment. These protocols have higher costs for message collisions, overhearing and idle listening.

Some protocols frequently used in WSNs, such as S-MAC, B-MAC, WiseMAC and X-MAC, are contention-based.

S-MAC [78] defines periodic frame structure divided into two parts, with nodes being active in the first fraction of the frame and asleep for the remaining duration. The length of each of the frame parts is fixed, according to the desired duty-cycle. Virtual clustering permits that nodes adopt and propagate time schedules, but it leads to the existence of multiple schedules, causing nodes at the border of more than one schedule to wake-up more often.

B-MAC [79] and WiseMAC [80] are based on Low-Power Listening (LPL) [79], that is, a very simple mechanism designed to minimize the energy spent in idle listening. Nodes periodically poll the medium for activity during a very short time, just enough to check if the medium is busy. If they find no activity, they return immediately to the sleep state for the rest of the period until the next poll. Nodes with data to send wake-up the radio transmitting a long preamble (with minimum length equal to an entire poll period). This simple scheme can be quite energy-efficient in applications with sporadic traffic. However, the preamble size (which is inversely proportional to the desired duty-cycle) must be carefully chosen not to be too large, since above a certain threshold it introduces extra energy consumption at the sender, receiver and overhearing nodes, besides impairing throughput and increasing end-to-end latency.

X-MAC [81] is also based in Low-Power Listening but reduces the overhead of receiving long preambles by using short and strobed preambles. This allows unintended receivers to sleep after receiving only one short preamble and the intended receiver to interrupt the long preamble by sending an ACK packet after receiving only one strobed preamble. However, even in X-MAC, the overhead of transmitting the preamble still increases with the wake-up interval, limiting the efficiency of the protocol at very low duty cycles.

3.1.2. Schedule-based MAC protocols

Schedule-based MAC protocols, such as Time-division multiple access (TDMA), have a time schedule, which eliminates collisions and removes the need for a backoff. This increased predictability can better meet the requirements for timely data delivery.

TDMA protocols, with a proper scheduling, allow nodes to get a deterministic access to the medium and provide delay-bounded services. TDMA is also power efficient, since it is inherently collision free and avoids unnecessary idle listening, which are two major sources of energy consumption. The main task in TDMA scheduling is to allocate time slots depending on the network topology and the node packet generation rates. TDMA is especially useful in critical real-time settings, where maximum delays must be provided.

TDMA protocols will schedule the activity of the network in a period in which all nodes will be active. In the idle times between data gathering sessions, nodes can turn off the radio interface and lie in a sleep state.

The disadvantages of TDMA protocols are related with a lack of flexibility to modifications, such as adding more nodes, or data traffic changes over time. Another issue is that nodes have to wait for their own sending slot.

There are various works addressing TDMA protocols. Several protocols have been designed for quick broadcast/convergecast, others for generic communication patterns. The greatest challenges are the time-slots, interference avoidance, low-latencies, and energy-efficiency.

In RT-Link [82] protocol, time-slot assignment is accomplished in a centralized way at the gateway node, based on the global topology in the form of neighbour lists provided by the WSN nodes. It supports different kinds of slot assignment, depending on whether the objective function is to maximize throughput or to minimize end-to-end delay. Interference-free slot assignment is achieved by means of a 2-hop neighbourhood heuristic, coupled with worst-case interference range assumptions.

WirelessHART [83] was designed to support industrial process and automation applications. In addition, WirelessHART uses at its core a synchronous MAC protocol called TSMP [84], which combines TDMA and Frequency Division Multiple Access (FDMA). The TSMP uses the benefits from synchronization of nodes in a multi-hop network, allowing scheduling of collision-free pair-wise and broadcast communication to meet the traffic needs of all nodes while cycling through all available channels.

GinMAC [2] is a TDMA protocol that incorporates topology control mechanisms to ensure timely data delivery and reliability control mechanisms to deal with inherently fluctuating wireless links. The authors show that under high traffic load, the protocol delivers 100% of data in time using a maximum node duty cycle as little as 2.48%. This proposed protocol is also an energy efficient solution for time-critical data delivery with neglected losses.

PEDAMACS [85] is another TDMA scheme including topology control and routing mechanisms. The sink centrally calculates a transmission schedule for each node, taking interference patterns into account and, thus, an upper bound for the message transfer delay can be determined. PEDAMACS is restricted by the requirement of a high-power sink to reach all nodes in the field in a single hop. PEDAMACS is analysed using simulations, but a real-world implementation and corresponding measurements are not reported.

SS-TDMA [86] is a TDMA protocol designed for broadcast/convergecast in grid WSNs. The slot allocation process tries to achieve cascading slot assignments. Each node receives messages from the neighbours with their assigned slots. The receiving node knows the direction of an incoming message, and adds a value to the neighbours slot number, in order to determine its own slot number. A distributed algorithm is required where each node is aware of its geometric position, limiting its applicability to grid topologies or systems where a localization service is available.

NAMA [87] is another TDMA protocol that tries to eliminate collisions dynamically: all nodes compute a common random function of the node identifier and of the time-slot, and the node with the highest value is allowed to transmit in the slot. NAMA is based on a 2-hop neighbourhood criterion (nodes at three hops of distance can reuse slots) for its decisions, but presents an additional drawback of being computationally intensive.

3.2. Scheduling and Network planning

When an operation middleware such as MidSN is applied to contexts with strict timing requirements, in particular in industrial environments such as the one in GINSENG, it is important to provide end-to-end timing guarantees.

In Chapter 7 we propose an approach to plan heterogeneous networks with WSN sub-networks to guarantee operation timings. The approach schedules operations, predicts latencies and subdivides wireless sensor networks until predicted latencies meet operation latency requirements.

In the next sub-sections we review scheduling and network planning approaches used to provide timing guarantees in WSNs. After that, we review some industrial protocols used to design distributed control systems, and to conclude this section we discuss works concerning latency modelling and analysis used in wireless sensor networks and wired distributed control systems.

3.2.1. Scheduling and Network Planning in WSNs

Researchers have quantified the impact of latencies and delays associated with various networks using different communication protocols. The distribution and characteristics of network-induced latencies are mainly influenced by the medium access control (MAC) protocol used. Schedule-based MAC protocols are well-suited to provide timing guarantees in wireless sensor networks, since a slot period of time is assigned to each node, minimizing interference and message collisions. In addition, schedule-based medium access approaches provide good data throughput characteristics [88].

There are a number of scheduling algorithms for WSN networks [89], [90], [91], [92], [93]. In [94], [95] the authors review existing MAC protocols for WSNs that can be used in mission-critical applications. The reviewed protocols are classified according to data transport performance and suitability for those applications.

RAP [89] uses a velocity monotonic scheduling algorithm that takes into account both time and distance constraints. It maximizes the number of packets meeting their end-to-end deadlines, but reliability aspects are not addressed.

SPEED [90] maintains a desired delivery speed across the sensor network by a combination of feedback control and non-deterministic geographic forwarding. It is designed for soft real time applications and is not concerned with reliability issues.

The Burst approach [91] presents a static scheduling algorithm that achieves both timely and reliable data delivery. This study assumes that a network topology is available and a deployment can be planned. Burst achieves end-to-end guarantees of data delivery in both the delay and reliability domains, and therefore it can support a mission-critical application.

The work in [92] describes the optimization problem of finding the most energy-preserving frame length in a TDMA system while still meeting worst-case delay constraints. The authors present an analytical approach to compute that value in generic sink-trees. They also present an implementation using the existing DISCO Network Calculator framework [96].

Traffic regulation mechanisms are also explored as means to provide end-to-end guarantees using queuing models. In [97], the combination of queuing models and message scheduler turns into a traffic regulation mechanism that drops messages when they lose their expectations to meet predefined end-to-end deadlines.

The authors of [93] propose an energy-efficient protocol for low-data-rate WSNs. The authors use TDMA as the MAC layer protocol and schedule the sensor nodes with consecutive time slots at different radio states while reducing the number of state transitions. They also propose effective algorithms to construct data gathering trees to optimize energy consumption and network throughput.

The work in [98] examines the performance of SenTCP, Directed Diffusion and HTAP, with respect to their ability to maintain low delays, to support the required data rates and to minimize packet losses under different topologies. The topologies used are simple diffusion, constant placement, random placement and grid placement. It is shown that the congestion control performance, and consequently the packet delay, in sensor networks, can be improved significantly.

Analysis: In these works, the authors study approaches to define a right scheduling to meet specific requirements, in particular, latency. They optimize the message path and data traffic to achieve their goals. In this thesis we also propose an algorithm to plan the network. A slot-based planning for wireless sensor sub-networks is assumed. The algorithm defines an operations schedule, predicts latencies and divides the network until latencies are according to the requirements. Instead of reducing data traffic or optimizing the message path, in our approach we assume a static tree topology to determine the first-cut scheduling. This assumption is applied within the context of pre-planned performance-guaranteed networks, the subject of GINSENG project. When this first-cut scheduling does not meet the latency requirements, the algorithm divides de network and creates independent schedules for each resulting sub-network. This way it will create partitions of the initial WSN which will meet latency requirements for all constraints in all partitions.

3.2.2. Wireless industrial networks

Wireless process control has been a popular topic in the field of industrial control [99], [100], [101]. Compared to traditional wired process control systems, wireless has a potential to save costs and make installation easier. Also, wireless technologies open up the potential for new automation applications.

There have been some studies on hybrid Fieldbus technology using IEEE 802.11 [102], [103], [104], [105].

In [102], [103] the authors adapt the concept of Fieldbus technology based on Profibus to a hybrid setting (wired plus wireless). In [104], [105] the authors propose R-fieldbus, a wireless Fieldbus protocol based on IEEE 802.11. According to [106], wireless Fieldbus based on IEEE 802.11 has reliability limitations and incurs in high installation and maintenance costs.

Several industrial organizations, such as HART [107], WINA (Wireless Industrial Networking Alliance) [108], ISA (International Society of Automation) [109] and ZigBee [110], have been pushing actively the application of wireless sensor technologies in industrial automation. Nowadays, it is possible to find WirelessHART [83], ISA-SP100 [111] and ZigBee [110] technologies and its protocols in those industrial applications. All of them are based on the IEEE 802.15.4 physical layer. IEEE 802.15.4 [77] is a standard which specifies the physical layer and media access control for low-rate wireless personal area networks (LR-WPANs).

WirelessHART [83] is an extension of wired HART, a transaction-oriented communication protocol for process control applications. To meet the requirements for control applications, WirelessHART uses TDMA technology to arbitrate and coordinate communications between network devices. The TDMA data-link layer establishes links and specifies the time slot and channel to be used for communication between devices. WirelessHART has several mechanisms to promote network-wide time synchronization and maintains time slots of 10 ms length. To enhance reliability, TDMA is combined with channel hopping on a per-transaction (time slot) basis. In a dedicated time slot, only a single device can be scheduled for transmission in each channel (i.e. no spatial re-use is permitted).

ISA-SP100 [111] is a standard for wireless communication in industrial environment. It was developed as a low-power, low-cost, low-data rate to provide robust, reliable and secure wireless operation for non-critical monitoring, alerting, supervisory control, open loop control, and closed loop control applications. It also boasts of providing reliable data communications in harsh industrial conditions and can tolerate latencies up to 100 milliseconds. It has strategized its plan to operate in wireless crowded environments by cooperative operation in order to minimize interferences.

ZigBee [110] is a specification for a suite of high level communication protocols using small, low-power digital radios based on an IEEE 802.15.4 standard. ZigBee devices are often used in mesh network form to transmit data over longer distances, passing data through intermediate devices to reach distant ones. It is targeted at applications that require a low data rate, long battery lifetime, and secure networking. The basic channel access mode is CSMA/CA. However ZigBee can send beacons on a fixed timing schedule which optimize the transmission and provide low latency for real-time requirements.

3.2.3. Planning wireless sensor networks for industrial applications

Generally, wireless sensor devices can be placed as they would have been placed with a wired installation. But several conditions must be considered and could result in a relocation of the wireless sensor devices (or at least the antenna).

Typically, the gateway is placed first, since this is the core element of a network. There are three basic options for placing a gateway:

- Where it is easiest to integrate with the distributed control system or plant network;
- Central to the planned network. Placing the gateway in the centre of the network provides the best position for most of the devices to have a direct communication link with the gateway;
- Central to the process unit. This placement provides the most flexibility for future expansion of the initially-planned network to other areas within the process unit.

It is desirable to have at least 25 % of the wireless devices with a direct communication path to the gateway [112]. This ensures an ample number of data communication paths for the devices farther away.

Similar to network-based fieldbus planning approach, to provide high-degree of timing guarantees, an operation schedule (cycle time) must be developed to avoid unwanted delays and latencies, which influences the overall system responsiveness. To develop the correct schedule, a static topology must be assumed, and the network must be sized according to timing requirements.

Analysis: Most of the previously described standards and protocols are used for deploying planned distributed control system networks in industrial sites. Fieldbus based-protocols can be applied to both wired and wireless networks, and both wirelessHart and ZigBee, running 802.15.4, were developed for wireless sensor networks.

We investigate performance control and timing guarantees in distributed control systems, where wireless sensor sub-networks with specific TDMA network protocol stacks connect configurable and computational-capable sensors wirelessly to a wired infrastructure. In that context, our approach for planning and operation time guarantees defines a network schedule and partitions wireless sensor networks to provide those guarantees. This has some correspondence to the Fieldbus planning in the sense that

Fieldbus defines segments and bus cycle time. In the case of Fieldbus, the maximum number of devices per segment is limited. It is assumed that the developer analyses requirements and decides the segments, the devices that will be in the segments and the bus cycle time to provide adequate data rates. In that approach, planning for operation timing guarantees is not explicit. By contrast, we explore a latency model and explicit operation times planning based on timing constraints, the latency model and a schedule construction. We do these assuming TDMA WSN sub-networks, as implied by the context of project GINSENG. This allows explicit planning of operation timing guarantees. As part of future work, we expect this approach to be adapted to Fieldbus-based architectures as well.

3.2.4. Latency modelling and analysis

In industrial contexts, real-time distributed control systems are typically implemented by a set of computational devices (sensors, actuators, controllers and control stations) that run one or several tasks and communicate data across a communication network. The successful design and implementation of real-time distributed control applications requires an appropriate integration of different parts.

Applying real-time protocols and planning methodologies, latencies and delays can be assessed (determined or, at least, bounded). For example, with respect to fieldbus communication, a formal analysis and suitable methodologies have been presented in [113], with the aim of guaranteeing before run-time that real-time distributed control systems can be successfully implemented with standard fieldbus communication networks.

There exist other approaches to monitor latencies and delays in distributed control systems based on wired component. The authors of [114] and [115] show two studies on modelling and analysing latency and delay stability of network control systems. They evaluate fieldbus protocols and propose mechanism to mitigate latency and delays. In [116] an approach for model end-to-end time delay dynamics for the internet using system identification tools is proposed. The work in [117] presents an analytical performance evaluation of the switched Ethernet with multiple levels from timing diagram analysis, and experimental evaluation from an experimental testbed with a networked control system.

These works assume a wired network. However, a distributed control system may include wireless and wired parts. Latencies and delay modelling in wireless networks has been studied in previous works [118], [119], [120]. However, those works do not concern wireless sensor networks.

There are also some works addressing latency and delays for WSNs [121], [122], [123], [124]. These works have considered the extension of the Network Calculus methodology [125] to WSNs. Network Calculus is a theory for designing and analysing deterministic queuing systems, which provides a mathematical framework based on min-plus and max-plus algebras for delay bound analysis in packet-switched networks.

In [121], the authors have defined a general analytical framework, which extends Network Calculus to be used in dimensioning WSNs, taking into account the relation between node power consumption, node buffer requirements and the transfer delay. The main contribution is the provision of general expressions modelling the arrival curves of the input and output flows at a given parent sensor node in the network, as a function of the arrival curves of its children. These expressions are obtained by direct application of Network Calculus theorems. Then, the authors have defined an iterative procedure to compute the internal flow inputs and outputs in the WSN, node by node, starting from the lowest leaf nodes until arriving to the sink. Using Network Calculus theorems, the authors have extended the general expressions of delay bounds experienced by the aggregated flows at each hop and have deduced the end-to-end delay bound as the sum of all per-hop delays on the path. In [122], the same authors use their methodology for the worst-case dimensioning of WSNs under uncertain topologies. The same model of input and output flows defined by [121] has been used.

In [123], the authors have analysed the performance of general-purpose sink-tree networks using network calculus and derived tighter end-to-end delay bounds.

In [124], the authors apply and extend the Sensor Network Calculus methodology to the worst-case dimensioning of cluster-tree topologies, which are particularly appealing for WSNs with stringent timing requirements. They provide a fine model of the worst-case cluster-tree topology characterized by its depth, the maximum number of child routers and the maximum number of child nodes for each parent router. Using Network Calculus, the authors propose “plug-and-play” expressions for the end-to-end delay bounds, buffering and bandwidth requirements as a function of the WSN cluster-tree characteristics and traffic specifications.

End-to-end delay bounds for real-time flows in WSNs have been studied in [126]. The authors propose closed-form recurrent expressions for computing the worst-case end-to-end delays, buffering and bandwidth requirements across any source-destination path in the cluster-tree assuming error free channel. They propose and describe a system model, an analytical methodology and software tool that permits the worst-case dimensioning and analysis of cluster-tree WSNs. With their model and tool, it is possible to dimension buffer sizes to avoid overflows and to minimize each cluster’s duty cycle (maximizing nodes lifetime), while still satisfying messages deadlines.

Analysis: The previous works concerned latency guarantees of messages over a path. In contrast, we consider timing guarantees of high-level monitor and control operations and over whole distributed control systems. Instead of considering only a wireless sensor network, our proposals address the problem of the worst-case dimensioning of operations in distributed systems with wireless sensor sub-networks, where we schedule operations, predict latencies and, if necessary, partition the wireless sensor sub-network in order to meet operation latency requirements. We also define a model with simple equations that expresses the end-to-end latencies for each node and operation in the network, as well as end-to-end bounds as function of all parts of the path between two nodes in the heterogeneous distributed system.

Chapter 4

Middleware Requirements for Heterogeneous Sensor Networks with WSN nodes

This chapter discusses application scenarios' requirements. The chapter starts by analysing some application scenarios. In this process, we first discuss the requirements of a set of target applications. To limit the number of applications that we must consider, we focus on a set of application scenarios that we believe are representative of a large fraction of the potential usage scenarios.

Next, Section 4.2 explores the middleware requirements that can be extracted from this set of application scenarios. These requirements were considered when designing the architecture of MidSN, which makes it able to support various application scenarios.

4.1. Application Scenarios

There are a lot of applications where sensor networks may be used. These applications range from military surveillance, in which a large number of sensor nodes are used, to health care applications, in which a very limited number of sensor nodes are used. Naturally, these applications have an impact on the specifications of the hardware and software for sensor nodes.

Some researchers have tried to identify possible application scenarios of wireless sensor networks [127], [128], [129], [130], [131]. In this chapter we review a list of possible application scenarios, to identify their requirements and to propose the middleware mechanisms.

We organized the description of scenarios as follows: scenario description is a brief introduction to the scenario. It gives a reader the basic concept about the scenario without going into the details. The description includes also an enumeration of the requirements which characterize the scenario. The following requirements will be analysed:

- Network lifetime – How long do the sensors function reliably;
- Scalability – The scale that a network is capable to grow to without failing to meet users' requirements. We see network size as a part of this requirement as well;
- Time Synchronization – The time precision to which a network has to be synchronized;
- Localization – If the location information is required, and if so, what is the required accuracy/uncertainty;
- Security – The degree of security that a network requires;
- Addressing – The addressing scheme that a network uses;
- Fault tolerance – Level of faults that a system can endure;
- Heterogeneity – If the network may be heterogeneous;
- Traffic characteristics – Most prominent behaviour of data traffic;
- Real-time/End-to-end delay – How critically does delay influence a system;
- Packet loss – Can the system deal with lost packets?
- Traffic diversity – Number of concurrent traffic flows with different characteristics.

4.1.1. Industrial monitoring and control

The value of wireless networks is becoming obvious to organizations that need real-time access to information about the environment of their plants, processes, and equipment, to prevent disruption [132], [133], [134]. Wireless solutions can offer lower system, infrastructure, and operating costs as well as improvement of product quality.

Process control: In the field of process control, nodes collect and deliver real-time data to the control operator and are able to detect in *situ* variations in the processes. Nodes may include different sensors and actuators to monitor and control a physical process. They must be able to adjust, for instance, the speed used by a motor, according to the required output. Wireless distributed networks that link different sensors make machine-to-machine communication possible and have the potential to increase the process efficiency in factories. Table 4.1 summarizes the system requirements for environmental application scenarios [135], [136].

Table 4.1 – System requirements of process control application scenarios

Requirements	**Level**
Network lifetime	1 year
Scalability	Tens
Time Synchronization	Second
Localization	Required (meter)
Security	Low
Addressing	address- centric
Fault tolerance	Middle
Heterogeneity	Yes
Traffic characteristics	Periodic, Queried, Event-based
End-to-end delay	1-3 Second
Packet loss	Occasional (<5%)
Traffic diversity	Medium

Health of Equipment monitoring: Equipment management and control is one application scenario used in industrial environments. Sensor nodes are continually monitoring to evaluate the "health of machines" as well as their usage. Sensors installed on different machines measure physical properties such as temperature, pressure, humidity or vibrations. The sensor nodes are able to communicate between each other and send data to the network where the data is processed. When critical values are found, the system immediately sends alarms, making predictive maintenance possible.

Table 4.2 summarizes the system requirements for environmental application scenarios [135], [136].

Table 4.2 – System requirements for equipment monitoring application scenario

Requirements	**Level**
Network lifetime	Forever
Scalability	Planned deployment
Time Synchronization	No synchronization
Localization	N/A
Security	Not required
Addressing	Address centric
Fault tolerance	Middle
Heterogeneity	Yes
Traffic characteristics	Queried, Event-based
End-to-end delay	1-3 Second
Packet loss	Occasional (<5%)
Traffic diversity	Low

In industrial application contexts, the network is managed by factory employers (no programming expertise), and requires monitoring or closed-loop configurations to control physical processes. Those configurations may change over time with specific process conditions. Failure of a control loop may cause unscheduled plant shutdown or even severe accidents in process-controlled plants. Users should be able to configure the network operations (monitor and control) easily.

4.1.2. Environmental monitoring

WSNs are deployed in particular environments including cities, forests, mountains and glaciers in order to gather environmental parameters during long periods [137], [138], [139], [140], [141]. Temperature, humidity or light sensor readings allow analysing environmental phenomena, such as the influence of climate change on rock fall in permafrost areas.

Sensor networks have evolved from passive logging systems that require manual downloading, into intelligent sensor networks. These networks are comprised of nodes and communication systems that actively transmit their data to a server where the data can be integrated with other environmental data sets.

The main goal of environmental monitoring is to supervise, and study several environment activities. For instance, a weather station provides information about rainfall, wind speed and direction, air temperature, barometric pressure, relative humidity, and solar radiation. These measurements can be useful to forecast the weather and to detect or predict harsh natural phenomena.

The main requirements for environment monitoring application scenarios are “detection”, “alarm” and information gathering and analysis. Long operation time is also an important requirement. Large amounts of data may need to be logged in the field because transmitting them over the network may take too much time and bandwidth, cost too much power, or is simply impossible because the network is isolated from the other parts. Nodes should be able to collect and store sensor data during a time interval and summarize it to be sent after a certain period.

Sensor may also be used for logging environmental data. As the system must be available during months or years, the data sent to the control station should be configured and must be delivered with very slow rates (e.g. 1 time per day) to save energy. The data must be processed in the node and can represent, for instance, a summary of the whole day or week.

Table 4.3 summarizes the system requirements for environmental application scenarios [128][49].

Table 4.3 – System requirements of typical environmental application scenarios

Requirements	**Level**
Network lifetime	2 years
Scalability	Thousands
Time Synchronization	Second
Localization	Required (meter)
Security	Low
Addressing	Location based
Fault tolerance	Middle
Heterogeneity	Yes
Traffic characteristics	Periodic, Queried, Event-based
End-to-end delay / jitter	Seconds
Packet loss	Occasional (<5%)
Traffic diversity	Low

4.1.3. Precise agriculture monitoring and control

Sensor networks are deployed in particular habitats to monitor and control. They may be used to monitor and control agriculture activities. In agriculture activities, sensor nodes play a critical role in measuring and monitoring the health of the soil and water quality at various culture stages [141]. In these kinds of applications nodes require a medium lifetime (they must be available during the culture period). Typically, they deliver data to a base station at slow rates (e.g. 4 times per day – morning, noon, afternoon, night). Similar to environmental monitoring applications, the delivered data can represent readings, statistical information, or aggregated values.

Wireless sensors are further used for irrigation systems. Nodes assume, for example, the tasks of irrigation control and irrigation scheduling using sensed data together with weather sensed data. Finally, sensors are used to assist in precision fertilisation. Based on sensor data, the base station calculates the quantity and spread pattern for a fertilizer.

Table 4.4 summarizes the system requirements for precise agriculture monitoring and control [128].

Table 4.4 – System requirements of precise agriculture monitoring and control

Requirements	**Level**
Network lifetime	3 ~ 5 months
Scalability	Thousands
Time Synchronization	Minute
Localization	Required (meter)
Security	Low
Addressing	Location based
Fault tolerance	Middle
Heterogeneity	Yes
Traffic characteristics	Periodic, Queried, Event-based
End-to-end delay / jitter	Seconds
Packet loss	Occasional (<5%)
Traffic diversity	Low

4.1.4. Smart buildings monitoring and control

Smart buildings rely on a set of technologies that enhance energy-efficiency and user comfort as well as the monitoring and safety of the buildings. These kinds of applications are used to monitor, for instance, heating, lighting and ventilation.

In smart buildings applications, sensor nodes are connected via specialised networks (wired – e.g. x11 or wireless), which allow them to be controlled remotely, e.g. switching off computers, monitors or lights when rooms and offices are not occupied [142].

Sensor nodes may deliver data to decision support systems with various rates (the rate must be configured to a specific sensor or monitor functionality). Nodes should also be able to raise alarms when a critical value is achieved. Those critical values should also be configurable and may change over the time.

Table 4.5 summarizes the system requirements for smart buildings monitoring and control application scenarios [142],[128].

Table 4.5 – System requirements of smart buildings monitoring application scenarios

Requirements	**Level**
Network lifetime	3 ~ 6 months
Real-time/End-to-end delay	Seconds or tens of seconds
Scalability	Hundreds or thousands
Synchronization	Milliseconds
Security	High
Addressing	Data centric
Heterogeneity	Yes (sensors and actuators)
Traffic characteristics	Periodic, Queried, Event-based
Traffic diversity	Medium

4.1.5. Warehouse tracking

Inventory control is a major problem for big companies. Management of assets (pieces of equipment, machinery, different types of stock or products) can be a predicament. Warehouse tracking in a company with large storage capacity will permit control over the products stored before their delivery to the end costumer. With the help of Wireless Sensor ID tags each container has an electronic re-configurable identifier able to transmit the product code, date of production, date of storage or other valuable data with the remote server. If each item is fixed with a tag, an inventory could be automatically updated if anything is removed or added.

The wireless sensor ID solution makes use of sensor devices with localization features. Low data rate is foreseen, link distance is relatively small and the warehouse is similar to a semi-open indoor environment, with localization accuracy in the range of 30 cm with high tracking capabilities. No critical requirements in terms of latency are considered. But energy consumption is an important feature, because the system will not be feasible if the tags are not equipped with a lasting battery.

Table 4.6 summarizes the system requirements for warehouse tracking application scenarios [128].

Table 4.6 – System requirements of warehouse tracking application scenarios

Requirements	**Level**
Network lifetime	6 ~ 12 months
Real-time/End-to-end delay	Seconds or tens of seconds
Scalability	Tens of thousands
Localization	Required (centimetre)
Synchronization	Second
Security	Low
Addressing	Data centric
Heterogeneity	Yes (sensors and RFID tags)
Traffic characteristics	Queried
Traffic diversity	Low

4.1.6. Transport logistics

Consider a shipment of bananas as it travels from the farm in Madeira, Portugal to a supermarket distribution centre in Lisbon. The bananas are packed in boxes stacked onto pallets, each equipped with a tracking device. From the farm, these pallets travel in trucks to a loading dock at the harbour, where they are loaded into shipping containers that carry them all the way to the supermarket chain’s distribution centre.

The logistic companies use the wireless sensors to monitor the conditions of the goods, expected arrival or delay times and more automatic shipping configuration. The transport logistic scenarios involve sensor network and RFID components in the transport vehicle and the transport centre. While the sensor network of the transport vehicle is connected by mobile radio networks or wireless local area networks at the transport centres, the sensor networks of the transport centres are connected by wired lines. Typically, this application scenario requires mobile data connections. To limit the data transmitted over mobile radio networks and the costs associated with that, a remote configuration of limits for events for event driven data transmission is required.

Table 4.7 shows the system requirements for transport logistic application scenarios [143].

Table 4.7 – System requirements of transport logistic application scenarios

Requirements	**Level**
Network lifetime	N/A
Real-time/End-to-end delay	Milliseconds or seconds
Scalability	Tens or hundred
Localization	Not required
Synchronization	Milliseconds
Security	High
Addressing	Data centric
Heterogeneity	Yes (different classes of nodes)
Traffic characteristics	Periodic, Queried, Event-based
Traffic diversity	Medium

4.1.7. Surveillance

WSNs have also been used for military and civil surveillance. Surveillance is taken as the process of monitoring the behaviour of people, objects, or processes within systems, for security or social control. WSN technology is very well suited for surveillance systems, mainly because wireless sensor networks do not require any wired infrastructure.

However, surveillance needs to be categorized in order to take into account the different requirements of the different cases, namely indoor and outdoor surveillance.

Indoor Surveillance: Indoor surveillance has two possible application scenarios: surveillance systems placed in a private environment such as houses, or in public buildings such as hospitals, museums, and airports. The nodes may have different energy capacities, processing capabilities, positions, and radio coverage. Wireless networks have many advantages over their wired counterparts that need to be taken into consideration as well. They are easily deployed, have ubiquitous connection, are low in maintenance, and are unobtrusive.

Sensors, such as thermal sensors or volumetric sensors allow estimating alarm conditions and examine the occupation in fixed areas of the building, where it is installed. Nevertheless, video surveillance systems are necessary to identify the alarm source detected by the sensors. Table 4.8 shows the system requirements for indoor surveillance application scenarios [144].

Table 4.8 – System requirements of indoor surveillance application scenarios

Requirements	**Level**
Network lifetime	N/A
Real-time/End-to-end delay	Milliseconds or seconds
Scalability	Tens or hundred
Localization	Not required
Synchronization	Milliseconds
Security	High
Addressing	Data centric
Heterogeneity	Yes (different classes of nodes)
Traffic characteristics	Periodic, Queried, Event-based
Traffic diversity	Medium

Outdoor Surveillance: Outdoor surveillance is also highly important for perimeter security, such as keeping prisoners inside the premises or keeping intruders out of a certain area [145]. When using invisible surveillance, it is fundamental that intruders are not able to detect its presence and then sabotage the detection system. For this reason, several technologies have been developed to yield a cost-effective solution for particular proprietors.

Outdoor surveillance applications are not exactly the same as monitoring or control applications. In those applications, data is transferred to the decision support system at regular intervals. However, in outdoor surveillance applications, communication is mostly event-driven. For instance, the communication is done only when intrusion activity patterns are received in the perimeter antenna, the system activates an alarm. Table 4.9 summarizes the system requirements for outdoor surveillance application scenarios [135].

Table 4.9 – System requirements of outdoor surveillance application scenarios

Requirements	Level
Network lifetime	3 ~ 6 Months
Real-time/End-to-end delay	Milliseconds or seconds
Scalability	Hundreds or Thousands
Localization	Required
Synchronization	Milliseconds
Security	High
Addressing	Location-based
Heterogeneity	No
Traffic characteristics	Queried, Event-based
Traffic diversity	Low

4.1.8. Health care

WSN technology could potentially impact a number of health-care applications, such as medical treatment, pre- and post-hospital patient monitoring [146], people rescue [147], [148], [149], and early disease warning systems [150].

The health care application scenario is a very special scenario, since nodes store the data until some instant in time. All sensors are connected to one node which is an embedded device. The node is placed on the body (human or animal body). Store and processing capabilities are the most important aspect in this scenario, because the node must collect and process data until a decision support system requests it. For instance, if a patient goes to the hospital and uses body sensors him to monitor the heartbeat during a day [147], the node must be able to store the data it collects during a day and unload the data only when the patient returns to the hospital.

Another application scenario [148] referring node processing and actuation is when a diabetic patient is monitored for insulin injection. In this example, the node collects and processes data (within itself) to control the quantity of insulin that must be injected on the patient.

Table 4.10 shows the system requirements for health care application scenarios [128], [151], [152], [153].

Table 4.10 – System requirements of health-care application scenarios

Requirements	**Level**
Network lifetime	700 hours
Real-time/End-to-end delay	It changes as vital parameter type and analysis of vital data. But not more than seconds
Scalability	Tens of nodes.
Localization	Not required
Synchronization	N/A
Security	High
Addressing	Requires data centric
Heterogeneity	Yes (sensors, sink nodes/PDAs)
Traffic characteristics	Queried, Event-based
Traffic diversity	Low

The application scenario for medical care can be extended further to incorporate other health monitoring applications like athletic performance monitoring, for example, tracking one's pulse and respiration rate via wearable sensors and sending the information to a personal computer for later analysis. Yet another extension is at-home health monitoring, for example, personal weight management [149]. The patient's weight may be wirelessly sent to a personal computer for analysis and storage.

4.2. Middleware Requirements

Several middleware requirements can be extracted from the previous scenarios. The design of the MidSN pays attention to functional and non-functional requirements. In this section we extract the main requirements from the previous application scenarios and discuss how MidSN deals with it. We consider data processing, configuration and user operations as functional requirements, while interoperability, adaptability and performance are considered as non-functional requirements.

Data acquisition and processing: Data acquisition is central to distributed system management architecture. Data have to be acquired and stored in the first place. Then, it needs to be processed (e.g., format adaptation, filtering), transferred, further processed or merged and delivered to users. Instead of having to program and deploy different parts and interconnections between them, any nodes, be it computers or sensor nodes, should be configured and operating the same way. Each node should be able to acquire, store, process and transmit data to other nodes. This turns a heterogeneous platform into a homogeneous operation layer.

Heterogeneity: MidSN should be modular and based on drivers and interfaces, which allow it to run over different hardware and software platforms. It must be developed only once for each operating-system.

When developing MidSN for a not-yet-supported operating system, the first thing to do is to develop a set of drivers that offer a common architecture defined API, translating the corresponding calls to operating system calls. The common API is then used by a uniform node-component.

Interoperability: MidSN should make different software platforms interoperate transparently. All nodes have to be provided with standard interfaces to access the data (different nodes have to be abstracted and accessed in the same way using a common API).

The API allows any developer to connect and manage the distributed system as well as to create inputs/outputs for enterprise systems. Given adequate API and data access capabilities, appropriate user interfaces can be developed.

Flexibility: The wide range of use scenarios being considered means that MidSN must be flexible and adaptive. Each application scenario will demand a slightly different mix of operations, lifetime, sample rates, response times and in-network processing. A wireless sensor network architecture must be flexible enough to accommodate a wide range of application behaviours.

System configuration and Adaptability: Alarm thresholds and actuations need to be configured for measurements performed by nodes. In various application contexts it is necessary to configure monitoring parameters. Sometimes, it is also necessary to store data locally.

MidSN has to deliver adaptive and configurable services, e.g., services have to adapt to engineering needs without requiring developers to write further code to handle such needs. This avoids programming bugs and reduces the time needed to reconfigure the network.

Users: Different classes of users, with different knowledge, have to be considered. MidSN must be simple to be managed by experts or non-experts. All of them should be able to configure system parameters and access node's data.

Performance: Response time has to be bounded and there must be the possibility to reconfigure due to timing requirements, in order to allow the use of the architecture in industrial scenarios (factories/refineries) with real-time constraints.

Table 4.11 shows, systematically, the system requirements used to develop MidSN architecture.

Table 4.11 – System requirements for MidSN architecture

Requirements	**Level**
Network lifetime	1 year
Scalability	Tens or Hundreds
Time Synchronization	Second
Localization	Required (meter)
Security	Low
Addressing	address- centric
Fault tolerance	Middle
Heterogeneity	Yes
Traffic characteristics	Periodic, Queried, Event-based
End-to-end delay	1-3 Second
Packet loss	Occasional (<5%)
Traffic diversity	Medium

Chapter 5

Middleware Mechanisms for Heterogeneous Nodes

An architecture capable of handing the requirements and application scenarios raised in Chapter 4 should be a module-based node-adaptable middleware. In this chapter, we propose the relevant mechanisms to design such a middleware.

The mechanisms should support both a wide-range of operations and application scenarios, and include a data management engine that is autonomous in each device and independent of node type.

The most important mechanisms proposed to handle heterogeneity and distributed operations are described in Sections 5.2, 5.4, 5.6 and 5.8. Section 5.2 describes the platform and communication protocol independency. In order to run the middleware over different hardware, software and communication protocols, it needs to be supported by drivers. In this section we describe each driver and its implementation.

Section 5.4 describes how node referencing and heterogeneity are achieved with the proposed architecture. The architecture defines a gateway component that translates between communication protocols (e.g. IP to/from Rime, IP to/from ZigBee).

Section 5.6 describes the data and processing model that offers flexibility in configuration and processing over the heterogeneous sensor network.

Lastly, Section 5.8 describes the user API. This API allows users to interact and configure nodes remotely for operation in a distributed system.

The objective is to propose mechanisms to design a middleware that provides **uniform stream-based configuration and processing** over **heterogeneous distributed systems with constrained embedded devices as well as other computing devices**.

5.1. Architecture

The architecture assumes networked distributed monitoring and possibly also actuation contexts. Nodes can be completely heterogeneous, consisting of different classes of devices. For instance, it may include resource-constrained sensor nodes, such as embedded devices, and more powerful nodes, such as computers or servers, as shown in Figure 5.1. In this model, computers are not just data sinks but may fully participate in the distributed computation environment.

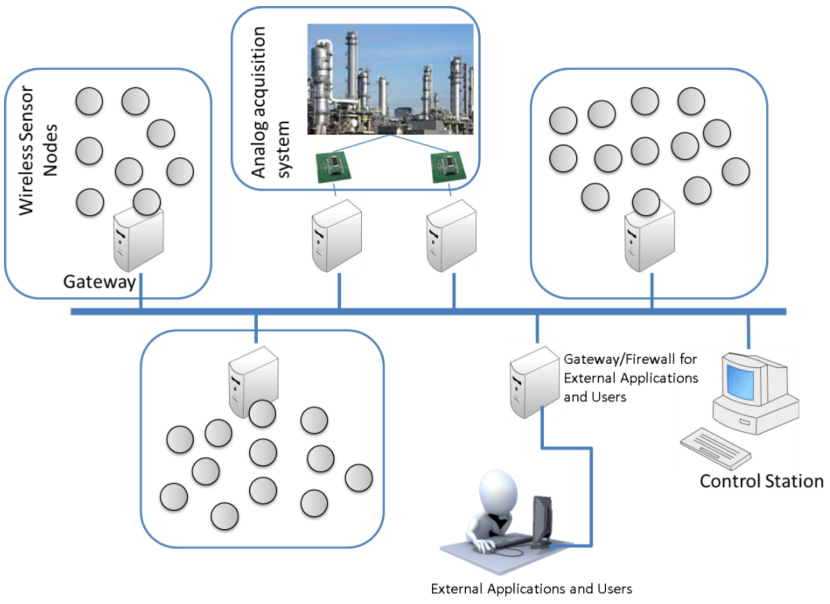

Wireless Sensor Nodes
Gateway
Analog acquisition system
Gateway/Firewall for External Applications and Users
Control Station
External Applications and Users

Figure 5.1 – Network structure example of a distributed control systems with wired and wireless nodes

The system in Figure 5.1 has three different WSN (sub) networks and wired sensors/actuators. The MidSN architecture allows external applications and users to see and use the system as a single, coherent distributed sensor, actuator and computing system.

Considering the network model shown in Figure 5.1, MidSN defines a node component (MidSN-NC) that must be included in all nodes of the sensor network, including any computer and any type of computation-capable node.

The architecture works on top of a network communication infrastructure that is used to exchange data messages between nodes, send configuration commands to nodes (MidSN-NC) and send acknowledgements from them.

Node interactions are based on a small set of primitives: user configuration requests are routed to the appropriate node(s) in the form of command messages that will configure node functionality; messages (which specify a type) are also used to send alarms, actuation values and notifications from and to nodes; streams are used to collect sensed data, to compute and filter that data and to exchange resulting data between nodes. Figure 5.2 shows the MidSN architecture. The architecture can be divided into two main components: node and remote configuration and data subscription components. The node component is composed by the operating system, the MidSN-NC and the necessary drivers to interconnect it to the operating system, and a debugging module. The other component consists on remote configuration and data subscription modules, as well as a Catalog to store network information (e.g. configuration and status). The remote configuration and data subscription modules can be deployed in a single machine or in a distributed fashion (e.g. using three different machines, one for the Catalog, another for the remote configuration module and a third machine for the publish/subscribe module). This second component also includes a Performance Monitor module.

The debugging module present in the node component collects information from operation execution in MidSN-NC, then formats and forwards information to the Performance Monitor module. The Performance Monitor gathers the status information coming from nodes, stores it in a database and processes it according to the metrics defined in Chapter 8. In Chapter 8 we describe in detail how the Debugging module and Performance Monitor module work.

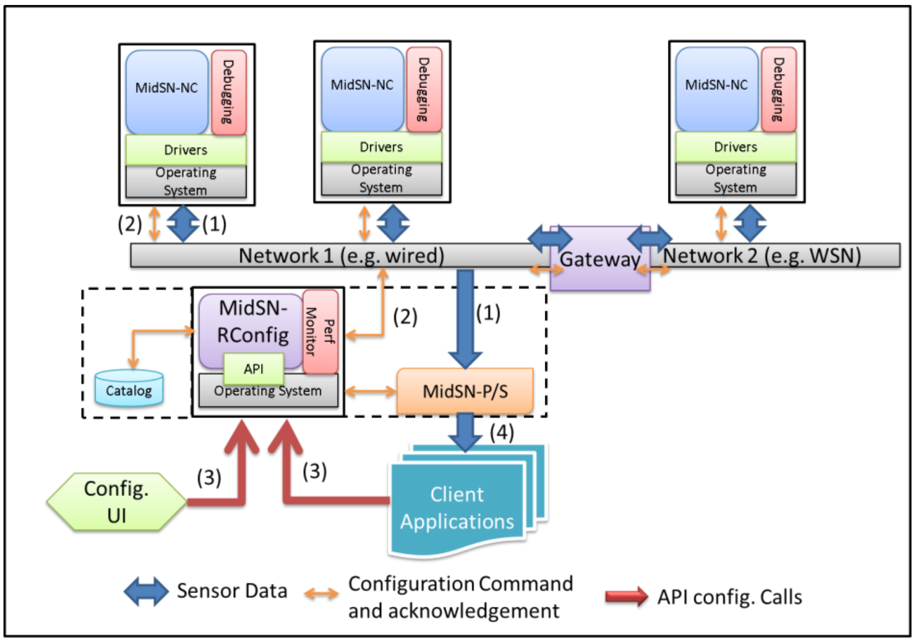


Figure 5.2 – MidSN architecture

Figure 5.2 shows also the interactions (flows) between different components of a distributed system. Flow (1), labelled "Sensor Data" represents sensor data flows. These can be to/from any two MidSN nodes. The flow (2), labelled "Config Command and Ack", represents commands that are sent through the network layer by the MidSN remote configuration component (MidSN-RConfig) to configure nodes, and the acknowledgement reply. This component allows to (re)configure any node in the network. User configurations will translate into command packets that are sent to nodes, generating configuration command flows. When a command is received by a node, an acknowledgement is generated and sent to MidSN-RConfig.

Outside applications and configuration user interfaces access the MidSN-RConfig component through an API provided by that component. The external applications submit calls to the API, which provides configuration-related calls. It also provides data subscription services, whereby external applications subscribe to streams to receive the data coming from somewhere in the networked system. For instance, when a client application wants to receive a sensor data stream, it is necessary to subscribe the data stream. This subscription is done through MidSN-RConfig API calls and a publish/subscribe module (MidSN-P/S) is available in computer nodes to provide the data to the outside applications. In Figure 5.2, API calls are represented by flow (3), labelled as “API Config calls” and data resulting from data streams subscription is represented by flow (4). The publish/subscribe module (MidSN-P/S) is described in Section 5.5.

We assume an extensible architecture, where there are a basic set of configuration commands and operations, but it is possible to add other types of commands and operations to fit the context requirements, by adding functionality to modules in the system.

5.2. Platform and Communication Protocol Independency (Drivers)

MidSN architecture proposes a node component (MidSN-NC). This node component was designed to be “platform and communication protocol” – independent in terms of design and formats, which allows the system to run over heterogeneous distributed platforms, with constrained embedded devices as well as other computing devices. It must be developed only once for each operating-system. Since every node of systems will use the same API and formats, and the approach also assumes gateway translation between different communication protocols, they will have total interoperability, with simple deployment of the nodes. In order for MidSN-NC to run over different hardware and communication protocols, it needs drivers to manage files, handle different communication protocols and sensors, as well as drivers to handle timer events and memory requirements (Figure 5.3). When developing MidSN-NC for a not-yet-supported operating system, the programmer needs to develop a set of drivers that offer a common architecture defined API, translating the corresponding calls to operating system calls.

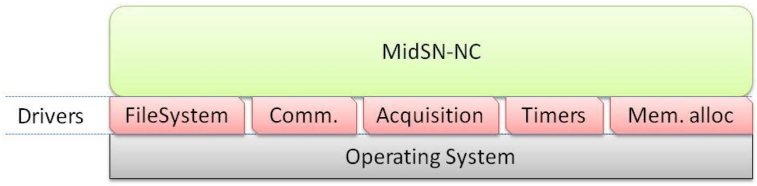


Figure 5.3 – MidSN-NC drivers

The **File system** driver manages data log files in the node. It allows the creation and deletion of log files from the node, as well as adding and reading data from them. Table 5.1 shows the primitives of a driver for handling file system operation.

All of these functions use a file name identifier to identify a file (FILE_NAME). The read and write functionalities use fseek parameters to indicate the start portion of read or write operations.

Table 5.1 – Primitives of file system driver

Functionality	Primitive
Create file	`boolean create_file( FILE_NAME );`
Open file	`boolean open_file( FILE_NAME );`
Close file	`void close_file( FILE_NAME );`
Read data from file	`CHARPT read(  FILE_NAME, fseek, length );`
Write data to a file	`boolean write ( FILE_NAME, fseek, CHARPT buff);`

The **Communication** driver is essential to abstract from the communication protocol primitives. It must offer functions to open and close a peer-to-peer connection and send and receive messages (configuration commands and data messages from other nodes). Each connection is an end-to-end connection where send and receive data/command functionalities are available.

Table 5.2 shows the required primitives to develop a communication driver.

Table 5.2 – Primitives of communication driver

Functionality	Primitive
Open connection	`boolean open_connection(ADDRESS, PORT);`
Close connection	`void close_connection(ADDRESS);`
Send message	`int send_to(ADDRESS, CHARPT packet, int length);`
Receive message	`CHARPT midsn_get_received_packet();`

MidSN-NC, at start up, starts a daemon, which listens for new connection requests sent from other nodes. When a node detects an incoming connection request, it starts up a separate thread to handle node requests and to answer to them. Each thread terminates its execution when the *close_connection* function is called.

The receive function must return a data message. Depending of the programming idiom implementation, the data buffer used by the send function and data message returned by the receive function represent a char pointer if C idiom is used or an object if object-oriented languages such as Java or C# are used.

The driver should be able to notify the MidSN-NC that a new message arrives when the operation system receives the new message. This notification is done using a *new_message_arrives* event.

Figure 5.4 shows a flow chart of the mechanism used to implement the reception of messages (command configurations or data messages) and how to indicate to the MidSN that a new message arrives. This is part of the communications driver that must be developed once for each operating system, as part of the MidSN-NC component.

As shown in the figure, the communication driver must be able to continuously listen to the connection, in order to receive messages. Upon reception, the driver must copy the data buffer to internal memory and notify the MidSN-NC that a new message is available.

After MidSN-NC receives the notification, the I/O adapter (a module of MidSN-NC described before) uses the function *midsn_get_received_packet()* offered by the driver to receive the message and further analyse it.

Appendix A shows two implementations of the communication driver. The first one was developed for ContikiOS, using Contiki-C and the other one for Linux, using java.

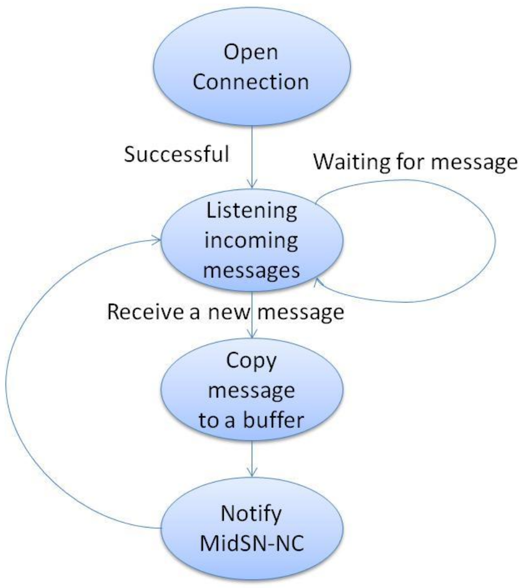


Figure 5.4 – Communication driver flow chart

The **Acquisition** driver must be present in nodes doing sensing. It performs all sensor actions required to gather data from sensors. It must offer the following methods:

- Setup – must be able to initialize sensors. It must return a boolean to check if the initialize step was successfully done or not.
- Read Sensor – allows reading a value from a sensor. The reading is represented by an integer value that must be returned.

The **Actuation** driver must be present in nodes doing actuations. It performs all actuator actions required to control the environment. It must offer the following methods:

- Setup – must be able to initialize actuators. It must return a boolean to check if the setup step was successfully done or not.
- Write a digital value to the actuator – allows setting the output value of a DAC presented in the actuator.

The **Timers** driver allows scheduling timer events. The driver must offer functionalities to set, reset and stop a timer, as well as one function to check if a timer has expired. Table 5.3 shows the required primitives.

Table 5.3 – Primitives of a timer driver

Functionality	Primitive
Set and start a timer	`void set_so_timer( TIMER_NAME, TIMER_PERIOD);`
Stop a timer	`void stop_so_timer( TIMER_NAME );`
Reset a timer	`void reset_so_timer( TIMER_NAME );`
Check if a timer is expired	`boolean so_timer_expired( TIMER_NAME );`

All of these functions use a timer identifier to identify a timer (TIMER_NAME). The set timer function also uses a timer period to indicate the period of the timer. This period is indicated in milliseconds.

Lastly, the **Memory Allocation** driver is used to interface the MidSN-NC memory functionalities with the operating system memory resources. This driver is composed by two functions: alloc and dealloc. The alloc function reserves a specific size of memory to be used by MidSN-NC. The dealloc function releases the unnecessary memory that is no longer needed.

These drivers must be developed according to the hardware and/or communication protocols and linked to the MidSN-NC code for installation of software in a node.

5.3. The Catalog

MidSN maintains a Catalog of nodes, with address (global IP address and proprietary communication address), current node configuration and node status for each node. It is also responsible for keeping a history of submitted configurations and network configuration. Appendix B shows the structure of the Catalog. It is XML-based.

Each MidSN node is identified in the Catalog by the tags <node>. It includes a node identifier (id), a global network IP, a reference to the communication protocol, an address for a specific communication protocol, and a port used to communicate, as well as the configurations of the node.

Each address (IP and non-IP address) is assigned by the user and should take into account the network and sub-network where the node is deployed. After concluding the address assignment process to a node, the Catalog is updated. The Catalog is stored on a catalog-holder control station, and nodes must be configured, before deployment, with the address of that control station.

The example of Catalog shown in Appendix B shows how the information about a network is stored. For instance, considering the node 1.1, represented in the Catalog, we can see which controllers were loaded to the node and which are running. Each user-programmed controller is referenced by the tag <controller> and has a name and a status indication (<running>).

In the same node, we can also verify which streams are configured and its properties. In the Catalog, each stream is referenced by <stream>. It includes parameters such as name, rate, window, output destination, information about its status (if it is running or not), measures and metrics that compose each data tuple of the stream.

Besides controllers and streams information, the node includes also information about configured alarms and actions. Each alarm includes all information of a stream but add and operator and one second measure to create a conditional output. It is identified by the tag <alarm>.

Each action includes all information of both stream and alarm and add an actuation that must be executed when the conditional output occurs.

Lastly, each node includes information about its address and status. The addressing information is described in the next section under the node referencing description. The status includes information about battery level, if the node is battery operated, messages sent, received, forwarded and lost.

In order to enable applications to discover what is available for use, each node includes information about which sensors and actuators are connected to the node.

5.4. Node Referencing and Heterogeneity

In a heterogeneous network, any node (wireless or wired) acts as a data and command source and destination within the network, and must be able to address other nodes for those functionalities. Communication and remote access rely on a uniform distributed addressing and communication layer. However, many platforms with WSN and control stations are heterogeneous, with embedded devices featuring operating systems such as TinyOS and specific communication protocols. In order to handle communication protocol heterogeneity, MidSN defines a gateway component. This provides support for communication with non-IP embedded devices.

The Catalog identifies IPs that must be routed through the gateway, and the gateway is an IP node itself which implements specific communication with a non-IP sub-network of a specific type.

Each gateway implements two interfaces: one used to establish communication through IP networks and the other one used to establish communication with embedded communication protocols such as Rime or uIP. The IP interface is referred by the iIP_address and iIP_port used in the Catalog (Appendix B).

The embedded communication protocol interface is referred by the iWSN_protocol, iWSN_address, iWSN_port and iWSN_channel properties shown in Appendix B.

Figure 5.5 is an illustration of the gateway mechanism of MidSN. The MidSN gateway translates between two network protocols – in the figure one gateway translates between Rime for Contiki and IP for the internet. The other gateway translates between uIP and IP.

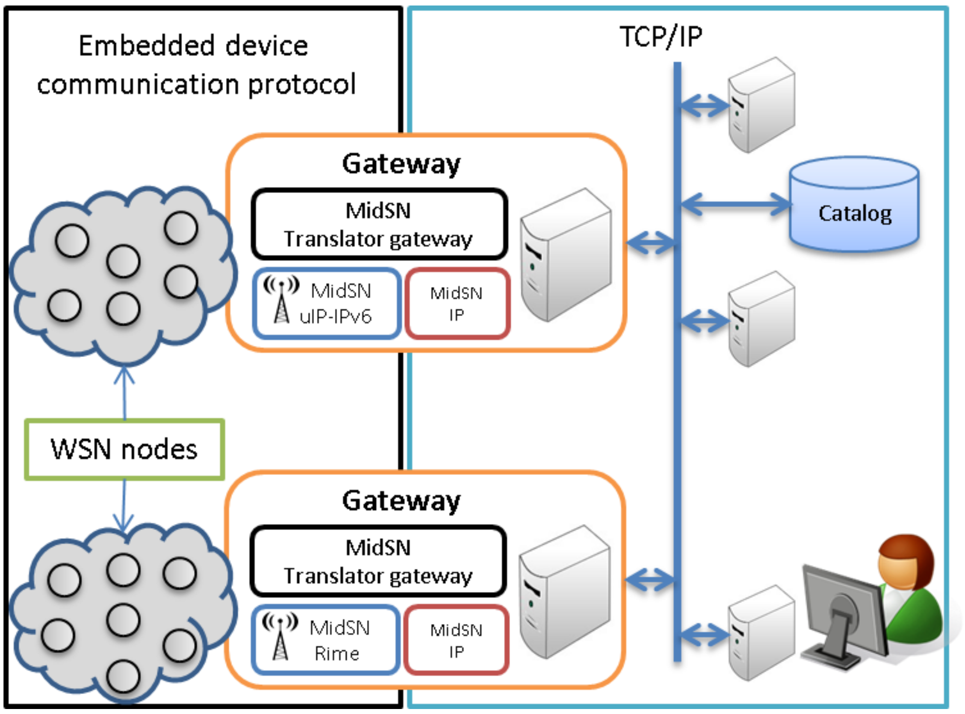


Figure 5.5 – MidSN Gateway component

Each gateway has a Translator module, which is an application level protocol translator. It receives the message at application level, looks up the destination address in the Catalog for translation, and resubmits the message using the communication function on the other side. Figure 5.6 shows how the addressing mechanism works with an example concerning a control station sending a command to a node.

The Catalog identifies nodes connected to a gateway through a specific tag (<Gateway>). If a node has IP support and can be directly addressed, the control station sends commands directly to the node. Otherwise, commands are sent through a gateway, which does the translation of addresses, if needed, and forwards the command to the target node.

The data messages sent by nodes connected to a gateway are sent to the gateway, which does the translation between protocols and addresses to forward it to the target node. Messages sent between any two nodes with the same communication protocol and in the same network are sent directly.

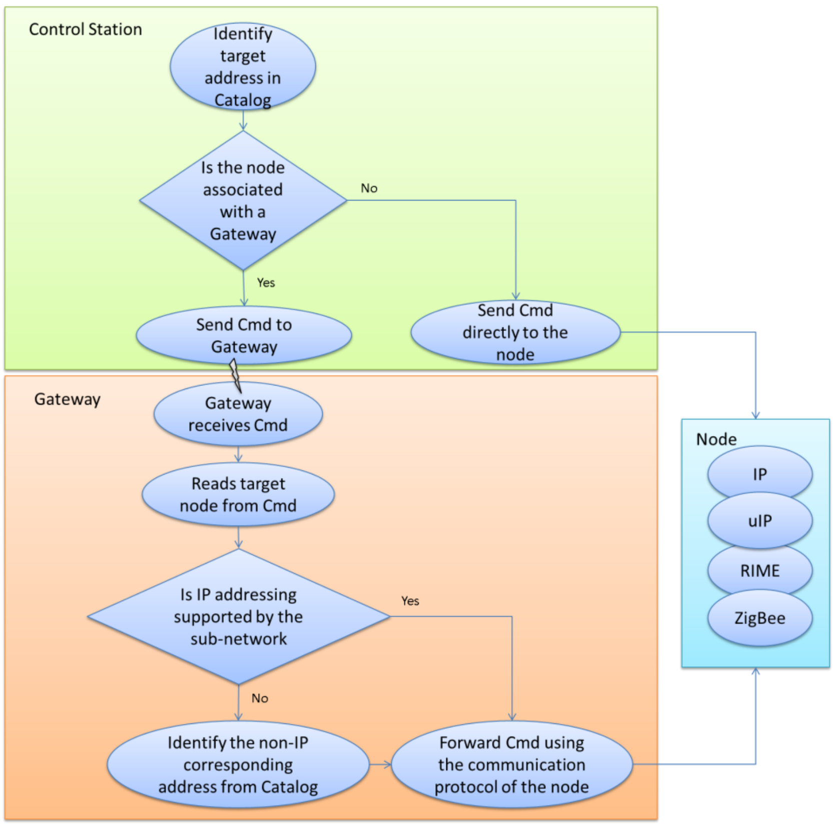

Control Station
Identify target address in Catalog
Is the node associated with a Gateway
No
Yes
Send Cmd to Gateway
Send Cmd directly to the node
Gateway
Gateway receives Cmd
Reads target node from Cmd
Is IP addressing supported by the sub-network
Yes
No
Identify the non-IP corresponding address from Catalog
Forward Cmd using the communication protocol of the node
Node
IP
uIP
RIME
ZigBee

Figure 5.6 – Flowchart of MidSN Gateway component

The example of Catalog shown in Appendix B includes one gateway and three nodes. Two of them are connected to the gateway while the third is an IP node that is connected directly to the heterogeneous sensor network. The gateway is identified by the id *net1* and has the address *10.3.3.82*. This gateway has two interfaces, one used by the IP side of the network, and other one used by the WSN sub-network. In this example, analysing the *iWSN_protocol* tag, we conclude that this gateway translates messages from IP protocols to Rime protocol and vice-versa.

As shown in the figure, from the WSN sub-network perspective, this gateway has the address *0x0000* and uses channel 20 to communicate with the nodes connected to it.

Each node is identified by a unique id (e.g. *1.1* and *1.2*) within a sub-network connected to the gateway, but also has a global IP address (e.g. 10.3.3.101, 10.3.3.102 for the same two nodes).

Depending of the communication protocol, the Catalog may include information concerning communication channel. This field is specific for WSN sub-networks.

5.5. Publish/Subscribe External Interface

The MidSN architecture includes publish/subscribe mechanisms to publish data stream content to external applications. Subscribers are users who request receiving data from a stream of interest to them. In this publish/subscribe context, external consumers are referred as subscribers, while the MidSN publishing mechanism is referred as publisher.

Users can subscribe to the data stream through the MidSN-RConfig component. Each subscription is represented by a “subscribe” API call, which includes the subscriber address, port, a connection timeout, and the stream source. Once receiving the call, the MidSN-RConfig configures the MidSN P/S module (Figure 5.2), which will allocate a unique id to the subscriber. At the same time, a profile with address and port will be created for the subscriber. The MidSN P/S module will use this information to establish/maintain the connection to the subscriber. The stream data will be published to the subscriber as soon as possible.

The publish/subscribe mechanisms can be divided into two parts: a subscribe/unsubscribe and a publisher. Figure 5.7 shows the flowchart of the subscribe/unsubscribe mechanism while Figure 5.8 shows the flowchart of the publishing mechanism.

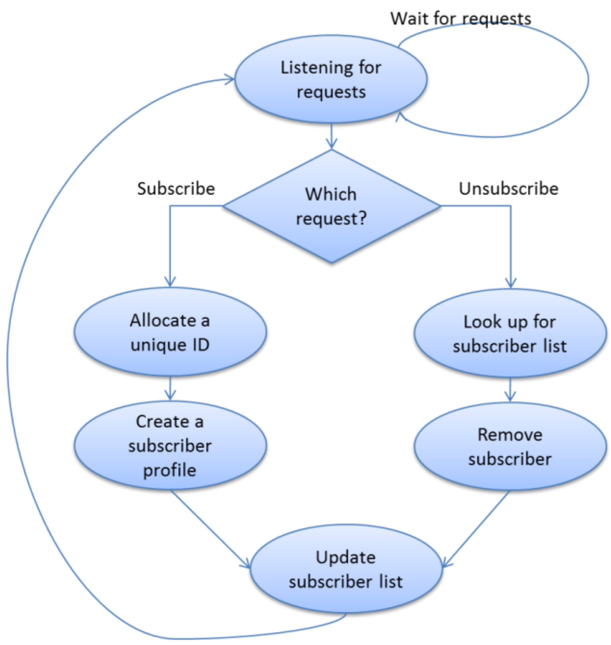


Figure 5.7 – Flowchart of MidSN-Subscriber mechanism

The subscribe/unsubscribe consists on a mechanism that listens to requests. When a request arrives, it is analysed to identify the request type (subscribe or unsubscribe). If the request corresponds to a subscription, the subscribe/unsubscribe mechanism allocates a unique id to the request, creates a memory structure where id, address, port, timeout and subscription data stream ids are stored, and lastly, updates the subscriber list. This list is shared with the publisher and contains all active subscriptions.

The publisher is continuously listening for data streams and checking if any timeout associated with connections to subscribers has expired.

When a data stream is received, the publisher looks up for subscribers and selects one by one. Next, the connection with the subscriber is checked. If there is a connection, data is sent. Otherwise, the publisher opens a connection to the subscriber. When the list of subscribers is empty, the publisher returns to the listening state.

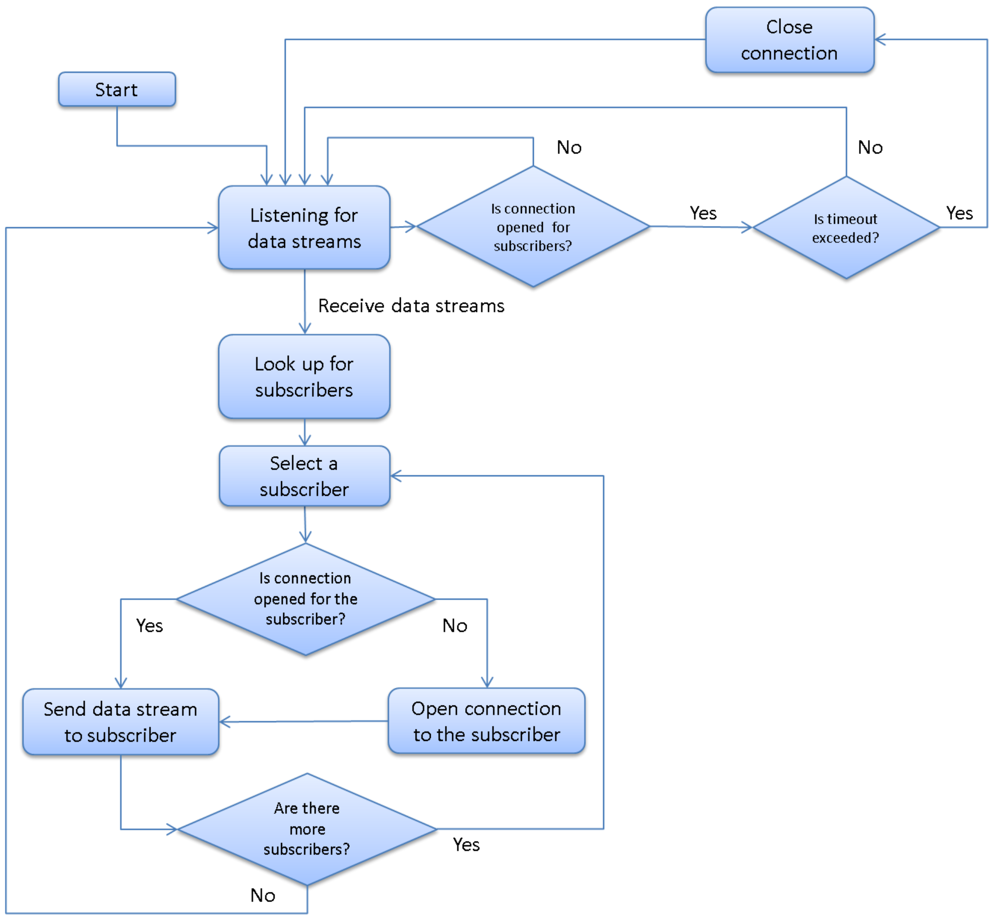

Close connection
Start
No
No
Listening for data streams
Is connection opened for subscribers?
Yes
Is timeout exceeded?
Yes
Receive data streams
Look up for subscribers
Select a subscriber
Is connection opened for the subscriber?
Yes
No
Send data stream to subscriber
Open connection to the subscriber
Are there more subscribers?
Yes
No

Figure 5.8 – Flowchart of MidSN-Publish mechanism

5.6. Data and Processing Model

The data and processing model of MidSN aims at offering flexibility in configuration and processing over the heterogeneous sensor networks.

MidSN implements a query processor. Based on queries formulated by users, MidSN parses and transforms them into logical configuration commands. These commands consist of high-level representations of the operations that need to be executed to obtain answers to the query. There are three types of queries:

- Queries that configure nodes to produce tuples (readings and data statistic results) continuously or once;
- Queries that configure nodes to receive tuples;
- Queries that configure inner operators to receive and produce tuples (e.g. Alarms and closed-loops).

A query must configure at least one source (a node producing the data), one or several operators to be executed in the source node, and clients, which are the nodes that receive the data. Each client must register to a publish/subscribe mechanism to receive tuples from the source node. Using this approach, each node can have a number of subscribers that get result tuples from one query.

Based on queries, MidSN uses a stream model to manage configurations and data. A stream is the query metadata and the data being produced (Figure 5.9).

meta-data
Data

Figure 5.9 – Stream structure

The MidSN metadata structure includes the following parameters:

- Id – a unique identifier.
- Processing period – is the time interval between executions. It can assume any value in milliseconds that are converted to internal time units of node (e.g.: MIPs for computer nodes or ticks for TelosB nodes).
- Window size – represents the number of samples that are considered when in-network processing techniques are applied (e.g. average, maximum, minimum, percentile).
- Number of measures, number of clauses and number of actions – indicates how many measures, condition clauses and actions are included in message.
- Measures – specifies each value that must be included in the result stream. This field assumes a single measure or a set of measures, depending on the number of measures indicated in the above field. Each measure is composed by two fields: data source and metric. The data source indicates which stream or sensor should be the source of the data for the stream (e.g. temperature, humidity, light, ADC0, and so on) and metrics indicate in-network data processing (e.g. the value itself (no in-network processing), an average, a sum, a standard deviation, and so on).

- Clauses (where Clauses) – represents a condition that should be checked for restricting the tuple that are included in the output or those that trigger an action. This field is composed by one or more expressions connected by AND or OR, where each expression has five fields (data source, metric, operator, data source, metric). The operator can assume one of the following values: >, <, =, >=, <=.

  This format allows users to define expressions such as (avg (temperature) > 25).

- Actions – represents the action or actions that must be executed if all conditions are true. This field assumes one or a combination of the following actions:
  - Send – send the result stream to other node or nodes in the network; the field is composed by a send identifier (`SEND_RESULT`) and one or more node destinations.
  - Actuate – write a value to an actuator; the field is composed by an actuator identifier and a value that must be written to the hardware.
  - Execute – Executes a specific operation. For instance, start or stop a stream.

Concerning data, each stream may contain data from sensor acquisitions, data coming from other nodes for a specific stream and data resulting from in-network processing.

The data is stored in circular arrays of values with configurable sizes. Each position has values as configured in the metadata. These may come from sensors, node parameters or other streams. The use of a circular array means that old readings, after the remaining window size has been filled, are overwritten by new readings.

5.7. Operations

Operations implement periodic data processing over data using the stream processing model. It is controlled through timer events. When the time comes for an event, the processor manages the operations applied to the stream (data insert, alarm detection, transformation), eventually the insertion of data in the stream or composition of output streams, and triggers actions if some conditions are met.

When a timer event arrives, the processor processes whatever are the event specifics and reschedules the timer for the next period. There may be several simultaneous timer events in a node.

Figure 5.10 illustrates an operation using streams with an example. In that figure, nodes were configured to collect, transform and deliver data into a control station for a visualization application. A signal sample is collected periodically by each of three sensors, placed into a stream and routed into a relaying node. The relaying node computes some statistics (e.g. avg or max) over the three values and forwards the resulting value to the control station.

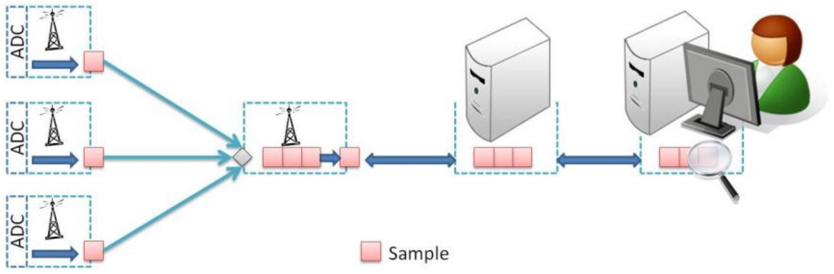
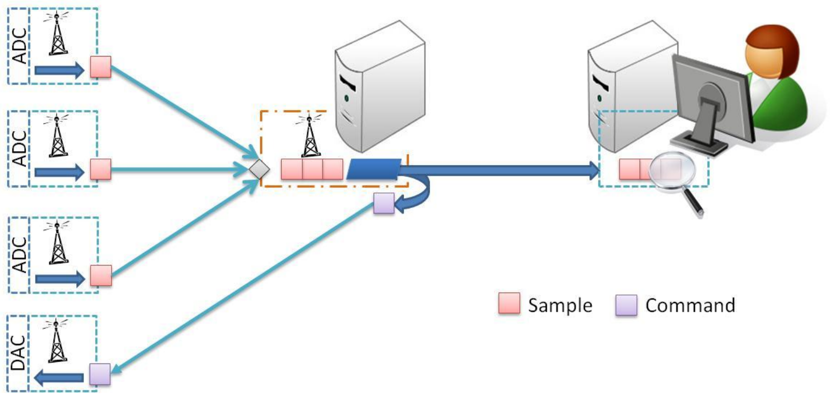


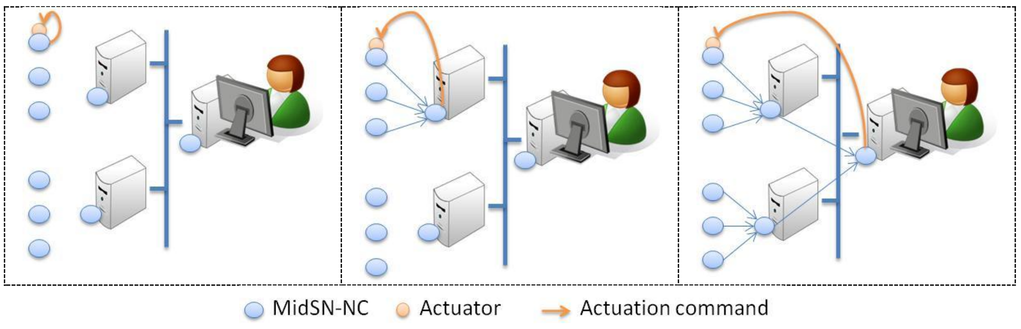
Figure 5.10 – Stream processing model

The control station keeps a stream with the last values that were received, and publish it. A client application subscribes to the data and plots it to allow visualization of the values. It would be possible to configure differently any of the nodes in this illustrative example. For instance, sensor nodes could each compute some statistics over 10 samples and forward those to the control station directly, which would show the stream values or compute some other statistics.

Using other configuration and processing capabilities, MidSN also supports closed-loop control. Figure 5.11 illustrates how closed-loop control is achieved using the same data processing model. In Figure 5.11a) sensors send values into a node that computes an actuation value and forwards an actuation value into an actuator node. Figure 5.11b) shows three alternative actuation scenarios, which are configured with simple commands issued by the MidSN-RConfig component. The scenarios include actuation through the control station, computer nodes, WSN-sink nodes or in a single WSN sensor and actuator node.



(a) Closing the loop



(b) Illustrating three Closed-Loop scenarios

Figure 5.11 – Illustration of closed-loop control

5.8. *User API*

MidSN includes an API which provides a set of functionalities used to interact and configure nodes remotely for operation in a distributed network. It allows external applications to submit configuration commands and to subscribe to data (streams) coming from nodes.

The API can be developed using, for instance, Web Services, REST or HTTP to easily interface with external client applications.

We list next a set of capabilities that are useful in various application contexts, but other configuration and operation functionalities can be added depending on the specific target context for the system or the amount of functionality that is desired.

The set of functionalities that we consider is:

- Node
  - Activate / deactivate nodes;
  - Activate / deactivate sensors and actuators connected to each node;
  - Request node status;
  - Reset a node;
- Operations and Filters
  - Create periodic operations (gather sensed value data with a sampling rate);
  - Drop periodic operations from nodes;
  - Start and stop operation;
  - Change operation periodicity (change the sampling rate);
  - Create data filter (where conditions);
  - Drop filter;
- Alarms
  - Create an alarm;
  - Delete an alarm;
  - Start and Stop alarm verification;
- Actions
  - Create action with periodicity;
  - Drop action;
- Actuation
  - Send and apply actuation value;
- Publish/Subscribe

    - Subscribe data
    - Unsubscribe data
- Agents
    - Send a custom agent code to nodes;
    - Load an agent;
    - Start and stop an agent;
    - Drop an agent;

These functionalities are categorized in seven categories: node, operations and filters, alarms, actions, actuations, publish/subscribe and agents. In Appendix C we describe each category and its calls.

This API is extensible and can include features to fit different application contexts. It is possible to define more powerful APIs and consequently larger MidSN components in more computational powerful nodes (e.g. computers), but this important subset of functionalities should be present in every node.

Chapter 6

Node and Configuration Middleware Components

In this chapter we detail the node (MidSN-NC) and remote configuration (MidSN-RConfig) components, which implement the mechanisms proposed in the previous chapter.

MidSN-NC is the node component that provides uniform stream-based configuration, processing and communication for nodes with different characteristics in the heterogeneous sensor. The MidSN-NC architecture builds an intermediate computing layer, which serves as an abstraction hiding different hardware and operating system implementations. It is detailed in Section 6.1, 6.2 and 6.3 and it was designed to meet all requirements described in Chapter 4 and implements the mechanisms described in Chapter 5.

The remote configuration component (MidSN-RConfig) is the MidSN component which allows applications and users to configure any node remotely, by submitting simple API commands. It sends configuration commands to nodes with combination of measures, conditions and actuations, which are used to enable easy remote configuration of system parts during the system lifetime. In Section 6.4 we describe MidSN-RConfig in detail. The message structure used to exchange commands and data between nodes is described in Appendix E.

MidSN architecture assumes that it is also possible to load custom code (agents) for handling more specific operations (e.g. a closed-loop controller). In Section 6.5 we describe that capability.

6.1. Node Component Architecture

The MidSN node component (MidSN-NC) is a stream-based operating machine that manages data and operations inside the node. It must be developed only once for each operating-system and offers functionality to allow any node to be configured remotely for the same operations.

MidSN-NC can be developed for hardware with minimum set of requirements. The minimum requirements to run it are: a programmable micro-controller; a communication stack for data exchange (wired or wireless); enough ROM memory to hold the MidSN-NC software component. The amount of RAM needed is configurable, since it depends on the number of operation-related structures that are allowed. Sensing nodes must include ADCs to connect to sensors and actuators with DACs to interface with external analog hardware. Figure 6.1 shows the MidSN-NC architecture.

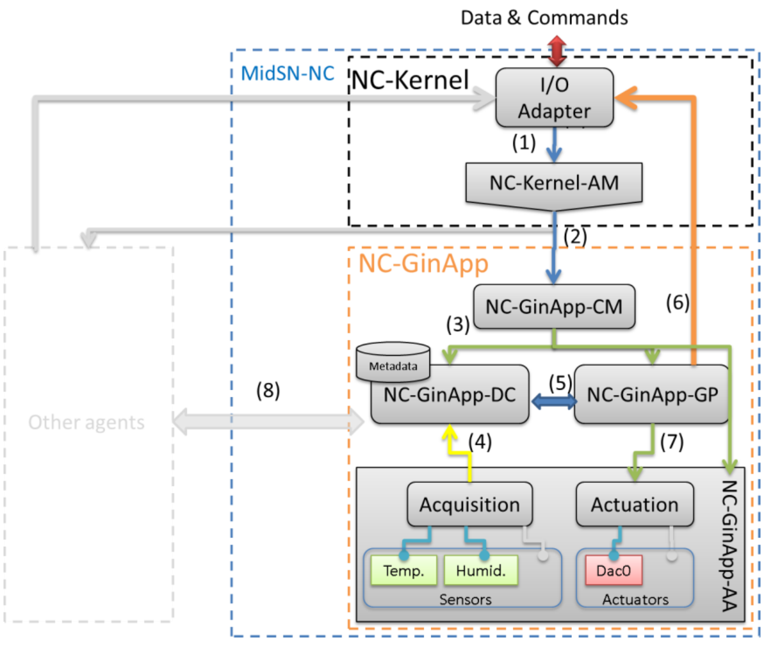


Figure 6.1 – MidSN-NC architecture

The MidSN-NC runs at application level and is composed of two main parts:

1. A **kernel** (NC-kernel) that is responsible for exchanging messages with any other node in the system and for managing agents. An agent is a code to execute specific functionality that is sent to the node;
2. A **small operating machine** (NC-GinApp) that provides:
    a. **Configuration management (NC-GinApp-CM)**, whereby nodes will be able to configure themselves based on commands provided by other nodes or servers. With this component, nodes can be configured to, for instance, raise alarms, issue actuations or compute measures such as averages or maximum values, as well as sending data stream to other nodes;
    b. **Data collector and processor capabilities (NC-GinApp-DC and NC-GinApp-GP)**, whereby the node will be able to look and compute on data it collects from either sensors or other nodes, to take decisions and to route data;
    c. **Acquisition and actuation capabilities (NC-GinApp-AA)**, whereby sensor nodes will be able to periodically acquire sensing values or issue actuation commands (e.g. WSN motes connected to wired analogue sensors through DACs and ADCs).

Figure 6.1 also shows the flows/interactions between components. Incoming messages are delivered by the I/O adapter to the Agent Manager module (flow (1)). This module will analyse the message and send it to the target agent. Each message must contain a target agent identifier field, which identifies the target agent.

If GinApp is the target agent (flow (2), then the NC-GinApp-CM module is called. This module will analyse the message and choose two possible operations:

1) If the message is a configuration command, it will configure some functionality of the node (flow (3));

2) If the message is a data message, it will send the data to the destination module (NC-GinApp-DC, NC-GinApp-GP or both) (flow (3)).

Flow (4) represents sensed data going to the NC-GinApp-DC module, while flow (7) represents actuation commands (actuator and value) going to the NC-GinApp-AA module. Actuation commands can be issued by the NC-GinApp-CM module directly or can be the result of the processing. For instance, if a closed-loop control is configured, after the NC-GinApp-GP processes data coming from either the NC-GinApp-DC or from other node(s), it can send an actuation to the NC-GinApp-AA module.

Flow (5) represents data streams going/coming to/from the NC-GinApp-GP module. These interactions can be to get data from sensors streams to be used on processing or to store stream results on the NC-GinApp-DC module.

Flow (6) represents data that is to be sent to other nodes (e.g. a stream periodically sends data from the node to another node).

The MidSN-NC component is flexible in the sense that it allows adding small agents to execute specific tasks. Custom coded agents can also use GinApp functionality. Flow (8) represents the interaction that can be done between any agent and GinApp. These interactions are done by directly calling a GinApp API.

6.2. NC-Kernel

Driven by the quest to enable flexibility and extensibility with a clear abstraction, MidSN-NC includes a kernel which manages communication between nodes. It also establishes communication with agents inside a node. This part of the node component is composed by two modules, communication module and agent manager.

6.2.1. Communication (I/O Adapter)

This module is responsible for managing all network traffic that flows in and out of the node. The module is able to publish data streams to other nodes or outside clients. Through command messages to the NC-GinApp-CM module, the node can be configured to send one or more data streams periodically (or alarms) to other nodes and to outside applications.

Any outside application that subscribes to streams (and/or alarms) receives the data in a port that was specified by it in the subscription call.

6.2.2. Agent manager (NC-Kernel-AM)

For all incoming packets, they are validated and parsed to assert their functionality or agent destination (*agent_dest*). The Agent Manager module (NC-Kernel-AM) stores information about each agent and forward messages for the correct target agent.

NC-Kernel-AM also includes functionalities to receive agents over-the-air from users, store them in flash memory or drop them from the node. Upon receiving an agent, it is stored on persistent memory (e.g. flash memory). NC-Kernel-AM offers support to load and unload agents to or from main memory, as well as functionalities to start and stop it. The start function can be called with parameters that may be used during the execution of the agent.

6.3. NC-GinApp

GinApp is a small stream operating machine that manages data and operations inside nodes. GinApp implements a modular approach for acquisition, actuation and data processing. GinApp also provides remote configuration or reconfiguration functionalities, to fit a wide range of application scenarios.

6.3.1. Acquisition & Actuation (NC-GinApp-AA)

This module must be present in nodes doing sensing and/or actuation. It performs all sensor actions required to gather data from sensors and all actuation actions necessary to actuate over the hardware. Commands are issued to configure the modules, for instance to activate or deactivate sensors or actuators. The NC-GinApp-AA module is based on drivers that do the interface between MidSN-NC and hardware. To add a new sensor or actuator, a programmer needs to develop a specific driver and link it to the MidSN-NC for installation in nodes. Depending on whether it is a sensor or an actuator, the programmer of a new sensor or actuator must add the specific sensor or actuator management code to the methods described in Table 6.1.

Table 6.1 – Sensor and actuator driver primitives

Type	Functionality	Primitive
Sensor	Initialize Sensor	`boolean res = init( SENSOR_NAME, <list of parameters> );`
	Read Sensor	`int value = readSensor( SENSOR_NAME);`
Actuator	Initialize Actuator	`boolean res = init( ACTUATOR _NAME, <list of parameters> );`
	Write to hardware	`boolean res =  writeToHW( ACTUATOR_NAME, int VALUE );`

6.3.2. Configuration management (NC-GinApp-CM)

The NC-GinApp-CM module, depicted in Figure 6.1, processes commands that configure and modify processing, acquisition and communication properties, and also manages data coming from other nodes. Command messages coming from Agent Manager Module are parsed to assert which actions should be taken and to execute them. Examples of actions include creating a stream, deleting a stream, starting or stop a sensor or an actuator.

A parser identifies the type of the message, which may be either a configuration command or a data message.

Data message – If the message is a data message (a data stream), then the data is forwarded to the Data Collector module for storage and operation.

Configuration command – If the message is a configuration command, then the node is configured to operate accordingly. There are two major configuration command types:

a) **Node Operation**: commands that configure hardware functionalities, such as activating a sensor or an actuator, and actuation commands. Actuation commands are commands that order a node to write a value to an actuator. They can be used for either commanding some hardware device directly or for supervisor nodes to command actuation remotely or locally (e.g. in closed-loop control).

b) **Configuration of periodical operations** – Streams, Alarms and Conditional Actuation: commands that configure individual data streams, alarms and condition-based actions to operate. The command includes a name for the periodical operation. When such a command arrives at a node, the configuration of the corresponding structure is updated or a new one is created if it does not exist yet.

Configuration of periodical operations sets up operation structures, that is, structures that keep information for timer events to know what to do when they are triggered. Those structures keep the following information:

- **Structure type** – whether the structure is a data stream, an alarm or a conditional periodic actuation;
- **Data destination** – address of destination nodes for the data that is output periodically by the stream;
- **Sensor acquisitions and periodicity** – when specified, this field indicates which sensor(s) is (are) to be acquired in order to fill the stream, and with what periodicity. As a consequence, the module also configures a timer event related to the acquisition;
- **Send expressions (operations and actuations)** – specifies the values the stream should retrieve, the operations over these values and sending them to other nodes requesting the data. We also call this selection a “select”, since it identifies which data is selected, operated and sent. As a consequence, the module also schedules a timer event related to the periodicity for operating and sending the stream data to other nodes;
- **Filter** – a condition that restrict (filters) the stream data. So that only the tuples conforming to the condition will be further operated upon. We also call a condition a “where”, since it corresponds to a where clause in an SQL statement;
- **Stream window size** – configures the stream data window size over which to operate, such as computing an average or a maximum over a window of size 10;

- **Alarm condition (where) and periodicity** – configures the threshold and condition for an alarm, and the corresponding condition evaluation periodicity. The condition is tested periodically and an alarm message is sent to specified nodes when an alarm is raised;
- **Actuation based on a condition (where)** – configures an actuation based on a condition. The command specifies a periodicity, a threshold, a condition for the actuation, and also an actuation value. The condition is tested periodically and an actuation done if the condition is true. This command also schedules a timer event for the periodical conditional actuation;

Closed-Loop Control operations are also possible in our architecture. It is configured by commanding periodical operation in a set of nodes: sensor nodes are configured to acquire data, supervision nodes are configured to receive that data (a stream), feeding it to a controller, and the controller sends actuation commands to nodes that, upon receiving the command, will perform the corresponding actuation. Closed-loops based on simple thresholds can implement the controller as a simple filter condition over the stream. More complex user-coded controllers can also be loaded.

6.3.3. Data Collector (NC-GinApp-DC)

The NC-GinApp-DC module manages data readings collected by sensors or received from other nodes and stores them in memory. The memory available for each data reading type is limited by a window size parameter specified during creation of the stream. The module uses a circular window, which means that old readings, after the remaining window size has been filled, are overwritten by new readings (the data is expected to have been consumed already when it is overwritten). This module can also store the stream in persistent storage in embedded devices with such capabilities.

NC-GinApp-DC offers an API that provides functionalities to create and drop sensor streams, save data into streams and read data from sensor streams. Table 6.2 shows the primitives developed to manage data in the NC-GinApp-DC module.

The creation of streams in NC-GinApp-DC module is done by calling the *create stream* method. This method requires four parameters:

- Stream ID, used as a unique identifier;
- Windows size, used to limit the amount of memory used by each array in the stream;
- Place to store, used to indicate if the stream is stored in main memory or in flash memory, as a file.

To read data from streams, NC-GinApp-DC offers the readDataFromStream method. This method requires the stream identifier and the number of samples that must be returned. The returned data correspond to N last samples stored into the stream. Each sample is stored using the writeToStream method. Lastly, there is a drop method that allows deleting the stream structure from memory.

Table 6.2 – Data collector API

Functionality	**Primitive**
Create Stream	`boolean result = createStream( int streamID, int windowSize, Boolean placeToStore, );`
Drop Stream	`boolean result = dropStream( int streamID );`
Read data from stream	`list <int value> = readDataFromStream( int streamID, int Nsamples );`
Store data into stream	`boolean result =  writeToStream( int streamID, <List VALUES> );`

6.3.4. Gin Processor (NC-GinApp-GP)

The NC-GinApp-GP module is triggered by a timer or network event and uses stream structure fields to determine how to process configurations.

The stream structure fields are used for:

- Selecting which data and which operations to use (computations over data, such as an average of the three last readings);

- Specifying which conditions should be tested against the data (conditional processing).

NC-GinApp-GP uses sensor data readings, data coming from other nodes or both to compose the stream output to send to other nodes.

Internally, NC-GinApp-GP manages the data processing mechanisms and periodic functionalities. It implements a time scheduler, where periodic events can be registered. The periodicity of each event depends on the user configurations.

When a timer event is triggered, it is processed as depicted in Figure 6.2.

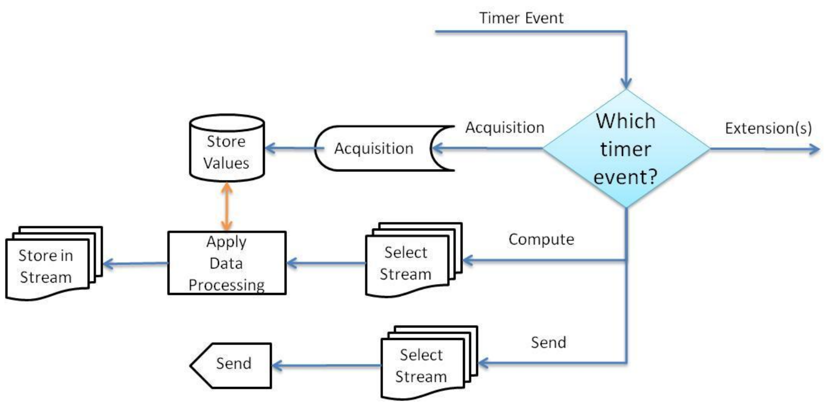


Figure 6.2 – Timer events flowchart

There are three main types of timer events, corresponding to the flows in Figure 6.2:

- **Acquisition**: triggers acquisition of sensor signals; For instance, when an acquisition event arrives, the NC-GinApp-AA module is instructed to sample the physical sensors and to send the value to the NC-GinApp-DC module to store it in the corresponding stream.
- **Computation**: computes from data that is in the NC-GinApp-DC module. For instance, to compute averages or maximum values, to filter data, to raise alarms, to merge data from streams. The computation operations require a selection of a stream to be processed.
- **Send**: sends data (streams or alarms) to other nodes or external applications. Typically, after computation of a stream, a sending event is

generated to send the result to another node. However the sending event can be decoupled from the computation event. So, when a sending event occurs, the node selects the corresponding stream and sends it to the destination node.

The NC-GinApp-GP has capabilities to do in-network processing. These capabilities are called when the computation event manifests itself. NC-GinApp-GP allows aggregating over any data (e.g. computing averages, sums, max, min, count) or just sending values without further processing.

Figure 6.3 shows the computation flowchart used by the NC-GinApp-GP. When a computation event arrives, the processor starts by the selection of a measure configured in the stream metadata. For each measure, the processor looks up the last *n* values according to the window size defined in the metadata, and applies where conditions (filters) to evaluate if values are considered to compute or to include in the output stream.

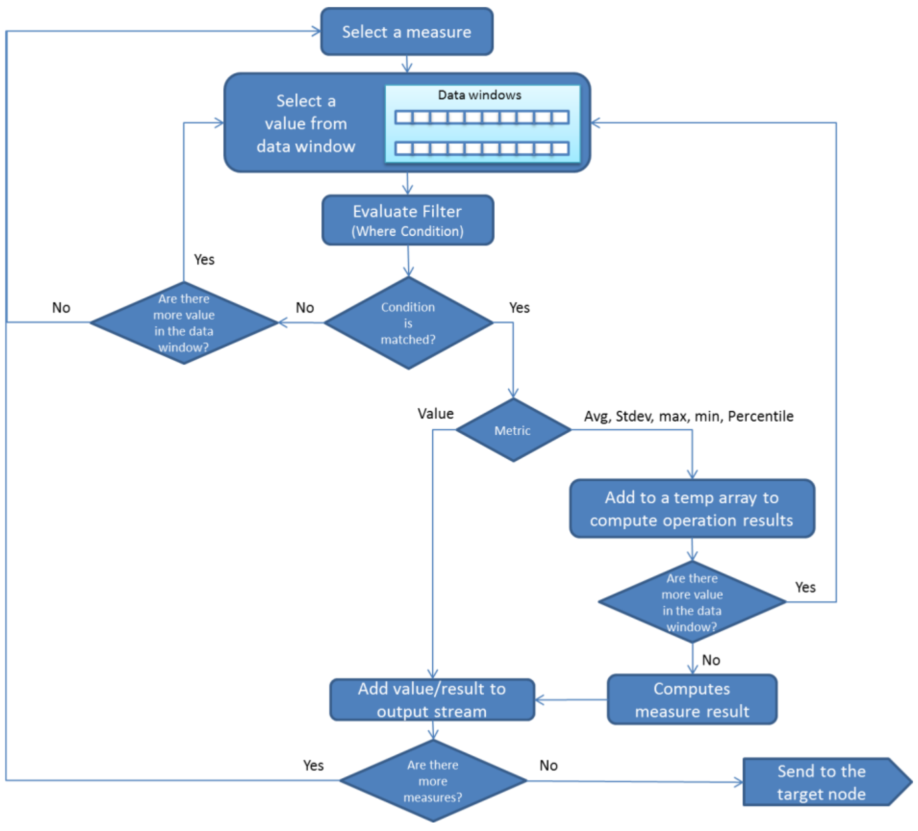

Select a measure
Select a value from data window
Data windows
Evaluate Filter (Where Condition)
Yes
No
Are there more value in the data window?
No
Condition is matched?
Yes
Value
Metric
Avg, Stdev, max, min, Percentile
Add to a temp array to compute operation results
Are there more value in the data window?
Yes
No
Add value/result to output stream
Computes measure result
Yes
Are there more measures?
No
Send to the target node

Figure 6.3 – NC-GinApp-GP – computation flowchart

Since NC-GinApp-GP can apply in-network computations, after verifying the filter condition and if aggregations are defined in the stream metadata, the processor stores the evaluated values in a temporary array that is used to compute the result, upon verifying all *n* values.

After evaluating a measure and getting the result, the processor adds the result to the stream output and re-runs again with another measure.

The number of measures used in each stream depends of its configuration. Upon evaluating all measures, the stream output is ready to be sent to the target node. In this case, the processor generates a sending event that will dispatch the stream output to the I/O adapter (Figure 6.1).

6.3.5. Extensibility of NC-GinApp

GinApp was designed to be an extensible architecture, where there are a basic set of configuration commands and operations, but it is possible to add other types of commands and operations to fit the context requirements. Users can extend GinApp by adding functionality to modules in the system. These functionalities are hand-programmed parts that are added to the base MidSN-NC code.

The API of MidSN is also extensible and can include features to fit different application contexts and functionalities added by users. For instance, assuming a new controller added by a user, the corresponding API call is needed to remotely configure the controller.

6.4. Remote Configuration Component (MidSN-RConfig)

The remote configuration component (MidSN-RConfig), depicted in Figure 6.4, is the MidSN component which allows applications and users to configure any node remotely, by submitting simple API commands. It allows configuration in any heterogeneous sensor network without requiring any programming.

MidSN-RConfig is constructed as a set of modules that deals with configuration commands, which are submitted via API calls, and translates them into commands. The commands are then sent as messages to any destination node. Nodes reply with an acknowledgment as soon as they apply the command.

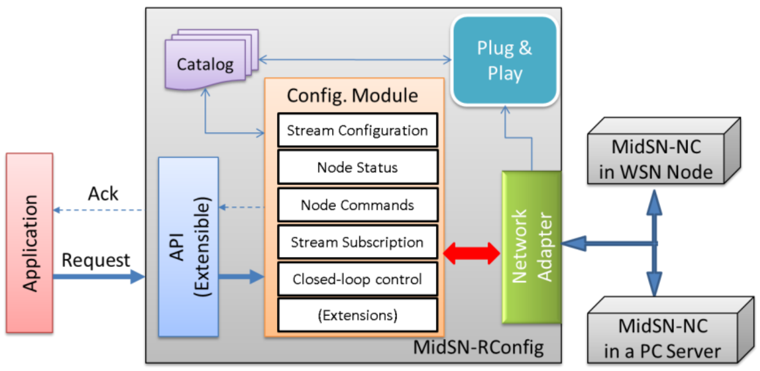


Figure 6.4 – MidSN-RConfig modules

MidSN-RConfig is composed by four main modules and the Catalog: an API, a configuration module, a network adapter and a plug&play module, plus a Catalog to hold all information concerning nodes and configurations.

The API provides functions for external applications to submit configuration commands and to subscribe to data (streams) coming from nodes. The API can be developed using, for instance, Web Services, REST or HTTP to easily interface with external client applications.

The **Configuration Module** is responsible for handling the configuration API calls and for configuring the heterogeneous sensor network. It is composed by a set of functions that deal with stream configurations, network status, node commands, stream subscriptions and closed-loop control.

Stream configuration: allows creating, removing or changing streams in any node or group of nodes in the network.

Sensor network status: collects configurations and node status information (e.g. battery, packet losses, whether a node is alive, what is running in a node).

Node commands: allows sending commands to nodes.

Data streams subscription: allows client applications to subscribe to data streams.

Closed-loop control: this functionality allows engineers to configure closed-loop control in the heterogeneous sensor network with WSN sub-networks. It configures:

- Any node with MidSN-NC to collect data from sensors;
- Any node in the system to collect data from other nodes and to apply a condition or controller, resulting in an actuation value;
- Any node in the system to receive actuation values and to actuate.

The **Network Adapter** (NA) is an interface between the MidSN-RConfig component and the network communication infrastructure, which allows communication with nodes (computers or embedded devices). This module implements and abstracts network protocols needed to be able to communicate with all nodes in the network.

The **Plug&Play** module (P&P) is a component that adds new nodes to the network. To join, a new node will need to install MidSN-NC and set a valid address. There are four alternatives for installation and address creation:

1) Manual MidSN-NC Installation: users can download the MidSN-NC release for the specific platform and operating system, install it and run it.
2) Plug & Play MidSN-NC installation for embedded devices with serial interface: the P&P module has a daemon that is listening on the serial ports to detect when a new embedded node is connected to serial port. When that occurs, the P&P asks the user about the hardware platform and software drivers, and installs the appropriate version of MidSN-NC into the node.
3) Addressing for non-IP embedded devices: in this case, after concluding installation of MidSN-NC into the node, P&P asks the user which gateway heads the sub-network where the node will be placed. Nodes that do not support IP protocol must be connected to a gateway.

4) Discovery-based address configuration: P&P also includes a daemon to listen to new IP connections. When this occurs, the module updates the Catalog to add the new IP.

Lastly, the **Catalog** stores addresses (global IP address and proprietary communication protocol address), configurations and node status. This module was described in detail in Section 5.3.

Figure 6.4 also shows the main flows between modules. An API call request is received by the API module, which sends it to the Config. Module. That module creates the corresponding command message, updates the Catalog with the new configuration and sends the configuration message to the network adapter. The network adapter will send the command to the target nodes. When a node receives a command, an acknowledgement is generated and it is returned to the API submitter to indicate that a command was received by the node. A second acknowledgement is generated and returned to the user when a command is done.

6.5. Custom Code Agents

Custom code refers to loading new code for specific functionality, for instance, a closed-loop controller applying a specific algorithm.

As reviewed in Chapter 2, there are many approaches in the literature that load full binary images into sensor nodes. Our architecture also allows users to add any customized code, but instead of replacing the whole node software image when a new functionality is needed, MidSN provides agent-based upgrades. These upgrades consist on a run-time loader and linker for new agents (functionalities). MidSN uses the Executable and Linkable Format (ELF) [154] files to specify code objects used for dynamic linking.

ELF is a standard format for object files and executable that is used in most modern Unix-like systems. An ELF object file includes both program code and data and additional information such as a symbol table, the names of all external unresolved symbols, and relocation tables. The relocation tables are used to locate the program code and data at other places in memory than those for which the object code was assembled originally. Additionally, ELF files can hold debugging information such as the line numbers corresponding to specific machine code instructions, and file names of the source files used when producing the ELF object.

MidSN explores the ELF objects to run agents that can be developed for specific needs. Each of these agents is developed by users and can interact with the small operating machine (NC-GinApp) included in MidSN-NC. Those interactions are done through configuration and operation commands (API) which allow configuring data operations and managing data in NC-GinApp.

For example, users can develop a specific closed-loop algorithm where the data input may be provided by NC-GinApp (data streams coming from other nodes or sensor streams with sensor readings). The output of this algorithm can issue an actuation command through NC-GinApp. Figure 6.5 represent a diagram of this example.

Figure 6.5 shows the command and data flows. Flow (1) represents API calls to configure MidSN-NC. In this example, that flow occurs to configure the data sent to the agent and to actuate over an actuator. Flow (2) represents the data streams that are sent by NC-GinApp to the agent.

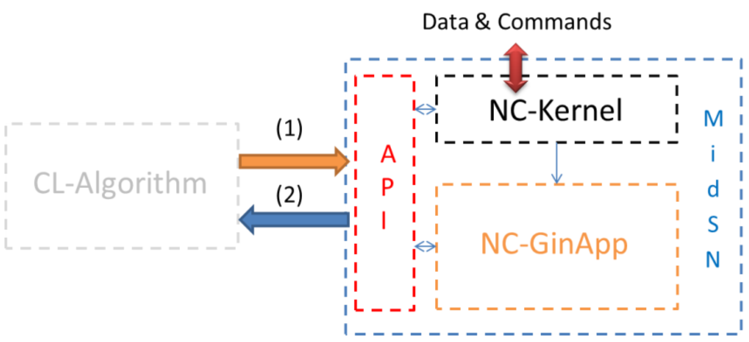


Figure 6.5 – Interactions between an agent and MidSN-NC

The agent upload is similar to full image upload, but instead of rebooting a node when the upload is completed, MidSN continues execution and users can issue specific commands to load an agent from external flash to main memory and to start and stop its execution. These commands are offered by the API of remote configuration component.

So, the full binary images approach has the following restrictions when compared to our proposal:

- The whole application code must be developed from scratch, where there are no predefined processing capabilities;
- All communication related with data and commands must be developed by hand;

Using the MidSN approach, the processing and communication capabilities are already available for use (NC-GinApp). The user/programmer only needs to call streams and command-level API interfaces to configure and use those capabilities. The uploading cost will also be reduced because the code image size is smaller (the OS and communication stack is already in the node).

Lastly, the MidSN custom code approach consists of a built-in agent-based paradigm with stream and command related API usage capabilities. This allows, for example, installing multiple agents in nodes and starting and stopping them selectively. As described before, MidSN-NC includes functionalities to start, stop, load and unload agents, as well as capabilities to store and drop agents from the node.

In Appendix D, we show the custom code for an agent that computes a closed-loop algorithm using the data collector module (NC-GinApp-DC) and the NC-GinApp-AA module.

Chapter 7

Network and Operations Planning

Some applications (e.g. industrial monitoring and control applications) require strict end-to-end operation timing guarantees. This chapter proposes an approach to plan for time guarantees over the middleware-ran heterogeneous distributed system. It discusses how to plan monitoring and closed-loop tasks with restricted time boundaries in the distributed heterogeneous system, assuming that the system has WSN sub-networks and that monitoring or control loop operations involve wireless sensor nodes.

When including wireless sensor networks in time critical applications, such as process control, they will be integrated in larger heterogeneous sensor and actuation networks composed by cabled networks, wireless sensor nodes, Programmable Logic Controllers (PLCs), computers and control stations. In that context, it is necessary to ensure that monitoring and control loop actuations happen within required time bounds.

Timing guarantees in WSN sub-networks are enforced using real-time algorithms, protocols and operating systems. In what concerns network protocols, pre-planned synchronous time-division algorithms are frequently used to enforce timing. But at the same time, operations timing requirements must be met over the whole heterogeneous system, regardless of what protocols and software is running in each part. The approach proposed in this chapter plans the network to guarantee operation timings. It schedules operations, predicts latencies and subdivides the wireless sensor network until the predicted latencies meet operation latency requirements.

Section 7.1 presents a typical distributed control system organization and Section 7.2 describes operations and requirements that can be defined over those systems.

Section 7.3 discusses the base latency model used to plan and estimate operation latency. Each part of latency is described, which allows us to understand which latencies are involved and why they are involved. Some of those latencies are due to resource constrained networks and other ones are due to wired and backbone networks. In Section 7.4 we describe how to integrate closed-loop operations in the base model and in Section 7.5 we describe how to characterize the latencies associated with non-real-time components.

Section 7.6 discusses the prediction model for maximum latency used to plan and estimate operation latency and Section 7.7 discusses the algorithm to plan networks with timing requirements. The approach is based in slot-based planning for the schedule-based parts of the system, plus operation time statistics for non-real-time parts. The approach dimensions the network to meet timing requirements. It allows predicting monitoring latency, latency of commands and latency of closed-loop operations. It also proposes an approach to reduce the latency of commands.

Timing issues determine the network size and layout. This layout determines a certain amount of energy consumption. Section 7.8 discusses energy consumption and lifetime prediction.

At the end of the chapter (Section 7.9) we discuss some considerations of slot size. In the planning algorithm we assume that the slot size is already defined by the communication protocol, but if a user wants to define a new size, we describe which timings should be considered to determine the correct size.

7.1. Organization of Distributed Control Systems with Wireless Sensors

A heterogeneous distributed control system is typically made of various sub-systems. Figure 7.1 shows an example with heterogeneous devices.

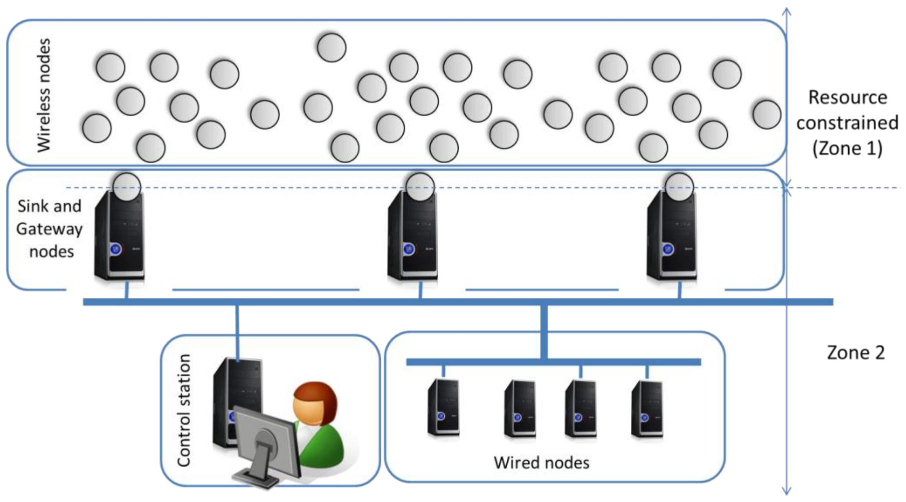


Figure 7.1 – Distributed control systems with heterogeneous devices together

It includes resource-constrained sensor nodes and more powerful nodes such as PLCs or computer nodes. We are assuming a distributed control system that includes wireless sensor nodes organized into wireless sensor sub-networks.

A TDMA protocol can be applied within each wireless sensor network to deliver a high degree of time predictability, while the rest of the distributed control system can be based on IP, FieldBus, ProfiBus, deviceNet over Controller Area Network (CAN) or other sub-networks.

TDMA protocols create a schedule or time frame for all the network activity: each node is assigned at least one slot in a time frame. The time frame is considered to be the number of slots required to get a packet from each source to the sink node. It is also called the epoch size (*E*). The schedule defined by the protocol allows latency to be predicted with some degree of accuracy.

Typically, nodes will send one message in their slot per epoch, which requires them to wait until the next epoch to send another message. If a very large WSN network was considered, the time to visit all nodes would be high and operation latency would be too high. To avoid this problem, network sizing is necessary, resulting in multiple smaller network partitions.

The TDMA protocol deployed in the wireless sensor sub-networks offers some degree of timing guarantees, while the timing characteristics of the remaining parts of the system must be characterized by statistical analysis based on long term observations.

Figure 7.2 shows an example of a TDMA epoch. In this example, the epoch has 1 second of length. It includes upstream slots, downstream slots, time synchronization and processing slots.

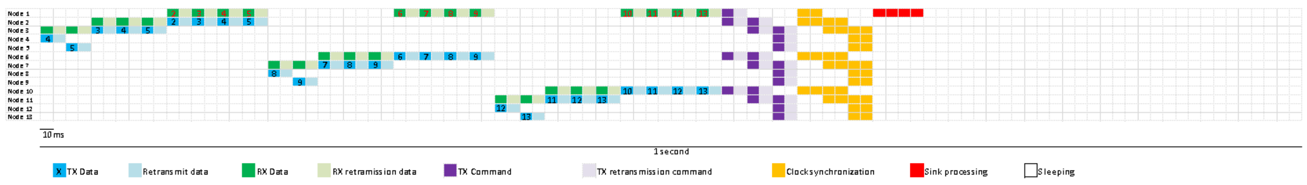


Figure 7.2 – TDMA - epoch definition

7.2. Operations and Requirements

High level operations, such as configuration, actuation, monitoring and closed-loop control, can be defined over those heterogeneous distributed control systems:

- **Configuration** – sending commands to a node to configure it;
- **Actuation** – sending commands to a node to actuate over a DAC connected to some physical process;
- **Monitoring** – sense and send data measures to a control station, where they will be processed/delivered to users. The monitoring operation can be periodic or event-based. If the operation executes with a specific period, it is periodic, but if the operation executes only when an external event occurs, it is event-based. Periodic monitoring is based on a configured sampling rate;
- The **closed-loop control** operation corresponds to sensing and sending data measures to supervision control logic, processing them, determining an actuation command to send to an actuator, sending the actuation command and actuating. Similar to monitoring, supervision control logic can react based on events (asynchronous control) or with a pre-defined periodicity (synchronous control). These are defined as:

- **Asynchronous or event-triggered** – Asynchronous control can be defined as: “upon reception of a data message, the supervision control logic computes a command and sends it to an actuator”. The control computations are based on events. For instance, we can configure a node to send data messages only if a certain threshold condition was met.
- **Synchronous or time-triggered** – Synchronous control can be defined as a periodic control, where the periodicity of computations is defined by users. The supervision control logic runs in specific instants (period). Typically, this type of controller involves multiple samples for the computation of the actuation commands, and multiple nodes may participate in sensing or actuation.

Each of those operations can be associated with timing requirements. For instance, users can specify the maximum latency for monitoring messages to be delivered to a control station. Since the message must travel through several parts of the distributed system, the latency should be predicted to conclude if the desired maximum monitoring latency bound is achieved or not. In general, each operation can be associated with timing requirements, and the system must be able to meet those requirements.

7.3. Base Latency Model

Some latency parts can be identified in a heterogeneous distributed control system with WSN sub-networks, where WSN sub-networks have a TDMA protocol.

Since nodes are configurable and the distributed control system may include actuators, the latency model can be decomposed in two parts: monitoring latency model (upstream) and commanding latency model (downstream).

Section 7.3.1 discusses monitoring latency. This latency corresponds to the latency that is measured from the sensing node to the control station. Section 7.3.2 discusses the command latency model. This latency model is used to access the latency of sending a configuration command or the latency associated with an actuation command resulting from a closed-loop operation.

7.3.1. Monitoring latency model

The monitoring latency can be divided into several parts (Figure 7.3):

- the time elapsed between when an event happens and its detection by the node (t_{Event});
- the latency to acquire a sensor/event value (t_{Aq});
- the time needed to reach the transmission slot (this time can be neglected if we synchronize acquisition with the upstream slot for the sensor node) ($t_{WaitTX\ Slot}$);
- WSN latency ($t_{WSN_{UP}}$);
- sink and gateway latencies, which is the time needed to write messages coming from the WSN to the gateway (t_{Serial} and $t_{Gateway}$);
- the local area network latency, which represents the latency needed to forward messages coming from the gateway to PLCs, computers or control stations (t_{LAN});
- the latency associated with processing in the control station ($t_{\mathrm{Processing}}$).

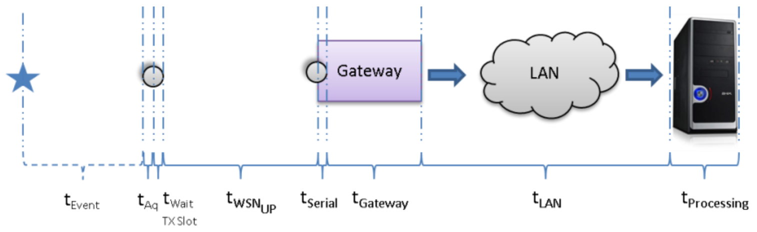


Figure 7.3 – Model for monitoring latency

Consider a reading going from a WSN node all the way to a control station. There is a latency associated with sensor sample (t_{Aq}), a latency associated with the elapsed time between sensor sample a data transmission instant ($t_{WaitTX\ Slot}$) and a latency associated with the WSN path ($t_{WSN_{UP}}$), which corresponds to the time taken to transmit a message from a source node to the sink node. Assuming the epoch defined in Figure 7.2, this time can be deduced from looking at the epoch.

If the instant when an external event manifests itself is considered, there is also a latency associated with the time elapsed between when the event first happened and its detection by the sensor node. As the event may occur in any instant (it is not synchronized with the sensor sampling and transmission slot), t_{Event} represents the amount of time between when an event manifests itself and when it is detected by the sampling mechanism. For instance, we are sampling temperature every second, and the event is a temperature above 90ºC. As shown in Figure 7.4, if the event manifests itself for instant *e* onwards, and the next sampling instant is in instant *a*, then the wait time is t_{Event}.

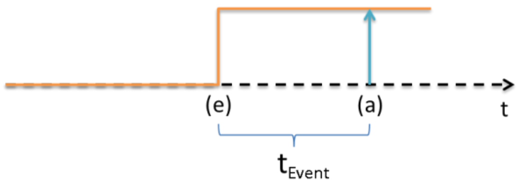
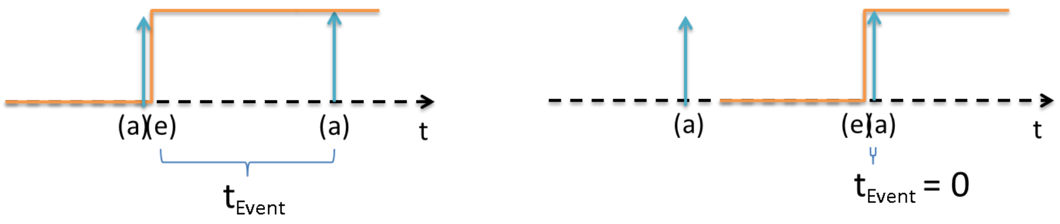


Figure 7.4 – Event detection

Assuming a schedule-based protocol for a WSN sub-network, t_{Event} can assume a value between zero and one epoch minus acquisition latency (t_{Aq}). As shown in Figure 7.5a), if the event manifests itself immediately after the sampling instant, that event needs to wait one epoch minus acquisition latency to be detected. On the other hand (Figure 7.5b)), if the event manifests itself immediately before the sampling instant, t_{Event} is minimum and can be neglected.



a) Maximum time **b) Minimum time**
Figure 7.5 – Time diagram (from event to detection by the sampling mechanism)

Similar to t_{Event}, $t_{WaitTX\ Slot}$ varies from 0 and one epoch. It is minimum when sensor acquisition occurs immediately before transmission (acquisition and sending are synchronized), and maximum when sensor acquisition occurs after transmission slot (acquisition and sending are not synchronized).

In the gateway, there are two time intervals that can be considered (t_{Serial} and $t_{Gateway}$). The first one (t_{Serial}) corresponds to the time needed by the sink node to write a message and gather it at the gateway side (e.g. if a serial interface is used, t_{Serial} is the time needed to write a message to the serial port, plus the time needed to read it by the gateway). The second component ($t_{Gateway}$) corresponds to the time needed for the gateway to get the message, do any processing that may have been specified over each message (e.g. translating the timestamps), and send it to a control station.

The third part (t_{LAN}) corresponds to LAN transmission time, communication and message handling software. Typically, this time is small because fast networks are used (FieldBus, Ethernet GigaBit) and have a significant bandwidth available. To simplify our model, we use $t_{Middleware}$ to refer to $t_{Gateway}$ + t_{LAN}.

Lastly, there is the time needed to perform the data analysis at control stations or other computer nodes ($t_{Processing}$).

The total amount of monitoring latency can be determined as following:

$$Monitoring_{Latency} = t_{WSN_{AqE}} + t_{Serial} + t_{Middleware} + t_{Processing} \tag{1}$$

Where $t_{WSN_{AqE}}$ represents the amount of latency introduced by the WSN sub-network. It is given by:

$$t_{WSN_{AqE}} = t_{Event} + t_{Aq} + t_{WaitTX\ Slot} + t_{WSN_{UP}} \tag{2}$$

7.3.2. Command latency model

In the down path, used by configuration or actuation commands, there are also latency parts that can be identified. Figure 7.6 shows those parts.

Consider a command sent from a control station to a node. Similar to upstream data transmission, there is LAN transmission latency (t_{LAN}). In the gateway, there are three time intervals that can be considered ($t_{Gateway}$, t_{Serial} and $t_{Wait\,for\,SinkTX\,slot}$). $t_{Gateway}$ and t_{Serial} are similar to upstream data latency. $t_{Gateway}$ corresponds to the time needed for the gateway to receive the command, do any processing, and send it to the serial port. t_{Serial} corresponds to the time needed by the sink node to read the command from the serial interface.

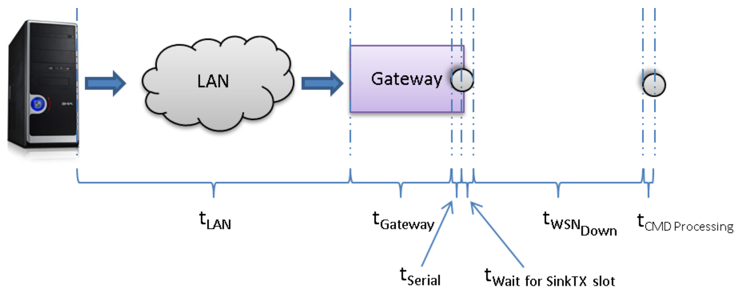


Figure 7.6 – Model for command latency

Upon receiving the command by the sink node, it needs to wait $t_{Wait\,for\,SinkTX\,slot}$ to reach the transmission slot to send the command to the target node. This latency part represents the amount of time that a command is kept in the sink node until it gets a downstream slot. Due to WSN time synchronization, this time can be reduced by choosing the correct position of the downstream slot.

Lastly, there are latencies associated with the WSN path ($t_{WSN_{Down}}$) and node processing command ($t_{CMDProcessing}$). $t_{WSN_{Down}}$ corresponds to the time taken to transmit a command from the sink node to the target node, while $t_{CMDProcessing}$ corresponds to the time taken to process the command inside the target node.

The total amount of command latency can be defined as following:

$$Command_{Latency} = t_{Middleware} + t_{Serial} + t_{WSN_{CMD}} + t_{CMD\mathrm{Processing}} \quad (3)$$

Where $t_{WSN_{CMD}}$ represents the amount of latency introduced by the sink node to send the command ($t_{Wait\,for\,SinkTX\,slot}$) plus the time needed to transmit the command to the target node ($t_{WSN_{Down}}$).

$$t_{WSN_{CMD}} = t_{Wait\,forSinkTXslot} + t_{WSN_{Down}} \quad (4)$$

7.4. Adding Closed-loops to Latency Model

Most performance-critical applications can be found in the domain of industrial monitoring and control. In these scenarios, control loops are important and can involve any node and any part of the distributed system.

Given computational, energy and performance considerations, closed-loop paths may be entirely within a single WSN sub-network or require intervention of control station (e.g. for applying more computational complex supervision controller), or it may span more than one WSN sub-networks, with supervision control logic residing in one of the distributed PLC outside the WSNs (middleware servers).

The closed-loop latency is the time taken from the sensing node to the actuator node, passing through supervision control logic. It will be the time taken since the sample is gathered in the sensing node to the instant when the action is performed in the actuator node.

The position of supervision control logic may depend on timing restrictions and data needed to compute decisions. For instance, if minimal latency is required and a single sub-network is considered, the supervision control logic must be deployed in the sink node. But the limited resources of the sink node (e.g. memory, computation capabilities) mean that supervision control algorithms that require heavy computational and/or memory capabilities need to be implemented in more powerful control stations.

Figure 7.7 shows a scenario example of closed-loop system where the supervision control logic is implemented in the sink node.

At the sink node, when a data message from sensing nodes participating in the decision arrives (asynchronous control), or at defined time periods (synchronous control), the condition and thresholds are analysed, and the actuator is triggered if one of the defined conditions is matched.

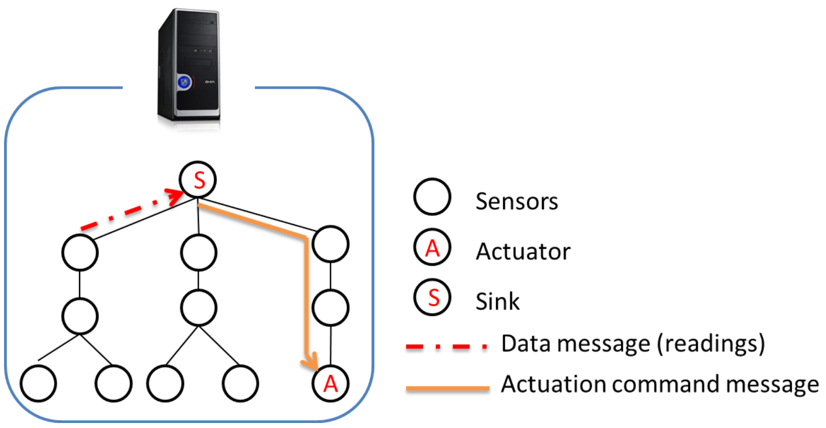


Figure 7.7 – Control decision at sink node

The closed-loop latency for asynchronous control can be estimated by:

$$CL_{Latency_{Async}} = t_{WSN_{AqE}} + t_{\text{Processing}} + t_{WSN_{CMD}} + t_{CMD\text{Processing}} \qquad (5)$$

Where $t_{WSN_{AqE}}$ is determined according eq. 2, while $t_{WSN_{CMD}}$ is determined according eq. 4. The $t_{CMD\text{Processing}}$ represents the time needed to process a command in the node, and $t_{\text{Processing}}$ represents the amount of time needed by the sink to decide an actuation command. This value should be obtained experimental and it is indicated by the user.

Figure 7.8 shows a diagram of $CL_{Latency_{Async}}$ for asynchronous control.

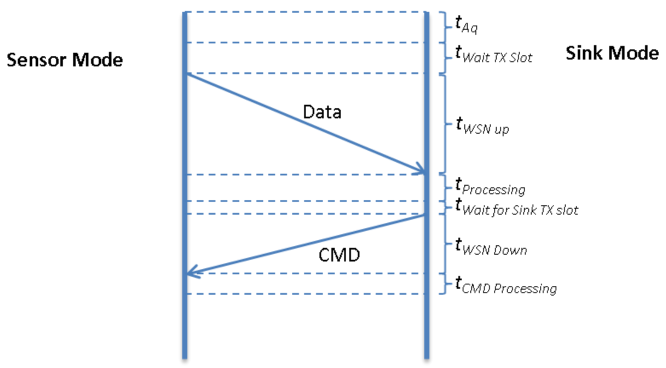


Figure 7.8 – Closed-loop latency (sink node)

Concerning synchronous control, the closed-loop latency for sink decisions can be estimated by:

$$CL_{Latency_{Sync}} = t_{\mathrm{Processing}} + t_{WSN_{CMD}} + t_{CMD\mathrm{Processing}} \qquad (6)$$

In this case we assume that data values are available at the supervision control logic when it runs. So, the $CL_{Latency_{Sync}}$ doesn't include the upstream data latency. Only $t_{\mathrm{Processing}}$, $t_{WSN_{CMD}}$ and $t_{CMD\mathrm{Processing}}$ are considered to compute the $CL_{Latency_{Sync}}$ latency.

If we want to evaluate the time taken from when an event happens (e.g. temperature above 90ºC) and when an actuation value incorporates that value, the latency $CL_{Latency_{End-to-end}}$ is given by eq. 5.

The other alternative for closed-loop control with more powerful resources is to deploy the supervision control logic in one of the distributed PLC, or computers outside the WSN. This alternative is also shown in Figure 7.9. In this case it is possible to read data from one or more WSNs, to compute a decision based in more complex algorithms, and to actuate over the distributed system. The closed-loop algorithm will receive data coming from sensors and will produce actuation commands for the actuator(s).

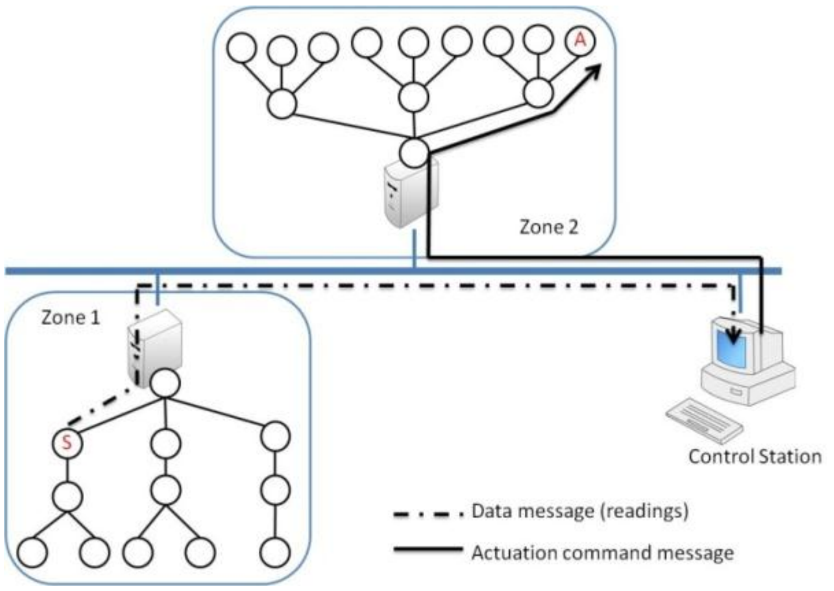


Figure 7.9 – Closed-loop over whole distributed system

In this case the control loop may traverse multiple, most probably non-real-time hardware and software systems, nevertheless the control loop will still need to be under expected time bounds.

One important issue in these closed-loop strategies is the position of the downstream slots. The position of downstream slots must be carefully planned to satisfy the closed-loop timings. For instance, to reduce the closed-loop actuation time, the downstream slots must be scheduled after the upstream slots of the sensing nodes.

The closed-loop latency for asynchronous control supervised through a control station can be estimated as following:

$$\begin{aligned} CL_{Latency_{Async}} = {} & t_{WSN_{AqE}} + t_{Serial} + t_{Middleware} \\ & + t_{\mathrm{Processing}} \\ & + t_{Middleware} + t_{Serial} + t_{WSN_{CMD}} \\ & + t_{CMD\mathrm{Processing}} \end{aligned} \tag{7}$$

The synchronous case latency is similarly obtained by extending eq. 6.

7.5. Adding Non Real-Time Components

We assume non-real-time cabled components, such as gateways, PLCs and computers running operating systems such as Windows, Unix or Linux, and communicating using TCP/IP. Those components are responsible for part of the end-to-end latencies. Those latencies must be characterized by system testing. This is done by running the system with exhaustive measurements under different loads/conditions, while collecting and computing latency statistics.

Next we show two setups (Figure 7.10 and Figure 7.11 corresponding to a small and a larger network respectively) that can be used to characterize latencies in those parts. The first example consists of a small network (Figure 7.10). It includes thirteen TelosB mote and two computers connected through a wired Ethernet. A TelosB is attached to a computer (Gateway) via serial interface. It receives data messages from other nodes and writes them in the serial interface. Each node generates a data message per second. The gateway computer has a dispatcher which forwards each message to the processing computer. Finally, the processing computer computes two different alternative operations to characterize the processing time: a simple threshold and a more complex algorithm to compute the PID gains and the actuation commands.

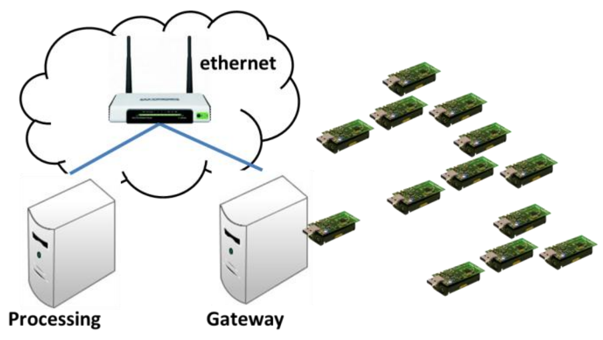


Figure 7.10 – Small network testing setup

Table 7.1 shows the latency characterization for this setup. All times are given in milliseconds.

The most important measure in Table 7.1 is the maximum value. This value allows bounding the latency. In this example, latency would be bounded by

(7.79+3.14+0.86) milliseconds in the case of threshold analysis operation and (7.79+3.14+86.22) milliseconds for the PID controller.

Table 7.1 – Non-real-time parts characterization [ms]

			$t_{\text{Processing}}$	
	t_{Serial}	$t_{Middleware}$	**Threshold analysis**	**PID computation**
Average	2.64	1.12	0.61	73.61
Standard Deviation	0.40	0.29	0.14	5.14
Maximum	7.79	3.14	0.86	86.22
Minimum	1.85	0.67	0.36	53.62

The second example (Figure 7.11) consists of a distributed control system with 3000 sensors. The setup includes 3000 sensors, 6 gateway computers and a control station. All computers are connected through a wired network. Each WSN sub-network is composed by 50 nodes and each gateway computer has 10 gateway processes running. Each node generates a data message per second.

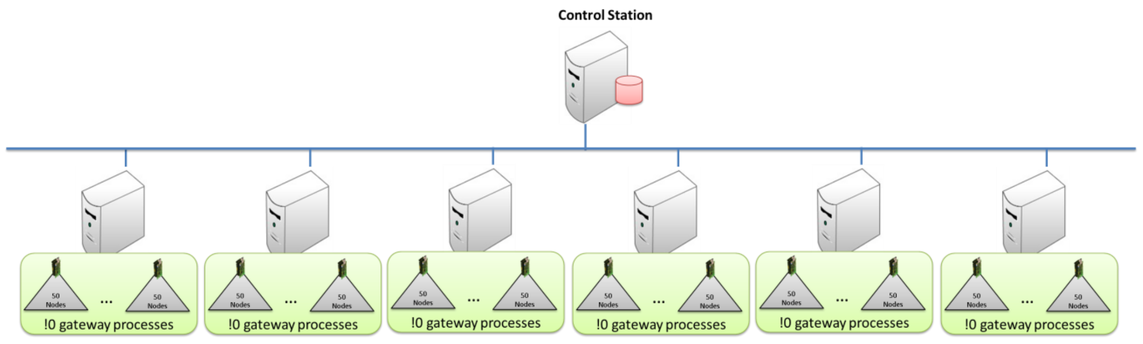


Figure 7.11 – Larger distributed control system - testing setup

Each gateway process also includes a dispatcher which forwards each message to the control station. Similar to GINSENG testbed, each message sent by the gateway is an xml message with the format shown in Appendix F.

Finally, the control station computes four different operations to characterize the processing time:

- Option 1: A simple threshold analysis is used to determine if the value is above a threshold. If it is above, a user interface is updated with the value and alarm information. The values were generated randomly, so that 50% were above the threshold. At the same time, the control station collects processing time and characterizes it.

- Option 2: An average of last 50 samples per nodes is computed to compare the result with the threshold value. Similar to the previous case, if value was greater than the threshold, an alarm is generated.
- Option 3: Insert into a database. Each message that arrives at the control station is stored in a database without further processing.
- Option 4: Insert into a database and request the database to compute the average of last received messages per node. Each message that arrives at the control station is stored in a database. After that, the control station submits an SQL query (Figure 7.12) to the database to compute the average temperature of messages received in the last 60 seconds.

```
SELECT avg(temp), TS
FROM sensorData
WHERE TS
BETWEEN (now-60000) AND now
```

Figure 7.12 – SQL query

Table 7.2 shows the non-real-time parts characterization for this setup. All times are given in milliseconds.

Table 7.2 – Non-real-time parts characterization – second setup [ms]

		$t_{\text{Processing}}$			
	$t_{Middleware}$	**Option 1**	**Option 2**	**Option 3**	**Option 4**
Average	3.50	2.56	3.43	7.67	12.39
Standard Deviation	0.87	0.51	0.48	1.52	1.21
Maximum	5.25	5.33	5.75	8.31	18.06
Minimum	1.75	0.23	0.27	7.04	10.80

In this example, and assuming the same serial latency as in Table 7.1, latency of message stored in the database would be bounded by (7.79+5.25+8.31) milliseconds.

7.6. Prediction Model for Maximum Latency

Equations 1 and 3 predict monitoring and commanding latencies. However, maximum values for each latency should be used because we are considering strict end-to-end operation timing guarantees. In this section we discuss how to predict maximum operation latencies and how to obtain the maximum value for each part of the latency.

The maximum monitoring latency can be determined as following:

$$\max\left(Monitoring_{Latency}\right) = \max\left(t_{WSN_{AqE}}\right) + \max\left(t_{Serial}\right) + \max\left(t_{Middleware}\right) + \max\left(t_{\mathrm{Pr}ocessin\,g}\right) \quad (8)$$

As described in the previous section, the non-real-time parts ($t_{Serial}, t_{Middleware}$ and $t_{\mathrm{Pr}ocessin\,g}$) must be characterized experimentally. The maximum values of these latency parts result from the computation of statistics for the time measures collected during experimental period.

The maximum value of $t_{WSN_{AqE}}$ can be predicted by analysing the maximum values of it parts (eq. 9).

$$\max\left(t_{WSN_{AqE}}\right) = \max\left(t_{Event}\right) + \max\left(t_{Aq}\right) + \max\left(t_{WaitTX\,Slot}\right) + \max\left(t_{WSN_{UP}}\right) \quad (9)$$

The values of t_{Event} and $t_{WaitTX\,Slot}$ were described in the sub-section 7.3.1. They can vary from 0 to one epoch size, where the maximum value for each is an epoch size. So, $t_{WSN_{AqE}}$ can assume one of the following four alternatives:

- $\max\left(t_{WSN_{AqE}}\right) = \max\left(t_{Aq}\right) + \max\left(t_{WSN_{UP}}\right)$ - when the acquisition instant is synchronized with the sending instant and we do not consider the event start instant.
- $\max\left(t_{WSN_{AqE}}\right) = \max\left(t_{Aq}\right) + \max\left(t_{WaitTX\,Slot}\right) + \max\left(t_{WSN_{UP}}\right)$ - when the acquisition instant is not synchronized with the sending instant and we do not consider the event start instant.

- $\max\left(t_{WSN_{AqE}}\right)=\max\left(t_{Event}\right)+\max\left(t_{Aq}\right)+\max\left(t_{WSN_{UP}}\right)$ - when the acquisition instant is synchronized with the sending instant and we are considering the event start instant.
- $\max\left(t_{WSN_{AqE}}\right)=\max\left(t_{Event}\right)+\max\left(t_{Aq}\right)+\max\left(t_{WaitTX\,Slot}\right)+\max\left(t_{WSN_{UP}}\right)$ - when the acquisition instant is not synchronized with the sending instant and we are considering the event start instant.

Similar to non-real-time parts, t_{Aq} must be characterized by experimental evaluation. It depends on the node platform and which sensor is sampled. The maximum value results from the collection of time measures during the experiment.

Lastly, $\max\left(t_{WSN_{UP}}\right)$ depends on the node position in the WSN topology. This value is always constant ($t_{WSN_{UP}}=\max\left(t_{WSN_{UP}}\right)$), since we are assuming a static WSN topology.

The prediction of command latency should also be based on maximum values of each part (eq. 10).

$$\begin{aligned}\max\left(Command_{Latency}\right)=\max\left(t_{Middleware}\right)\\ +\max\left(t_{Serial}\right)+\max\left(t_{WSN_{CMD}}\right)\\ +\max\left(t_{CMD\,\mathrm{Processing}}\right)\end{aligned} \tag{10}$$

$t_{Middleware}$, t_{Serial} and $t_{CMD\,\mathrm{Processing}}$ are obtained experimentally, while $\max\left(t_{WSN_{CMD}}\right)$ is determined by eq. 11.

$$\max\left(t_{WSN_{CMD}}\right)=\max\left(t_{Wait\,forSinkTXslot}\right)+\max\left(t_{WSN_{Down}}\right) \tag{11}$$

Where $\max\left(t_{Wait\,forSinkTX\,slot}\right)$ can be an epoch size. For instance, we are sending an actuation command to a WSN node. As shown in Figure 7.13, if the command arrives immediately after the sending instant, and the next sending instant is in instant *s*, then the wait time is $t_{Wait\,forSinkTX\,slot}$. If only one downstream slot was provided per epoch, $\max\left(t_{Wait\,forSinkTX\,slot}\right)=epoch\,size$.

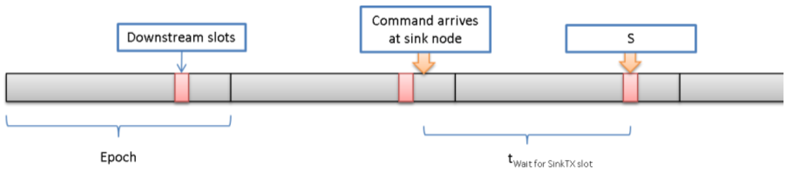


Figure 7.13 – Instant of command sending by the sink node

7.7. Algorithm for Planning Time Guarantees

The planning algorithm that we propose next allows users to dimension the network and conclude whether desired latency bounds are met. Figure 7.19 shows a diagram of the planning approach proposed in this and the next sections.

7.7.1. User inputs

The proposed approach allows users to dimension the network based on two alternatives. One alterative assumes the user provides a complete TDMA schedule and operation timing requirements. The algorithm checks if latencies are met and modifies the schedule (may even determine the need to divide the network) to meet the latency bounds.

The other alternative allows the user to specify only the minimum possible amount of requirements, and the algorithm creates the whole schedule taking into consideration all constraints.

- **Network layout** – the first network layout is completely defined by the user. It takes into account the physical position of the nodes, their relation (which nodes are leaf nodes and their parents) and a schedule to define how data and commands flow in the network.
  Appendix H shows a text-based example of how this could be specified (this example was taken from the GINSENG testbed case).

- **Network configuration and data forwarding rule** – the network configuration is indicated by the user and takes into account the physical position of the nodes (which nodes are leaf nodes and their parents) and data forwarding rule indicates how the schedule must be defined to forward data messages from sensing nodes to the sink node. The node slot positioning is directly dependable of the network configuration, but should be optimized to reduce latencies or the number of radio wake ups. The data forwarding rule can assume one of the following options:

- Each node sends data to its parent. Each parent receives data from a children and forwards immediately to its parents.

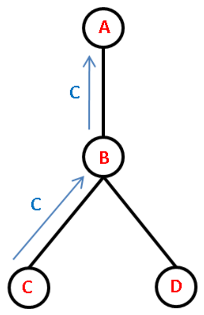


Figure 7.14 – Data forwarding rule – option 1

- Each node sends data to its parent. Each parent collects data from all children and only forwards up after receiving from all children.

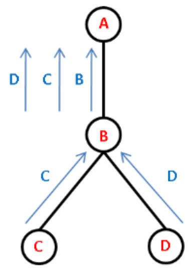


Figure 7.15 – Data forwarding rule – option 2

- Each node sends data to its parent. Each parent collects data from all children, aggregates data messages from all children, and only forwards a merged message to its parents.

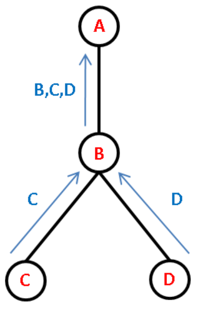


Figure 7.16 – Data forwarding rule – option 3

Appendix G shows one text-based example of how this alternative could be specified.

The algorithm assumes that users provide the information needed to use one of the above alternatives. It is also necessary for users to indicate some other parameters:

- $t_{Clock_{Sync}}$ - this is the clock synchronization interval, i.e. the time between clock synchronizations, which are necessary to keep the WSN node clocks synchronized, avoiding clock drifts between them.
- $\max\left(t_{CMDProcessing}\right)$ - this is a small time needed to parse and process a command in any WSN node. The user must specify a maximum bound for this time.
- $\max\left(t_{CL(i)_{Processing}}\right)$ - for each closed-loop operation, the user should specify the maximum required time taken to process, which corresponds to the computation time needed at the decision maker node to take a decision, for commanding the required actuation.

The values of $\max\left(t_{Serial}\right)$, $\max\left(t_{Middleware}\right)$, $\max\left(t_{CL(i)_{Processing}}\right)$ and $\max\left(t_{CMDProcessing}\right)$, defined in Section 7.3, should be given by the user. They are previously obtained by distributed control system testing. A desired sampling rate should also be indicated.

Lastly, the algorithm needs to be configured concerning downstream slots positioning rule. The proposed algorithm supports two alternatives to position those:

- **Equally spaced in the epoch** – the downstream slots are positioned equally spaced in the epoch. Assuming that, due to latency requirements, two downstream slots are added by the algorithm, Figure 7.17 shows how these slots are positioned according to this alternative.

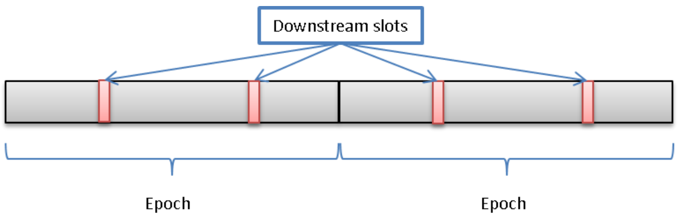


Figure 7.17 – Downstream slots equally spaced in the epoch

- **After a specific slot in the epoch** – it allows to optimize the processing and command latency, by positioning these slots after a specific slot. For instance, assuming a closed-loop asynchronous operation in the sink node, where the actuation command is decided based on sensed data coming from a sensor node, the processing and downstream slots should be positioned after the upstream slots that complete the path from the sensor node to the sink node. This allows the sink node to process and command an actuator immediately after receiving sensed data without having to wait more time for a downstream slot. This reduces the command latency and, consequently, the closed-loop latency. Figure 7.18 shows an illustration of this alternative.

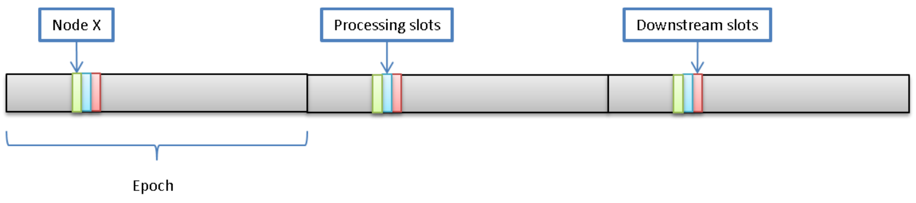


Figure 7.18 – Downstream slots positioned to optimize asynchronous closed-loop latency

7.7.2. Overview of the algorithm

If the user choose to provide a network configuration plus forwarding rule as an input (see example in Appendix G), the algorithm starts by defining the upstream part of the TDMA schedule for all nodes, according to the forwarding rules (steps 1, 2 & 3).

This results in a first TDMA schedule sketch, which has all upstream slots, but still needs to be completed.

Instead of this alternative, the user may have provided the network layout as an input (see example in Appendix H). In that case, the network layout given is already a TDMA schedule and the algorithm checks and analysis it to guarantee latencies.

After the first TDMA schedule and the current epoch size are determined, the algorithm analyses latency requirements and determines how many downstream slots are needed (step 5 & 6). The necessary number of slots is added to the TDMA schedule according to the processing and downstream slots positioning requirements (step 7 & 3). The current epoch size is determined again. Next, based on $t_{Clock_{Sync}}$, the algorithm determines how many slots are needed for time synchronization and adds them to the schedule (step 8 & 9). The new epoch size is determined, and if the user specified equally spaced downstream slots, the number of downstream slots is verified to check if they are enough for the current epoch size. If they are not enough, more downstream slots are added and a new schedule is recomputed.

Based on latency requirements, the algorithm verifies if those are met (step 10 and 11). If latency requirements are not met, it means that it is not possible to meet the timing requirements with the current size of the network, therefore the network is partitioned (step 12) and the algorithm re-starts with each partition. After a network or network partition meets the requirements, the maximum epoch size is determined (step 13) to verify if an inactivity period can be added to the schedule. This inactivity period can be added to maximize the node lifetime (step 15).

After adding the inactivity period, it is necessary to rerun the algorithm from step 3 onwards to verify latencies again.

After re-running all the steps and if all of them are ok, the desirable lifetime is attainable. If the lifetime is not achieved, the network must be divided again (step 12), and the algorithm re-runs for each resulting network.

To conclude the algorithm, a communication channel must be defined for each sub-network, and a sampling rate is assigned (step 16).

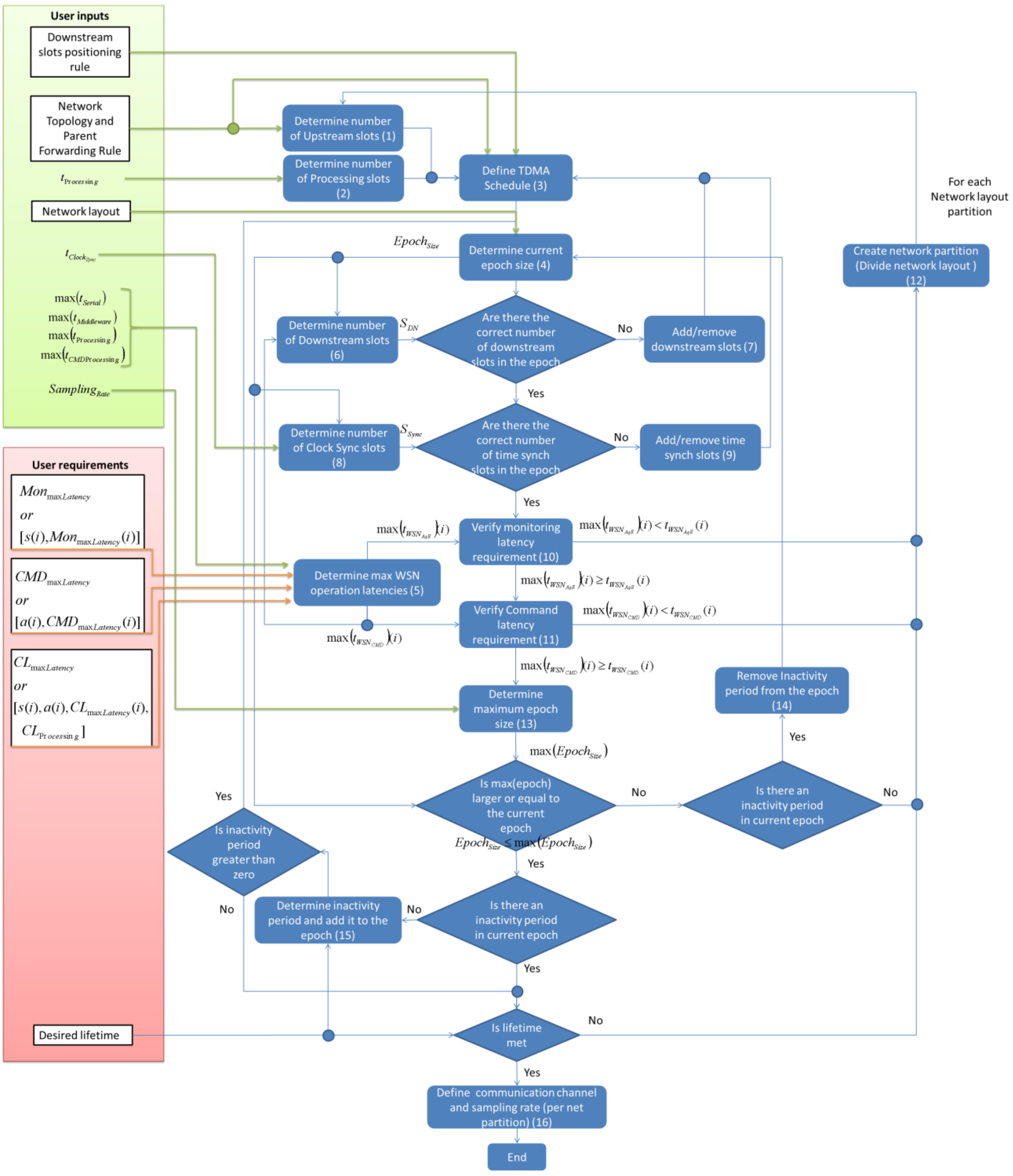

User inputs
Downstream slots positioning rule
Network Topology and Parent Forwarding Rule
$t_{Processing}$
Network layout
$t_{Clock_{Sync}}$
$\max(t_{Serial})$
$\max(t_{Middleware})$
$\max(t_{Processing})$
$\max(t_{CMDProcessing})$
$Sampling_{Rate}$
User requirements
$Mon_{\max Latency}$ or $[s(i), Mon_{\max Latency}(i)]$
$CMD_{\max Latency}$ or $[a(i), CMD_{\max Latency}(i)]$
$CL_{\max Latency}$ or $[s(i), a(i), CL_{\max Latency}(i), CL_{Processing}]$
Desired lifetime
Determine number of Upstream slots (1)
Determine number of Processing slots (2)
Define TDMA Schedule (3)
$Epoch_{Size}$
Determine current epoch size (4)
Determine number of Downstream slots (6)
S_{DN}
Are there the correct number of downstream slots in the epoch
No
Add/remove downstream slots (7)
Yes
Determine number of Clock Sync slots (8)
S_{Sync}
Are there the correct number of time synch slots in the epoch
No
Add/remove time synch slots (9)
Yes
$\max(t_{WSN_{AqE}})(i)$
Verify monitoring latency requirement (10)
$\max(t_{WSN_{AqE}})(i) < t_{WSN_{AqE}}(i)$
Determine max WSN operation latencies (5)
$\max(t_{WSN_{AqE}})(i) \geq t_{WSN_{AqE}}(i)$
Verify Command latency requirement (11)
$\max(t_{WSN_{CMD}})(i) < t_{WSN_{CMD}}(i)$
$\max(t_{WSN_{CMD}})(i)$
$\max(t_{WSN_{CMD}})(i) \geq t_{WSN_{CMD}}(i)$
Determine maximum epoch size (13)
$\max(Epoch_{Size})$
For each Network layout partition
Create network partition (Divide network layout) (12)
Remove Inactivity period from the epoch (14)
Yes
Is max(epoch) larger or equal to the current epoch
$Epoch_{Size} \leq \max(Epoch_{Size})$
No
Is there an inactivity period in current epoch
No
Yes
Is inactivity period greater than zero
No
Yes
Determine inactivity period and add it to the the epoch (15)
No
Is there an inactivity period in current epoch
Yes
Is lifetime met
No
Yes
Define communication channel and sampling rate (per net partition) (16)
End

Figure 7.19 – Planning algorithm

The user indicated desired sampling rate is checked against the schedule to verify if it needs to be increased (e.g. the user specified 1 sample per second, but it is necessary to have 2 samples per second in order to meet latencies). The algorithm determines a schedule based on latency and other requirements. In order to meet latencies with that schedule, there must be at least one acquisition per epoch, therefore this setup is guaranteeing that there is at least on sample per epoch.

7.7.3. Determine the first network layout (step 1, 2 & 3)

Given a network configuration and data forwarding rule, the first schedule is created. This first schedule only includes slots for sending data from sensing nodes to the sink node. Figure 7.20 shows the pseudo-code of the algorithm used to create the schedule based on network configuration and data forwarding rule.

```
if data forwarding rule == option 1 then
  for each node in the network configuration without slot for its data
transmission
      allocate a slot to it;
      look up for the path to the sink node;
      for each parent in the path to the sink node:
        allocate a slot to it;
else if data forwarding rule == option 2 or option 3 then
  C = set of child nodes for node i;
  slotAssignment(C(sink));
end if

slotAssignment (C){
  for each node i in C
    if there are child nodes connected to it then
      slotAssignment (C (i));
    else
      allocate slot to node i
      if data forwarding rule == option 2 then
        allocate a slot for each child connected to node i
}
```

Figure 7.20 – Slot assignment algorithm – pseudo-code

The algorithm starts by identify which data forwarding rule is used. If option 1, described in Section 7.7.1, is used, the algorithm will select node by node from the network configuration, determine the path to the sink and allocate forwarding slots for the node and for each node between its position and the sink node.

If option 2 or 3 is used, the algorithm runs in a recursive way to allocate slots from the leaf nodes to the sink node. If option 2 is used, the algorithm allocates a slot per node plus one slot for each child node connected to it. In case of option 3, slots to forward data from the child nodes are not added. In this case we assume that all data coming from child nodes can be merged with node data in a single message.

The slot assignment algorithm can also use one or more than one slot if re-transmission is desired to increase reliability.

7.7.4. Determine current epoch size (step 4)

The current epoch size is the number of slots in the current schedule. This schedule could be indicated by the user as an input or can result from applying the algorithm.

This step is recomputed several times during the algorithm flows because the current schedule is changing along the flow. For instance, if the user introduces a network configuration and data forwarding rule, the algorithm creates whole schedule. In steps 1, 2 and 3, the algorithm creates a first schedule and determines the current epoch size. Based on this current epoch size, the algorithm determines how downstream slots are needed according to latency requirements and user inputs. After that, these downstream slots are added to the schedule, resulting in a different epoch size. So, the step 4 is called again to determine the current epoch size.

7.7.5. Determine maximum WSN latencies (step 5)

Once the schedule is defined (and consequently the epoch size), the algorithm will predict the WSN part of latency.

Assuming a maximum latency for monitoring messages ($Mon_{MaxLatency}$) and based on eq. 8, we determine the maximum admissible latency for the WSN sub-network (eq. 12).

$$\max\left(t_{WSN_{AqE}}\right) = Mon_{MaxLatency} - \left(\max\left(t_{Serial}\right) + \max\left(t_{Middleware}\right) + \max\left(t_{\mathrm{Processing}}\right)\right) \qquad (12)$$

Instead of a single maximum monitoring latency $Mon_{MaxLatency}$, the algorithm allows the user to specify pairs [$node(i)$, $Mon_{MaxLatency}(i)$]. In this case eq. 8 must be applied for each node, resulting in a maximum admissible latency per WSN node (eq. 13).

$$\max\left(t_{WSN_{AqE}}\right)(i) = Mon_{MaxLatency}(i) - \left(\max\left(t_{Serial}\right) + \max\left(t_{Middleware}\right) + \max\left(t_{\mathrm{Processin}g}\right)\right) \qquad (13)$$

Similar to the monitoring latency, users can define maximum latencies to deliver a command to a WSN node. Assuming that $CMD_{MaxLatency}$ is specified as maximum command latency, based on eq. 10, $\max\left(t_{WSN_{CMD}}\right)$ can be determined as:

$$\max\left(t_{WSN_{CMD}}\right) = CMD_{MaxLatency} - \left(\max\left(t_{Middleware}\right) + \max\left(t_{Serial}\right) + \max\left(t_{CMD\mathrm{Processin}g}\right)\right) \qquad (14)$$

If pairs of [$node(i)$, $CMD_{MaxLatency}(i)$] are given, the algorithm applies eq. 14 per each pair, resulting in a set of maximum command latencies. Based on that set of latencies, the algorithm chooses the strictest latency, and dimensions the network to meet that latency (steps 6 and 7).

7.7.6. Determine the number of downstream slots (step 6 & 7)

In equation 14 we determine $\max\left(t_{WSN_{CMD}}\right)$. Through the discussion given in Section 7.6 and eq. 11 we can see that it is directly dependent of the number of the downstream slots per epoch.

Each epoch must accommodate slots for transmitting configuration and actuation commands. The minimum number of downstream slots (S_{DN}) that must exist can be calculated as follows:

$$S_{DN} = H \cdot s \qquad (15)$$

Where H represents the number of levels and s represents the number of slots per level (one by default, plus one or more for enhanced reliability, to accommodate retransmissions). This is the case where the whole epoch has one downstream slot available for the sink node to forward a message downwards.

The worst case latency for this case is larger than an epoch size, as shown and discussed in eq. 10 and eq. 11. Since epoch sizes may be reasonable large (e.g. 1

second), this may result in an undesirable command latency, in particular it may not meet latency requirements.

The number of downstream slots can be increased to meet user command latency requirements. Figure 7.21 shows an example of a worst case wait time for a command arriving at the sink and waiting for its downstream slot.

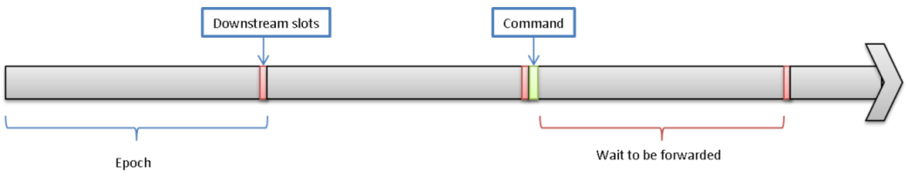


Figure 7.21 – Worst case: schedule with one downstream slot

As show in Figure 7.21, if one downstream slot is provided per epoch, the command must wait, in the worst case, a full epoch to be transmitted. Additionally, in average a command will have to wait half the epoch size.

In order to shorten the $\max\left(Command_{Latency}\right)$, there are two major alternatives: reducing the epoch length (we assume this is unwanted, since it was already determined to reflect a network with a certain number of nodes, energy and latency requirements); adding extra downstream slots.

Next we discuss the addition of more downstream slots to reduce $t_{Wait\,for TX\,slot}$ of eq. 11. Those can be placed equally-spaced in the epoch to minimize the maximum expected $t_{Wait\,for TX\,slot}$, given that number of downstream slots. As an example, Figure 7.22 shows an epoch with two downstream slots. In this case, when a command arrives at the sink, it must wait a maximum time of $\frac{Epoch}{2}$.

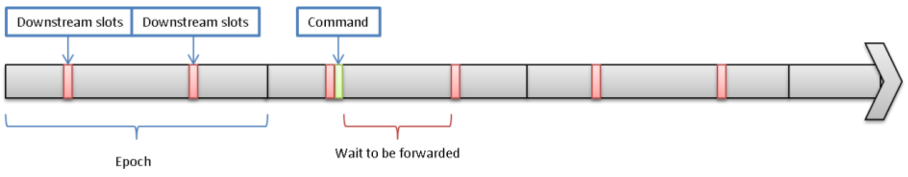


Figure 7.22 – Schedule with two downstream slots

In the next example (Figure 7.23), the epoch includes four downstream slots which allows the maximum wait time to be reduced to $\frac{Epoch}{4}$.

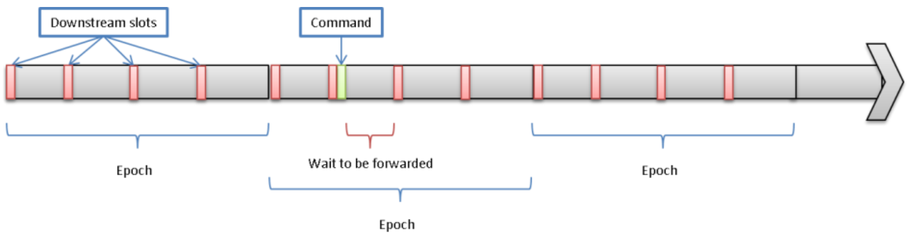


Figure 7.23 – Schedule with four downstream slots

More generically, adding n downstream slots results in $\max(t_{Wait\,for TX\,slot})$ of $\frac{Epoch}{n}$.

In order to guarantee a $CMD_{MaxLatency}$ the number of downstream slots should be dimensioned. Replacing $\max(t_{Wait\,for TX\,slot})$ in eq. 11 we obtain:

$$\max(t_{WSN_{CMD}}) = \frac{Epoch}{n} + \max(t_{WSN_{Down}}) \tag{16}$$

Where we can extract the number of slots (n) (eq. 17).

$$n = \left\lceil \frac{Epoch}{\max(t_{WSN_{CMD}}) - \max(t_{WSN_{Down}})} \right\rceil \tag{17}$$

Replacing $\max(t_{WSN_{CMD}})$ from eq. 11 in eq. 17, we obtain the number of downstream slots in function of $CMD_{MaxLatency}$ (eq. 18).

$$n = \left\lceil \frac{Epoch}{\left(CMD_{MaxLatency} - \left(\max(t_{Middleware}) + \max(t_{Serial}) + \max(t_{CMD\mathrm{Processing}})\right)\right) - \max(t_{WSN_{Down}})} \right\rceil \tag{18}$$

It is also necessary to dimension the downstream slots to meet closed-loop operation latency requirements. If a synchronous closed-loop operation is defined, $CL_{MaxLatency}$ corresponds to $CMD_{MaxLatency}$ latency, because, in this case we consider only the latency from the closed-loop supervisor to the actuator. Therefore, the number of downstream slots is determined by eq. 18.

If an asynchronous closed-loop operation is considered, $CL_{MaxLatency}$ includes monitoring and command parts (eq. 19).

$$\begin{aligned}\max\left(CL_{Latency_{Async}}\right) = {} & \max\left(t_{WSN_{AqE}}\right) + \max\left(t_{Serial}\right) + \max\left(t_{Middleware}\right) + \max\left(t_{\text{Processing}}\right) \\ & + \max\left(t_{Middleware}\right) + \max\left(t_{Serial}\right) + \max\left(t_{WSN_{CMD}}\right) + \max\left(t_{CMD\text{Processing}}\right)\end{aligned} \tag{19}$$

Replacing eq. 15 in eq. 18 and applying mathematic operations, we obtain the number of downstream slots to meet asynchronous closed-loop latencies (eq. 20).

$$n = \left\lceil \frac{Epoch}{\left(\max\left(CL_{Latency_{Async}}\right) - \begin{pmatrix}\max\left(t_{WSN_{AqE}}\right) + \max\left(t_{Serial}\right) + \max\left(t_{Middleware}\right) + \max\left(t_{\text{Processing}}\right) + \\ \max\left(t_{Middleware}\right) + \max\left(t_{Serial}\right) + \max\left(t_{CMD\text{Processing}}\right)\end{pmatrix}\right) - \max\left(t_{WSN_{Down}}\right)} \right\rceil \tag{20}$$

If we assume that the serial and middleware latencies for upstream and downstream are equal, eq. 20 may be simplified and results in eq. 21.

$$n = \left\lceil \frac{Epoch}{\left(\max\left(CL_{Latency_{Async}}\right) - \begin{pmatrix} \max\left(t_{WSN_{AqE}}\right) \\ + 2 \cdot \max\left(t_{Serial}\right) \\ + 2 \cdot \max\left(t_{Middleware}\right) \\ + \max\left(t_{\mathrm{Processing}}\right) \\ + \max\left(t_{CMD\mathrm{Processing}}\right) \end{pmatrix} \right) - \max\left(t_{WSN_{Down}}\right)} \right\rceil \tag{21}$$

After determining the number of downstream slots needed to meet command latency or closed-loop latency requirements, the algorithm adds them to the schedule and re-computes the current epoch size (step 4).

7.7.7. Number of clock synchronization slots (step 8 & 9)

In this model, we are assuming a TDMA protocol, which requires slots for clock synchronization ($N_{Slots_{Sync}}$). $N_{Slots_{Sync}}$ is determined based on the $Epoch$ and the parameter clock interval ($t_{Clock_{Sync}}$), which is indicated by the user. It is determined as follows:

$$N_{Slots_{Sync}} = \left\lceil \frac{Epoch}{t_{Clock_{Sync}}} \right\rceil \tag{22}$$

After determining $N_{Slots_{Sync}}$ the algorithm adds them to the schedule and re-computes the current epoch size (step 4).

7.7.8. Verify if latency requirements are met with the current epoch, network layout and schedule (step 10 & 11)

Based on equations 2 and 4, the algorithm determines the latency required to transmit a data message from a WSN leaf node to the sink node ($t_{WSN_{AqE}}$) and the latency to send a command from the sink node to the WSN leaf node ($t_{WSN_{CMD}}$).

After determining these two latencies, the algorithm compares the values with the latency determined through eq. 16 and 17, which results from the user requirements, and concludes if all latencies are met.

If any of the latencies are not met, the algorithm needs to partition the network to find a schedule and an epoch size for each new sub-network that meets the requirements (step 12).

7.7.9. Network partitioning (step 12)

When the algorithm detects that a network must be divided to meet user requirements, step 12 is called.

Assuming that a user gives a network configuration and a parent forwarding rule, the algorithm divides the initial network configuration in 2 parts. This division is done automatically, if the initial network configuration has the same configuration for all branches. Otherwise, the user is requested to split the network and restart the algorithm with the new configuration.

On the other hand, if a network layout is given, the algorithm divides the network and the schedule into two parts. The downstream, processing and clock sync slots are copied for both parts.

7.7.10. Determine the maximum epoch size (step 13 & 14)

The maximum epoch size is defined as the epoch size which is able to guarantee desired latencies and conforms to the sampling rate. This can be determined as:

$$\max(Epoch_{Size}) = \min\{Sampling_{Rate}, \max(Epoch_{Size})_{Latency}\} \tag{23}$$

Where $\max\left(Epoch_{Size}\right)_{Latency}$ is determined according to the following configurations:

- If acquisition instant is not synchronized with sending instant and we do not consider the event start instant, $\max\left(t_{WSN_{AqE}}\right)$ corresponds to the sum of the time waiting for the transmission slot plus the time taken for the data to travel from source node to the sink node. In this case, $\max\left(Epoch_{Size}\right)_{Latency}$ is defined by:

$$\max\left(Epoch_{Size}\right)_{Latency} = \max\left(t_{WSN_{AqE}}\right) - \left(\max\left(t_{Aq}\right) + \max\left(t_{WSN_{UP}}\right)\right) \tag{24}$$

- If acquisition instant is synchronized with sending instant and we are considering the event occurrence instant, $\max\left(t_{WSN_{AqE}}\right)$ corresponds to the time for event detection and its transmission from source node to the sink node. In this case, $\max\left(Epoch_{Size}\right)_{Latency}$ assumes the same value of the previous case, as defined by eq. 24.
- If acquisition instant is not synchronized with sending instant and we are considering the event occurrence instant, $\max\left(t_{WSN_{AqE}}\right)$ corresponds to the time for event detection, plus the time waiting for the transmission instant, plus the travel time from source node to sink node. In this case, $\max\left(Epoch_{Size}\right)_{Latency}$ is defined as:

$$\max\left(Epoch_{Size}\right)_{Latency} = \frac{\max\left(t_{WSN_{AqE}}\right) - \left(\max\left(t_{Aq}\right) + \max\left(t_{WSN_{UP}}\right)\right)}{2} \tag{25}$$

In these equations, the maximum WSN latency ($\max\left(t_{WSN_{UP}}\right)$), which was defined in Section 7.3, corresponds to the time taken to transmit a message from a leaf node to the sink node using the current network layout. It depends on the maximum number of levels included in the monitoring operation. t_{Aq} corresponds to the acquisition latency, as discussed in Section 7.3.

If the $\max(Epoch_{Size})_{Latency}$ is smaller them the current epoch size (from step 4), them either cut the inactivity period of the current schedule (step 14), or otherwise divide the network (step 12) because it is not possible to meet latency requirements or sampling rate with the current size of network. The sampling rate parameter for each new sub-network will be defined as the user-defined sampling rate.

7.7.11. Inactivity period (step 15)

The nodes of the WSNs may or may not be battery operated. If they are battery operated, the user may have defined lifetime requirements, or he may have specified that he wants to maximize lifetime. In order to meet lifetime requirements, it may be necessary to add an inactivity period to the epoch, during which nodes have low consumption, because they turn their radio off.

Latency and sampling rate specifications may restrain the inactivity period that would be necessary to guarantee the desired lifetime. In that case the algorithm can divide the network into two sub-networks and re-run over each, trying to accommodate both the latency and lifetime requirements.

The inactivity period can be determined as follows:

$$Inactivity_{Period} = \min\{Inactivity_{Period}(\max(Epoch_{Size})), Inactivity_{Period}(Lifetime)\} \qquad (26)$$

Where the inactivity period due to the epoch size ($Inactivity_{Period}(\max(Epoch_{Size}))$) is determined by eq. 27.

$$Inactivity_{Period}(\max(Epoch_{Size})) = \max(Epoch_{Size}) - Epoch_{Size} \qquad (27)$$

The inactivity period required by lifetime requirements is defined in Section 7.9. After determining this $Inactivity_{Period}$ quantity, it is added to the schedule, and the algorithm restarts again to verify if all constraints are achieved. This is required because the addition of the inactivity period may have consequences concerning command latencies and the synchronization slots that need to be recalculated.

7.7.12. Network communication channel and sampling rate (step 16)

Since the initial network may need to be divided into several smaller networks, different communication channels should be defined for each resulting sub-network to avoid communication interferences between different sub-networks.

Lastly, the sampling rate is defined as the epoch size or a sub-sampling alternative if the user wishes:

$$Sampling_{rate} = Epoch_{Size} \tag{28}$$

If users want to have sub-sampling (multiple samples per sampling period), the sampling rate would be:

$$Sampling_{rate} = \frac{Epoch_{Size}}{n} \tag{29}$$

Where n represents the number of samples per period. Multiple samples per period allow, for instance, to apply smoothing operations in order to remove noise.

7.8. Lifetime Prediction

Energy consumption is important in networked embedded systems for a number of reasons. For battery-powered nodes, energy consumption determines their lifetime.

The radio transceiver is typically the most power-consuming component. In a TDMA protocol, all network nodes are synchronized to a common clock. Nodes wake up on dedicated time slots at which they are ready to either receive or transmit data.

The node energy consumption can be estimated as [155], [156]:

$$E = \left(I_{on} \cdot Radio_{DutyCycle} + \left(1 - Radio_{DutyCycle}\right) \cdot I_{off}\right) \cdot T \cdot V_{Bat} \tag{30}$$

Where the radio duty cycle is measured as:

$$Radio_{Dutycyle} = \frac{t_{Active(Listen+Transmit)}}{Epoch} \tag{31}$$

The currents I_{on} and I_{off} represent the amount of consumed current in the each state of the radio (*on* and *off*). V_{Bat} is the power supply voltage to power up the node and T represents the amount of time where E is measured.

So, the node lifetime (T_{Total}) can be extracted from eq. 30, where E represents the total charged battery capacity.

$$T_{Total} = \frac{E}{\left(I_{on} \cdot Radio_{DutyCycle} + \left(1 - Radio_{DutyCycle}\right) \cdot I_{off}\right) \cdot V_{Bat}} = Node_{Lifetime} \tag{32}$$

Consequently, network lifetime is defined as the lifetime of the node which discharges its own battery first.

In the previous section we discussed how to plan a network to meet latency and lifetime requirements. To optimize the TDMA schedule and to provide desirable lifetime, an inactivity period must be added to the schedule. Through eq. 26, we define this inactivity period as:

$$Inactivity_{Period} = \min\left\{Inactivity_{Period}\left(Epoch_{Size}\right), Inactivity_{Period}\left(Lifetime\right)\right\}$$

Therefore, the $Inactivity_{Period}\left(Lifetime\right)$ can be calculated through eq. 32, replacing the $Radio_{Dutycyle}$ defined in eq. 31 and solving it.

$$Inactivity_{Period}\left(Lifetime\right) = \left(\frac{t_{Active(Listen+transmit)} \cdot T_{Total} \cdot \left(I_{off} \cdot V_{Bat} - I_{on}\right)}{I_{off} \cdot T_{Total} \cdot V_{Bat} - E}\right) - C\_Epoch_{Size} \tag{33}$$

Where C_Epoch_{Size} represents the current epoch size resulting from step 4 of the algorithm.

7.9. Slot Size Considerations

The slot size should be as small as possible to reduce the epoch size and consequently the end-to-end latency. To determine the slot time, the following times must be taken into account:

- Time to transfer a message from the MAC layers data FIFO buffer to buffer of radio transceiver (t_{ts});
- Time to transmit a message (t_{xm});
- Time a receiver needs to process the message and initiate the transmission of an acknowledgment message (t_{pm});
- Time to transmit an acknowledgment (t_{xa});
- Time to transfer and process the acknowledgment from the radio transceiver and to perform the associated actions for received/missed acknowledgment *(t_{pa})*.

Also, a small guardian time is required at the beginning and end of each slot to compensate for clock drifts between nodes (t_g). Thus, the minimum size of a transmission slot is given as:

$$T_{st} = t_{ts} + t_{xm} + t_{pm} + t_{xa} + t_{pa} + t_g \tag{34}$$

In our testbed the slot size was 10 ms, which allows starting the communication, sending a data packet with 128 Bytes of maximum size and receiving the acknowledgment.

Chapter 8

Performance and Debugging

Operations performance monitoring is important in contexts with timing constrains. For instance, in the previous chapter, we propose an algorithm to plan for timing guarantees in distributed control systems with heterogeneous components. In this chapter we define measures and metrics for surveillance of expectable time bounds and an approach for performance monitoring bounds on these metrics.

This surveillance can be used in any distributed system to verify performance compliance. Assuming that we have monitoring or closed-loop tasks with timing requirements, this allows users to constantly monitor timing conformity.

In the context of networked control systems with heterogeneous components and non-real-time parts, where latencies were planned, this allows the system to monitor and determine the conformity with timing requirements.

We define measures and metrics which create an important basis for reporting the performance to users and for helping them to adjust deployment factors. Those sets of measures and metrics are also used for debugging, using tools and mechanisms to explore and report problems. We also propose an approach to monitor the operation timings.

The time bounds and guarantees must be based on well-defined measures and metrics. In Sections 8.1 and 8.2 we discuss these measures and metrics. Section 8.3

discusses metric information for analysis. We will discuss time bounds setting and time bounds metrics, and how message loss information is collected.

The measures can be taken per message or statistically per time periods. We describe how both alternatives are used in the approach. Measures can also be classified. Each message is classified according to each time bound as in-time, out-of-time, waiting-for or lost-message.

Section 8.4 describes the addition of debugging modules to the MidSN architecture and Sections 8.5 and 8.6 describe debugging node component and operation performance monitor component. It is described how the performance information is collected and processed. An example of operation performance monitor UI is presented, which allows users to evaluate the performance.

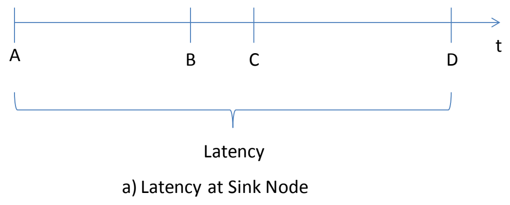
8.1. Measures

Operation timing issues in terms of monitor and closed-loops control can be controlled with the help of two measures, which we denote as **Latency** and **Delay of Periodic Events**.

8.1.1. Latency

Latency consists of the time required to travel between a source and a destination. Sources and destinations may be any site in the distributed system. For instance, the latency can be measured from a WSN leaf node to a sink node, or from a WSN sensing node to a computer, control station or backend application. It may account for the sum of all components, including queuing and the propagation times, and it is additive – the latency between two points is the sum of the latencies in the path going through all intermediate points that may be considered between the two points. Figure 8.1a) shows an example of how latency is measured when a leaf node transmits its data to a sink node in a 3-3 tree topology.

A represents the instant when message transmission starts. The transmission takes several milliseconds and the message is received by an intermediate node at instant *B*. The intermediate node saves the message in a queue until it gets its slot transmission time. When the slot is acquired (instant *C*) the message is transmitted to the next upper level and reaches the sink node at instant *D*. In the example 6a) the latency is given by the sum of all latency parts ((*B* – *A*) gives the latency from leaf node to the intermediate node; (*C* – *B*) gives the queue latency and (*D* – *C*) gives the latency from intermediate node to sink node).



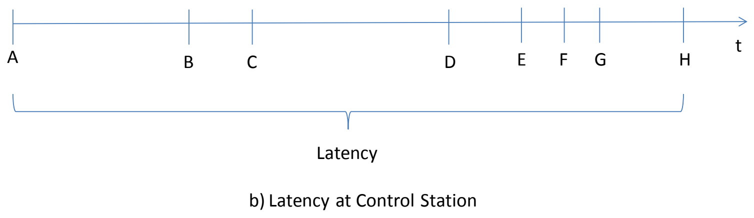


Figure 8.1 – Latency diagram

Figure 8.1b) shows the latency from the same node considered in Figure 8.1a), but in this case the destination point is a control station. It includes the previous latency and adds the latency from the sink node to the control station. Latency from sink node to the control station is given by the sink node to gateway latency (*E* – *D*), plus gateway processing time (*F* – *E*), plus LAN transmission (*G* – *F*), plus control station processing time (*H* – *G*).

8.1.2. Delay of periodic events

Given a specific periodic event, such as monitoring or closed-loop control, the delay will be the extra time taken to receive a message with respect to the predefined periodic reception instant. Figure 8.2 shows how the delay is measured.

The instant E_{-1} represents the instant of the last occurrence of a specific periodic event. It is expected to receive that event with a specific cadence (period). E_{Exp} represents the instant of expected reception of the event. But the event may arrive delayed (instant E). So, the delay is the time elapsed between E_{Exp} and E instants.

Each period time interval is measured from the last reception instant to the next reception instant.

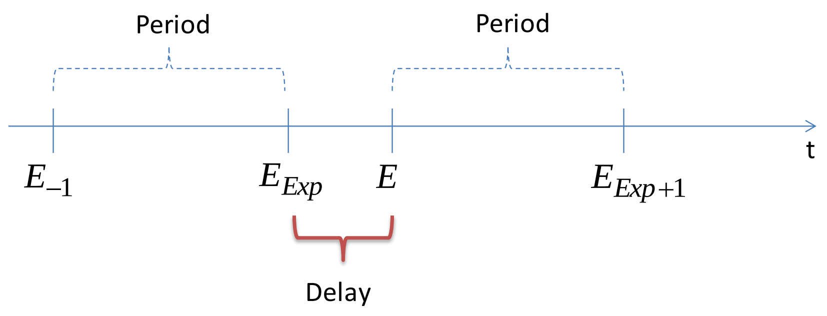


Figure 8.2 – Delay illustration

8.2. Metrics

Given the above time measures, we define metrics for sensing and control. The metrics allow us to quantify timing behaviour of monitoring and closed-loops.

8.2.1. Monitoring latencies

Monitoring latency is the time taken to deliver a value from sensing node to the control station, for display or alarm computation. If a value determines an alarm, it will be the time taken since the value (event) happens to the instant when the alarm is seen in the control station console. Since the value must traverse many nodes and parts of the system, this latency can be decomposed into latencies for each part of the path. It is useful to collect the various parts of the latency – Acquisition latency (sensor sample latency plus latency associated with waiting for data transmission instant), WSN latency (time for transmission between leaf node and sink node), latency for sink-gateway (the time taken for the message to go from the sink to the gateway, plus gateway processing time), latency for middleware transmission (e.g transmission between gateway and Control Station), Control Station processing latency and end-to-end latency (leaf node to Control Station). The following latency metrics are therefore all considered:

- Acquisition latency;
- WSN latency;
- WSN to Gateway interface latency;
- Middleware latency;

- Control Station processing latency;
- End-to-end latency;

8.2.2. Monitoring delays

The delay measure was already defined as the amount of extra time from the moment when a periodic operation was expected to receive some data to the instant when it actually received. When users create a monitoring task, they must specify a sensing rate. The control station expects to receive the data at that rate, but delays may happen in the way to the control station, therefore delays are recorded.

8.2.3. Closed-loop latency for asynchronous or event-based closed-loops

In this case, the closed-loop latency is the time taken from sensing node to actuator node, passing through the supervision control logic. It will be the time taken since the value (event) happens at a sensing node to the instant when the action is performed at the actuator node. Since the value must cross several parts of the system, this latency can be decomposed into latencies for each part of the path: upstream part (from sensing node to the control station) and downstream part (from control station to the actuator). The first part (upstream) is equivalent to monitoring latency and can be sub-divided in the same sub-parts. The second part (downstream) corresponds to the path used by a command to reach an actuator. The following latency metrics should be considered to determine the closed-loop latency:

- Acquisition latency;
- WSN upstream latency;
- WSN to Gateway interface latency;
- Middleware latency;
- Control Station processing latency;
- Middleware latency;
- Gateway to WSN interface latency;
- WSN downstream latency;
- Actuator processing latency;
- End-to-end latency;

8.2.4. Closed-loop latency for synchronous or periodic closed-loops

Synchronous or periodic closed-loops can be associated with two latencies: Monitoring latency and Actuation latency. The Monitoring latency can be defined as the time taken from sensing node to the supervision control logic (monitoring latency). The Actuation latency corresponds to the time taken to reach an actuator. We also define an end-to-end latency as the time from the instant when a specific value is sensed and the moment when an actuation is done which incorporates a decision based on that value.

The following latency metrics should be considered to determine the closed-loop latency for synchronous or periodic closed-loops:

- Acquisition latency;
- WSN upstream latency;
- WSN to Gateway interface latency;
- Middleware latency;
- Wait for the actuation instant latency;
- Control Station processing latency;
- Middleware latency;
- Gateway to WSN interface latency;
- WSN downstream latency;
- Actuator processing latency;
- End-to-end latency;

8.2.5. Closed-loop delays

In synchronous closed-loop operations, actuation is expected within a specific period. However, operation delays may occur in the control station and/or command transmission. The closed-loop delay is the excess time.

In asynchronous closed-loops, there can be monitoring delays. This means that a sample expected every x time units may be delayed.

In summary: The proposed measures and metrics include:

- Delays:
    - Monitoring delay;
    - Synchronous closed-loop actuation delay;
- Latencies:
    - Monitoring latencies:
        - All - End-to-end latency;
        - Acquisition latency;
        - WSN latency;
        - WSN to Gateway interface latency;
        - Middleware latency;
        - Control Station processing latency;
    - Closed-loop latencies:
        - All - CL End-to-end latency;
        - Acquisition latency;
        - WSN upstream latency;
        - WSN to Gateway interface latency;
        - Middleware latency;
        - Wait for the actuation instant latency; (synchronous)
        - Control Station processing latency;
        - Middleware latency;
        - Gateway to WSN interface latency;
        - WSN downstream latency;
        - Actuator processing latency;

8.3. Metric Information for Analysis

For each of the previous metrics, we can have per-message values, per-time interval statistics, as well as per-time interval bounds statistics. In this section we define bounds and describe how each message is classified according to each bound. Then we describe how to count message and packets losses.

8.3.1. Bounds: waiting, in-time, out-of-time, lost

Bounds over latency and delay measures allow users to configure bounds violation alarms and to keep information on how often the bounds are broken. A bound may be defined as a threshold over some measure. It specifies an acceptable limit for that measure. We classify the events with respect to that bound as:

Waiting – the process is waiting to receive the event, its final status with respect to the bound is yet undefined;

In-time – the event arrived, and the time measure is within the specified bound;

Out-of-time – the event arrived, and the time measure is out of the bound;

Lost – the event didn’t arrive, and the timeout has expired.

Figure 8.3 shows a state diagram with message classification. Each event is classified as “waiting” until reception, or lost, if timeout was exceeded. When an event is received within a specific bound, it is classified as “In-time”. If an event was received but the bound was exceeded, it is classified as “out-of-time”. Lastly, if an event is expected but is not received and the timeout has elapsed, it is classified as “lost”.

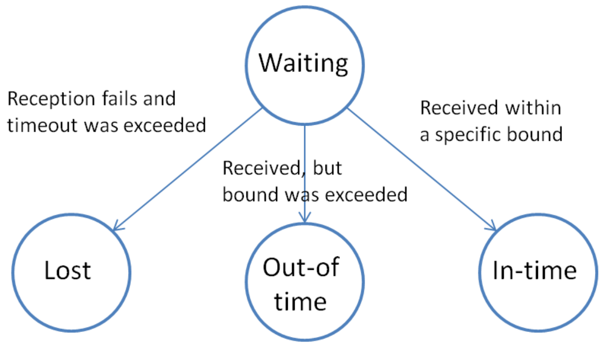

Waiting
Reception fails and timeout was exceeded
Received within a specific bound
Received, but bound was exceeded
Lost
Out-of time
In-time

Figure 8.3 – Event state diagram

Figure 8.4 shows an example of bound specification and corresponding event classification (in-time and out-of-time). Figure 8.4a) shows an event (*E*) that arrives at destination with some delay but within a specified bound. In this case, the event is classified as in-time. Figure 8.4b) shows an example where an event is classified as out-of-time. In this case the delay is greater that the specified bound.

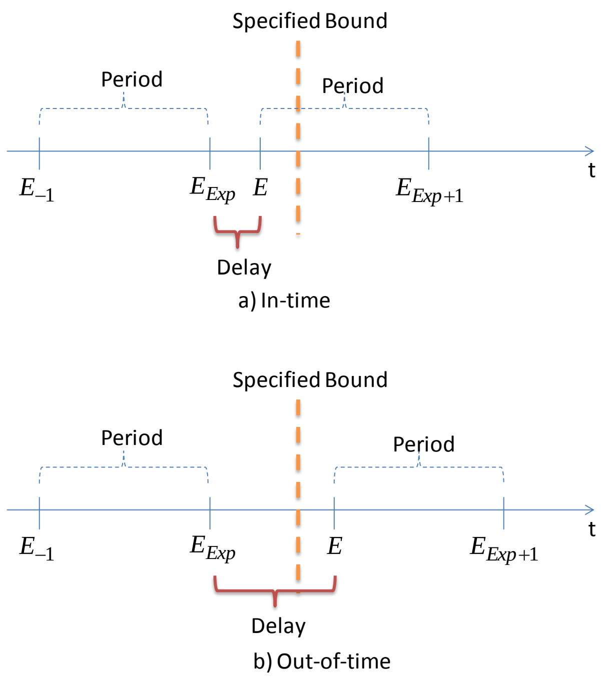


Figure 8.4 – Bounded event classification

It should be possible to specify delay bounds, latency bounds or both.

8.3.2. Messages and packet losses

Besides operation performance monitoring, the number or ratio of lost messages or packets are also important measures, revealing problems that may be due to several factors, including interference and disconnection, but also messages being dropped somewhere in the system when some transmission buffer or queue fills up for lack of servicing capacity. For instance, in the context of a pre-planned network with TDMA, a sink node has to service the data coming from one node in a single slot of time. This includes receiving the data, sending it to the gateway through some serial interface and processing downstream command messages coming from the gateway. At some point it may overload and drop messages.

The simplest and most common approach to count end-to-end message losses is based on sequence numbers and a timeout (configurable).

Given a timeout, defined as the time that elapsed since a message with a specific sequence number arrived, if the time expires and some lower sequence number is missing, that corresponding message is considered lost. Figure 8.5 shows an example of a timeline to evaluate if a message is lost or not.

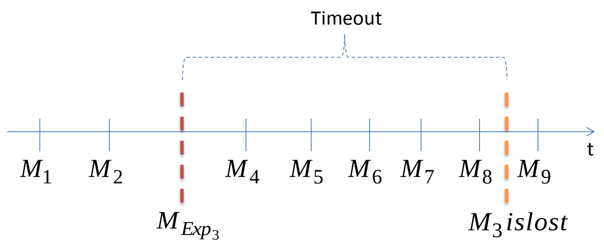


Figure 8.5 – Message lost evaluation

In the above figure we can see that Message 3 (M_3) doesn´t arrive at the control station. The M_{Exp_3} shows the instant when M_3 should be received. M_3 is considered lost when the timeout is exceeded.

In closed-loop control scenarios, actuation command losses are accounted for by means of timeout in acknowledgment message. Each actuation command has an acknowledgement message associated with it. A command is marked as lost when the acknowledgement is not received by the command issuer in a certain timeout after the command message was sent.

Network protocol-level count of packet losses can also be used to analyse losses.

8.3.3. Statistics: avg, stdev, max, min, percentile

Statistic information can be computed for each part of the system (e.g. WSN part, gateway interface, gateway processing) and it will be important to diagnose timing problems and where they come from. The following types of statistical information are included:

Maximum, this is useful to detect the worst case time measure, one that should not be over a limit.

Percentile (e.g. P99, P95, P90): it is useful to characterize the measure under evaluation, while removing outlier events (e.g. large delays existing less than 0.01% of the cases).

In systems with strict timing requirements, maximum and percentile measures are the most important.

Averages and Standard Deviation, which is another the common metric used to evaluate the stability of time measures.

Minimum: It is a less important metric in our context, but provides the best case time measure.

These are important metric data items for reporting the performance to users and for helping users adjust their deployment factors. Another important aspect is that this statistic information (in short avg, stdev, max, percentile, min) can be taken for different periods of time, for each node or part of the system, and it is important to keep those with different granularities, so that it becomes possible to arrive at the trouble spots. Finally, it is useful to correlate those with operating system and networks performance information, to detect the culprit of the problems.

8.4. The Debugging Module of MidSN

In the previous sections we discussed measures and metrics useful to evaluate operation performance. In this section we will discuss the debugging module present in the MidSN architecture.

The Debugging module (DM) stores all information concerning node operation (e.g. execution times, battery level) and messages (e.g. messages received, messages transmitted, transmission fails, transmission latencies). This information is stored inside the node. It can be stored either in main memory, flash memory or other storage device.

DM is an optional module that can be activated or deactivated. It generates a debugging report, either by request or periodically, with a configurable period.

DM has two modes of operation:

- Network debugging – the DM runs in all nodes and keeps the header information of messages (source ID, destination ID, Msg Type, Ctrl Type and Sequence

Number (Message format – Figure E.1)), where it adds timestamps corresponding to arrive and departure instants. It also keeps information about MidSN-NC execution. After, this information is sent periodically or by request to the Performance Monitor (described in the next section), which is able to calculate metrics. This operation mode may be deactivated in constrained devices, because it consumes resources such as memory and processing time.

- High-level operation debugging – instead of collecting, storing and sending all information to the Performance Monitor, the DM can be configured to only add specific timestamps to messages along the path to the control station.

Assuming a monitoring operation in a distributed control system with WSN sub-networks, where data messages are sent through a gateway, the DM can be configured to add timestamps in the source node, sink node, gateway and control station. Figure 8.6 illustrates nodes, gateways and a control station in that context.

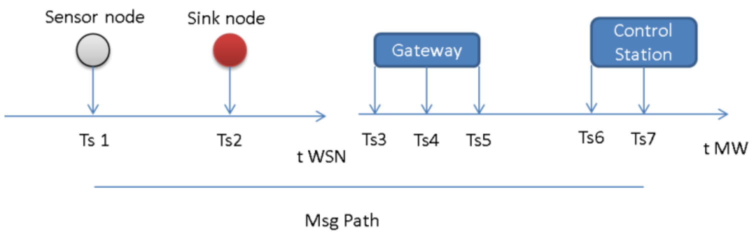


Figure 8.6 – Message path - example

The approach assumes that WSN nodes are clock synchronized. However, they may not be synchronized with the rest of the distributed control system. Gateways, computers and control stations are also assumed clock synchronized (e.g. the NTP protocol can be used).

In Figure 8.6, the DM starts by adding a generation timestamp (source timestamp) in the sensor node (*Ts1*). When this message is received by the sink node, it adds a new timestamp (*Ts2*) and indicates to the gateway that a message is available to be written in the serial interface. Upon receiving this indication, the gateway keeps a timestamp that will be added to the message (*Ts3*), and the serial transmission starts. After concluding the serial transmission, the gateway takes note of the current timestamp (*Ts4*) and adds *Ts3* and *Ts4* to the message.

Upon concluding this process and after applying any necessary processing to the message, the gateway adds another timestamp (*Ts5*) and transmits it to the control station. When the message is received by the control station, it adds a timestamp (*Ts6*), processes the message and adds a new timestamp (*TS7*), which indicates the instant of message processing at the control station was concluded. After that, at the control station, the Performance Monitor module (described in the next section) receives the message and, based on the timestamps that come in the message, it is able to calculate metrics.

If there is only one computer node and the control station, there will only be *Ts1*, *Ts6* and *Ts7*.

8.5. The Performance Monitor Module and UI

In this section we describe the Performance Monitor module (PMM), which debugs operations performance in the heterogeneous distributed system. The PMM stores events (data messages, debug messages), latencies and delays into a database. It collects all events when they arrive, computes metric values, classifies events with respect to bounds, and stores the information in the database. Bounds should be configured for the relevant metrics.

Assuming the example shown in Figure 8.6, PMM collects the timestamps and processes them it to determine partial and end-to-end latencies.

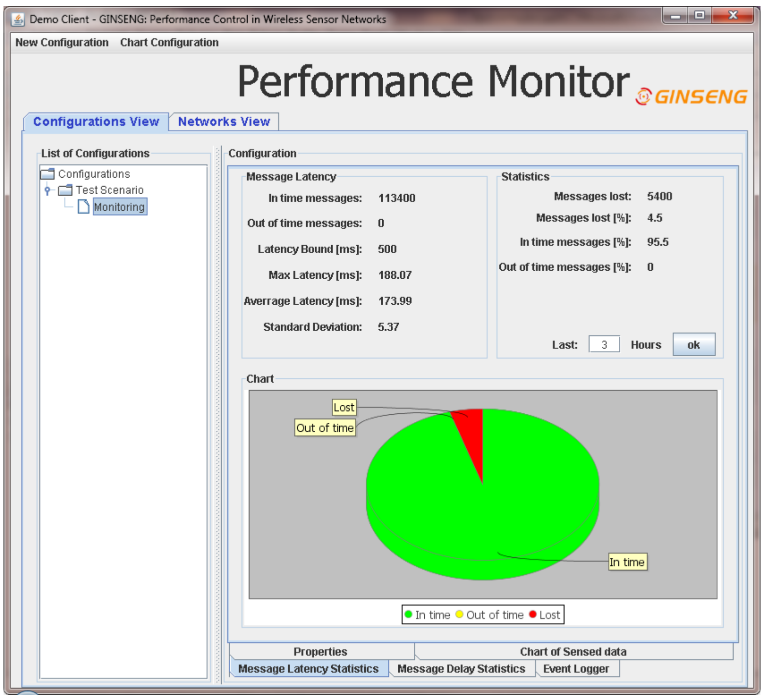
The following partial latencies are calculated:

- WSN upstream latency (Ts2 – Ts1)
- WSN to Gateway interface latency (Ts4 – Ts3)
- Middleware latency (Ts6 – Ts5)
- Control station latency (Ts7 – Ts6)
- End-to-end ((Ts2 – Ts1) + (Ts4 – Ts3) + (Ts6 – Ts5) + (Ts7 – Ts6))

After concluding all computations, PMM stores the following information in the database: Source node id, Destination node id, Type of message, MsgSeqId, [Timestamps], partial latencies, end-to-end latency. This information is stored for each message, when the second operation mode of debugging is running. When the first operation mode of the debugging component is running, a full report with link-by-link information and end-to-end information is also stored.

The PMM user interface shows operations performance data, and alerts users when there is a problem detected by metric exceeds bounds. Statistical information is also shown and is updated for each event that arrives or for each timeout that occurs.

Figure 8.7 shows a screenshot of PMM. We can see how many events (data messages) arrived in-time, out-of-time (with respect to defined bounds), and the corresponding statistical information. This interface also shows a pie chart to give an overall view of the performance.



Message Latency

In time messages:	113400
Out of time messages:	0
Latency Bound [ms]:	500
Max Latency [ms]:	188.07
Averrage Latency [ms]:	173.99
Standard Deviation:	5.37

Statistics

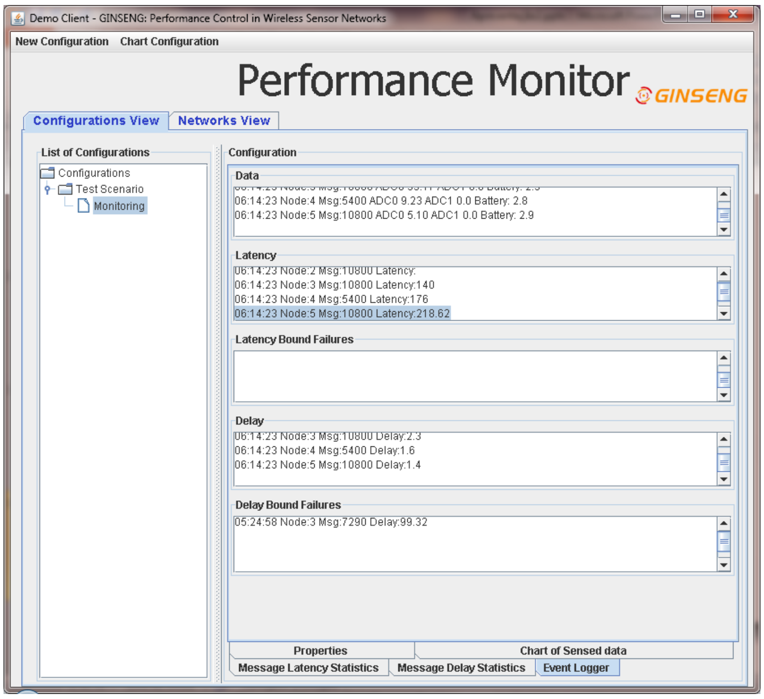
Messages lost:	5400
Messages lost [%]:	4.5
In time messages [%]:	95.5
Out of time messages [%]:	0

Last: 3 Hours ok

Figure 8.7 – PMM user interface

Figure 8.8 shows the event logger of PMM. This logger shows the details on failed events. A list of failures is shown and the user can select one of each and see all details, including the latency in each part of the distributed control system.

When a problem is reported by the PMM, the user can explore the event properties (e.g. delayed messages, high latencies) and find where the problem occurs. If a problem is found and the debugging report is not available at the PMM, nodes are requested to send their debugging report. If a node is dead, the debugging report is not retrieved and the problem source may be due to the dead node. Otherwise, if all reports are retrieved, the PMM is able to detect the message path and check where it was discarded or where it took longer than expected.



Data:

```
06:14:23 Node:4 Msg:5400 ADC0 9.23 ADC1 0.0 Battery: 2.8
06:14:23 Node:5 Msg:10800 ADC0 5.10 ADC1 0.0 Battery: 2.9
```

Latency:

```
06:14:23 Node:2 Msg:10800 Latency:
06:14:23 Node:3 Msg:10800 Latency:140
06:14:23 Node:4 Msg:5400 Latency:176
06:14:23 Node:5 Msg:10800 Latency:218.62
```

Delay:

```
06:14:23 Node:3 Msg:10800 Delay:2.3
06:14:23 Node:4 Msg:5400 Delay:1.6
06:14:23 Node:5 Msg:10800 Delay:1.4
```

Delay Bound Failures:

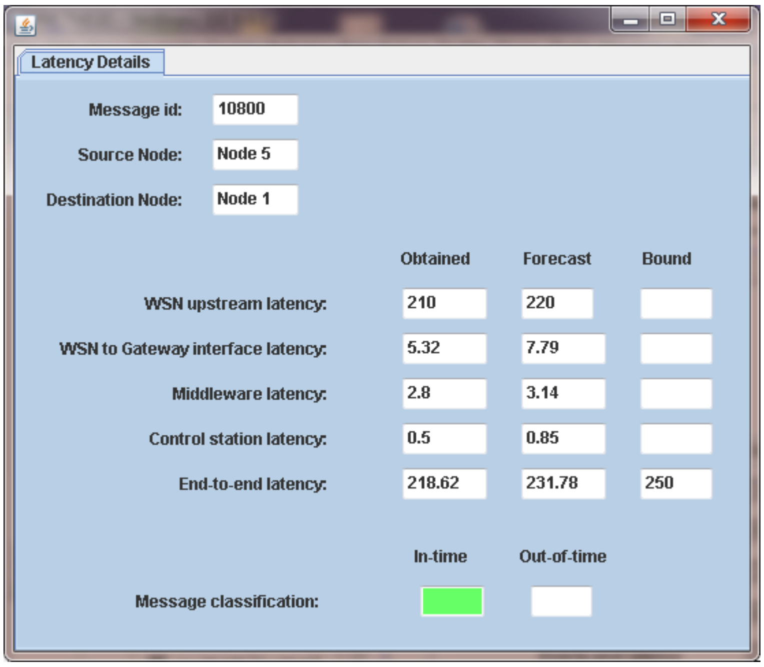
```
05:24:58 Node:3 Msg:7290 Delay:99.32
```

Figure 8.8 – PMM user interface – event logger

PMM allows users to select one message and see all details, including the latency in each part of the distributed control system. Figure 8.9 shows the details of a message.

In the interface of Figure 8.9, users can see the latency per parts, as well as the end-to-end latency. This interface also includes information about the forecasted values and bounds. Each bound is specified by the user and can be specified for all parts or only for specific parts. In the example of Figure 8.9, only an end-to-end bound is defined.

Lastly, this interface also includes information about message classification. This information is filled only when the end-to-end latency bound is defined.


Latency Details

Message id:	10800		
Source Node:	Node 5		
Destination Node:	Node 1		

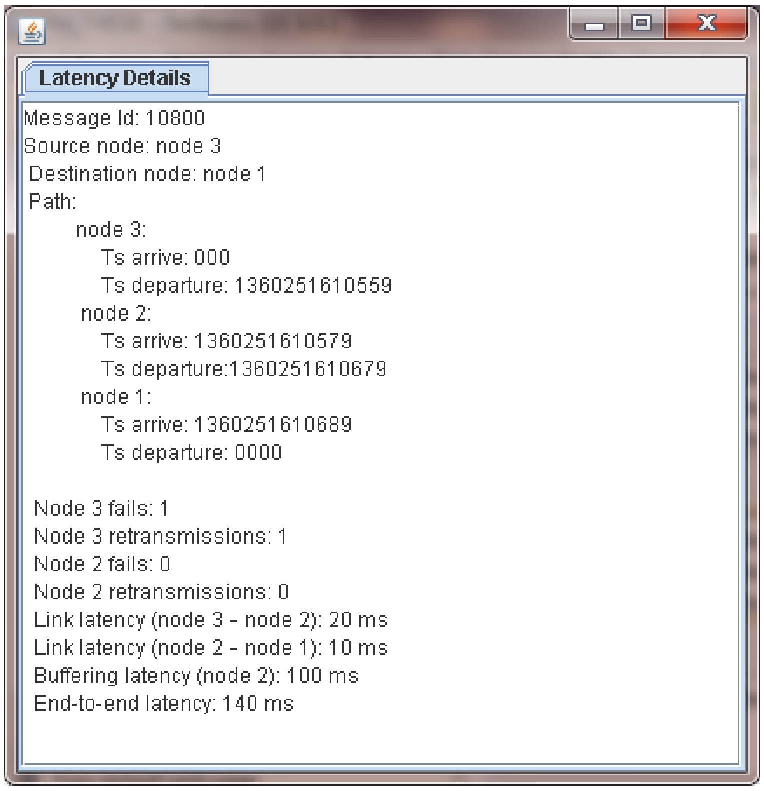
	Obtained	Forecast	Bound
WSN upstream latency:	210	220	
WSN to Gateway interface latency:	5.32	7.79	
Middleware latency:	2.8	3.14	
Control station latency:	0.5	0.85	
End-to-end latency:	218.62	231.78	250

	In-time	Out-of-time
Message classification:	🟩	

Figure 8.9 – PMM user interface – latency details

Since DM can be configured to keep information about all messages and MidSN-NC operations, reporting those to the PM, PM is able to compute metrics and show report information. Figure 8.10 shows an example of a message travelling from node 1 to node 10. This also allows users to inspect details when they detect a problem by looking at the logger of the Performance Monitor (PM).

In this example we can see the arriving and departure timestamps, fails and retransmissions per node. Based on the timestamps collected along the path, PMM computes link latencies for each link, buffering latencies, and the end-to-end latencies. Each link latency is determined through the computation of the difference between arriving and departure timestamps of receiving node and sending node, respectively. The buffering latency is determined based on the arriving and departure times of each node. This time represents how long the message is kept in a node. Lastly, the end-to-end latency is the sum of all parts of the latencies.



```
Message Id: 10800
Source node: node 3
 Destination node: node 1
 Path:
         node 3:
              Ts arrive: 000
              Ts departure: 1360251610559
          node 2:
              Ts arrive: 1360251610579
              Ts departure:1360251610679
          node 1:
              Ts arrive: 1360251610689
              Ts departure: 0000

  Node 3 fails: 1
  Node 3 retransmissions: 1
  Node 2 fails: 0
  Node 2 retransmissions: 0
  Link latency (node 3 - node 2): 20 ms
  Link latency (node 2 - node 1): 10 ms
  Buffering latency (node 2): 100 ms
  End-to-end latency: 140 ms
```

Figure 8.10 – PMM user interface – latency details

Chapter 9

Evaluation of MidSN

In this chapter, we present the results of experimental evaluation of MidSN and the mechanisms that were proposed in the thesis. The objectives are to evaluate metrics related to:

- Deploying and using MidSN-NC for different platforms;
- Comparison of performance, code and memory requirements over different platforms;
- Operating in a constrained device, i.e. whether the code fits into a constrained device and comparison of RAM versus flash operations in terms of time and energy, since constrained devices typically have low amounts of RAM;
- Battery lifetime issues;
- Latencies in a networked control testbed.

The results show that MidSN is a middleware with a small footprint and supports the operations defined in the architecture. We also show the system running on different platforms, where latencies are assessed over more than one platform.

Section 9.1 reports results concerning evaluation of MidSN-NC for multiple platforms. It concerns memory footprint (code size and RAM) and execution times. Due to many WSN device limitations, in Section 9.2 we analyse some issues raised by those limitations. Section 9.3 reports results concerning latencies and execution times extracted from a network built with several different classes of nodes.

Results are shown as charts, however for further completeness Appendix I details those values in tables formats.

9.1. Evaluation of MidSN-NC for Multiple Platforms

As MidSN-NC can be deployed on WSN devices or more powerful nodes, such as computers, in this section we will evaluate the MidSN-NC when implemented in computer, Raspberry PI, TelosB and Arduino platforms. Those platforms had the following characteristics:

- **Computer** – platform based on an Intel Pentium D CPU, running at 3.4 GHz. It has 2 GB of RAM and an Ethernet 10/100 BaseT connector.
- **Raspberry PI** – platform based on an ARM1176JZFS with 512MB of RAM memory and an Ethernet 10/100 BaseT connector. A SD card is used to load the Operating System and to store application data.
- **TelosB** – platform based on a MSP430 16-bit CPU with an on-board 8 MHz oscillator. It includes a CC2420 radio chip, 48 kB of program flash and 10 kB of RAM.
- **Arduino (Mega)** – platform based on an ATmega2560 with a 16 MHz oscillator. It has 256 KB of program flash where 8 KB are used by boot loader, and 8 kB of RAM. A Wifly module is used to connect it to a Wi-Fi network.

Both computer and Raspberry PI platforms run a Linux operating system with java support. The computer runs Ubuntu 10.2 with JDK 7, while the Raspberry runs the debian6 for Raspberry PI operating system.

The MidSN-NC implementation for computer and Raspberry PI platforms is written in Java. The embedded java virtual machine (Java SE Embedded 7) is used to run MidSN-NC in Raspberry PI platform. The TelosB implementation is written in

Contiki-C language and is supported by the Contiki operating system, while the Arduino version is written in C++ language using the Arduino IDE version 1.0.

9.1.1. Development and porting between platforms

The first version of MidSN-NC was developed in Contiki-C for the TelosB platform using the Contiki operating system. This derives from the GINSENG project, where the TelosB platform was the choice to be applied in the refinery.

As MidSN-NC architecture is divided in several modules, we created a structure of folders and files that represented each module and its functionalities. Each folder had the name of the corresponding module.

All functionalities were isolated from the operating system and hardware by using drivers which abstract MidSN-NC implementation. Each driver was developed according to the specification of Chapter 5.

Concerning RAM memory, we needed to take in consideration the amount of memory used by temporary variables and arrays. We needed to use pointers to data and configuration structures to avoid duplication of variables and to prevent memory overflow. A timer structure was created and limited to 5 scheduled events, and the number of parallel threads was reduced to avoid memory leaks due to insufficient memory and synchronization problems.

Since computer and Raspberry PI platforms are less constrained devices, their implementation is less stringent. MidSN-NC implementation for these platforms was quite simple. Once the Contiki-C version was developed, we only needed to follow the code structure and write it in java. However, the Raspberry PI implementation introduced some minor modifications over the computer java version. Due to the java embedded virtual machine, its implementation needs to take into account available libraries.

Lastly, the Arduino implementation was the hardest. Similar to Java version, we followed the Contiki-C code structure but in the Arduino implementation we needed to develop many functions, especially at driver level, since the Arduino doesn’t include a full operating system and the IDE doesn't include libraries as Java does. For instance, some functionalities of MidSN-NC are based on scheduling timer events. In the Arduino implementation we needed to develop all structures to create the scheduling, checking elapsed time and generating events to indicate to the MidSN-NC when an event arrives.

From these implementations and porting we concluded that the reference architecture of MidSN is quite helpful, since it specifies which modules to implement and how should work. Porting to new platforms is also simple.

9.1.2. Memory and performance

In this sub-section we will detail the amount of memory needed to implement MidSN-NC in described platforms, and will confront it with the memory available in the device.

Figure 9.1 shows the amount of memory needed by each component of MidSN-NC in the different platforms.

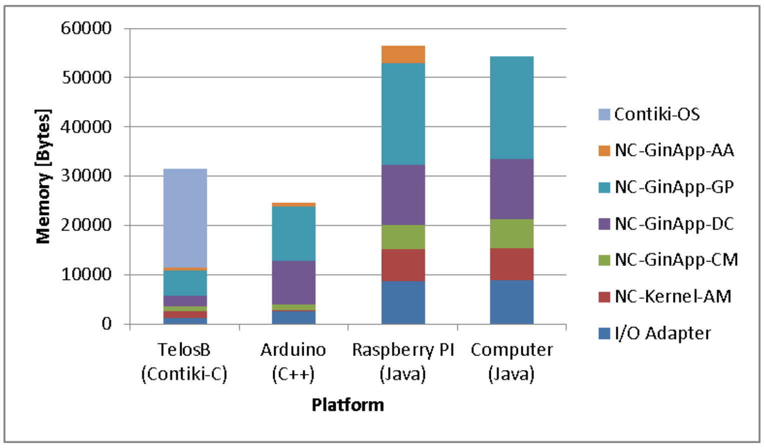


Figure 9.1 – Programming memory consumption for all platforms

From Figure 9.1 we can conclude that MidSN-NC implementation was significantly small to fit all devices that were tested. Implementations for either computer or Raspberry nodes need less than 60 KB (without operating systems). These consume more space than implementations for other platforms because they are java-based, but both computers and Raspberry PI resources do not pose any constraints on such code sizes. The Arduino implementation is smaller than the other ones, because it is written in C++ and it is not loaded with a full Operating System. From the figure we can also conclude that the processor component (NC-GinApp-GP) is the one that needs more code memory in all platforms.

Another important issue for some embedded devices with constrained resources is the quantity of RAM memory needed to run each implementation. Figure 9.2 shows the amount of memory needed by each component and in total for each platform.

From Figure 9.2 we conclude that the Contiki-C version is the implementation that needs less RAM. However, this is not accounting for the RAM used by the Contiki Operating System. In general, the amount of RAM needed is small and fits nicely into each platform. For instance, 2.6 kBytes of RAM memory are used in the Contiki-C implementation, 5.6 kBytes in the Arduino, 13.2 kBytes in the Raspberry PI node and 13.7 kBytes in a computer node. Similar to programming code, the NC-GinApp-GP is the component that needs more RAM memory. Prior to Arduino Mega we tried to fit the design into Arduino Uno, but MidSN-NC did not fit into its 2 kBytes data memory.

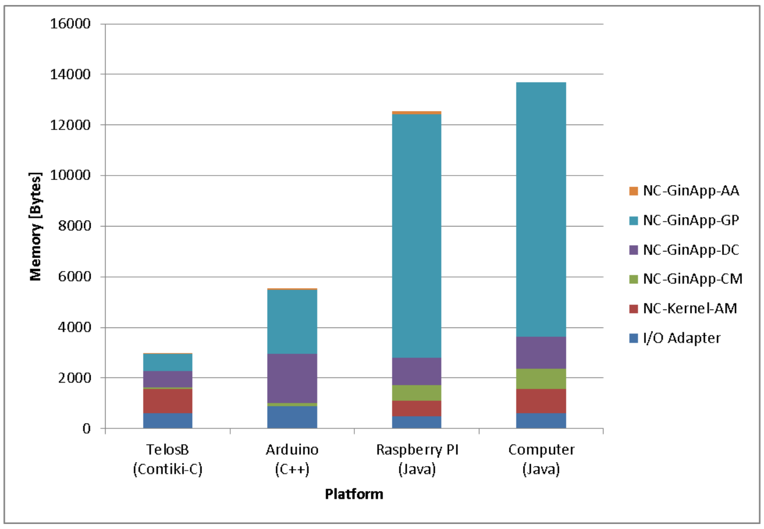


Figure 9.2 – RAM memory consumption for all platforms

Figure 9.3 shows the time to compute an average in each platform. Due to scarceness of resources in TelosB and Arduino platforms, the number of tuples used to do the computation is limited to 50. Both computer and Raspberry PI platforms were evaluated with 1000 tuples as maximum.

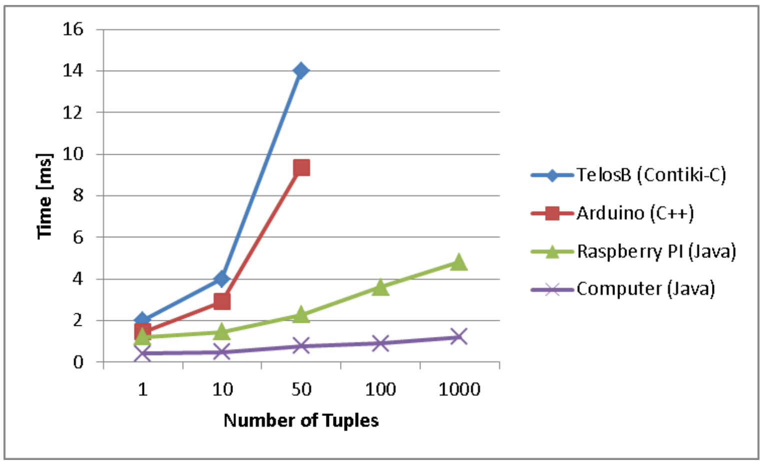


Figure 9.3 – Time required computing an average

From Figure 9.3 we conclude that TelosB is the slowest platform. It has the slowest oscillator that runs at 8 MHz, while Arduino, Raspberry PI and the computer operate at 16 MHz, 700 MHz and 2.5 GHz, respectively. The Raspberry PI and computer platforms also have more RAM and computational capabilities, which contributes to reduce the execution time.

Lastly, Figure 9.4 shows the operation execution time for a window with 50 tuples. The select AGGREG represents any operation with any number of aggregation functions over measures, where the aggregation can be any of (average, maximum, minimum, count, sum, and variance). The same time is taken to compute any such metric independently of which aggregation functions are specified because in all cases aggregations are computed incrementally as the values are being scanned.

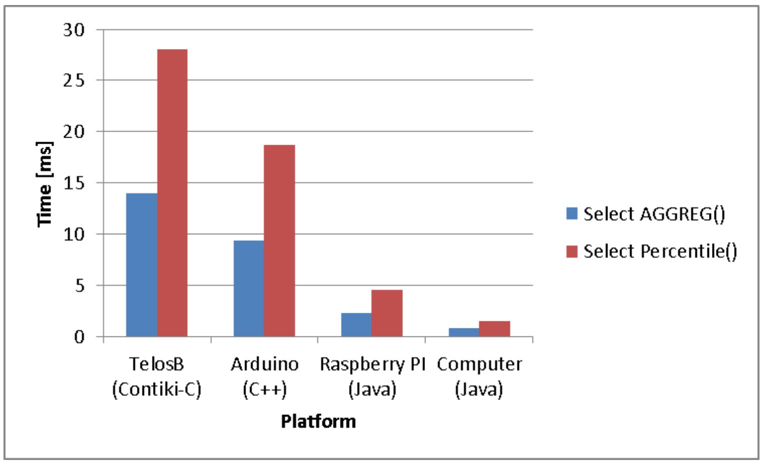


Figure 9.4 – Time required per operation over a stream in memory

The execution times are similar (high) to TelosB and Arduino platforms, while the Raspberry PI and computer platforms are much faster. Again, this is due to internal oscillators and resources available to do the computation.

The computation of percentiles requires more time that other statistical measures because the data tuples must be ordered. These computations were implemented with the insertion sort algorithm in our prototype. The computation of percentiles consumes about 27 ms in TelosB, 18 ms in Arduino, 4.5 ms in Raspberry and 1ms in a computer.

9.2. Operation Processing in Constrained Devices

Typically, many WSN devices have limited memory and computation capabilities. In this section we analyse some of the issues raised by those constraints, taking a TelosB plus Contiki platform as the case study.

9.2.1. Memory footprint

In this sub-section we will detail the amount of memory needed when MidSN-NC is deployed in the TelosB, and will confront it with the memory available in the device. Table 9.1 shows the amount of memory needed by each component of MidSN-NC when it is ported to that platform using Contiki-OS.

Table 9.1 – Program memory consumption

Component	ROM Memory [Bytes]
I/O Adapter	1136-1260
Rime Driver	768
IP Driver	890
NC-Kernel-AM	1344
NC-GinApp-CM	880
NC-GinApp-DC	2270
Main Memory	1639
Flash Memory	631
NC-GinApp-GP	5104
Events	1546
Computation	3558
NC-GinApp-AA	544
Contiki-OS	20009
Total	31287

The Contiki implementation of MidSN-NC consumes 31.3 KB of program memory, where about 11KB is occupied by MidSN-NC and the rest is operation system.

Our implementation is based on several modules. Figure 9.5 shows the program memory distribution used by MidSN-NC.

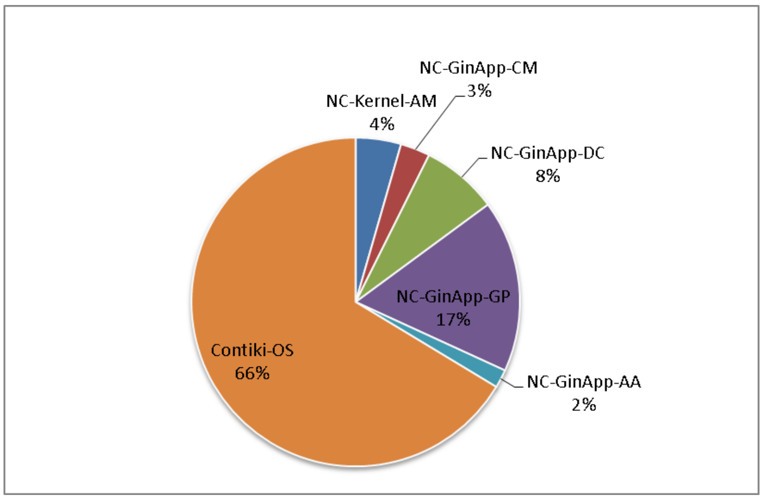


Figure 9.5 – Program memory distribution

As seen in Table 9.1 and Figure 9.5, most of the code memory (66%) is occupied by the operating system. Concerning MidSN, 17% is occupied by the processor module. This module has capabilities to manage periodic events and perform computation over streams.

The data collector module (NC-GinApp-DC) used to manage all data coming from sensor or from other nodes, occupies 8% of the total code memory. This module includes capabilities to manage data in main memory and in external memory (flash memory).

The NC-Kernel-AM module, which offers support to receive agents over-the-air (submitted by users), store them in flash memory or drop them from the node, occupies 4% of code memory.

The NC-GinApp-CM module occupies 2% of memory. The memory occupied by this module is proportional to the API functionalities offered by the MidSN. This value results from the implementation of the functionalities described in Chapter 5 (the amount of memory can increase if more functionalities are added).

Lastly, there is the acquisition module (NC-GinApp-AA), which consumes 2% of the memory. This module includes the necessary drivers to collect sensor data.

Another important issue in constrained devices such as TelosB motes is RAM memory consumption. This platform has 10kB of memory that must be shared by the operating system and the MidSN-NC middleware.

Table 9.2 shows the amount of RAM memory needed by each component of MidSN-NC when it is ported to TelosB using Contiki-OS. From Table 9.2 we conclude that our middleware needs 2.6 kB, which fits the memory requirements of the patform. The TelosB platform has 10 kB of RAM memory, but about 6.5 kB is consumed by the Contiki operating system, which means that only about 900 Bytes are available to create streams and store data tuples inside them.

Table 9.2 – RAM memory consumption

Component	RAM Memory [Bytes]
I/O Adapter	366-486
Rime Driver	240
IP Driver	360
NC-Kernel-AM	631
NC-GinApp-CM	62
NC-GinApp-DC	636
Main Memory	454
Flash Memory	182
NC-GinApp-GP	664
Events	156
Computation	508
NC-GinApp-AA	38

One question that can arise when we analyse these values is “*if develop code from scratch for an application, how much memory will it need, and how does that compare with MidSn-NC alternative?*”

Table 9.3 shows that, effectively, the memory used by a specific application is less, but with MidSN-NC we have flexibility to configure or modify the operation on-the-fly without any programming needs. Furthermore, MidSN-NC needs only 2.6kB of RAM, which is enough for most WSN node platforms.

Table 9.3 – Programming and RAM comparison between MidSN-NC and hand-coded application

Operation	ROM [Bytes]	RAM [Bytes]
MidSN-NC	11278	2397
Hand-coded	1448	467
Hand-coded with average computation (10 samples)	2224	647
Hand-coded with average computation (100 samples)	2224	827

9.2.2. Performance and energy consumption: RAM versus Flash

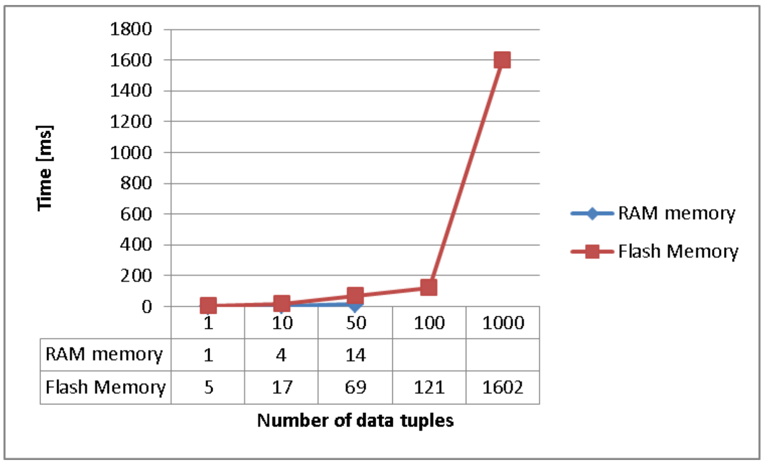
In this sub-section, we compare execution of a Micro-benchmark using RAM versus flash memories. Since TelosB has a quite small RAM memory capacity and much larger datasets can be stored and operated from the flash, it is important to compare the performance of operations over RAM-ready streams and flash-ready streams.

The Micro-benchmark tested scan and aggregation of 1 tuple, 10 tuples, 50 tuples, 100 tuples or 1000 tuples. It was run for one hour for each case and the results are average execution times. Figure 9.6 shows the results. All times are measured in a scale of milliseconds.

Since TelosB only has 10 kB of RAM memory and only 900 Bytes are available to create streams and store data, 50 tuples was the maximum number of tuples stored and processed in main memory. In this case, the execution time for select an average varies from 1ms for one tuple to 14ms for 50 tuples.

Figure 9.6 also shows results concerning benchmark execution when data is stored into flash memory. Here, the selection of an average varies from 5ms, for 1 tuple, to 1600 ms for 1000 tuples.

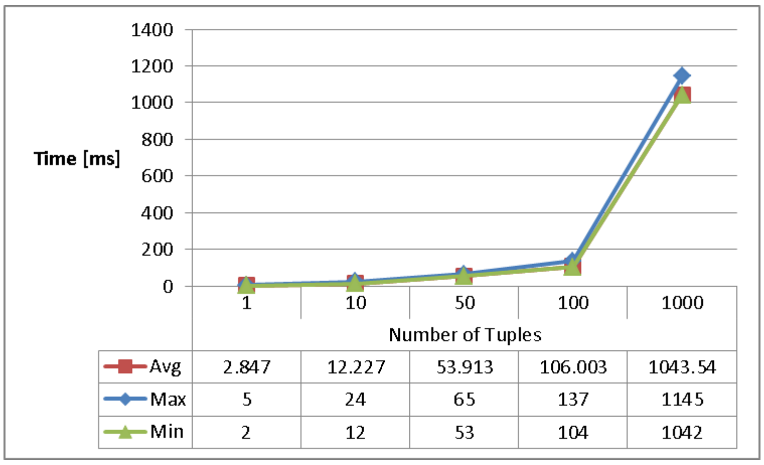
From Figure 9.6 we can conclude that operation times over flash memory are about four times slower when compared with RAM memory. However, computations over flash memory may include more data, which may not be possible in RAM memory.



	1	10	50	100	1000
RAM memory	1	4	14		
Flash Memory	5	17	69	121	1602

Figure 9.6 – Operation execution times over RAM versus flash memory

To further analyse the execution times of elementary operations in flash memory, we show in Figure 9.7 the maximum, average and minimum times necessary to read data tuples from flash memory in a TelosB mote.



	1	10	50	100	1000
Avg	2.847	12.227	53.913	106.003	1043.54
Max	5	24	65	137	1145
Min	2	12	53	104	1042

Figure 9.7 – Time to read data from flash memory

Other operations which can access to the flash memory are the "*create a stream*" and "*write a tuple into a stream*" operations. Depending on the configuration, MidSN-NC can create a stream in main memory or in the flash. To analyse the overhead of "*create a stream*" and "*write a tuple*", Table 9.4 shows the execution times for each of those operations.

These results show that operations over main memory are much faster. Operations such as write and read in flash memory, in average, take twice the time for main memory.

Table 9.4 – Time required creating a stream and writing a tuple

Operation	Maximum [ms]	Average [ms]	Minimum [ms]
Create Stream			
Main Memory	3	2.366	2
Flash Memory	119	15.522	11
Write a Tuple			
Main Memory	2	1.11	1
Flash Memory	72	3.793	2
Read a Tuple			
Main Memory	2	1.05	1
Flash Memory	35	1.972	1

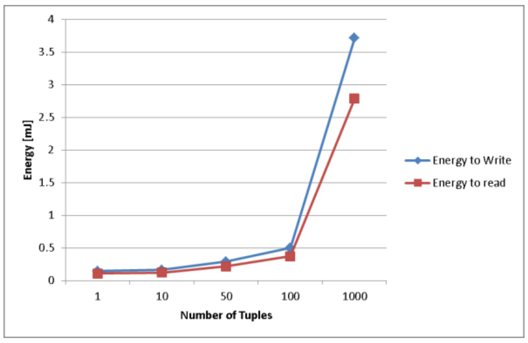
9.2.3. Data processing versus lifetime

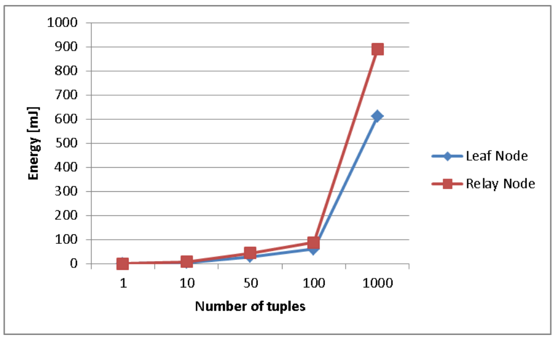
Communication is costly in sensor networks. The radio transceiver has comparatively high power consumption. To reduce the energy consumption of the radio, a radio duty cycling mechanism must be used. With duty cycling, the radio is turned off as much as possible, while being on often enough for the node to be able to participate in network communication. By contrast, flash memory chips can stay in low-power sleep mode as long as no I/O occurs, and do therefore not need to be duty cycled.

To quantify the trade-off in energy consumption between storage and communication, we measure the energy consumption of transmitting and receiving a packet, as well as reading and writing to flash on a TelosB. The results shown in this sub-section were obtained using the Contiki's Powertrace tool [157].

The Powertrace tool allows measuring the average power spent by the system in different power states, including flash reading and writing.

Figure 9.8a) shows the energy consumed to read and write a data tuple into a stream in flash memory, while Figure 9.8b) shows the energy consumption for a leaf node and an intermediate node when the same number of tuples are processed and sent to a sink node (2 hops).





a) Energy needed to read and write tuples from/to flash memory **b) Energy needed to transmit data tuples**

Figure 9.8 – Consumed energy for data tuples manipulation

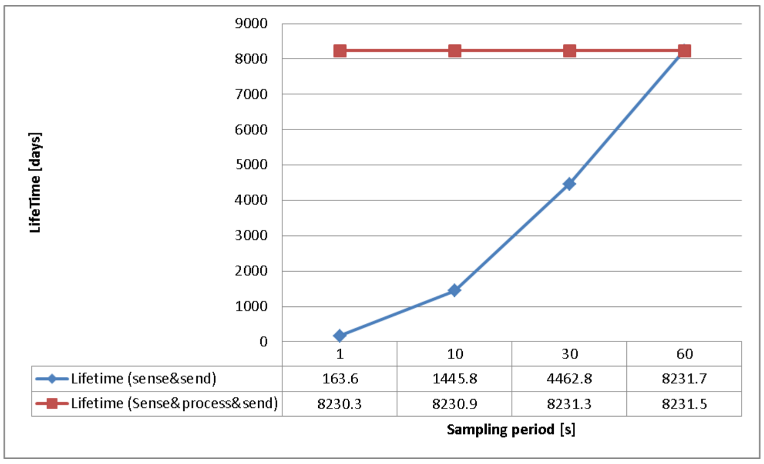
Assuming a scenario where data is collected every second and statistical information is only needed per minute, data samples can be collected during one minute, stored in memory, processed in the node after the collection period, and only the computation results are sent to the user. Alternatively, we can send each data sample to the control station immediately after collecting. From Figure 9.8b), if this last alternative is used, the node will dispend about 30.6 mJ to send the information (60 samples). Considering the first alternative (Figure 9.8a)), the node will dispend about 0.3 mJ to store 60 samples into flash, 0.2 mJ to read them back and 0.8 mJ to process and send the result to the user. In total, the node will only drain 1.3 mJ per minute.

Assuming that a node has two batteries, each with 1.5 V and 800 mAh, the total available energy can be calculated as:

$$\begin{aligned}\text{Battery Energy [Joules]} &= V \cdot Ah \cdot 3600 \\ &= 3 * 0.8 * 3600 \\ &= 8640 \text{ Joules}\end{aligned}$$

Consequently, the node lifetime can be estimated. Figure 9.9 shows the node lifetime estimation for the two alternatives described above.

As we can see, if one sample is collected and transmitted every second, the node will operate only during 163.6 days. However, if the flash memory and computation capabilities of the node are used, the node can operate during about 8230 days.



	1	10	30	60
Lifetime (sense&send)	163.6	1445.8	4462.8	8231.7
Lifetime (Sense&process&send)	8230.3	8230.9	8231.3	8231.5

Figure 9.9 – Node lifetime

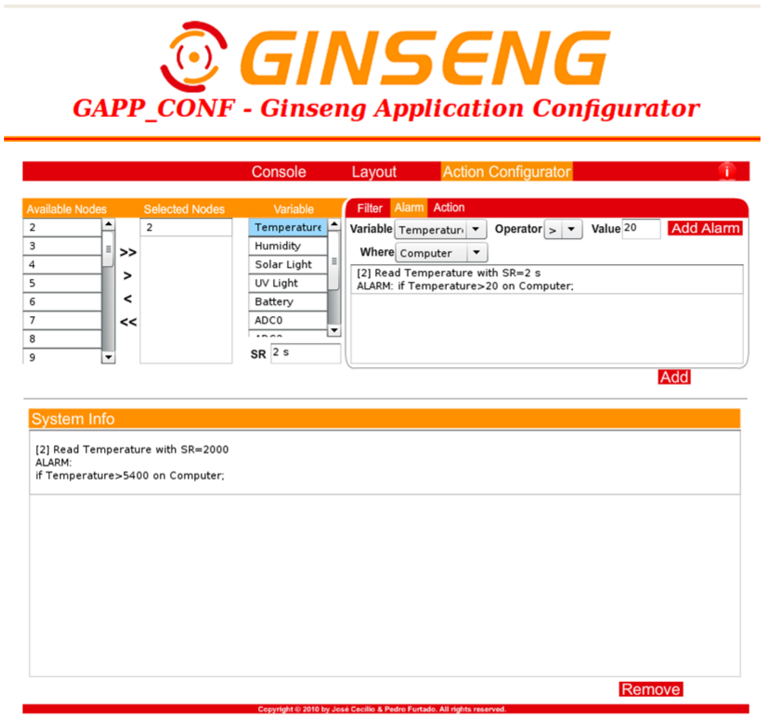
9.3. Networked Execution and Performance Evaluation

In this section we discuss the use of the MidSN capabilities. The discussion, demonstration and results in this section are intended to show that MidSN is able to function as a middleware over heterogeneous distributed systems that include WSN sub-networks, computer nodes and a control station. This abstraction allows to configure/reconfigure and process over a heterogeneous distributed network without any programming, only configuring.

We first introduce the MidSN-RConfig user interface that we used for configuring the system, and then we have the following sub-sections: Section 9.3.2 describes the setup; in Section 9.3.3 we show runtime results concerning command latency. Sections 9.3.4 and 9.3.5 show results concerning configuration and performance evaluation of monitoring and closed-loop operations.

9.3.1. MidSN-RConfig user interface

As part of the evaluation, we created a client application that allows us to configure and display information of all nodes in the network. Here, we describe the implementation of GApp_Conf, an interactive application built on top of MidSN-RConfig. This application can be used to configure or reconfigure operations and to view status information. Figure 9.10 shows an extract of the configuration interface of GApp_Conf.



Variable: Temperature Operator: > Value: 20
Where: Computer
Selected Nodes: 2
SR: 2 s

```
[2] Read Temperature with SR=2 s
ALARM: if Temperature>20 on Computer;
```

```
[2] Read Temperature with SR=2000
ALARM:
if Temperature>5400 on Computer;
```

Figure 9.10 – GApp_Conf – configuration interface

The application is capable of controlling several features of the network, such as creating closed-loops or alarms, performing actions, enabling/disabling sensors, setting thresholds and including/excluding nodes in the network. It allows users to, for example, define a set of rules to trigger certain actions based on a specified event.

9.3.2. Experimental setup

To demonstrate MidSN-NC running over an heterogeneous distributed control system, we built a testbed with 10 TelosB nodes, one Arduino node, one Raspberry PI node and one sink computer that acts as a gateway to the TelosB-based sub-network. The setup also includes a control station that receives the sensor samples, alarm messages and allows configuring operations, such as monitoring and/or closed-loop over whole network.

The sink computer and the Raspberry node are connected with an Ethernet cable through a GigaBit router network adapter. This router also offers a Wi-Fi interface which is used to connect Arduino to the network. The MidSN-RConfig is running in the control station, which allows us to configure the whole network. Figure 9.11 shows our setup.

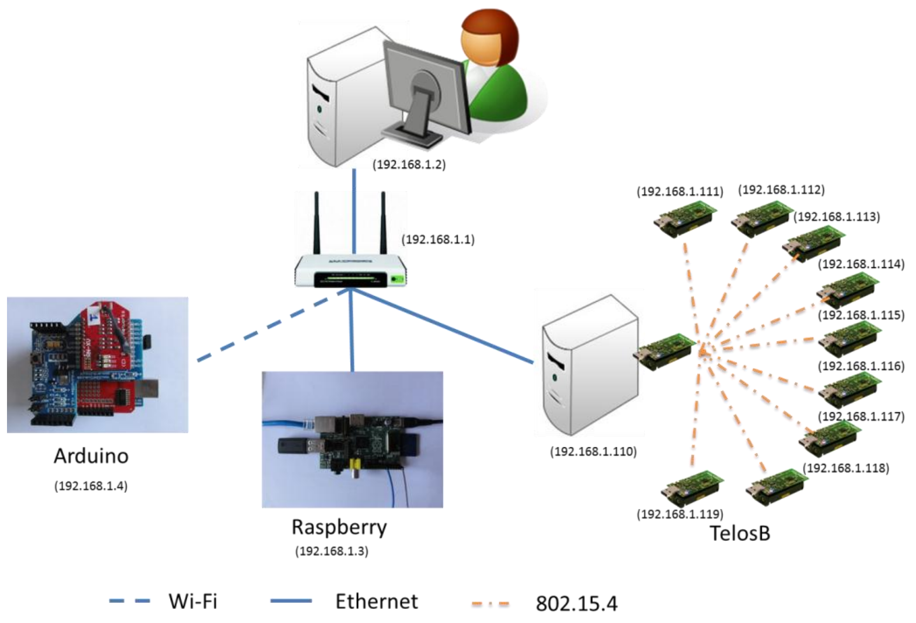


Figure 9.11 – Experimental setup

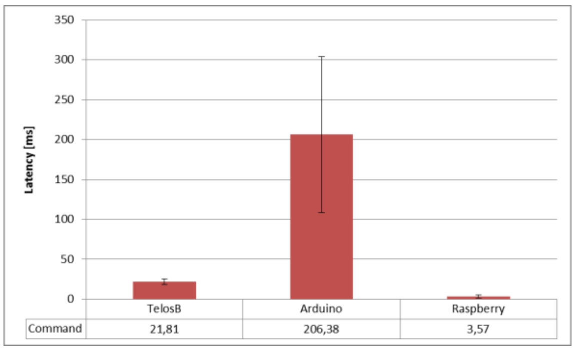
For this experiment the TelosB WSN nodes run Contiki-MAC [158] over Contiki. One of these nodes is used to interconnect the sink computer to the WSN network. Nodes are organized in a star, communicating in a single hop fashion with the sink node. All TelosB nodes were configured with the Plug&Play mechanism described in Chapter 5 and their initial configuration and addresses are stored in the Catalog.

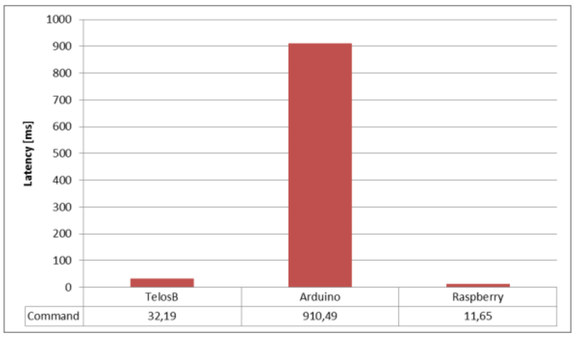
9.3.3. Command configuration and latency

After deploying our setup, we start the MidSN-NC evaluation by sending a set of commands to nodes and measuring delivery latency for each platform. The experiment consists of sending a command every 10 seconds to each node during one hour. The command used in this experiment was `MIDSN.Node.ping(AllNodes);`.

Figure 9.12 shows the average and maximums of latency for the three platforms (TelosB, Arduino and Raspberry).

From Figure 9.12 we conclude that Arduino was very slow when compared with the TelosB and Raspberry PI platforms. As we mentioned before, the Arduino has a WiFly module attached to it which communicates with our cabled network. This WiFly module uses a specific library and the SPI interface of the Arduino. Moreover, the WiFly library implements a SPI-to-UART bridge, which makes transmission slower.





a) Average latency with stdev **b) Maximum latency**

Figure 9.12 – Command latency for the three platforms

9.3.4. Monitoring operation

We configured nodes to generate and send one data tuple per second to the control station. Each tuple includes temperature and light measures. We use the internal board sensors in the TelosB platform to read temperature and light. A LM 75 temperature sensor and one LDR were used to read those values in the Arduino platform, while random values were generated in the Raspberry node to simulate the sensed values.

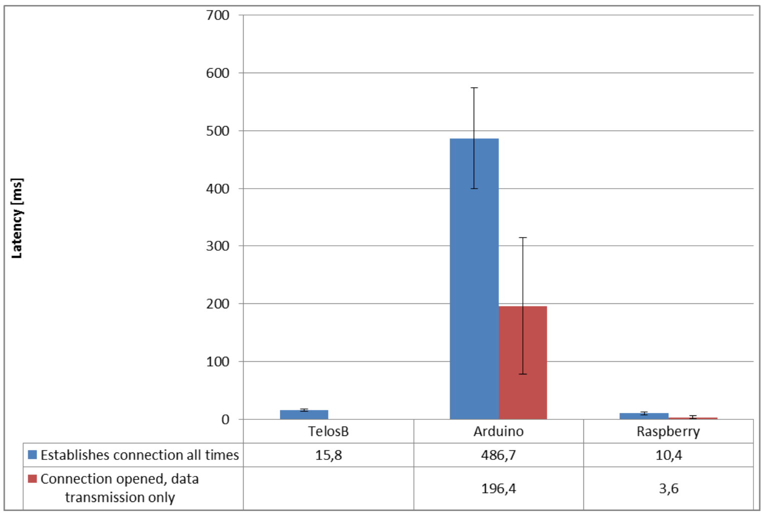
The distributed control system is configured to create a monitoring operation using the following method calls:

```
// create stream operation for reading from sensor
(a) MIDSN.Operation.create(AllNodes ,
                        "lightStream",
                        1s,
                        1s,
                        1,
                        {(TEMPERATURE, VALUE),(LIGHT, VALUE)},
                        {ControlStation }
                  );

// save stream at control station
(b) MIDSN.Operation.create(ControlStation ,
                        "lightStreamCS",
                        -1,
                        -1,
                        10000,
                        {( "lightStream" , VALUES)},
                        -1
                  );
```

Figure 9.13 – Configuration of a monitoring operation

The above code programmed all nodes to generate one data item at every 1 second with the temperature and light values and send it to the `ControlStation`. It also creates a stream at the control station to receive data tuples from the stream "`lightStream`" and store them inside the stream "`lightStreamCS`".

Figure 9.14 shows the average and standard deviation for data latency.

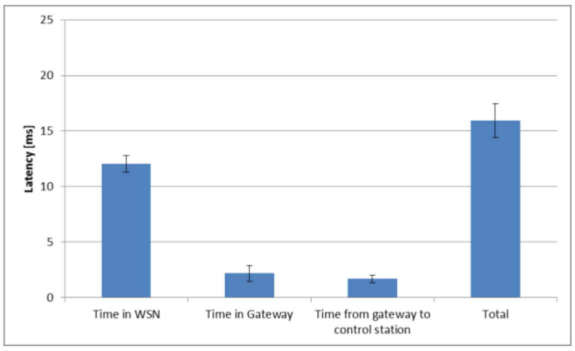


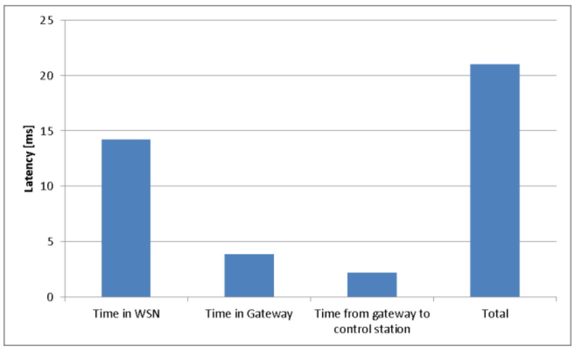
	TelosB	Arduino	Raspberry
🟦 Establishes connection all times	15,8	486,7	10,4
🟥 Connection opened, data transmission only		196,4	3,6

Figure 9.14 – Data latency for the three platforms

The Arduino and Raspberry platforms are connected to our network through IP connections. These two platforms support UDP and TCP connections. In this experiment we opted to use the TCP connection. There are two options to use the TCP connection: open the connection, send data and close it after transmission (establishes a new connection all the times); or open the connection only once and keep it opened (connection opened, data transmission only). The first alternative is used for slow data rates, where socket timeout may occur and the connection is closed. The other alternative is more suitable for frequent data transmission, such as high data rates. The results concerned to these two alternatives are shown in Figure 9.14.

The results concerning the TelosB platform can be decomposed into several parts: the latency of WSN (latency from sensing node to sink node), gateway latency (time needed to transfer data messages from the sink node to the gateway computer), and the time from the gateway to the control station. Figure 9.15 shows the average and maximum values of latency per each part and the total for the TelosB platform.





a) Average latency **b) Maximum latency**

Figure 9.15 – Data latency for TelosB per part

In Figure 9.14 we can see that a sample sent by one TelosB takes, in average, 15.8 ms to be delivered at the control station. In Figure 9.15 we can analyse how the latency is distributed by the network parts in the path to the control station. In average, a sample takes 12 ms to be delivered at the sink; 2 ms to be written be the sink and received by the gateway and 1.3 ms to be transmitted and received by the control station.

Figure 9.15b) shows the maximum latencies that occur in each part during our experiment.

9.3.5. Closed-loop over heterogeneous devices

The stream-based configuration offered by MidSN also provides functionalities to configure closed-loop control over a heterogeneous network. The closed-loop decision can be taken on data source nodes (sensors), on any node in the network, as well as in the control station. The fact that whole parts of the system contain the same configuration and processing component and are directly referenced by an address variable allows uniform configuration in spite of being very different platforms.

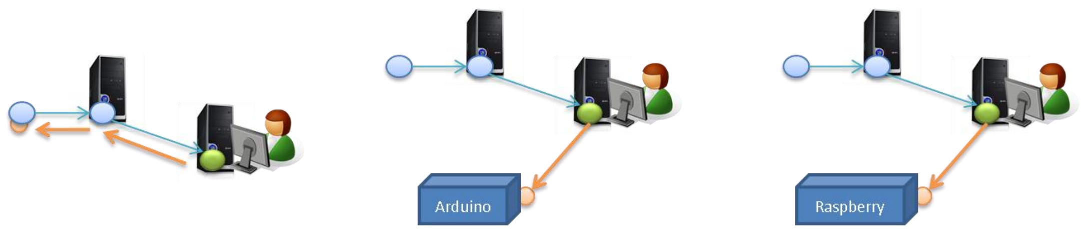
We can configure the network to create a closed-loop control operation using the following method calls:

```
// create stream operation for reading from sensor
(a) MIDSN.Operation.create(Zone1Sensors,
                        "lightMessagesfromZone1Sensors",
                        5s,
                        5s,
                        1,
                        {(LIGHT, VALUE)},
                                {ControlStation }
                          );

      // save stream at control station
      (b) MIDSN.Operation.create(ControlStation ,
                          "lightMessagesfromZone1SensorsCS",
                          -1,
                          -1,
                          10000,
                          {( lightMessagesfromZone1Sensors, VALUES)},
                          -1
                    );

      // Evaluate data stream
      MIDSN.Action.create(ControlStation,
                          "turnOnLight",
                          "lightMessagesfromZone1SensorsCS",
                          CONDITION ((LIGHT, VALUE), <, (VALUE, 105)),
                          ACTUATION (LIGHT_ACTUATOR, ON, 1.1)
                          );
```

Figure 9.16 – Configuration of a closed-loop operation with decision logic in the control station

In the above example we program the nodes from zone 1 to generate one data item at every 5 seconds with the light value and configure the control station to receive data from the stream `lightMessagesfromZone1Sensors` that has the light values sensed by the nodes from zone 1. The control station is also configured to analyse the data tuples and verify if light values are less than 105. When the condition is matched, an actuation command with the instruction of `ON` is generated and sent to the actuator `LIGHT_ACTUATOR` connected to node 1.1.

To exercise the heterogeneous remote configuration capabilities, we define three different alternatives where the actuator is connected. The first alternative consists on connecting the actuator to one TelosB node. The second alternative consists on attaching the actuator to the Arduino node, while in the third alternative the actuator is attached to the Raspberry node. Figure 9.17 shows a sketch of these three alternatives.



a) TelosB ➔ TelosB **b) TelosB ➔ Arduino** **c) TelosB ➔ Raspberry**

Figure 9.17 – Closed-loop alternatives

Figure 9.18 shows the results concerning average and maximum closed-loop latencies for the three different alternatives shown in Figure 9.17. The latency is measured as the time taken from sensing node to actuation command at the target node, passing through the supervision control logic residing at the control station.

To collect the results of Figure 9.18 we ran the experiment for one hour. The results show that, when we sense and actuate over the TelosB platform, the closed-loop takes about 50 ms and is always less than 100 ms. If the sensing is done in the TelosB and the actuation is done by the Arduino, the closed-loop takes on average 250 ms, about 5 times mores. However, if the actuation is done over the Raspberry PI node, the closed-loop time is about half of the TelosB.

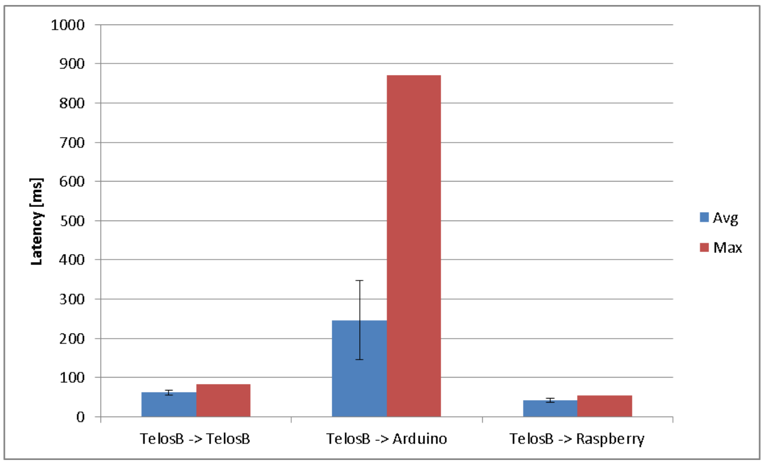
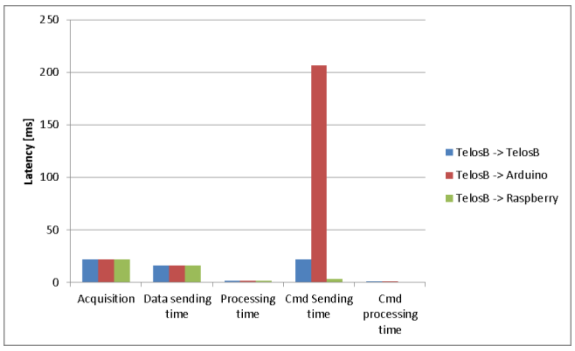
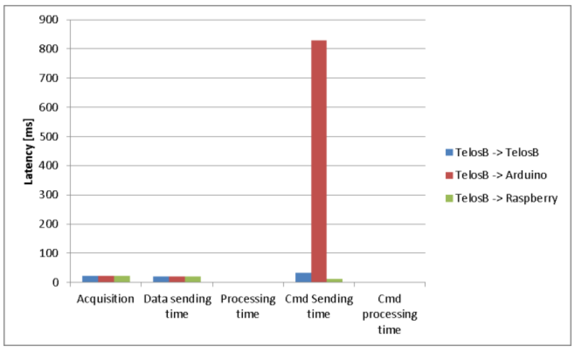


Figure 9.18 – Closed-loop latency over heterogeneous network

These results can be decomposed into several parts: acquisition, transmission of data values from sensing node to the control station, processing time, command sending time to send the actuation command to the actuator, and the command processing time at the target node. This decomposition identifies which parts consume more time. Figure 9.19 shows the average and maximum values of latency per part.





c) Average latency per part **d) Maximum latency per part**

Figure 9.19 – Closed-loop latency over heterogeneous network per system parts

From Figure 9.19 we can conclude that the big latency in the second alternative (sensing on TelosB and actuation over Arduino) is due to the time needed to deliver a command to the Arduino node.

Chapter 10

Evaluation of Planning and Monitoring Approaches

In this chapter, we present the results of experimental evaluation of the planning and monitoring approaches that were proposed in Chapters 7 and 8. We will consider a heterogeneous network where WSN sub-networks coexist with cabled-networks. In the following sections we will describe our setup (Section 10.1). Based on this setup we will present results comparing observed latencies with the values estimated by the system planning tool. Section 10.2 reports results concerning monitoring operation over the setup. Since operation timings may be relative to event occurrence, Section 10.3 reports results when considering event occurrence instant. In Section 10.4 we show how the algorithm splits the network to meet strict monitoring latency requirements. Section 10.5 shows results concerning the planning algorithm applied to closed-loop operation, Section 10.6 shows results concerning the position of downstream slots used to send actuation commands to nodes, and Section 10.7 experiments with the number of downstream slots. In Section 10.8 we show results concerning multiple actuators with different timing requirements.

In our approach, to reduce the actuation latency, we add more downstream slots. The number of downstream slots has significant impact in network lifetime. Section 10.9 reports results concerning lifetime versus the number of downstream slots used to meet timing requirements.

Lastly, Section 10.9 reports results concerning bounds and debugging tool. We create a simulation environment where we introduce some random delays in the messages, to demonstrate how the debugging tool works and its usability.

For completeness, the charts in this chapter are complemented by appendix J, where we detail the values in table format.

10.1. Setup

The testbed comprises a control station, a cabled network, one gateway and a WSN sub-network. Figure 10.1 shows a sketch of the setup in our lab.

The WSN sub-network includes 12 TelosB nodes organized hierarchically in a 1-1-2 tree and one sink node (composed by one TelosB node and one computer that acts as gateway of the sub-network). The setup also includes a control station that receives the sensor samples, monitors operations performance using bounds, and collects data for testing the debugging approach.

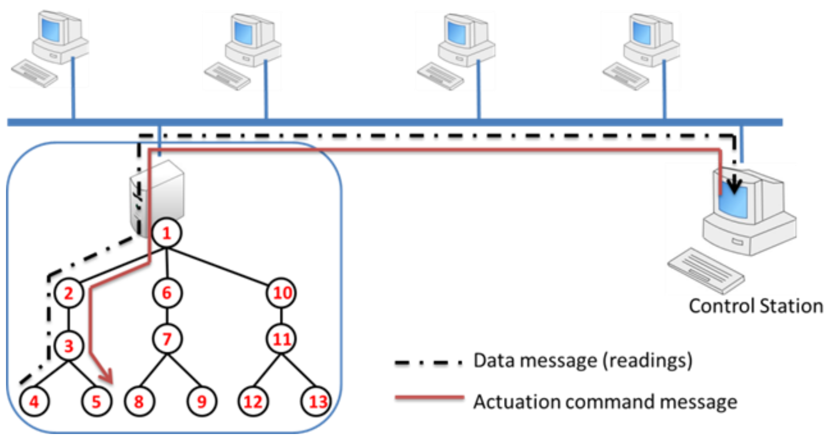


Figure 10.1 – Setup

The control station is a computer with an Intel Pentium D, running at 3.4 GHz. It has 2 GB of RAM and an Ethernet connection. The gateway connecting the WSN sub-network to the cabled network is another computer with similar characteristics.

All computer nodes (gateway and control station) are connected through Ethernet cables and GigaBit network adapters. The WSN sub-network is connected to the gateway using the serial interface provided by TelosB nodes. That interface is configured to operate at 460800 baud/second.

All computers run Linux OS and have specific components developed using java to do specific tasks. For instance, the gateway computer has a gateway software component to read data from the serial interface and send it to the control station, and to receive messages destined for the WSN and deliver them through the serial interface. The control station has the MidSN-RConfig component to implement remote configuration and to perform functionality such as closed-loop control.

The WSN sensor nodes run Contiki OS and generate one message per time unit with a specified sensing rate. Each message includes data measures such as temperature and light. GinMAC [2] is used at the mac layer by the WSN nodes.

10.2. Planning of Monitoring Operations: Evaluation

The approach proposed in Chapter 7 dimensions the network to meet monitoring latencies required by applications and users.

Consider that a user provides as user inputs, as defined in Section 7.7.1 of Chapter 7:

- The network configuration represented in Figure 10.1 (in the format exemplified in appendix G);
- Indication of option 2 of data forwarding rule (each parent collects data from all children and only forwards after receiving from all child nodes)
- Monitoring operation (periodic sensor sampling without event consideration) to run 120 days, at least, over the distributed control system, with a minimum sampling rate of 1 second and a maximum desirable monitoring latency of 200 ms.

Based on network configuration, data forwarding rule, latencies and lifetime requirements, the algorithm creates a schedule for the network. We followed the steps of the algorithm, creating the schedule of Figure 10.2. This schedule has an epoch with

1 second of length and an inactivity period of 330 ms, determined by eq. 26. It includes sufficient slots for each node to transmit its data upwards, where one retransmission slot is added to each transmission slot to enhance reliability. The schedule also includes transmission slots for sending configuration or actuation commands, slots for time synchronization and slots for node processing.

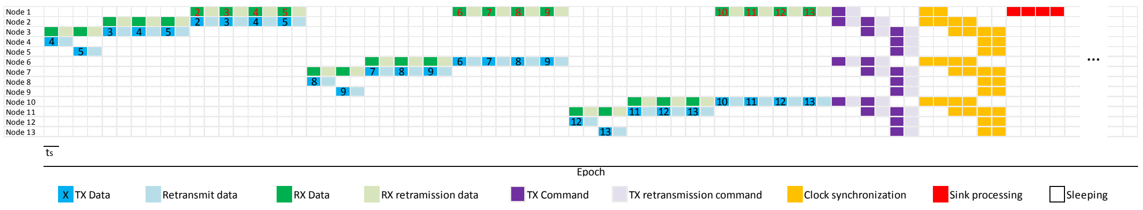


Figure 10.2 – TDMA schedule

In the next sub-sections we will verify latency requirements for this resulting schedule using the latency formulas described in Chapter 7, and we will report results from the experimental testbed to compare with the values forecasted.

10.2.1. Verifying latencies using the formulas

Based on user requirements and applying eq. 8 of Chapter 7, we obtain:

$$200 = \max\left(t_{WSN_{AqE}}\right) + \max\left(t_{Serial}\right) + \max\left(t_{Middleware}\right) + \max\left(t_{\text{Processing}}\right)$$

The amount of latency due to non-real-time parts (t_{Serial}, $t_{Middleware}$, $t_{\text{Processing}}$) is characterized by network testing. To characterize those parts from our setup, the setup ran during 1 hour and we collected time statistics. Table 10.1 shows the characterization of latencies of non-real-time parts for this setup. All times are given in milliseconds.

Table 10.1 – Non-real-time parts characterization [ms]

	t_{Serial}	$t_{Middleware}$	$t_{\text{Processing}}$
Average	2.64	1.12	0.51
Standard Deviation	0.40	0.29	0.12
Maximum	7.79	3.14	0.85
Minimum	1.85	0.67	0.32

Replacing the non-real-time latencies in eq. 8, we obtained:

$$\max\left(t_{WSN_{AqE}}\right) = 200 - (7.79 + 3.14 + 0.85) = 188.22[ms]$$

Based on eq. 9, assuming that acquisition and sending instants are synchronized ($t_{WaitTX\,Slot} = 0$), that event occurrence instant is not considered ($t_{Event} = 0$), according to the prediction model for maximum latencies

$$\max\left(t_{WSN_{AqE}}\right) = \max\left(t_{Aq}\right) + \max\left(t_{WSN_{UP}}\right)$$

where $\max\left(t_{Aq}\right)$ is determined by node testing. We assume it to be 20 ms.

The amount of latency due to WSN sub-network is predictable by the analysis of the schedule. In the schedule (Figure 10.2), each node placed in level 1 (near the sink node) takes between 10 and 20 ms to deliver a message to the sink node (slot time plus retransmission slot time). Nodes of the second level take between 90 and 100 ms, while nodes of the third level take between 150 and 160 ms. This can be deduced from looking at the schedule of Figure 10.2.The 10 ms tolerance is due to the retransmission slot in the last link of the path to the sink node. Table 10.2 summarizes the WSN latencies for the schedule represented in Figure 10.2.

Since the setup is organized as a tree hierarchy with three levels, $t_{WSN_{AqE}}$ will assume three different values. Depending on the node position, $t_{WSN_{AqE}}$ can be 40 ms (20 ms for acquisition ($\max\left(t_{Aq}\right)$) plus 20 ms for $\max\left(t_{WSN_{UP}}\right)$, according to Table 10.2), 120 ms (20 ms for $\max\left(t_{Aq}\right)$ plus 100 ms for $\max\left(t_{WSN_{UP}}\right)$ - Table 10.2) or 180 ms (20 ms for $\max\left(t_{Aq}\right)$ plus 160 ms for $\max\left(t_{WSN_{UP}}\right)$ - Table 10.2) (eq. 9). Since all of those values are less than $\max\left(t_{WSN_{AqE}}\right) = 188.22[ms]$ we conclude that latency requirements are met with the network layout provided by the user.

Table 10.2 – Maximum WSN sub-network latency per topology level (resulting from looking the sub-network schedule)

Level	Maximum Latency [ms]
1	20
2	100
3	160

Applying eq. 1 of Chapter 7 with the values of Table 10.1, the monitoring latency for nodes at level 1 can be predicted as:

$$Monitoring_{Latency} = 40 + 7.79 + 3.14 + 0.85$$

The same equation is applied to the other levels, resulting in the times shown in Table 10.3. All nodes at the same level have the same forecast.

Table 10.3 – End-to-end operation latency estimation per topology level

Level	Latency [ms]
1	51.78
2	131.78
3	191.78

10.2.2. Testbed run results

To assess the latency model, we tested the network layout resulting from the algorithm (Figure 10.2) and compared the latency results with the expected latencies calculated in the previous sub-section, which were given by applying the latency formulas of Chapter 7. Figure 10.3 shows statistical information of latency per node, gathered from an experiment that ran for 3 days.

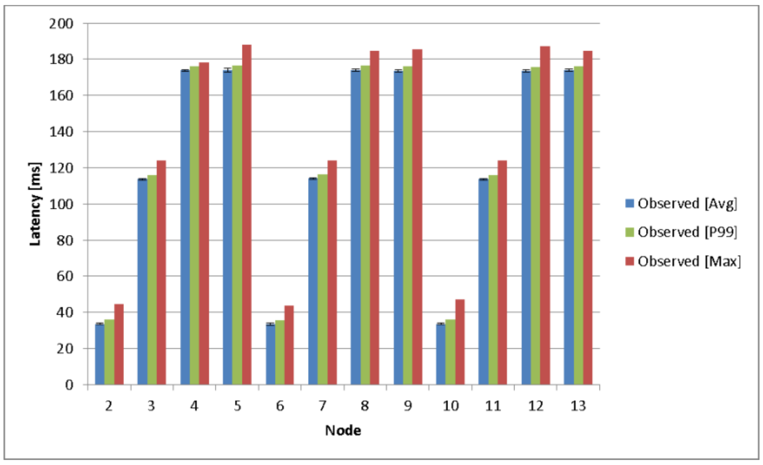


Figure 10.3 – End-to-end monitoring latency per node

From Figure 10.3 we can see that nodes at the same level have similar latencies. Figure 10.4 shows statistical information of latency (per level), as well as the values corresponding to the prediction given by the planning formulas (Table 10.3).

From Figure 10.4 we can conclude that the planning approach predicts well for this setup. The observed maximum value gathered during the test is always below the prediction. It is near, but below the planned maximum.

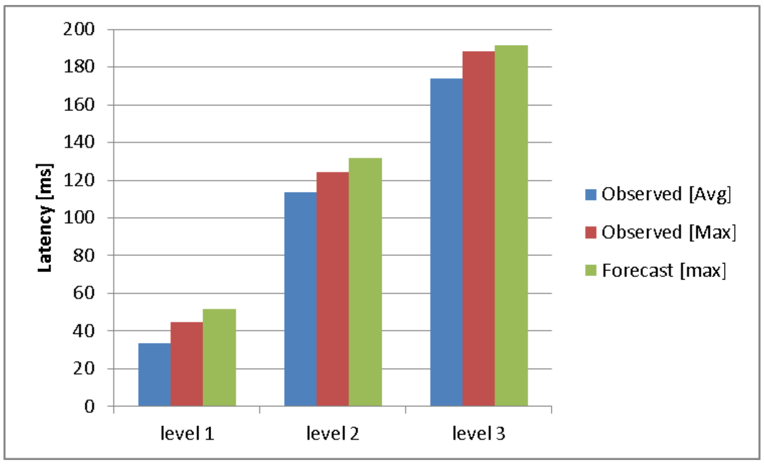
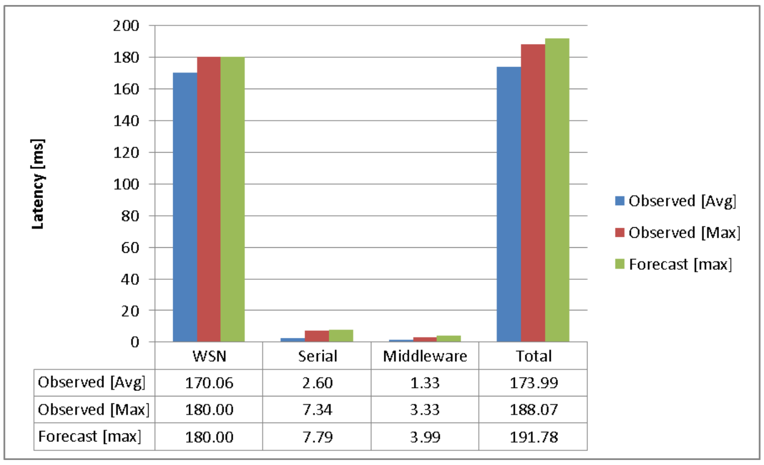


Figure 10.4 – Monitor latency per level with forecast

The latencies shown in Figure 10.4 can be decomposed into the following latencies: WSN latency, serial interface latency and the middleware latency. Figure 10.5 shows those latencies for a node in the third level.



	WSN	Serial	Middleware	Total
Observed [Avg]	170.06	2.60	1.33	173.99
Observed [Max]	180.00	7.34	3.33	188.07
Forecast [max]	180.00	7.79	3.99	191.78

Figure 10.5 – Monitor latency per network part

The WSN latency is, in average, 170.06 ms, but it can grow up to 180 ms. This maximum value agrees with the prediction (forecasted values). Concerning other parts such as serial and middleware latencies, the obtained maximum values show a little difference to the predicted values, but all of them are below the prediction.

10.3. Considering Event Occurrence Instant

Events may occur in any instant. In order to provide timing guarantees, considering the instant the event occurs, it is necessary to account for an upper bound on the extra time from the event instant to the acquisition instant.

Based on the network configuration (Figure 10.1) and the schedule (Figure 10.2) presented in the previous sections, and assuming 1200 ms as maximum desirable monitoring latency, eq. 8 determines the maximum monitoring latency for the WSN sub-network ($\max\left(t_{WSN_{AqE}}\right)$).

$$\max\left(t_{WSN_{AqE}}\right) = 1200 - (7.79 - 3.14 - 0.85)$$

t_{Serial}, $t_{Middleware}$, $t_{\mathrm{Processin}g}$ were already shown in Table 10.1.

Since the epoch size defined by the schedule shown in Figure 10.2 has 1 second (1000 ms), $t_{WSN_{AqE}}$ for the first level of the tree is given by eq. 2 and results in:

$$t_{WSN_{AqE}} = 1000 + 20 + 0 + 20 = 1040[ms]$$

The values used for this calculation concerning $t_{WSN_{UP}}$ are from Table 10.2. Applying the same equation to the other levels, we obtain 1120 ms and 1180 ms for levels 2 and 3, respectively.

In order to test latencies considering event occurrence instance, we inject an external event at a random instant in each epoch. This experiment ran for 2 hours. Figure 10.6 shows the observed average and maximum values of latency, as well as the values forecasted by the formula given in eq. 1, for comparison.

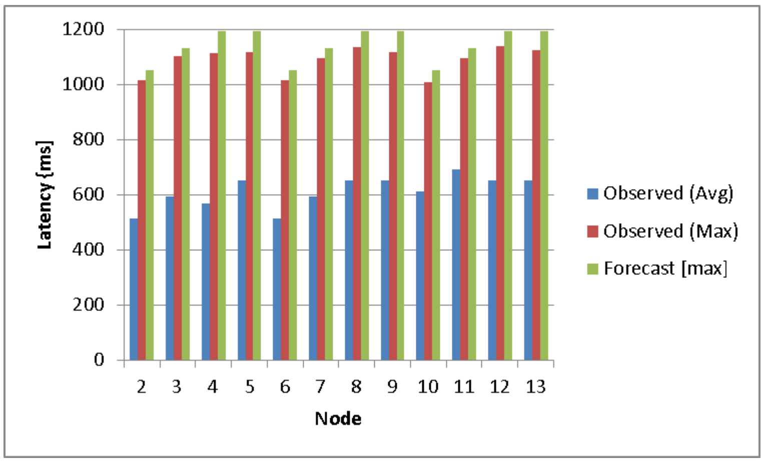


Figure 10.6 – Event detection latency per node (observed and forecast)

From Figure 10.6 we can see that an event takes, in average, half of the epoch to be identified in the control station. The observed maximum values are similar to the forecasted maximum latencies, as expected.

10.4. Planning with Network Splitting

The approach proposed dimensions the WSN network to meet latency requirements. For instance, considering the same setup of the previous experiment, if a user specifies 500 ms as maximum latency, event occurrence should be reported within that timing constraint. Since the forecast of maximum latency was much larger (1200 ms), the network should be re-sized to define an epoch which meets the latency requirement.

Applying the planning algorithm, the initial network is divided into three sub-networks where each includes one branch only, resulting on the schedule shown in Figure 10.7.

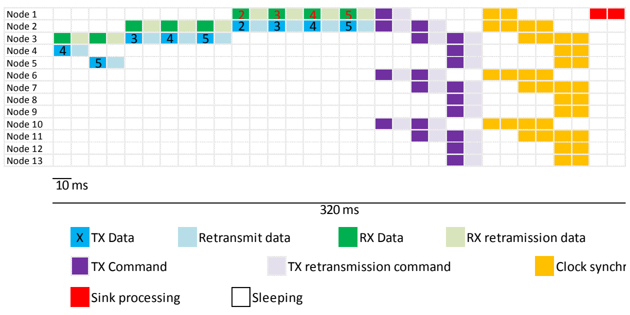


Figure 10.7 – TDMA schedule to meet the event latency

To evaluate the resulting sub-network and schedule, we built a testbed and ran it during 2 hours. Figure 10.8 shows the statistical and forecasted values of latency.

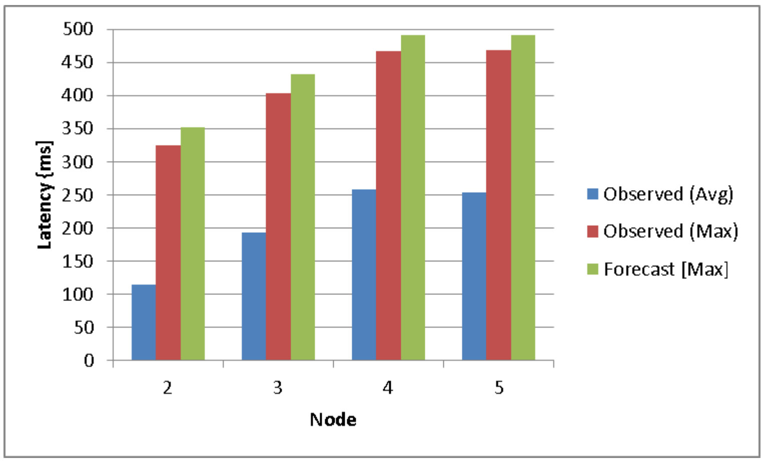


Figure 10.8 – Event detection latency (observed and forecast)

From Figure 10.8 we can see that the new layout guarantees the timing constraints and the forecast values agree with our testbed (maximum latency bounded by 500 ms).

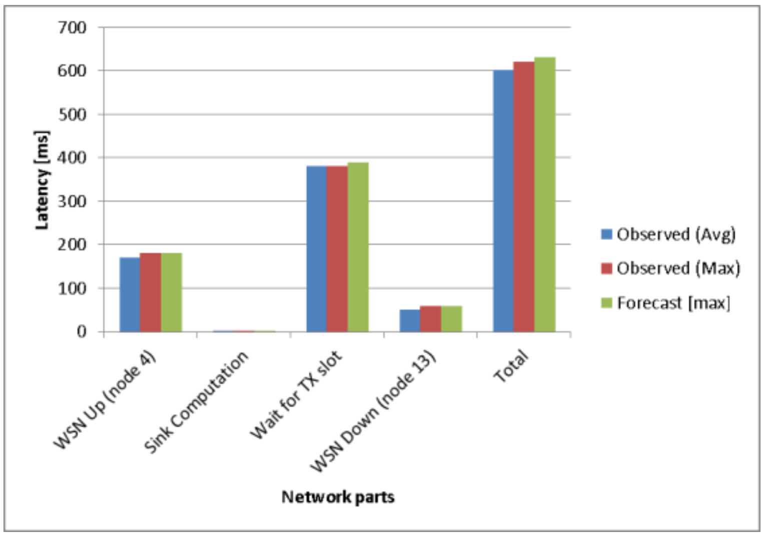
10.5. Planning of Closed-loop Operation: Evaluation

Closed-loop operations can be configured to be processed inside the WSN sub-network (through the sink node) or outside the WSN sub-network (through a control station). Asynchronous closed-loops through the sink node involve sensor nodes sending sensed values to the sink node, the sink node evaluating a threshold and sending an actuation command to an actuator node.

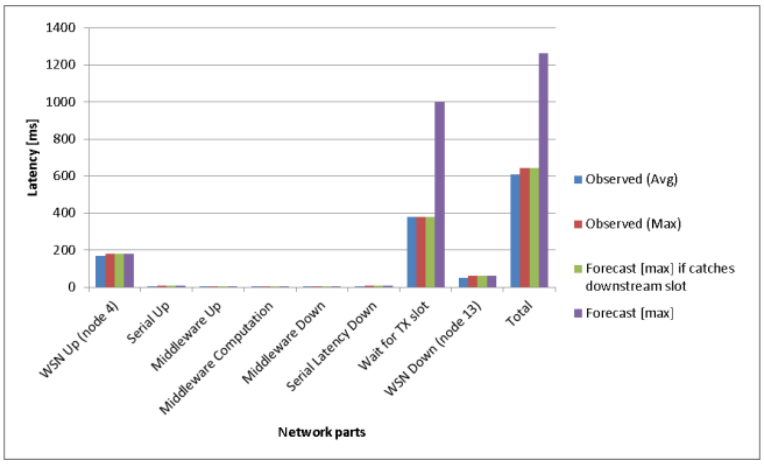
Asynchronous closed-loop control outside the WSN network (through a control station) involves a sensor node sending its sensed value to the control station though the sink node, the control station computes a decision based in closed-loop control algorithms, and sending an actuation command back to the sink, which will forward it to an actuator node.

To exercise closed-loop operations and latency predictions, the setup described in Section 10.1 was configured for node 4 to be a sensor node and node 13 an actuator. The setup ran for 3 hours, and we collect the statistic information shown in Figure 10.9.

Figure 10.9 shows the observed closed-loop latency and the closed-loop latency forecasted by the formulas of Chapter 7. We considered both closed-loops within a WSN sub-network, and closed-loops outside the WSN sub-network (through a control station). The results are for the schedule represented in Figure 10.2.



a) Supervision control logic inside embedded devices (sink node)



b) Supervision control logic in the control station

Figure 10.9 – Closed-loop latencies

Figure 10.9 shows the maximum and average observed values and the forecast values obtained though eq. 5 and 7. The value "forecast [max] *if catches downstream slot*" represents the forecast considering that commands reach the sink before the downstream slot arrives. The "forecast [max]" represents the other possibility, which occurs when the command reaches the sink node after the schedule reaches the downstream slots, and that command must wait one more epoch to be transmitted to the target node.

From Figure 10.9 we conclude that the observed maximum latencies were always within the bounds defined by the forecasted maximum latencies. In Figure 10.9b) we can also conclude that the observed maximum latencies were always below the "forecast [max] *if catches downstream slot*" bound. This means that the time taken for the sensed data message to go from the sink to the control station, to compute the threshold condition and to send the actuation command back to the sink node was sufficiently small to catch the downstream slot in the same epoch.

10.6. Changing the Position of Downstream Slots

From previous latency results (Figure 10.9) we conclude that a large fraction of the total latency is due to the $t_{Wait\,forTX\,slot}$. To reduce $t_{Wait\,forTX\,slot}$, we can configure the planning algorithm to position the downstream slots after a specific slot in the epoch as described in Section 7.7.1 of Chapter 7. For instance, if the position of downstream slots is changed to just after the upstream slots for the branch where the sensing node is included (branch 1) (Figure 10.10), $t_{Wait\,forTX\,slot}$ is reduced.

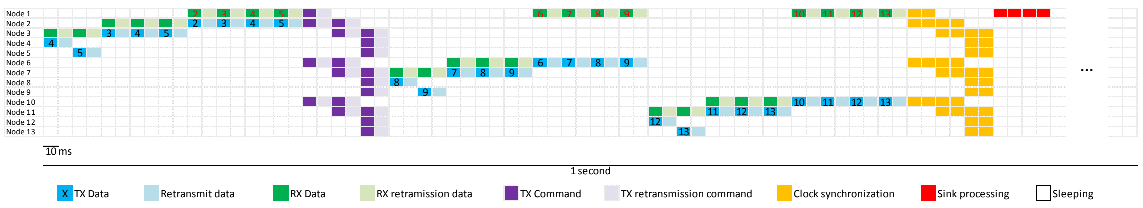
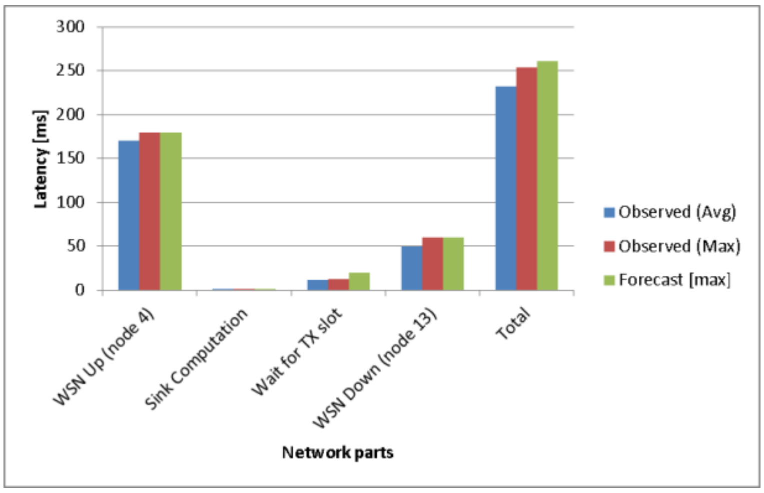


Figure 10.10 – TDMA schedule

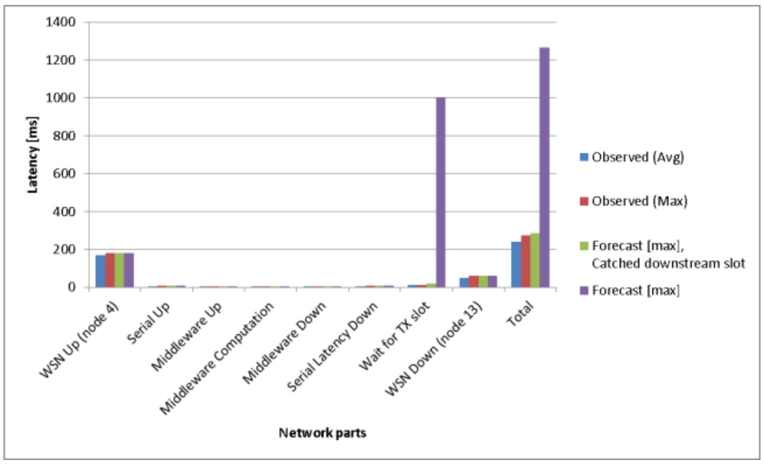
The schedule represented in Figure 10.10 has the downstream slots placed just after the upstream slots for branch 1.

Using the same setup for the previous example but with this new schedule, we ran the experiment again during 3 hours and collected statistics, Figure 10.11 shows the observed results, compared with forecasted values determined by eq. 5 and 7. Figure

10.12 is a detail concerning $t_{Wait\,forTX\,slot}$ for experiments reported in Figure 10.9 and Figure 10.11.



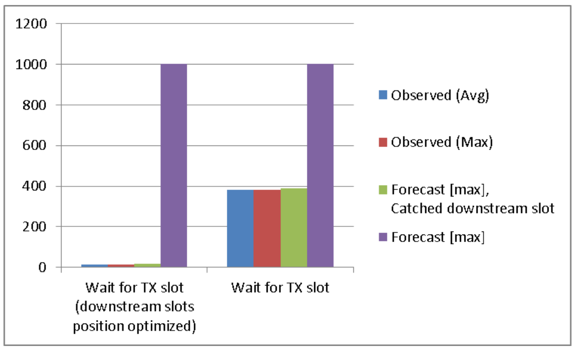
a) Supervision control logic inside embedded devices (sink node)

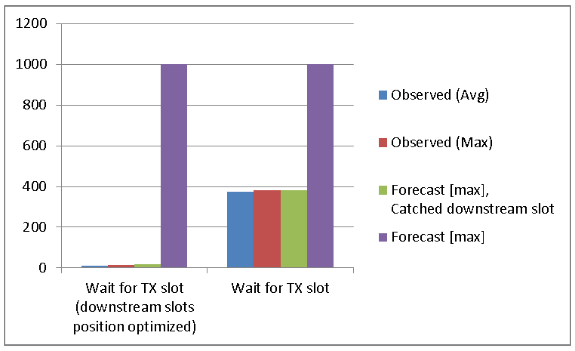


b) Supervision control logic inside the control station

Figure 10.11 – Closed-loop latencies

From Figure 10.9 and Figure 10.11 (the comparison detail is also in Figure 10.12) we conclude that the value of $t_{Wait\,forTX\,slot}$ was reduced significantly when the downstream slot position was moved. This modification also reduced the "forecast [max] if catches downstream slot", although the "forecast [max]" remains the same, since it considers always a full epoch.





a) Supervision control logic inside embedded devices (sink node)

b) Supervision control logic inside the control station

Figure 10.12 – Detail of $t_{Wait\ for TX\ slot}$ latency comparison

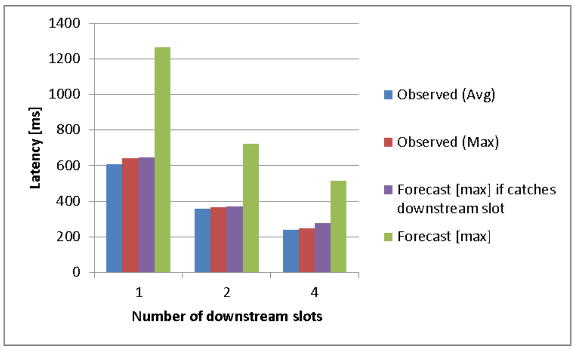
The downstream slots were moved 380 ms backwards in the schedule of Figure 10.10, when compared with the schedule of Figure 10.2. From Figure 9.12 we can see that the difference between $t_{Wait\ for TX\ slot}$ in the two cases is about the same value, as expected.

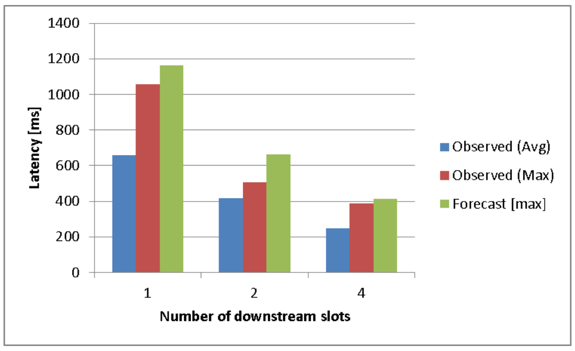
10.7. Adding Downstream Slots Equally Spaced in the Epoch

In Section 7.7.1 of Chapter 7, we have proposed that command latencies can be decreased by adding equally spaced downstream slots. To guarantee latency reduction, the number of downstream slots can be increased.

Figure 10.13 shows the influence of the number of downstream slots in the end-to-end closed-loop latency. In this experiment the network shown in Figure 10.1 and the schedule of Figure 10.2 were used. The closed-loop decision was configured to run in the control station and the network was configured to send one sample per second. Six 40 minutes experiments were ran corresponding to different number of downstream slots (1, 2 or 4) and the two closed-loop alternatives (asynchronous and synchronous).

Figure 10.13a) shows results concerning asynchronous closed-loop control, while Figure 10.13b) corresponds to synchronous closed-loop control.





a) Asynchronous **b) Synchronous**

Figure 10.13 – Closed-loop latency versus number of downstream slots

From Figure 10.13a) we can see that asynchronous closed-loops are able to catch the downstream slot in the same epoch where the sensing happened. This is seen by comparing the observed [max] with "forecast [max] *if catches downstream slot*". From Figure 10.13b) we can conclude that if one downstream slot is used per epoch, the command waits a maximum of one epoch (1000 ms) to be transmitted to the target node. In average, a command is delivered in 1/2 of the epoch time plus travel time. In both Figure 10.13b) and Figure 10.13a), if more slots are used, $t_{Wait\,for TX\,slot}$ is successively reduced to half for each added slot.

10.8. Multiple Close-loops

In this section we consider the case of multiple closed-loops with decision logic in the control station, with different timing requirements. In this case, the number of equally spaced downstream slots should be dimensioned to meet the most restrictive timing requirement. For instance, consider the setup of Figure 10.1 and that we have four actuators placed in nodes 5, 7, 10 and 13, which must be controlled according to the sensor data collected by sensors 4, 2, 6 and 12, respectively, with the time requirements shown in Table 10.4.

Table 10.4 – Closed-loop latency requirements

Case	Sensor	Actuator	Closed-loop latency [ms]
1	4	5	500
2	2	7	250
3	6	10	150
4	12	13	500

From Table 10.4, we can see that the strictest closed-loop time is 150 ms. Applying eq. 21 of Chapter 7, we obtain the number of downstream slots (Table 10.5) needed to meet each latency of Table 10.4. Table 10.6 shows the values of each parameter used in eq. 21 to determine how many slots are needed for each case.

Table 10.5 – Number of downstream slots required to meet latency requirements

Case	Sensor	Actuator	Number of downstream slots
1	4	5	5
2	2	7	7
3	6	10	10
4	12	13	5

Table 10.6 – Latency parameters [ms]

Parameter	Case 1: [4, 5, 500]	Case 2: [2, 7, 250]	Case 3: [6, 10, 175]	Case 4: [12, 13, 500]
$\max\left(t_{WSN_{AqE}}\right)$	180	40	40	180
$\max\left(t_{Serial}\right)$	7.79	7.79	7.79	7.79
$\max\left(t_{Middleware}\right)$	3.14	3.14	3.14	3.14
$\max\left(t_{\mathrm{Processing}}\right)$	0.86	0.86	0.86	0.86
$\max\left(t_{CMD\mathrm{Processing}}\right)$	0.91	0.91	0.91	0.91
$\max\left(t_{WSN_{Down}}\right)$	30	20	10	30

From Table 10.5 we conclude that 10 downstream slots are needed to meet the closed-loop time constraint for all configurations of Table 10.4. Figure 10.14 shows the obtained schedule that meets all closed-loop times. This schedule results from the planning algorithm, when we configure it to place the downstream slots equally spaced in the epoch.

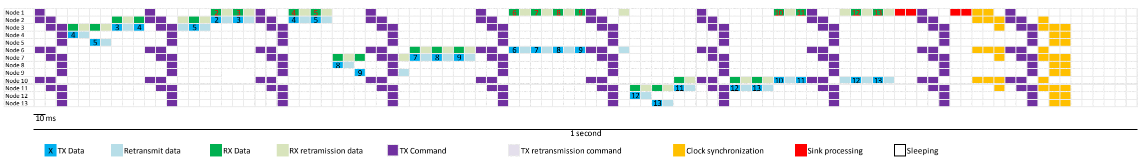
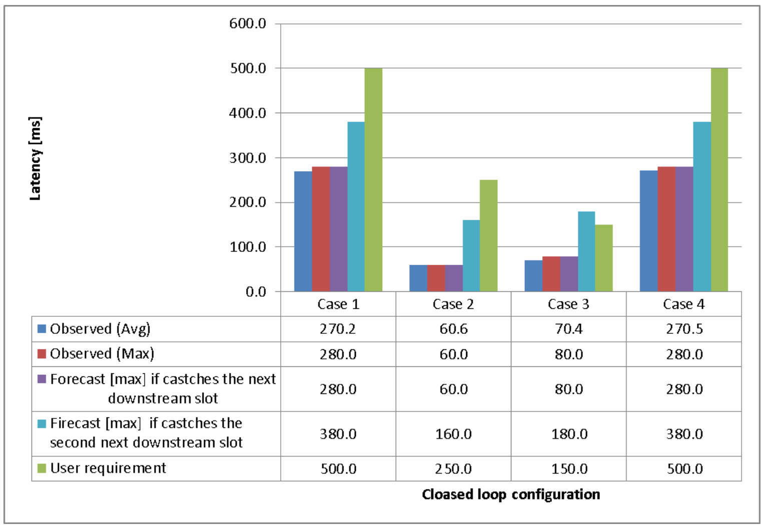


Figure 10.14 – TDMA schedule that meets the strictest closed-loop latency

Figure 10.15 shows the closed-loop latencies that were observed, the forecasted values and the maximum admissive latency indicated by the user for the closed-loops (user requirement).



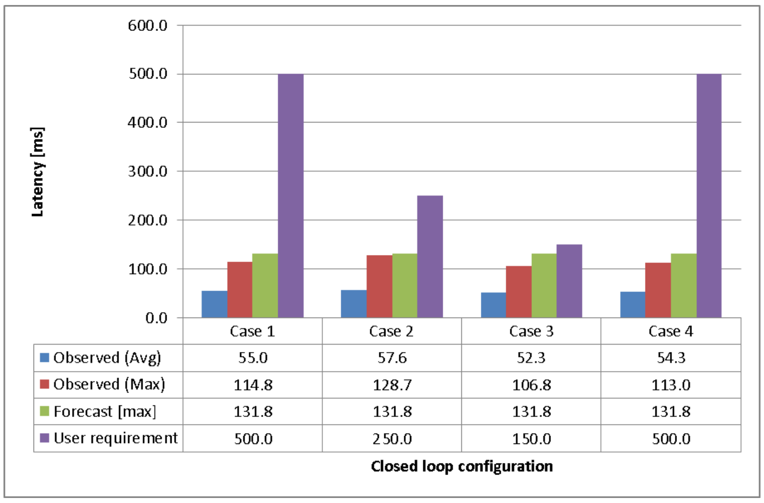
	Case 1	Case 2	Case 3	Case 4
🟦 Observed (Avg)	270.2	60.6	70.4	270.5
🟥 Observed (Max)	280.0	60.0	80.0	280.0
🟪 Forecast [max] if castches the next downstream slot	280.0	60.0	80.0	280.0
🟦 Firecast [max] if castches the second next downstream slot	380.0	160.0	180.0	380.0
🟩 User requirement	500.0	250.0	150.0	500.0

Figure 10.15 – Asynchronous closed-loop latency for all configurations

Figure 10.15 shows that all forecast [max] bounds were met in the experiment. The total latencies vary from case 1 to case 4 because the sensors and actuators are in different positions of the network layout. In cases such as case 1 and case 4 more than one downstream slot were passed by before the actuation command was determined and ready to go down.

From Figure 10.15 we can also conclude that required timings were always met if actuation commands catch the first downstream slot ahead after receiving the sensed data by the sink node. Actuator 10 has the strictest latency requirement (150 ms) but, as we can see in the figure, that restriction is met. However, the figure also shows that if actuation commands did not catch the first downstream slot ahead, and the second downstream slot would be caught instead, the latency requirement for case 3 would not be met. The command would take about 180 ms to be delivered to the actuator at node 10, while the requirement was 150 ms.

Figure 10.16 shows the results concerning latency for synchronous closed-loop control.



	Case 1	Case 2	Case 3	Case 4
🟦 Observed (Avg)	55.0	57.6	52.3	54.3
🟥 Observed (Max)	114.8	128.7	106.8	113.0
🟩 Forecast [max]	131.8	131.8	131.8	131.8
🟪 User requirement	500.0	250.0	150.0	500.0

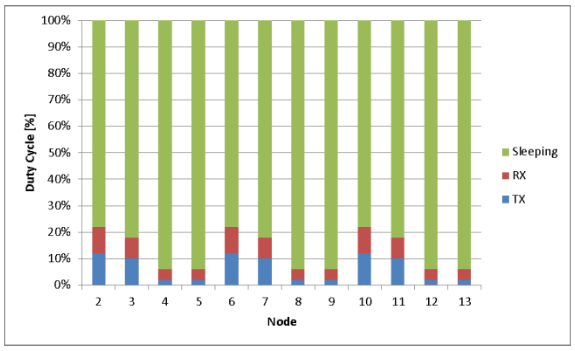
Figure 10.16 – Synchronous closed-loop latency for all configurations

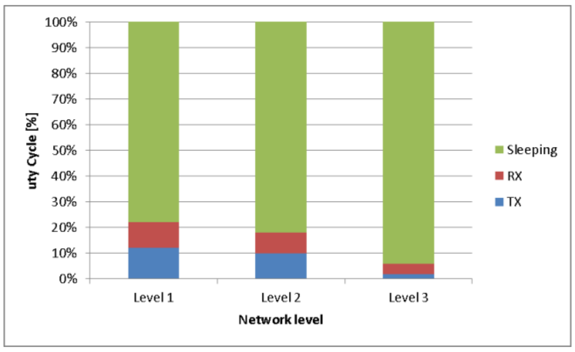
From Figure 10.16 we can conclude that all commands are delivered within 132 ms, as expected from applying the forecast formulas of the algorithm. This means that the strictest constraint (150 ms of case 3) is met, while other configurations show a large tolerance. In this experiment all closed-loops took more or less the same times because synchronous closed-loop latencies concerns control station to actuator latencies only (eq. 6 of Chapter 7).

10.9. Energy and Lifetime Issues

When we add more downstream slots, the node will wake up more times in the epoch, which will decrease its lifetime. Figure 10.17 shows the radio duty-cycle of nodes when one downstream slot is used (schedule of Figure 10.2). Figure 10.17a) shows the duty-cycle per node and Figure 10.17b) shows per network level.

From Figure 10.17 we can see that nodes of the first level are awake about 22% of the epoch while nodes of the third level are awake only 4% of the epoch. If we increase the number of downstream slots, all nodes will awake more time per epoch, which will decrease the lifetime.





a) Duty-cycle per node **b) Duty-cycle per network level**

Figure 10.17 – Radio duty-cycle

Assuming that nodes have two batteries, each with 1.5V and 800mAh, the total available energy can be calculated as:

$$\begin{aligned}\text{Battery Energy [Joules]} &= V \cdot Ah \cdot 3600 \\ &= 3 * 0.8 * 3600 \\ &= 8640 \text{ Joules}\end{aligned}$$

Considering that a node consumes a constant current of 0.9 mA when the radio is on and 0.1 mA when the radio is off, the node lifetime can be estimated by applying eq. 32 described in the Chapter 7. For example, considering the schedule represented in Figure 10.2 and a node placed at the third level of the tree, its lifetime is given by:

$$\begin{aligned}T_{Total} &= \frac{8640}{\left(0.9\times10^{-3}\cdot0.06+(1-0.06)\cdot0.1\times10^{-3}\right)\cdot3} \\ &= 19459459 \text{ cycles} \\ &= 225{,}2252 \text{ days}\end{aligned}$$

Table 10.7 shows the lifetime prediction for each level of the tree.

Table 10.7 – Lifetime prediction

Level	Node Lifetime [days]
1	120,7729
2	136,6120
3	225,2252

Figure 10.18 shows how the lifetime decreases with the number of downstream slots. If one downstream slot was used, the nodes of the first level can operate during about 120 days. Nodes of the second level will be available during 136 days, while nodes of the third level can operate during 225 days. These values will decrease according to the number of downstream slots. For instance, if we add 10 downstream slots, we will reduce the configuration or actuation command latency, but the lifetime of the node will decrease drastically. In this situation, a node of the first level will only be available for 59 days.

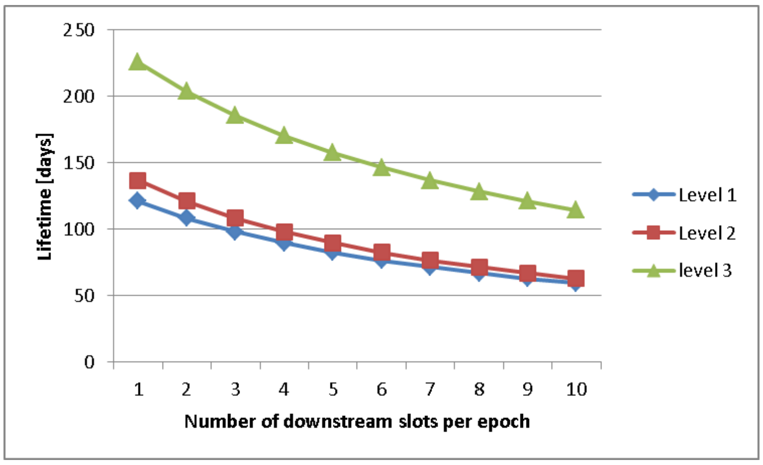


Figure 10.18 – Radio duty-cycle estimation

10.10. Testing Bounds and the Performance Monitoring Tool

The proposed approach for performance and debugging allows defining bounds to classify each message. The bounds can be applied to latencies, delays or both.

To exercise the use of bounds and debugging (Chapter 8), we created a monitor operation and introduced a “liar” node which injects 10 ms of delay in the first of every two consecutive messages that travel through it. Using the setup shown in Figure 10.1, copied to Figure 10.19, we will replace node 3 by the “liar” node.

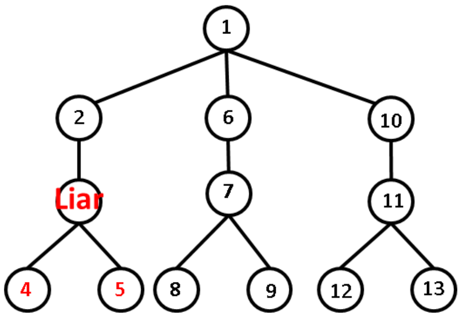


Figure 10.19 – Setup with “liar” node

Moreover, to simulate some losses in the network, we changed the node 4 configuration to consecutively send one message and discard the next message. This allows us to simulate 50% of message losses.

Figure 10.20 and Figure 10.21 show the results concerning message delays. Figure 10.20 reports values concerning delay without the “liar” node, while Figure 10.21 reports delays after replacement of node 3 by the “liar” node.

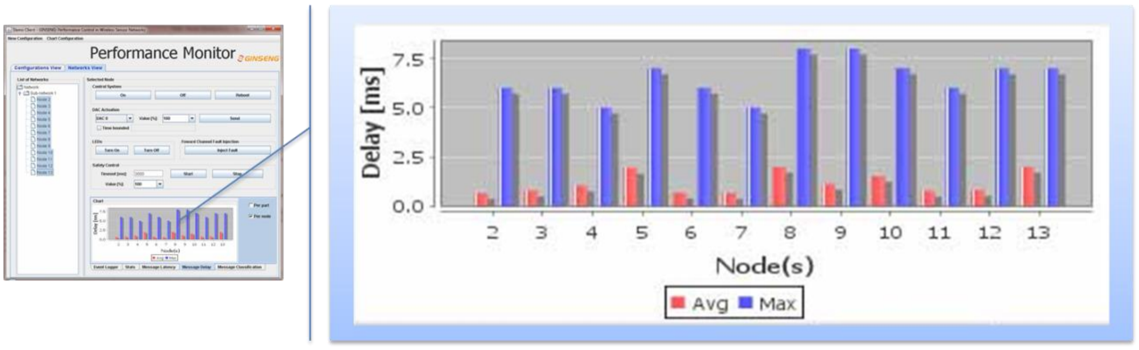


Figure 10.20 – Message delay without “liar” node

From Figure 10.20 we can conclude that consecutive messages arrive at the control station, in average, within 0.5 to 2 ms. This value can grow up to a maximum of 8 ms.

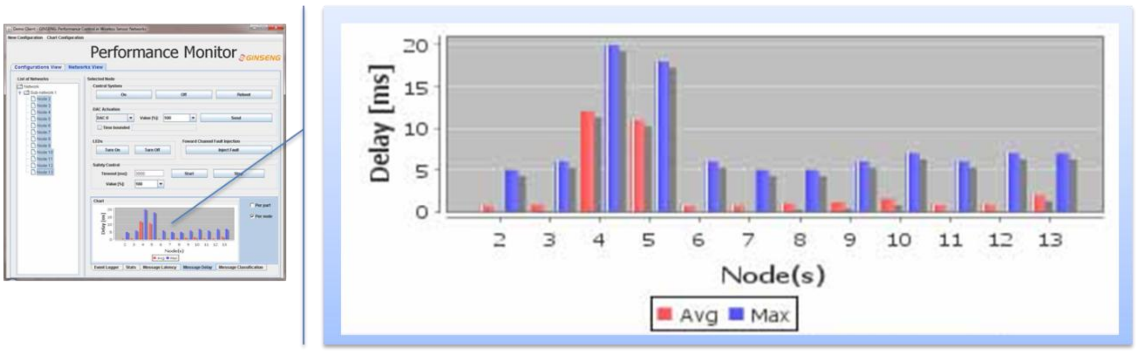


Figure 10.21 – Message delay with "liar" node

After introducing our "liar" node and running the monitoring operation for 24 hours, we obtained the chart of Figure 10.21. This figure shows that the delay of nodes 4 and 5 increased. These nodes send their messages to the control station passing through the "liar" node, which is node 3. In this case, we can see that the delay of two consecutive messages increased, in average, to 12 ms, and up to a maximum of 20 ms.

Using the PM described in Chapter 8, we can also define bounds for the message delay. Assuming that each message should arrive at the control station within a maximum delay of 10 ms, we can define a delay bound and analyse the results.

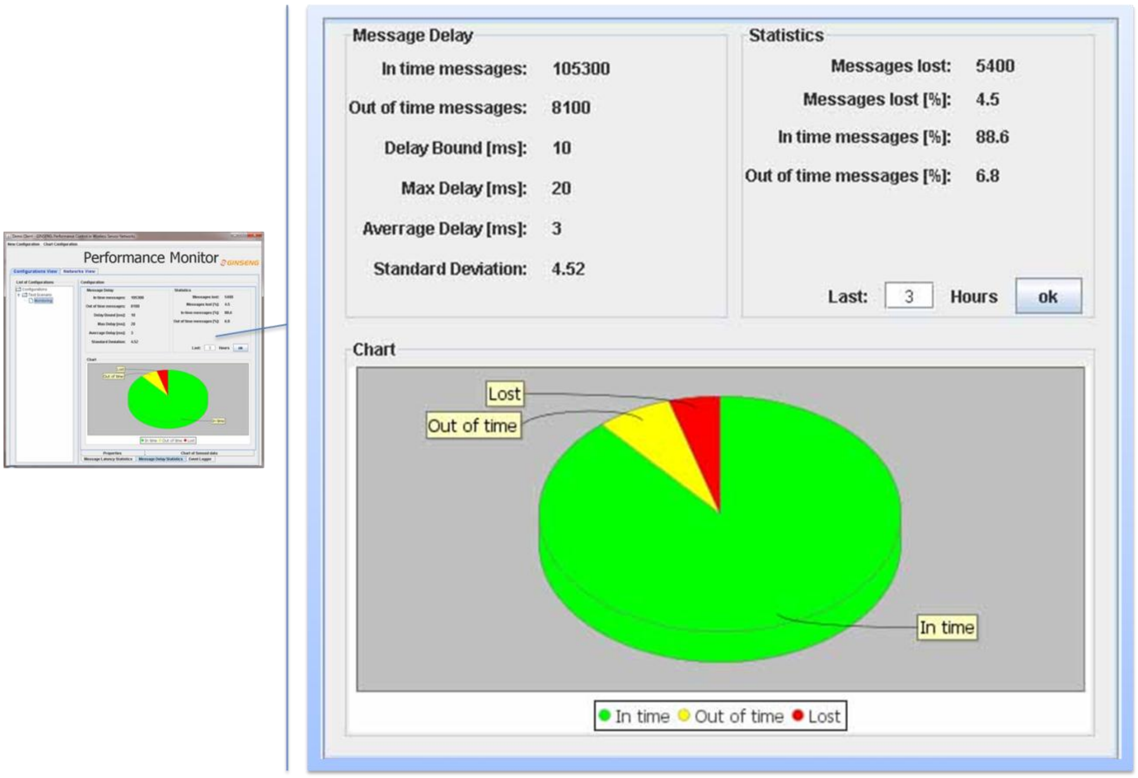
Figure 10.22 shows the percentage of messages classified into each category (in-time, out-of-time, lost), according to a delay bound of 10 ms and lost timeout of 1s (the timeout when a message is considered lost).

From Figure 10.22 we conclude that 88.6% of the messages are delivered within the bound, but 6.8% are delivered out of bounds and 4.5% are lost. These numbers are as expected:

- Messages lost – there are 12 nodes sending data messages, node 4 fails one in every two messages. That results in $\frac{1}{12*2}$ losses, which agrees with the result of 4.5% losses that was obtained.
- Messages out of bound – there are 12 nodes sending data messages, node 4 sends only half of its messages and half of them arrive delayed. Concerning node 5, half of its messages arrive delayed. That results in

$\frac{1}{12*3}+\frac{1}{12*2}$ messages out of bound, which agrees with the result of 6.8% out of bound that was obtained.

- Messages in time – 88.6% of messages are delivered in time, that results from the total number of expected received messages minus the number of losses and out of bound $\left(1-\left(\frac{1}{12*2}+\frac{1}{12*3}+\frac{1}{12*2}\right)\right)$.


Message Delay

In time messages: 105300

Out of time messages: 8100

Delay Bound [ms]: 10

Max Delay [ms]: 20

Averrage Delay [ms]: 3

Standard Deviation: 4.52

Statistics

Messages lost: 5400

Messages lost [%]: 4.5

In time messages [%]: 88.6

Out of time messages [%]: 6.8

Last: 3 Hours ok

Figure 10.22 – Message classification according to delay bound of 10 ms

As we described in Chapter 8, the user can also explore event properties (in this case, delays) and find where the problem occurred. For instance, the user interface includes per node evaluation such as the one in Figure 10.23, which shows which node(s) is failing.

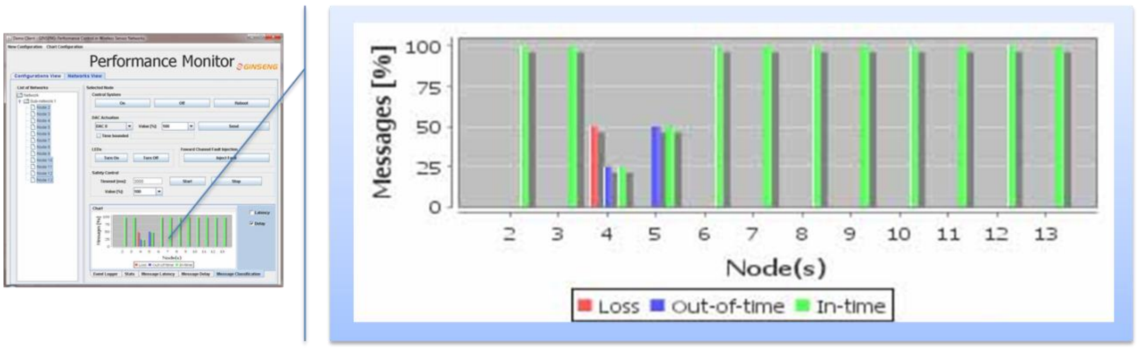


Figure 10.23 – Message classification according to delay bounds per node

From Figure 10.23 we can conclude that node 4 is responsible for the losses represented in Figure 10.22. It is losing 50% of the messages, as expected.

Figure 10.23 also shows that the delay bound is not met by nodes 4 and 5. In this case, further debugging allows the user to identify the path of each message and to check where it took longer than expected.

For instance, if we explore the path and delay parts of node 5, we can conclude that messages sent by that node are waiting, in average, 10 ms in the transmission queue of node 3 (our "liar" node), which is greater than the expected average delay value of 2 ms for node 3 sending messages to the control station seen in Figure 10.20.

Chapter 11

Conclusions and Future Work

This thesis proposed a middleware architecture (MidSN) to handle operation over heterogeneous distributed systems with embedded devices, and to provide timing guarantees in those contexts. It proposed mechanisms to achieve node referencing and homogenization of heterogeneous underlying systems (hardware and software). A data and processing model, and operations were also proposed, which provide flexibility in configuration and processing over the heterogeneous sensor network.

State-of-the-art in middleware and remote configuration techniques used in wireless sensor network platforms was reviewed, and we discussed their applicability in heterogeneous sensor networks with WSN sub-networks.

A middleware architecture and operations model for uniform configuration and operation were proposed. The model views the whole system as a distributed system and any computing device as a node (inside or outside of the WSN, regardless of hardware or operating system) with the same remote configuration capabilities and operation interfaces.

We have described the architecture and details of the approach. In the experimental evaluation, an implementation of the middleware was developed for four different hardware and software platforms. These implementations have shown the advantage of having defined the architecture and that its embodiments are able to run over resource constrained-devices or over more powerful devices such as Raspberry PI or computers.

We also built a testbed and defined a set of tests that show that the implementation of the architecture is able to configure both sensor nodes and control stations easily and using exactly the same calls. From our test runs we extracted logs and displayed results concerning memory, execution time, processing capabilities and correct configuration of all devices.

Another contribution of this thesis is a network and operations planning algorithm and tool designed to meet timing requirements over the middleware architecture. In order to make heterogeneous distributed systems more reliable in practical contexts with constraints such as timing requirements, there is a need for approaches to help a user correctly plan the network and its operations (e.g. monitor, control). We proposed and evaluated an approach to plan the network, monitor and debug the performance. The approach schedules operations, predicts latencies and subdivides the wireless sensor network until the predicted latencies meet operation end-to-end latency requirements.

The approach also provides mechanisms to classify messages concerning their bounds and provides feedback about how many messages arrived at the control station in-time or out-of-time. Our experimental results show that it correctly forecasts execution latencies in several situations.

Since integration is easy in the middleware architecture and no programming is required from users, the cost of inclusion of new platforms is reduced, which allows creating heterogeneous distributed systems more quickly and by any user.

Because of the amplitude and the issues raised by heterogeneity, we expect that this work can be used as a starting point for future research in extending this proposal and its findings.

The thesis is not a closed proposal and raises interesting issues that require further investigation. Since MidSN is based on drivers, it can incorporate different alternatives as underlying platforms. We already developed the node component of the middleware for Java Virtual Machines, in particular for computers (e.g. PCs) and Raspberry PI. It is interesting to investigate the development of micro Java Virtual Machines for other tiny devices, in which case MidSN would be deployable directly without further development for those devices. Other IP-based communication-related protocols that can be applied to tiny devices, such as 6LowPan [66], CoAP [74], HTTP, REST and Web-services, are also useful for future research regarding MidSN. They provide a network heterogeneity hiding layer over which MidSN is able to offer configuration and operations of the whole heterogeneous distributed system, without any programming.

The issue of how to deal with configuration and operation in distributed systems with thousands of nodes and how to plan and provide timing guarantees in such large scale systems should also be investigated.

The trade-off between application specificity and middleware generality is another research challenge. It is important to integrate application knowledge into the services provided by the middleware, because it makes the middleware ready for more applications.

Besides these challenges, there are challenges related to adapting the MidSN structure to tiny devices, since MidSN needs a certain amount of resources that may not be available in some devices. Conversely, MidSN can evolve into a full-featured stream processing engine when it runs in platforms with high computation capabilities.

The operation times planning approach proposed in this thesis can be used regardless of specific operating systems and network protocols of the heterogeneous system. Research issues in that context include how these findings can be applied to industrial standards and protocols such as Fieldbus, Hart and wirelessHart. Additionally, it is interesting to investigate and evaluate the approach while replacing the non-real-time components by real-time components, such as RT Kernel or java real time.

Another interesting issue is how operations planning and timing guarantees would be conceived for token-based distributed control system architectures and protocols such as Fieldbus.

Generally, the planning approach and latency models described in this thesis can be used as starting points for further future research in timing guarantees for heterogeneous distributed systems.

Appendix A

Communication Driver – Code Example

In this Appendix, we show the implementation of the communication driver for ContikiOS, using Contiki-C and for Linux, using java.

In Figure A.1a) we show the implementation for Contiki-OS. This implementation includes four methods:

packet_recv – allows receiving new data packets and notify the MidSN-NC that a new packet is available to be consumed. A *new_message_arrives* flag is analysed by the method to check if the last received message was consumed or not. If a new message arrives and the last received message was not consumed (*new_message_arrives* different of zero), the driver discards it.

open_connection – allows starting a new peer-to-peer connection between two nodes. As described in the driver specifications, this method receives as arguments an address and a port to the target node.

send_to – is the method that allows sending messages (data or commands) to other nodes. It receives an address and a buffer with data/command. A low-level packet is built to be sent as soon as possible by the communication protocol.

```c
static void packet_recv(struct abc_conn *c)
{
    if (new_message_arrives == 0) {
        msg_len =
packetbuf_copyto(&packet);
        new_message_arrives = 1;
    } else
        printf("MidSN-NC: Packet
Dropped\n");

    process_poll(&midsn_ioAdapter);
}

static const struct abc_callbacks abc_call
= {packet_recv};
static struct abc_conn abc;

void open_connection(char * address, u16_t
port ) {
    abc_open(&abc, 128, &abc_call);
}

void send_to(char * address, struct
midsn_packet * packet) {
         uint16_t * addr = (uint16_t * )
address;
    packetbuf_clear();
    packetbuf_copyfrom((u8_t *) packet,
(packet->size + MIDSN_PKT_HEADER_SIZE));

packetbuf_set_addr(PACKETBUF_ADDR_RECEIVER,
addr );
    if (abc_send(&abc))
        printf(" %u bytes sent\n",packet-
>size + MIDSN_PKT_HEADER_SIZE);
    else
        printf("Msg sending fail!\n");
}

struct midsn_packet *
midsn_get_received_packet() {
    new_message_arrives = 0;
    return &packet;
}
```

```java
private byte[] packet = new byte[256];
public int new_message_arrives = 0;
public void open_connection(String ip, int
serverPort) {
  try {
    s = new Socket(ip, serverPort);
    createInputOutputStrems();
  } catch (IOException e) {
  System.out.println("Socket:" +
e.getMessage());
  }
}
public void close_connection() {
 if (s != null) {
  try {
    s.close();
  }catch (IOException e) {/*close failed*/}
 }
}
public void send_to(string address, byte[]
msg, int size) {
  try {
    output.writeInt(size);
    output.write(msg);
  } catch (IOException ex) {
    System.out.println("DRIVER (send_to):"
+ ex.getMessage());
  }
}
public void run(){
  while (true) {
   if (new_message_arrives == 0){
     int nb;
     try {
       nb = input.readInt();
       byte[] packet = new byte[nb];
       for (int i = 0; i < nb; i++) {
          packet[i] = input.readByte();
       }
       new_message_arrives = 1;
       notify(); //midsn_ioAdapter is the
observer that is notified
     } catch (IOException ex) {
        System.out.println("DRIVER (run):"
+ ex.getMessage());
     }
    }
   }
}
public byte[] midsn_get_received_packet() {
 new_message_arrives = 1;
 return packet;
}
```

a) Using Contiki-OS **b) Using Java for Linux**

Figure A.1 – Implementation of the communication driver

midsn_get_received_packet – is the method that gives the received message to the io_Adapter of MidSN_NC. When the message is pulled by the MidSN-NC, this method sets the *new_message_arrives* flag to zero, which means that there isn't a new message to be consumed by the MidSN_NC.

Similar to the Contiki-OS implementation, Figure A.1b) shows the communication driver implementation using Java, an object-oriented language. In this implementation we use a TCP socket. The functionality of each method is equivalent to the previous description, where the *run* method represents the *packet_recv* method described above.

Appendix B

The Catalog Structure

In this Appendix, we show the structure of the MidSN Catalog (Figure B.1). It is XML-based Catalog. This Catalog is used to keep information about nodes concerning addresses (global IP address and proprietary communication address), current node configuration and node status. It is also responsible for keeping a history of submitted configurations and network configuration.

```
<MidSN_Catalog>
       <Gateway>
         <id> net1 </id>
         <ip_Address> 10.3.3.82 </ip_Address>
         <iIP_address> 10.3.3.82 </iIP_address>
         <iIP_port> 5000 </iIP_port>
         <iWSN_protocol> Rime </iWSN_protocol>
         <iWSN_address> 0x0000 </iWSN_address>
         <iWSN_port> 5000 </iWSN_port>
         <iWSN_channel> 20 </iWSN_channel>
         <node>
           <id> 1.1 </id>
           <ip_Address> 10.3.3.101 </ip_Address>
           <comm_protocol> Rime </comm_protocol>
           <protocol_address> 0x0001 </protocol_address>
           <protocol_port> 5000 </protocol_port>
           <protocol_channel> 20 </protocol_channel>
           <controllers>
              <controller>
                <name> PID controlled </name>
                <running> no <running>
              </controller>
              <controller>
                <name> PD controlled </name>
                <running> yes <running>
              </controller>
           </controllers>
           <streams>
              <stream>
                <name> monitorStream  </name>
                <rate> 3s </rate>
                <window> 1 </window>
                <sendTo> 10.3.3.82 </sendTo>
       <deactivated> no </deactivated>
       <measures>
         <measure>
             <measure_name> Level </measure_name>
             <measure_metric> value </measure_metric>
         </measure>
         <measure>
             <measure_name> Pressure </measure_name>
             <measure_metric> value </measure_metric>
         </measure>
       </measures>
     </stream>
  </streams>
  <alarms>
     <alarm>
       <name> pressureAlarm  </name>
       <rate> 1s </rate>
       <window> 1 </window>
       <sendTo> 10.3.3.82 </sendTo>
       <deactivated> yes </deactivated>
       <measure>
             <measure_name> Pressure </measure_name>
             <measure_metric> value </measure_metric>
       </measure>
       <operator> > </operator>
       <measure>
             <measure_name> Value </measure_name>
             <measure_metric> 2 </measure_metric>
       </measure>
     </alarm>
  </alarms>
  <actions>
     <action>
       <name> pressureAlarm  </name>
       <rate> 1s </rate>
       <window> 1 </window>
       <sendTo> -1 </sendTo>
       <deactivated> no </deactivated>
       <measure>
             <measure_name> Pressure </measure_name>
             <measure_metric> value </measure_metric>
       </measure>
       <operator> > </operator>
       <measure>
             <measure_name> Value </measure_name>
             <measure_metric> 3 </measure_metric>
       </measure>
       <actuation>
             <actuator> DAC0 </actuator>
             <value> 100 </value>
       </actuation>
     </action>
  </actions>
  <status>
     <battery> 2.9 </battery>
     <msg>
       <sent> 20 </sent>
       <received> 2 </received>
       <forwarded> 2 </forwarded>
       <lost> 2 </lost>
     </msg>
     <sensors>
       <sensor> Level </sensor>
       <sensor> Pressure </sensor>
     </sensors>
     <actuators>
       <actuator> DAC0 </actuator>
       <actuator> DAC1 </actuator>
     </actuators>
  </status>
</node>
<node>
```

```
           <id> 1.2 </id>
           <ip_Address> 10.3.3.102 </ip_Address>
           <comm_protocol> Rime </comm_protocol>
           <protocol_address> 0x0002 </protocol_address>
               (...)
         </node>
               (...)
       </Gateway>
       <node>
         <id> pc1 </id>
         <ip_Address> 10.3.3.87 </ip_Address>
         <comm_protocol> IP </comm_protocol>
         <protocol_address> 10.3.3.87 </protocol_address>
         <protocol_port> 5000 </protocol_port>
               (...)
       </node>

       (...)

</MidSN_Catalog>
```

Figure B.1 – MidSN-Catalog (structure)

Appendix C

User API

In this Appendix, we describe the user API included by the MidSN architecture. As discussed before, this API is extensible and can include features to fit different application contexts.

The functionalities of the MidSN API described here are divided into seven categories. Next, we describe each category and its calls.

C.1. Node

The MidSN API has a set of functionalities that allows controlling the nodes. It includes primitives to activate/deactivate nodes, i.e. all executions inside a node are suspended if the deactivate node call is issued to the node.

Besides activate/deactivate nodes, there are also primitives to activate/deactivate sensors, request node status, reset a node and ping a node.

Activate/deactivate sensors allows controlling which sensors are able to sample. This allows, for instance, to save energy by switching on and off the sensors.

The request node status primitive allows collecting information about nodes (e.g. battery, messages losses, and latencies), while the reset function resets a node and starts MidSN-NC with default configurations.

The ping command is used to verify that a node is live and that it can communicate with another.

Table C.1 shows the default node primitives included in MidSN API. All of these primitives use a list of nodes ([NODE]) as argument to identify the nodes where the operation is applied. This list of nodes can include WSN nodes and/or PCs.

Table C.1 – Node primitives

Function	Primitive
Activate / deactivate nodes	`MIDSN.Node.run([NODE,], TRUE or FALSE);`
Activate / deactivate sensors and actuators connected to each node	`MIDSN.Node.Sensors.run (([NODE,], [SENSOR,], TRUE or FALSE);`
Request node status	`MIDSN.Node.status([NODE,]);`
Reset a node	`MIDSN.Node.reset([NODE,]);`
Ping a node	`MIDSN.Node.ping([NODE,]);`

C.2. Operations and filters

Operations and filters functionalities allow creating operations and filters to be processed in nodes. This category of API functionalities includes primitives to create operations (periodical or one time operations), activate/deactivate their execution, change the execution periodicity or delete them from nodes.

Each operation corresponds to data collection, processing and sending tasks according to the operation configurations. It allows, for instance, collecting sensor readings, computing an average and sending the output result to other node.

The activate/deactivate operation execution primitives are used to suspend the execution of a stream. For instance, we can use these primitives to suspend execution of an operation during maintenance and resume execution afterwards.

The change execution periodicity primitive allows changing the rate of an operation. It is useful to allow changing only the rate instead of changing all operation configurations.

This category of API functionalities also includes primitives to create smoothing filters over measures, which are associated to operations. Assuming an industrial environment where noise is an important issue and appears coupled to the measures, this filter allows, for instance, to reduce the influence of noise by always averaging over a set of sensed values.

Table C.2 shows the MidSN API Primitives concerning operations and filters.

Table C.2 – Operations and filters primitives

Function	Primitive
Create Operation	`MIDSN.Operation.create([NODE,],` `OP_NAME,` `ACQUISITION_RATE,` `SEND_RATE,` `WINDOW_SIZE,` `<List> MEASURE (source, metric),` `<List> CLIENT (address, port)` `);`
Delete operation from node	`MIDSN.Operation.drop(([NODE,],` `OP_NAME` `);`
Start and stop operation execution	`MIDSN.Operation.run(OP_NAME,` `TRUE or FALSE` `);`
Change operation periodicity	`MIDSN.Operation.setPeriodicity( OP_NAME,` `NEW_RATE` `);`
Create data filter (where)	`MIDSN.Filter.create([NODE,],` `FILTER_NAME,` `OP_NAME,` `CONDITION (MEASURE, operator, MEASURE)` `);`
Drop data filter	`MIDSN.Filter.drop([NODE,],` `FILTER_NAME` `);`

Depending on the functionality, some arguments are needed to manage operations and filters. For instance, the Create Operations call creates a stream that is processed and sent to another node or control station. This primitive needs arguments such as a unique identifier, an acquisition rate, an execution rate, a list of measures that must be included into the stream, and a list of client (other nodes) that will receive the stream output.

Each operation can be executed periodically (with a period defined through the `SEND_RATE` field) or one time (`SEND_RATE = 0`). It is also possible to create operations that only collect data and store it in the node (`SEND_RATE = -1`). In this case, data received by a node is stored inside it without any processing.

Moreover, operations have a window size parameter associated to them. This value is used to limit data values used to process computations, and/or to limit the number of tuple stored in the corresponding stream.

The stream output will be constructed based on the configured measures. Each measure has a measure name (e.g. temperature, humidity, light) and a metric (e.g. average, maximum, minimum, percentile). Figure C.1 shows the calls to configuration web-service API methods that were issued by the configuration software to start a sensor collection operation in nodes 1.1, 1.2, 1.3 with 3 seconds of acquisition and sending rates.

```
// create stream operation for reading from sensor
(a) MIDSN.Operation.create( [1.1, 1.2, 1.3] ,
                             "pressureStream",
                             3s,
                             3s,
                             1,
                             {(PRESSURE, VALUE)},
                             {ControlStation, PS}
                       );
```

Figure C.1 – Piece of code to create periodic operation and send data to the control station

Besides configuring nodes to send data readings, it is needed to configure the control station to collect it. Figure C.2 shows how to configure the control station to collect the sensor readings. This configuration is done automatically by the RConfig component when the configuration of Figure C. is called. However its configurations can be changed by the user.

```
      // save stream at control station
      (b) MIDSN.Operation.create(ControlStation ,
                                   "pressureStreamCS",
                                   -1,
                                   -1,
                                   10000,
                                   {("pressureStream", VALUES)},
                                   -1
                             );
```

Figure C.2 – Piece of code to collect sensor reading in control station

In the example of Figure C.2 we create an operation with the identifier "`pressureStreamCS`" that will receive values from the stream "`pressureStream`" and store them until reaching the maximum number of samples, in this case 10000 samples. The fields `ACQUISITION_RATE`, `SEND_RATE`, and `CLIENT` are filled with -1 to indicate that it is not a periodic operation with acquisition or sending parts and the values are not sent to other nodes.

During the lifetime of the network, we can change its configuration. In the next example we will use the API to change the rate from 3 to 5 seconds, and 1 minute after that we stop the operation. The following code extracts are used to perform this.

```
      //Modify the operation rate to every five seconds:
      MIDSN.Operation.setPeriodicity("pressureStream", 5s);

      //Stop operation execution (then restart):
      MIDSN.Operation.run("pressureStream", FALSE);
      (MidSN.Operation.run("pressureStream", TRUE); )
```

Figure C.3 – Piece of code to change operation rate, stop and start the execution

In some applications, especially in industrial applications, the readings gathered from physical sensors may need to be filtered. For instance, to reduce the influence of noise, it is possible to apply averaging over some values which is a smoothing filter over the measures. Figure C.4 shows another example of how to create a filter over the pressure values. In this example, we removed all values that were above 6 bars.

```
//Modify the operation rate to every five seconds:
MIDSN.Filter.create([1.1, 1.2, 1.3],
  "Pressure_NoNoise",
  "pressureStream",
  CONDITION ((PRESSURE, VALUE), <, (VALUE, 6))
);
```

Figure C.4 – Piece of code to create a filter

C.3. Alarms

In some applications, alarms are needed to inform a user of an imminent or occurring emergency. MidSN alarm API functionalities presented here were created to establish a configuration interface for those applications, where functionalities to create, drop, start and stop alarms were included. Those alarms are based on conditions that are submitted by users.

Alarms are used to specify an operation that is sent each time a specific condition occurs in node(s). The condition is defined by a set of parameters such as, an identifier, a condition and a list of clients that will receive the alarm values.

The alarm configuration includes a condition that needs to be true to send data to the clients (other nodes). Each condition is defined by two measures and one operator, where the operator can assume one of the following symbols: <=, <, =, >, >=. It allows, for instance, creating conditions similar to avg(temperature) > 30 ºC.

Table C.3 shows the API Primitives concerning alarms.

Table C.3 – Alarm primitives

Function	Primitive
Create Alarm	`MIDSN.Alarm.create ([NODE,], ALARM_NAME, OP_NAME, CONDITION (MEASURE, operator, MEASURE), <List> CLIENT (address, port) );`
Delete Alarm from node	`MIDSN.Alarm.drop( ALARM_NAME );`
Start and Stop Alarm verification	`MIDSN.Alarm.run( ALARM_NAME, TRUE or FALSE );`

In the next example (Figure C.5) we configure an alarm in the control station when pressure sensor data is above a certain threshold (3 bars). We also create another alarm on a sensor node 1.1 for the same effect, but with a different threshold (2 bars).

```
      //Raise alarm on the control station every time pressure goes
above a value of 3 bars:
      MIDSN.Alarm.create(controlSation,
                        "ServerPressureAlarm",
                        "pressureStream",
                        CONDITION ((PRESSURE, VALUE), >, (VALUE, 3)),
                        -1
                        );

      //An alarm is also to be raised on the sensor node every time
pressure goes above a value of 2 bars:
      MIDSN.Alarm.create( 1.1,
                        "pressureAlarm",
                        "pressureStream",
                        CONDITION ((PRESSURE, VALUE), >, (VALUE, 2)),
                        {ControlStation}
                        );
```

Figure C.5 – Piece of code to create an alarm

C.4. Actions

MidSN also provides functionalities to create and manage actions. Actions are used to actuate in nodes when a specific condition occurs. Each action is associated to an operation and includes a condition and an actuation. The condition is defined the same way as creating an alarm.

The actuation is defined though the indication of an actuator, the value that the node must write to the actuator and the node address where the actuator is connected. If the actuation is done in the same node, this field is filled with -1. Table C.4 shows the syntax of these functionalities.

Table C.4 – Action primitives

Function	Primitive

Create Action	`MIDSN.Action.create([NODE,],`
	`  ACTION_NAME,`
	`  OP_NAME,`
	`  CONDITION (MEASURE, operator, threshold),`
	`  ACTUATION (`
	`          ACTUATOR_NAME,`
	`          value,`
	`          TARGET_NODE`
	`          )`
	`);`

Drop Action	`MIDSN.Action.drop(ACTION_NAME);`

Actions can be configured to prevent accidents. Assuming that pressure above 5 bars can explode a pipe (for example), we can specify an action inside a node to switch on a valve when pressure goes above 5 bars. Each action is executed every time that the corresponding operation is executed. Figure C.6 shows how to configure an action to handle those specifications.

This example consists on evaluating (inside node 1.1) the condition over each pressure value of the stream `"pressureStream"`, and actuating over the `PIPE_VALVE` (connected to the same node) if the condition is true.

```
      MIDSN.Action.create( 1.1,
                           “pressureAction”,
                           “pressureStream”,
                           CONDITION ((PRESSURE, VALUE), >, (VALUE, 5)),
                           ACTUATION (PIPE_VALVE, OPEN, -1)
                           );
```

Figure C.6 – Piece of code to create an action

C.5. Actuations

MidSN allow users to submit actuation commands directly to each node. Table C.5 shows the primitive used to submit actuation commands directly to nodes. This primitive receives as arguments, a list of nodes where the actuation will be performed and the actuation parameters. Each actuation is defined by the actuator identifier and the value that we want to apply to the actuator.

Table C.5 – Actuation primitives

Function	Primitive
Send and apply actuation value	`MIDSN.Actuate([NODE,],` `ACTUATION (ACTUATOR_NAME, <parameters>)` `);`

C.6. Publish/Subscribe

To interface MidSN with external applications, the proposed architecture includes a publish/subscribe mechanism. Table C.6 shows the API primitives that allow subscribing and unsubscribing stream data.

Table C.6 – Publish/Subscribe primitives

Function	Primitive
Subscribe Data	`MIDSN.PS.subscribe(SUBSCRIBER_ADDRESS,` `SUBSCRIBER_RECEPTION_PORT,` `OP_NAME,` `CONNECTION_TIMEOUT` `);`
Unsubscribe data	`MIDSN.PS.subscribe(SUBSCRIBER_ADDRESS,` `OP_NAME` `);`

The subscribe stream data function allows to subscribe stream data. To do a subscription, external applications need to call this function with the network address, a port where data will be received, the stream data name and the timeout used to close connection between the publisher (MidSN) and the external application that wants to receive stream data. This function must be called for each stream data subscription. Figure C.7 shows an example of using the subscription functionality.

```
    MIDSN.PS.subscribe(10.3.1.132,
                                5000,
                                "pressureStream",
                                60000 //ms
                         );
```

Figure C.7 – Piece of code to subscribe stream data

The example consists on a subscription to the `"pressureStream"` configured in Figure C.. Upon receiving stream data, the publisher will send it to the address and port specified in the call, 10.3.1.132 and 5000, respectively. Lastly, the timeout value (60000) is used to close the connection with the client. For instance, if the connection of the nodes 1.1, 1.2, 1.3 is lost, the stream `"pressureStream"` will never arrive at the publish/subscribe. After this timeout, the publisher will close the connection to the external application, because it is not needed. Meanwhile, a new connection is opened if the stream `"pressureStream"` arrives at the publish/subscribe.

C.7. Agents

MidSN also provides functionalities to receive agents over-the-air from users. It offers functions to dynamically send agents to nodes, load an agent from flash memory and prepare it to run with default values, start/stop agents, drop an agent or change parameters used by the agent. Table C.7 shows the primitives developed to manage agents in nodes.

Table C.7 – Agent primitives

Functionality	**Primitive**

Send an Agent	`MidSN.Agent.create ([NODE,],`
	`  AGENT_ID,`
	`  AGENT_CODE_PATH`
	`);`

Drop Agent	`MidSN.Agent.drop ([NODE,],`
	`  AGENT_ID`
	`);`

Load Agent	`MidSN.Agent.load([NODE,],`
	`  AGENT_ID`
	`);`

Unload Agent	`MidSN.Agent.unload([NODE,],`
	`  AGENT_ID`
	`);`

Start and Stop Agents	`MidSN.Agent.start([NODE,],`
	`  AGENT_ID,`
	`  <INIT_VALUES>,`
	`  TRUE or FALSE`
	`);`

Send Parameters to Agents	`MidSN.Agent.setParameters([NODE,],`
	`  AGENT_ID,`
	`  <List> VALUES`
	`);`

Users who want to add functionalities dynamically to a node should use this set of primitives. It allows, for instance, sending an agent to a node. To do this, users need to develop the agent code, compile it and call the "`MidSN.Agent.create`" method. This method will receive as argument the target node(s), an agent id to identify the agent and the path to the binary image code. Figure C.8 shows an example of using this functionality.

```
MidSN.Agent.create (Node1.1,
                  "PID_controller",
                  C:\\controllers\pid.ihex
               );
```

Figure C.8 – Piece of code to send an agent

After the binary image is loaded by nodes, the user needs to call the load and stats functions to execute the agent in the node. Figure C.9 shows how to do this.

Since we are assuming that our controller doesn't need initial parameters, we use -1 in the `<INIT_VALUES>` field.

```
// load the agent
      MidSN.Agent.load (Node1.1,
                  "PID_controller"
               );

// start it execution
      MidSN.Agent.run (Node1.1,
                  "PID_controller",
                  -1,
                  TRUE
               );
```

Figure C.9 – Piece of code to send an agent

Appendix D

Custom Code Agents – Code Example

In this Appendix, we show the code for an agent who computes a closed-loop algorithm using the data collector module (NC-GinApp-DC) to read data from a stream and to actuate over the environment through the NC-GinApp-AA module.

In this example we develop an agent to configure a stream to run inside the small operating machine of the MidSN-NC, read data from the stream to our agent periodically, and compute a PID value. After the computation, the agent calls the MidSN-NC to write the actuation value to the DAC0 actuator.

```
      PROCESS_THREAD(test_blink, ev, data)
      {
        static struct etimer t;
        PROCESS_BEGIN();
        uint16_t period = CLOCK_SECOND;
        etimer_set(&t, period);

        structure SQL sql;
        sql.select[0] = SENSOR_TEMPERATURE;
        sql.select[1] = END;
        sql.from[0] = SENSOR_TEMP;
        sql.from[1] = END;

        uint16_t nSamples = 1;
        uint16_t   result   =   MidSN.createStream(   1,   period,   sql,
nSamples, false );

        if ( result == 0 ){
            leds_on(LEDS_ GREEN); //run
            while(1) {
              PROCESS_WAIT_UNTIL(etimer_expired(&t));
              etimer_reset(&t);
              uint16_t  data[nSamples]  =  MidSN.readDataFromStream(  1,
nSamples );
              uint16_t Kp = 0.60 * data[0];
              uint16_t Ki = ( 2 * Kp ) / Pu;
              uint16_t Kd = ( Kp * Pu ) / 8;
              uint16_t PID = Kp + Ki + Kd;
              MidSN.writeToActuator( DAC0, PID);
            }
        } else leds_on(LEDS_RED); //Error
        PROCESS_END();
      }
```

Figure D.1 – Example of Contiki program that determines an actuation using MidSN-NC capabilities

Appendix E

Message Formats

In this Appendix, we detail the message structure used to exchange commands and data between nodes.

The MidSN network and I/O adapters provide communication primitives for inter node interactions. It implements two types of messages: command messages and data messages. These two types are identified by a flag “Msg Type“ that is included in each message. Table E.1 shows the necessary types with a small description of each one.

Table E.1 – Message type

Types of messages	Description
`MSG_CMD`	Used to send a command to the node.
`MSG_ACK`	Used to send a confirmation that a message was received by the node.
`MSG_CMDDONE`	Indicates that a command was performed.
`MSG_DATA`	Indicates that payload data contains the data of a stream.

The message structure of MidSN is show in Figure E.1. Each message includes two unique IDs to identify the node source and the node destination, one field to indicate the message type, a control type to introduce the information of a command (for `MSG_CMD` type), a sequence number, the length of payload and a set of parameters (payload).

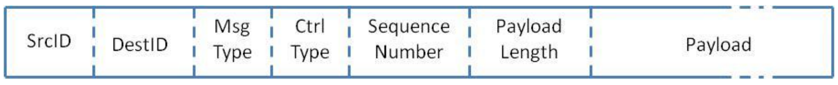


Figure E.1 – Format of message

The message type field indicates the type of message. For instance, if message type is filled with `MSG_DATA`, the payload of message corresponds to a stream data. In this case, the control type field is neglected. However, when a message corresponds to a command (`MSG_CMD`), the control type field is used to specify the command that is sent in the message. Depending on the command, the payload field is filled with all parameters needed to configure a node correctly.

Command types can be divided into two main categories: node operation and configuration of periodical operations.

Node operation: The node operation type is related with node's operation and supports commands such as start, stop, reboot, ping, register new API functionalities, upload or drop agents, load agents from flash memory and to register new sensor or actuator. Table E.2 shows the command types related with node operation, including a small description of each one.

Table E.2 – Node operation – Command types

Command types	**Description**
`MSG_REBOOT`	Restart whole software in the node.
`MSG_STATUS`	Get Status from node
`MSG_REG_API_METHOD`	Register new API method in the node
`MSG_UP_AGENT`	Upload new agent to the node
`MSG_LOAD_AGENT`	Load an agent from flash memory to main memory for execution.
`MSG_DROP_AGENT`	Remove an agent from the node
`MSG_START_AGENT`	Initialize an agent in the node
`MSG_STOP_AGENT`	Stops the execution of an agent in the node
`MSG_REG_SENSOR`	Upload a driver to a new sensor and initialize it

`MSG_REG_ACTUATOR`	Upload a driver to a new actuator and initialize it

The node operation commands can be divided into two categories: basic commands (e.g. reboot, status) and advanced ones (the remaining methods shown in Table E.2). For instance, if we want to send a reboot command to a node, we only need to fill the header of the message. Figure E.2 shows how to create this subset of messages.

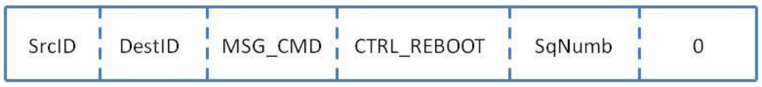


Figure E.2 – Format of start message

The advanced commands assume a payload specification that must be filled according to the command. For instance, if we want to send a start, stop or drop agent command to a node, we need to fill the payload with the agent identification (*agent_ID*). However, if we want to register a new API method or upload new agents to the node, we need to send the byte code in the payload.

Periodical operations: This type of command allows users to configure data streams, alarms and condition-based actions to operate in the node. These commands require that the payload be filled with operation configuration parameters. Figure E.3 shows how to fill the payload to configure an existing or a new periodical operation.

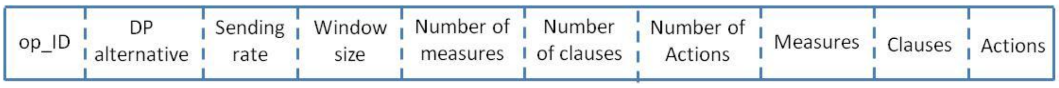


Figure E.3 – Payload specification for operation configuration

The payload includes all parameters needed to define a stream (see Section 5.6). The stream configuration may include the definition of an in-network processing technique. Table E.3 shows which metrics are now supported by our prototype:

Table E.3 – Supported metrics

Metric	Description
`FIELD_P0 to FFIELD_P99`	Indicates a percentile (0-99)
`FIELD_VALUE`	Current sensor value
`FIELD_MAX`	Maximum over window

`FIELD_MIN`	Minimum over window
`FIELD_AVG`	Average over window
`FIELD_COUNT`	Count over window
`FIELD_SUM`	Sum over window
`FIELD_VARIANCE`	Variance over window
`FIELD_MERGE`	Merge and send over window
`FIELD_RLE`	Compute RLE compression over window
`FIELD_LAST`	Last sensor value over window
`FIELD_FIRST`	First sensor value over window

Appendix F

XML Message Generated by the Gateway – Example

In this Appendix, we show an example of XML message (Figure F.1) generated and sent by the gateway. That message includes the sensor values (temperature, humidity, light), timestamps (generation time, dispatcher in, dispatcher out), performance, debugging and command information (fields related to debugging and command information are filled by the sending node to be analysed by the control station).

```
<wsnMessage messageMode="102" sourceID="0005" messageSeqNo="69">
   <parameter name="genTime">1327449360300</parameter>
   <parameter name="dispIn">1327449360400</parameter>
   <parameter name="dispOut">1327449360401</parameter>
   <parameter name="delivery_time">90</parameter>
   <parameter name="hops">2</parameter>
   <parameter name="ginmac_seqno">76</parameter>
   <parameter name="slotNumber">125577445</parameter>
   <parameter name="format">17</parameter>
   <parameter name="hwid">102</parameter>
   <parameter name="tx_count">35148</parameter>
   <parameter name="light_photosynthetic">135</parameter>
   <parameter name="light_solar">68</parameter>
   <parameter name="temp">0</parameter>
   <parameter name="humidity">0</parameter>
   <parameter name="adc0">1509</parameter>
```

```
        <parameter name="adc1">2209</parameter>
        <parameter name="battery">2635</parameter>
        <parameter name="battery_indicator">0</parameter>
        <parameter name="type_of_last_recieved_cmd">0</parameter>
        <parameter name="nseq_last_received_cmd">0</parameter>
        <parameter name="time_to_deliver_last_cmd">0</parameter>
        <parameter name="elapsed_time_between_last_two_cmds">0</parameter>
        <parameter name="endToEndPktLossRate">0.6726</parameter>
        <parameter name="endToEndPktResendRate">0.0000</parameter>
        <parameter name="totalPacketCount">139121</parameter>
        <parameter name="totalLostPacketCount">942</parameter>
        <parameter name="totalRetransmittedPacketCount">0</parameter>
        <parameter name="serialLatency">3.742904</parameter>
     </wsnMessage>
```

Figure F.1 – Example of xml message generated by the gateway

Appendix G

Network Configuration - Example

In this Appendix, we show an example of how to define a network configuration.

Considering the network configuration represented in Figure G.1, the user needs to convert it to a plan-text format to introduce it in the planning algorithm. Figure G.2 shows the corresponding plan-text format.

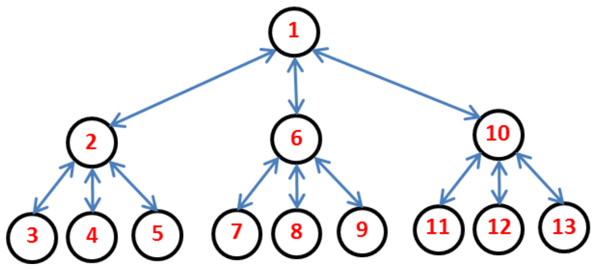


Figure G.1 – Network configuration

The plan-text format is defined by two structures: tree structure, which includes information about node connectivity. For each leaf node, it includes which nodes are part of the path between the leaf node and sink node; and a treeAddr structure. This structure maps the node id to a specific node address. This address depends on the network communication protocol. In this example we are using the Rime addressing scheme.

```
const STATICTOP Tree [MAX_LEAF_NODES] ={
       {3, 2, 1},
       {4, 2, 1},
       {5, 2, 1},
       {7, 6, 1},
       {8, 6, 1},
       {9, 6, 1},
       {11, 10, 1},
       {12, 10, 1},
       {13, 10, 1}
};

const STATICTADDR treeAddr[MAX_NODES] ={
       {1, 0x0000},
       {2, 0x0002},
       {3, 0x0003},
       {4, 0x0004},
       {5, 0x0005},
       {6, 0x0006},
       {7, 0x0007},
       {8, 0x0008},
       {9, 0x0009},
       {10,0x000A},
       {11,0x000B},
       {12,0x000C},
       {13,0x000D}
};
```

Figure G.2 – Network configuration - plan-text format

Appendix H

Network Layout - Example

In this Appendix, we show an example of how to define a network layout to be recognized by the planning algorithm.

The network layout used is based on the GINSENG and GinMAC definitions. It includes all information of about the network configuration (see Appendix G) and a TDMA schedule. This schedule is defined by the user. It must be introduced in the algorithm in a plan-text fashion as shown in Figure H.1.

```
const SLOT_ADDRS epoch[SLOTS_PER_EPOCH] = {

       //Upstream B1
       {0x0003,0x0002},
       {0x0003,0x0002},
       {0x0004,0x0002},
       {0x0004,0x0002},
       {0x0005,0x0002},
       {0x0005,0x0002},
       {0x0002,0x0000},
       {0x0002,0x0000},
       {0x0002,0x0000},
       {0x0002,0x0000},
       {0x0002,0x0000},
       {0x0002,0x0000},
       {0x0002,0x0000},
       {0x0002,0x0000},

       //Upstream B2
       {0x0007,0x0006},
       {0x0007,0x0006},
       {0x0008,0x0006},
       {0x0008,0x0006},
       {0x0009,0x0006},
       {0x0009,0x0006},
       {0x0006,0x0000},
        {0x0006,0x0000},
        {0x0006,0x0000},
        {0x0006,0x0000},
        {0x0006,0x0000},
        {0x0006,0x0000},
        {0x0006,0x0000},
        {0x0006,0x0000},

        //Upstream B3
        {0x000B,0x000A},
        {0x000B,0x000A},
        {0x000C,0x000A},
        {0x000C,0x000A},
        {0x000D,0x000A},
        {0x000D,0x000A},
        {0x000A,0x0000},
        {0x000A,0x0000},
        {0x000A,0x0000},
        {0x000A,0x0000},
        {0x000A,0x0000},
        {0x000A,0x0000},
        {0x000A,0x0000},
        {0x000A,0x0000},

        //Processing slots (= SLOTS_PROC)
        {0x0, 0x0},
        {0x0, 0x0},

        //TS (Time synchronization)
        {0x0000,0xffff},
        {0x0001,0xffff},
        {0x0002,0xffff},
        {0x0003,0xffff},

        //Downstream slots
        {0x0000,0xffff},
        {0x0002,0xffff},
        {0x0006,0xffff},
        {0x000A,0xffff}
};
```

Figure H.1 – TDMA schedule – plan-text format

Appendix I

Evaluation of MidSN – Details

In this Appendix, we detail in form of tables the values shown in charts of Chapter 9. Each table includes in the caption the number of the corresponding Figure that the values are related to.

Table I.1 – Programming memory footprint for all platforms [Bytes] - Figure 9.1

Component	TelosB (Contiki-C)	Arduino (C++)	Raspberry PI (Java)	Computer (Java)
I/O Adapter	1260	2524	8703	8803
NC-Kernel-AM	1344	312	6400	6500
NC-GinApp-CM	880	1180	5000	6000
NC-GinApp-DC	2270	8732	12100	12100
NC-GinApp-GP	5104	11104	20700	20890
NC-GinApp-AA	544	834	3600	0
Contiki-OS	20009			

Table I.2 – RAM memory footprint for all platforms [Bytes] - Figure 9.2

Component	TelosB (Contiki-C)	Arduino (C++)	Raspberry PI (Java)	Computer (Java)
I/O Adapter	486	882	616	616
NC-Kernel-AM	631	20	963	963
NC-GinApp-CM	62	118	592	792
NC-GinApp-DC	636	1932	1078	1278
NC-GinApp-GP	664	2548	9644	10044
NC-GinApp-AA	38	52	120	0
Total	2517	5552	13013	13693

Table I.3 – Time required per operation over a stream in memory [ms] - Figure 9.4

Operation		TelosB (Contiki-C)	Arduino (C++)	Raspberry PI (Java)	Computer (Java)
Select AGGREG()	Avg	14.00	9.35	2.28	0.76
	Stdev	0.35	1.20	0.17	0.04
	Max	15.00	12.00	2.90	0.86
Select Percentile()	Avg	28.00	18.70	4.56	1.52
	Stdev	0.21	1.43	0.50	0.27
	Max	29.00	21.00	5.10	1.80

Table I.4 – Consumed energy for data tuples manipulation [mJ] - Figure 9.8a)

Operation		1 Tuple	10 Tuples	50 Tuples	100 Tuples	1000 Tuples
Write	Avg	0.1427	0.1604	0.2868	0.4970	3.7128
	Stdev	0.1179	0.2108	0.4690	0.5722	1.0989
	Max	0.7733	0.9598	1.7174	2.6668	5.0633
Read	Avg	0.1070	0.1203	0.2151	0.3727	2.7846
	Stdev	0.0884	0.1064	0.1018	0.6792	1.0678
	Max	0.5800	0.4699	1.1038	2.7501	4.5475

Table I.5 – Consumed energy for data tuples manipulation [mJ] - Figure 9.8b)

Node	1 Tuple	10 Tuples	50 Tuples	100 Tuples	1000 Tuples
Leaf	0.611	6.111	30.556	61.111	611.111
Relay	0.889	8.892	44.462	88.923	889.234

Table I.6 – Command latency for the three platforms [ms] - Figure 9.12

Command latency	TelosB	Arduino	Raspberry PI
Avg	21.81	206.38	3.57
Stdev	3.51	97.82	2.15
Max	32.19	910.49	11.65
Min	13.56	52.01	1.92

Table I.7 – Data latency for the three platforms [ms] - Figure 9.14

Data latency		**TelosB**	**Arduino**	**Raspberry PI**
Establishes connection all times	Avg	15.8	486.7	10.4
	Stdev	1.5	87.8	2.7
	Max	18.0	630.8	13.2
	Min	11.7	52.0	6.9
Connection opened, data transmission only	Avg	15.8	196.4	3.6
	Stdev	1.5	117.8	2.2
	Max	18.0	370.5	5.1
	Min	11.7	51.0	2.8

Table I.8 – Data latency for TelosB per part [ms] - Figure 9.15

Data Latency	**Time in WSN**	**Time in Gateway**	**Time from gateway to control station**	**Total**
Avg	12.05	2.20	1.70	15.95
Stdev	0.73	0.72	0.34	1.54
Max	14.23	3.86	2.20	20.99
Min	9.70	2.80	1.00	13.69

Table I.9 – Closed-loop latency over heterogeneous network [ms] - Figure 9.18

Latency	**TelosB -> TelosB**	**TelosB -> Arduino**	**TelosB -> Raspberry PI**
Avg	62.05	246.65	43.14
Stdev	6.60	101.38	5.22
Max	84.03	870.14	53.99
Min	49.86	136.95	30.13

Table I.10 – Closed-loop latency over heterogeneous network per system parts [ms] - Figure 9.19

Configuration		Acquisition	Data sending time	Processing time	Cmd Sending time	Cmd processing time
TelosB -> TelosB	Avg	22.00	15.95	1.36	21.81	0.92
	stdev	0.00	1.54	0.47	3.51	0.08
	Max	22.00	20.99	2.00	32.19	1.20
	Min	22.00	13.69	0.80	13.56	0.86
TelosB -> Arduino	Avg	22.00	15.95	1.36	206.38	0.95
	stdev	0.00	1.54	0.47	97.82	0.55
	Max	22.00	20.99	2.00	830.14	2.00
	Min	22.00	13.69	0.80	52.01	0.87
TelosB -> Raspberry PI	Avg	22.00	15.95	1.36	3.57	0.25
	stdev	0.00	1.54	0.47	2.15	0.06
	Max	22.00	20.99	2.00	11.65	0.69
	Min	22.00	13.69	0.80	1.92	0.07

Appendix J

Evaluation of Planning and Monitoring Approaches – Details

In this Appendix, we detail in form of tables the values shown in charts of Chapter 10. Each table includes in the caption the number of Figure that the values are related to.

Table J.1 – Monitor latency per node - Figure 10.3

Node	Observed [Avg]	Stdev	Observed [Max]	Observed [P99]
2	33.68	0.55	44.70	36.05
3	113.69	0.52	124.04	115.96
4	173.90	0.45	178.30	176.06
5	173.99	0.90	188.07	176.65
6	33.52	0.48	43.86	35.49
7	114.07	0.55	124.10	116.52
8	173.96	0.63	184.58	176.40
9	173.63	0.55	185.73	175.95
10	33.68	0.52	47.28	35.94
11	113.63	0.50	124.16	115.91
12	173.58	0.69	187.07	175.83
13	173.91	0.60	184.58	176.20

Table J.2 – Monitor latency per level with forecast - Figure 10.4

level	Observed [Avg]	Stdev	Observed [Max]	Minimum	Forecast [max]
Level 1	33.68	0.55	44.70	12.95	51.78
Level 2	113.69	0.52	124.04	92.91	131.78
Level 3	173.99	0.90	188.07	153.01	191.78

Table J.3 – Monitor latency per network part - Figure 10.5

Network part	Observed [Avg]	Stdev	Observed [Max]	Minimum	Forecast [max]
WSN	170.06	0.78	180.00	170.00	180.00
Serial	2.60	0.36	7.34	1.98	7.79
Middleware	1.33	0.28	3.33	0.67	3.99
Total	173.99	0.90	188.07	173.01	191.78

Table J.4 – Event latency per node with forecast - Figure 10.6

Node	Observed [Avg]	Stdev	Observed [Max]	Observed [P99]
2	33.68	0.55	44.70	36.05
3	113.69	0.52	124.04	115.96
4	173.90	0.45	178.30	176.06
5	173.99	0.90	188.07	176.65
6	33.52	0.48	43.86	35.49
7	114.07	0.55	124.10	116.52
8	173.96	0.63	184.58	176.40
9	173.63	0.55	185.73	175.95
10	33.68	0.52	47.28	35.94
11	113.63	0.50	124.16	115.91
12	173.58	0.69	187.07	175.83
13	173.91	0.60	184.58	176.20

Table J.5 – Closed-loop latency - Figure 10.9a)

Network part	Observed [Avg]	Stdev	Observed [Max]	Forecast [max]
WSN Up (node 4)	170.03	0.56	180.00	180.00
Sink Computation	0.61	0.14	0.86	1.00
Wait for TX slot	380.00	0.00	380.00	390.00
WSN Down (node 13)	50.00	0.20	60.00	60.00
Total	600.65	0.61	620.86	631.00

Table J.6 – Closed-loop latency - Figure 10.9b)

Network part	Observed (Avg)	Stdev	Observed (Max)	Forecast [max] if catches downstream slot	Forecast [max]
WSN Up (node 4)	170.03	0.56	180.00	180.00	180.00
Serial Up	2.60	0.36	7.34	7.79	7.79
Middleware Up	0.94	0.19	2.86	3.99	3.99
Middleware Computation	0.61	0.14	0.86	1.00	1.00
Middleware Down	1.33	0.28	3.99	3.99	3.99
Serial Down	2.60	0.36	7.34	7.79	7.79
Wait for TX slot	379.97	0.56	380.00	380.00	1000.00
WSN Down (node 13)	50.00	0.20	60.00	60.00	60.00
Total	608.09	0.83	642.38	644.56	1264.56

Table J.7 – Closed-loop latency - Figure 10.11a)

Network part	Observed [Avg]	Stdev	Observed [Max]	Forecast [max]
WSN Up (node 4)	170.13	0.57	180.00	180.00
Sink Computation	0.59	0.12	0.86	1.00
Wait for TX slot	11.42	0.04	13.00	20.00
WSN Down (node 13)	50.00	0.20	60.00	60.00
Total	232.14	0.61	253.86	261.00

Table J.8 – Closed-loop latency - Figure 10.11b)

Network part	Observed (Avg)	Stdev	Observed (Max)	Forecast [max] if catches downstream slot	Forecast [max]
WSN Up (node 4)	170.01	0.58	180.00	180.00	180.00
Serial Up	2.62	0.31	7.24	7.79	7.79
Middleware Up	0.92	0.22	2.92	3.99	3.99
Middleware Computation	0.61	0.15	0.79	1.00	1.00
Middleware Down	1.31	0.29	3.79	3.99	3.99
Serial Down	2.56	0.34	7.44	7.79	7.79
Wait for TX slot	11.42	0.87	13.00	20.00	1000.00
WSN Down (node 13)	50.00	0.10	60.00	60.00	60.00
Total	239.45	0.83	275.18	284.56	1264.56

Table J.9 – Asynchronous closed-loop latency for all configurations - Figure 10.15

	Case 1	**Case 2**	**Case 3**	**Case 4**
Observed (Avg)	270.20	60.60	70.40	270.50
Stdev	0.31	0.15	0.22	0.34
Observed (Max)	280.00	60.00	80.00	280.00
Forecast [max] if castches the next downstream slot	280.00	60.00	80.00	280.00
Firecast [max] if castches the second next downstream slot	380.00	160.00	180.00	380.00
User requirement	500.00	250.00	150.00	500.00

Table J.10 – Synchronous closed-loop latency for all configurations - Figure 10.16

	Case 1	**Case 2**	**Case 3**	**Case 4**
Observed (Avg)	55.0	57.6	52.3	54.3
Stdev	28.8	29.4	28.6	29.4
Observed (Max)	114.8	128.7	106.8	113.0
Forecast [max]	131.8	131.8	131.8	131.8
User requirement	500.0	250.0	150.0	500.0

References

[1] Ginseng Members, “GINSENG - Home.” [Online]. Available: http://www.ict-ginseng.eu/. [Accessed: 05-Mar-2013].

[2] P. Suriyachai, J. Brown, and U. Roedig, “Time-Critical Data Delivery in Wireless Sensor Networks,” in *Distributed Computing in Sensor Systems*, 2010, vol. 6131, pp. 216–229.

[3] Crossbow, *M ICAz Datasheet*. 2007.

[4] Crossbow, *TelosB Datasheet*. 2004, pp. 1–28.

[5] Libelium, *Waspmote Datasheet*. 2010.

[6] F. Semiconductor, *Econotag Datasheet*. 2011.

[7] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, “Tinyos: An operating system for sensor networks,” in *Ambient Intelligence*, 2005, vol. II, no. August, pp. 115–148.

[8] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *29th Annual IEEE International Conference on Local Computer Networks*, 2004, pp. 455–462.

[9] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, “MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms,” in *Mobile Networks and Applications*, 2005, vol. 10, no. 4, pp. 563–579.

[10] C. C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, “SOS: A dynamic operating system for sensor networks,” in *Third International Conference on Mobile Systems Applications And Services (Mobisys)*, 2005, pp. 163–176.

[11] M. Kuorilehto and T. Alho, “SensorOS : A New Operating System for Time Critical,” in *Interface*, 2007, pp. 431–442.

[12] R. Barr, J. Bicket, D. Dantas, B. Du, T. W. Kim, B. Zhou, and E. G. Sirer, “On the need for system-level support for ad hoc and sensor networks,” in *ACM SIGOPS Operating Systems*, 2002, vol. 36, no. 2, pp. 1–5.

[13] A. Eswaran, A. Rowe, and R. Rajkumar, “Nano-RK: an energy-aware resource-centric RTOS for sensor networks,” in *26th IEEE International RealTime Systems Symposium RTSS05*, 2005, vol. 0, pp. 256–265.

[14] “ERIKA Enterprise |.” [Online]. Available: http://erika.tuxfamily.org/drupal/. [Accessed: 23-Aug-2013].

[15] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesC language: A holistic approach to networked embedded systems,” in *ACM SIGPLAN*, 2003, vol. 38, no. 5, pp. 1–11.

[16] E. Monmasson and M. N. Cirstea, “FPGA Design Methodology for Industrial Control Systemsamp;#x2014;A Review,” in *IEEE Transactions on Industrial Electronics*, 2007, vol. 54, no. 4, pp. 1824–1842.

[17] H. Hinkelmann, P. Zipf, and M. Glesner, “Design Concepts for a Dynamically ReconfigurableWireless Sensor Node,” in *First NASAESA Conference on Adaptive Hardware and Systems AHS06*, 2006, pp. 436–441.

[18] C. H. Zhiyong, L. Y. Pan, Z. Zeng, and M. Q. H. Meng, “A novel FPGA-based wireless vision sensor node,” in *Automation and Logistics 2009 ICAL09 IEEE International Conference on*, 2009, no. August, pp. 841–846.

[19] Y. Sun, L. Li, and H. Luo, “Design of FPGA-based Multimedia Node for WSN,” in *Simulation*, 2011.

[20] P. Muralidhar and C. B. R. Rao, “Reconfigurable wireless sensor network node based on Nios core,” in *2008 Fourth International Conference on Wireless Communication and Sensor Networks*, 2008, pp. 67–72.

[21] G. Chalivendra, R. Srinivasan, and N. S. Murthy, “FPGA Based Re-Configurable Wireless Sensor etwork Protocol,” in *Electronic Design*, 2008, pp. 1–4.

[22] J.-G. Tong, Z.-X. Zhang, Q.-L. Sun, and Z.-Q. Chen, “Design of Wireless Sensor Network Node with Hyperchaos Encryption Based on FPGA,” in *2009 International Workshop on ChaosFractals Theories and Applications*, 2009, pp. 190–194.

[23] S. Brown and C. J. Sreenan, “Updating software in wireless sensor networks: A survey,” in *Dept of Computer Science National Univ of Ireland Maynooth Tech Rep*, 2006.

[24] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava, “Sensor network software update management: a survey,” in *International Journal of Network Management*, 2005, vol. 15, no. 4, pp. 283–294.

[25] A. Liu, P. Ning, and C. Wang, “Lightweight Remote Image Management for Secure Code Dissemination in Wireless Sensor Networks,” in *IEEE INFOCOM 2009 The 28th Conference on Computer Communications*, 2009, pp. 1242–1250.

[26] A. Hagedorn, D. Starobinski, and A. Trachtenberg, “Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes,” in *2008 International Conference on Information Processing in Sensor Networks ipsn 2008*, 2008, vol. 00, pp. 457–466.

[27] C. H. Lim, “Secure Code Dissemination and Remote Image Management Using Short-Lived Signatures in WSNs,” in *IEEE Communications Letters*, 2011, vol. 15, no. 4, pp. 362–364.

[28] M. Rossi, N. Bui, G. Zanca, L. Stabellini, R. Crepaldi, and M. Zorzi, “SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes,” in *IEEE Transactions on Mobile Computing*, 2010, vol. 9, no. 12, pp. 1749–1765.

[29] M. D. Krasniewski, R. K. Panta, S. Bagchi, C.-L. Yang, and W. J. Chappell, “Energy-efficient on-demand reprogramming of large-scale sensor networks,” in *ACM Transactions on Sensor Networks TOSN*, 2008, vol. 4, no. 1, pp. 1–38.

[30] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A. F. Harris III, and M. Zorzi, “SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks Using Fountain Codes,” in *2008 5th Annual IEEE Communications Society Conference on Sensor Mesh and Ad Hoc Communications and Networks*, 2008, no. June 2009, pp. 188–196.

[31] N. Tsiftes, A. Dunkels, and T. Voigt, “Efficient Sensor Network Reprogramming through Compression of Executable Modules,” in *2008 5th Annual IEEE Communications Society Conference on Sensor Mesh and Ad Hoc Communications and Networks*, 2008, pp. 359–367.

[32] P. Levis and D. Culler, “Maté: a tiny virtual machine for sensor networks,” in *ASPLOSX Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002, vol. 36, no. 5, pp. 85–95.

[33] C. L. Fok, G. C. Roman, and C. Lu, “Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications,” in *25th IEEE International Conference on Distributed Computing Systems ICDCS05*, 2005, pp. 653–662.

[34] R. Müller, G. Alonso, and D. Kossmann, “SwissQM: Next Generation Data Processing in Sensor Networks,” in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research CIDR07*, 2007, pp. 1–9.

[35] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “TinyDB: an acquisitional query processing system for sensor networks,” in *ACM Transactions on Database Systems*, 2005, vol. 30, no. 1, pp. 122–173.

[36] Y. Yao and J. Gehrke, “The cougar approach to in-network query processing in sensor networks,” in *ACM SIGMOD Record*, 2002, vol. 31, no. 3, p. 9.

[37] C. C. Shen, C. Srisathapornphat, and C. Jaikaeo, “Sensor information networking architecture and applications,” in *Ieee Personal Communications*, 2001, vol. 8, no. 4, pp. 52–59.

[38] X. Yu, K. Niyogi, and S. Mehrotra, “Adaptive middleware for distributed sensor environments,” in *IEEE Distributed*, 2003.

[39] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy, and G. Picco, “Mobile data collection in sensor networks: The TinyLime middleware,” in *Pervasive and Mobile Computing*, 2005, vol. 1, no. 4, pp. 446–469.

[40] A. L. Murphy, G. P. Picco, and G. C. Roman, “LIME: a middleware for physical and logical mobility,” *Proceedings 21st International Conference on Distributed Computing Systems*, vol. 21, no. Apr 2001, pp. 524–533, 2001.

[41] T. Liu and M. Martonosi, “Impala: a middleware system for managing autonomic, parallel sensor systems,” in *System*, 2003, vol. 38, no. 10, pp. 107–118.

[42] Boulis, Han, Shea, and Srivastava, “SensorWare: Programming sensor networks beyond code update and querying,” in *Pervasive and Mobile Computing*, 2007, vol. 3, no. 4, pp. 386–412.

[43] D. Janakiram, R. Venkateswarlu, and S. Nitin, “COMiS : Component Oriented Middleware for Sensor Networks,” in *proceedings of 14th IEEE Workshop on Local Area and Metropolitan Networks LANMAN*, 2005.

[44] L. Szumel, J. Lebrun, J. D. Owens, and O. S. Ave, “TOWARDS A MOBILE AGENT FRAMEWORK FOR SENSOR NETWORKS,” in *Computer Engineering*, 2005, pp. 79–88.

[45] J. Schiller, A. Liers, H. Ritter, R. Winter, and T. Voigt, “ScatterWeb - Low Power Sensor Nodes and Energy Aware Routing,” in *System Sciences 2005 HICSS 05 Proceedings of the 38th Annual Hawaii International Conference on*, 2005, vol. 00, no. C, pp. 1–9.

[46] F. Oldewurtel, J. Riihijarvi, K. Rerkrai, and P. Mahonen, “The RUNES Architecture for Reconfigurable Embedded and Sensor Networks,” in *2009 Third International Conference on Sensor Technologies and Applications*, 2009, pp. 109–116.

[47] K. K. Khedo and R. K. Subramanian, “A Service-Oriented Component-Based Middleware Architecture for Wireless Sensor Networks,” in *Journal of Computer Science*, 2009, vol. 9, no. 3, pp. 174–182.

[48] P. Costa, L. Mottola, A. L. Murphy, and G. Pietro Picco, “TeenyLIME: transiently shared tuple space middleware for wireless sensor networks,” in *Proceedings of the international workshop on Middleware for sensor networks*, 2006, pp. 43–48.

[49] S. Tennina, M. Bouroche, P. Braga, R. Gomes, M. Alves, F. Mirza, V. Ciriello, G. Carrozza, P. Oliveira, and V. Cahill, “EMMON: A WSN System Architecture for Large Scale and Dense Real-Time Embedded Monitoring,” in *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*, 2011, pp. 150–157.

[50] D. Gelernter, “Generative communication in Linda,” *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[51] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, “Java on the bare metal of wireless sensor devices: the squawk Java virtual machine,” in *Memory*, 2006, pp. 78–88.

[52] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner, “Mires: a publish/subscribe middleware for sensor networks,” in *Personal and Ubiquitous Computing*, 2005, vol. 10, no. 1, pp. 37–44.

[53] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, “The abstract task graph: A methodology for architecture-independent programming of networked sensor systems,” in *Proceedings of the 2005 workshop on Endtoend senseandrespond systems applications and services*, 2005, no. Eesr 05, pp. 19–24.

[54] A. Rezgui and M. Eltoweissy, “Service-oriented sensor–actuator networks: Promises, challenges, and the road ahead,” in *Computer Communications*, 2007, vol. 30, no. 13, pp. 2627–2648.

[55] E. Cañete, J. Chen, M. Díaz, L. Llopis, and B. Rubio, “A Service-Oriented Middleware for Wireless Sensor and Actor Networks,” in *2009 Sixth International Conference on Information Technology New Generations*, 2009, vol. 25, no. 6, pp. 575–580.

[56] A. Murphy and W. Heinzelman, “Milan: Middleware Linking Applications and Networks,” in *Heart*, 2002, pp. 1–16.

[57] K. Aberer, M. Hauswirth, and A. Salehi, “The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks,” in *Network*, 2006, no. 5005.

[58] D. J. Abadi, Y. Ahmad, M. Balazinska, J. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, “The Design of the Borealis Stream Processing Engine,” in *Time*, 2005, pp. 277–289.

[59] P. B. Gibbons, B. Karp, S. Nath, and S. Seshan, “IrisNet: An architecture for a worldwide sensor web,” in *Ieee Pervasive Computing*, 2003, vol. 2, no. 4, pp. 22–33.

[60] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh, “Hourglass: An Infrastructure for Connecting Sensor Networks and Applications,” in *Harvard Technical Report TR2*, 2004, vol. 1, no. TR-21–04.

[61] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong, “Design considerations for high fan-in systems: The HiFi approach,” in *CIDR 2005 Proceedings of Second Biennial Conference on Innovative Data Systems Research*, 2005, vol. pages, pp. 290–304.

[62] L. Gurgen, C. Roncancio, C. Labbé, A. Bottaro, and V. Olive, “SStreaMWare: a service oriented middleware for heterogeneous sensor data management,” in *Management*, 2008, pp. 121–130.

[63] S. Ahn, “Building a bridge for heterogeneous sensor networks,” in *and Ubiquitous Systems, 2006 and the*, 2006, pp. 121–126.

[64] S. Rooney, D. Bauer, and P. Scotton, “Techniques for integrating sensors into the enterprise network,” in *IEEE Transactions on Network and Service Management*, 2006, vol. 3, no. 1, pp. 43–52.

[65] T. Kobialka, R. Buyya, C. Leckie, and R. Kotagiri, “A Sensor Web Middleware with Stateful Services for Heterogeneous Sensor Networks,” in *2007 3rd International Conference on Intelligent Sensors Sensor Networks and Information*, 2007, pp. 491–496.

[66] G. Mulligan, “The 6LoWPAN architecture,” in *Proceedings of the 4th workshop on Embedded networked sensors*, 2007, pp. 78–82.

[67] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, “Tiny web services: design and implementation of interoperable and evolvable sensor networks,” in *Computer*, 2008, no. Figure 1, pp. 253–266.

[68] D. Yazar and A. Dunkels, “Efficient application integration in IP-based sensor networks,” in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for EnergyEfficiency in Buildings BuildSys 09*, 2009, p. 43.

[69] S. Dawson-haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler, “sMAP: a simple measurement and actuation profile for physical information,” in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, 2010, pp. 197–210.

[70] M. Kovatsch, M. Weiss, and D. Guinard, “Embedding Internet Technology for Home Automation,” in *Proceedings of the 2010 IEEE Conference on Emerging Technologies and Factory Automation ETFA*, 2010, vol. 33, no. 3, pp. 463–72.

[71] S. Mayer, D. Guinard, and V. Trifa, “Facilitating the Integration and Interaction of Real-World Services for the Web of Things,” in *Intelligence*, 2010.

[72] D. Guinard, V. Trifa, and E. Wilde, “A Resource Oriented Architecture for the Web of Things,” in *Evolution*, 2010, pp. 1–8.

[73] A. P. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi, “Architecture and protocols for the Internet of Things: A case study,” in *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops PERCOM Workshops*, 2010, pp. 678–683.

[74] Z. Shelby, C. Bormann, and B. Frank, “Constrained Application Protocol (CoAP),” in *An online version is available at httpwww ietf orgiddraftietfcorecoap01 txt 0807 2010*, 2011, vol. 07, no. draft-ietf-core-coap-07.txt, pp. 1–81.

[75] W. Colitti and K. Steenhaut, “Integrating Wireless Sensor Networks with the Web,” in *Lossy Networks (IP+ SN 2011*, 2011, pp. 2–6.

[76] M. Kovatsch, S. Duquennoy, and A. Dunkels, “A Low-Power CoAP for Contiki,” in *2011 IEEE Eighth International Conference on Mobile AdHoc and Sensor Systems*, 2011, pp. 855–860.

[77] I. Aliance, “IEEE Std 802.15.4-2006,” in *IEEE Std 8021542006 Revision of IEEE Std 8021542003*, 2006, pp. 0_1–305.

[78] W. Ye, J. Heidemann, and D. Estrin, “An energy-efficient MAC protocol for wireless sensor networks,” in *ProceedingsTwentyFirst Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002, vol. 3, no. c, pp. 1567–1576.

[79] J. Polastre, J. Hill, and D. Culler, “Versatile low power media access for wireless sensor networks,” in *SenSys ’04 Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, vol. 1, pp. 95–107.

[80] A. El-Hoiydi and J. D. Decotignie, “WiseMAC: an ultra low power MAC protocol for the downlink of infrastructure wireless sensor networks,” in *Proceedings ISCC 2004 Ninth International Symposium on Computers And Communications IEEE Cat No04TH8769*, 2004, vol. 1, no. 28 June-1 July 2004, pp. 244–251.

[81] M. Buettner, G. V Yee, E. Anderson, and R. Han, “X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks,” in *Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006, vol. 76, no. May, pp. 307–320.

[82] A. Rowe, R. Mangharam, and R. Rajkumar, “RT-Link: A Time-Synchronized Link Protocol for Energy- Constrained Multi-hop Wireless Networks,” in *Sensor and Ad Hoc Communications and Networks, 2006. SECON ’06. 2006 3rd Annual IEEE Communications Society on*, 2006, vol. 2, no. C, pp. 402–411.

[83] Hart Communication Foundation, “WirelessHART Technical Data Sheet,” in *ReVision*, 2007, p. 5.

[84] K. S. J. Pister and L. Doherty, “TSMP: Time synchronized mesh protocol,” in *Networks*, 2008, vol. 635, no. Dsn, pp. 391–398.

[85] S. C. Ergen and P. Varaiya, “PEDAMACS: power efficient and delay aware medium access protocol for sensor networks,” in *IEEE Transactions on Mobile Computing*, 2006, vol. 5, no. 7, pp. 920–930.

[86] S. S. Kulkarni and M. U. Arumugam, “SS-TDMA: A self-stabilizing MAC for sensor networks,” in *Sensor network operations*, 2006, pp. 1–32.

[87] L. Bao and J. J. Garcia-Luna-Aceves, “A New Approach to Channel Access Scheduling for Ad Hoc Networks,” in *MobiCom ’01 Proceedings of the 7th annual international conference on Mobile computing and networking*, 2001, pp. 210–221.

[88] Y. E. Sagduy and A. ; Ephremides, “T HE P ROBLEM OF M EDIUM A CCESS C ONTROL IN W IRELESS S ENSOR N ETWORKS,” in *IEEE Wireless Communication Magazine*, 2004, vol. 11(6), no. December, pp. 44–53.

[89] C. Lu, B. M. Blum, T. F. Abdelzaher, J. A. Stankovic, and T. He, “RAP: a real-time communication architecture for large-scale wireless sensor networks,” in *Proceedings Eighth IEEE RealTime and Embedded Technology and Applications Symposium*, 2002, vol. 00, no. c, pp. 55–66.

[90] J. A. Stankovic and T. Abdelzaher, “SPEED: a stateless protocol for real-time communication in sensor networks,” in *23rd International Conference on Distributed Computing Systems 2003 Proceedings*, 2003, vol. 212, no. 4494, pp. 46–55.

[91] S. Munir, S. Lin, E. Hoque, S. M. S. Nirjon, J. A. Stankovic, and K. Whitehouse, “Addressing burstiness for reliable communication and latency bound generation in wireless sensor networks,” in *Proceedings of the 9th ACMIEEE International Conference on Information Processing in Sensor Networks IPSN 10*, 2010, no. May, p. 303.

[92] N. Gollan and J. Schmitt, “Energy-Efficient TDMA Design Under Real-Time Constraints in Wireless Sensor Networks,” in *In 15th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS’07). IEEE*, 2007.

[93] Y. Wu, X.-Y. Li, Y. Liu, and W. Lou, “Energy-Efficient Wake-Up Scheduling for Data Collection and Aggregation,” in *IEEE Transactions on Parallel and Distributed Systems*, 2010, vol. 21, no. 2, pp. 275–287.

[94] P. Suriyachai, U. Roedig, and A. Scott, “A Survey of MAC Protocols for Mission-Critical Applications in Wireless Sensor Networks,” in *IEEE Communications Surveys Tutorials*, 2011, no. 99, pp. 1–25.

[95] K. Kredoii and P. Mohapatra, “Medium access control in wireless sensor networks,” in *Computer Networks*, 2007, vol. 51, no. 4, pp. 961–994.

[96] N. Gollan, F. A. Zdarsky, I. Martinovic, and J. B. Schmitt, “The DISCO Network Calculator,” in *14th GIITG Conference on Measurement Modeling and Evaluation of Computer and Communication Systems MMB 2008*, 2008.

[97] K. Karenos and V. Kalogeraki, “Real-Time Traffic Management in Sensor Networks,” in *2006 27th IEEE International RealTime Systems Symposium RTSS06*, 2006, vol. 0, pp. 422–434.

[98] V. Vassiliou and C. Sergiou, “Performance Study of Node Placement for Congestion Control in Wireless Sensor Networks,” in *Workshop of International Conference on New Technologies, Mobility and Security*, 2009, pp. 3–10.

[99] J. Song, A. Mok, and D. Chen, “Improving pid control with unreliable communications,” in *ISA EXPO Technical Report*, 2006, no. October 2006, pp. 17–19.

[100] L. Zheng, “Industrial wireless sensor networks and standardizations: The trend of wireless sensor networks for process autometion,” in *SICE Annual Conference 2010 Proceedings of*, 2010, pp. 1187–1190.

[101] V. Lakkundi and J. Beran, “Wireless Sensor Network Prototype in Virtual Automation Networks,” in *The First IEEE International Workshop on Generation C Wireless Networks*, 2008.

[102] A. Willig, “An architecture for wireless extension of PROFIBUS,” in *IECON03 29th Annual Conference of the IEEE Industrial Electronics Society IEEE Cat No03CH37468*, 2003, vol. 3.

[103] S. Lee, K. C. Lee, M. H. Lee, and F. Harashima, “Integration of mobile vehicles for automated material handling using Profibus and IEEE 802.11 networks,” in *IEEE Transactions on Industrial Electronics*, 2002, vol. 49, no. 3.

[104] L. Rauchhaupt, “System and device architecture of a radio based fieldbus-the RFieldbus system,” in *4th IEEE International Workshop on Factory Communication Systems*, 2002.

[105] J. Haehniche and L. Rauchhaupt, “Radio communication in automation systems: the R-fieldbus approach,” in *2000 IEEE International Workshop on Factory Communication Systems Proceedings Cat No00TH8531*, 2000, pp. 319–326.

[106] D. Choi and D. Kim, “Wireless fieldbus for networked control systems using LR-WPAN,” in *Journal of Control Automation and Systems*, 2008, vol. 6.

[107] “HART Communication Protocol and Foundation - Home Page.” [Online]. Available: http://www.hartcomm.org/. [Accessed: 27-Aug-2013].

[108] “Wireless Industrial Networking Alliance.” [Online]. Available: http://www.wina.org/. [Accessed: 27-Aug-2013].

[109] “ISA100, Wireless Systems for Automation | ISA.” [Online]. Available: http://www.isa.org//MSTemplate.cfm?MicrositeID=1134&CommitteeID=6891. [Accessed: 27-Aug-2013].

[110] ZIGBEE, “ZigBee Alliance > Home.” [Online]. Available: http://www.zigbee.org/. [Accessed: 22-Feb-2013].

[111] T. Hasegawa, H. Hayashi, T. Kitai, and H. Sasajima, *Industrial wireless standardization Scope and implementation of ISA SP100 standard*. IEEE, 2011, pp. 2059–2064.

[112] P. Automation, “TECHNICAL WHITE PAPER PLANNING AND DEPLOYING,” in *Technical Report of Process Automation 2011*, 2011.

[113] H. Gao, X. Meng, and T. Chen, “Stabilization of Networked Control Systems With a New Delay Characterization,” in *IEEE Transactions on Automatic Control*, 2008, vol. 53, no. 9, pp. 2142–2148.

[114] K. Sato, H. Nakada, and Y. Sato, “Variable rate speech coding and network delay analysis for universal transport network,” in *IEEE INFOCOM 88Seventh Annual Joint Conference of the IEEE Computer and Communcations Societies Networks Evolution or Revolution*, 1988.

[115] J. Wu, F.-Q. Deng, and J.-G. Gao, “Modeling and stability of long random delay networked control systems,” in *2005 International Conference on Machine Learning and Cybernetics*, 2005, vol. 2, no. August, pp. 947–952 Vol. 2.

[116] E. Kamrani and M. H. Mehraban, “Modeling Internet Delay Dynamics Using System Identification,” in *2006 IEEE International Conference on Industrial Technology*, 2006, no. c, pp. 716–721.

[117] K. C. Lee, S. Lee, and M. H. Lee, “Worst Case Communication Delay of Real-Time Industrial Switched Ethernet With Multiple Levels,” in *IEEE Transactions on Industrial Electronics*, 2006, vol. 53, no. 5, pp. 1669–1676.

[118] E. Uhlemann and T. Nolte, “Scheduling relay nodes for reliable wireless real-time communications,” in *2009 IEEE Conference on Emerging Technologies Factory Automation*, 2009, pp. 1–3.

[119] N. Yigitbasi and F. Buzluca, “A control plane for prioritized real-time communications in wireless token ring networks,” in *2008 23rd International Symposium on Computer and Information Sciences*, 2008.

[120] I. H. Hou and P. Kumar, “Real-time communication over unreliable wireless links: a theory and its applications,” in *Ieee Wireless Communications*, 2012, vol. 19, no. 1, pp. 48–59.

[121] J. Schmitt, F. Zdarsky, and U. Roedig, “Sensor Network Calculus with Multiple Sinks,” in *Proceedings of IFIP NETWORKING 2006 Workshop on Performance Control in Wireless Sensor Networks*, 2006, pp. 6–13.

[122] U. Roedig, N. Gollan, and J. Schmitt, “Validating the Sensor Network Calculus by Simulations,” in *Network*, 2007.

[123] L. Lenzini, L. Martorini, E. Mingozzi, and G. Stea, “Tight end-to-end per-flow delay bounds in FIFO multiplexing sink-tree networks,” in *Performance Evaluation*, 2006, vol. 63, no. 9–10, pp. 956–987.

[124] A. Koubaa, M. Alves, and E. Tovar, “Modeling and Worst-Case Dimensioning of Cluster-Tree Wireless Sensor Networks,” in *2006 27th IEEE International RealTime Systems Symposium RTSS06*, 2006, no. October, pp. 412–421.

[125] D. Q. Systems, “NETWORK CALCULUS A Theory of Deterministic Queuing Systems for the Internet,” in *Online*, 2004, vol. 2050, pp. xix – 274.

[126] P. Jurcik, R. Severino, A. Koubaa, M. Alves, and E. Tovar, “Real-Time Communications Over Cluster-Tree Sensor Networks with Mobile Sink Behaviour,” in *2008 14th IEEE International Conference on Embedded and RealTime Computing Systems and Applications*, 2008, pp. 401–412.

[127] N. Xu, “A Survey of Sensor Network Applications,” in *Energy*, 2002, vol. 40, no. 8, pp. 1–9.

[128] R. Neves, S. Della Luna, D. Marandin, A. Timm-, and V. Gil, “Report on WSN applications, their requirements, application-specific WSN issues and evaluation metrics,” in *European IST NoE CRUISE deliverable*, 2006.

[129] V. Cantoni, L. Lombardi, and P. Lombardi, “Future scenarios of parallel computing: Distributed sensor networks,” in *Journal of Visual Languages Computing*, 2007, vol. 18, no. 5, pp. 484–491.

[130] R. Mac Ruairí, M. T. Keane, and G. Coleman, “A Wireless Sensor Network Application Requirements Taxonomy,” in *2008 Second International Conference on Sensor Technologies and Applications sensorcomm 2008*, 2008, no. 25–31 Aug. 2008, pp. 209–216.

[131] A.-B. García-Hernando, J.-F. Martínez-Ortega, J.-M. López-Navarro, A. Prayati, and A. P. and L. R.-L. Juan-Manuel López-Navarro, *Problem Solving for Wireless Sensor Networks*. London: Springer London, 2008, pp. 177–209.

[132] B. R. Conant, "Wireless sensor networks : Driving the New Industrial Revolution," in *Industrial Embedded Systems*, 2006, pp. 8–11.

[133] M. Antoniou, M. C. Boon, P. N. Green, P. R. Green, and T. A. York, "Wireless sensor networks for industrial processes," in *2009 IEEE Sensors Applications Symposium*, 2009, vol. 19, no. 6, pp. 13–18.

[134] M. Shanmugaraj, R. Prabakaran, and V. R. S. Dhulipala, "Industrial utilization of wireless sensor networks," in *2011 International Conference on Emerging Trends in Electrical and Computer Technology*, 2011, pp. 887–891.

[135] C. J. Sreenan, J. S. Silva, L. Wolf, R. Eiras, T. Voigt, U. Roedig, V. Vassiliou, and G. Hackenbroich, "Performance control in wireless sensor networks: the ginseng project - [Global communications news letter]," in *IEEE Communications Magazine*, 2009, vol. 47, no. 8, pp. 1–4.

[136] A. Klein, D. Agrawal, Z. Jerzak, P. Furtado, J. Cecílio, A. Cardoso, L. Silva, and J. do Ó, "Ginseng Report: Application scenario requirements and architecture specification," in *European GINSENG deliverable*, 2009, pp. 1–61.

[137] P. Padhy, K. Martinez, A. Riddoch, H. L. R. Ong, and J. K. Hart, "Glacial Environment Monitoring using Sensor Networks," in *RealWSN*, 2005, pp. 10–14.

[138] K. Yifan and J. Peng, "Development of Data Video Base Station in Water Environment Monitoring Oriented Wireless Sensor Networks," in *2008 International Conference on Embedded Software and Systems Symposia*, 2008, pp. 281–286.

[139] W. Zhengzhong, L. Zilin, L. Jun, and H. Xiaowei, "Wireless Sensor Networks for Living Environment Monitoring," in *Medical Engineering & Physics*, 2009, vol. 3, no. 11, pp. 22–25.

[140] P. J. Croft, F. Qi, P. Morreale, and A. Trzopek, "URBAN NET: URBAN ENVIRONMENT MONITORING AND MODELING WITH A WIRELESS SENSOR NETWORK," in *Observing and Assimilation Systems for the*, 2010.

[141] Y. Zhu, J. Song, and F. Dong, "Applications of wireless sensor network in the agriculture environment monitoring," in *Procedia Engineering*, 2011, vol. 16, pp. 608–614.

[142] F. Zhao and L. J. Guibas, *Wireless Sensor Networks: An Information Processing Approach*. Morgan Kaufmann, 2004, p. 376.

[143] V. Q. Son, B. L. Wenning, A. Timm-Giel, and C. Gorg, "A model of Wireless Sensor Networks using context-awareness in logistic applications," in *Intelligent Transport Systems TelecommunicationsITST2009 9th International Conference on*, 2009.

[144] J. Kenyeres, S. Sajban, P. Farkas, and M. Rakus, *Indoor experiment with WSN application*. IEEE, 2010, pp. 863–866.

[145] B. Liu, O. Dousse, J. Wang, and A. Saipulla, "Strong barrier coverage of wireless sensor networks," in *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing MobiHoc 08*, 2008, vol. 35, no. 8, pp. 411–420.

[146] B. P. L. Lo, S. Thiemjarus, R. King, and G.-Z. Yang, "BODY SENSOR NETWORK – A WIRELESS SENSOR PLATFORM FOR PERVASIVE HEALTHCARE MONITORING," in *Architectural Design*, 2005, vol. 13, no. 2–3, pp. 77–80.

[147] M. Blount, V. M. Batra, A. N. Capella, M. R. Ebling, W. F. Jerome, S. M. Martin, M. Nidd, M. R. Niemi, and S. P. Wright, "Remote health-care monitoring using Personal Care Connect," in *IBM Systems Journal*, 2007, vol. 46, no. 1, pp. 95–113.

[148] F. Zhou, H. Yang, J. M. R. Álamo, J. S. Wong, and C. K. Chang, "Mobile Personal Health Care System for Patients with Diabetes," in *Aging Friendly Technology for Health and Independence*, 2010, vol. 6159, pp. 94–101.

[149] I. M. Lopes, B. M. Silva, J. J. P. C. Rodrigues, J. Lloret, and M. L. Proenca, “A mobile health monitoring solution for weight control,” in *2011 International Conference on Wireless Communications and Signal Processing WCSP*, 2011, pp. 1–5.

[150] T. Gao, D. Greenspan, M. Welsh, R. Juang, and A. Alm, “Vital signs monitoring and patient tracking over a wireless network.,” in *Conference Proceedings of the International Conference of IEEE Engineering in Medicine and Biology Society*, 2005, vol. 1, no. September, pp. 102–105.

[151] E. Hughes, M. Masilela, P. Eddings, A. Raflq, C. Boanca, and R. Merrell, “VMote: A Wearable Wireless Health Monitoring System,” in *eHealth Networking Application and Services 2007 9th International Conference on*, 2007, pp. 330–331.

[152] S. A. Taylor and H. Sharif, “Wearable Patient Monitoring Application (ECG) using Wireless Sensor Networks,” in *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*, 2006, vol. 1, pp. 5977–5980.

[153] M. Al Ameen and K. Kwak, “Social Issues in Wireless Sensor Networks with Healthcare Perspective,” in *Security*, 2011, vol. 8, no. 1, pp. 52–58.

[154] T. I. S. C. May, “Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification,” in *Proceedings of the 12th ACM conference on Electronic commerce EC 11*, 1995, no. May, p. 29.

[155] J. Polastre, R. Szewczyk, and D. Culler, “Telos: enabling ultra-low power wireless research,” in *IPSN 2005 Fourth International Symposium on Information Processing in Sensor Networks 2005*, 2005, vol. 00, no. C, pp. 364–369.

[156] H. A. Nguyen, A. Forster, D. Puccinelli, and S. Giordano, “Sensor node lifetime: An experimental study,” in *2011 IEEE International Conference on Pervasive Computing and Communications Workshops PERCOM Workshops*, 2011, pp. 202–207.

[157] A. Dunkels, J. Eriksson, N. Finne, and N. Tsiftes, “Powertrace : Network-level Power Profiling for Low-power Wireless Networks Low-power Wireless,” in *SICS Technical Report T2011:05*, 2011.

[158] A. Dunkels, “The contikimac radio duty cycling protocol,” in *SICS Technical Report T2011:13*, 2011.