INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Multi-Functional Platform for Indoor and Outdoor Monitoring

Frederico Mendonça Passos de Lopes e Gonçalves

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Prof.ª Dr.ª Ana Maria Severino de Almeida e Paiva (DEI)
Orientador:	Prof.ª Dr.ª Teresa Maria Sá Ferreira Vazão Vasques (DEEC)
Co-Orientador:	Prof. Dr. Ricardo Jorge Feliciano Lopes Pereira (DEI)
Vogal:	Prof. Dr. Artur Miguel do Amaral Arsénio (DEI)

Outubro de 2011

Agradecimentos

Em primeiro lugar e em especial, gostaria de agradecer à Professora Doutora Teresa Vazão pelo seu apoio, orientação e motivação, ao longo da realização deste trabalho. Também gostaria de agradecer a orientação, o apoio e os conselhos do Professor Doutor Ricardo Pereira, sem também o qual a realização deste trabalho não seria possível.

Estendo também os meus agradecimentos ao grupo de redes do INESC-ID no pólo do Taguspark, incluindo professores e colegas, que sempre contribuíram com novas ideias e críticas construtivas ao longo da realização do trabalho.

Gostaria também de agradecer ao grupo do projecto MIAVITA, pelo seu trabalho e ajuda para testar a solução apresentada neste trabalho.

Gostaria ainda de agradecer aos meus colegas António Novais, Vasco Fernandes, João Trindade e António Fonseca pela sua amizade, apoio e incentivo ao longo deste trabalho.

À minha família, dedico esta dissertação e todo o esforço despendido ao longo do meu percurso académico.

Resumo e Palavras-chave

As redes de sensores sem fios têm revolucionado a industria e a ciência. Em particular estas redes desempenham um papel crucial na monitorização interior e exterior de eventos. Embora existam diversas soluções que resolvem múltiplos problemas encontrados no domínio da monitorização com redes de sensores sem fios, muitas provam ser demasiado específicas. Isto torna difícil a adaptação de aplicações para diversos tipos de monitorização. A presente dissertação de mestrado propõe uma plataforma única para redes de sensores destinadas à monitorização de eventos, que permita a fácil implementação de aplicações para este domínio e que seja o mais extensível e reconfigurável possível.

De modo a perceber quais as principais funcionalidades oferecidas por aplicações já existentes no domínio da monitorização, foi realizado um estudo sobre o estado da arte sobre as mesmas. Tal estudo revelou que a maior parte oferece sincronização temporal dos eventos e tem como base a poupança de recursos energéticos dos nós da rede. Assim, a plataforma desenvolvida incorpora dois protocolos. Um de sincronização de eventos denominado CLOWDE e um protocolo de agregação na rede. Este último destina-se a reduzir a congestão no meio sem fios, que origina corrupção dos pacotes e por sua vez leva os nós a retransmitirem os dados. Estas retransmissões têm um impacto significativo no consumo energético dos nós e podem ser evitadas com o protocolo de agregação desenvolvido. Ambos os protocolos podem ser usados ao mesmo tempo e são totalmente transparentes à aplicação, bastando configurar a plataforma de acordo com as necessidades da rede.

Os testes efectuados ao protótipo desenvolvido permitiram verificar o seu correcto funcionamento. O protótipo foi testado em dois projectos diferentes com objectivos diferentes. Um dos projectos, o projecto MIAVITA, tem como objectivo a monitorização de eventos vulcânicos. O segundo projecto, encontra-se integrado no projecto do MIT Portugal destinado à eficiência energética de edifícios. Ao testar o protótipo nestes dois ambientes é possível verificar a adaptação da plataforma a diferentes cenários de utilização.

Palavras-chave

Sistemas de monitorização, redes de sensores, sincronização, agregação de dados na rede.

Abstract and Keywords

Wireless sensor networks have revolutionized the industry and science. In particular, these networks play a crucial role in monitoring indoor and outdoor events. While there are several solutions that solve multiple problems encountered in the field of monitoring with wireless sensor networks, many prove to be too specific. This makes it difficult to adapt applications for various types of monitoring. This dissertation proposes a unique platform for sensor networks for monitoring events, allowing easy implementation of applications for this area and which is extensible and reconfigurable as much as possible.

In order to understand what are the main features offered by existing applications in the field of monitoring, a study was conducted on the state of the art on them. This study revealed that most offer synchronization of events and are based on saving energy resources of network nodes. Thus, the developed platform incorporates two protocols - a synchronization protocol named CLOWDE and an in-network aggregation protocol. The latter is intended to reduce congestion in the wireless environment, which leads to corruption of packets and in turn leads nodes to retransmit the data. These retransmissions have a significant impact on energy consumption and can be avoided with the developed aggregation protocol. Both protocols can be used simultaneously and are completely transparent to the application, simply requiring a configuration of the platform according to the needs of the network.

The prototype tests allowed to check its proper functioning. It was tested in two different projects with different goals. One project, the project MIAVITA, aims at monitoring of volcanic events. The second project is integrated into the design of the MIT Portugal for the energy efficiency of buildings. Testing the prototype in these two environments allows to check the adaptation of the platform to different usage scenarios.

Keywords

Monitoring systems, wireless sensor networks, synchronization, in-network aggregation.

Contents

List of Figures

List of Tables

Acronym list

PDU Protocol Data Unit

UDP User Datagram Protocol

TCP Transmission Control Protocol

IPC Inter Process Communication

TS Technologic Systems

OS Operating System

IP Internet Protocol

NF Netfilters

FIFO First In First Out

XML Extensible Markup Language

NACK Not Acknowledge

ACK Acknowledge

IACK Implicit Acknowledge

OSN Oldest Sequence Number

GPS Global Positioning System

OSI Open Systems Interconnection

IEEE Institute of Electrical and Electronics Engineers

RTS Request To Send

CTS Clear To Send

DIFS Distributed Coordination Function Interframe Space

SIFS Short Interframe Space

MTU Maximum Transmission Unit

ISA Instruction Set Architecture

DIO Digital Input Output

SPI Serial Peripheral Interface

ADC Analog-to-Digital Converter

SKB Socket Buffer

USB Universal Serial Bus

SD Secure Digital

ARM Acorn RISC Machine

CPU Central Processing Unit

WSN Wireless Sensor Network

AODV Ad Hoc On Demand Distance Vector

DADMA Data Aggregation and Dilution by Modulus Addressing

MAC Medium Access Control

NTP Network Time Protocol

RTT Round Trip Time

RBS Reference Broadcast Synchronization

FTSP Flooding Time Synchronization Protocol

RFID Radio Frequency Identification

UTC Coordinated Universal Time

MIAVITA MItigate and Assess risk from Volcanic Impact on Terrain and human Activities

PPS Pulse per Second

CDF Cumulative Distribution Function

CLOWDE Cross Layer One Way Delay Estimation

CBR Constant Bit Rate

1 Introduction

1.1 Background

In the last few decades, technological evolution has provided us with smaller equipments with higher computational resources and a broad range of wireless interfaces. Wireless Sensor Network (WSN) have become more powerful providing means to develop a broad range of applications. Indoor and outdoor monitoring systems are one of the many examples of applications using WSNs [1–5]. Typically each case gives origin to a different monitoring system, which contributes to increase the developing time and the overall cost. Also, it is difficult to adapt already developed applications to new scenarios. A paradigm that enables buildings with sensing capacities is pervasive computing. In this paradigm, spaces need to be filled with sensor networks which continuously monitor users in order to act accordingly with their needs [6]. Some pervasive solutions which could be adapted for indoor monitoring systems include the Odyssey [7] architecture, the Aura project [8] and the GaiaOS [9]. However, their main aim is to provide environments with pervasive mechanisms and are focused in specific pervasive issues, thus not being suitable for developing generic monitoring applications.

Outdoor monitoring has been in the WSNs application domain for several years and many solutions have been developed. FloodNet [10] and Vineyard [5] are two great examples of such solutions. Maté [11], EnviroTrack [4], Kairos [12] and Impala [13] are examples of platforms for WSNs that provide means to develop monitoring applications. However, these solutions suffer from the same issue as they are too specific and do not facilitate the developing of new applications for this kind of systems.

Adding to these facts the evolution of programming languages has eased the burden of developers by providing new paradigms and advanced data structures. This has provided the possibility of developing several customization tools for many WSNs applications. Therefore, the need for a common platform capable of supporting multiple applications and requirements for monitoring systems can now be satisfied. Such platform must be capable of supporting both indoor and outdoor monitoring requirements and ease the developing of new applications for such scenarios.

There are many issues which this platform must take into account. WSNs typically have throughput and reliability requirements, need to be extremely power-aware, must use appropriate routing protocols, synchronize sampled data and adapt to resource variations [14]. This thesis focuses in three key issues - synchronization, power-awareness and data transport. It proposes a synchronization protocol, which synchronizes data in one way by piggybacking time stamps in data packets. It also proposes a generic in-network aggregation protocol, which avoids congestion in the wireless medium avoiding energy consumption in the retransmission of packets. Finally as a proof of concept, it also proposes a reliable transport protocol based in negative acknowledgments, which avoids flooding the network with control messages.

1.2 Goal Statement

The goal of this thesis is to create a multi-functional platform, which can ease the development of monitoring applications and deal with network issues transparently. Specifically the work focus is to reduce energy

wasting due to packet loss and provide a generic synchronization mechanism, which can cooperate with the energy reduction technique.

1.3 Contributions

The present thesis contributes with a multi-function platform that can be used in different monitoring scenarios using wireless sensor networks. The proposed platform incorporates a synchronization and in-network aggregation protocol, which can be used simultaneous and were also developed in the context of this work. Synchronization is achieved without the use of acknowledgments nor broadcast beacons, thus being more power-aware than available solutions. The aggregation protocol was developed to be generic in order to be as transparent as possible to applications. This work was developed in parallel with another master's thesis by António Calçada Novais, which proposes and implements a service discovery application and a monitoring application. Both applications use the developed platform.

The proposed synchronization protocol gave origin to a paper with the title *"CLOWDE - Cross-Layer One-Way Delay Estimation"*, submitted to the European conference on Wireless Sensor Networks.

1.4 Document Structure

This document is organized in six chapters. The first one, where this section is integrated provides the project background and its motivation. Chapter 2 provides a study in the current state of the art of the project's application domain. Chapter 3 and 4 describe the proposed prototype and its implementation respectively. Chapters 5 tests and validates such prototype. Finally, chapter 6 presents the main conclusions of the developed work and future work that this thesis provides.

2 State of The Art

2.1 Monitoring Systems

This section presents the main functionalities of indoor and outdoor monitoring systems. It begins by providing a study in current issues in sensing in pervasive environments, which is a paradigm from where many solutions can be withdrawn to be applied in indoor monitoring systems. A similar study in environmental sensor networks is also presented.

2.1.1 Indoor Monitoring Systems

Nowadays with the advances in technology, pervasive computing has become a more tangible paradigm and pervasive environments begin to emerge. A pervasive environment is an environment saturated with devices with computational and communicational capacities. Some devices are so well integrated with it that they seem to disappear. An example of such environments may be an enclosed area such as a meeting room or corridor, or a well-defined open area such as a courtyard [15].

The first attempt to create a pervasive environment was done at Xerox Palo Alto Research Center described in [16]. The environment was composed of three kinds of devices, which were called tabs, pads and boards. Tabs resembled small post-it notes. Pads were made to resemble sheets of paper and boards were made to resemble blackboards.

Mark Weiser used these small devices in large intelligent environments. The Xerox ParcTab [17] provided limited display, user input, and infrared communication in a palm-sized unit. Small, attachable Radio Frequency Identification (RFID) tags helped identifying the approximated position and identity of tagged objects in spaces equipped with specialized readers. However, this was a vision too far ahead for that time and the research fell short, mainly because the required hardware did not exist.

Today, with the progress made in hardware many critical elements of pervasive computing are available as viable commercial products. Components such as handheld and wearable computers, wireless LANs and devices to sense and control appliances are now available to the common user [15].

One advantage of using these devices is the fact that it is possible to connect rich sensor arrays to them, creating ad-hoc wireless sensor networks. Sensor networks play an essential role in pervasive computing, as personal mobile devices interact with them in the environment [18]. A simple scenario described in [19], states that sensors can be used to capture vital signs from patients in real-time and relay data to handheld computer carried by medical staff in a pervasive computing environment.

Therefore, it is logical to assume that issues related to wireless sensor networks can be found in pervasive computing. Sensing in pervasive environments must be reliable, persistent, easy to interact with, and transparent [20]. However, sensing requires an elaborate networking infrastructure, able to support both high and low bandwidth data transmission as determined by context and sensor abilities. For example, sometimes video needs to be transmitted, while other times only extracted labels need to be transmitted.

Typically, sensor networks collect samples of data at a predefined sampling frequency. However, the high sampling rates of modern digital sensors are usually not needed in sensor networks. The power efficiency of such sensors are much more important [19].

Sensors have rigorous energy limitations, which make them more failure prone [19]. Sensor network protocols must focus on power conservation [21]. This issue is closely related to the availability of the sensor network, since the functioning of the network as a whole should not be endangered by failures of single nodes [22]. Therefore, reliability is crucial as opposed to ongoing maintenance.

Typically collected data is relayed to a base station, or sink node, but such technique causes non-uniform power consumption patterns and may overburden forwarding nodes [19]. This is even worse on nodes providing end links to base stations. These nodes may end up relaying traffic coming from all other nodes, thus forming a bottleneck for network throughput [23]. A technique known as clustering [24], helps in transmitting fewer packets. Moreover uniform energy consumption patterns may be achieved by re-clustering.

It is crucial that sensors nodes have trade-off mechanisms, which give users the option of prolonging network lifetime at the cost of lower throughput or higher transmission delay [21].

As already pointed out, sensor networks must stay available at all times. Therefore, these systems should be reliable, that is, fault tolerant and able to guarantee reliable point-to-point communication if needed [21]. Fault tolerance is the ability to sustain sensor network functionalities without any interruption due to sensor node failures [25]. Reliable point-to-point communication is the delivery of messages tolerating losses and tempering.

Finally, time synchronization is a big advantage in promoting cooperation among nodes, such as data fusion, channel access, coordination of sleep mode, or security-related interaction [19]. Often accurate time stamps are needed in order to organize each collected sample [26]. Due to power constrains and precision issue, standard time synchronization protocols, such as Network Time Protocol (NTP) [27], are not suitable for this type of networks [28, 29]. A solution in [30], presents an algorithm for smart environments to generate time stamps using unsynchronized clocks. Another solution called *post-facto synchronization* [31] is also based in unsynchronized clocks. Yet another approach is presented in [29] for time stamping sampled data. This issue is further addressed in section 2.2.1.

This concludes the analysis in monitoring techniques available in pervasive computing. The next section provides a study in environmental monitoring with sensor networks.

2.1.2 Outdoor Monitoring Systems

An environmental sensor network is a set of sensors with communication capability that provides ways for collected data to reach a given server [32]. There are systems which send messages to sensor nodes in order to obtain the collected data, while there are other systems which allow the sensor nodes to send the data autonomously to a server.

A study in [32] classifies environmental networks in three categories:

1. Large scale single function networks - Characterized by covering a large geographical area with expensive and large sensors. Normally measure one or more variable for a single purpose.
2. Localized multifunction sensor networks - Characterized by covering a small geographical area with small sensors. Normally measure sets of simple properties.
3. Biosensor networks - Characterized by using bio technology. These networks measure bio properties related to living beings in the environment.

This work is focused in the second type of networks, because they correspond to the characteristics already described and are better suited in the scope of this thesis. As with indoor monitoring systems the main properties that characterize these networks are explained.

Usually, environmental sensor networks are wireless ad hoc networks, which can be formed by different sensor nodes, or by clusters of sensor nodes. These clusters may have any kind of topology and routing policy, however there must exist a node that represents all others and interacts with the other representative nodes [32, 33].

Sensor nodes try to collect data about certain properties of the environment, therefore they must be totally integrated with it. The goal is to be unobtrusive to avoid disturbing the natural environment and collecting false data [34]. This kind of invisibility can be compared with pervasive computing invisibility, because both have a similar goal - blend into the environment. However, in this type of environment, technology does not need to be hidden from possible environment users, it just needs to be unobtrusive.

Typically, sensor nodes are placed in sites where there is no cabled power supply. They must use alternate energy resources, such as batteries, or solar panels. Systems must manage this energy efficiently in order to prevent human intervention for a given amount of time [34]. Nevertheless, in some environments sensor nodes might be mobile. In fact, they might require location systems for being too mobile [32]. One important feature is to be able to tell from which geographical position the data has been sent and which sensor node sent it.

As opposed to pervasive computing this mobility has to due only with device mobility. Remote management is also required, so the system can be accessed from other geographic points and reduce the need for human intervention [33]. This enables deploying, fixing and removing software from each node.

As devices move around the environment or even when computational resources are scarce, adaptation may help to prevent wasting these resources [32]. Adaptation is a key aspect in prolonging the network's life time, as it compensates for lack of resources such as network bandwidth and power.

Depending on the type of data being collected, security might be required as a system feature. For example, habitat monitoring does not typically require secure connections [34]. However, as stated in [33] some systems do require secure connections for data transmission.

This section has presented the major properties for an environmental sensor network, which aims at outdoor monitoring. Combining the information here described and in section 2.1 it becomes clear that several features must be integrated in a platform, which aims at providing support for both kind of monitoring systems. This work is focused in synchronization, data aggregation and data transport. Therefore, the following sections deepen these three features by analyzing specific protocols for each one.

2.2 Important Functionalities in WSNs

This section provides a more detailed study in the state of the art in synchronization protocols, in-network aggregation techniques and data transport techniques for WSNs.

2.2.1 Synchronization Protocols for Wireless Sensor Networks

Time synchronization has been studied and applied in distributed systems for several years. Solutions such as NTP [27] are well adapted to such systems. However, they are not suitable for WSNs [28]. Even logical time [35] is not enough for WSNs, because they only preserve causality between events. The authors in [28] also point the fact that the 802.11 standard has synchronization beacons built in the Medium Access Control (MAC) sub-layer. However, these beacons do not work beyond a single broadcast domain, thus are not suitable for WSNs. Therefore, intensive work has been done to provide new approaches for synchronizations in this area. Each solution presents its own trade-offs - Unnecessary synchronization wastes resources, while insufficient synchronization degrades application performance. This eventually leads to the fact that synchronization must be a tunable feature.

Post-facto synchronization [31] was developed for low-powered WSNs. The main idea behind this kind of synchronization is to synchronize events after they actually happen. When an event occurs, each node that sensed it takes a time stamp using their local clocks. After this, a node which did not sense the event will broadcast a synchronization beacon. Nodes which receive this beacon use it to normalize their time stamps. The authors admit that *post-facto* synchronization is not suited for long distances because of clock drift. They suggest using NTP to synchronize clock frequency in order to avoid clock drift and skew.

Another approach that can be related to WSNs is a protocol specially designed for mobile ad-hoc networks [30]. The authors admit that nodes are able to keep a connection between them enough time to transmit at least two messages. These messages are an event notification done by the sender and an Acknowledge (ACK) sent by the receiver. More specifically the sender takes a time stamp and sends it to its neighbor. The message is acknowledged by its receiver with the same time stamp. Then, the sender will use this time stamp to estimate the Round Trip Time (RTT) and piggyback it in the next message. This process continues until eventually the message will arrive at its destination. All the RTTs and time stamps can be used to estimate a time stamp for such message, according to the destination's local clock. It is important to note that the time stamp is composed of two values - one for its lower-bound and one for its upper-bound. The problem with this approach is that it requires an acknowledgement for each message. While this approach may not be an issue for mobile ad-hoc networks, in WSNs most applications continuously send data across the network. If each message would be acknowledged the network would be flooded and the probability of collisions would increase. As shown in section 2.2.2 this wastes valuable resources.

Reference Broadcast Synchronization (RBS) is yet another synchronization technique described in [36]. In this protocol, nodes periodically send beacons using the physical layer to their vicinity. Other nodes use these beacons to synchronize themselves. The algorithm is based in the principle that receivers in the coverage area of the sender will receive these beacons approximately at the same time. Thus, receivers would set their clocks as soon as a beacon arrives. However, in wireless mediums packets can be lost or corrupt and receivers are not able to promptly record the beacon's time. To minimize this issue, a set of synchronization beacons is transmitted instead of only one and used to estimate clock offset and skew. The authors recognize that this approach has a limitation - it requires the network to have a physical broadcast channel. Also, as indicated in section 2.2.2 periodically sending data to the network can lead to collisions in the wireless medium and waste node's energy resources.

The main issue with RBS is that it requires several messages exchanged in order to communicate local time stamps between nodes. Flooding Time Synchronization Protocol (FTSP) is an approach to synchronization which avoids this issue [37]. FTSP synchronizes the sender's local clock using only one message time stamped at the sender and receiver. It uses linear regression to compensate for clock drift. However, field tests in [38] show that FTSP is not sufficiently stable. A different protocol known as Z-sync [38] was designed to compensate for FTSP's instability. It uses an hybrid implementation with Global Positioning System (GPS) attached to some nodes. Nodes which do not have a GPS synchronize themselves with the ones who do. However, this approach can become to expensive if the network becomes large enough and several nodes need to incorporate GPS devices.

Finally, an approach described in [39] uses one way synchronization to time stamp each sample sent by each node. The idea is to gather the time each sample spent inside each node and use this time to subtract it to the destination's local clock. Thus, samples are time stamped according to the destination's local clock and do not require acknowledges from receivers. The authors claim that the overall propagation time is in the order of nanoseconds and do not use it in the time stamping equation. However, depending on the Operating System (OS) and hardware used it may be complicated to measure processing, queuing and transmission delays making this solution difficult to port.

In sum, several approaches exist for time synchronization in WSNs. Broadcast and two way synchronization approaches can lead to collisions in the wireless medium and waste needed resources. One way synchronization is another approach, however it is important to account for every delay in order to obtain a good estimation for the messages' time stamp.

2.2.2 Data Aggregation Techniques

Aggregation emerges as one of the most studied techniques to reduce energy consumption in sensor networks [40]. Data aggregation at the network level is known as in-network aggregation. Specifically it is defined as being the process of gathering and routing data through the network, while processing it at intermediate nodes. The authors in [40] define two types of in-network aggregation:

- In-network aggregation with size reduction

  As implied by its name, this kind of aggregation reduces the payload size. Typically this is done by collecting several data values to aggregate, compute the mean value and send said value instead of a packet with all values. It is important to notice that this kind of aggregation depends highly in the type of data being sent, thus it also depends on the application of the sensor network.
- In-network aggregation without size reduction

  In this kind of aggregation, data cannot be reduced. This usually has to due with the fact that nodes may measure orthogonal properties. As an example, consider a node which receives data from two sensors. One measures temperature and the other measures wind velocity. These two properties are not related and therefore it is not possible to compute a mean value for both. For this reason aggregate packets will always contain both readings.

The authors in [40] also identify four common paradigms for aggregation techniques. Aggregation techniques may compress the resulting aggregate packets. If it is possible to recover all the original data from a compressed packet, the aggregation paradigm is said to be *lossless*. However in the other hand, if it is not possible to recover all the original data, the aggregation paradigm is said to be *lossy*. The paradigm may also be *duplicate sensitive* or *duplicate insensitive*. In sensor networks, nodes may receive copies of the same data from different nodes. If the aggregation protocol is *duplicate sensitive* the final result will depend on the number of times the same data has been considered. Otherwise it is said to be *duplicate insensitive*.

Several solutions have been developed which explore in-network aggregation. The remainder of this section presents the most common ones and analyzes each one. TAG [41] is a generic aggregation service for low-power wireless sensor networks. It is inspired in database query languages and is prepared to distribute aggregation queries in a power-efficient way. It works in two phases. First phase distributes queries among sensor nodes. The second phase routes sensor readings through the routing tree. This routing tree is called the aggregation tree and the routing protocol is optimized for a better aggregation. As data is routed through the network, TAG will discard irrelevant readings and combine relevant ones.

Cougar [42] is another approach to aggregation using a declarative query language. It is partially suitable to perform *duplicate sensitive* aggregation. This has to due with the fact that Cougar uses Ad Hoc On Demand Distance Vector (AODV), which does not duplicate packets. Cougar has some similarities with TAG. Both are most suitable for monitoring applications and the way aggregation is perform is highly dependable on the routing protocol. Although this means that the degree of aggregation can usually be greater than a generic approach, it also means that these approaches are highly dependable on the sensor network application.

TiNA [43] is yet another approach to in-network aggregation, which uses a query language. It can work on top of TAG and Cougar. It uses temporal correlation between sensor readings and discards readings that do not affect the quality of the resulting aggregate packet. This makes TiNA dependable on the kind of data sensors are collecting. Also, because it uses an approach based in routing trees it depends on the routing protocol to maximize the aggregation technique.

Another aggregation technique known as Data Aggregation and Dilution by Modulus Addressing (DADMA), which uses a query language is presented in [44]. This approach treats the sensor network as a distributed relational database. A single view of this database is created by joining records stored locally at each node. Such view is created and maintained at the sink node. The idea behind DADMA is to provide means to aggregate data from a group of sensors. This means that data from some sensors is excluded from the aggregate packet. Such data is said to be diluted. DADMA also depends on the routing algorithm in use and because it uses a dilution technique the user must decide which data to retrieve.

The presented aggregation techniques suffer from a common problem, which is application dependence. Although, they achieve a high aggregation degree it is very difficult to adapt them to other scenarios. A more generic solution known as AIDA [45] was developed to be application independent. AIDA is application independent in the sense that the aggregation logic is implemented in the MAC sub-layer of the Data Link layer. This way it can aggregate packets based only in their headers and does not need to process data sent from applications. However, using this approach has a disadvantage. Because aggregation happens at the MAC sublayer, packets have already been routed. This means that aggregate packets must be desegregated

at each hop to be rerouted and aggregated again. Although this approach permits aggregation of packets with different destinations, the authors recognize that in the majority of cases packets will have the same destination. This has to due with the fact that most sensor networks collect data and send it to a sink node, as shown in section 2.2.3.

In sum, aggregation techniques have a common goal which is reducing the amount of data transfered in a sensor network. Previous work has shown that such techniques are also able to reduce the energy consumption of each node. However, most solutions are application dependable and difficult to adapt to generic scenarios.

2.2.3 Transport in Wireless Sensor Networks

In WSNs, primary traffic flows in the upstream direction from the nodes to the sink [46]. Applications normally collect data and send it at a predefined rate, as explained in section 2.1. Some applications do not need reliability when sending this data. However, some do require such property. The problem in WSNs is that the percentage of packet loss is much higher than in cabled networks, thus making reliability more difficult to implement. Adding to this fact, near the sink node congestion might become an issue if applications send data at high rates [47]. Therefore, reliable transport protocols in WSNs have two major goals, which are to reduce packet loss and congestion. Moreover, most solutions try to consume as less energy as possible.

This is the main reason why Transmission Control Protocol (TCP) is not suitable for WSNs [48]. TCP flow and congestion control are not suitable to answer packet loss, because they waste sensor energy. Other problems associated with TCP include overhead associated with connection establishment, degraded throughput in wireless systems and it guarantees successful transmission of every packet when sometimes only a subset of packets is required to achieve the needed reliability [46].

Reliability in WSNs is normally classified in two types. Packet reliability is ensured when every packet transmitted by applications is successfully delivered to their destinations. Event reliability [49] happens when applications do not require all packets to be delivered successfully, but require reliable event detection. In order to avoid packet loss, mechanisms for loss detection and recovery have been studied and evaluated.

The typical method for loss detection is the use of sequence numbers. Packets are marked sequentially and in the event that some get lost it is possible to detect gaps in the received sequence numbers [50]. This scheme is a receiver-based loss detection mechanism. It is also possible to have loss detection implemented as a sender-based mechanism. In this scheme, packet loss is typically detected through timeouts or overhearing mechanisms. In the first, the sender sets up a timeout after which it will resend some packets. In the second, the sender sends a packet and then listens the medium for that packet retransmission. If no retransmission occurs, it will resend that packet. Figure 2.1 explains this mechanism. Node A sends a packet to node C. After sending it, node A will listen for B's retransmission to node C. If B does not retransmit A's packet it means that the packet was lost or corrupted. This approach requires that nodes be able to listen the medium for retransmissions, which is not always possible.

Loss detection can further be classified as end-to-end or hop-by-hop. In an end-to-end loss detection, only sender and receiver may detect packet loss, therefore overhearing mechanisms are not part of end-to-end approaches. In a hop-by-hop loss detection, every node through which the packet passes plays a role in

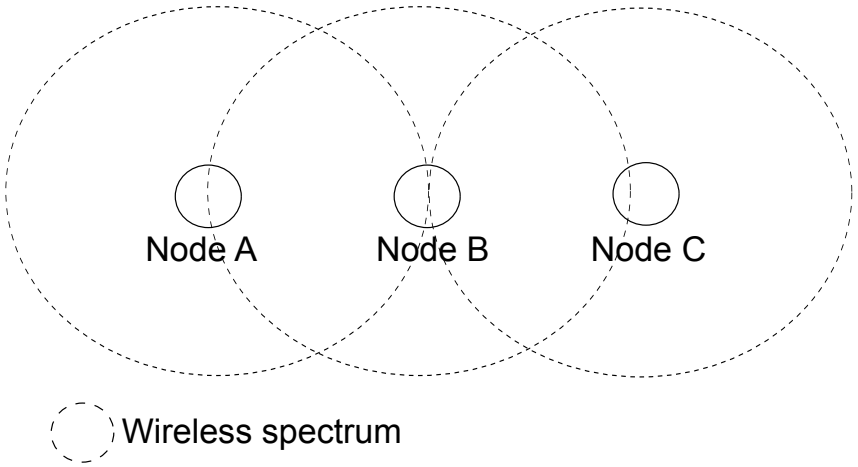


Figure 2.1: Implicit Acknowledgement scheme example. Node A sends data to node C, which will be routed through node B. Notice that node A is not able to directly communicate with node C.

detecting its loss.

Loss recovery can also happen on end-to-end basis or hop-by-hop basis. If the loss detection is receiver-based, then the sender needs to be notified to proceed to loss recovery. Three main schemes exist to notify the sender. ACK schemes or positive acknowledgements are used to notify the sender of how many packets the receiver has already received. Not Acknowledge (NACK) schemes or negative acknowledgements are used to notify the sender of which packets the receiver has not received. In Implicit Acknowledge (IACK) schemes acknowledgements are piggybacked in messages. The scenario described in figure 2.1 is an example of an IACK scheme.

ACK schemes are worse than NACK schemes in WSNs, because the rate of success delivery is greater than the rate of packet loss [51]. Therefore, in an ACK scheme the network would be flooded with ACK messages, whereas in a NACK scheme only from time to time would a NACK message be generated. In contrast, IACK schemes would be the best choice as they do not create extra control messages.

End-to-end approaches require that ACK and NACK messages to traverse the entire route, which will waste node energy [46]. In contrast, hop-by-hop approaches do not require control messages to traverse the entire route, only a subset of nodes. However, such approaches have some disadvantages. Intermediate nodes have to cache packets and if some fail, packet delivery is not guaranteed. Nevertheless, if packet reliability is required then a hop-by-hop approach should be used. If event reliability is required then an end-to-end approach will suffice.

In sum, reliable transport in WSNs should aim to reduce energy consumption and congestion. An IACK scheme with an hop-by-hop approach is the best mechanism to achieve this, but not always possible. Nodes may not be able to hear the medium for retransmissions, or the network may not be fault tolerant (applications may require event reliability instead of packet reliability). Therefore, several solutions use NACK messages with an end-to-end approach.

3 System Architecture

3.1 MIAVITA and MIT Project Architectures

The aim of this thesis is to develop a platform capable of working in different scenarios. To validate the adaptability of such platform, two projects with completely different requirements have been chosen to test it. The MItigate and Assess risk from Volcanic Impact on Terrain and human Activities (MIAVITA) project is an outdoor monitoring system, which continuously monitors volcanic tremor to early detect a volcanic eruption. The MIT people counting project is an indoor monitoring system that gathers events related to people movement and location.

Although these two projects have different requirements and goals, it is possible to define a multi-function platform which will work on both. The next sections overviews these two projects.

3.1.1 MIAVITA

The MIAVITA project main goal is to make the risk assessment and management of volcanic activity. One part of the project is to build a sensor network to measure the seismic activity and early detect volcanic eruptions, using cheap and of the shelf hardware. This network will be placed in several sites within each volcano, which typically are of difficult access to people. A typical monitoring scenario would not last more than two days. Therefore, nodes do not need to be as energy efficient as typical sensors in wireless sensor networks and may be powered by battery or solar panel. Also, nodes are not required to be small in size.

The MIAVITA's wireless sensor network monitor the ground displacement and sample it at a maximum of 50 Hertz, with samples of 24 bits. The seismic wave will be reconstructed based on these samples. A precise reconstruction of the original seismic wave is very important for the geophysics community and so time synchronization, geo-positioning and reliability are needed.

Figure 3.1 shows Mi-Vita's sensor network topology. According to the figure, there are three kinds of nodes - collector nodes, sink nodes and backup nodes. Communication between each node is done using the Institute of Electrical and Electronics Engineers (IEEE) 802.11b standard.

Collector nodes are connected to a geophone, which is a sensor capable of reading ground displacement. These nodes sample data at a given rate and send it across the network to a sink node. The sink node is an unique node in the network, which sends the received data to a base station located several kilometers from the sensor network. Thus, the sink node must be equipped with another network interface to support this remote communication. The sink is the only node in the network that requires a GPS. Backup nodes exist to compensate for collector node faults. One key aspect of this network is that although each node has a specific function, everyone is connected to a geophone and samples data at a given rate. Therefore in this sense every node is a collector node.

3.1.2 MIT People Counting and Detection of Patterns of Movement

The idea behind the MIT project for detection of people movement patterns is to reduce the energy consumption in indoor environments. If it is possible to determine the zones most used in a building, then it is also

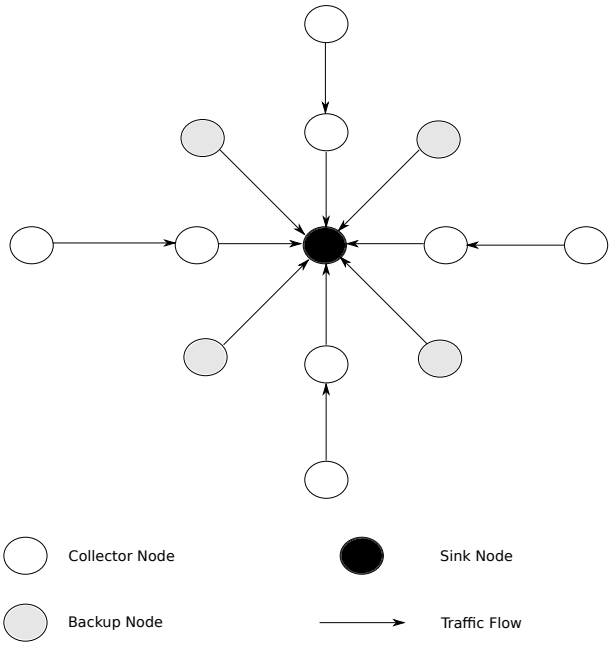


Figure 3.1: MIAVITA network topology. Nodes send data to a unique node called a sink node. Backup nodes will compensate for other node faults.

possible to reduce energy consumption in less-used zones. The project's approach is to enrich each zone with several sensors that are able to determine if it is being used or not and if so how many persons are using it. Then, the gathered information must be correlated to determine movement patterns and improve the system's intelligence.

Correlating information is achieved through a sensor network, with an architecture specified in figure 3.2. Each building is separated in zones, which may overlap in the sense that a zone *A* may contain two zones *B* and *C*. Each zone contains several spotter nodes and is managed by a zone manager. Each zone manager will communicate with the aggregator node.

Spotter nodes are directly connected to different sensors such as bluetooth and infrared devices. These

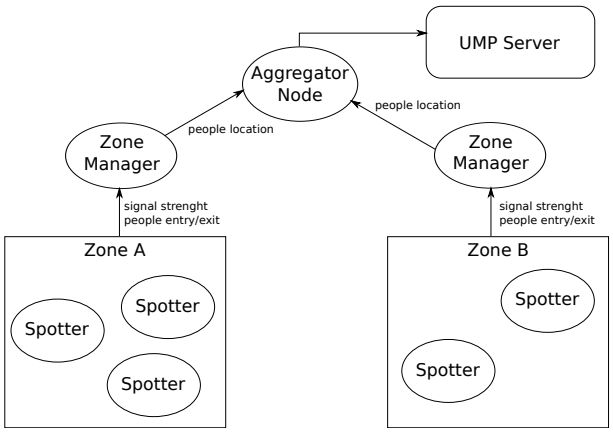


Figure 3.2: MIT people counting project architecture. Spotter nodes send data to zone manager, which in turn will send data to the aggregator node. This last one will send data to an UMP server.

nodes collect simple information such as how many persons have entered each zone and signal strength of their mobile devices. This information will be delivered to the zone manager, which will triangulate each person's location. The triangulated locations are sent to the aggregator node, which will correlate information to eliminate inconsistent results.

As in MIAVITA's project, nodes may have a considerable size and communicate using the IEEE 802.11b standard. However, here monitoring must last for long periods of time in order to obtain good results. Moreover, since this is indoor monitoring nodes can be powered by the building's energy system, providing more durability.

This project does not require a synchronization as precise as MIAVITA's sensor network, because time stamps are used only to solve conflicts when correlating information. These conflicts may arise when spotters tell zone managers that a given person is in two different places at the same time. This person is expected to pass through other spotter nodes and when both information is correlated it is possible to remove information that does not make sense, using the samplers' time stamp.

Since spotters collect information each time they sense a person, this monitoring is considered event monitoring. To ensure that all kind of nodes in the architecture are available, this project incorporates a service discovery component. Also, because the project deals with sensible information, such as people location, security is a major issue to consider. In fact, privacy must be ensured in order to avoid tracking people in inappropriate places.

3.2 Requirements Specification

The previous section describes two distinct projects - MIAVITA's and MIT's project - that will be used as a proof of concept of our multi-function platform. These projects are targeted to different application scenarios, as the first one relates to real-time outdoor monitoring and the second one to event-driven indoor monitoring. In spite of their differences, both of them share some important requirements, which can be used as the baseline for the design of our communication platform. The next section presents these requirements.

Modularity

The proposed solution should be usable in different scenarios. However, if a monolithic solution is developed every feature will be joined into a single implementation. This may cause an undesirable overhead in applications that do not need to use every feature supported by the platform. Therefore, the provided solution must be developed in a modular way. This also enforces the platform's extensibility, because it facilitates the construction of new modules.

Configurability

Each monitoring application has its own particularities. For example, some might require data to be synchronized with high precision, whereas others might require a more relaxed synchronization. Therefore, it is important that the developed architecture provides means to configure its different modules to be adapted to different scenarios.

Adaptability

The proposed architecture is intended to work in different scenarios. In order to provide this its essential

to have support for adaptability. For example, some techniques which reduce congestion in wireless medium require delayed packet transmission. However, some applications in WSNs may require to send data as soon as possible. Moreover, some may require to do this only at specific times. For such applications it is crucial to provide means to force packet transmission.

Security

The platform is meant to work with different applications. Some applications may require a certain degree of security, such as ciphered communication and encrypted storage. Therefore, the platform should incorporate mechanisms to support these different levels of security.

Service discovery

Each node in the platform may have different roles and may offer different services to the other nodes. Announcing and discovering these services is essential to promote the cooperation among the nodes.

Different Types of Transport

Some scenarios require reliable data delivery, while others may use unreliable delivery to reduce energy consumption. It is important to have both options, giving the possibility of choosing between them. Moreover, it should be possible to configure the platform to work with reliable and unreliable delivery per application. This will enable a single node to have different applications using different data delivery policies.

Synchronization

It should be possible to synchronize data transparently, that is, without applications knowing the synchronization algorithm used and without using external devices, such as GPS devices. Moreover, the synchronization algorithm should not introduce too much overhead and should offer the possibility of time stamping messages with a global time stamp. As for global time stamps, some applications may require time stamps to be in Coordinated Universal Time (UTC) format.

Traffic aggregation

Since communication is often done using wireless technology to guarantee an adequate level of performance it is important to reduce the network traffic, so that congestion in the wireless medium that leads to data loss can be reduced. By doing so, less retransmissions are required and nodes reduce their energy consumption. It is also important to provide this feature without having to force applications to implement it. Therefore, as with synchronization this feature should be as transparent as possible.

Different Routing Protocols

Routing is yet another theme tightly related to sensor networks. Most applications require specific routing protocols. Therefore it is important to make the platform as independent from the routing protocol as possible. This will ensure support for multiple routing protocols.

The following sections will describe how each requirement is satisfied by the proposed solution.

3.2.1 Requirement Implementation

From the set of requirements previously described, we select the ones that most fit the immediate needs of the two projects. Thus, the proposed solution addresses two of the most generic requirements described before: modularity and configurability. In spite of the importance of adaptability and security they were left for

further study, as they were considered major topic of research. From the remaining set of requirements we decided to focuses our efforts on three of them: support of different transport protocols, synchronization and traffic aggregation. The remaining requirements will be addressed by other members of the team.

A modular architecture is a key feature to make the platform portable for different WSNs applications. It is important to separate functionalities across several modules. For example, synchronization should not be performed by the same module which performs configuration. This allows to turn off synchronization in scenarios which do not require it, but will always require configuration.

Configurability is addressed from two different scopes. First, the configuration module performs a generic system configuration. Properties such as which modules are loaded in the current node are configured. The second scope is performed within each module individually, defining what characteristics need to be supported.

The type of transport required is application dependent. It is difficult to address this issue with only one module. Therefore, the proposed solution addresses this requirement as follows. A generic module, capable of supporting different transport protocols is present in every node in the WSN. Applications can create instances of this module with a specific transport protocol. These instances are denominated *handlers*.

Synchronization and network traffic reduction are all implemented using packet manipulation techniques. To allow extra flexibility, these requirements are implement in two different submodules. These submodules must be controlled by a master module which will intercept packets sent by applications and pass them to each submodule. These in turn will perform any manipulation necessary. These submodules are denominated *interceptors*.

3.3 Architecture Specification

3.3.1 Architecture Overview

In order to satisfy each requirement identified in section 3.2 the proposed architecture was designed to be highly modular and configurable. However, before delving with detail into each module of the proposed architecture, it is necessary to provide an overview of each one. Figure 3.3 shows such overview.

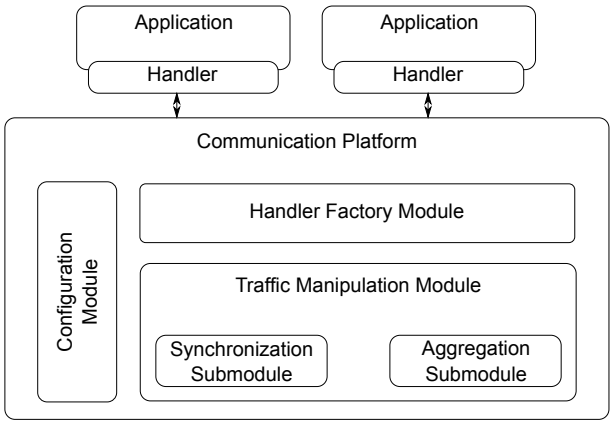


Figure 3.3: Overview of each module present in the proposed architecture.

The communication platform incorporates three different modules. A configuration module, which configures the system and the modules. A generic handler factory, which is responsible for creating instances of handlers with specific transport protocols. A traffic manipulation module, which incorporates two submodules. One submodule performs synchronization and the other aggregation of packets sent by applications. By adding new submodules it is possible to implement new traffic manipulation techniques. Applications use handler instances to communicate with the platform and request data to be sent.

In order to better understand the concept, figure 3.4 shows a more detailed overview of the proposed platform. It highlights how modules interact with each other and how handler instances communicate with the platform. The generic module which creates handler instances is omitted for simplicity. Also, the traffic manipulation module is generalized to demonstrate that it supports more than just synchronization and aggregation functionalities.

Handlers communicate with other handlers by means of messages with a well defined format. They are also responsible for applying the necessary transport control to such messages. This control is necessary for example when reliable transport is required between two applications. The designed solution is capable of incorporating several types of control. It is possible to use a reliable control with flow and congestion control like TCP. A reliable control without flow and congestion control based on User Datagram Protocol (UDP) with packet loss detection and recovery. It is also possible to use unreliable protocol, like UDP, which in fact does not perform any kind of control.

Each application may have more than one handler instantiated that abstract the application from the underlying architecture. This means that it is possible for an application to communicate locally and remotely, regardless the type of transport or traffic manipulation techniques in use. For this reason applications need only to identify the destination handler for which they are going to send specific data. The source handler will resolve the destination handler identifier into an address, which specifies if the destination handler is located on the same node or in another one. This is achieved by communicating with a local registry, designed specifically for this purpose.

If the destination handler is located in the same node, then messages can be sent without the need for applying traffic manipulation. However, if the destination handler is located in another node, traffic manipulation may be required. This is accomplished by the traffic interceptors, also illustrated in figure 3.4. The idea of traffic interception is to provide functionalities such as synchronization, aggregation and even security, transparently to applications in the network. In order to be correctly performed, interception must be done at a specific point. For example, if an application requires traffic to be encrypted, then a security interceptor must have a way of distinguish outgoing traffic from other traffic. Likewise, a security interceptor on the destination node must be able to distinguish between incoming traffic from other traffic, in order to be able to perform decryption. This is the reason for interceptors to be located in the network layer of each node, where they are able to intercept incoming traffic, outgoing traffic and forwarded traffic.

A good outcome of such design is the decoupling between traffic interceptors and handlers, which makes applications and the type of transport used independent of traffic manipulation. A clear advantage of such approach is that any node in the network may perform traffic manipulation without having handlers instantiated. This is especially important in aggregation techniques, where a node simply needs to aggregate forwarded

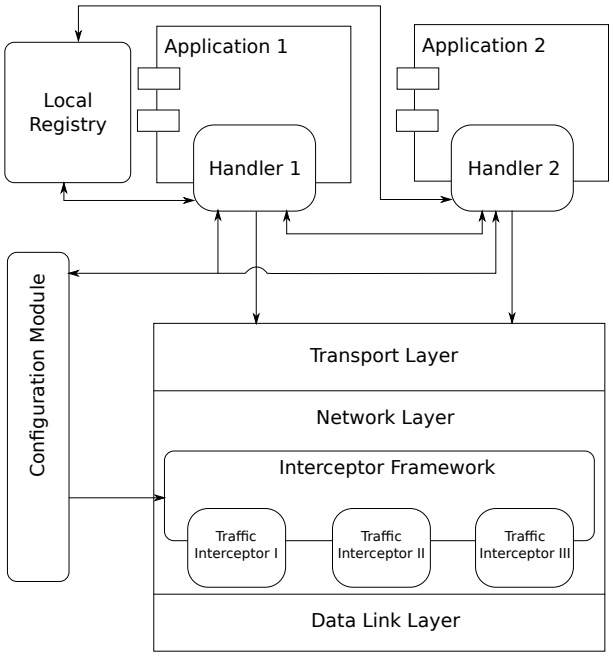


Figure 3.4: Architecture overview of the proposed platform. Each node in the network has a configuration module, a local registry and a set of active traffic interceptors. Applications in such node instantiate handlers in order to communicate with each other.

traffic. Another advantage of such decoupling is that traffic generated by different handlers may be manipulated by the same traffic interceptor, without the need for having multiple interceptors doing the same kind of manipulation. Having functionalities split across different handlers promotes the system's modularity. The downside of this approach is that interceptors must know how to treat data sent by every handler. Section 3.5 proposes a common message format between handlers and traffic interceptors, in order to overcome such disadvantage.

Finally, a configuration module responsible for configuring handlers and interceptors in a given node is proposed. Configuration is required in order to adapt the platform for different scenarios. Handlers and interceptors may communicate with the configuration module and change their settings accordingly.

Adaptability and security can easily be implemented using this architecture. Security can be implemented using an interceptor which encrypts and decrypts packets. This interceptor can also verify the application payload in each packet and analyze if it compromises users' privacy and act accordingly. As for adaptability, interceptors may be loaded and unloaded dynamically so it is possible to create another module which implements adaptive policies. For example, applications may communicate with this module requesting a system reconfiguration to adapt to a change in the environment. This module is then able to unload and load interceptors to provide this new configuration.

3.3.2 Handler

Handlers are the main structure of the proposed platform. The idea behind these structures is to abstract applications from transport details and traffic manipulation as shown in figure 3.5.

Applications instantiate handlers and configure them based on the kind of transport they require. Handlers

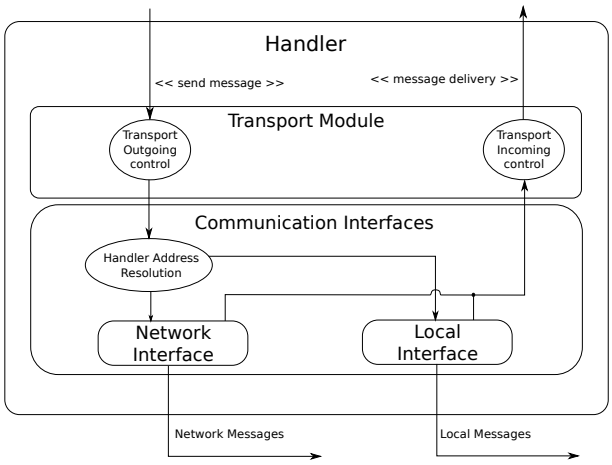


Figure 3.5: Handler architecture. A handler is composed of two main parts - a transport module and a communication interface module.

will then perform the necessary control to satisfy such configuration. This task is assigned to a transport module in each handler. Other configurations such as network packet aggregation and synchronization are configured by the configuration module, because they are part of the system's configuration and not specific to each handler.

It is important to notice that handlers provide a single point of interaction with the proposed platform to applications, creating a simpler design. Applications just need to know the destination handler's address and send data to it. However, a handler may have more than one communication interface. One of them performs local communications with other modules and another performs network communication. Therefore, handlers have more than one address. In order to overcome such problem we propose that handlers have a unique identifier, known to each application that can be mapped into handlers addresses.

Handler address resolution is perform by communicating with the local registry present in each node. Upon resolving the destination handler's identifier, the source handler chooses the local or network interface. We propose a division between local and network interfaces to avoid manipulation of local traffic. If all traffic generated by an handler would be sent through the same interface, then local traffic would also be manipulated. However, in general local traffic does not require manipulation since most manipulation techniques have goals that overcome some network issue. Manipulating both traffic would impose extra load in each node, which can be easily avoided by this separation.

As for communication between application and handlers, we propose an asynchronous approach. Applications request handlers to send messages. However, it is the transport module's responsibility to decide when the message actually is sent to the communication interfaces. Likewise, whenever a message is received, it is the transport module's decision when to deliver it to the application. This approach is necessary in order to delegate transport control to the transport module. If for example, messages are to be sent over a reliable channel it is necessary to control message flow in order to guarantee reliability properties. This must be done by the transport module.

3.3.3 Traffic Interceptors

Traffic interceptors are responsible for traffic manipulation. We propose an architecture based in a plug-in scheme, as shown in figure 3.6. Each interceptor is treated as a plug-in that can be registered with the interceptor framework and perform traffic manipulation. This allows decoupling between interceptors and eases the developing of new ones. Another advantage of such scheme is the capability of loading and unloading interceptors at runtime.

The interceptor framework is composed by an interceptor manager, a rule manager, several rules and several filters. The interceptor manager is responsible for registering and unregistering interceptors with the interceptor framework. This manager communicates with the rule manager, requesting it to register rules for a given interceptor. These rules describe which packets should be intercepted and when. The rule manager is responsible for registering and unregistering rules. Rules are composed by several filters, which will filter packets at different interception points - incoming packets, outgoing packets and forwarded packets. In other words, filters describe which packets should be intercepted and rules combine several filters and describe at which interception point each filter should act.

Filters provide more flexibility, in the sense that an interceptor may be aware of only a subset of all traffic. For example, with this scheme an interceptor may manipulate only traffic destined to a specific handler or from a specific node. Each filter has a callback registered by the interceptor which created the filter. This callback is called each time a packet is received and matches a given filter. However, depending on the kind of manipulation an interceptor does, it may require more than one filter at different interception points. As an example consider a security interceptor, which must cipher and decipher outgoing and incoming traffic respectively. Applications may request this interceptor to secure their traffic. However, this interceptor's implementation is completely transparent to the applications. Therefore, they should not know that to secure connections two filters must be registered at specific points. The ideal method would be to just request the security interceptor to secure a specific traffic flow and let it choose which filters to register. Thus we propose a higher abstraction to filters and call it a rule. A rule comprises a set of filters. Rules for this security interceptor would have two filters. The gain with such approach is that applications request interceptors to create a rule for a specific traffic flow, without having to know which filters to register. If the security protocol changes, filters may change as needed without affecting rules and therefore application code.

For the framework itself, a layered architecture is proposed. Each interceptor registers itself with the interceptor manager, which in turn will register rules with the rule manager. Layered architectures have the advantage of separating responsibilities between layers, making interceptor and rule management much easier.

3.3.4 Handler Registry

In section 3.3.2 it is stated that handlers may have more than one communication interface and that each handler needs to resolve handler identifiers into communication interfaces addresses. This information is stored in a local registry in each node as shown in figure 3.7.

Each registry entry should map a handler identifier into an address. The address resolution is achieved

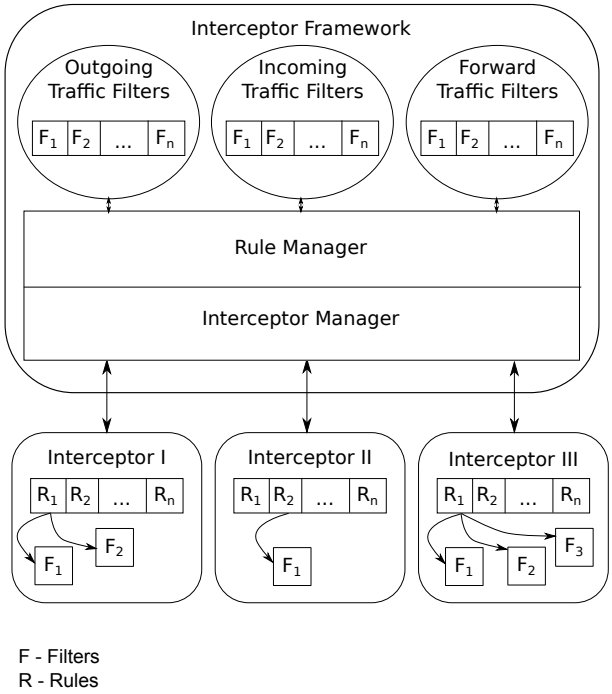


Figure 3.6: Interceptor framework architecture. Each interceptor registers itself with the interceptor manager, which in turn will register interceptor rules with the rule manager.

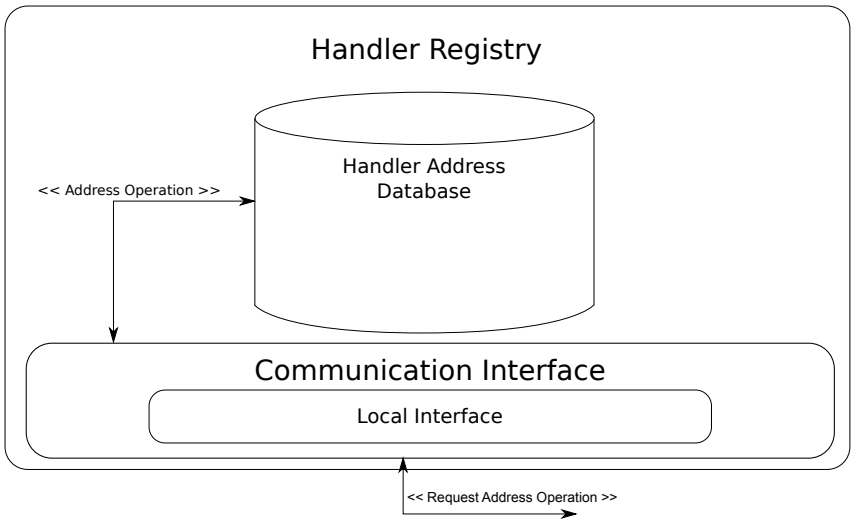


Figure 3.7: Handler registry. Processes may access the database by communicating with the registry daemon through the registry's local interface.

by each handler by communicating with the registry. Traffic between handlers and registry does not need to be manipulate by interceptors, so we propose that the registry has only one local interface.

Also, the handler registry should provide means for managing its current translation table. Address operations may involve creating, deleting and obtaining handler addresses. This is important for applications such as service discovery, which need to store the address of each service. If such application would be using the proposed architecture, means for manipulating the registry translation tables are crucial.

3.3.5 System Configuration

The proposed architecture is required to work in different scenarios. Therefore, as studied in section 2, the architecture must be configurable to such scenarios. A configuration module with an architecture represented in figure 3.8 is proposed.

This module is responsible for configuring the interceptors and handler parameters in a node. This should not be confused with the configuration an application does when instantiating an handler. The configuration module only stores information about parameters each handler should use, such as how many messages the transport module can buffer and how many should it keep buffered in case of reliable transmission. When created, an handler should query the configuration module for the needed parameters. Like in the local registry there is no need for traffic manipulation between handlers and the configuration module. For this reason, it has only one local interface.

The configuration module will also configure and load each interceptor according to the given configuration. However, to avoid having handlers sending messages before every interceptor is loaded and configured we propose a boot component. This component will load and configure each interceptor before populating the parameter database. This will ensure that handlers can only be configured if the boot component has already run.

3.4 Functional Specification

3.4.1 Handler Functionalities

Handlers are the structures used to send and receive data by applications, which need to use the proposed architecture. The send and receive operations are asynchronous. This means that when sending a message, applications request some handler to send it but do not block. Whether the message is sent or not is left to the handler to decide. Likewise, applications do not block to listen for incoming messages. The handler structure will notify the application when a message is received. When compared to synchronous send and receive, this scheme makes it easier for handlers to manage message flow. In fact, applications may proceed with other tasks while handlers take care of every transport detail.

Prior to sending messages, handlers add the necessary headers. These headers create a common message format, which also helps handlers and interceptors in managing traffic. However, the Protocol Data Unit (PDU) header is of most importance for the handlers' transport module. Most of its fields are used in the developed transport protocols.

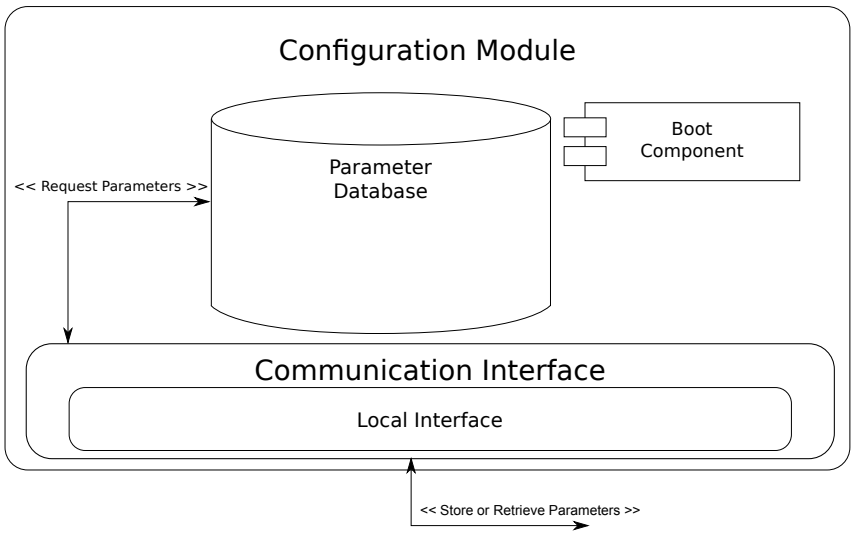


Figure 3.8: Configuration module architecture. Modules may communicate with this module using its local interface.

Handlers will also perform address resolution for applications. This means that applications need only to specify the destination handler's identifier for which they need to send data. The address resolution is performed by the handlers' communication interface. In sum, handlers are the bridge between applications and the underlying platform. They must be used to provide a flexible and easy method for applications to deliver data.

3.4.1.1 Transport protocols

In section 2 we described two main types of transport protocols used in wireless sensor networks - reliable and unreliable transport. Since the developed platform requires a good extensibility these two types of transport must be supported by it. The unreliable protocol is very simple as it sends each message without applying any kind of control. Each time an application sends data to the transport module of a certain handler using unreliable transport, it will be marked with a sequence number and a time stamp (If synchronization is required). Messages are sequentially marked only to give applications an easy way of identifying the data that is being sent. Likewise, upon receiving a message there is no procedure done by the unreliable transport control. Messages are delivered to applications as soon as they are received.

As for reliable transport, the control protocol is based in a NACK scheme. As in unreliable transport messages are marked sequentially. However, in reliable transport these sequence numbers are used to control message flow and guarantee reliability. It is also necessary to provide error checking for each message received. This can be achieved through checksums performed in the data sent by each handler.

The reliable transport module functionality can be separated in three components. One that is responsible for sending application data, another which is responsible for data reception and another which delivers messages to applications and sends the necessary NACK messages. The simplest component is the sender component. When a message is to be sent, this component must mark it with an appropriate sequence number and checksum the data. The reliability check is done by the control component. This component is responsible for checking if received messages can be delivered to applications and send NACKs for missing

messages. These components are explained in detail in the following section.

State Machines

Each component in the reliable transport implements a state machine. Figure 3.9 shows the sender state machine.

Upon initialization, the sender component puts himself to sleep until there are messages to be sent. In the eventuality that a message is ready to be sent it will first verify if it has a connection to the message's destination. If it does not, such connection will be created. Connections are used to identify the appropriate sequence number that should be given to a certain message. This is important because sequence numbers must be specific for a given connection, otherwise the receiver is not able to identify lost messages.

After getting a sequence number and being checksummed, the message is sent to its destination and buffered in case it needs to be resent. Algorithm 3.1 demonstrates pseudo code for this procedure. Line 1 declares the sending messages buffer as a global variable to be accessed by other functions. Line 7 sends a message to its destination and line 8 buffers it for possible retransmissions.

Algorithm 3.1 Send application messages

```
1: global sending_buffer
2: while TRUE do
3:     message ← get_next_message(sending_buffer)
4:     if not have_connection_for(message.destination) then
5:         connection ← create_connection_for(message.destination)
6:     end if
7:     send_message(message)
8:     buffer_message(connection.buffered_messages, message)
9: end while
```

Figure 3.10 shows the receiver component state machine. Upon receiving a normal message, that is, a message that contains application data, the receiver will first check if a connection to the message's source has already been created. If not it will create such connection and buffer the received message. Buffered messages will later be delivered to applications by the control component, explained below. After buffering a message the receiver will return to the receiving state and wait for more incoming messages. If a NACK arrives, then it will produce missing messages. This means that older messages will be buffered to be resent by the sender component. Algorithm 3.1 shows pseudo code for the reliable receiver. If a NACK is received

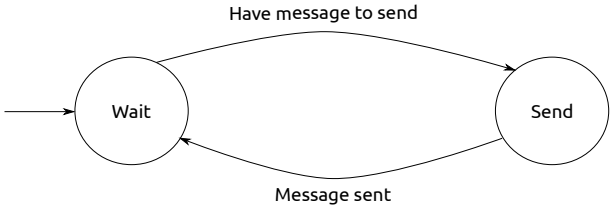


Figure 3.9: Handler transport module sender state machine.

a function *handle_nack* is called, which implements algorithm 3.2. If in the other hand a normal message is received, it is buffer to later be delivered to application by the control component.

Algorithm 3.2 Handle a received NACK

Require: pdu as input PDU of type NACK

```
1: sequence_gap ← pdu.data
2: connection ← get_connection_for_node(pdu.src)
3: i ← sequence_gap.left_sn
4: for i to sequence_gap.right_sn do
5:     produce_message(i, sending_buffer. connection.buffered_messages)
6: end for
```

Algorithm 3.3 Receive messages

```
1: while TRUE do
2:     message ← receive_message()
3:     if not have_connection_for(message.destination) then
4:         create_connection_for(message.destination)
5:     end if
6:     if message.type = NACK then
7:         handle_nack(message)
8:     else
9:         buffer_received_message(message)
10:    end if
11: end while
```

Figure 3.11 shows the state machine for the control component. In its initial state the control component will wait a specific amount of time named *nack timeout*. When the timeout happens, the control will check if messages can be delivered to the application and if there are messages missing. A message with a sequence number S can be delivered to the application if and only if a message with a sequence number $S - 1$ is delivered before it. Therefore, the control component will check the received buffered messages and deliver in order all messages which sequence number obeys such rule. In the same procedure, it will check if there are any messages missing and if so it will send a NACK to the messages' source. It is important to notice that received messages end up in different buffers, that is, they are grouped by source. Having messages buffered by source helps the control component in knowing which sequence numbers are missing. Therefore, a buffer of received messages is created in a per connection basis. Algorithm 3.4 gives pseudo code for the control component. Line 1 declares a global list of buffers, where received messages are stored. This list contains a buffer per connection. Lines 3 to 10 search for consecutive received messages and deliver them to the application. A function *send_nack* will send a NACK for the gap of messages beginning at the message sequence number given as an argument. After these steps, the control component will wait for a period specified as NACK_TIMEOUT.

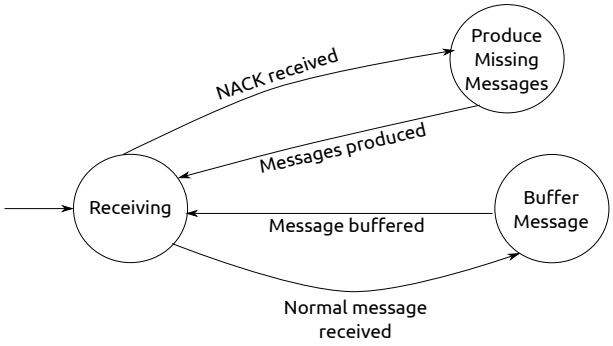


Figure 3.10: Handler transport module receiver state machine.

Algorithm 3.4 Deliver messages to applications

```
1:  global received_msgs
2:  while TRUE do
3:      for buffer in received_msgs do
4:          for m in buffer do
5:              if m then
6:                  deliver_message(m)
7:              else
8:                  break
9:              end if
10:         end for
11:         send_nack(m)
12:     end for
13:     sleep(NACK_TIMEOUT)
14: end while
```

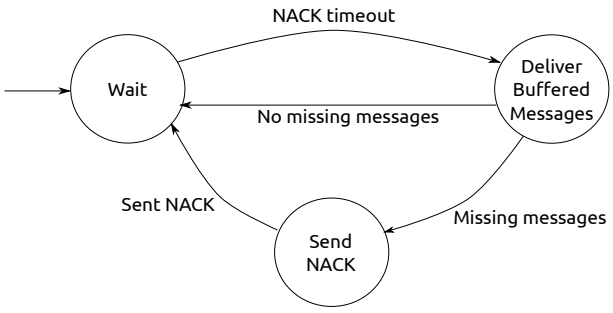


Figure 3.11: Handler transport module control state machine.

Although this NACK scheme is more ideal for wireless sensor networks, than an ACK scheme, it still suffers from a small issue. If there are several messages missing, then multiple NACK messages will be sent. This can end up flooding the network, which is not desirable. Therefore, to avoid such problem we proposed an alternative NACK scheme, where a NACK message contains all sequence numbers from missing messages. Moreover, it does not carry them explicitly. Instead it identifies gaps of missing messages by sending the smallest and largest sequence numbers in a gap of missing messages. This will keep the NACK message size small and will avoid multiple NACK messages.

As an example, consider figure 3.12 where a buffer for received messages is shown. Each message is represented by the letter M and is subscripted by its sequence number. If the control component would check such buffer in this state, it would first deliver message M_1 and then message M_2. After this, it would send a NACK for messages 3 to 6.

3.4.2 Traffic Interceptors Functionalities

As described in section 3.3.3, traffic interceptors register different filters for certain interception points. The interceptor framework must check each Internet Protocol (IP) packet against such filters and for each match it must call the filter's callback. First, pseudo code for filter matching is show in algorithm 3.5.

The idea behind such algorithm is to ensure that an IP packet matches a given filter. Notice that the algorithm only tries to match the packet's fields that are also present in the given filter. This is important to ensure that filters are applied based only in the fields they specify. For example, if a filter specifies a destination port then algorithm 3.5 will check if such port is the same as the one present in the packet's transport header. No assumption is made about other fields of the packet.

Each time an IP packet passes through an interception point, the interceptor framework will try to match every filter registered for that interception point. Assuming that a function *match_filter* exists and implements algorithm 3.5, it is possible to define a procedure that matches each filter in an interception point. Algorithm 3.6 shows pseudo code for such procedure.

The algorithm requires as input an interception point and an IP packet intercepted at said point. Line 1 obtains every filter registered for the given interception point. The for loop will try to match each filter in the interception point with the given IP packet. If a filter matches such IP packet, then its callback will be applied to the IP packet. Then, based on the decision returned by this callback, algorithm 3.6 will stop filter matching or continue. This is shown in line 6, where d stores the returned decision. Finally, notice that algorithm 3.6 will always return the last decision given by a filter callback. This ensures that the interceptor framework will act accordingly to the filters' decisions. For example, if a filter chooses to drop a certain packet, then the interceptor framework must drop it.

Figure 3.12: Reliable receiver message buffer example.

Algorithm 3.5 Match a filter against an IP packet

Require: F as input filter

Require: IP as input IP packet

```
1: if F.filtering_by_protocol ∧ F.protocol ≠ IP.protocol then
2:     return FALSE
3: end if
4: if F.filtering_by_src_addr ∧ F.src_addr ≠ IP.src_addr then
5:     return FALSE
6: end if
7: if F.filtering_by_dst_addr ∧ F.dst_addr ≠ IP.dst_addr then
8:     return FALSE
9: end if
10: TP ← get_transport_packet(IP)
11: if F.filtering_by_src_port ∧ F.src_port ≠ TP.src_port then
12:     return FALSE
13: end if
14: if F.filtering_by_dst_port ∧ F.dst_port ≠ TP.dst_port then
15:     return FALSE
16: end if
17: return TRUE
```

Algorithm 3.6 Match filters against an IP packet

Require: INTP as input interception point

Require: IP as input IP packet

```
1: filters ← get_filters_for(INTP)
2: i ← 0
3: for f ← filters[i], filters.length do
4:     if match_filter(f, IP) then
5:         d ← f.apply_callback(IP)
6:         if d ≠ CONTINUE then
7:             return d
8:         end if
9:     end if
10: end for
11: return CONTINUE
```

3.4.2.1 Synchronization Interceptor Functionality

Given the state of the art presented in section 2.2.1 an algorithm for delay estimation is proposed to synchronize data sent by applications. In this solution, each node gives an estimation of the time it contributed to delaying a certain packet and sends it to the next node. The sum of these delays will give a fairly good approach to the time passed since the packet was created until it arrived at its destination. The destination node can then compute the packet's creation time according to its local clock. Moreover if this clock is externally synchronized, the computed time will be globally accurate. In order to compute such delays it is necessary to intercept packets at key points, such as when leaving a node and entering one. Therefore, the proposed solution is to create an interceptor which implements the synchronization protocol.

The basic idea of the synchronization protocol is to count the time PDUs spent in each node and their transmission delays. When a PDU reaches its destination, it will contain the total time elapsed since it was created. Considering a path with N nodes, a PDU's creation time can be computed as:

$$T_{creation} = T_{reference} - \sum_{n=1}^{n-1}(Tn_n) - \sum_{n=1}^{n-1}(Tt_n) \tag{3.1}$$

Where $T_{reference}$ is the time taken by any node, which needs to know the creation time of the PDU. Tn_n is the time the PDU spent inside node n and Tt_n is the time node n calculated for the PDU's transmission.

Computing the time a PDU spends in each node is done in two steps. First, when a PDU is received by a node it is given a time stamp. This time stamp is actually the PDU's creation time according to the node's local clock. Step two takes place when the PDU is ready to leave the node. In this step, a time stamp is taken and the PDU's creation time is subtracted to it. This will yield the time elapsed since the PDU was created. Notice that this time is relative, which means it can be used by any node in the network. These steps are repeated by each node in the network until eventually the PDU will reach its destination and only step one is performed. At this time, the computed value is the PDU's creation time according to the destination's local clock. As a final remark, when PDUs are created in a node they are given a time stamp according to that node's local clock. This will ensure that step two converts correctly the PDU's creation time to a relative time.

Algorithms 3.7 and 3.8 show pseudo code for step one and step two respectively. It is assumed that a function *get_current_time* exists and is able to return the node's current time with an arbitrary known precision. This function is used to obtain time stamps. Both algorithms receive a PDU as input. It is important to notice that although the code for both algorithms is the same, the resulting time has different meanings. Algorithm 3.7 computes the PDU's creation time according to the node's local clock. Algorithm 3.8 computes the time elapsed since the PDU was created.

Algorithm 3.7 Synchronize incoming PDUs

Require: PDU as input PDU

1: $t_{in} \leftarrow$ get_current_time()
2: $t_{pdu} \leftarrow$ get_time_value_from_pdu(PDU)
3: $t_r \leftarrow t_{in} - t_{pdu}$
4: set_time_value_in_pdu(PDU, t_r)
5: **return** PDU

Algorithm 3.8 Synchronize outgoing PDUs

Require: PDU as input PDU

1: $t_{out} \leftarrow$ get_current_time()

2: $t_{pdu} \leftarrow$ get_time_value_from_pdu(PDU)

3: $t_r \leftarrow t_{out} - t_{pdu}$

4: set_time_value_in_pdu(PDU, t_r)

5: **return** PDU

In order to better understand this solution, we consider the scenario depicted in figure 3.13. Node A sends a PDU to node S (sink node), which will be routed through node B. Each node has a rule, which will synchronize said PDU. For sake of simplicity, time is expressed in seconds since each node started running. It is clear from the depicted scenario that the PDU spent 3 seconds in Node A and 1 second in node B. Therefore it is expected at node S to have a delay of 4 seconds.

When the PDU is sent to a transport module in an handler on node A, it receives its first time stamp $t_{pdu} = 7s$. When node A sends it to the network, the synchronization interceptor takes another time stamp $t_{out} = 10s$ and computes the time the PDU spent in node A, which is $t_r = 3s$. This step is depicted in algorithm 3.8.

Upon reception in node B, the synchronization interceptor takes a time stamp $t_{in} = 5s$ and computes its initial time stamp according to its clock $t_r = 2s$. This step is depicted in algorithm 3.7. When the PDU leaves node B, the synchronization interceptor takes yet another time stamp and computes the value which will indicate how much time it spent on node A and B, which is $t_r = 4s$. This step is depicted in algorithm 3.8.

When node S receives the PDU, the delay in it is exactly what was expected, that is, 4 seconds. The synchronization interceptor in node S can compute the PDU's initial time stamp by taking its own time stamp and subtract it the delay present in the PDU, which is done by algorithm 3.7. According to node S, the PDU sent by node A was created 98 seconds after node S started running.

An important point to notice is that in the given example, transmission time and propagation was not accounted for in order to keep the example understandable. However, in real live scenarios those times may need to be calculated. In most sensor network cases, nodes transmit data using wireless communications, which may take time in the order of milliseconds. Therefore, a method to calculate transmission and propagation time of a given message is necessary.

Propagation time between two nodes is simply the distance between them divided by the speed of light ($3 \times 10^8 m/s$). In most applications of WSNs the distance between two nodes is sufficiently small to ignore propagation time. As for transmission time in general, it is possible to estimate it based on the size of the

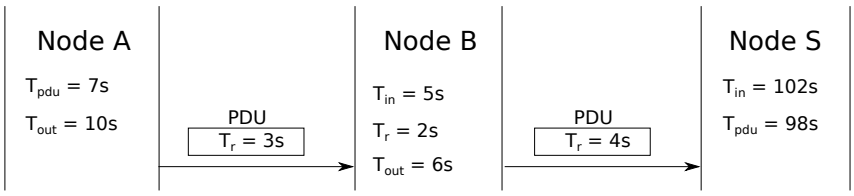


Figure 3.13: Synchronization protocol example.

frame being transmitted and its transmission rate. This yields the formula:

$$T_{trans} = \frac{\text{Frame Size}}{\text{Transmission Rate}} \tag{3.2}$$

However, this time may vary depending on the method used to transmit each frame. For example, in IEEE 802.11 standard times such as Distributed Coordination Function Interframe Space (DIFS) and Short Interframe Space (SIFS) must be included in order to achieve an accurate transmission time. It may also be necessary to include retransmission and back off times. Since estimation can be done based on these known values, transmission time may be calculated before sending a certain frame. This allows it to be added to the PDU's time value before sending it. It is possible to extend algorithm 3.8 to estimate transmission time when wireless medium is being used. Assuming a function *estimate_frame_transmission_time_for_pdu* exists and is able to calculate T_{trans} with the same precision as get_current_time, then algorithm 3.8 can be extended to algorithm 3.9.

Algorithm 3.9 Synchronize outgoing PDUs

Require: PDU as input PDU

1: $t_{out} \leftarrow$ get_current_time()
2: $t_{pdu} \leftarrow$ get_time_value_from_pdu(PDU)
3: $t_r \leftarrow t_{out} - t_{pdu}$
4: set_time_value_in_pdu(PDU, t_r)
5: **if** Using Wireless Transmission **then**
6: $\quad t_{trans} \leftarrow$ estimate_frame_transmission_time_for_pdu(PDU)
7: $\quad t_r \leftarrow$ get_time_value_from_pdu(PDU) + t_{trans}
8: $\quad$ set_time_value_in_pdu(PDU, t_r)
9: **end if**
10: **return** PDU

3.4.2.2 Aggregation Interceptor Functionality

Aggregation is done at interceptor level. Packets are aggregated by destination address. This ensures that aggregated packets are all destined to the same node, which will ease the computation procedure for the desegregation protocol. Algorithm 3.10 shows pseudo code for the aggregation protocol.

Line 1 obtains the buffer for the IP packet's destination. Lines 2 to 6 will check if the buffer is full. The field $data\_length$ evaluates to the size of the data already buffered and the field $capacity$ evaluates to the total amount of data the buffer can handle. If it is full, the algorithm will call a function *flush_buffer*, which will create the aggregated packet to be sent. Then a decision to continue with the packet's processing is returned. Notice that the packet will now be a different IP packet, that is, it will be the aggregated packet. In the case that the buffer still has room for the input IP packet, the algorithm will buffer it and return a decision to stop its processing. It is important to notice that packets are aggregated in decreasing order, that is, the oldest packet will be at the tail of the aggregated buffer. This eases the computation of the time value for each packet. Algorithm 3.11 shows the pseudo code for the function *buffer_packet*.

Algorithm 3.10 Aggregate packet

Require: IP as input IP packet

1: $b \leftarrow$ get_buffer_for(IP.dst_addr)
2: **if** b.data_length + IP.total_length > b.capacity **then**
3: $old\_ip \leftarrow$ IP
4: IP $\leftarrow$ flush_buffer(b)
5: buffer_packet(old_ip, b)
6: **return** CONTINUE
7: **end if**
8: buffer_packet(IP, b)
9: **return** STOP

Algorithm 3.11 Buffer packet

Require: IP as input IP packet

Require: B as input packet buffer

1: **if** using synchronization **then**
2: $head \leftarrow$ head_packet(B)
3: $pdu \leftarrow$ get_pdu_from(IP)
4: set_time_value_in_pdu($head$, get_time_value_from_pdu(pdu) - $head$.time_value)
5: **end if**
6: push_packet(IP, B)

In its simplest form the algorithm will just push the received IP packet into the buffer. However, if synchronization is being used it is necessary to compute the relative time for the packet at the head of the buffer. This will ensure that the synchronization interceptor needs to know only about the first PDU in the aggregated packet. Therefore, algorithms 3.7 and 3.9 will still work if aggregation is being used. Algorithm 3.12 shows pseudo code for the creation of the new aggregated packet. The resulting packet will replace the one being intercepted by the framework in order to force other interceptors to process it.

Algorithm 3.12 Flush buffer

Require: B as input packet buffer

1: $head \leftarrow$ head_packet(B)
2: $ip \leftarrow$ new_ip_header_from($head$)
3: ip.total_length $\leftarrow$ IP_HEADER_SIZE + B.data_length
4: ip.source_address $\leftarrow$ THIS_NODE_IP_ADDR
5: calculate_ip_checksum(ip)
6: add_ip_payload(ip, B.data)
7: empty_buffer(B)
8: **return** ip

First, the algorithm will fetch the packet at the buffer's head. It does this to be able to copy every field in such packet to the new IP packet. This is done in lines 1 and 2. After this copy it needs to change three fields in the new IP header - the IP total length, source address and checksum. It is assumed that a constant *IP_HEADER_SIZE* is available and evaluates to the size of an IP header without any options nor padding. It is also assumed that it is possible to obtain the node's current IP address, which is represented by *THIS_NODE_IP_ADDR*. After such computations are made, the payload for the new IP packet can be added and the buffer cleaned. This payload is the data present in the buffer and is added by the function *add_ip_payload*.

3.4.2.3 Desegregation Interceptor Functionality

Desegregation functionality is separated from aggregation to provide extra flexibility. This way a node can desegregate traffic without needing to load the aggregation interceptor and vice versa. The desegregation interceptor will separate packets inside an aggregated IP packet and inject then in the node's network stack. For this purpose it relies on a function *inject*, which is capable of doing such procedure. Algorithm 3.13 shows the pseudo code for the desegregation protocol.

The algorithm cooperates with the synchronization interceptor in order to time stamp each PDU correctly. To achieve this goal it first grabs the time value present in the first PDU in the IP packet - lines 1 to 4. Notice that this must had already been synchronized by the synchronization interceptor, specifically with algorithm 3.7. Then, the algorithm will loop through each packet present in the IP aggregated packet and compute their time stamps - lines 6 to 12. Here an optimization is made. Due to the way algorithm 3.10 aggregates packets, the first packet in the aggregate packet will be the one with the highest sequence number. Therefore, it is necessary to deliver packets starting at the last one in the aggregate packet. In order to avoid several loops,

Algorithm 3.13 Desegregate IP packet

Require: IP as input IP packet

1: $pdu \leftarrow$ get_pdu_from(IP)
2: **if** using synchronization **then**
3: $total\_acc\_time \leftarrow$ get_time_value_from_pdu(pdu)
4: **end if**
5: **for** i to $scatter\_buffer$.size **OR** IP contains packets **do**
6: $l \leftarrow$ current_packet_in(IP)
7: **if** using synchronization **then**
8: set_time_value_in_pdu(l, get_time_value_from_pdu(l) $-total\_acc\_time$)
9: $total\_acc\_time \leftarrow total\_acc\_time+$ get_time_value_from_pdu(l)
10: **end if**
11: prepend_scatter($scatter\_buffer$, l)
12: **end for**
13: **for** p in $scatter\_buffer$ **do**
14: inject(p)
15: **end for**

an optimization was made to desegregate packets as quick as possible. The aggregate packet is iterated once and pointers to the beginning of each PDU are kept in a buffer called *scatter_buffer* - lines 5 to 11. At the end of the desegregation procedure, this buffer is iterated and packets are delivered - lines 13 to 15. Notice that to keep the order correct, packets are putted at the beginning of the buffer - line 11.

3.5 Data Format Specification

Each handler in the implemented prototype communicates using a predefined PDU. The general idea is to create a message header with application data as payload and send it to other handlers. The transport module of each handler will prefix application payload with a PDU header, while the communication interfaces module will prefix each PDU with a control byte. This byte is a set of flags, which will help interceptors determine if the packet is for example aggregated at the network level.

Figure 3.14 shows the Open Systems Interconnection (OSI) stack from the point of view of the proposed platform. The handler layer is where handlers add the PDU header and control byte when sending data. This layer expects received data to be prefixed by a control byte and a PDU header. Therefore, data sent to handlers must conform to this format.

Although the control byte is used only by interceptors, handlers are the ones responsible for adding it. This ensures that the format of each packet is according to specification even if there are no interceptors loaded in the current node.

Applications do not have access to the PDU header. However, important fields such as the PDU's time of creation are passed to an application callback upon its reception.

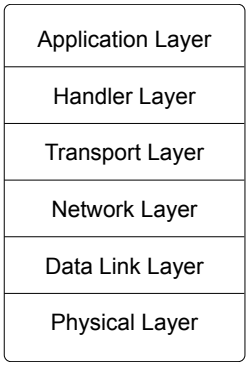


Figure 3.14: OSI stack from the perspective of the proposed platform. The new layer - *handler layer* - adds the necessary headers to application payload.

3.5.1 PDU Specification

Figure 3.15 shows the structure of a PDU, which has a total of 24 bytes. The fields of the PDU are organized as follows:

Creation Time

This field contains the PDU's creation time as marked by the synchronization protocol. It occupies eight bytes to enable high resolution, that is, time with nanosecond precision.

Air Time Estimation

This field contains the PDU's air time estimation as marked by the synchronization protocol. This is an estimation of the PDU's transmission time. It occupies seven bytes since it will always be in the order of milliseconds.

Sequence Number

A sequence number generated by the transport module of each handler. It has a total of four bytes and it is treated as an unsigned integer, thus yielding a total of 2^{32} packets without it overflowing. This value range is enough for the reliable transport control to identify when the sequence number has overflown without discarding messages.

Length

This field indicates the number of bytes used by application payload. It occupies two bytes yielding a theoretical maximum payload size of 2^{16}. However, the network Maximum Transmission Unit (MTU) must be taken into account when sending a large payload of data.

Identifier

This field fully identifies the node and handler that sent the PDU. It occupies two bytes and must be unique. Section 4.2.1.2 explains how the prototypes guarantees such uniqueness.

Flags

This field is mostly used by transport modules in each handler. It occupies one byte. The first bit indicates that the PDU is a NACK message. The second indicates that it was synchronized by the application itself, thus preventing the synchronization algorithm from trying to synchronize said PDU. The third bit indicates that the PDU is an Oldest Sequence Number (OSN) message, used by the

reliable transport control. The rest of the bits are not used and can therefore extend the prototype functionality.

3.5.2 Control Byte Specification

Additionally to a PDU header, the communication interface in each handler adds another header composed of only one byte. This byte is used by interceptors and its purpose is to control the payload in each IP packet. A control byte has the following structure:

Synchronized

Indicates if the IP packet has already been synchronized.

Application Aggregated

Indicates that the IP packet contains application aggregated packets.

IP Aggregated

Indicates that the IP packet contains aggregated IP packets.

Remaining bits

These bits are not used by the platform and can be used to extend its functionality.

The presented structure for a PDU is used by every handler in the network, which means that each handler expects messages with this header. Moreover, before sending messages and receiving them, the communication interface in each handler treats messages as having a control byte, followed by a PDU header and possibly some application data.

3.5.3 Aggregated Packet Specification

The aggregation protocol will try to join packets in two different ways as explained in section 4.2.2.2. The aggregated packets are created at the IP layer, thus they need to have an IP header and a transport protocol header before any aggregated data. The implemented prototype uses the UDP protocol and therefore this section explains the aggregated packet structures with such protocol. However, other transport protocols may be used as long as each packet has an IP header, followed by a transport header and a control byte.

3.5.3.1 Application Aggregated Packet Specification

The aggregation protocol can aggregate application packets using PDU headers sent by each handler. Figure 3.16 shows an application aggregated packet ready to be sent to the data link layer.

The aggregated packet consists of two PDUs, which might have been sent by different applications. However, these applications are all local. To ensure that PDUs reach the correct destination, all aggregated

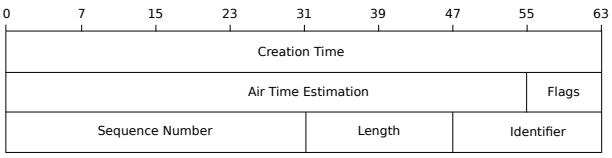


Figure 3.15: PDU header description.

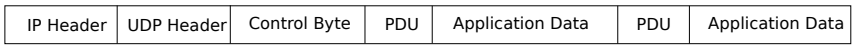


Figure 3.16: Application aggregated packet example.

packets contain PDUs with the same destination. The control byte will indicate that this packet is an application aggregated packet. The desegregation protocol uses the length field in each PDU to know where to split aggregated packets.

As a final remark, it is important to notice the order in which PDUs are aggregated. The aggregation protocol uses a First In First Out (FIFO) scheme, which means that in the packet shown in figure 3.16, PDUs are ordered in decreasing order in respect to the sequence number.

3.5.3.2 Network Aggregated Packet Specification

Aggregation at IP level is also possible. Figure 3.17 shows an IP aggregated packet ready to be sent to the data link layer.

The aggregated packet consists in two IP packets, which might have been sent by different nodes. In fact, the first IP has been highlighted to show this particularity. Although it is not shown in figure 3.17, application aggregated packets may also be aggregated at IP level. For example, the highlighted packet may contain more than one PDU. The desegregation protocol uses the total length field of the IP header of each aggregated packet to split them.

Figure 3.17: Network aggregated packet example.

4 Implementation

4.1 Used Technologies

4.1.1 TS-7500 Boards

The developed prototype was implemented and tested in Technologic Systems (TS) 7500 boards. These boards are embedded computers with an Acorn RISC Machine (ARM) processor from Cavium Networks, capable of booting Linux OS. Their small size and low energy requirements makes them attractive to sensor network developing. In fact, TS 7500 boards are being used in MIAVITA and MIT projects presented in section 3, which makes them essential for developing the proposed architecture. These boards run Linux Kernel version 2.6.24.4 provided by the board vendor.

Another advantage of such boards is related to the OS they use. Linux is used amongst a wide number of users and therefore it has a larger set of working drivers than other OS's, such as TinyOS. Developing is also easier, since it is possible to write programs in well known programming languages, such as C.

The board is capable of booting a minimal Linux Kernel in less than three seconds. The filesystem resides in an Secure Digital (SD) card. It has 64MB of memory available as well as two Universal Serial Bus (USB) ports and one Ethernet port. It also has another USB port used to power up the board, which means it can run with only five volts.

The processor is compatible with an ARM9, meaning it only supports a subset of the ARM9 Instruction Set Architecture (ISA). Finally, the board has several Digital Input Output (DIO) pins, which can be used to communicate with external device using Serial Peripheral Interface (SPI) bus. In MIAVITA's project, these DIO pins are connected to the Analog-to-Digital Converter (ADC), which samples data from the geophone. In MIT's project they are used to communicate with different devices, such as infra-red sensors.

4.1.2 Loadable Kernel Modules

The Linux Kernel has improved its design, becoming modular over time. Instead of being a huge block of code, the Linux Kernel is divided in small blocks called modules. When a system boots, only a subset of these modules need to be loaded into memory instead of the whole Kernel. This particular feature contributed to make Linux an OS capable of running in embedded systems, which typically have low memory resources.

Also, one feature present in Linux is the capability of extending and removing functionality from the running Kernel. This is possible because modules are made of object code that can be dynamically linked and unlinked. Therefore, modules can be loaded and unloaded into the running kernel without the need for a system reboot. Modules can belong to one or more classes, depending on the way they interact with the Kernel subsystems and functionalities they offer. Figure 4.1 shows the main overview of such classes.

- *Process Management*

  This class covers every aspect related to process abstraction. In other words, this class is where the Kernel implements the abstraction of multiple processes running in the same Central Processing Unit (CPU). One very important component that belongs to this class is the Kernel scheduler, which as the name implies is responsible for managing every process's execution.

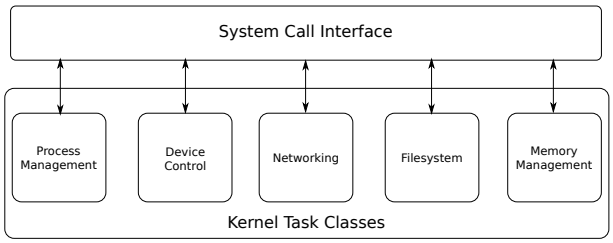


Figure 4.1: Linux Kernel classes organization.

- *Device Control*

  In Linux every peripheral will eventually be mapped into a virtual device. These devices must be controlled at Kernel level by modules called device drivers. Therefore this class encompasses every module which function is to perform communication between the Kernel and some peripheral device.

- *Networking*

  Every issue related to routing and address resolution is implemented in the Kernel. Moreover, receiving packets is an asynchronous event that must be handled by the Kernel before delivering packets to processes. Each module that deals with these details is said to belong to the networking class.

- *Filesystem*

  Linux is heavily based in the concept of a file, that is, almost everything in Linux can be treated as a file. Moreover, it supports several types of filesystems. This abstraction to hardware must be implemented at the Kernel level, so processes can easily treat almost every entity as a file. Modules that implement such features are said to belong to the filesystem class.

- *Memory Management*

  Kernel needs to manage memory access in a system in order to abstract processes from hardware details. It creates a virtual addressing space, where it can map the memory used by every entity in the system. This method also eases the use of access policies, creating a more secure memory usage. Each module present in the Kernel that deals with these details belongs to the memory management class.

The implemented prototype uses the kernel loadable modules feature and makes heavy usage of the network subsystem, thus belonging to the networking class. In order to manipulate traffic it is crucial to use the Kernel's network subsystem, as this is the only way of accessing the IP layer implemented by Linux, which in turn is required to know if traffic is to be forwarded or not.

Although not clearly marked throughout the Kernel code, Linux implements its network stack based on the Internet model. User level programs typically run in the application layer and communicate with the Kernel using sockets. If the socket is using a transport protocol, it will send data to the transport layer, which in turn will pass it to the network layer. Afterwards, the Kernel will send it to the data link layer, which in turn will send the packet to the network adapter driver. A structure called Socket Buffer (SKB) carries packets sent by user level applications and passed from layer to layer. During this process it gets filled with each layer's protocol header. For example, in the network layer the Kernel prefixes data in the SKB with an IP header. The SKB structure is the basis of the Linux network subsystem and is therefore essential for the developed prototype,

which achieves traffic manipulation by altering the SKB's content.

Another feature present in the Linux Kernel is related to module communication. It is possible for a module M_1 to access symbols in a module M_2 and vice-versa. This promotes interaction between modules and helps developers in using existing code loaded into the Kernel. It is also possible to communicate from user level application to Kernel level and vice-versa. The Netlink family is a special family of sockets created for this purpose. Developers often use "proc entries" to achieve the same result, although they were not built with such goal. A "proc entry" is a special file in the filesystem, which writes and reads are handled directly in kernel level by the module that created it. The module can parse whatever is written to a proc entry and it can also write back when it is read.

4.1.3 Netfilters API

The Netfilters (NF) API is already integrated into the Linux Kernel and provides means to interact with network traffic that traverses the IP layer. It is possible to register callbacks within the NF API, which will be called when an IP packet is sent, forwarded or received in the Kernel. Such callbacks are called NF hooks. Each IP packet treated by the Kernel will eventually reach the NF API. More specifically, an IP packet will eventually pass through one or more points, called NF hook points. These are *pre-routing*, *post-routing*, *forward*, *local in* and *local out* NF hook points as shown in figure 4.2. Each of these points may have zero or more NF hooks registered. When an IP packet reaches a certain NF hook point, the NF API will call every registered hook for that point sequentially. Then, for each hook called this way, it will pass it an SKB containing the hooked IP packet. Each hook will then examine and possibly change the packet and return a decision to the NF API. Depending on the decision given, the packet will continue its path, be dropped or queued for later processing. If the decision is other than to continue with the packet's processing, the NF API will stop calling the remainder hooks on the given NF hook point.

In order to interact with the NF API, a Kernel module must be built. This module must define a set of hooks and register them with the NF API. When an IP packet is received by the Kernel, the NF API will call hooks registered by that module at pre-routing. Then, if the packet was not queued nor dropped it will be routed by the kernel. If it is destined for the current node, hooks registered at local in point will be called. If it is destined for other nodes, hooks registered at forward point will be called instead. After this, only hooks registered at post-routing may be called. Packets generated locally and sent to the network will be processed by registered

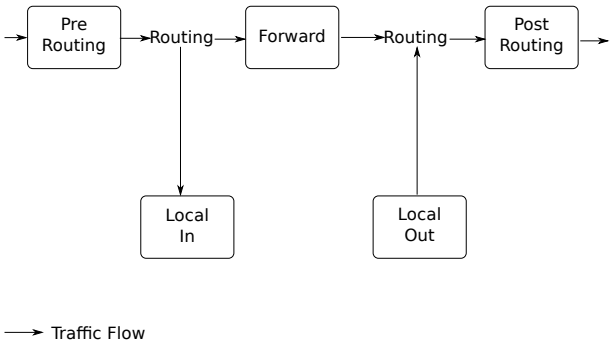


Figure 4.2: Netfilter hooking points. Packets flow according to the arrows.

hooks at local out point. A module does not need to define hooks for every NF hook point. It can also define more than one hook for the same NF hook point.

Decisions returned by hooks will determine the packets' fate. They can be dropped, causing no more hooks to run on them. They can be accepted causing no disturbance in their route. They can also be queued and until dequeued no hook will process them and they can be reprocessed, that is, the same hook may decide to process the same packet again.

There are some key features in NF hooks that deserve attention. First, when a hook runs it only has access to the IP packet, which means it no longer has access to the data link layer headers. Also, IP packets are never fragmented when a hook runs on them. The defragmentation process is handled by the NF API before hooks are called. This makes packet handling much easier.

The NF API was used to implement the proposed interceptor framework and corresponding interceptors. Such implementation is described in section 4.2.2.

4.1.4 Linux IPC

The Linux Inter Process Communication (IPC) provides means for processes to exchange messages without having to pass through network interfaces. This means that traffic generated by processes that use Linux IPC will not be caught by the NF API. There are five types of mechanisms that can be used for IPC:

- *Signals*

  Signals are mostly used for asynchronous communication. Processes register themselves to treat specific signals. When the registered signal is fired, the registered callback will be executed. Signals are the oldest IPC present in Linux and have some limitations. If a process receives two different signals at the same time it has no guarantees about the order they will be handled. Also, the number of signals is limited to the word size of the processor and there is no mechanism to treat multiple signals of the same kind, that is, if one process receives multiple signals of the same kind it will act as if it had only received one.

- *Named Pipes*

  Pipes are data structures that implement a FIFO policy, which means that the first data written to a pipe will be the first data read from it. Named pipes are pipes with an associated name, which can be used to reference the pipe. These structures offer unidirectional communication and therefore are not suitable for process communication which operates in a request-response paradigm. Also, pipes have a limited amount of data that can be stored, which means that a process will block if it writes to much data to a pipe.

- *Message Queues*

  Message Queues offer the possibility for multiple processes to write and read from them. Message queues can be thought as linked lists, where messages are written to the tail of the list and read from the head. Linux also imposes a maximum size for message queues, after which a process will block if it tries to write to the queue.

- *Shared Memory*

  With this mechanism processes communicate through memory that appears in their virtual memory

space. Although, addresses may not need to be the same for all processes, there is no guarantee about data accessing. When processes use shared memory they must rely on other mechanisms to synchronize access to such memory.

- *Sockets*

  Sockets are commonly known for their usage in network communication. However, Linux implements a new family of sockets known as sockets Unix, which can be used for IPC. The semantics are the same as in network sockets, thus making them suitable for a request-response paradigm.

Section 3 describes several modules that make up the platform. These modules must communicate with each other. Linux IPC was designed for such purpose, therefore sockets Unix are used with this goal throughout the implementation.

4.2 Prototype Implementation

The prototype was developed using the programming language C. Part of it is implemented as a dynamic link library for application development and the rest is implemented as Linux Kernel modules. Figure 4.3 shows an overview of its architecture.

Applications may have more than one handler. Each time an application creates an handler it must specify its configuration as described in section 4.2.1.1. Handlers are part of the developed shared library, thus they run in the same process as the application that is using them.

Before sending messages, handlers need to contact the local registry. This registry is a separate process running in the background accepting requests in a Unix socket. Handlers request the local registry to resolve handler identifiers into their addresses. For example, if a message is destined to an handler with identifier 1001, then the handler which sends said message will query the local registry for handler 1001 address.

The returned address may be a local or remote address. If it is local, then the message will not pass through the interceptor framework because it is sent through a Unix socket. If it is a remote address, then it will be sent using an Inet socket and may be intercepted by the interceptor framework.

Three interceptors were implemented. As implied by their names, one is responsible for temporal synchronizing messages and the other two are responsible for message aggregation and desegregation. The interceptor framework and each interceptor are implemented as separated Linux Kernel modules. This features promotes the extendability of the platform's functionality by easing the development of new interceptors. Also, as shown in figure 4.3 the NF API is used by the interceptor framework, which makes network traffic interception much more flexible.

Finally, the configuration module was also implemented as a separated process running in the background, which can be queried through a Unix socket. This module has a boot component responsible for making the first configurations and later on can be queried for transport related parameters.

4.2.1 Handler Implementation

The handler structure was implemented in two different parts in order to conform to the architecture proposed in section 3.3.2. Figure 4.4 outlines the implemented handler structure. Messages sent by applications

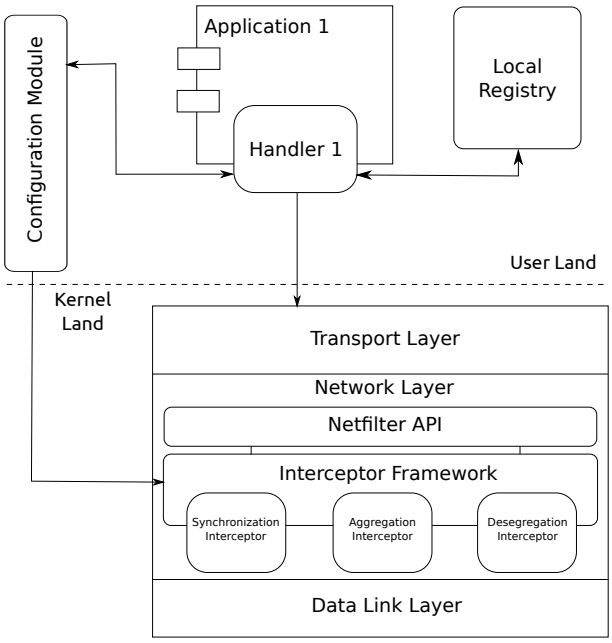


Figure 4.3: Overview of the implemented architecture. The interceptor framework was implemented at the kernel level, while the other modules were implemented in user level.

are stored in a message buffer inside the transport module. The transport module control will decide when messages are dequeued from the buffer and sent to the socket interface. The number of messages an application can send to the transport module is limited and if such limit is reached the application will block. Applications have no knowledge about the PDU structure used by handlers, which is prefixed to application data at the moment it is stored in the message buffer.

Upon dequeuing a message from the buffer and applying any control necessary, the transport module will send it to the socket interface. The socket interface was implemented with the goal of abstracting the transport module from the type of interface used. The handler may use sockets Unix for local communication and sockets Inet for network communication. Before sending a message to a socket, the socket interface must perform an address resolution. This is done by contacting the local registry through a socket Unix and

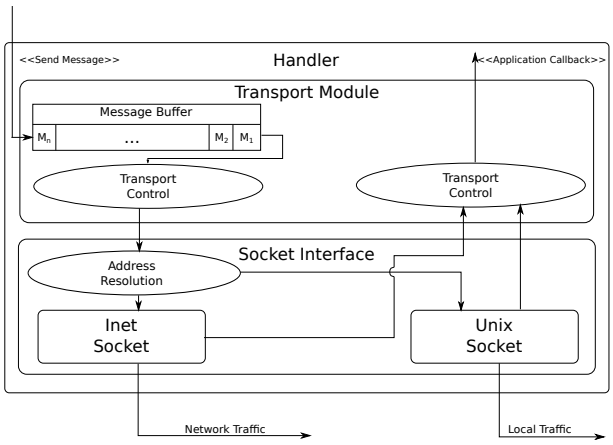


Figure 4.4: Implemented handler architecture.

obtaining a structure representing a generic address (Explained in section 4.2.1.2). After knowing which family the address belongs to, the appropriate socket will be used to send it.

As for message reception, any of the sockets in an handler may receive a message at any moment. When applications create handlers they need to specify a callback for message reception. Because enough buffering is made by the underlying architecture of a socket, there is no need for buffering received messages. When the transport module receives a message and after performing the necessary control it will call the application registered callback.

4.2.1.1 Handler Configuration

The prototype configuration module holds the implemented system configuration. This configuration specifies parameters necessary for configuring the transport module, such as how many messages an application can send to the transport module without block and how many messages should the reliable transport buffer in order to avoid message loss. However, if an handler uses reliable or unreliable transport must be configured at the moment of its creation. Also, an handler may or may not use both socket types, so the configuration must also specify such fact.

Applications may configure an handler by specifying a path to a file in the filesystem or pass a string with the configuration directly in code. Nevertheless, the format for such configuration must be Extensible Markup Language (XML), which has two great advantages. First, it can be used as a common language between different systems, making the configuration portable from system to system. Second, it can easily be extended to incorporate new configuration fields. Figure 4.5 is an example of an handler configuration.

The example shows a configuration for an handler with id 123, which uses unreliable transport, Inet socket bound to IP 192.168.0.1 on port 57843 and an Unix socket with path `/tmp/123`.

When configuring an handler, sockets marked as Unix will be automatically registered within the local registry. As explained in section 4.2.1.2, it does not make sense to automatically register Inet sockets. Therefore, if no Unix socket is specified, no connection to the local registry will be made.

4.2.1.2 Handler Management

This section explains how the handler registry is maintained and what strategy was implemented to guarantee the uniqueness of handlers' identifiers. Since an handler may have two kinds of sockets, the registry must be able to support both. The standard C library for Linux implements sockets based on a generic address structure. Typically such structure has sixteen bytes, which first two specify the socket family and the other fourteen are used for the address itself. In the case of IP version 4 Inet sockets, the IP address and port number fit into the last fourteen bytes, which means that a socket of Inet family can safely be casted to a

```
<handler id="123">
    <transport type="unreliable" />
    <address type="inet" addr="192.168.0.1" port="57843" />
    <address type="unix" addr="/tmp/123" />
</handler>
```

Figure 4.5: Example of an handler configuration.

generic address structure. However, in the case of Unix sockets the same is not always true. An Unix socket has an address that corresponds to a path in the filesystem, which may have a length of a hundred and seven characters. Nevertheless, most functions in the standard C library, which interact with sockets require a generic address structure and an additional parameter specifying the size of the address.

In order to take advantage of this fact and make the implementation simpler, the implemented prototype limits addresses of Unix sockets to thirteen characters and uses the last byte to terminate the string. This allows a safe cast from an handler's Unix socket address to a generic address structure and provides a more generic registry module, since there is no need to check the family of the address. In short, the registry module holds references to handlers' addresses. These addresses can either be local or network addresses.

Three operations are provided by the local registry. It is possible to register handlers, get their addresses and delete the created registry entries. In every operation, the desired address is identified by the handler's identifier. Therefore, it is required that this address be unique. However, each node may have its own registry module running and without a proper strategy different addresses may end up with the same identifier. In order to avoid this problem, the following scheme may be used to guarantee uniqueness amongst all handler identifiers. Each identifier is composed of two bytes. The first one identifies the node and the second identifies the handler inside that node. Figure 4.6 depicts such scheme.

Each node in the network must have a different host name. It is possible to use host names to generate different node identifiers. For example if every host name begins with a given pattern followed by a unique byte, this byte can be used to provide the node's identifier. As for the handler uniqueness, the registry module already ensures that no two handlers on the same node may have the same identifier. Thus, global uniqueness is guaranteed.

As a final remark, the implemented registry module can be queried through an Unix socket, avoiding traffic manipulation by loaded interceptors. Because local nodes communicate using local sockets, there is no need to register Inet sockets of local handlers. Moreover, if local Inet sockets were registered and used in local communications, local traffic would also be manipulated be traffic interceptors.

4.2.1.3 Supported Transport Methods

Reliable and unreliable transport were implemented, in order to provide applications with different methods for transferring their data. These are implemented in the handler's transport module.

In the case of unreliable transport, the transport module does not perform any kind of control. However, application data is still prefixed with the PDU header and sequentially marked. Also, if synchronization is being used, messages will be time stamped. After a message is sent, the transport module will discard any copies

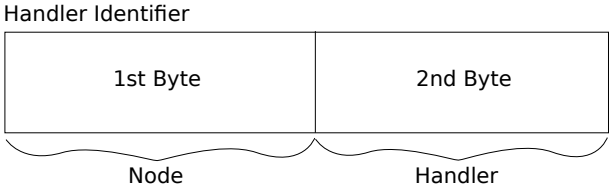


Figure 4.6: Structure of an handler's address.

of said message. Likewise, when a message is received by the unreliable transport module, it is immediately delivered to the application.

As for reliable transport, the implemented prototype supports the end-to-end protocol described in section 3.4.1.1. Messages are marked sequentially and UDP is used to transmit them, which checksums the application data. These two features provide ordered delivery and guarantee that messages are delivered without errors to the application. Sequence numbers start at zero and are incremented one unit per message. An handler using reliable transport must communicate with another handler using the same reliable transport implementation, to ensure reliability. Each time a message is sent from one handler to another, the transport control will lookup for a connection to that message's destination. In the eventuality that such connection does not exists it will be created. After a message is successfully sent, the reliable transport control will copy it to a temporary buffer. This buffer will store messages that can be resent and has a predefined size, which can be configured in the system's configuration.

Upon receiving each message, the transport receiver component will try to buffer it. However, since receiving buffers have a maximum size it is possible that there is no space in the buffer for the received message. In such case, the implemented protocol chooses to drop said message in the sense that it will send a NACK message for it later on. Figure 4.7 illustrates a case where there were no messages lost. For sake of simplicity, only one receiving buffer is shown. However, according to the system's architecture the implemented prototype stores a receiving buffer per connection.

From time to time the transport control component will try to deliver messages in each receiving buffer. In this case, messages M_1 and M_2 can be delivered to the application. Also, after doing this procedure the transport control will check if there are messages missing and if so it will send NACK messages for them.

However, due to the nature of protocols based in a NACK scheme, there is no way for handler A to be sure when to remove messages from the sending buffer. As a result of this fact and given that the sending buffer has a limited size, it is possible that handler A receives NACK messages for messages it no longer has. In such situation there are two possible solutions, either the connection is terminated with an error or those messages are skipped. In sensor networks, most applications require a continuous delivery of messages, which should not stop. Therefore, the implemented protocol chooses to skip those messages and notifies the receiving handler about this fact. Figure 4.8 shows an example where handler A notifies handler B about its decision to skip messages, by sending handler B the oldest sequence number present in the sending buffer.

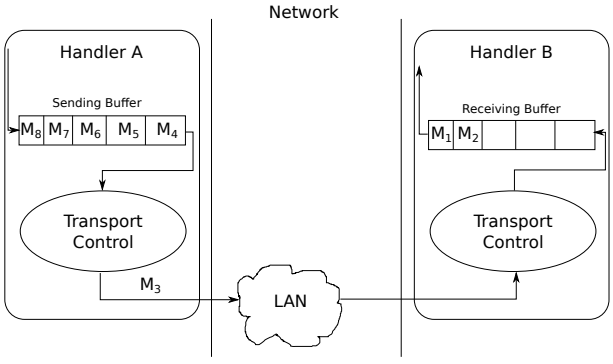


Figure 4.7: Example of reliable transmission buffers with no losses.

This message is called OSN. It is important to notice that messages are sent in order, thus it suffices to send the oldest sequence number present in the sending buffer. In sum, the implemented protocol is capable of guaranteeing ordered delivery with no errors nor losses, while preventing network flooding.

4.2.2 Interceptor Framework Implementation

The interceptor framework was implemented as a module for the Linux kernel. In order to take advantage of the Linux modules feature, each interceptor is also a module. This means that interceptors can be loaded and unloaded at runtime and enforces the decoupling between them. Figure 4.9 shows an overview of the implemented solution for the interceptor framework. Essentially, it provides two functions for interceptors. One is to register an interceptor and the other to unregister it. However, before being registered an interceptor must fill a structure, which indicates its name and a callback that will create the interceptor rules according to a given filter specification.

A filter is composed of four elements, as seen in figure 4.10. The first one is the interception point at which the filter should run. The second is the priority of the filter, which provides means to run some filters before others and may assume 256 different priority levels. Zero corresponds to the highest priority and 255 the lowest. The third structure in a filter is a callback placed by the interceptor, which created the filter. The fourth field in a filter is a structure with the filter specification. This structure holds information about how to match a filter. It provides filtering by different fields in the packet. The implemented prototype supports filtering by transport level protocol, destination and source address, destination and source port as specified in the filter architecture. Likewise, in order to conform with the framework's architecture if a filter specifies a destination port, every packet that has that destination port will be processed by the interceptor that registered such filter, that is, no assumption is made about other fields in the packet. This means that it is possible to register filters with empty specifications, which will result in the handling of every packet.

The proposed interceptor framework architecture suggested that it should be possible to manipulate incoming, outgoing and forwarded traffic. However, to take advantage of the NF API, not only does the prototype support those three kinds of traffic, but it also supports pre-routed traffic and post-routed traffic. Upon initialization, the interceptor framework will register five NF hooks, one for each NF hook point as show in figure 4.2 . Each time an IP packet triggers a NF hook, it will be checked against registered filters on that specific hook's interception point. For example, if the pre-routing hook is triggered, than the framework will check for filters registered at pre-routing. For each filter matching to a packet's specifications, the filter callback will be called. The interceptor which registered said filter will run the callback and return a NF decision. If such decision is other than keep processing the packet, the interceptor framework will stop matching other filters against that packet. This is described in algorithm 3.6.

Filters are registered by each interceptor upon receiving a filter specification. More precisely, the interceptor manager calls the registered callback of a given interceptor for rule creation and passes it a filter specification. The interceptor will then create a rule, which in turn will contain the necessary filters for the given specification. After this step, the interceptor manager will ask the rule manager to register the newly created rule. Finally, the rule manager will register the rule's filters according to their priorities. It is important to notice that the configuration module is able to register rules with the framework using a "proc" entry created

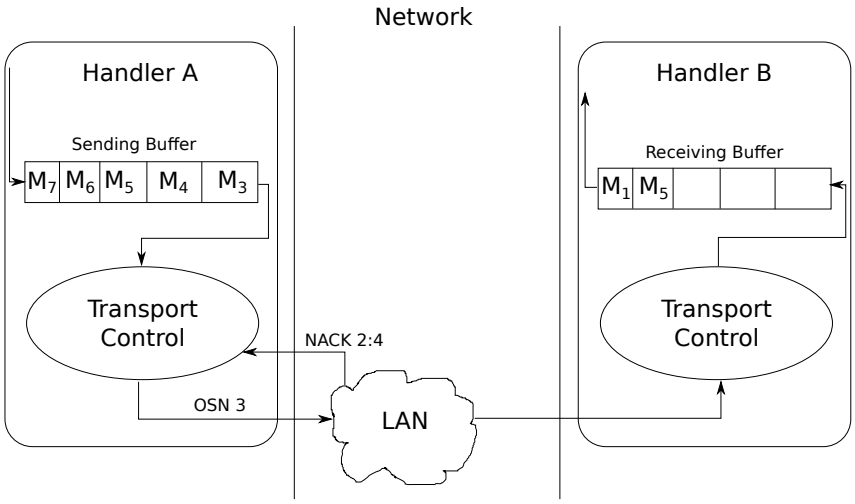


Figure 4.8: Example of reliable transmission buffers with losses and all messages recoverable.

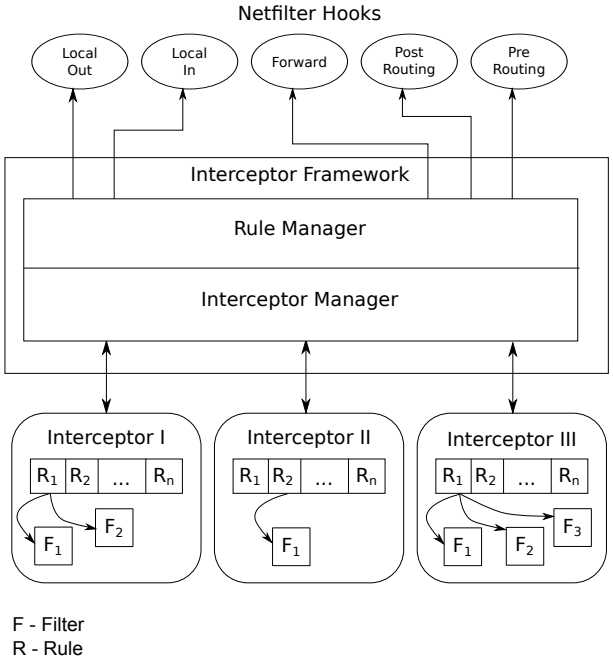


Figure 4.9: Implemented interceptor framework architecture. Like the framework itself, interceptors are Linux Kernel modules. Filters are registered with the Netfilter API.

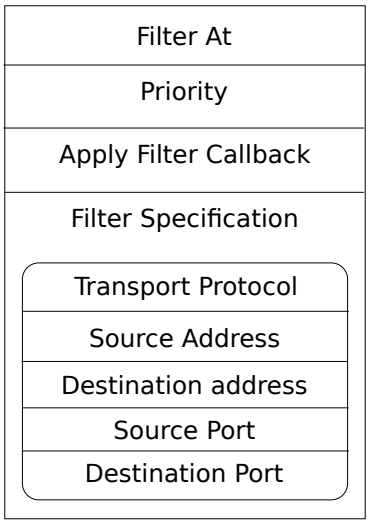


Figure 4.10: Interceptor framework filter architecture. Each filter will be applied at a specific point, which corresponds to the Netfilter hooking points.

for this purpose. This makes it possible for rules to be register at the same moment the configuration module initializes, which means that the interceptor framework will be fully configured before any handler can use it.

In the implemented solution, within the interceptor framework rules have a unique identifier. This identifier is used to easily delete the rule when it no longer applies. Deleting rules may become important for future work. Interceptors may be load and unloaded dynamically, which provides means to create an adaptive system. Therefore, it is extremely important to have an efficient way to delete interceptor rules, when that interceptor is unloaded. Rules also keep track of their filters, which helps the cleanup process when a rule is deleted.

4.2.2.1 Synchronization Interceptor Implementation

The synchronization protocol is implemented via an interceptor and called Cross Layer One Way Delay Estimation (CLOWDE). Each rule for the synchronization interceptor has two filters. One will be applied at pre-routing and the other at post-routing. The filter applied at pre-routing implements algorithm 3.7 and the filter at post-routing implements algorithm 3.8.

Each handler in a node using synchronization will time stamp the data in the moment it is sent to the transport module. This will be treated as the PDU's creation time. This means that an application, which requires synchronization with high precision must send the data to the transport module as soon as possible.

Filters at post-routing will run with the lowest priority possible to guarantee that synchronization is the last step in a PDU transmission. Filters at pre-routing will run with the highest priority to guarantee that synchronization is the first thing applied to a received PDU.

As already described in section 4.1.1, the experimental boards run a version of Linux as their OS and data is transmitted using its implementation of the OSI model. One important characteristic of this implementation is that each layer buffers data before passing it to the next layer. Therefore, it is necessary to measure the transmission time when a PDU leaves or enters the last layer, in order to get an accurate reading. Thus,

algorithm 3.9 is implemented in two parts. The synchronization interceptor can perform the algorithms 3.7 and 3.8. However, since it is implemented at the IP layer it is not able to get an accurate estimation of the frames' transmission time. Therefore, it relies on a modified Linux driver, which is capable of doing this estimation for it. The driver is the last piece of code a frame passes through before being transmitted. It also provides counters, which indicate how many transmission failures and retries have already occurred. This information is sufficient to provide an estimation of the transmission time a frame has suffered.

The created prototype uses USB WiFi cards with Ralink 2501 chipset. The standard driver provided by the Linux Kernel developers for this chipset and TS 7500 Kernel version does not allow for ad-hoc operation. However, this is essential in WSNs scenarios so an open source alternative was used. The driver provided by the WiFi card's vendor is capable of ad-hoc operation and uses the IEEE 802.11b standard without Request To Send (RTS) and Clear To Send (CTS) frames. This is called basic access mode. An example of a transmission without errors and with the medium free is illustrated in figure 4.11. In basic access mode, a node which has frames to send senses the medium before initiating a transmission. If the medium is free then it will wait a well defined time known as DIFS and send the data afterwards. Upon reception of a frame a node waits a time period known as SIFS before acknowledging the frame.

The protocol tries to estimate the time between t_0 and t_1, which corresponds to the DIFS time plus the time to transmit a frame. In 802.11b standard, the DIFS time has a well defined value, which is 50 microseconds and the time to transmit a frame can be calculated with the formula as shown in section 3.4.2.1:

$$T_{trans} = \frac{\text{Frame Size}}{\text{Transmission Rate}} \tag{4.1}$$

It is important to notice that parts of the frame, such as the preamble are transmitted at lower rates (1Mb/s). Therefore, if the medium is free and a frame suffers no losses its transmission time can be computed simply as:

$$T_{transmission} = DIFS + T_{trans} + T_{propagation} \tag{4.2}$$

However, if the medium is occupied, before initiating a transmission a node will wait a random back off time, chosen from 0 to the contention window size. Although the contention window is specific for each 802.11 standard, there is no defined way for how the protocol should implement the random procedure. Therefore,

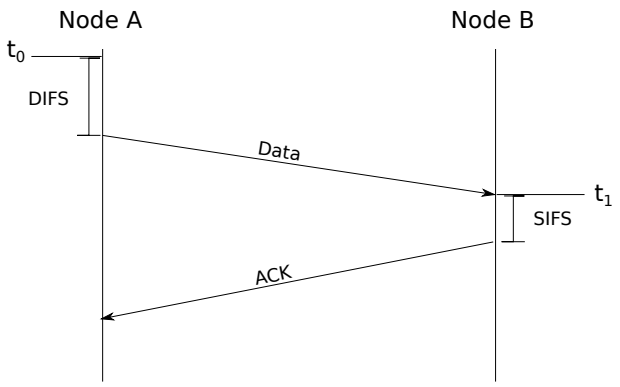


Figure 4.11: IEEE 802.11 basic access mode. In this mode there is no RTS CTS frames.

the random back off time depends on the wifi card's vendor and if no information is given it is not possible to correctly determine it. The used wifi card does not provide any information about back off times, thus we used an interpolation technique. Each time a frame is ready to be sent and the wifi card is free to transmit data, the driver copies said frame to a shared memory position and waits feedback from the board's firmware. The transmission time ($T_{transmission}$) is estimated at the moment the driver copies a frame to the shared memory position. Then, the wifi card tries to send the frame and updates both failure and retry counters. When another frame is ready to be sent, the modified version of the driver will store in it its transmission time and the number of retries or failures that the last one suffered. Upon reception, frames which suffered retries can be interpolated. However, this solution requires applications to send data at a known rate, which also is a common scenario in sensor network applications. By knowing the sampling rate it is possible to correlate received frames, which suffered retries with others which did not and give an estimation to their creation time.

Delay Estimation Error

The provided synchronization algorithm does not cover the entire PDU's path in each node. It fails to consider time spent in the data link layer and USB transmission delays. In order to minimize the error, the modified driver includes yet another estimation accounting for these times. The following equation computes the estimation for USB transmission time:

$$\epsilon = \frac{\text{Frame Size}}{\text{USB rate}} \cdot 2 + \sigma \tag{4.3}$$

Where the USB rate is 12 Mb/s for USB 1.1 and σ is a constant value. The equation includes a factor of 2 to include the receivers USB transmission delay. The value for σ was determined experimentally and represents an upper bound to minimize the delay estimation error. The procedure used to determine this constant is explained in section 5.

4.2.2.2 Aggregation Interceptor Implementation

The aggregation protocol was also implemented via an interceptor. The developed protocol supports aggregation at two different levels. The simplest aggregation is done to each PDU sent by applications in a node. The second level of aggregation is performed to IP packets, where it is possible to aggregate packets from different nodes. Both levels share the same intuition, which is to send multiple packets in one larger packet. However, these packets must all share the same destination, otherwise they may be delivered to the wrong handlers. The developed protocol implements the algorithm shown in 3.10. However, because the developed architecture relies on a transport protocol such as UDP, buffers must be identified not only by packets' destination IP address but also by their destination port. This information is required to build an aggregated packet correctly, which can be safely delivered to the transport layer in the Linux Kernel. The procedures to aggregate application packets and IP packets only differ in how the control byte is prefixed to packets. Assuming that the correct filters are registered, the general procedure for aggregating packets with the aggregation interceptor can still be described by algorithm 3.10.

Nevertheless, there is one particularity to consider in this general procedure which is related to the implemented solution. A control byte is always prefixed to the UDP payload of the aggregated packet. As specified

in section 3.5 this will help determine what kind of packet it is.

Aggregation at application level is done to each PDU sent by different handlers on the same node. Filters for this level of aggregation are applied with the highest priority at the local out NF hook, which ensures that PDU's are all generated by local handlers. This level of aggregation is useful in nodes which have multiple applications sending PDU's to the same node. When an aggregated packet is ready to be sent, the aggregation interceptor will prefix it with a control byte marked as an application aggregated packet.

As for aggregation at IP level, filters are applied at the post-routing NF hook before synchronization. This will ensure that not only local generated IP packets are aggregated, but also forwarded ones. When an aggregated packet is ready to be sent, the aggregation interceptor will prefix it with a control byte marked as an IP aggregated packet.

As a final remark, the aggregation interceptor in each node can be configured to optimize its usage. It is possible to configure the size of the aggregated buffers for both levels of aggregation and which levels are supported by the aggregation interceptor. Also, the implemented solution is able to cooperate with the synchronization interceptor, enabling synchronization and aggregation to work at the same time.

4.2.2.3 Desegregation Interceptor Implementation

The implementation of the desegregation protocol as an independent interceptor was necessary, because nodes aggregating packets may not need to desegregate them and vice-versa. Therefore, it is crucial to provide a separation between the aggregation and desegregation protocol. Each rule in this interceptor will register only one filter at the local in NF hook. This ensures that aggregated packets are destined to the local node and need to be desegregated. As specified in section 3.5, packets are aggregated so that the oldest one is the last one in the IP packet. Therefore, it is necessary to desegregate them by the same order, thus guaranteeing their ordered delivery. The desegregation procedure will also keep track of time stamps in order to correctly time stamp each packet.

Upon receiving an aggregated packet, this interceptor will check its control byte and apply the appropriate procedure. Although application aggregated packets are dealt differently from IP aggregated packets, the general procedure is the same. Thus, the desegregation interceptor implements algorithm 3.13.

Using the described procedure, the desegregation interceptor ensures that packets are correctly synchronized. Moreover, by injecting desegregated packets into the network it guarantees that other NF hooks will run on them and therefore registered interceptors. This feature makes it possible to aggregate application packets and IP packets at the same time, because desegregated IP packets that contain more than one PDU will be desegregated again. Also, there is no risk of flooding the network with injected packets. Linux Kernel routing facility will detect that such packets are to be delivered locally and will automatically call the IP local delivery functions, which in turn will call the registered NF hooks.

4.2.3 Configuration Module Implementation

The configuration module holds information about which interceptors should be loaded at boot time and several transport module parameters. When this module starts, it reads a given configuration file and configures

the platform according to that file. As with the configuration format for handlers, the one chosen for the configuration file was XML. Figure 4.12 shows a configuration example file.

```
<configuration>
 <interceptor_directory path="/tmp/interceptors/" />

 <transport_types>
   <transport type="reliable">
     <parameter name="message_buffer_size" value="40" />
     <parameter name="window" value="10" />
   </transport>
   <transport type="unreliable">
     <parameter name="message_buffer_size" value="40" />
   </transport>
 </transport_types>

 <interceptors>
   <interceptor module="aggregate.ko" aggregate_ip_packets="1" aggregate_app_packets="0"/>
   <interceptor module="synchronization.ko"/>
 </interceptors>

 <interceptor_rules>
  <rule interceptor="aggregation" saddr="192.168.0.123" sport="57843" />
 </interceptor_rules>
</configuration>
```

Figure 4.12: Example of a system configuration. The aggregation and synchronization interceptor will be loaded and a rule for aggregating packets will be registered. Then the database will be populated with the transport information for handlers.

The implemented prototype configures the system in two steps. The first one will be performed as soon as the module starts and it consists in loading and configuring interceptors. This must be the first step in the configuration process, because they must be loaded before any handler can send traffic to the network. There are three essential fields which need to be configured. The configuration module needs to know where interceptor Kernel modules are located, which parameters and values should be passed when loading them and which rules should be created. The configuration file in figure 4.12 specifies that interceptors are located in `/tmp/interceptors`. The aggregation and synchronization interceptors will be loaded when the configuration modules starts. It also indicates that the aggregation interceptor will only aggregate IP packets and that a rule to aggregate traffic with IP source address 192.168.0.123 and source port 57843 should be created.

The second step is to build a configuration table, where handlers can obtain property values such as reliable transport NACK timeout. In order to do this, handlers must contact the configuration module through a Unix socket. The configuration file shown in figure 4.12, configures two types of transport - reliable and unreliable. It defines how many message both types can buffer and how many should the reliable transport keep in memory so it can retransmit them in case of NACK messages.

As already explained in this document, the configuration module does not configure handlers it just configures the characteristics of each available types of transport. If an handler uses reliable transport it will query the configuration module for properties related to such transport type. This scheme has the advantage that a node may have applications running handlers with different types of transport.

5 Tests

5.1 Test overview

The main goal of this thesis was to develop a multi-functional platform that could be adapted to different monitoring systems. Three components were developed in order to validate the created platform. A synchronization protocol capable of synchronizing data in one way by piggybacking time stamps in packets sent by applications. An in-network generic aggregation protocol, which main goal was to reduce wireless congestion. Finally, a reliable transport protocol which goal is to demonstrate that the platform is capable of supporting multiple transport protocols. Such protocol was designed to be energy-efficient, thus it does not have congestion nor flow control.

In order to validate the implemented prototype several tests were developed. Each component was tested individual to better study their advantages and disadvantages. The synchronization protocol was tested and validated against results given by GPS devices. The goal of such test was to compare the delay given by the synchronization protocol and the one provided by the GPS devices. As for the aggregation protocol, the conducted tests were devised to show if the protocol can reduce wireless medium congestion and what percentage of packet overhead it reduces. Finally, the transport protocol was tested to prove that it can be used with the platform. Also, results about the number of control messages it generates are analyzed in several cases.

5.2 Synchronization Protocol

To validate the performance of our synchronization algorithm (CLOWDE) we installed a GPS device in the sender and receiver nodes and compared the results achieved with our algorithm with the results of the GPS device. Thus, as there are only two GPS devices available a very limited set of tests were possible. Given this limitation, we assessed CLOWDE using two simple, but yet effective, tests. The first one uses the most simple network, comprising only one sender and one receiver. In this case, the results obtained with CLOWDE should be very precise, as most of the delays are effectively taken into account. In a second case, there is an intermediate node, which introduces additional delay to process and that may cause collisions in the wireless medium. Hence, one might expect less precise results with our algorithm.

The next sections detail the methodology used to measure the GPS time, the tests that have been performed and the results that have been achieved.

5.2.1 Used Methodology

5.2.1.1 Obtaining the GPS Time

The used GPS receptor is the Trimble Lassen iQ, which provides a serial port for communication. Time accuracy is within a tenth of millisecond. This feature was used only to learn the current time up to the second. However, the designed protocol requires a higher precision, thus a different approach was developed to get an accuracy of microseconds.

The GPS device provides a Pulse per Second (PPS) signal, with an accuracy of 50 nanoseconds. The PPS was used to set the seconds on the TS boards. Therefore, an interrupt line was connected from the PPS to the board's processor and an interrupt handler was written to accurately set the board's clock. At boot time, a user level program reads the time from the GPS and sets a variable in the Kernel, which holds the GPS seconds. The interrupt handler will increment this variable by one each time it is triggered. Algorithms 5.1 and 5.2 provide pseudo code for the initialization of the variable and interrupt handler respectively.

Algorithm 5.1 Initialize GPS second counter

Require: s as input representing GPS seconds

1: **global** $time \leftarrow$ get_kernel_time()

2: $time$.secs $\leftarrow s$

Algorithm 5.2 GPS interrupt handler

1: $time$.usecs $\leftarrow$ get_kernel_time().usecs

2: $time$.secs $\leftarrow time$.secs $+1$

In the initialization of the GPS second counter the board's clock is read onto a global variable $time$. This will make sure that the current microseconds are also stored in the same variable, which in turn will enable user level programs to retrieve the correct value of microseconds that elapsed since the initialization of the variable. The final step is to store the GPS seconds in the $secs$ field of the $time$ variable. The interrupt handler simply increments this field each time an interrupt arrives and stores the current microseconds. This last step helps prevent errors due to the interrupt latency, which was measured and has an upper bound of 90 microseconds.

Finally, user level programs obtain the time using a system call designed for this purpose. The system call's latency was measured and has an upper bound of 4 microseconds. Algorithm 5.3 shows pseudo code for this system call.

Algorithm 5.3 Get GPS second counter system call

1: $t \leftarrow$ get_kernel_time()

2: **return** $time$.secs $*1000000 + (t$.usecs $- time$.usecs $)$

5.2.1.2 Delay Error Minimization

In order to minimize the delay error several tests were ran to determine a value for σ in equation 4.3. In these tests only two nodes were used. Packets were sent from one to another at regular intervals with 25Hz frequency. The sending application placed the GPS time inside each packet and sent it to the receiving application. For each packet received by this application the GPS time was read and the delay was computed as $d = t_{packet} - t_{gps}$, where t_{packet} is the time inside the packet taken by the sender's GPS and t_{gps} is the receiver's GPS time. In these tests the formula presented in equation 4.3 was not used.

The experiment was repeated using different payload sizes (20, 500 and 1000 Bytes) and with different transmission rates (1, 5.5 and 11 Mb/s). Each run consisted in sending 100 packets. It was observed that

the error of the synchronization protocol delay estimation, the difference between the GPS delay and the synchronization protocol, varied little for each message although the values were in the order of milliseconds. Table 5.1 shows the average delay estimate error for each run.

We can observe that the error increases with the packet size, as expected due to the USB transmissions. It also decreases as the transmissions rate increases. We then calculated a σ value of 645 microseconds for use with Equation 4.3 in order to minimize the delay estimation error as follows. First we computed the mean of the delay error, which is 1406.98×10^{-6} seconds. This is the value for ϵ in equation 4.3. We can compute σ as:

$$1406.98 \times 10^{-6} = \frac{\text{frame size} \cdot 8}{12 \times 10^{6}} \cdot 2 + \sigma$$

The above formula gives the upper bound of 645 microseconds for σ.

5.2.2 Performance tests

Scenario One

The first test scenario used only two nodes, a sender and a receiver. Each one is connected to a GPS device used to measure packet delay between them. Figure 5.1 shows the network topology used for this test.

In each test, the sender node sends 100 packets to the receiver, using a Constant Bit Rate (CBR) traffic generation. Both GPS and synchronization protocol delays are measured to be compared. Different payload sizes have been used (20, 500 and 1000 Bytes), with different sampling frequencies (25, 50 and 100 Hz). At the 802.11 network card, the transmission rate has been adjusted to 1 Mb/s, 5.5 Mb/s and 11 Mb/s. The parameter σ has been set to the value previously mentioned.

Table 5.2 presents the average estimated delay and the average error for the delay estimated by the synchronization protocol when using a single hop. We can see that the protocol underestimates in some scenarios and overestimates in others. However, the delay error is very small, usually in the order of the tenths of microseconds and never crossing above 0.7 milliseconds.

Figure 5.2 compares the delay estimation provided by the synchronization protocol against the time measured using the GPS for two nodes at different transmission rates, 1000 Bytes of payload and 100Hz sampling frequency. Notice that the difference varies little as the synchronization protocol and the GPS show the same behavior.

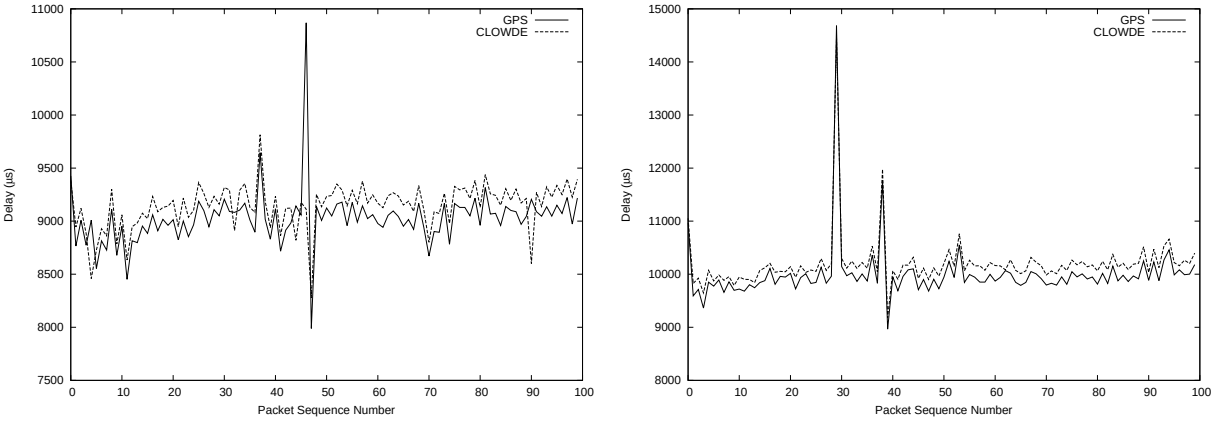
Table 5.1: Synchronization protocol delay error without ϵ correction using 2 nodes at 25Hz

	Delay estimation error (μs)			
Packet size (B)	Transmission rate=1Mb/s	Transmission rate=2Mb/s	Transmission rate=5.5Mb/s	Transmission rate=11Mb/s
20	1132.18	870.28	847.94	773.07
500	1409.42	1443.82	1274.04	1320.55
1000	2107.17	1911.77	1961.84	1831.71

Figure 5.1: Network topology used for the first test scenario.

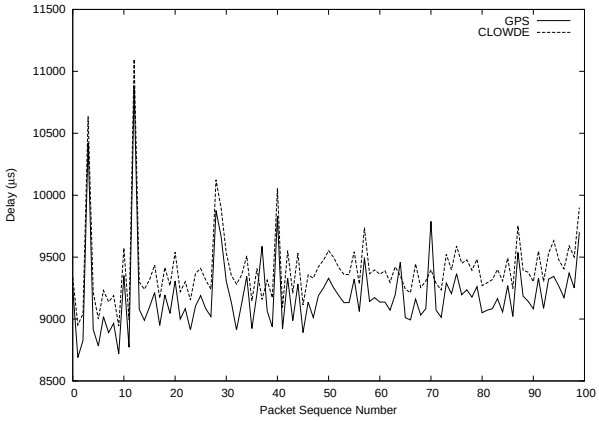
Table 5.2: Synchronization protocol average delay estimation error with 2 nodes

Transmission rate (Mb/s)	Payload size (B)	Mean delay (μ) / Delay estimation error (μ/s)					
		Sample rate=25Hz		Sample rate=50Hz		Sample rate=100Hz	
1	20	3292.44	-441.78	2519.90	-42.58	2515.15	45.73
	500	12711.70	73.62	7749.24	-85.89	8558.54	-82.53
	1000	17173.80	-63.06	8476.54	-196.48	9145.22	-118.43
5.5	20	2650.31	48.74	2583.94	49.85	2719.54	1.67
	500	8284.61	-147.29	9027.87	-74.39	8776.41	-63.99
	1000	9474.41	-213.66	10234.80	-204.08	10195.60	-200.28
11	20	3428.54	-44.48	2625.93	41.27	2494.13	39.80
	500	9536.90	-132.41	8363.14	-132.04	8261.68	690.46
	1000	9219.82	-198.25	9459.00	-236.85	9405.00	-212.20



(a) 1 Mb/s

(b) 5.5 Mb/s



(c) 11 Mb/s

Figure 5.2: Synchronization protocol delay estimation versus GPS delay using 100Hz, 1000B and different transmission rates.

Figure 5.3 shows the Cumulative Distribution Function (CDF) of the error present in the synchronization protocol estimated using packets produced at 100Hz, under the various packet sizes and one hop.

Scenario Two

The second test scenario used three nodes, a sender, a hop and a receiver. Sender and receiver are connected to a GPS device used to measure packet delay between them. All generated packets pass through the hop node. Figure 5.4 shows the network topology for this test.

As in the first scenario, the sender node sends 100 packets to the receiver, using a CBR traffic. Both GPS and synchronization protocol delays are measured to be compared. Transmission rates of 1 Mb/s, 5.5 Mb/s and 11 Mb/s were tested as well as different payload sizes (20, 500 and 1000 Bytes) and sampling frequencies (25, 50 and 100 Hz). The parameter σ has been set to the value previously mentioned.

Table 5.3 shows the same information as table 5.2, for 100Hz, using the three hop scenario. The introduction of the third node caused the average delay to vary more widely. This variation was largely confirmed by the GPS and may be explained by the greater medium access competition.

In this scenario, the synchronization protocol provided less accurate delay estimations, however in most situations the average error remained below 10% or 1.5 milliseconds.

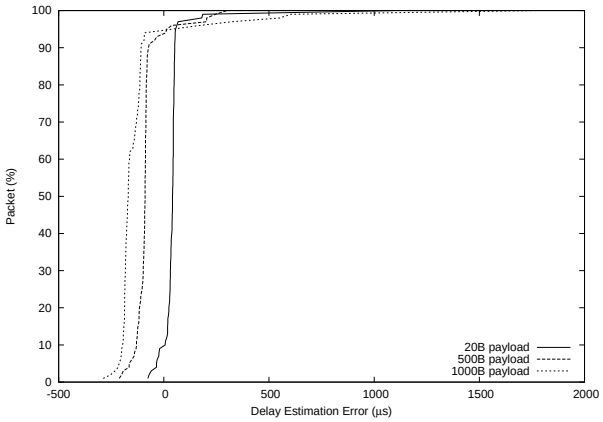
Figure 5.5 displays the same results as figure 5.2 for the three node scenario where the same trend can be observed. In both tests, it stands out that the synchronization protocol provides accurate detection of delay variation between consecutive packets. Such asymmetry would impact the effectiveness of RTT based synchronization techniques.

Figure 5.6 shows the CDF of the error present in the synchronization protocol estimated using packets produced at 100Hz, under the various packet sizes and two hops.

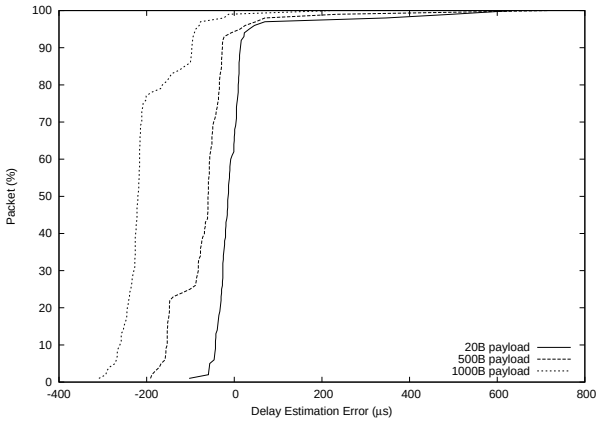
5.2.3 Results Discussion

As a first analysis of the graphics shown in figures 5.2 and 5.5 it is possible to conclude that the synchronization protocol is able to capture delay variation. Each peak registered by the GPS devices is also registered by the synchronization protocol. Tables 5.2 and 5.3 provide information about the error of the synchronization protocol delay estimation. With one hop the error is usually in the order of tenth of microseconds never crossing above 0.7 milliseconds. With two hops the delay varies more widely, which can be explained by the greater medium access competition.

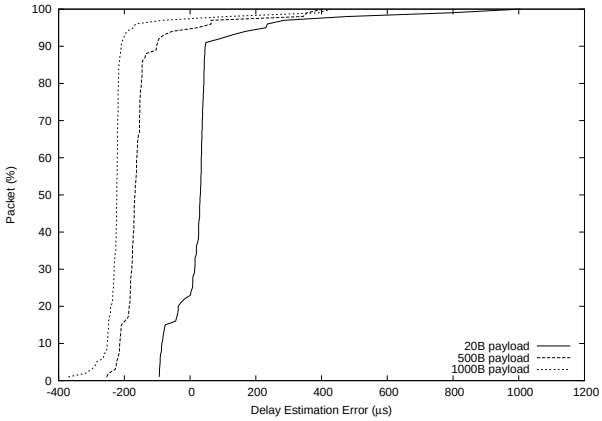
From the CDFs represented in figures 5.3 and 5.6 one can observe that a large percentage of packets have similar delay errors. Also, the delay error is not significantly impacted by the variation of the payload. It is also possible to notice that a small percentage of the packets suffers from larger delay errors. These can be filtered by application in scenarios where the delay is bounded. In general, the protocol is capable of giving a good estimation suffered by packets as they cross the network. The greatest advantage of this protocol is the fact that it can provide one way synchronization.



(a) 1 Mb/s



(b) 5.5 Mb/s



(c) 11 Mb/s

Figure 5.3: Synchronization protocol error CDF using 1Mb/s rate and 100Hz.

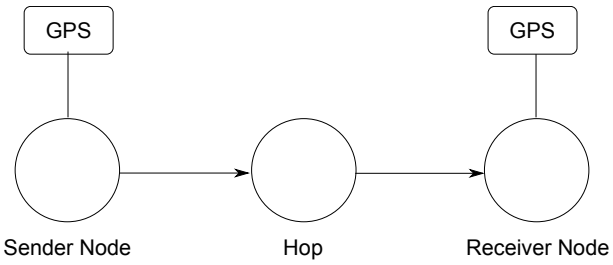
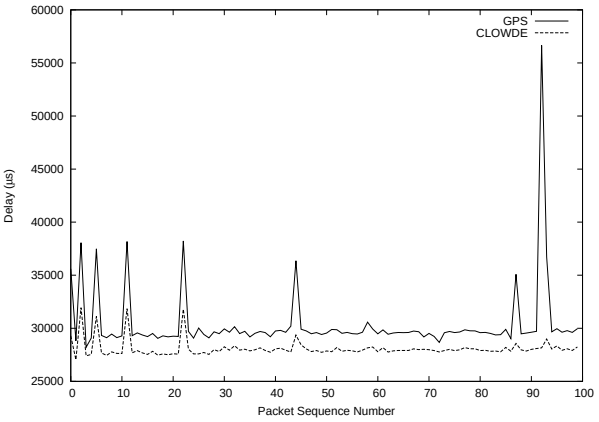
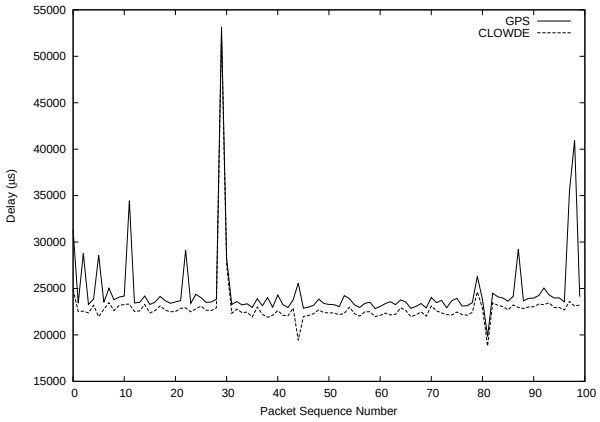


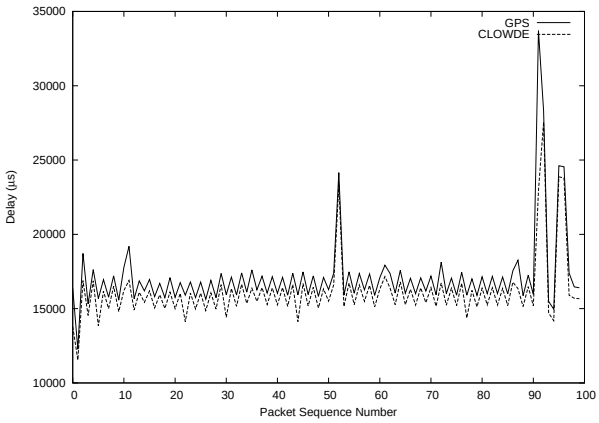
Figure 5.4: Network topology used for the second test scenario.



(a) 1 Mb/s

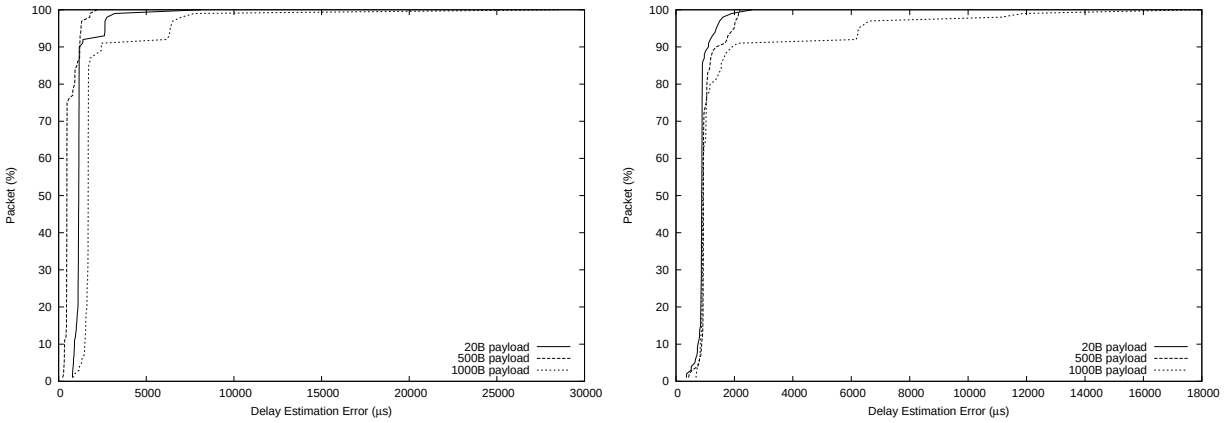


(b) 5.5 Mb/s



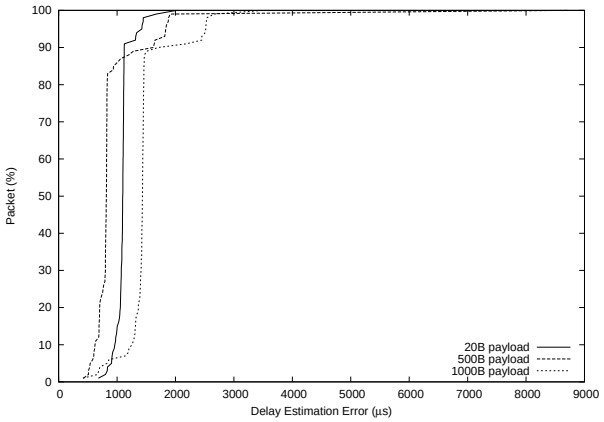
(c) 11 Mb/s

Figure 5.5: Synchronization protocol delay estimation versus GPS delay using 100Hz, 1000B and different transmission rates.



(a) 1 Mb/s

(b) 5.5 Mb/s



(c) 11 Mb/s

Figure 5.6: Synchronization protocol error CDF using 1Mb/s rate and 100Hz.

Table 5.3: Synchronization protocol average delay estimation error with 3 nodes

Transmission rate (Mb/s)	Payload size (B)	Sample rate=100Hz Mean delay (μ)	Delay estimation error (μ/s)
1	20	5604.68	1287.89
	500	19343.80	1913.34
	1000	28084.80	2315.46
5.5	20	4749.30	919.42
	500	13012.80	911.73
	1000	22949.02	1704.05
11	20	4318.88	1102.50
	500	14668.40	1331.00
	1000	16112.70	894.03

5.3 Aggregation Protocol

5.3.1 Performance tests

To study the benefits and hindrances of using aggregation, two different test scenarios have been prepared. The first one analyzes the overhead reduction of each type of aggregation (application versus network level) and the delay caused by the implemented protocol, while the second one analyzes how aggregation can reduce network congestion.

Overhead reduction is computed always relative to the tests with no aggregation, the used formula for overhead reduction was:

$$\frac{\text{total amount of bytes without aggregation} - \text{total amount of bytes with aggregation}}{\text{total amount of bytes without aggregation}} \cdot 100 \tag{5.1}$$

Packet delay was also measured. At the initialization of each packet a time stamp is sent as application payload. Upon receiving, the destination application takes another time stamp and computes the delay as:

$$t_{payload} - t_{application} \tag{5.2}$$

Where $t_{payload}$ is the time inside the packet and $t_{application}$ is the time taken by the destination application. For this purpose receiver and sender nodes must be correctly synchronized. NTP was used for this purpose, because the achieved precision suffices for the obtained delays.

Scenario One

The aim of this test is to assess the overhead introduced by the different aggregation strategies used, using different parameterization and test conditions. Since the interest is to measure aggregation overhead reduction, the tests were executed using a cabled network. This prevents most of packet loss and delays related with wireless protocols enabling more precise results.

The scenario one's network topology is shown in figure 5.7, comprising four nodes: two senders (Node A and Node B), receiver (Node D) and intermediate node (Node C). Node A and Node B send 100 packets to node D, at a rate of 25 Hz. Overhead reduction is measured at node D and computed as shown in equation 5.1.

Different PDU payload's sizes have been tested, as well as different sizes for the aggregated packets. The two different aggregating levels were tested (application and network level) and the results are compared against the results achieved without aggregation.

Three different PDU sizes have been tested - 32, 64 and 128 Bytes. For each PDU size, six different aggregation strategies were tested:

No_Aggreg - without any aggregation, which will serve as a control test.

Apl_Aggreg_640 - with application layer aggregation, with an aggregated packet size of 640 bytes.

Apl_Aggreg_1280 - with application layer aggregation, with an aggregated packet size of 1280 bytes.

Net_Aggreg_320 - with network layer aggregation, with an aggregated packet size of 320 bytes.

Net_Aggreg_680 - with network layer aggregation, with an aggregated packet size of 680 bytes.

Net_Aggreg_1280 - with network layer aggregation, with an aggregated packet size of 1289 bytes.

Application aggregation is always performed in the nodes A and B, while network aggregation is always performed by node C (figure 5.7).

Figure 5.8 demonstrates the obtained delays with no aggregation (**No_Aggreg**) and application aggregation (**Apl_Aggreg_1280**). Only 100 packets are shown since the behavior is the same for the rest of the packets and with 100 packets it is possibly to get a better view of the delay behavior. Figures 5.8(a) and 5.8(b) demonstrate the delay computed at application level with no aggregation and application aggregation, respectively, computed as shown in equation 5.2. Figures 5.8(c) and 5.8(d) show the delay it takes to aggregate and desegregate each packet, respectively.

Table 5.4 shows the number of packets received by the network layer of the receiver node. The first column indicates the PDU size used. Each other column indicates the number of received packets when using a certain aggregation type. Columns with an aggregation type indicate the size of the aggregated packet in the network level. For example, the last column indicates how many packets were received by the receiver's network layer, when application and network level aggregation were in use. Application aggregated packets have 320 Bytes, while network aggregated packet have 1280 bytes.

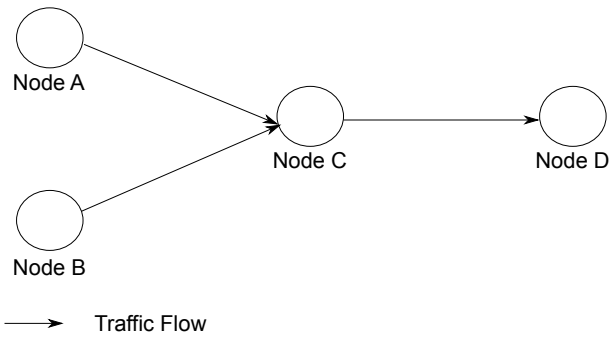
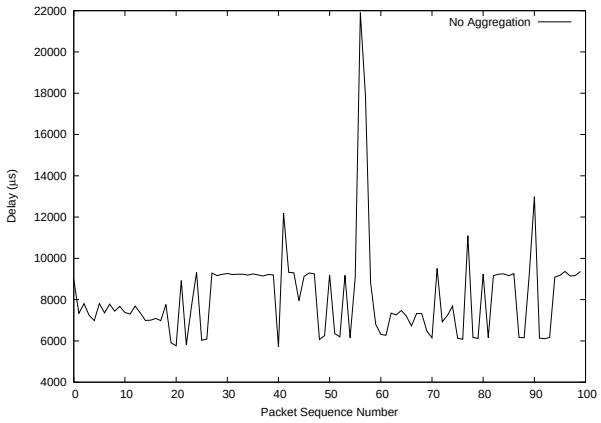
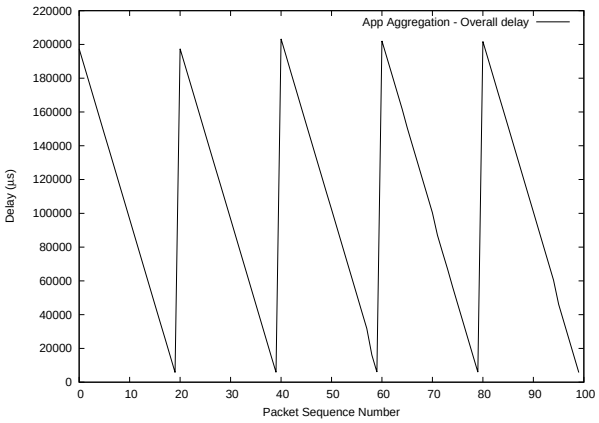


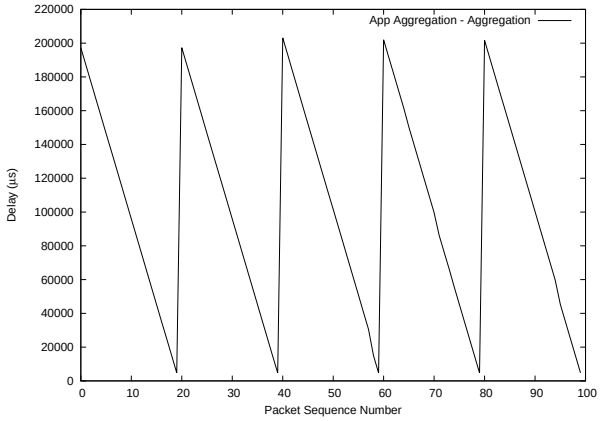
Figure 5.7: Network topology scenario one. Node A and B send data to node D.



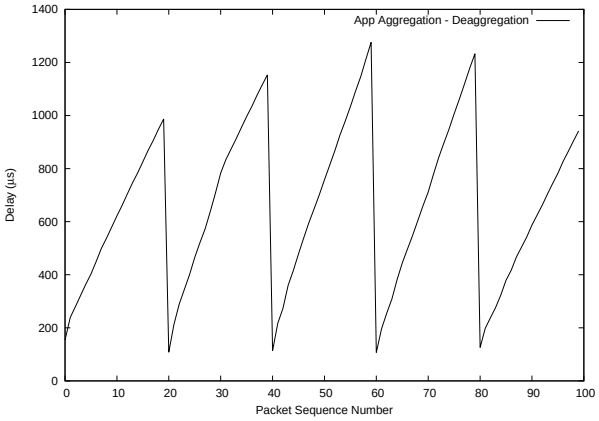
(a) No aggregation delay at 100Hz



(b) Aggregation overall delay at 100Hz



(c) Aggregation delay at 100Hz



(d) Desegregation delay at 100Hz

Figure 5.8: Comparison between aggregation delay and no aggregation delay.

Table 5.4: Number of packets that effectively arrived at the sink node measured in the IP layer. Each node sent originally 100 packets.

Packet Size (Bytes)	Aggregation Type					
	No Aggregation	Application (640 Bytes)	Application (1280 Bytes)	Network (640 Bytes)	Network (1280 Bytes)	Application and Network (320 and 1280 Bytes)
32	200	10	5	20	10	5
64	200	20	10	34	16	10
128	200	40	20	50	25	25

Table 5.5 shows similar information as table 5.4, but instead of counting the number of packets it shows the total amount of bytes received at the data link layer by the receiver node. It also shows the percentage of overhead reduction for better comparison. Overhead reduction is always computed relative to the test where no aggregation was performed. As an example, column two shows the total bytes received by the data link layer and the overhead reduction, when using aggregated packets with 640 bytes.

Scenario Two

Test scenario two evaluates if the aggregation protocol can reduce congestion in the network.

Figure 5.9 shows the network topology used in this scenario.

In test scenario two only 64 Byte sized PDUs were used. Nodes A and C send data to the same destination B. To induce congestion, both senders send 1000 PDUs as quick as possible, that is, there is no predefined sampling rate. The receiver node will log each received PDU so packet loss can be calculated after the test ended. In the wireless medium packet loss may happen basically due to two different causes. Either there were collisions in the wireless medium and packets were corrupted and the sender gave up retransmitting such packets, or the receiver node is to far to be able to receive packets. For the purpose of this test, nodes were placed inside the coverage area of each other to prevent the second cause of packet loss. As for the first cause, collisions in wireless medium are good indication of congestion, therefore measuring packet loss is also a good indication of congestion in the wireless medium.

Three different tests were conducted. First no aggregation (**No_Aggreg**) was used. Then application aggregation was turned on. Finally, application aggregation was turned off and network aggregation turned on. Each application aggregated packet has a total size of 1280 Bytes (**Apl_Aggreg_1280**) as well as network aggregated packets (**Net_Aggreg_1280**). To obtain accurate results each test was repeated several times. Sixty tests were performed in total, twenty for each case.

Figure 5.10 shows the CDF of the packet loss percentage. Each test was performed using PDUs with 64 Bytes.

5.3.2 Results Discussion

As it is clear from the test results in scenario one, the number of packets which arrive at the destination is significantly smaller when compared with the no aggregation scenario. Also, this number depends highly in

Table 5.5: Number of bytes that effectively arrived at the sink node measured in the MAC layer. Each node sent originally 100 packets.

Packet Size (Bytes)	Aggregation Type											
	No Aggregation		Application (640 Bytes)		Application (1280 Bytes)		Network (640 Bytes)		Network (1280 Bytes)		Application (320 and 1280 Bytes)	Application and Network
	Size	Reduction	Size	Reduction	Size	Reduction	Size	Reduction	Size	Reduction	Size	Reduction
32	15000 B	-	6830 B	54.47 %	6615 B	55.90 %	13660 B	8.93 %	13230 B	11.80 %	6615 B	55.90 %
64	21400 B	-	13660 B	36.17 %	13230 B	38.18 %	23222 B	*8.51 %	21168 B	1.08 %	13230 B	38.18 %
128	34200 B	-	27320 B	20.12 %	26460 B	22.63 %	34150 B	0.14 %	33075 B	3.29 %	33075 B	3.29 %

* This value represents an increase in overhead instead of a reduction.

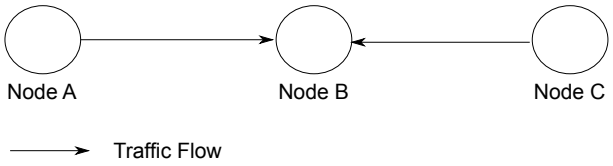


Figure 5.9: Network topology scenario two. Node A and C send data to node B.

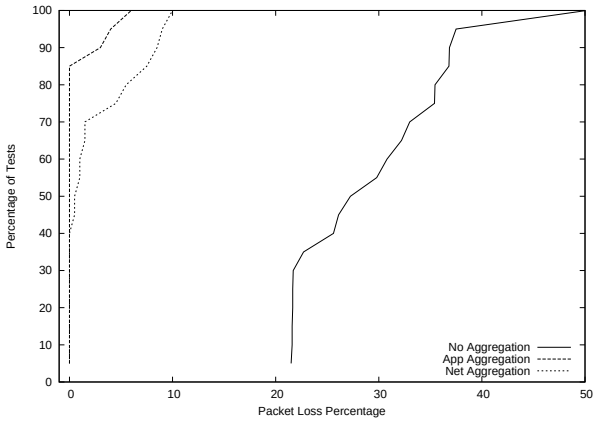


Figure 5.10: CDF of the packet loss percentage.

the aggregated packet size and the PDUs size. In particular, considering the tests where a 32 Byte sized PDU was used, it is clear from table 5.4 that network level aggregation performs poorly when compared to application level aggregation. This can be explained by the headers introduced by the transport and network layer. Because these headers are needed when desegregating the packet, the aggregation protocol needs to conserve them. Therefore, the aggregated packet will carry every PDU plus their transport and network headers.

Turning our attention to table 5.5 it is possible to better understand the numbers provided by table 5.4. Although in most cases the header overhead reduction is small in percentage, the amount of bytes transmitted is significantly smaller. However, there are cases where the aggregation protocol performed poorly. As an example, consider the test where 64 Byte sized PDUs and 640 Byte sized network level aggregation packets were used. In this case the total amount of transmitted bytes is bigger than the case where no aggregation was used. This can be explained by the fact that the aggregated packet size is not a multiple of packet's size. Each PDU has 64 bytes, therefore the network level packet will have 93 Bytes (1 byte introduced by the aggregation protocol, 8 bytes from the UDP header and 20 from the IP header). This results in the last aggregated packet not being complete and some bytes are wasted. Also, using both types of aggregation at the same time helps only when the PDU size is significantly smaller than the aggregated packet.

The overhead reduction is smaller than what would be expected from an aggregation protocol, because the implemented protocol is a generic aggregation technique. Therefore it is not possible to make any assumptions about the data being aggregated, resulting in the fact that every packet needs to be aggregated as it is. However, what the protocol lacks in overhead reduction it compensates for congestion reduction as it is clear from scenario two's results. When aggregation was not used, about 30% of the transmitted packets were lost. However, when aggregation was turned on there were almost no losses. The losses registered in

some tests when using aggregation result in a loss of several PDUs. For this reason and as a proof of concept the implemented prototype is capable of transmitting data using reliable transport, which prevents data from being lost. Also, by combining reliable transport with packet aggregation the network efficiency is improved, since there are lesser losses there will also be fewer retransmissions when compared with a scenario that does not use aggregation.

As expected, with aggregation turned on the obtained delay increases significantly. Moreover, graphic 5.8(b) shows what is expected from the aggregation protocol. As it is clear from it, the last PDU being aggregated is the one which has the smallest delay. In contrast the first one is the one which suffers the largest delay. At 100 Hz each PDU is generated with a period of 10 milliseconds. Since each aggregated packet has room for 20 PDUs it is expected that the first PDU would have a delay at least 19 times bigger than the sampling rate, that is, $10ms \cdot 19 = 190ms$. Graphic 5.8(d) shows that the desegregating procedure is significantly faster than the aggregation procedure (graphic 5.8(c)). For example, the first PDU is desegregated in 200 microseconds. Therefore, the overall overlay observed in graphic 5.8(b) shows that the first PDU suffered a delay of approximately 190 milliseconds, which is the expected value since desegregation does not contribute as much as aggregation for the experienced delays. In fact, by comparing both graphics 5.8(b) and 5.8(c) it is clear that they are very similar, hence it is safe to conclude that the overall delay is tightly connected to the delay imposed by the aggregation procedure. Graphic 5.8(b) also makes it possible to conclude that the receiver is capable of handling desegregation at high sampling rates without having to discard aggregated packets, since queues are never full. Due to software limitations, TS 7500 boards cannot handle a sampling frequency higher than 100 Hz. This happens because the Kernel in such boards counts time in intervals of 10 milliseconds, which corresponds to 100 Hz. Therefore, higher frequencies would have the same effect as 100 Hz, since the Kernel is not capable of counting with a higher granularity. Thus, it is not possible to conclude on the limitations of the aggregation protocol, that is, at which sampling frequency would the receiver node start dropping packets.

The main goal with the developed aggregation protocol was to reduce energy waste caused by congestion in the wireless medium, which led to packet retransmission. Since the obtained results show that most packet loss can be avoided by using the developed protocol it is safe to conclude that fewer retransmissions occur. This leads to the fact that nodes do not waste energy in retransmitting packets and therefore the overall energy waste is reduced.

5.4 Use Cases

5.4.1 Used Methodology

To demonstrate that the platform may be used in both projects presented in section 3, two different tests were conducted. Each one uses aggregation at application level with a buffer of 1024 bytes and synchronization. A total of 100 PDUs were sent, each with 64 bytes. Reliable transport was used and the network had only two nodes - one sender and one receiver.

In the first test, random data was collected and sent at 25 Hz. In the second test, triggered events caused data to be collected and sent. At the time of writing this thesis, the MIT project for people counting was not

fully implemented. Therefore, in order to prove that the platform is able to work with events like a sensor triggering the test program was changed to emulate such events.

The program registers a callback to send data upon receiving a Unix signal. Then, an external program periodically reads a byte from a special device in the file system, which provides random bytes. If this byte is bigger than 128, the program will send the signal to the test program.

Figures 5.11(a) and 5.11(b) show the obtain time given by the synchronization protocol for the first test and second, respectively. In each graphic, time is expressed in microseconds since the first packet was received. No transport control messages were registered.

5.4.2 Results Discussion

As it is clear from the test results in scenario one, every packet was received with a linear time stamp. This is due to the fact that data was collected at a predefined sampling rate. In fact, 25Hz corresponds to 40 milliseconds, thus the last packet arrives exactly 4 seconds latter than packet zero. As opposed to this, the graphic plotted for scenario two demonstrates that events were collected without a constant sampling rate. Figure 5.11(b) shows a behavior that seems like data was collected with a certain periodicity. However, this has to due with the used random procedure, as it is not random as the events which might trigger sensors in the MIT people counting project.

As expected, although application aggregation was used, there is no impact in the synchronization protocol. This has to due with the fact that both were designed to cooperate with each other. As for reliable transport, no control messages was registered, therefore no overhead was noticed during the tests.

5.5 Reliable Transport

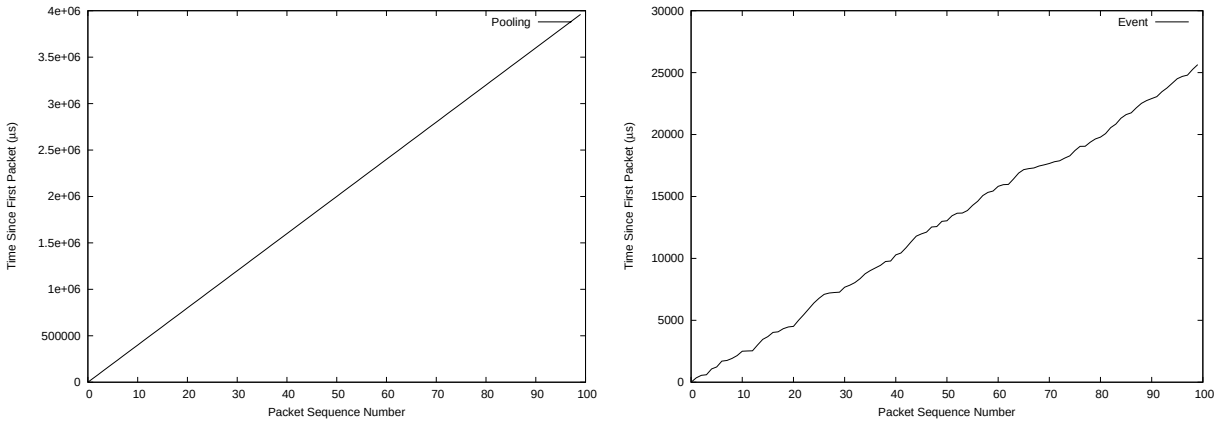
5.5.1 Performance tests

The reliable transport protocol tests serve as a proof of concept. The protocol was designed only to show that the platform supports multiple transport protocols. Therefore the presented tests demonstrate only the control overhead introduced by the developed protocol. Control overhead is measured as the number of control messages (NACK and OSN messages) sent by nodes.

The scenario used to test the prototype is composed of only two nodes. The sender sends 100 packets at three different sampling rates (25 50 and 100 Hz) through cabled network. A special kernel module was built to drop specific packets once. At module insertion time, a list of sequence numbers is specified. The module will drop packets with the specified sequence number the first time they arrive at the destination. Each receiving buffer has space for 5 packets, meaning that a NACK will at most request 5 packets. Table 5.6 shows the obtained results for these tests. Each column shows the number of control messages for the following cases:

No drops - no sequence numbers were specified to the drop module and there were no NACKs sent.

3 contiguous drops - 3 sequence numbers were specified to the drop module. These numbers are all contiguous, such as 10, 11, 12.



(a) Sampling at 25 Hz

(b) Event generated

Figure 5.11: Use case tests for both MIAVITA and MIT project.

7 contiguous drops - 7 sequence numbers were specified to the drop module. These numbers are all contiguous, such as 10, 11, 12, 13, 14, 15, 16. This scenario will ensure that at least two NACKs should be sent.

3 intercalated drops fitting into a NACK - 3 sequence numbers were specified to the drop module. These numbers are intercalated, but the difference between the smallest and largest one is less than the size of the receiving buffer. For example 10, 12, 14.

3 intercalated drops not fitting into a NACK - 3 sequence numbers were specified to the drop module. These numbers are intercalated, but the difference between one and another is greater than the size of the receiving buffer. For example 10, 17, 64.

5.5.2 Results Discussion

As it is clear from table 5.6, a negative acknowledgment protocol generates far less control traffic than a positive acknowledgment protocol. This happens because in positive acknowledgment almost every message is acknowledge by the receiver. Therefore, the first case scenario where no packets were dropped does not generate control traffic.

The tests where 3 and 7 contiguous drops were forced, generated the expected NACK messages. In the first test one NACK message was sent, since 3 contiguous sequence numbers fit into the receiving buffer.

Table 5.6: Reliable protocol message overhead.

Sampling rate	No drops		3 contiguous drops		7 contiguous drops		3 intercalated drops fitting into a NACK		3 intercalated drops not fitting into a NACK	
(Hz)	NACK	OSN	NACK	OSN	NACK	OSN	NACK	OSN	NACK	OSN
25	0	0	1	0	2	0	3	0	3	0
50	0	0	1	0	2	0	3	0	3	0
100	0	0	1	1	2	1	3	1	3	2

However, in the second test this is not true and therefore two NACK messages have been sent. It is also possible to notice that at a sampling rate of 100 Hz both tests generated an OSN message. This has to due with the fact that the sender produces messages too fast and when a NACK was sent by the receiver some packets were no longer available in the retransmission buffer. This results in some losses, which can be avoided if the retransmission buffer was bigger.

In case where 3 intercalated drops, which fit into the receiving buffer were generated, 3 NACKs have been sent. Although one might think that a single NACK would suffice, the way the protocol was designed corresponds exactly to the observed results. Since the protocol works by examining the receiving buffer for gaps in the sequence number, it will encounter exactly 3 gaps - the intercalated sequence numbers. Therefore, 3 NACK messages will be sent. As in with the previous tests, this one generates OSN messages at 100 Hz sampling rate. This is also related with the retransmission buffer size. Finally, the last test case generates 3 NACK messages exactly for the same reason. It also generates 2 OSN messages at 100 Hz for the same reason.

In sum, the tests demonstrate what was expected from the designed protocol. They also demonstrate that when using aggregation the receiving buffer size should have enough space for at least one aggregated packet. If it has less space, then in the case where an aggregated packet gets lost more than one NACK message will be generated when one would be enough. Moreover, the protocol performs better when lost packets have contiguous sequence numbers, since the probability of sending only one NACK is far greater than when lost packets have intercalated sequence numbers.

6 Conclusion

This thesis main goal was to develop a multi-functional platform for indoor and outdoor monitoring, which could be easily extended and configurable to different scenarios. A study in the current state of the art of the project's application domain has revealed a set of properties common to both indoor and outdoor monitoring systems. In particular, this thesis focused in WSNs which perform this kind of monitoring and should be power-aware and need to synchronize their data. Therefore, synchronization and in-network aggregation protocols were developed, which had this in consideration. A modular architecture was designed to provide extendability to new features and ease the implementation of both synchronization and aggregation protocols. Current solutions for both protocols revealed to be too specific and therefore difficult to port to other applications. A more generic approach was necessary to provide these functionalities to both indoor and outdoor monitoring systems. Although, a generic aggregation protocol was also analyzed in the state of the art it revealed to have a critical flaw. In this protocol, packets were aggregated in the MAC sub-layer, which forced intermediate nodes to desegregate them in order to correctly route them. This approach causes an unnecessary overload in the WSN, thus the designed aggregation protocol was planned to be generic without having the same issue.

The implemented prototype was tested in two different projects - MIAVITA project and MIT People Counting and Detection of Patterns of Movement. These projects are ideal for testing such prototype since they have different goals. The first one monitors volcanic activity in remote locations, while the other performs monitors people movement inside buildings. The MIAVITA project WSN acquires data from several sensors and sends it to a sink node, which in turn will send it to a base station. Each packet needs to be synchronized. At the present time, the MIAVITA project has no known solution which avoids energy wasting. Nevertheless, it has become a concern and therefore the developed aggregation protocol was tested in MIAVITA with this goal in mind as well. As for the MIT's project it too requires synchronization to avoid conflicts in collected data, however with lesser precision than MIAVITA's project. Aggregation was not tested in this scenario, since WSNs in this project are powered by the buildings electrical network and therefore do not require a solution to avoid energy wasting. One important conclusion achieved with these two projects is that although the developed architecture is capable of supporting both scenarios, it is crucial that it also supports configurability to enable the activation and deactivation of supported services.

Tests to the prototype revealed satisfactory results. The synchronization protocol revealed to be able to synchronize packets in the order of few milliseconds. The designed solution synchronizes data without the use of acknowledgements from other nodes nor broadcast beacons. Therefore it avoids wasting energy that would be used to transmit such frames. As for the aggregation protocol, tests demonstrate that although it provides little reduction in the overhead of transmitted packets it is capable of minimizing congestion in the wireless medium. Therefore, less collisions occur, preventing nodes from retransmitting corrupted frames and thus not wasting energy associated with these retransmissions. Aggregation has shown that a need for reliable transport may arise, since losing an aggregate packet will end in loosing several application packets. As a proof of concept of the platforms configurability a reliable transport protocol was implemented in the prototype, which can be used if applications so require. Both protocols can be used together enabling the use of both techniques in the same scenario.

As for future work, the developed architecture provides a design which can easily be extended. New

interceptors are easy to develop and integrate with the platform, without having to change previous work. Some features present in some WSN can be implemented by means of interceptors. As examples, data compression and security may be implemented as interceptors. As data leaves the sender node, it can be compressed or ciphered and when it arrives at its destination, it may be decompressed or deciphered. Security may also involve privacy, in which case data may also be checked by the security interceptor for sensible information. Another feature left for future work is dynamic adaptability. In monitoring systems, WSNs usually sample data from a specific event. This data may be crucial depending on several factors related with the monitoring application. Crucial data should not be lost and should be delivered as soon as possible to the sink node. Therefore it should be possible to provide means for an application to tell the platform it needs to start or stop sending crucial data. The platform would need to adapt itself by reconfiguring traffic interceptors. Although interceptors were designed to be dynamically inserted and removed, no module which can perform this was developed. Therefore, perhaps the next step would be to provide such module to applications, which would alert this module when they have crucial data to send.

As for the developed protocols, the synchronization algorithm may be moved to lower layers in the OSI stack to account for more delays related with the layers' buffering mechanisms. Also the USB delay estimation should also be improved to account for multiple transfers. The aggregation protocol may be improved to reduce more the overhead introduced by the network and transport headers. As a final remark, every test was conducted inside a controlled environment, thus field tests should be perform to provide more information about challenges that a controlled environment is not capable of offering.

Bibliography

[1] LaMarca, A., Chawathe, Y., Consolvo, S., Hightower, J., Smith, I., Scott, J., Sohn, T., Howard, J., Hughes, J., Potter, F., et al.: Place lab: Device positioning using radio beacons in the wild. Pervasive Computing (2005) 301–306

[2] Want, R., Hopper, A., Falcão, V., Gibbons, J.: The active badge location system. ACM Transactions on Information Systems (TOIS) **10**(1) (1992) 91–102

[3] Bahl, P., Padmanabhan, V.: Radar: An in-building rf-based user location and tracking system. In: INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. Volume 2., Ieee (2002) 775–784

[4] Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., et al.: Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In: Distributed Computing Systems, 2004. Proceedings. 24th International Conference on, IEEE (2004) 582–589

[5] Burrell, J., Brooke, T., Beckwith, R.: Vineyard computing: sensor networks in agricultural production. IEEE Pervasive computing (2004) 38–45

[6] Xie, W., Shi, Y., Xu, G., Mao, Y.: Smart platform-a software infrastructure for smart space (siss). In: Multimodal Interfaces, 2002. Proceedings. Fourth IEEE International Conference on, IEEE (2002) 429–434

[7] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J., Flinn, J., Walker, K.: Agile application-aware adaptation for mobility. In: ACM SIGOPS Operating Systems Review. Volume 31., ACM (1997) 276–287

[8] Sousa, J., Garlan, D., et al.: Aura: an architectural framework for user mobility in ubiquitous computing environments. In: Proceedings of the IFIP 17th World Computer Congress-TC2 Stream/3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, Citeseer (2002) 29–43

[9] Roman, M., Hess, C., Ranganathan, A., Madhavarapu, P., Borthakur, B., Viswanathan, P., Cerqueira, R., Campbell, R., Mickunas, M.: Gaiaos: An infrastructure for active spaces. (2001)

[10] Zhou, J., De Roure, D.: Floodnet: coupling adaptive sampling with energy aware routing in a flood warning system. Journal of Computer Science and Technology **22**(1) (2007) 121–130

[11] Levis, P., Culler, D.: Maté: a tiny virtual machine for sensor networks. In: ACM Sigplan Notices. Volume 37., ACM (2002) 85–95

[12] Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using kairos. Distributed Computing in Sensor Systems (2005) 126–140

[13] Liu, T., Martonosi, M.: Impala: a middleware system for managing autonomic, parallel sensor systems. In: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM (2003) 107–118

[14] Pottie, G.: Wireless sensor networks. In: Information Theory Workshop, 1998, IEEE (2002) 139–140

[15] Satyanarayanan, M.: Pervasive computing: Vision and challenges. IEEE Personal Communications **8** (2001) 10–17

[16] Weiser, M.: The computer for the 21st century. Scientific American **272**(3) (1995) 78–89

[17] Schilit, B., Adams, N., Gold, R., Tso, M., Want, R.: The parctab mobile computing system. In: Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on, IEEE (1993) 34–39

[18] Estrin, D., Culler, D., Pister, K., Sukhatme, G.: Connecting the physical world with pervasive networks. Pervasive computing (2002) 59–69

[19] Puccinelli, D., Haenggi, M.: Wireless sensor networks: applications and challenges of ubiquitous sensing. Circuits and Systems Magazine, IEEE **5**(3) (2005) 19–31

[20] Shi, E., Perrig, A.: Designing secure sensor networks. Wireless Communications, IEEE **11** (2004) 38–43

[21] Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer networks **38**(4) (2002) 393–422

[22] Ganesan, D., Govindan, R., Shenker, S., Estrin, D.: Highly-resilient, energy-efficient multipath routing in wireless sensor networks. SIGMOBILE Mob. Comput. Commun. Rev. **5** (2001) 11–25

[23] Haenggi, M.: Energy-balancing strategies for wireless sensor networks. In: Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on. Volume 4., IEEE (2003) IV–828

[24] Younis, O., Fahmy, S.: Heed: a hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. IEEE Transactions on Mobile Computing (2004) 366–379

[25] Hoblos, G., Staroswiecki, M., Aitouche, A.: Optimal design of fault tolerant sensor networks. In: Control Applications, 2000. Proceedings of the 2000 IEEE International Conference on, IEEE (2000) 467–472

[26] Demirkol, I., Ersoy, C., Alagoz, F.: Mac protocols for wireless sensor networks: a survey. Communications Magazine, IEEE **44** (2006) 115–121

[27] Mills, D.: Internet time synchronization: The network time protocol. Communications, IEEE Transactions on **39** (2002) 1482–1493

[28] Elson, J., Römer, K.: Wireless sensor networks: A new regime for time synchronization. ACM SIGCOMM Computer Communication Review **33**(1) (2003) 149–154

[29] Sichitiu, M., Veerarittiphan, C.: Simple, accurate time synchronization for wireless sensor networks. In: Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE. Volume 2., IEEE (2003) 1266–1273

[30] Römer, K.: Time synchronization in ad hoc networks. In: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing, ACM (2001) 173–182

[31] Elson, J., Estrin, D.: Time synchronization for wireless sensor networks. (2001)

[32] Hart, J., Martinez, K.: Environmental sensor networks: A revolution in the earth system science? Earth-Science Reviews **78**(3-4) (2006) 177–191

[33] Martinez, K., Hart, J., Ong, R.: Environmental sensor networks. Computer **37** (2004) 50–56

[34] Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., Mainwaring, A., Estrin, D.: Habitat monitoring with sensor networks. Communications of the ACM **47**(6) (2004) 34–40

[35] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (1978) 558–565

[36] Elson, J., Girod, L., Estrin, D.: Fine-grained network time synchronization using reference broadcasts. ACM SIGOPS Operating Systems Review **36**(SI) (2002) 147–163

[37] Maróti, M., Kusy, B., Simon, G., Lédeczi, Á.: The flooding time synchronization protocol. In: Proceedings of the 2nd international conference on Embedded networked sensor systems, ACM (2004) 39–49

[38] Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., Welsh, M.: Fidelity and yield in a volcano monitoring sensor network. In: Proceedings of the 7th symposium on Operating systems design and implementation, USENIX Association (2006) 381–396

[39] Xu, N., Rangwala, S., Chintalapudi, K., Ganesan, D., Broad, A., Govindan, R., Estrin, D.: A wireless sensor network for structural monitoring. In: Proceedings of the 2nd international conference on Embedded networked sensor systems, ACM (2004) 13–24

[40] Fasolo, E., Rossi, M., Widmer, J., Zorzi, M.: In-network aggregation techniques for wireless sensor networks: a survey. Wireless Communications, IEEE **14**(2) (2007) 70–87

[41] Madden, S., Franklin, M., Hellerstein, J., Hong, W.: Tag: a tiny aggregation service for ad-hoc sensor networks. ACM SIGOPS Operating Systems Review **36**(SI) (2002) 131–146

[42] Yao, Y., Gehrke, J.: The cougar approach to in-network query processing in sensor networks. SIGMOD Rec. **31**(3) (2002) 9–18

[43] Sharaf, M., Beaver, J., Labrinidis, A., Chrysanthis, P.: Balancing energy efficiency and quality of aggregate data in sensor networks. The VLDB journal **13**(4) (2004) 384–403

[44] Cayirci, E.: Data aggregation and dilution by modulus addressing in wireless sensor networks. Communications Letters, IEEE **7**(8) (2003) 355–357

[45] He, T., Blum, B., Stankovic, J., Abdelzaher, T.: Aida: Adaptive application-independent data aggregation in wireless sensor networks. ACM Transactions on Embedded Computing Systems (TECS) **3**(2) (2004) 426–457

[46] Wang, C., Sohraby, K., Li, B., Daneshmand, M., Hu, Y.: A survey of transport protocols for wireless sensor networks. Network, IEEE **20**(3) (2006) 34–40

[47] Hull, B., Jamieson, K., Balakrishnan, H.: Mitigating congestion in wireless sensor networks. In: Proceedings of the 2nd international conference on Embedded networked sensor systems, ACM (2004) 134–147

[48] Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer networks **38**(4) (2002) 393–422

[49] Sankarasubramaniam, Y., Akan, Ö., Akyildiz, I.: Esrt: event-to-sink reliable transport in wireless sensor networks. In: Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing, ACM (2003) 177–188

[50] Park, S., Vedantham, R., Sivakumar, R., Akyildiz, I.: A scalable approach for reliable downstream data delivery in wireless sensor networks. In: Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing, ACM (2004) 78–89

[51] Santos, L.: Sos net: Rede de emergência para gestão de risco ambiental. Master's thesis, Instituto Superior Técnico, Portugal (2008)