



M 2014



FIXED SENSORS INTEGRATION FOR FUTURE CITIES USING M2M

ANDRÉ DA SILVA E SÁ

DISSERTAÇÃO DE MESTRADO APRESENTADA

À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM

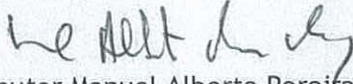
Engenharia Eletrotécnica e de Computadores

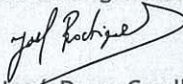
A Dissertação intitulada

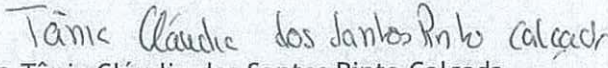
“Fixed Sensors Integration for Future Cities Using M2M”

foi aprovada em provas realizadas em 09-10-2014

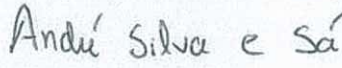
o júri


Presidente Professor Doutor Manuel Alberto Pereira Ricardo
Professor Associado do Departamento de Engenharia Eletrotécnica e de
Computadores da Faculdade de Engenharia da Universidade do Porto


Professor Doutor Joel José Puga Coelho Rodrigues
Professor Auxiliar do Departamento de Informática da Universidade da Beira Interior


Doutora Tânia Cláudia dos Santos Pinto Calçada
Investigador Auxiliar do Departamento de Engenharia Eletrotécnica e de
Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.


Autor - André da Silva e Sá

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Fixed Sensors Integration for Future Cities Using M2M

André da Silva e Sá

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Dra. Tânia Calçada

Second Supervisor: Prof. Dra. Susana Sargento

October 14, 2014

Abstract

The actions of today will shape the future of our cities. Understanding how cities' dynamics are changing and taking actions to improve citizens' life quality is imperative in our rapidly urbanised world. This dissertation arises in the context of the European project *Future Cities* that aims to turn the city of Porto into an *urban-scale living lab* where researchers and companies can test new technologies, in order to improve citizens' quality of life. In the scope of the *Future Cities* project, sense units based on the open-source Raspberry Pi platform were sparsely distributed across the city to measure environmental air parameters.

The data gathered by the sensors need to be transferred to the cloud, preferably taking advantage of existent local infrastructures instead of cellular networks. The data processing does not have real time constraints, which makes it possible to be transferred in a opportunistic way using the municipal Wi-Fi hotspots around the city or the vehicular network deployed in six hundred vehicles as part of the same project.

The goal of this dissertation is to design and implement a solution to opportunistically transfer sensors data from sense units to the cloud using the vehicular network. This communication scenario is widely known as Delay Tolerant Network (DTN) scenario. Moreover, the bundle protocol, specified by the Internet Engineering Task Force (IETF), addresses the issues of communicating in DTNs. The proposed architecture uses the IBR-DTN, an open-source implementation of the bundle protocol, as a framework to able the developed applications to send sensors data through the DTN. All devices involved in sensors data opportunistic transmission uses the same communication protocol.

Resumo

O futuro das cidades é definido pelas ações que tomámos no presente. Num mundo cada vez mais urbanizado, é importante perceber a alteração da dinâmica das cidades e procurar soluções para melhorar a qualidade de vida dos cidadãos. Esta dissertação surge no contexto do projeto Europeu *Cidades do Futuro*, cujo objetivo é transformar a cidade do Porto num *laboratório vivo à escala urbana* onde será possível testar novas tecnologias que permitirão melhorar a qualidade de vida dos cidadãos. No âmbito do projeto *Future Cities*, foram instalados de forma distribuída por toda a cidade diversas unidades de sensores baseadas na plataforma livre Raspberry Pi para medir os parâmetros ambientais do ar.

Os dados adquiridos pelos sensores têm que ser transferidos para a nuvem usando preferencialmente as infraestruturas locais existentes, em vez das redes celulares. Devido à inexistência de restrições de processamento dos dados em tempo real, pode-se transferir os dados de forma oportunista usando os pontos de acesso Wi-Fi existentes na cidade ou através da rede veicular que foi instalada em seiscentos veículos, no âmbito do mesmo projeto.

O objetivo desta dissertação é desenhar e implementar uma solução para transferir os dados dos sensores para a nuvem de forma oportunista, usando a rede veicular. Este cenário de comunicação é amplamente conhecido como sendo uma rede tolerante a atrasos. Além disso, o IETF especificou o protocolo *Bundle* que procura resolver os problemas inerentes à comunicação em redes tolerantes a atrasos. A arquitetura proposta utiliza o protocolo IBR-DTN, que pode ser utilizado de forma livre e implementa o protocolo *Bundle*, como uma ferramenta que possibilita às aplicações desenvolvidas o envio dos dados dos sensores através de redes tolerantes a atrasos. Todos os dispositivos envolvidos no envio dos dados dos sensores de forma oportunista utilizam o mesmo protocolo de comunicação.

Acknowledgements

It is a pleasure to show my gratitude to those who made this "final step" possible.

Firstly, I would like to thank to Dra. Tânia Calçada and Prof. Dra. Susana Sargento, who gave me the opportunity to do my dissertation in such an interesting project. An opportunity that has certainly changed my academic path in a very positive way. Their motivation, knowledge and comprehension were important factors to me in reaching this point.

I could not forget the friends and colleagues I have had since my first day at University, who studied and worked by my side, creating a stimulating and fun environment to grow and learn.

Also the many great professors, who shared their knowledge and experience, making Electrical and Computing Engineering subjects even more interesting.

Lastly, and most importantly, I would like to show my love and gratitude to my girlfriend and to my family, especially my parents and my brother, who have always supported me in everything I have needed. Here I also include my closer, lifetime friends, who have followed me in every moments of my life. You can count on me for everything you need.

André da Silva e Sá

"Haste is the enemy of perfection"

Portuguese saying

Contents

1	Introduction	1
1.1	Contextualisation	1
1.2	Motivation and Problem Characterisation	2
1.3	Goals	3
1.4	Results	3
1.5	Document Structure	4
2	<i>Future Cities</i> Project	5
2.1	Overview	5
2.2	<i>UrbanSense</i> Platform	6
2.2.1	Sense Units	6
2.2.2	Cloud <i>UrbanSense</i> Server and Database	7
2.3	Vehicular Network	8
2.4	Summary	9
3	State-of-Art	11
3.1	The Mule Architecture	11
3.2	DTN Architecture	13
3.2.1	DTN Architectural Principles	14
3.2.2	Bundle Security Protocol	17
3.2.3	<i>Bundle Protocol</i>	18
3.2.4	Convergence Layer Protocols	21
3.3	DTN Implementations	21
3.3.1	DTN-2	22
3.3.2	IBR-DTN	24
3.3.3	Implementations' Comparison	26
3.4	Summary	27
4	The Proposed Architecture	29
4.1	System Overview	29
4.2	Opportunistic Communications	30
4.3	Types of Data	32
4.3.1	Sensors Data	32
4.3.2	Acknowledgement data	33
4.4	Sensor Samples Management	33
4.5	Summary	34

5	Implementation	37
5.1	Implementation Details	37
5.2	IBR-DTN Application Programming Interface (API)	38
5.2.1	Bundle Register	38
5.2.2	Commands	39
5.2.3	Parsing Bundles from IBR-DTN API	41
5.3	<i>UrbanSense</i> Sense Units Application	41
5.3.1	Sense Units Clock Synchronisation	42
5.3.2	Sense Unit and On Board Unit (OBU) Connection Test	42
5.3.3	Sensors Data Transmission	43
5.4	<i>UrbanSense</i> Server Application	45
5.5	Summary	47
6	Measurements and Analysis	49
6.1	Controlled Environment Tests Scenarios	49
6.2	Real-World Environment Applications Validation	51
6.3	Results	52
6.3.1	Bundles Delay	52
6.3.2	Bundles Transmissions	53
6.3.3	Real-World Environment Test Overview	54
6.4	Summary	57
7	Conclusions and Future Work	59
7.1	Future Work	60
A	IBR-DTN Configuration File	61
A.1	Sense Units IBR-DTN configuration file	61
	References	65

List of Figures

2.1	Sense Unit Platform Components	6
2.2	Sense Unit Platform Real-Image	7
2.3	a) OBU Device Real-Image. b) Road Side Unit (RSU) Device Installed on a Traffic Light Real-Image	8
2.4	<i>Future Cities</i> Project Network Devices	9
3.1	The Three Layers of the Mule Architecture	12
3.2	Store and forward operation: a) node B getting close node A, b) opportunistic contact, c) node B moving around, and d) data delivering to its destination	14
3.3	<i>Bundle Protocol</i> Architecture	19
3.4	DTN2 Daemon Architecture	22
3.5	IBR-DTN Daemon Architecture	24
4.1	Opportunistic Communications Proposed Architecture	31
4.2	Sensor Samples Structure	32
4.3	Sensors Metadata Structure	33
4.4	Sensors Acknowledgement Data Structure	33
5.1	Sense Unit and OBU Connection Test	43
5.2	Sense Units Application Flowchart	45
5.3	Cloud <i>UrbanSense</i> Server Application Flowchart	46
6.1	First Controlled Environment Test Scenario	50
6.2	Second Controlled Environment Test Scenario	51
6.3	Real-Word Environment Test Scenario	51
6.4	Bundles Delay Histogram	53
6.5	Bundles Number of Transmission Distribution	54
6.6	Real-Word Environment Test Overview	55
6.7	Opportunistic Contacts Mean Delay	56

List of Tables

3.1	Bundle Primary Block	21
3.2	DTN2 and IBR-DTN RFC 5050 Features Comparison	26
4.1	Sensor Samples Management Table	34
5.1	IBR-DTN Time Synchronisation Configuration	42
6.1	Sense Unit Application Configuration Parameters	52
6.2	Bundles Delay Information	53
6.3	Real-World Environment Test Details	55
6.4	Opportunistic Contacts Amount of Transferred Data, Mean Delay and Delay Standard Deviation	56
6.5	Quantity of Bundles per Bundles Amount of Sensors Data	57

Abbreviations

ADU	Application Data Unit
API	Application Programming Interface
BAB	Bundle Authentication Block
BD	Bundle Daemon
BSR	Bundle Status Report
CB	Confidentiality Block
DHCP	Dynamic Host Configuration Protocol
DTN	Delay Tolerant Network
DTNRG	Delay Tolerant Networking Research Group
EID	Endpoint Identifier
GPS	Global Position System
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPND	IP Neighbour Discovery
IRTF	Internet Research Task Force
NTP	Network Time Protocol
OBU	On Board Unit
PDU	Protocol Data Unit
PSB	Payload Security Block
QoS	Quality of Service
RSU	Road Side Unit
SDNV	Self-Delimiting Numeric Value
STCP	Service of Transport, City of Porto
TCP	Transmission Control Protocol

UDP User Datagram Protocol

URI Uniform Resource Identifier

V2V Vehicle-to-Vehicle

Chapter 1

Introduction

1.1 Contextualisation

The intense concentration of people and companies in cities brings new challenges in cities' management and economy. The predictions are that by 2050, 67% of the World population will be urban (United Nations, 2011). This global trend is leading to the cities' air quality and people' mobility deterioration, as well as the increase of stress and noise. This dissertation arises in the context of the European project *Future Cities* intended to turn the city of Porto into an *urban-scale living lab*, where researchers and companies can test technologies, products and services, exploring subjects such as *sustainable mobility*, *urban-scale sensing*, as well as citizens' *quality of life* improvement. To accomplish that, three platforms have been implemented: (1) a *crowdsensing* smartphone application, which gather citizens' mobility and wearable clothes sensors data, (2) a vehicular network, in which six hundred vehicles including buses, taxis and trucks communicate among them and to the cloud, and (3) an *UrbanSense* platform, which is composed of sense units sparsely installed around the city of Porto and mobile sense units installed on-board the city of Porto transport service buses. This dissertation work focus on the *UrbanSense* platform and the vehicular network integration, in order to transfer fixed sense units data to the cloud.

The appearance of low cost computing devices with wireless network capabilities is making *urban-scale sensing* a reality, from a device perspective. However, from the network point of view, there are still many challenges. As an example, measuring and analysing the air quality in an urban-scale requires installing hundreds of sparsely distributed sensors around the city to measure relevant air parameters. Consequently, it is required to transfer the acquired sensors data to the cloud for later processing. Depending on each sensor location, there might be more than one ways of doing that. We analyse the most cost/benefit appealing solutions to accomplish that goal.

One possible solution could be using cellular technologies. Installing a GSM module in each node may be a simple solution from the installation point of view. However, due to the large

amount of sense units, the amount of data that needs to be transferred and the monthly fee that telecommunications companies charge for that kind of services, make it costly unattractive as a global solution. Other possible solution is to use the IEEE 802.15.4 standard. Particularly when sensors nodes have low power consumption constrains, using this standard may be a good solution. However, the maximum distance between nodes is not enough to highly sparse scenarios, requiring a great number of gateways to do the interface between the sensors and the cloud. Other possible solution is to use the IEEE 802.11b/g/n standard and send sensors data to the cloud through the city of Porto Internet hotspots. Although some of them are paid, its utilisation price has been decreasing lately. Other possible solution is to opportunistically transfer sensors data to the cloud through passing vehicles. Taking into account that the city of Porto has a vehicular network already installed, this is an appealing solution.

1.2 Motivation and Problem Characterisation

This dissertation work tackles the issues of using the city of Porto's vehicular network as a solution to reliably transfer sensor data to the cloud *UrbanSense* database. The way data is acquired and stored in the local database, how it goes through the vehicular network and reach the Internet, how sensors data is processed and uploaded to the cloud *UrbanSense* database, as well as all issues regarding mobile sense units are beyond the scope of this dissertation. The challenges we face up are: (1) detect opportunistic contacts and establish a connection between fixed sense units and vehicular network nodes; (2) send sensors data and receive cloud *UrbanSense* server acknowledgement data, through the vehicular network; and (3) manage sense units database, based on sensors data uploading to cloud *UrbanSense* database success or a failure.

First, fixed sense units have to detect the vehicular network announced SSID, configured in managed mode, and transfer sensors data to it through the IEEE 802.11b/g/n interface. Furthermore, they may also receive acknowledgement data from the vehicular network. Second, when sensors data reaches the vehicular network "border", an application in the cloud *UrbanSense* server has to extract sensors data from the vehicular network and send it to an application, which accepts Transmission Control Protocol (TCP) socket connections and process sensors data. Third, that application uploads sensors data to the cloud *UrbanSense* database and answers acknowledgement data that needs to be sent to the sense unit that originated that data, through the vehicular network. Final, the acknowledgement data has to be processed by the sense unit and sensors data successfully uploaded to the cloud *UrbanSense* database has to be deleted from sense unit local database. This network scenario is widely known as a Delay Tolerant Network, which is characterised by several minutes or hours delays and frequent network partitions that may cause an end-to-end inexistent contemporaneous connection path.

1.3 Goals

The communication problem we have to solve has many tricky aspects involved, which are addressed by the DTN architecture. Especially, the way fixed and mobile nodes detect the opportunistic contact and establish a connection, dealing with unexpected connection interruptions, data fragmentation for improved efficiency, as well as end-to-end reliability and security. The DTN architecture defines a new layer on top of the transport layer and below of the application layer, called bundle layer to address the DTN issues. There are several bundle layer open-source protocols implementations [1][2][3][4]. These open-source implementations can be used as frameworks to make applications the ability to communicate through DTN enabled network nodes.

First, this dissertation goal is to use the studies around DTN to integrate the *UrbanSense* and vehicular network platforms, in order to solve the communication problem between sense units and the cloud *UrbanSense* server. Second, it is necessary to develop the necessary applications upon a bundle layer open-source implementation, to reliably transfer sensors data to the cloud *UrbanSense* server, receive acknowledgement data from cloud *UrbanSense* server and manage the sense units local database according to sensors data uploading to cloud *UrbanSense* database success or failure. Final, metadata has to be added to data transferred in sense units to cloud *UrbanSense* server direction, to make the ability to understand the overall system performance through metadata analysis.

1.4 Results

This dissertation work specifies an architecture to reliably send sensors data from sense units to cloud *UrbanSense* server using the city of Porto's vehicular network. The proposed architecture relies on the IBR-DTN protocol utilisation as a framework, in order to developed applications communicate through the DTN, as well as an application running on the cloud *UrbanSense* server that accepts TCP socket connection, receives sensors data, uploads it to the cloud *UrbanSense* database and answers acknowledgement data about sensors data uploading success or failure.

Metadata information is added to sensor samples, in order to understand the overall system performance. Statical metadata analysis, permits to understand the delay since bundles leave sense units until reach the cloud *UrbanSense* database, the retransmitted sensor samples percentage, the amount of sensors data transferred in a period of time or in opportunistic contacts and how bundles amount of sensors data varies.

The proposed architecture was validated in a real-world environment single test, conducted during one hour, thirty two minutes and forty five seconds. This test metadata analysis permitted to conclude that sense units to cloud *UrbanSense* database sensors data transmission through the vehicular network opportunistic contacts can be a reality at an urban-scale and that the developed applications perform very well. Moreover, the bundles delay varies in accordance with the roads traffic conditions. The minimum, mean and maximum bundles delay were 27 s, 140 s and 257 s, respectively. The percentage of sensor samples transmitted in first, second and third transmission

attempts was 75%, 20% and 5%, respectively. However, the sensor samples transmissions that failed had been attempted to be transmitted over TCP, before the sending over the vehicular network test started. To sum up, the test had a duration of 5565 s, 11 opportunistic contacts between the sense unit and the OBU and were transferred 250 bundles with a total 145.85 kilobytes of sensors data.

1.5 Document Structure

Beyond the introduction, this dissertation has more 6 chapters. Chapter 2 presents the *Future Cities* project overview, as well as details the vehicular network and the *UrbanSense* platform devices roles. Chapter 3 presents the mule architecture and the theoretical background necessary to understand the DTN concepts and the applications developed in the scope of this dissertation. Chapter 4 presents the proposed architecture to send sensors data from sense units to cloud the *UrbanSense* server, using the vehicular network. Chapter 5 presents a sense units detailed explanation and the cloud *UrbanSense* server applications. Chapter 6 presents and discusses the results obtained in a single real-world environment test, which permitted to understand the overall system performance. Finally, chapter 7 presents the dissertation work conclusions and future work.

Chapter 2

Future Cities Project

This chapter presents the *Future Cities* project. First, the project overview is presented and described its three main platforms: the *crowdsensing* Android application, the *UrbanSense* platform and the city of Porto vehicular network. Final, the *UrbanSense* platform and vehicular network devices and applications are detailed. These two platforms are further detailed, because this dissertation goal is to use the vehicular network to reliably send sensors data from sense units to the cloud *UrbanSense* server and send acknowledgement data in the reverse direction.

2.1 Overview

The *Future Cities* project is a FP7 European funded project. The key goal of the project is to boost the research and development of commercial products for smart cities turning the city of Porto into an *urban-scale living lab* to test technologies and services thought to improve the citizens' quality of life. To accomplish that, it aims to work in an interdisciplinary way, sharing knowledge among engineers, psychologists, social scientists and urban studies specialists. Three main urban-scale platforms have already been implemented and are continuously being improved: the *crowdsensing* application a vehicular network and an *UrbanSense* platform.

The *crowdsensing* application, also known as *SenseMyCity*, runs on Android devices and enables the use of smartphones' embedded and external sensors to collect data from users. Logging their ECG information using a *vital jacket* - ambulatory ECG system, is an example of an external sensor that people can use with this application. The analysis of data acquired with this kind of sensors, can lead to detection of heart diseases and city locations where people' level of stress is higher. The platform main targets of study are smart mobility and psychology.

The *UrbanSense* platform includes twenty five sense units sparsely installed in city and fifty mobile sense units installed on board of the Service of Transport, City of Porto (STCP) buses. Sensors data from mobile sense units is acquired when buses are stopped. Both types of sense units gather data such as the number of people or vehicles in a restricted area, air quality, noise

and meteorological conditions. The gathered data permits to identify critical urban areas and evaluate the impact of urban intervention actions. This platform also enables the development of research and public interest projects in areas such as public health, urban transportation planning and environment management. Furthermore, companies can use it as a proof of concept for their products.

The vehicular network testbed has already been installed on over six hundred vehicles, including buses, taxis and trucks, representing the largest Vehicle-to-Vehicle (V2V) communication platform in the world, at the time of the submission of this dissertation. Each participating vehicle carries an OBU communication device with four communication interface cards: Wi-Fi, 802.11p, 3G and Global Position System (GPS). Moreover, six RSU devices are connected to the city of Porto high-speed optical network bringing to the vehicular network the ability to reach the Internet. RSUs and OBUs communicate using the IEEE 802.11p technology, which is the IEEE standard for vehicular communications. The OBU and RSU devices and interfaces are further explained in chapter 2.3. This platform offers a wide range of services, which includes Internet access to bus passengers, fleet management and urban-traffic enforcement. Moreover, due to the platform huge dimension and potential, new services such as data mulling can be efficiently implemented.

2.2 UrbanSense Platform

This dissertation work focus on the integration of the *UrbanSense* and vehicular network platforms. The next sub-sections will further detail the sense units, cloud server and cloud database *UrbanSense* platform devices, especially its software and hardware components.

2.2.1 Sense Units

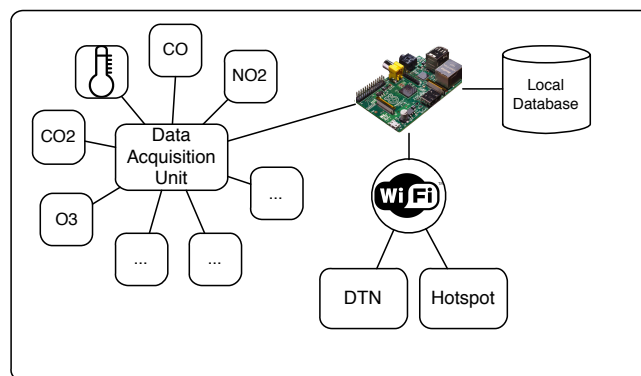


Figure 2.1: Sense Unit Platform Components

The sense units platform, represented in Figure 2.1, has three main modules: a *DataAcquisitionUnit*, a local database and a Raspberry Pi. Furthermore, it has a Wi-Fi 802.11n interface that enables sense units to connect to OBUs and Internet hotspots Wi-Fi interfaces. Since sense units

are power sourced by the national electrical network grid, sensors energy consumption is not a concern in this dissertation work.



Figure 2.2: Sense Unit Platform Real-Image

The *DataAcquisitionUnit* hardware is composed of a micro-controller, sensors signal conditioning electronic components and several different types of sensors, which includes: temperature, relative humidity, anemometer, luminosity, solar radiation, ozone, carbon dioxide/monoxide and noise. The micro-controller stores the newer sensors acquired samples in registers in RAM and update their values in accordance with sensors acquisition time interval. Furthermore, the *DataAcquisitionUnit* answers to different types of sensor samples requests from the Raspberry Pi, through a serial communication, to exchange data between both. In other words, a client-server model is implemented between both modules, in which the Raspberry is the client and the *DataAcquisitionUnit* is the server. Raspberry Pi requests sensor samples from the *DataAcquisitionUnit* and statically stores it in a SD card database. Sensor samples statically storage increases the *UrbanSense* platform robustness, in case of system failure.

Figure 2.2 shows a sense unit real image, installed on a city of Porto traffic light in Damião de Gois street. It was the sense unit used to conduct a real-word environment test and obtain the results discussed in chapter 6.3.

2.2.2 Cloud *UrbanSense* Server and Database

The cloud *UrbanSense* server is the sensors data destination. In the server, runs an application that accepts **TCP** socket connections to receive sensors data and upload it to the cloud *UrbanSense* database. It is prepared to receive data in JSON format and once it uploads data to the cloud *UrbanSense* database, it sends acknowledgement data back through the same TCP socket. The acknowledgement data contains information about the number of sensor samples successfully uploaded to the cloud *UrbanSense* database, the number of unknown sensor samples types and the number of sensor samples unsuccessfully uploaded. This application was developed to receive sense units data in case they are connected to an Internet hotspot and there is an end-to-end **TCP**

connection. The tables structure to store sensors data in cloud *UrbanSense* database is equal to the tables structure used in sense units local database.

2.3 Vehicular Network

The vehicular network platform is composed of **OBU**s - devices installed on board of buses, trucks and taxis, and **RSU**s - devices strategically installed on traffic lights in places with a great number of vehicles belonging to the vehicular network. Sub-Figures 2.3a and 2.3b show real images of an **OBU** and a **RSU**, respectively.

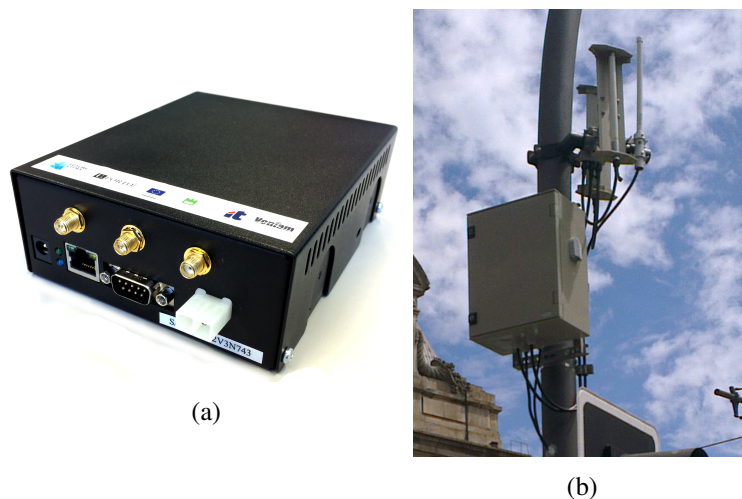


Figure 2.3: a) **OBU** Device Real-Image. b) **RSU** Device Installed on a Traffic Light Real-Image

Figure 2.4 shows all *Future Cities* project network devices used in this dissertation work and the interfaces they use to communicate among them. **OBU**s have three network interfaces: (1) Wi-Fi - configured in the managed mode, through which bus passengers and other devices connect to the **OBU** and have access to Internet and other services; (2) 802.11p - used to form the vehicular network and communicate with **RSU**s; and (3) 3G - used as a last resource to send data to the Internet, when connectivity through the 802.11p interface is unavailable, as well as **OBU**s clock synchronisation purposes. Moreover, **OBU**s have a **GPS** service that can be used to tag the sensors data location, clock synchronisation services and other purposes.

RSUs have an 802.11p interface for communications between **RSU**s and **OBU**s, a optical fiber interface connected to the Internet and a Wi-Fi interface. **RSU**s are used as gateways from the vehicular network to the Internet. The **RSU**s Wi-Fi interface is not used in the scope of this dissertation.

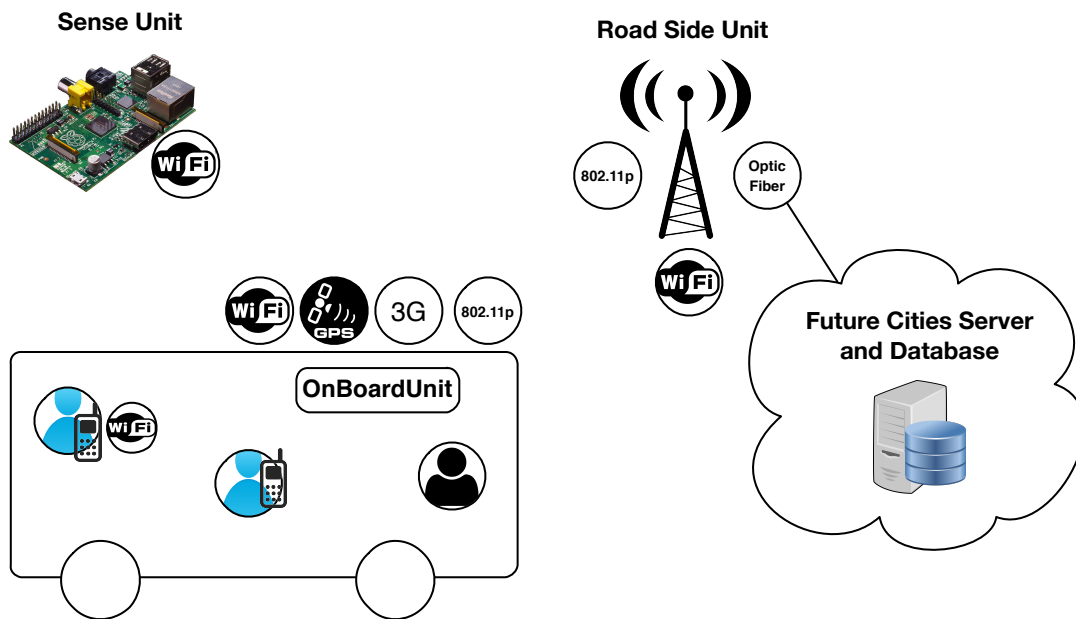


Figure 2.4: *Future Cities* Project Network Devices

2.4 Summary

First, this chapter presented the *Future Cities* project overview and described its three main platforms. At this moment, the reader understands the project huge dimension and how it is "segmented". Final, the *UrbanSense* and the vehicular network platforms are described in minute detail, due to this dissertation goal of integrating both. Sense units can either be installed on-board the vehicular network vehicles or in fixed locations. However, this dissertation work focus on send sensors data from fixed sense units to the cloud *UrbanSense* server, using the vehicular network. Consequently, along this dissertation, the "sense unit" term refers to sense units installed in fixed locations.

Chapter 3

State-of-Art

This dissertation work uses the vehicular network to reliably send sensors data from sense units to the cloud *UrbanSense* server. This chapter presents a particular DTN scenario which uses data mules, as well as some state-of-art information related to Delay Tolerant Networks (DTNs). First, the mule architecture is presented and discussed the principal advantages and disadvantages of carrying data instead of forwarding. Second, the DTN architecture and the *bundle protocol* specification are presented. Last, two open-source *bundle protocol* implementations are presented and compared.

3.1 The Mule Architecture

The mule architecture consists on using data mules, mobile entities with network capabilities, to carry data from sense units until an wired access point, which ables the mule to offload sensors data to the Internet. This architecture has been studied in [5] [6]. In [5], the mule architecture analysis is made through an analytical model, which considers mules random walks to provide insight into data transmission success rate and buffer sizes performance metrics. In [6], a TOSSIM simulator permitted to test their proposed ADT protocol, which tries to optimise the amount of data transferred on opportunistic contacts. Figure 3.1 shows the three different network nodes types used in the mule architecture: sensors, data mules and access-points.

Lower layer — sensors: Sensors acquire data, communicate via a short-range radio, and have limited memory. The amount of work performed by sensors should be minimised because they have the most constrained resources among the three layers. In our project, sensors are called sense units and its power resources consumptions is not a concern, because they are power sourced by the national electrical network grid.

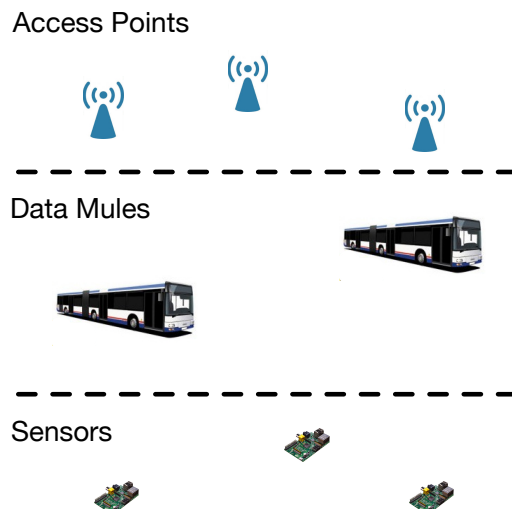


Figure 3.1: The Three Layers of the Mule Architecture

Middle layer — data mules: Data mules are mobile entities with large storage capacities (relative to sensors), renewable power, and have the ability to communicate with sensors and access-points. A data mule has the responsibility of sensors and access-points discovery and carry data between them. In our project, sense units have the responsibility to detect data mules, which are the vehicular network members.

Upper layer — access-points: These are the destination of sensor data. These are nodes with Internet connectivity and enhanced power, storage and processing capabilities. They are used to offload the data collected by and stored in the data mules. The *Future Cities* project has six access-points, called [RSUs](#), which data mules use offload their collected data.

3.1.0.1 Mule Discovery Protocol

A data mule needs to discover a nearby sensors node to be able to collect its data. An important aspect in the mule discovery protocol is mule mobility, which can be characterised as random, predictable or controllable [7]. Different node detection approaches, depending on mule mobility and node transmission initiation, were found in the literature. In [8], transmissions are scheduled and initiated from sense units taking into account energy minimisation and the data being transmitted. In the articles [7] [9] [10] [6], a mule is considered to be within the node range when the sensor detects the first beacon from the mule. On the other hand, a mule is considered to be out of range when the number of missed ACK messages exceeds a pre-defined threshold [9] [10] [6]. According to [7], beacons from data mules are continuously sent at fixed time intervals and a mule is considered to be away when beacons are not received for a certain period of time.

3.1.0.2 Trade-offs

When compared to vehicular networks, where generally data is routed instead of being carried, the mule architecture presents some advantages and disadvantages:

3.1.0.3 Advantages

Energy Efficient Substantial energy is saved because sensors communicate over a short range. Moreover, as sensors do not forward data between each other, the sensors near the gateway to the cloud are not overloaded.

Spatial Reuse The mule architecture exploits spatial reuse of bandwidth by using short-range communication avoiding radio communication complexities such as collisions.

Routing overhead The mule architecture has less routing protocol overhead.

Robustness Performance degrades gracefully as mules fail. Any single mule failure does not lead to a disconnected network. The primary effect of a mule failure on the overall system is a slight increase in latency as there are now fewer mules to pick up data.

Scalable The mule architecture is easily scalable as deployment of new sensors or mules do not requires network reconfiguration.

Simplicity The data routing aspect of the mule architecture is very simple and lightweight for all nodes evolved.

3.1.0.4 Disadvantages

Latency The mule architecture has high latency what limits its applicability to realtime applications.

Best-effort delivery Data delivery is best-effort. After collecting sensors data, the mule may take some time to reach near an access-point to deliver it.

3.2 DTN Architecture

Delay Tolerant Network is a store and forward network architecture that seeks to address the technical issues in heterogeneous networks. This network environment is characterised by very long delay communications and frequent network partitions, which may take to the lack of an end-to-end contemporaneous path. DTN defines a new layer on top of the transport layer and below of the application layer, called bundle layer. The bundle layer is responsible for the end-to-end delivery mechanism of messages, called virtual message forwarding. This problem contrasts with

the end-to-end connected data networks which typically selects the shortest policy-compliant path to send data. Examples of DTNs are: deep space networks, sensor-based networks, terrestrial wireless networks, underwater acoustic networks and satellite networks [11].

3.2.1 DTN Architectural Principles

The DTN architectural principles presented here, RFC 4838 [11], were defined by the Internet Research Task Force (IRTF), but do not define any kind of Internet standard. They are an upgrade from the architectural principles previously defined for the Interplanetary Internet.

Virtual Message Switching using Store-and-Forward Operation A DTN-enabled application creates messages of arbitrary length called Application Data Units (ADUs). Then, at the bundle layer, ADUs are transformed into one or more Protocol Data Units (PDUs) called bundles containing two or more blocks of data. Each block may contain either application data or control information used to deliver the bundle payload to its destination(s). Due to the lack of an end-to-end connected path, bundles may have to be stored in the network for long periods. Consequently, persistent storage is required to cope with hardware failures and increased reliability.

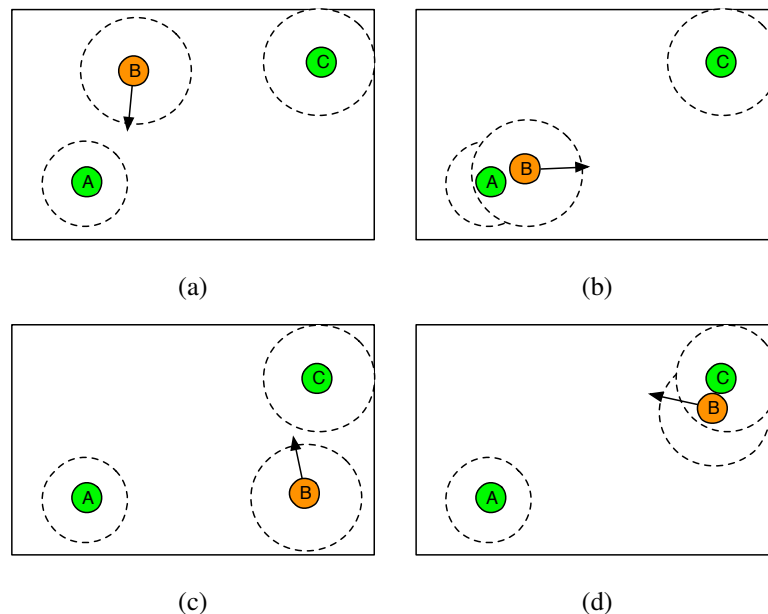


Figure 3.2: Store and forward operation: a) node B getting close node A, b) opportunistic contact, c) node B moving around, and d) data delivering to its destination

Figure 3.2 depicts how the store-and-forward mechanism works and how the node B mobility is used to deliver a bundle from the node A to the node C. In subfigure 3.2a, the node A can not reach any node in its neighbourhood. However, the node B is moving toward it. As shown in subfigure 3.2b, the node B mobility allowed bundles to be transferred from the node A to the node B. In subfigure 3.2c the node B is not yet able to forward the message to the node C. If the node

B had come across another node trying to send bundles, node B could also collect its bundles. In subfigure 3.2d, the node B can finally deliver the collected bundles to their destination.

Naming and Addressing Mechanism Every node in the DTN architecture is identified through an Endpoint Identifier (EID), but in case of multicast or anycast, an EID may refer to multiple DTN nodes. In other words, a single EID may point to one or more nodes, and a single node may have more than one EID. However, every node must be a member of at least one “singleton” endpoint, i.e., have a unique (singleton) EID. An EID is a name, expressed using the general syntax of Uniform Resource Identifiers (URIs), which has been designed as a way to express names or addresses for a wide range of purposes, and is therefore useful for constructing names for DTN nodes. URIs intended for use with DTNs should be compliant with the guidelines given in [12]. EIDs are used to identify the source and destination endpoints of a bundle, the endpoint to which bundle status reports are to be sent, and the current custodian of the bundle. The following is an example of an EID:

```
dtn://Server.dtn/ServerA (dtn://<scheme>/<scheme-specific part>)
```

A DTN-enabled application can subscribe ADUs destined for a particular EID through a "registration", which is generally persistently stored by DTN nodes. When an application does a registration with an EID, it signals that these particular application wishes to receive bundles sent to that EID. If the node receives bundles destined to a registered EID, they are handed over to the application that performed the registration.

In contrast to the name resolution process of the Internet, in which the destination IP address is determined at the source before data is sent, DTN also incorporates the concept of late binding. It means that the mapping between a destination EID and its lower-layer address is not necessarily performed at the source node. In DTN, the mapping may occur at the source, during transit or possibly at the destination. This is advantageous because the transit time of a bundle may exceed the validity time of a binding. The use of name-based routing with late binding may also reduce the amount of administrative traffic in the network.

Custody Transfer and Return Receipt DTNs can implement data reliability at the transport layer and the bundle layer. However, due to DTN supports different transport layer protocols between nodes, end-to-end reliability has to be implemented at the bundle layer. It is made through a *return receipt*, sent from the destination to the source node, when a bundle reaches its destination. The source node must keep sent bundles until their *return receipt* have been received, and retransmit in case it had failed.

The bundle layer also supports hop-by-hop reliability by means of *custody transfers*. The custody of the bundle traverses the network until the final destination is reached or the bundle is discarded, if its time-to-live expires. When the source application request *custody transfer*, it sends a bundle and starts a time-to-acknowledge retransmission timer. If the next hop node accepts custody, it returns an acknowledgment to the sender. If no acknowledgment is returned before the

retransmission time expires, the sender retransmits the bundle. A custodian node must store the bundle until another node accepts custody or the bundle's time-to-live expires. [13]

Priority Classes The DTN architecture implements a Quality of Service (QoS) mechanism which classifies delivering ADUs priority as low, medium or high. However, this mechanism doesn't guarantee a bundle will be delivered within a fixed time interval, it just offers a relative priority between ADUs. According to the DTN-enable application desire to affect the ADU delivery, the QoS mechanism is implemented at the bundle layer through the primary bundle block flags.

Delivery Options and Administrative Records DTN applications can choose eight delivery options for sending ADU. The first four are designed for ordinary use by applications, while the last four are primarily designed for diagnostic purposes. Due to the extra overhead, their use may be restricted or limited to environments subject to congestion or attack.

- **Custody Transfer Requested** Requests sent bundles to be delivered with enhanced reliability using custody transfer procedures.
- **Source Node Custody Acceptance Required** Requires the source DTN node to provide *custody transfer* for the sent bundles.
- **Report When Bundle Delivered** requests a *bundle delivery status report* to be generated when the subject ADU is delivered to its intended recipient(s).
- **Report When Bundle Acknowledged by Application** Requests an *acknowledgement status report* to be generated when the subject ADU is acknowledged by a receiving application.
- **Report When Bundle Received** Requests a *bundle reception status report* to be generated when each sent bundle arrives at a DTN node.
- **Report When Bundle Custody Accepted** Requests a *custody acceptance status report* to be generated when each sent bundle has been accepted using custody transfer
- **Report When Bundle Forwarded** Requests a *bundle forwarding status report* to be generated when each sent bundle departs a DTN node after forwarding.
- **Report When Bundle Deleted** Requests a *bundle deletion status report* to be generated when each sent bundle is deleted at a DTN node.

If the *bundle security protocol* procedures defined in chapter 3.2.2 are also enabled, then three additional delivery options become available:

- **Confidentiality Required** Requires the subject ADU be made secret from parties other than the source and the members of the destination EID.

- **Authentication Required** Requires all non-mutable fields in the bundle blocks of the sent bundles be made strongly verifiable .
- **Error Detection Required** Requires modifications to the non-mutable fields of each sent bundle be made detectable with high probability at each destination.

According to the chosen delivery options, *administrative records* are used at the bundle layer to report bundles status information or error conditions. It is accomplished by a method for uniquely identifying bundles based on a *transmission timestamp* and *sequence number*. There are two types of administrative records defined: *Bundle Status Reports (BSRs)* and *custody signals*. The following *BSRs* are currently defined: *bundle reception*, *custody acceptance*, *bundle forwarded*, *bundle deletion*, *bundle delivery* and *acknowledged by application*. Additionally to status reports, the *custody signal* indicates the status of a custody transfer. It indicates either a successful or a failed custody transfer attempt, through a Boolean indicator. These are sent to the current-custodian *EID* contained in an arriving bundle.

Fragmentation and Reassembly There are two types of fragmentation/reassembly currently designed. They are used to improve the efficiency of bundle transfer by ensuring that contact volumes are fully utilised and by avoiding retransmission of partially-forwarded bundles.

- **Proactive Fragmentation** Based on contact history or predicted contact time, a *DTN* node may divide the bundles into smaller ones and transmit each of them as an independent bundle.
- **Reactive Fragmentation** When a bundle is only partially transferred, the receiving *DTN* node modifies the incoming bundle to indicate it is a fragment, and forwards it normally.

Timestamps and Time Synchronisation The *DTN* architecture depends on time synchronisation among *DTN* nodes for four primary purposes: bundle and fragment identification based on the timestamp, routing with scheduled or predicted contacts, bundle expiration time computations, and application registration expiration.

Neighbour Discovery Identity and meeting schedule among *DTN* nodes may be unknown. The ability to discovering and exchanging data between *DTN* nodes is accomplished through a neighbour discovery mechanism. In [14] is specified the *DTN* IP Neighbour Discovery (*IPND*). *IPND* sends and listens IP UDP "beacons" to detect opportunistic contacts. Depending on the remote target neighbours, beacons can be addressed to IP unicast, multicast or unspecified local neighbours.

3.2.2 Bundle Security Protocol

The *DTN* architecture security can be achieved through the *bundle security protocol* specified in [15]. It implements separately hop-by-hop and end-to-end authentication and integrity mechanisms. The purpose of this distinction is to be able to handle access control for data forwarding

and storage separately from application-layer data integrity. On one hand, the end-to-end mechanism provides authentication for a principal such as a user. On the other hand, the hop-by-hop mechanism is intended to authenticate DTN nodes as legitimate transceivers of bundles to each other.

The DTN security architecture has the following goals:

- Prevent unauthorised applications from having their data carried through or stored in the DTN.
- Prevent unauthorised applications from taking control over the DTN infrastructure.
- Prevent applications from sending bundles at a rate or class of service for which they lack permission.
- Discard bundles that are damaged or improperly modified in transit.
- Detect and de-authorise compromised entities.

To summarise, bundle security is concerned with the authenticity, integrity, and confidentiality of bundles transmitted among bundle nodes. This is accomplished via the use of three independent security-specific bundle blocks, which may be used together to provide multiple bundle security services or independently of one another, depending on the secure level required. The Bundle Authentication Block (**BAB**) ensures the authenticity and integrity of bundles on a hop-by-hop basis between bundle nodes. The **BAB** allows each bundle node to verify a bundle authenticity before processing or forwarding the bundle. In this way, entities that are not authorised to send bundles will have unauthorised transmissions blocked by security-aware bundle nodes. Additionally, the Payload Security Block (**PSB**) provides "security-source" to "security-destination" bundle authenticity and integrity. It can be checked along the delivery path by any security-aware entity. Finally, payload confidentiality is provided using the Confidentiality Block (**CB**). The **CB** indicates the cryptographic algorithm and key IDs that were used to encrypt the payload [16].

3.2.3 *Bundle Protocol*

The bundles carried between **DTN** nodes obey to the *bundle protocol* specification in [16]. The *bundle protocol* is the primary **DTN** protocol running at the bundle layer and its location within the standard protocol stack is as shown in figure 3.3. The *bundle protocol* end-to-end successful operation depends on the operation of underlying protocols at what is termed the "convergence layers". These protocols accomplish point-to-point communication between **DTN** nodes. To implement the main **DTN** principles, *bundle protocol* has to be able to deal with intermittent connectivity, supports late binding of reactively fragmented bundles, implements discovery mechanisms and custody based retransmission.

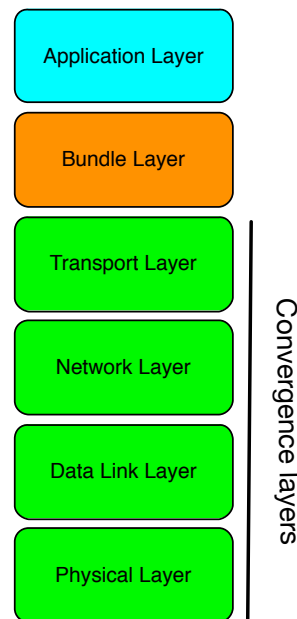


Figure 3.3: *Bundle Protocol* Architecture

The *bundle protocol* uses Self-Delimiting Numeric Values (**SDNVs**) encoding scheme in order to minimise the transmission bandwidth consumption. The **SDNVs** are closely adapted from the Abstract Syntax Notation One Basic Encoding Rules for sub-identifiers within an object identifier value [17]. An **SDNV** is a numeric value encoded in N octets.

The *bundle protocol* is designed with a mandatory *primary block*, an optional *payload block* and a set of optional *extension blocks*. The following fields are present in the primary block, figure 3.1, and therefore are present for every bundle and fragment:

- **Bundle Processing Control Flags** Is an **SDNV** that contains the bundle processing control flags depending on the *Delivery Options* 3.2.1.
- **Block Length** Contains the length of all remaining fields of the block.
- **Destination Scheme Offset** Contains the scheme name of the **EID** of the bundle destination.
- **Destination SSP Offset** Contains the scheme-specific part of the **EID** of the bundle destination.
- **Source Scheme Offset** Contains the scheme name of the **EID** of the bundle source node.
- **Source SSP Offset** Contains the scheme-specific part of the **EID** of the bundle source.
- **Report-to Scheme Offset** Contains the scheme name of the **EID** to which status reports pertaining to the forwarding and delivery of this bundle are to be sent.

- **Report-to SSP Offset** Contains the scheme-specific part of **EID** to which status reports pertaining to the forwarding and delivery of this bundle are to be sent.
- **Custodian Scheme Offset** Contains the scheme name of the current custodian **EID**.
- **Custodian SSP Offset** Contains the scheme-specific part of the current custodian **EID**.
- **Creation Timestamp** Is a pair of SDNVs that, together with the bundle source **EID** and (if the bundle is a fragment) the fragment offset and payload length, serve to identify the bundle.
- **Lifetime** Indicates the time at which the bundle payload will no longer be useful.
- **Dictionary Length** Contains the length of the dictionary byte array.
- **Dictionary** Is an array of bytes formed by concatenating the null-terminated scheme names and SSPs of all **EIDs** referenced by any fields in this Primary Block together with, potentially, other **EIDs** referenced by fields in other **DTN** protocol blocks. Its length is given by the value of the Dictionary Length field.
- **Fragment Offset** If the bundle *processing control flags* of this primary block indicate that the bundle is a fragment, then the *fragment offset field* is an SDNV indicating the offset from the start of the original application data unit at which the bytes comprising the payload of this bundle were located. If not, then the *fragment offset field* is omitted from the block.
- **Total Application Data Unit Length** If the bundle *processing control flags* of this primary block indicate that the bundle is a fragment, then the total **ADU** length field is an SDNV indicating the total length of the original **ADU** of which this bundle payload is a part. If not, then the total **ADU** length field is omitted from the block.

Version	Proc.Flags
Block length	
Destination scheme offset	Destination SSP offset
Source scheme offset	Source SSP offset
Report-to scheme offset	Report-to SSP offset
Custodian scheme offset	Custodian SSP offset
Creation timestamp time	
Creation timestamp sequence number	
Lifetime	
Dictionary length	
Dictionary byte array	
Fragment offset	
Total application data unit length	

Table 3.1: Bundle Primary Block

3.2.4 Convergence Layer Protocols

The major categorisation splits convergence layers into connection oriented and non-connection oriented groups. There are significant differences between the ways in which the two types operate. On one hand, in connection oriented convergence layers, a communication link is established before any useful data can be transferred. TCP convergence layer is an example of this communication mode and provides reliable and ordered packets delivery. On the other hand, in non-connection oriented convergence layers bundles are simply transferred. This communication mode neither implement flux control mechanisms nor reliability, what results in a reduced communication overhead. User Datagram Protocol (UDP) convergence layer is an example of this communication mode.

3.3 DTN Implementations

Considering our network scenario and the use of the Raspberry Pi platform, there are two *bundle protocol* open-source implementations, developed to run on embedded systems, available: DTN2 and IBR-DTN.

3.3.1 DTN-2

DTN2 [18] is the reference implementation of the *bundle protocol* by the Delay Tolerant Networking Research Group (DTNRG). The core implementation is written in C++ using a framework called Oasys that is designed to provide a uniform interface to the DTN2 code. DTN2 optionally supports the *bundle security* protocol, Ethernet and Bluetooth convergence layers, routing mechanisms including static, epidemic, prophet, DTLSR and TCS routing, as well as storage mechanisms including file system, Berkeley database, RAM memory, SQLite and MySQL.

3.3.1.1 Architecture

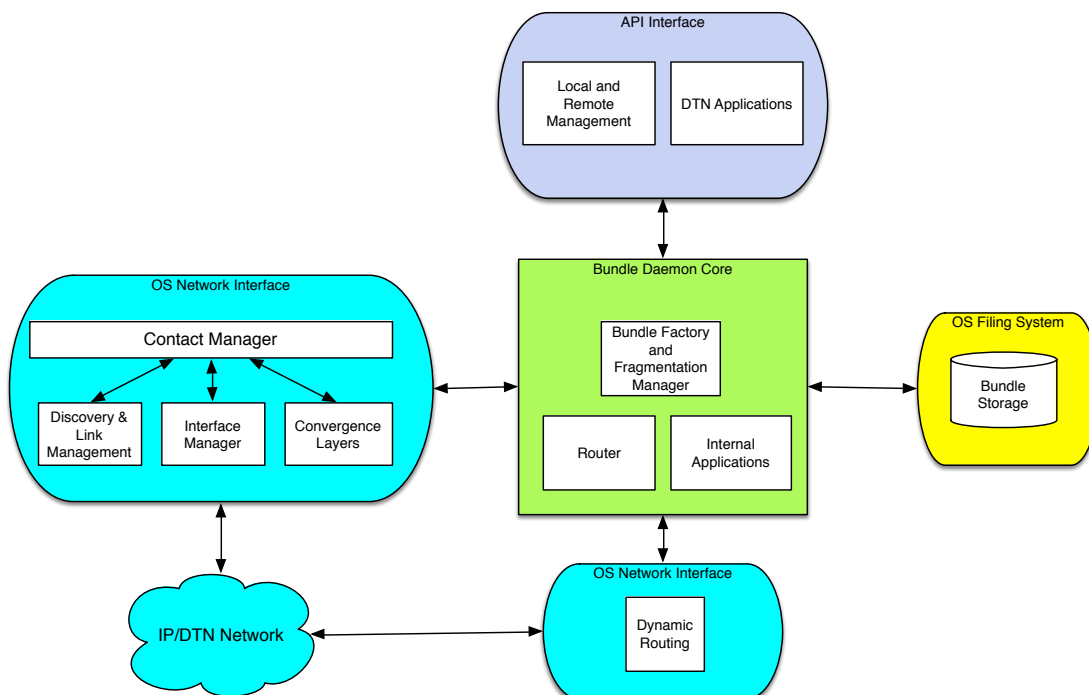


Figure 3.4: DTN2 Daemon Architecture

The *bundle protocol* agent and all its support code is implemented as a user space daemon called Bundle Daemon (BD). Figure 3.4 shows the DTN2 architecture. The main blocks are: the API interface, the *Bundle Daemon Core*, *OS Network Interface* and *OS Filing System*.

3.3.1.2 API Interface

Local and Remote Management Provides configuration and monitoring functionalities of the operations of the DTN2 daemon.

DTN Applications Handles RPC-based communication between applications and the Bundle daemon process.

3.3.1.3 Bundle Daemon Core

Bundle Daemon Core is the master component of the DTN2 daemon. It is responsible for statistics, event distribution and event handler.

Bundle Factory and Fragmentation Manager Deal with bundles as defined in the *bundle protocol 3.2.3* and the *bundle security protocol 3.2.2*. The Bundle Factory is responsible for creating the bundles. Fragmentation Manager is responsible for organising sets of fragments of a bundle and determining when the available fragments "cover" a bundle so that it can be reassembled and delivered to local applications.

Bundle Router The Bundle Router is the main decision maker for all routing and forwarding decisions related to bundles. It receives events from the Bundle Daemon Core after they have been processed in the bundle core. Routing mechanisms can be classified into static and dynamic mechanisms. For static mechanisms the routes are worked out in advance either due to static configuration or a schedule. For dynamic mechanisms, information (normally known as metadata) is exchanged between **BDs** that use it to determine routes and whether to forward bundles on particular links.

3.3.1.4 OS Network Interface

Dynamic Routing Is only viable if a low latency reliable link can be established locally between nodes involved in an opportunistic encounter. The local information exchange will therefore take place using a separate TCP-based connection rather than being encapsulated in bundles.

Contact Manager Is the central actor in the communications sub-system of the DTN2 daemon. Is responsible for managing communications links from the local node to other nodes over which bundles can be sent.

Interface Manager Maintains a table of Interfaces that are the low level communication end-points for the DTN2 daemon. Interfaces are created and activated through administrative action, typically by initial configuration of the DTN2 daemon and abstract a bundle transport communication end point.

Discovery and Link Management Maintains the set of active Discovery Agents and provides means for adding new ones and deleting ones that are no longer required. A Discovery Agent can be configured to advertise one or more convergence layers. Advertisements are sent out periodically according to a configured interval. An advertisement contains the name of the convergence layer to be used and address information that can be used to either send bundles towards for non-connection oriented convergence layers or to establish a connection for connection oriented convergence layers. On receiving an advertisement from a prospective next-hop neighbour, the

Discovery Agent will pass the advertisement to the Link manager, to determine if there is an existing reusable link or, if not, to create a new opportunistic link to handle communications.

Convergence Layers encapsulates the protocol used for communication on a particular link during a contact, and manages the interactions between the local communication devices and the DTN2 daemon in establishing a link, closing down a link, and handling data reception and transmission on the link.

3.3.2 IBR-DTN

IBR-DTN [3] is a *bundle protocol* implementation developed specifically for embedded systems and is designed for and tested against uClibc. It provides a DTN daemon based on an event mechanism that supports different routing schemes, different convergence layers, persistent and non-persistent storage, security mechanisms, proactive and reactive fragmentation, as well as two discovery mechanisms.

3.3.2.1 Architecture

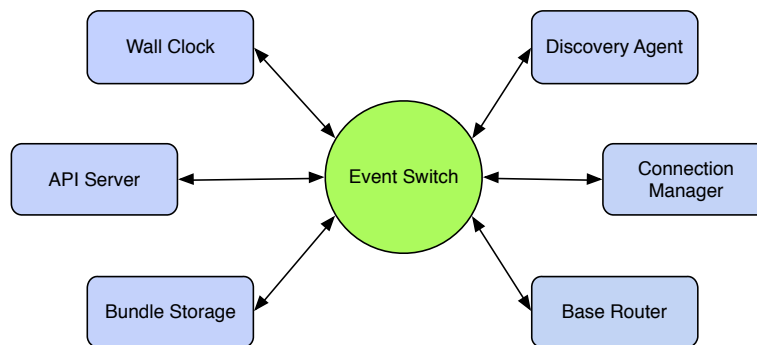


Figure 3.5: IBR-DTN Daemon Architecture

Figure 3.5 depicts an architectural overview of IBR-DTN. Due to its modular implementation, when compiling IBR-DTN, different modules can be chosen to best fit the implementation scenario. The IBR-DTN standard modules are [3]:

Event Switch At the core of IBR-DTN, *Event Switch* dispatches a set of standard events to all relevant sub-modules. Existing and new modules can receive and raise events to communicate with other parts of the daemon.

Discovery Agent Is responsible for implementing a discovery mechanism through pluggable discovery modules. Currently, there are DTN IPND version one and two as specified in 3.2.1 and

IP-Discovery frames compatible to DTN2. The *Discovery Agent* events are triggered by neighbour discovery and disappearance.

Connection Manager The *Connection Manager* manages the instances of convergence layer modules, based on the daemon configuration. Each convergence layer provides an interface to transfer bundles to neighbouring nodes. IBR-DTN (version 12.0) has five built-in convergence layers:

- **TCP Convergence Layer** Compatible with IETF draft [19]. TCPCL uses a handshake mechanism before useful data transmission and an acknowledge mechanism for data flux control between DTN nodes.
- **UDP Convergence Layer** Compatible with IETF draft [20]. UDPCL is the simpler way of bundles transfer between DTN nodes. Since it requires that a bundle fits into a single UDP datagram, the maximum bundle size is limited.
- **Hypertext Transfer Protocol Convergence Layer** Can use an HTTP server to send and receive bundles. This convergence layer is based on libcurl4.
- **LowPersonal Area Network Convergence Layer** Supports the IEEE 802.15.4 MAC standard commonly used in sensor networks.
- **Generic Datagram Convergence Layer** Is a protocol abstraction layer that supports the IEEE 802.15.4 MAC and UDP standard.

Bundle Storage IBR-DTN supports the three types of bundle storage following described:

- **Memory-based storage** This is a non-persistent bundle storage and is used by default if no storage path is set. All bundles are kept in RAM in this case.
- **File-system based storage** This is a persistent storage based on simple files. The functionality is equal to the memory-based storage, but all bundles are serialised to disk.
- **SQLite** SQLite uses a database as backend and can store more metadata information for bundles than the other solutions.

Base Router The *Base Router* is able to manage one or more different routing modules at the same time. Receiving events by routing modules are triggered every time discovered and nodes disappeared occurs, as well as whenever new bundles arrive in the storage. When a routing module needs to send a bundle, it generates an event to the *Connection Manager* requesting the appropriate converge layer to send the bundle to the next hop.

Wall Clock The Wall Clock module determines the current global time in the DTN through its local clock. Additionally to basic time querying functionality this module provides a global time tick event, which triggers the initiation of simple modules recurring tasks.

IBR-DTN API IBR-DTN reuses the bundle streaming protocol to provide a socket based API interface. This can either be a TCP socket which allows to run the daemon and DTN applications on different machines, or a Unix Domain Socket.

Implemented tools IBR-DTN (version 0.12.0) distribution provides eight tools: (1) and (2) `dtnsend` and `dtnrecv` - can send and receive files and standard in/output; (3) `dtnping` - sends out bundles to a DTN Endpoint Identifier and waits for a bundle with the same payload as reply; (4) `dtntracepath` - the path of a bundle is discovered using the *bundle forwarded* reports from each hop as defined in [16]; (5) and (6) `dtninbox` and `dtnoutbox` - automatise the process of sending and receiving bundles to and from the DTN. The bundles in the outbox folder are automatically transferred to the inbox folder; (7) `dtnstream` - it allows receiving a stream on a DTN node; (8) `dtnttrigger` - calls an executable every time a bundle to a specific EID is received.

3.3.3 Implementations' Comparison

		DTN2	IBR-DTN
The Basic Bundle Protocol		X	X
Self Describing Numeric Values		X	X
Endpoint Identifiers		X	X
Bundle Expiration		X	X
Registrations		X	X
Persistent Bundle Storage		X	X
Reactive Bundle Fragmentation		X	X
Proactive Bundle Fragmentation		-	X
Bundle Fragment Reassembly		X	X
Custody Transfer		X	X
Custody Acceptance		X	X
Bundle Status Reports	Received	X	X
	Forwarded	X	X
	Delivered	X	X
	Deleted	X	X
	Acknowledge by Application	-	-
Extension Blocks		X	X
Bundle Security Blocks		X	X

Table 3.2: DTN2 and IBR-DTN RFC 5050 Features Comparison

In [21], the DTN2 and the IBR-DTN performance characteristics are evaluated, using a DTN emulation testbed. For 2-3 nodes, they concluded that both the DTN2 and the IBR-DTN functioned as expected. Increasing the DTN2 scale, its performance degraded quickly, leading to poor results for 26-node scenarios, even if only some of them are mobile. IBR-DTN behaved as expected in all the tested scenarios. In [22], IBR-DTN and DTN2 throughput between two nodes connected via GBit Ethernet is measured. IBR-DTN almost reaches the theoretical limit of the link (940 MBit/s) for large bundle sizes with a throughput of up to 843.341 MBit/s for disk storage. Whereas, the DTN2 reaches a maximum of 719.977 MBit/s with memory storage. A lower bandwidth indicates that bundle processing overhead in a given daemon might be too high. Table 3.2 compares the DTN2 and IBR-DTN RFC 5050 features. Both implementations implements most of the RFC 5050 DTN architectural principles.

3.4 Summary

Firstly, the mule architecture is presented and detailed the advantages and disadvantages of carrying bundles instead of forward them. Instead of using a complex routing protocol to forward the sensors data from node to node until the destination, a data mule carries it until reach an access point and offloads it to the Internet. Secondly, DTN principles are presented. The DTN architecture addresses many of the problems of heterogeneous networks that must operate in environments subject to long delays and discontinuous end-to-end connectivity. It also proposes a model for securing the network infrastructure against unauthorised access through the *bundle security* protocol. Clearly, DTN architecture abodes the sense units to cloud *UrbanSense* communication problem issues, what makes the use of the work around DTNs a good start point to this dissertation work. Finally, DTN2 and IBR-DTN open-source *bundle protocol* implementations are presented and compared. Both *bundle protocol* implementations, implements most of the DTN principles.

Chapter 4

The Proposed Architecture

This chapter presents the proposed architecture to send *UrbanSense* sense units data to the *UrbanSense* server in the cloud. Firstly, the system overview is presented and explained the sense units types of communication infrastructures available to send sensors data to the cloud *UrbanSense* server. Secondly, this dissertation proposed architecture is presented, in order to reliably send sensors data from sense units to the cloud *UrbanSense* server using opportunistic communications. Thirdly, it is presented the type and format of data that flows in sense units to the cloud *UrbanSense* server and in the inverse direction. Finally, our solution to make sensors data transmission reliable is presented.

4.1 System Overview

The *UrbanSense* sense unit platforms, presented in chapter 2.2, can either be installed on-board the vehicular network vehicles or in fixed locations. The *UrbanSense* mobile sense units integration is beyond the scope of this dissertation. In this dissertation, the sense unit term refers to sense units installed in fixed locations. A Wi-Fi interface, configured in client mode, is the only network capability sense units can use to send sensors data to the cloud *UrbanSense* server. It makes them the ability to communicate with vehicular network vehicles and Internet hotspots. Although the city of Porto has a wide variety of public hotspots around: *PortoDigital*, *MEO*, *NOS*, at this moment only *PortoDigital* can be used. However, new accords can be celebrated in the future.

Both the vehicular network and Internet hotspots are configured in managed mode and announce an SSID. A network manager application running on sense units aware of the types of networks around it, decides if data is sent to the cloud *UrbanSense* server through Internet hotspots or opportunistic communications. Internet hotspots are the preferred option to accomplish that, because of their stability and larger bandwidth. When they are available, sense units connect to them, establish an end-to-end TCP connection to the cloud *UrbanSense* server and send their locally stored sensors data. Only after sensors data has been uploaded to the cloud *UrbanSense*

database, sensors data is deleted from the local storage. When there is not any Internet hotspot available, sensors data is transferred to the cloud *UrbanSense* server through the vehicular network. This dissertation work focus on sense units and vehicular network integration, to reliably send sensors data to the cloud *UrbanSense* server, in an opportunistic way.

All devices involved in opportunistic sensors data transmission from sense units to the cloud *UrbanSense* server, make part of the DTN. It means they use a *bundle protocol* implementation, which implements the RFC 5050 *bundle protocol* specification [16], to encapsulate data into bundles and communicate among them. Introduced in the previous chapter 3.2.3, the *bundle protocol* is a layer five protocol, developed to communicate in DTN network scenarios. It uses a store and forward mechanism to transmit bundles from the source to the destination.

4.2 Opportunistic Communications

Figure 4.1 represents all the architecture devices involved in data transmission between sense units and the cloud *UrbanSense* server, as well as the type of data exchanged between them. The left side, represents a sense unit *UrbanSense* platform device. The center, represents the vehicular network devices, namely the OBUs installed on board of buses and the RSUs that are connected to the cloud *UrbanSense* server through optical fiber. In the right side, it shows the cloud *UrbanSense* server and database. As we can see in Figure 4.1, sensors data is transferred from sense units to cloud *UrbanSense* server, whereas application level acknowledgement data is transferred in the reverse direction.

The *bundle protocol* runs on both Wi-Fi and 802.11p OBU interfaces. Otherwise, OBUs would not be able to communicate with sense units and RSUs. The RSUs also run a *bundle protocol* to communicate with OBUs and the cloud *UrbanSense* server through the 802.11p and optical fiber interfaces, respectively. Although communications between RSUs and the cloud *UrbanSense* server are stable, they use a *bundle protocol* to communicate. The cloud *UrbanSense* database is not part of the DTN.

Figure 4.1 shows the data transmission proposed architecture. In phase 1, when an opportunistic contact between a sense unit and an OBU happens, sensors data is encapsulated into bundles and transferred from the sense unit to the OBU. In phase 2, the OBU carries bundles until it find a RSU. In phase 3, bundles are transferred from the OBU to the RSU. In phase 4, bundles are transferred from the RSU to the cloud *UrbanSense* server. In phase 5, sensors data is extracted from bundles and sent to an application that process sensors data and uploads it to the cloud *UrbanSense* database. In phase 6, that application answers the application level acknowledgement data that is sent to the sense unit data source. Acknowledgement data is encapsulated into bundles and transferred in cloud *UrbanSense* server to sense units direction, when opportunistic contacts among DTN devices happen.

Although the *bundle protocol* specifies the necessary mechanisms to store and forward bundles from the source to the destination, we have to design two applications, one in the cloud *UrbanSense* server and the other in sense units, in order to send and receive data over the DTN.

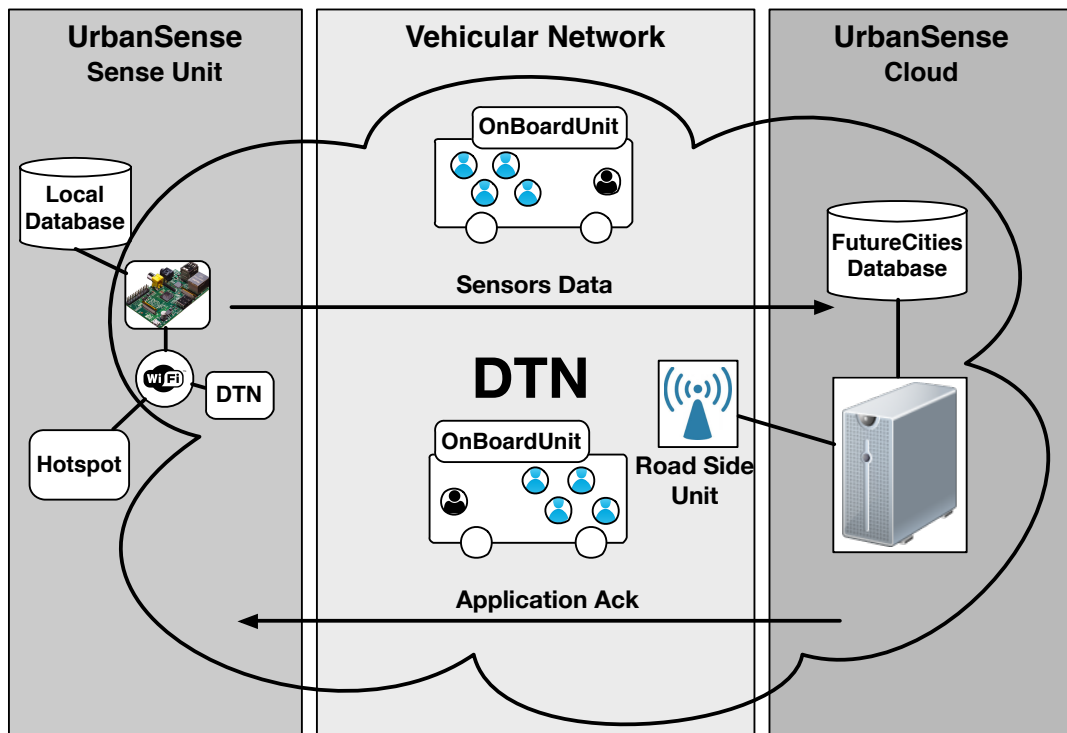


Figure 4.1: Opportunistic Communications Proposed Architecture

The application that runs in the cloud *UrbanSense* server (1) checks if there are bundles available, (2) gets the bundle sensors data, (3) establishes a **TCP** socket connection to the application that process sensors data, and (4) sends sensors data to that application. After uploading sensors data to the *UrbanSense* database, that application sends acknowledgement data through the **TCP** connection, which our application sends through the **DTN** to the sense unit that originated that data. The application that runs in sense units (1) detects the opportunistic contact with **OBUs**, (2) sends and receives bundles over the **DTN**, and (3) manages the local database in accordance with sensors data transmission success or failure. When acknowledgement data reaches the sense unit, the locally stored sensors data that correspond to that acknowledgement data is deleted.

The application that runs in the cloud *UrbanSense* server process sensors data in the same way, either if data is sent through an end-to-end **TCP** connection, in case sense units are connected to a Internet hotspot, or **DTN** plus **TCP** connections, when data is sent over the **DTN** and "converted" to **TCP**. The application that "converts" **DTN** to **TCP** and **TCP** to **DTN**, can run both on **RSUs** or in the server. Ideally, due to the stable Internet connection between both, it should should run in the **RSU**, reducing the communication overhead. However, we did not propose to the vehicular network managers to install it on **RSUs**.

4.3 Types of Data

Types of data carried in bundles is divided in two categories: sensors data and acknowledgement data. Sensors data flows from sense units to the cloud UrbanSense server and acknowledgement data flows in cloud UrbanSense server to sense units direction.

4.3.1 Sensors Data

Sensors data includes sensor samples data, metadata and data about that transmission. Figure 4.2 shows the sensor samples structure. The cyan fields corresponds to transmission information, the yellow fields corresponds to sensor samples and green fields corresponds to metadata information. The *SERIAL*, *VERSION* and *SEQ* fields, stores the network node identifier, data structure version and transmission sequence number, respectively. The *DATA* variable contains sensor samples and metadata. For each type of sensor sample type, it has the acquisition time instant and the respective sensor value measured.

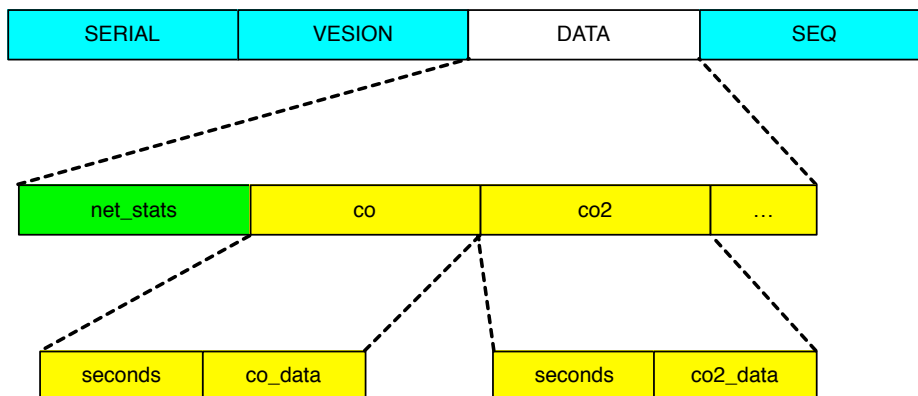


Figure 4.2: Sensor Samples Structure

Metadata statistical analysis permits to understand the system performance, namely the delay, percentage of retransmissions, amount of sensors data sent on each bundle and the total amount of sensors data transmitted on each contact. Figure 4.3 shows the metadata data structure, emphasising the metadata information. Metadata information is represented in the green fields. The *connection_type* distinguish if that data was sent over the DTN or not. If it is sent over DTN, its value is equal to 1. The *SRC* and *DST* variables store the source and destination EIDs, respectively. The *seconds* variable stores the instant that bundle was created. The *tx_count* variable stores the number of times the same sensor sample was sent. The *payload_size* variable identifies the size of sensors data.

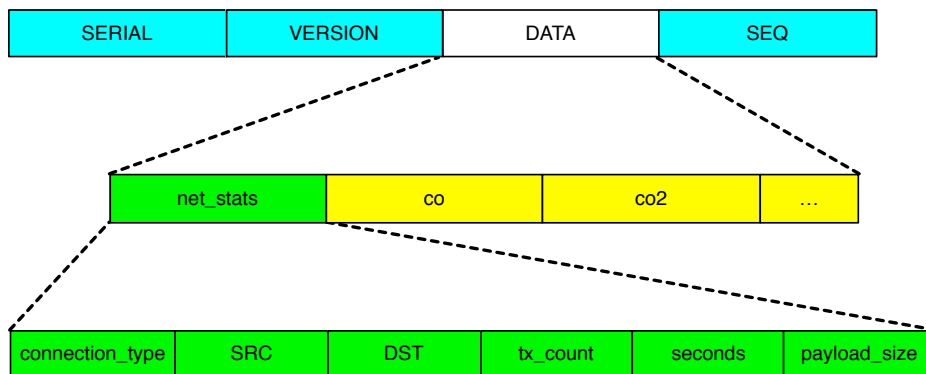


Figure 4.3: Sensors Metadata Structure

4.3.2 Acknowledgement data

Acknowledgement data contains data about the transmission and data about sensors data uploading attempt to the cloud *UrbanSense* database. Figure 4.4 shows the acknowledgement data structure. The cyan fields corresponds to transmission information and orange fields corresponds to acknowledgement data. The *SEQ* field stores the transmission sequence number and acknowledgement data has four parameters: (1) *FAIL* - number of sensor measurement samples that failed to write in database; (2) *NEW* - number of sensor measurement samples different from already stored ones; (3) *UNK* - number of sensor measurement samples unknown; and (4) *REC* - number of sensor measurement samples recorded.

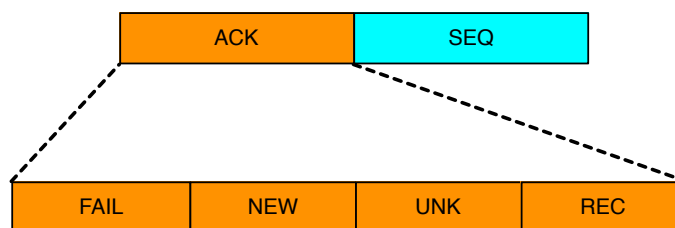


Figure 4.4: Sensors Acknowledgement Data Structure

4.4 Sensor Samples Management

Sense units locally stored sensor samples can be deleted, only after they have been successfully uploaded to the cloud *UrbanSense* database. Moreover, transmitted sensor samples can not be stored eternally. To accomplish these two goals, sense units have a *sensor samples management* table in the local database, which stores information about transmitted sensor samples. That information permits to identify which sensor samples can be deleted when an acknowledgement data

bundle is processed, identifying which sensor samples are expired and retransmitting sensor samples a pre-defined number of times, after sensor samples timeout. When sensor samples reach the maximum number of retransmissions, they are deleted from sense units local database.

Table 4.1 shows sense units *sensor samples management* table. The red rectangle highlights two table entries that correspond to two types of sensor samples transmitted in a single bundle. The *id* column stores the transmission identifier. The *sequence* column stores a sensor sample transmission sequence number. All sensor samples transmitted in the same bundle have the same sequence number. When an acknowledgement data bundle arrives, the sequence number is used to identify which sensor samples can be deleted from the local database. The *sampleType* variable identifies the type of a sensor sample. The *seconds* variable stores sensor samples acquiring time instant. The *timeout* variable stores the time instant sensor samples expire. The *txCnt* variable stores how many times sensor samples have already been sent.

Sensor samples management table entries are created, updated and deleted in accordance with bundles creation and retransmission, as well as acknowledgement data bundles processing. A new entry per each sensor sample in a bundle is added, every time a new bundle is created. All entries that correspond to samples transmitted in a bundle are updated, if that sensor samples are retransmitted. All entries that correspond to sensor samples sent in a bundle are deleted, if an acknowledgement data bundle confirms that those sensor samples were successfully uploaded to the cloud *UrbanSense* database. Entries involved on this operations are determined through their sequence number.

id	sequence	sampleType	seconds	timeout	txCount
349	9	8	1406645458	2014-07-29 14:52:39.422931+00	1
350	9	9	1406645458	2014-07-29 14:52:39.422931+00	1
347	8	8	1406645362	2014-07-29 14:52:49.855998+00	2
348	8	9	1406645362	2014-07-29 14:52:49.855998+00	2
345	7	8	1406645272	2014-07-29 14:53:00.275209+00	3
346	7	9	1406645272	2014-07-29 14:53:00.275209+00	3
351	10	8	1406645489	2014-07-29 14:53:11.102923+00	1
352	10	9	1406645489	2014-07-29 14:53:11.102923+00	1

Table 4.1: Sensor Samples Management Table

4.5 Summary

This chapter presented our proposed architecture to reliably send sensors data to the cloud *UrbanSense* server. It relies on the use of a *bundle protocol* implementation, as specified in RFC 5050 [16], and an application running on cloud *UrbanSense* server, which processes sensors data and generates an acknowledgement data report about data uploading success or failure. That application receives and sends data through a [TCP](#) socket. Moreover, the sensors data and the

acknowledgement data format are specified. Sensors data term refers to sensor samples, metadata and transmission information, which is sent in sense units to cloud *UrbanSense* server direction. Finally, sense units *sensor samples management* table functions are described. It identifies sensor samples sent in each bundle through a *sequence number*, making the ability to delete sensor samples only after an acknowledgement data bundle confirms it was successfully uploaded to cloud *UrbanSense* database.

Chapter 5

Implementation

After laying down the conceptual groundwork in the previous chapters, this chapter presents the implementation of the two opportunistic communication applications developed in the scope of this dissertation. Sense units and server applications use the open-source IBR-DTN protocol ¹ as a framework to able developed applications to send data over a DTN. First, the implementations details are presented, specifying the hardware and software used in this dissertation work. Second, the IBR-DTN *API bundle register* and *API* commands are detailed, in order to the reader easily understand the developed applications, as well as the applications methods to parse data received from the IBR-DTN *API* are explained. Final, sense units and cloud *UrbanSense* server applications flow charts are presented and explained applications operation in minute detail.

5.1 Implementation Details

The IBR-DTN protocol runs on all devices involved in opportunistic sensors data transmission from sense units to the cloud *UrbanSense* server. In sensors data transmission intermediary nodes, it is not necessary to create further software to make data flow from the source to the destination. IBR-DTN protocol implements a store and forward mechanism to accomplish that and has a configuration file that permits to configure IBR-DTN daemons. Sense units IBR-DTN daemons configuration file, can be seen in appendix A.1. Among other things, it permits to configure the convergence layers, routing protocol, as well as security and time synchronisation mechanisms.

The sense unit and the cloud *UrbanSense* server applications developed in the scope of this dissertation are in the edges of the DTN. They are a tool to add and remove bundles from the DTN and implement the sense units and cloud *UrbanSense* reliably sensors data transmission. The two applications were developed in python programming language and use IBR-DTN protocol as a framework to communicate through the DTN. Sense units also uses the *Django* framework to

¹<https://trac.ibr.cs.tu-bs.de/project-cm-2012-ibrdtm/wiki>

access to the local database. The local database structure and the code to insert and to extract sensors data from the local database was developed by a member of the *Future Cities* project.

Sense units have a Raspberry Pi model *B* running the operating system version 7, (*Wheezy*) and have a *TP-link TP-WN722N* model wireless interface. Controlled environment scenarios virtual machine have installed the Linux operating system version 12.10, *Quantal Quetzal*. Cloud *UrbanSense* server have installed the Linux operating system version is 12.04.4 LTS, *Precise Pangolin*. All devices have installed Python programming language version 2.7.3. All devices have installed IBR-DTN version 0.12.0.

5.2 IBR-DTN API

The IBR-DTN, presented in chapter 3.3.2, is an open-source implementation of the *bundle protocol* RFC 5050 specification [16]. IBR-DTN implementation was specifically developed for embedded systems and offers a **TCP** socket based **API**, in order to applications interact with IBR-DTN daemons. The **API** documentation is available on-line in the IBR-DTN project wiki page ². The IBR-DTN **API** commands presented in the following subsections were not developed in the scope of this dissertation.

The sense unit and cloud *UrbanSense* server applications establish a **TCP** socket connection to IBR-DTN daemons **API** running on their machines and interaction between them is made through commands. Our applications are clients and the IBR-DTN **API** is a server. Whenever the **API** receives a command from a client, it responds a status code. As an example, if the application sends the *protocol extended* command, the **API** responds *200 API_STATUS_OK* in case of success, or *400 ERROR* in case of failure. Binary or plain format can be used to input and output data from a daemon. Due to its simplicity, our applications use plain format. IBR-DTN daemons may receive bundles that are destined to him and bundles that it should store and transfer to another IBR-DTN daemons, when opportunistic contacts happens. IBR-DTN **API** implements a bundle notification mechanism to notify IBR-DTN daemons, whenever a bundle to a registered destination **EID** arrives. That message is of the form *602 NOTIFY BUNDLE <timestamp> <seq_nr>[<fragment_offset>]<source_eid>*.

5.2.1 Bundle Register

IBR-DTN daemons use "registrations" to define particular bundles destination **EIDs** it want to be notified about. Only this way, it is notified and is able to access to bundles content. If a bundle destination **EID** matches a registered **EID**, that bundle is queued on a first in first out queue of bundles waiting to be processed. The *bundle register* is used to add or remove bundles from the daemon. For instance, if we want to get a bundle data, we use the *bundle load queue* command to load the first bundle in the queue to the *bundle register*. After that, we can use the *bundle get*

²<https://trac.ibr.cs.tu-bs.de/project-cm-2012-ibrdsn/wiki/docs/api>

plain command, which sends the bundle content to the client in plain tthe format. The following subsection presents further API commands.

5.2.2 Commands

The IBR-DTN API commands are divided in accordance with their function. Commands that apply to the bundle in *bundle register* start with *bundle* or *payload*. Registration management commands start with *registration*. Additionally, there are *set*, *neighbor* and *protocol* commands. Our applications use the following commands to interact with IBR-DTN daemons:

Protocol Command

protocol extended Changes the IBR-DTN API mode to extended. Otherwise, the following commands cannot be used.

Neighbor Command

neighbor list Shows the singleton EID of IBR-DTN daemons in the neighbourhood. Bundles sent to those EIDs reach their IBR-DTN daemon in a single hop.

Set Command

set endpoint <endpoint_affix> Configures the singleton EID identifier. The *endpoint_affix* is concatenated to the EID of the daemon. For example, if the daemon EID is "dtn://raspberrypi" and the *endpoint_affix* is "nodeA", the singleton EID would be "dtn://raspberrypi/nodeA".

Registration Command

registration add <endpoint> Adds an EID to a registration list. Every time a bundle destination EID matches an EID in the registration list, it is queued and applications have access to their content. For instance, "dtn://cloudServer/sensorsdata" is a valid EID. This command permits an application to be notified about bundles sent to group EIDs, as well as to unique EIDs.

Bundle and Payload Commands

bundle load queue Loads the bundle in the first position of the bundle queue to the *bundle register*. If it loads successfully, API will answer *bundle load <timestamp> <seq_nr> [<fragment_offset>] <source_eid>*. The cloud *UrbanSense* application parses this API answer of all bundles it processes. It stores sensors data bundles source EID until receive their acknowledgement data about sensors data uploading success or failure, and send it to sensors data bundles source EID.

```

1 Processing flags: 16
2 Timestamp: 0
3 Sequencenumber: 0
4 Source: dtn:none
5 Destination: dtn:none
6 Reportto: dtn:none
7 Custodian: dtn:none
8 Lifetime: 3600
9 Blocks: 1
10
11 Block: 10
12 Flags: LAST\char`_BLOCK REPLICATE\char`_IN\char`_EVERY\char`_FRAGMENT
13 Length: 5
14 Encoding: base64
15
16 gYXksjc=

```

Listing 5.1: IBR-DTN API Bundles Input/Output Format

bundle get plain Sends the entire bundle in the *bundle register* to the client, in plain format. Before sending the bundle, API sends a `200 BUNDLE GET PLAIN <timestamp> <source_eid>` message. Listing 5.1 shows an example of a bundle content. Lines one to nine corresponds to the bundle header, which is the first block, and lines eleven to sixteen corresponds to the payload block, which contains the sensors data or acknowledgement data. If security mechanisms were used, bundles would have additional blocks.

Bundle header parameters were explained in chapter 3.2.3. Payload Block header parameters are: (1) *Block*, line eleven - stores an *int* value that identifies the block number; (2) *Flags*, line twelve - is a space separated flag specifier that depends on the delivery options chosen and if the block is the last or not. Delivery options were presented in chapter 3.2.1. Developed applications only use the *LAST_BLOCK* flag; (3) *Length*, line thirteen - stores the length of data on that block; (4) *Encoding*, line fourteen - identifies the encoding scheme used to encode data. Line sixteen is data encoded in base sixty four scheme.

bundle clear Clears the bundle in the *bundle register*.

bundle free Similarly to bundle clear, it clears the bundle in the *bundle register*, but also removes it from the *bundle storage*.

bundle put plain Adds a bundle to the *bundle register* in plain format. After *bundle put plain* command, application have to send each bundle line in the same format as the bundle example in listing 5.1.

bundle send Sends the bundle in *bundle register*. In fact, the bundle is stored in the *bundles storage* and sent to another IBR-DTN daemon when an opportunistic contact happens.

```

1
2 def get_bundle_data(self):
3     data = []
4     while True:
5         line = Globals.fsock_thread.readline()
6         if line == "\n":
7             return "".join(data)
8             payload.append(line)
9
10 def parse_headers(self, headerstr):
11     headers = {}
12     for line in headerstr.splitlines():
13         k, _, v = line.partition(":")
14         headers[k] = v.strip()
15     return headers

```

Listing 5.2: *Get_Bundle_Data* and *Parse-Headers* Methods

payload <block number> get Sends the bundle block *<block number>* in the *bundle register* to the client.

5.2.3 Parsing Bundles from IBR-DTN API

IBR-DTN API command answers can have different types of data. As a result, two methods to parse IBR-DTN API data were developed in the scope of this dissertation, in order to developed applications are able to parse API answers data. Bundles always have a bundle header and can have a variable number of data blocks with a header each. Header information is represented of the form: the description of the variable followed by ":" and the variable data. For instance: *Processing flags: 16*. As represented in listing 5.1 line 10, an empty line delimits a bundle header and a block header, a block header and block data, line 15, as well as a end of a block data and a start of a block header, not represented in listing 5.1.

Before parsing a bundle a header information, it is necessary to get that part of the bundle. Applications developed use the method *get_bundle_data* in listing 5.2, to read every line of the bundle until it finds a "\n" character, and store all lines in a *list* type variable. Then, applications use the *parse_headers* method in listing 5.2 to parse the bundle data returned by *get_bundle_data* method. It finds the ":" character in each *list* line and adds each word pair before and after ":" to a dictionary. Parsing blocks header data procedure is the same of parsing bundle header data. Applications also use the *get_bundle_data* method to get the blocks data.

5.3 UrbanSense Sense Units Application

Sense Units have to connect to OBUs, send sensors data, store information about transferred sensor samples in the local database and delete sensor samples after an acknowledgement data bundle confirms it was successfully upload to cloud *UrbanSense* database. Types of data sent on each

direction were discussed in chapter 4.3 and *sensor samples management* table was presented in chapter 4.4. First, this section presents the sense units clock synchronisation mechanism, when sense units do not have Internet access through hotspots. Second, the sense units and OBU connection test is detailed. This test is carried out in order to trigger bundles transmission to OBUs and to solve a problem related with sense units wireless interface Internet Protocol (IP) configuration. Finally, it presents the data transmission protocol flowchart, which sums up how the entire sense unit applications work.

5.3.1 Sense Units Clock Synchronisation

Clock synchronisation is an important aspect in the system, because sense units have to tag the time instant sensors data is acquired and the time instant bundles are created accurately. In scenarios where sense units have Internet access, they can synchronise their local clock through a Network Time Protocol (NTP) server. However, in situations it is not possible, clock synchronisation is accomplished through the IBR-DTN feature that permits to synchronise sense units local clock through the IBR-DTN clock. Table 5.1 depicts OBUs and sense units IBR-DTN time synchronisation configurations. Due to OBUs ability to synchronise their clock through 3G and GPS, they are configured as master and sense units are configured as slave. *Time Set Clock* configuration parameter set to yes, permits to use the IBR-DTN clock to set the sense units local clock. It is possible only if IBR-DTN daemon is running as root. Otherwise, sense units have not permissions to change the local clock time. The entire IBR-DTN configuration parameters are in appendix A.1.

	Master (OBU)	Slave (Fixed Sensor)
Time Reference	Yes	No
Time Synchronise	Yes	Yes
Time Discovery Announcements	Yes	Yes
Time Set Clock	No	Yes

Table 5.1: IBR-DTN Time Synchronisation Configuration

5.3.2 Sense Unit and OBU Connection Test

Sense Units and OBUs connection serve two purposes: (1) when connections between sense units and OBUs fails, drop sense unit IP received from OBUs Dynamic Host Configuration Protocol (DHCP) server, and (2) check if sense units are connected to OBUs and trigger bundles transmission. Only when sense units are connected to OBUs, they attempt to transmitting sensors data. The wireless interface has to be turned off and turned on, in order to drop the configured IP address when sense units and OBU are not connected. Figure 5.1 shows the flowchart of the

method that permits realise if the sense units wireless interface should be "restarted" or not. First, a sense unit tries to connect to the vehicular network announced SSID. If it fails, the sense unit keeps trying. If connection succeeds, the sense unit tries to ping the OBU interface. IF ping succeeds, it means that the sense unit is effectively connected to the OBU and it can send bundles to the OBU. On the contrary, if ping fails, the sense unit wireless interface is turned off and turned on. After that, the sense unit tries to connect to an OBU again.

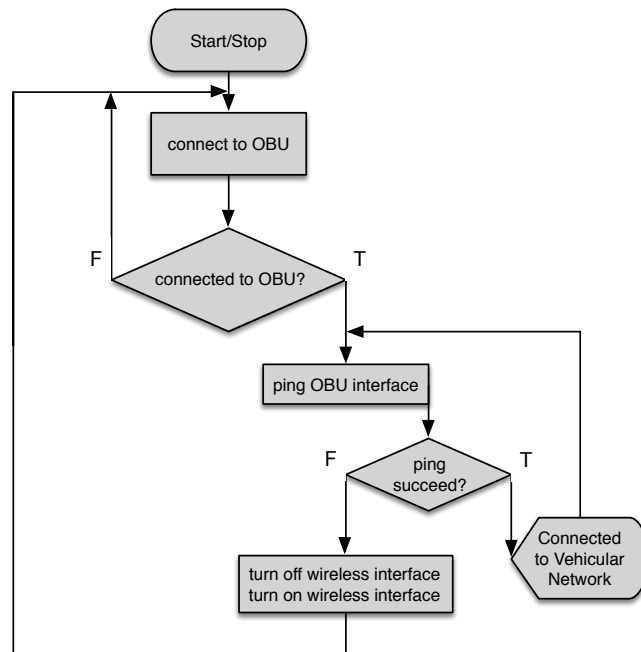


Figure 5.1: Sense Unit and OBU Connection Test

5.3.3 Sensors Data Transmission

Sense units application is divided in two threads: one to send and other to receive bundles. It has this architecture because it uses blocking functions. Otherwise, it would not be able to send and receive bundles, at the same time.

Figure 5.2, in the left side branch, shows the sending thread. After a IBR-DTN daemon is running, sense units application establish a TCP socket connection to the IBR-DTN daemon running on its machine, sends the *protocol extended* command to change the daemon's API to extended mode and sends the *set endpoint <endpoint_affix>* command to configure the sense unit singleton EID (Figure 5.2 box 1). Then, it uses the *sense unit and OBU connection test* to verify if it is connected to a vehicular network OBU (Figure 5.2 box 2). If the test fails, it keeps trying to connect vehicular network OBUs. On the contrary, if the test succeeds, from the connection point of view, sense units can send bundles to the OBU. Bundles transmitted in sense units to OBUs direction, encapsulate sensor samples, metadata and transmission information data.

Expired transmitted sensor samples have higher priority to be transferred to cloud *UrbanSense* server than sensor samples that have never been sent. To accomplish that, the application checks if sensor samples number of simultaneous transmissions have not reached the *maxSimultTX* value and if there is expired sensor samples to send (Figure 5.2 box 3). If it is true, expired sensor samples number of transmissions counter is checked (Figure 5.2 box 4). If it is higher than the *MAX_TX_COUNT* value, sensor samples are dropped (Figure 5.2 box 5). Otherwise, metadata, sensor samples and transmission information is encapsulated in a bundle and sent to the opportunistic contact *OBU* (Figure 5.2 box 6). At that instant, *sensor samples management* table *txCnt* and sensor samples timeout values are updated (Figure 5.2 box 7).

If sensor samples number of simultaneous transmissions have not reached the *maxSimultTX* value and if there is not expired sensor samples to send (Figure 5.2 box 3), the application tries to send sensor samples that have never been transmitted to *OBUs* (Figure 5.2 box 8). To accomplish that, it checks if there is sensor samples in database that have never been sent (Figure 5.2 box 9). If the check fails, the program goes to the beginning. If check succeeds, metadata, sensor samples and transmission information is encapsulated in a bundle and sent to the opportunistic contact *OBU* (Figure 5.2 box 10). At that instant, *sensor samples management* table is updated (Figure 5.2 box 11).

Figure 5.2, in the right side branch, shows the receiving thread. Similarly to the sending process, it first establishes a *TCP* socket connection to the *IBR-DTN API*, changes the *API* to extended mode and configures the singleton *EID* (Figure 5.2 box 12). If the application uses the same *TCP* socket connection to the *IBR-DTN API* for sending and receiving bundles, the *API* would use the same *bundle register*, which may cause conflicts.

After the the initial *API* configuration, the application has to be aware of bundles that may arrive. To accomplish that, it sends the *bundle load queue* command to the *IBR-DTN API* (Figure 5.2 box 13). If there is not bundles available, the *API* answers *400 ERROR* and application goes to (Figure 5.2 box 13) again. If there is bundles available, the *API* answers *bundle load <timestamp> <seq_nr> [<fragment_offset>] <source_eid>* (Figure 5.2 box 14). In that case, the application gets the bundle *source EID* through the parsing of the *bundle load queue* command answer (Figure 5.2 box 15). After that, the application has to get the acknowledgement data from the bundle. To accomplish that, the application sends the *payload 100 get* command to the *IBR-DTN API* and it answers the data block number 100 (Figure 5.2 box 16). The acknowledgement data is always in the bundle last block. Even if bundle last block is not the block number 100, the last block number is for sure less than 100. When this situation happens, *API* sends the block with higher number, which is the bundle last block. After application gets the bundle last block, which contains the acknowledgement data, the bundle is deleted from the daemon *bundle register* and daemon *bundle storage* through the *bundle free* command (Figure 5.2 box 17).

At this point, acknowledgement data needs to be processed (Figure 5.2 box 18). If *FAIL* and *UNK* acknowledgement data variables values are equal to zero, it means that sensors data was successfully uploaded to cloud *UrbanSense* database. Consequently, sensor samples can be deleted from the sense unit local database and *sensor samples management* table entries that correspond

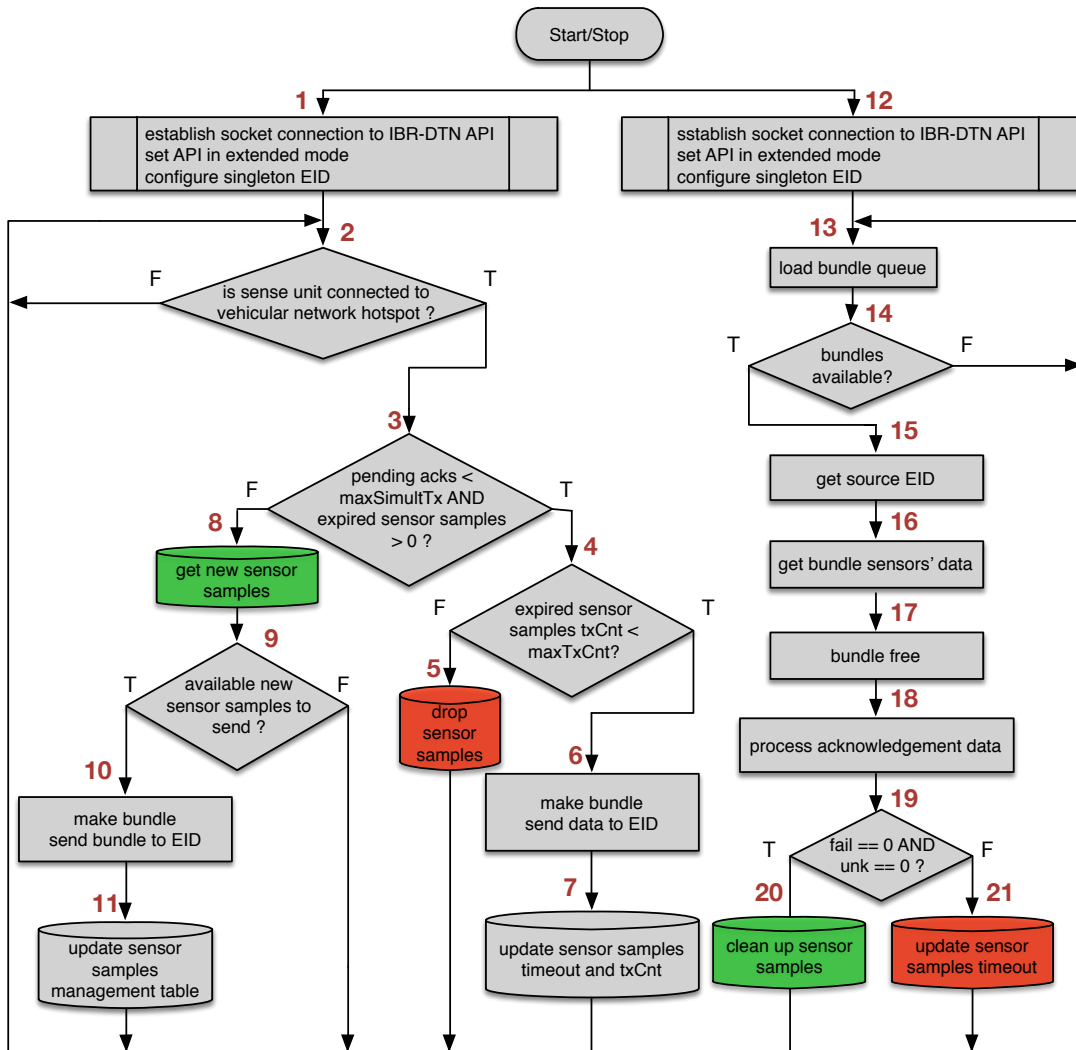


Figure 5.2: Sense Units Application Flowchart

to these sensor samples can be deleted too (Figure 5.2 box 20). On the contrary, If *FAIL* and *UNK* variables values are not equal to zero, the sensor sample entries in *sensor samples management* table that correspond to that bundle are set as expired.

5.4 UrbanSense Server Application

The cloud *UrbanSense* server application works as the interface between the *DTN* and Internet. Conceptually, (1) the server application receives bundles from the vehicular network; (2) gets sensors data from bundles and send it to the application that processes it through a *TCP* socket; (3) waits for the sensors data correspondent acknowledgement data; and (4) finally sends the acknowledgement data to the sense unit that originated that sensors data. Figure 5.3 shows the cloud

UrbanSense server application flowchart. When it is started, it establishes a socket to the IBR-DTN API, set the API in extended mode and configures the singleton and group EIDs. The group EID is the sense units bundles destination EID. Furthermore, it establishes a socket connection to the application that accepts TCP socket connections, process JSON format sensors data and answers acknowledgment data about sensors data uploading success or failure (Figure 5.3 box 1).

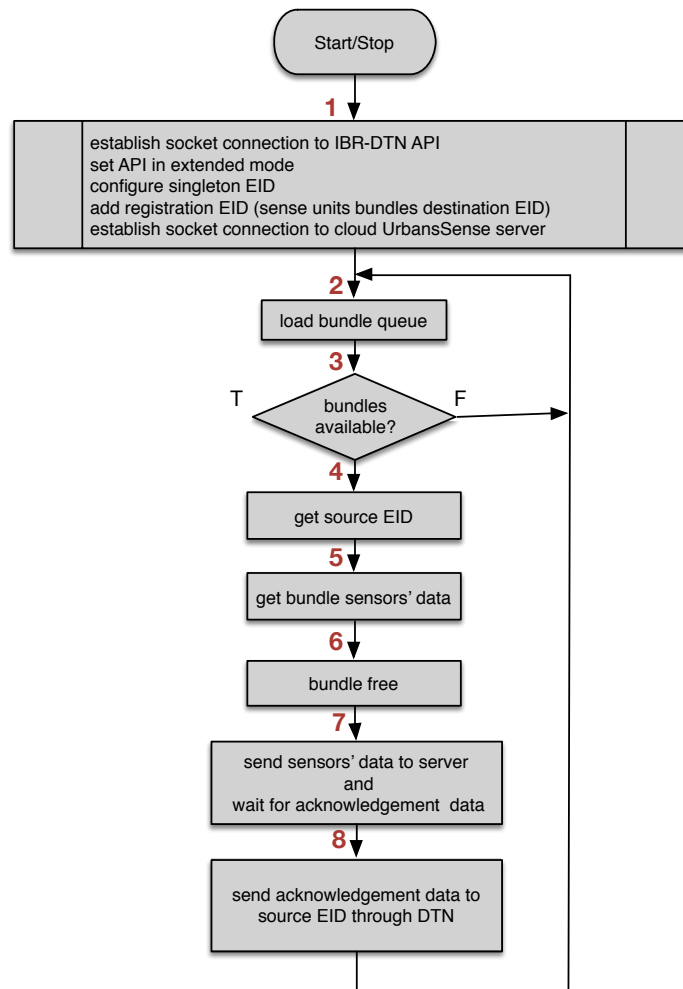


Figure 5.3: Cloud *UrbanSense* Server Application Flowchart

The cloud *UrbanSense* application code correspondent to Figure 5.3 boxes 2, 3, 4, 5 and 6 is equal to the code used in sense units application in chapter 5.3.3 and its detailed explanation is not repeated in this sub-section. After the initial configuration, application loads the bundle queue until there is a bundle available (Figure 5.3 boxes 2 and 3), gets the bundle *source EID* and stores it (Figure 5.3 box 4). This is the EID that the application has to send the acknowledgement data. After that, the application gets the sensors data from the bundle (Figure 5.3 box 5) and delete the bundle from *bundle register* and *bundle storage* (Figure 5.3 box 6). At this point, the application sends sensors data to the application that process sensors data in JSON format and waits for the

acknowledgement data correspondent to sensors data uploading to the cloud *UrbanSense* database (Figure 5.3 box 7). When the acknowledgement data is received, the application encapsulates the acknowledgement data in a bundle and sends it to the **EID** of the sense unit that originated the sensors data and the program sequence ends (Figure 5.3 box 8).

5.5 Summary

This chapter details the two applications that run in sense units and cloud *UrbanSense* server, developed in the scope of this dissertation. These applications able us to reliably send sensors data from sense units to the cloud *UrbanSense* server using the vehicular network. Firstly, the implementation details are presented, including the software and hardware used on this dissertation. Secondly, the IBR-DTN **API** is presented, namely the *bundle register*, IBR-DTN **API** commands and bundles parsing methods developed in the scope of this dissertation. IBR-DTN **API** does the interface between developed applications and IBR-DTN daemons. Thirdly, sense units application implementation is presented, explaining every step involved in sensors data transmission and acknowledgement data reception and processing. Finally, server application flowchart is presented and explained how it does the "conversion" from **DTN** to **TCP** and **TCP** to **DTN**.

Chapter 6

Measurements and Analysis

This chapter presents the validation of the architecture and implementation described earlier and shows how the entire system performs in a real-world environment scenario. First, the two controlled environment test scenarios are presented. These scenarios were used to carry out a set of basic tests. Second, we describe the real-world environment tests carried out to validate the developed applications and understand the overall system performance. Final, the results obtained are presented and discussed.

6.1 Controlled Environment Tests Scenarios

Preliminary IBR-DTN configuration and applications controlled environment tests were performed prior to the real-world environment test, because that way is easier to debug and improve applications without compromise other services utilisation. Moreover, the IBR-DTN had being installed in the vehicular network and cloud *UrbanSense* server, when the implementation described in 5 was in its final stage of deployment. To cope with that, two controlled environment test scenarios were created.

Figure 6.1 depicts the first controlled environment test scenario. The left side, represents the sense unit local database, Raspberry Pi and Wi-Fi interface configured in client mode. In this controlled environment test scenario, instead of using sensors data acquired from real sensors, the sense unit local database tables correspondent to the different types of sensor samples were populated with random values. The Figure 6.1, on right side, does not represent any of the devices described in our proposed architecture. The Linux virtual machine runs the software that is supposed to run in the cloud *UrbanSense* server and has an Wi-Fi interface configured in managed mode, which is the **OBU**s Wi-Fi interface configuration. The Wi-Fi interface managed mode configuration was accomplished through a *hostapd* daemon application. Both the sense unit and the Linux run the IBR-DTN daemon in the Wi-Fi interfaces. As detailed in proposed architecture chapter 4.2, IBR-DTN runs in sense units Wi-Fi interface, **OBU**s Wi-Fi and 802.11p interface, **RSU**s

802.11p and optical fiber interfaces and cloud *UrbanSense* server Ethernet interface. Once the sense unit connects to the Linux wireless interface, both IBR-DTN daemons became neighbours and are able to exchange bundles between them. The sense unit and Linux run the applications developed to send data using the IBR-DTN protocol as a framework. The sense unit application sends the locally stored data to the Linux IBR-DTN daemon and the application developed to run in the server, extracts sensors data from bundles and sent it to the application that process sensors data through a **TCP** socket. That application stores sensors data in a Linux database and answers the acknowledgement data. The developed server application sends the acknowledgement data to the sense unit through the **DTN**. The sense unit application deletes the locally stored data that the received acknowledgement data bundle reports to had been successfully uploaded to the Linux database.

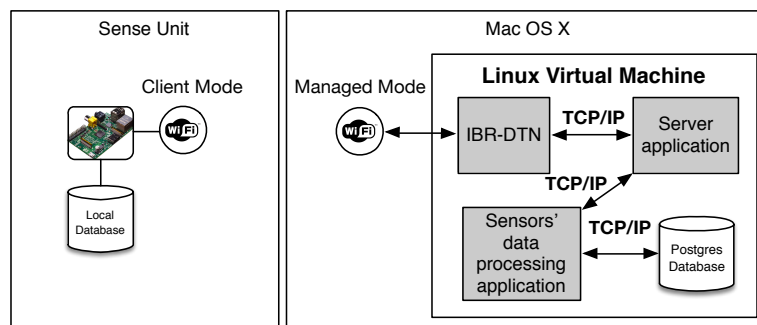


Figure 6.1: First Controlled Environment Test Scenario

Figure 6.2, shows the second controlled environment test scenario. In this setup, the Linux virtual machine can be looked as if it was a **RSU**. The only difference from a real one is that it runs in a different hardware, communicate through a Wi-Fi 802.11n interface instead of 802.11p and communicates with a sense unit instead of an **OBU**. Similarly to the first scenario, it uses a sense unit and a Linux virtual machine, but the server application developed in the scope of this dissertation and the application that process sensors data run in the cloud *UrbanSense* server. Furthermore, the database and IBR-DTN daemon run in the cloud *UrbanSense* server. At this point, cloud *UrbanSense* server runs all software modules it runs in the real-world environment scenario. This test environment was very useful because it permitted to test the statical connection IBR-DTN feature between the Linux and the cloud *UrbanSense* server. The statical connection IBR-DTN feature permits to establish a **TCP** connection between **RSUs** and the cloud *UrbanSense* server, making the ability to both IBR-DTN daemons became neighbours and exchange bundles over the Internet. In this controlled environment test scenario, cloud *UrbanSense* runs the same application it runs in real-world scenario.

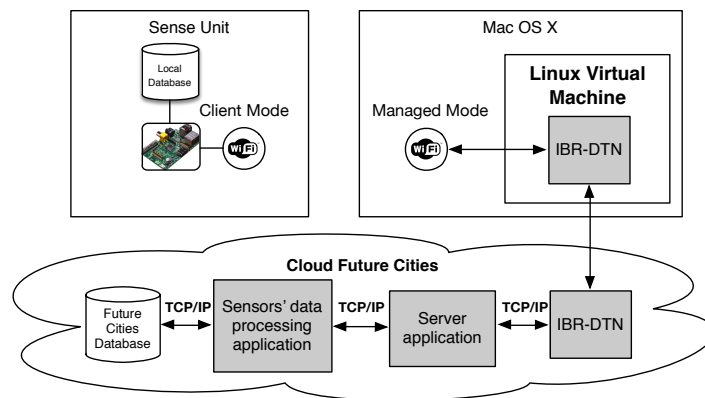


Figure 6.2: Second Controlled Environment Test Scenario

6.2 Real-World Environment Applications Validation

At the time of this dissertation submission, IBR-DTN had not been running on the vehicular network yet. Therefore, an **OBU** similar to those installed on buses was installed on a private car to do this test. Figure 6.3 shows the test scenario in the city of Porto. The sense unit platform was installed on a traffic light of Damião de Gois street, the **OBU** was installed in a car that followed the path represented by the dot-slash line, and the **RSU** was the one installed on a traffic light of Marquês square. Dotted ellipses on Figure 6.3, represent the approximate wireless communication range of the sense unit, **OBU** and **RSU**.

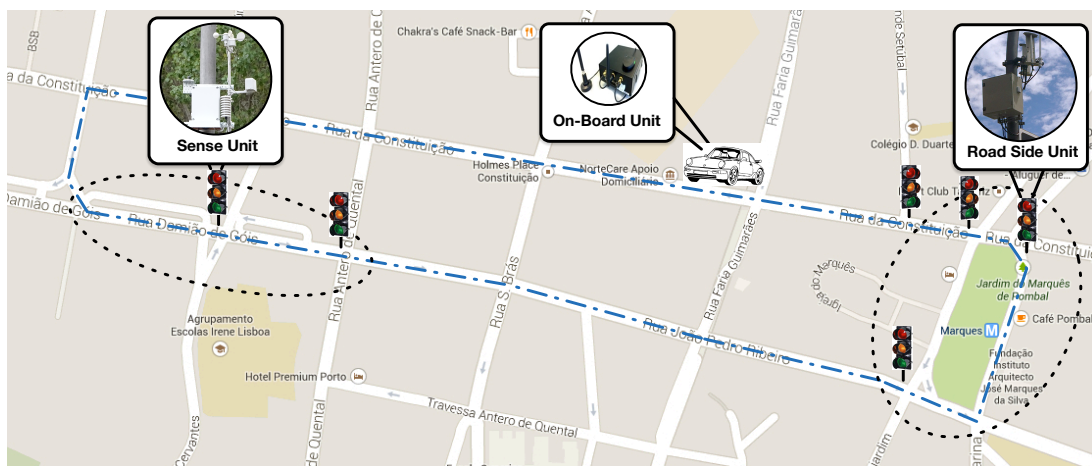


Figure 6.3: Real-Word Environment Test Scenario

The sensor samples acquisition interval was 60 s. The system was configured to send data just over the **DTN** and the test took 5525 s. Table 6.1 shows the sense unit configuration parameters. The sense unit is configured to connect to the SSID **VENIAM_TEST** announced by the **OBU**

installed on a private vehicle. The maximum number of attempts to transmit each sensor sample was 3 times, the sensor samples expired after 1 hour and the maximum number of bundles in transit was 1000.

Fixed Sensors Application	
Source EID	dtn://raspberrypi/nodeA
Destination EID	dtn://sensorsdata/serverA
Bundle Timeout	3600 (seconds)
Max. Tx. Attempts	3
Max. Simultaneous Tx.	1000
OBU SSID	VENIAM_TEST
IBR-DTN API HOST	127.0.0.1
IBR-DTN API PORT	4550

Table 6.1: Sense Unit Application Configuration Parameters

6.3 Results

These results presented in the current section were obtained through the analysis of bundles metadata information, sense unit and server applications log, obtained in the real-world environment test, detailed in chapter 6.2. The real-world environment test was conducted using one sense unit, one **OBU** and one **RSU**. The first two sub-sections refer to the bundles delay and percentage of sensor samples retransmissions and the last sub-section presents the test overview.

6.3.1 Bundles Delay

Bundles delay is the time difference between a bundle creation and its cloud *UrbanSense* database uploading instants. The instant of a bundle creation is added to the bundles metadata immediately before bundles are transmitted from a sense unit to a **OBU**. The instant a bundle is uploaded to the cloud *UrbanSense* database is automatically filled when sensors data is uploaded to the cloud *UrbanSense* database. The sense unit builds bundles after the sense unit and **OBU** establish a connection and then the sense unit attempt to transmit them to the **OBU**.

Table 6.2 shows further information related with bundles delay and the test total number of transmitted bundles. The minimum delay was 27 s, the average delay was 141 s and the maximum delay was 254 s, in a total of 250 bundles.

Figure 6.4 shows the histogram of delays experienced in bundles transmitted from sense units to cloud *UrbanSense* server. Six equally spaced delay intervals are represented along the x-axis,

Number of Bundles	250
Bundles Minimum Delay	00:00:27.00 (H:M:S)
Bundles Mean Delay	00:02:20.94 (H:M:S)
Bundles Maximum Delay	00:04:17.00 (H:M:S)

Table 6.2: Bundles Delay Information

whereas y-axis represents the number of bundles that experienced delays within that interval. The histogram shows that bundles delay in this test had a bimodal distribution. The median values were 96 s and 188 s.

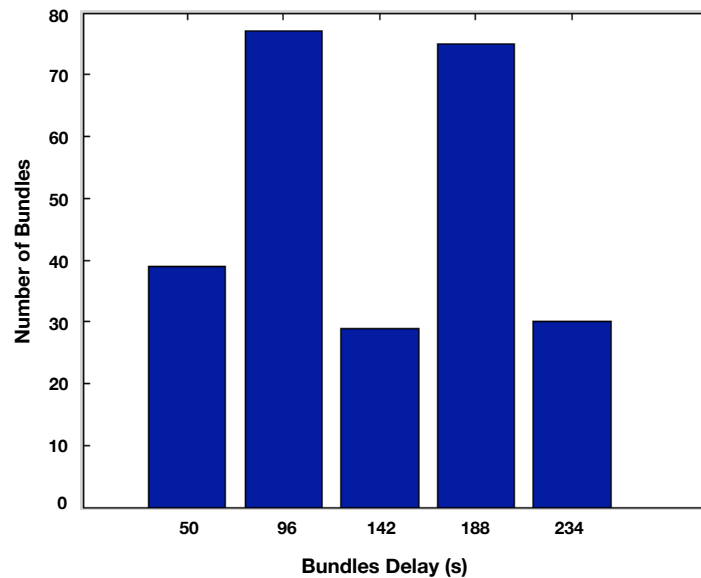


Figure 6.4: Bundles Delay Histogram

6.3.2 Bundles Transmissions

The percentage of bundle transmissions was calculated through the *txCnt* metadata information. Discussed in chapter 4.4, it is a *sensor samples management* table variable that identifies to which sensors data transmission attempt a bundle corresponds. If the acknowledgement of a transmitted bundle did not arrive within 3600 s and the maximum of *MAX_TX_COUNT* transmission attempts had not been reached, a bundle was retransmitted. After reaching the *MAX_TX_COUNT* number, data was discarded. Figure 6.5 shows the percentage of bundles that corresponded to the first, second and third transmission attempts. The bundles that corresponded to second and third transmissions had already been transmitted over TCP and unacknowledged before our test started. It happened because the sense unit we used in the test had been sending data over TCP, before the

sending over DTN test had started. The transmitted sensor samples expired after 3600 s. Consequently, in a test that took 5565 s, the same sensor sample could not have expired three times. During the sensors data transmission over DTN test, all sensors data was transmitted in a single attempt.

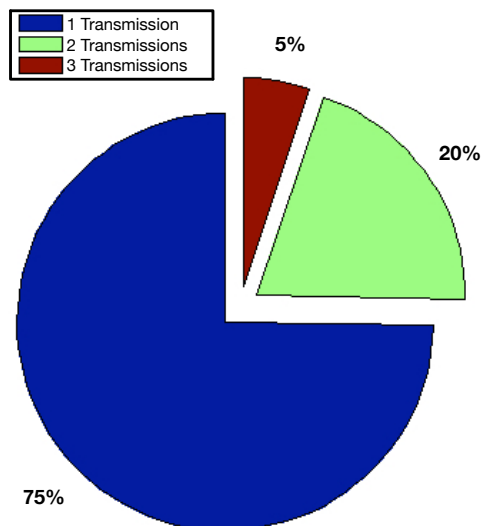


Figure 6.5: Bundles Number of Transmission Distribution

6.3.3 Real-World Environment Test Overview

The real-world environment test occurred throughout a day during 5565 seconds. Table 6.3 shows additional information about the test. The term "opportunistic contact" refers to a passage of the OBU close to the sense unit, in which may have existed more than one connection establishment moment between them. There were a total of 11 opportunistic contacts between the sense unit and the OBU on-board the vehicle and 145.85 kilobytes of sensors data was transferred from the sense unit to the cloud *UrbanSense* database.

Figure 6.6 shows the delay experienced while transmitting the bundles during the test. The y-axis represents the bundles delay and the x-axis represents the bundles creation time instant. The different marks distinguish the 11 opportunistic contacts. Bundles were created after the sense unit and the OBU had established connection. In C1, yellow circles, all bundles were created at closely instants, but a delay of approximately 140 s was verified between the two clusters of transmitted bundles. This difference existed, because the connection between the OBU and the RSU failed at the middle of the transmission and not all bundles were transmitted at the same time. In contacts C2, C3, C5, C9, C10 and C11, the sense unit transmitted at closely instants all sensor samples expired and sensor samples that had never been attempted to be transmitted and then, a few seconds later, the sense unit transferred two or three bundles more. The isolated transferred bundles correspond to sensor samples acquired after all other sensor samples had been transferred. In C4, represented by red circles, the OBU got into the sense unit range, received a

Test Started Time	18:49:53 (H:M:S)
Test Ended Time	20:22:38 (H:M:S)
Test Duration	5565 (S)
Date of Test	21-Aug-2014
Total Transferred Data	145.7 kilobytes
Number of Contacts	11

Table 6.3: Real-World Environment Test Details

few bundles, but the connection between them was temporarily lost. After the connection had been re-established, the OBU was able to receive more bundles. In C6, all bundles were transferred at once. One of the bundles created during the contact C7 was transferred only in contact C8. The connection test between the sense unit and OBU was successful and the bundle was created, but the connection failed right after that, which caused the transmission to fail. However, that bundle was transferred in the following opportunistic contact and sensors data was not lost.

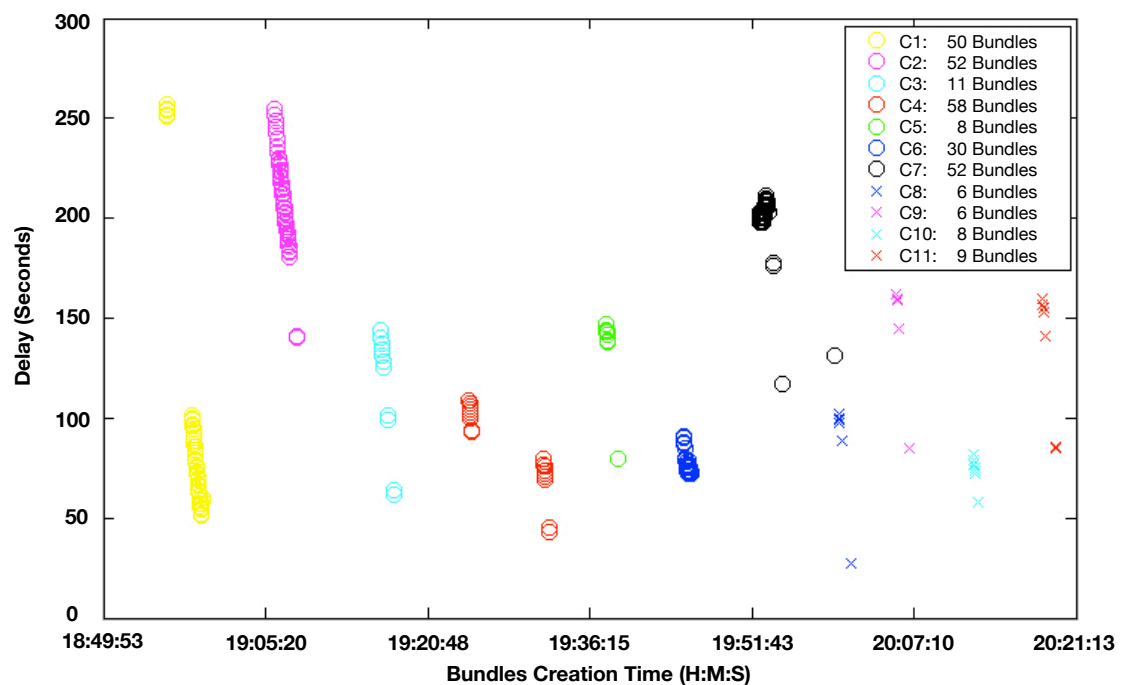


Figure 6.6: Real-Word Environment Test Overview

Table 6.4 shows the amount of sensors data transferred on each opportunistic contact and in

total, as well as the mean delay and delay standard deviation. It could had been transferred a higher amount of sensors data, in most of the opportunistic contacts. However, the sense unit sensor samples acquisition interval used in this test was 60 s, which resulted in the acquisition of a few sensor samples between consecutive opportunistic contacts. Figure 6.7, shows a bar plot of the mean delay correspondent to each opportunistic contact, presented in Table 6.4.

Opportunistic Contact	1	2	3	4	5	6	7	8	9	10	11	Total
Length of Transferred Data (kilobytes)	29.9	30.7	6.25	10.1	4.4	17.2	30.7	3.6	3.6	4.4	5	145.85
Mean Delay (s)	92.28	208.94	115	84.67	134.5	77.3	199.21	85.83	145.16	71.88	139	140.94
Standard Deviation Delay (s)	56.39	24.03	29.5	20.42	22.2	5.42	16.46	29.17	30.12	9.09	30.81	63.82

Table 6.4: Opportunistic Contacts Amount of Transferred Data, Mean Delay and Delay Standard Deviation

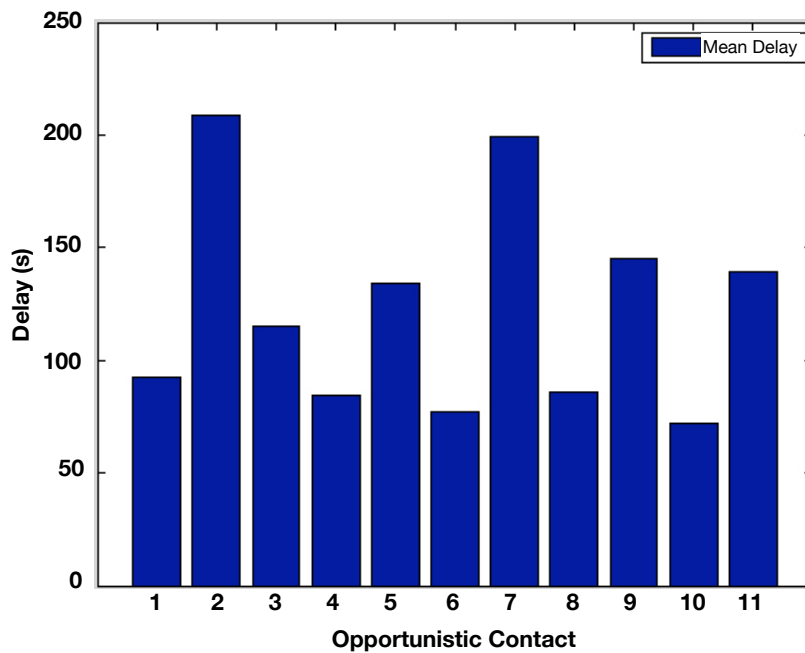


Figure 6.7: Opportunistic Contacts Mean Delay

The sense unit builds bundles with one sample of each type of sensors available data. Table 6.5 shows the quantity of bundles per bundle sensors amount of data. It shows that most of them had more than 600 bytes correspondent to sensors data. It corresponded to bundles that contained one sensor sample of each sensor type. The bundles with a lower amount of sensors data might correspond to bundles that had sensors data acquired during the opportunistic contacts. At the

beginning of the opportunistic contact all stored data was transferred and then, as local database was filled with new sensor samples, new bundles were created with less types of sensors samples.

Size of Transferred Data (Bytes)	337	339	463	511	515	608	608	610	611	612	Total
Quantity of Bundles	9	1	1	1	8	3	1	6	26	194	250

Table 6.5: Quantity of Bundles per Bundles Amount of Sensors Data

6.4 Summary

First, this chapter presents the controlled environment test scenarios that permitted to test IBR-DTN and developed applications in laboratory conditions, which took to the reduction of application development time. Second, the methodology to validate developed applications in real-world environment is described. Final, the real-world environment test results are presented and discussed. The roads traffic conditions are determinant to the bundles delay. The minimum, mean and maximum bundles delay were 27 s, 140 s and 257 s, respectively. The percentage of sensor samples transmitted at the first, second and third transmission attempts was 75%, 20% and 5%, respectively. However, the failed sensor samples transmissions attempts had been done over TCP, before the sending over the vehicular network test started. To sum up, the test had a duration of 5565 s, 11 opportunistic contacts between the sense unit and the OBU and 250 bundles where transferred, corresponding to a total 145.85 kilobytes of sensors data.

Chapter 7

Conclusions and Future Work

This dissertation work presents an approach to reliably send sense units data to the cloud *UrbanSense* server, in an opportunistic way, using the city of Porto existing vehicular network. Furthermore, in order to understand the overall system performance, metadata information is added by side sensor samples and transmission information, which is encapsulated in bundles and flows in sense units to cloud *UrbanSense* server direction. The first chapter, presents the dissertation contextualisation, discusses a few approaches to send sensors data from sense units to the cloud and presents some of the communications challenges. The second chapter, makes an overview of the *Future Cities* project, as well as details the *UrbanSense* and vehicular network platforms. The third chapter, presents the mule architecture, the DTN communication principles and the *bundle protocol*, which addresses the DTN issues. This dissertation uses the studies around DTN, as part of the solution to reliably transfer sensors data from sense units to cloud *UrbanSense* server. The chapter four, presents this dissertation proposed architecture, which involves sense units and server devices from *UrbanSense* platform, as well as OBUs and RSUs devices from the vehicular network platform. Moreover, the proposed architecture also relies on the IBR-DTN protocol, which implements the RFC 5050 *bundle protocol* specification [16], two applications developed in the scope of this dissertation, running on sense units and cloud *UrbanSense* server, and a application that runs in cloud *UrbanSense* server that process sensors data. The chapter five, details the two applications implemented on sense units and cloud *UrbanSense* server, which use the IBR-DTN as a framework to communicate over the vehicular network DTN. Finally, chapter six, presents the overall system performance results and discussion.

This dissertation work main contributions are the sense units and cloud *UrbanSense* applications, as well as the ability to understand the overall system performance through metadata statical analysis. Now, sense units can be installed in places without Internet hotspots and 3G coverage, and be able to reliably send sensors data to the cloud *UrbanSense* server. Hence, sense units still need a vehicular network around. Even in places with 3G coverage, sending sensors data in an opportunistic way may have some advantages. For instance, using opportunistic communications

instead of 3G, takes to the reduction of 3G utilisation service costs. Turning to the system performance, its analysis can be useful for adapting sensors data acquisition in accordance with moments of the day there are more vehicular network bandwidth available and understand if developed application are working properly.

Although the applications and metadata validation test was not conducted at an urban-scale, the test metadata analysis permits to conclude that the developed applications accomplishes all the proposed goals, particularly the reliable transmission of sensor data from the sense units to the cloud *UrbanSense* server in an opportunistic way. Furthermore, the test metadata analysis, permitted to have an idea of the kind of analysis that can be performed at an urban-scale. For each sense unit, metadata permits to know sense units bundles delay, percentage of sensors data retransmissions, amount of sensors data transferred in period of time and how size of sensors data in bundles varies.

7.1 Future Work

In the future work, there is plenty room for improvements on IBR-DTN utilisation, the applications developed and overall system performance analysis. For instance, IBR-DTN protocol implements a lot of pretty good functionalities that we did not test, such as proactive and reactive fragmentation, security mechanisms and data compression. Instead of using the IBR-DTN [API](#), developed applications functionalities could be embedded in IBR-DTN code. Although their performance may increase, it has the disadvantage of when a new IBR-DTN version comes out, porting everything to the new version may be hard work. Also, [DTN](#) performance analysis could be performed in both directions, from sense units to the server and in the reverse way. Finally, at this moment, sense units can run the sending sensors data over [TCP](#) and [DTN](#) applications independently or both at the same time. However, in these ways, sending sensors data over [TCP](#) is not prioritised. As future work, is necessary to implement the network manager discussed in [chapter 4](#), which integrates both applications and ables sense units to automatically choose the best interface to accomplish sensors data transmission.

Appendix A

IBR-DTN Configuration File

In [A.1](#) the IBR-DTN sense units configuration file is presented. Vehicular network devices configuration files is not presented, because their configuration is beyond the scope of this dissertation.

A.1 Sense Units IBR-DTN configuration file

```
#####IBR-DTN daemon#####  
    #local_uri = dtn://node.dtn  
logfile = /var/log/ibrdtm/ibrdtm.log  
#timezone = +1  
#limit_blocksize = 1.3G  
#limit_foreign_blocksize = 500M  
#limit_predated_timestamp = 604800  
#limit_lifetime = 604800  
#limit_bundles_in_transit = 5  
#api_socket = /tmp/ibrdtm.sock  
#api_interface = any  
#api_port = 4550  
#fragmentation = yes  
#limit_payload = 500K  
stats_traffic = no  
#blob_path = /tmp  
#storage_path = /var/spool/ibrdtm/bundles  
#storage = default  
#limit_storage = 20M  
  
#####convergence layer configuration #####
```

```
#discovery_address = ff02::142 224.0.0.142
#discovery_timeout = 5
#discovery_short = 0
discovery_version = 2
#discovery_announce = 0
discovery_crosslayer = yes
net_interfaces = lan0
#net_autoconnect = 60
#net_internet = eth0
#configuration for a convergence layer named lan0
net_lan0_type = tcp
net_lan0_interface = wlan0
net_lan0_port = 4556
#configuration for a convergence layer named lan1
#net_lan1_type = udp
#net_lan1_interface = eth0
#net_lan1_port = 4556
#tcp_nodelay = yes
#tcp_chunksize = 4096
#tcp_idle_timeout = 0
#P2P configuration#
#p2p_ctrlpath = /var/run/wpa_supplicant/wlan1

#####routing configuration#####
routing = epidemic
routing_forwarding = yes
#scheduling = no
#route1 = dtn://[:alpha:].moon.dtn[:alpha:] dtn://router.dtn
#static1_address = 192.168.150.10
#static1_port = 4556
#static1_uri = dtn://raspberrypi
#static1_proto = tcp
#static1_immediately = yes
#static1_global = yes
#static2_address = 192.168.0.10
#static2_port = 4556
#static2_uri = dtn://node-ten.dtn
#static2_proto = udp
#static1_immediately = no
```

```
#####prophet configuration#####

#prophet_p_encounter_max = 0.7
#prophet_p_encounter_first = 0.5
#prophet_p_first_threshold = 0.1
#prophet_beta = 0.9
#prophet_gamma = 0.999
#prophet_delta = 0.01
#prophet_time_unit = 1
#prophet_i_typ = 300
#prophet_next_exchange_timeout = 60
#prophet_forwarding_strategy = GRTR
#prophet_gtmx_nf_max = 30

#####bundle security protocol#####

#security_level = 0
#security_bab_default_key = /etc/ibrdsn/bpsec/default-bab-key.mac
#security_path = /etc/ibrdsn/bpsec/keys
#security_key = /etc/ibrdsn/tls/local.key
#security_trusted_ca_path = /etc/ibrdsn/tls/
#security_tls_required = yes
#security_tls_disable_encryption = yes

#####time synchronization#####

time_reference = no
time_synchronize = yes
time_discovery_announcements = yes
#time_sigma = 1.001
#time_psi = 0.9
#time_sync_level = 0.15
time_set_clock = yes

#####DHTNameService settings#####

#dht_enabled = yes
```

```
#dht_port = 9999
#dht_id = <randomstring>
#dht_enable_ipv4 = yes
#dht_enable_ipv6 = yes
#dht_bind_ipv4 = 127.0.0.1
#dht_bind_ipv6 = ::1
#dht_nodes_file = <filepath>
#dht_bootstrapping = yes
#dht_bootstrapping_domains = dtndht.ibr.cs.tu-bs.de
#dht_bootstrapping_ips = 192.168.0.1; 192.168.0.2 8888;
#dht_blacklist = yes
#dht_self_announce = yes
#dht_min_rating = 1
#dht_allow_neighbour_announcement = yes
#dht_allow_neighbours_to_announce_me = yes
#dht_ignore_neighbour_informations = no
```

References

- [1] Marc Blanchet, Simon Perreault, and Jean-Philippe Dionne. Postellation: an enhanced delay-tolerant network (dtn) implementation with video streaming and automated network attachment. 2012.
- [2] Scott Burleigh. Interplanetary overlay network: An implementation of the dtn bundle protocol. In *Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE*, pages 222–226. IEEE, 2007.
- [3] Sebastian Schildt, Johannes Morgenroth, Wolf-Bastian Pöttner, and Lars Wolf. Ibr-dtn: A lightweight, modular and highly portable bundle protocol implementation. *Electronic Communications of the EASST*, 37, 2011.
- [4] Melissa Ho Jain and Robin Patra. Implementing delay tolerant networking. December 2004.
- [5] R.C. Shah, S. Roy, S. Jain, and W. Brunette. Data mules: modeling a three-tier architecture for sparse sensor networks. In *Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*, pages 30–41, May 2003. doi:10.1109/SNPA.2003.1203354.
- [6] G. Anastasi, M. Conti, Emmanuele Monaldi, and A. Passarella. An adaptive data-transfer protocol for sensor networks with data mules. In *World of Wireless, Mobile and Multimedia Networks*, pages 1–8, June 2007. doi:10.1109/WOWMOM.2007.4351776.
- [7] A.A. Somasundara, A. Kansal, D.D. Jea, D. Estrin, and M.B. Srivastava. Controllably mobile infrastructure for low energy embedded networks. *IEEE Transactions on Mobile Computing*, 5(8):958–973, August 2006. doi:10.1109/TMC.2006.109.
- [8] Damla Turgut and Ladislau Bölöni. Heuristic approaches for transmission scheduling in sensor networks with multiple mobile sinks. *The Computer Journal*, 54(3):332–344, March 2011.
- [9] Giuseppe Anastasi, Eleonora Borgia, Marco Conti, and Enrico Gregori. A hybrid adaptive protocol for reliable data delivery in wsns with multiple mobile sinks. April 2010.
- [10] Sushant Jain, Rahul C. Shah, Waylon Brunette, Gaetano Borriello, and Sumit Roy. Exploiting mobility for energy efficient data collection in wireless sensor networks. *Mob. Netw. Appl.*, 11(3):327–339, June 2006. URL: <http://dx.doi.org/10.1007/s11036-006-5186-9>, doi:10.1007/s11036-006-5186-9.
- [11] Vinton Cerf, Scott Burleigh, Adrian Hooke, Leigh Torgerson, Robert Durst, Keith Scott, Kevin Fall, and Howard Weiss. Delay-tolerant networking architecture, rfc 4838 (informational). April 2007.

- [12] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. Uniform resource identifier (uri): Generic syntax, rfc 3986 (internet standard). January 2005.
- [13] Forrest Warthman. “*Delay-Tolerant Networks (DTNs): A Tutorial*”, 2003.
- [14] Alex Gladd, Daniel Brown, Daniel Ellard, and Richard Altmann. *DTN IP Neighbor Discovery (IPND)*, draft-irtf-dtnrg-ipnd-02 (work in progress), November 2012.
- [15] Susan Symington, Stephen Farrell, Howard Weiss, and Peter Lovell. Bundle security protocol specification, rfc 6257 (experimental). *Work Progress*, May 2011.
- [16] K. Scott and S. Burleigh. Bundle protocol specification. rfc 5050 (experimental). November 2007.
- [17] PT Barry. Abstract syntax notation-one (asn. 1). In *Formal Methods and Notations Applicable to Telecommunications, IEE Tutorial Colloquium on*, pages 2–1. IET, 1992.
- [18] N4C. Functional specification for dtn infrastructure software, n4c-wp2-023-dtn-infrastructure-fs,. April 2010.
- [19] Joerg Ott, Michael Demmer, and Simon Perreault. *Delay Tolerant Networking TCP Convergence Layer Protocol*, draft-irtf-dtnrg-tcp-clayer-05 (work in progress), January 2013.
- [20] Samuel Jero, Hans Kruse, and Shawn Ostermann. *Datagram Convergence Layers for the DTN Bundle and LTP Protocols*, draft-irtf-dtnrg-dgram-clayer-05 (work in progress), October 2013.
- [21] Razvan Beuran, Shinsuke Miwa, and Yoichi Shinoda. Performance evaluation of dtn implementations on a large-scale network emulation testbed. In *Proceedings of the Seventh ACM International Workshop on Challenged Networks*, CHANTS '12, pages 39–42, New York, NY, USA, 2012. ACM. URL: <http://doi.acm.org/10.1145/2348616.2348624>, doi:10.1145/2348616.2348624.
- [22] Wolf-Bastian Pöttner, Johannes Morgenroth, Sebastian Schildt, and Lars Wolf. Performance comparison of dtn bundle protocol implementations. In *Proceedings of the 6th ACM workshop on Challenged networks*, pages 61–64. ACM, 2011.