Project no: 269317

nSHIELD

new embedded Systems arcHItecturE for multi-Layer Dependable solutions
Instrument type: Collaborative Project, JTI-CP-ARTEMIS
Priority name: Embedded Systems

D4.2: Preliminary SPD Network Technologies Prototype

Due date of deliverable: M18 –2013.02.28
Actual submission date: M18 – 2013.02.28

Start date of project: 01/09/2011 Duration: 36 months

Organisation name of lead contractor for this deliverable:
University of Genova, UNIGE

Revision [Version 2.0]

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2012)		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	X
CO	Confidential, only for members of the consortium (including the Commission Services)	

Document Authors and Approvals

Authors		Date	Signature
Name	**Company**		
Kresimir Dabcevic	UNIGE	04/02/13	
Lucio Marcenaro	UNIGE	04/02/13	
Virgilio Esposto	SES	04/02/13	
Ester Artieda	INDRA	12/02/13	
Iñaki Arenaza	MGEP	20/02/13	
Roberto Uritbeetxeberria	MGEP	20/02/13	
Iñaki Eguia Elejabarrieta	TECNALIA	20/02/13	
Arkaitz Gamino	TECNALIA	20/02/13	
Luca Geretti	UNIUD	22/02/13	
Antonio Abramo	UNIUD	22/02/13	
Kiriakos Georgouleas	HAI	23/02/13	
Dimitris Geneiatakis	TUC	23/02/13	
Georgios Hatzivasilis	TUC	23/02/13	
Alexandros Papanikolaou	TUC	23/02/13	
Konstantinos Rantos	TUC	23/02/13	
Harry Manifavas	TUC	23/02/13	
Andreas Papalambrou	ATHENA	27/02/13	
Dimitrios Serpanos	ATHENA	27/02/13	
Reviewed by			
Name	**Company**		
Approved by			
Name	**Company**		

Applicable Documents		
ID	**Document**	**Description**
[01]	TA	nSHIELD Technical Annex

Modification History		
Issue	**Date**	**Description**
V1.0	04.02.2013	First version of the document, by UNIGE
V1.1	12.02.2013	Add link layer security, by INDRA
V1.2	20.02.2013	Add Intrusino Detection for WSN, by MGEP
V1.3	22.02.2013	Added contribution by UNIUD
V1.4	23.02.2013	T-GPSR implementation, by HAI
V1.5	23.02.2013	Formatting, by UNIGE
V1.6	27.02.2013	ATHENA contribution added
V2.0	27.02.2013	Final proofreading and formatting, by UNIGE

Executive Summary

This deliverable is focused on the detailed description of the network technologies that are currently under development in work package 4, conforming to the preliminary architecture and the composability requirements specified in deliverables D2.4 and D2.5. These technologies will be made available to the application scenarios and can be used as building blocks for the project demonstrators. This deliverable will be updated and refined in the second part of the project based on the final requests received from the application scenarios and on the refined system architecture, metrics and composition strategy to be followed.

Contents

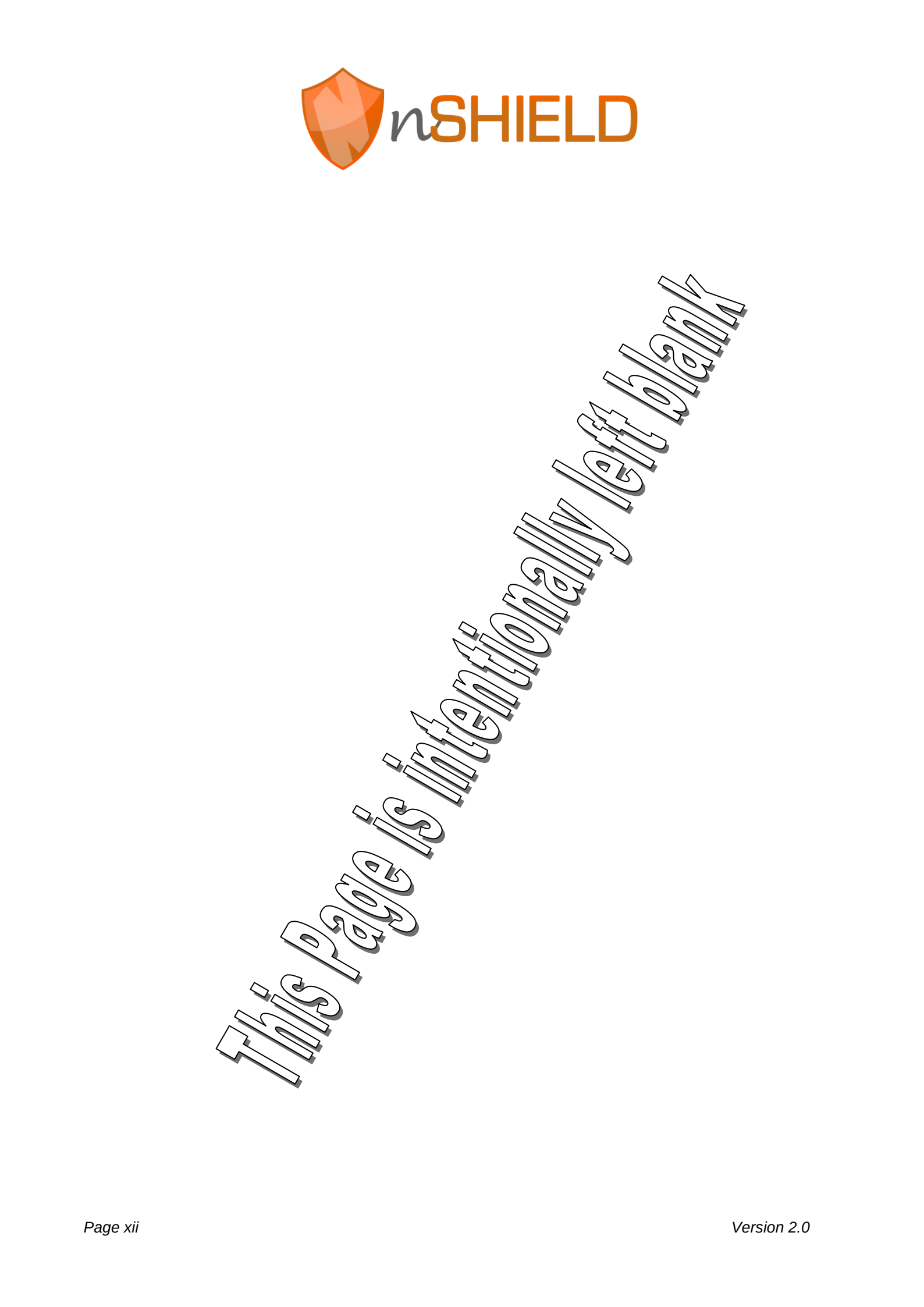
Figures

Tables

Programing Listing

Glossary

Please refer to the Glossary document, which is common for all the deliverables in nSHIELD.


This Page is intentionally left blank

1 Introduction

The nSHIELD project proposes a layered architecture to provide intrinsic SPD features and functionalities to embedded systems. In this layered architecture, building on top of the node functionalities defined in the WP3, Work Package 4 deals with implementation of the SPD functionalities at the network layer.

The workload encompassed in work package four is divided into four complementing work tasks:

- T4.1 Smart Transmission Layer;
- T4.2 Distributed self-x models;
- T4.3 Reputation-based resource management technologies;
- T4.4 Trusted and dependable Connectivity

Each of the tasks places focus on independent development and application of different SPD technologies at the network layer. As such, the technologies' performance will at first be evaluated on an individual basis, categorized with respect to their complexity and suitability for the proposed SPD levels and capabilities of different node classes. This will provide an output useful for merging the contributions into a system consisting of a set of mutually-collaborating approaches.

Deliverable D4.2 provides a technical perspective on the developed Network prototypes, focusing on the development platforms and technologies, whereas the complimentary deliverable D4.3 presents an overview of the prototypes' operational characteristics, as well as the results that have reached demonstrable level.

D4.2 is structured as follows:

1. Introduction – overview of the document
2. SPD-driven Smart transmission layer – describes details of the hardware implementation of the Smart Transmission Layer prototype, with the corresponding developed and tested functionalities (section 2.1), as well as pieces of software describing the proprietary C++ network simulator used for development of the anti-jamming mechanisms (section 2.2).
3. Distributed self-x models – provide means for reducing vulnerabilities present in unmanaged and hybrid managed/unmanaged networks. Two prototypes are presented: Recognizing & modelling of denial-of-service attacks and Model-based framework for dependable distributed computation.
4. Reputation-based resource management technologies – schemes for reputation-based cooperation enforcement and scalable resource management based on distributed mechanisms aiming at identifying malicious users and performing a secure routing through secure paths. Technical details of two prototypes are given: Reputation based secure routing and nSHIELD Reputation scheme.
5. Trusted and dependable Connectivity - algorithms for provisioning security on link and network layers are presented, as well as the access control methods in Smart Grid networks.

2 SPD-driven Smart Transmission Layer

SPD-driven Smart Transmission Layer is a set of services deployed at the network level designed for nSHIELD SDR-capable Power nodes, whose goal is ensuring smart and secure data transmission in critical channel conditions. For achieving this, concepts of Software Defined Radios and Cognitive Radios are being utilized.

Along with the section 2.1 of the deliverable D4.3, the preliminary prototype is described in the following section.

2.1 Smart Transmission Layer test-bed prototype

2.1.1 Basic description

The proposed Smart Transmission Layer SDR/CR test bed prototype consists of a number (currently: 2, but to-be-increased to 3) of Secure Wideband Multi-role - Single-Channel Handheld Radios (SWAVE HHs), each interconnected with the OMBRA v2 multi-processor embedded platform (nSHIELD Power node).

2.1.2 Prototype setup

SWAVE HH (from now on referred to as HH) is a fully operational SDR radio terminal capable of hosting a multitude of wideband and narrowband waveforms.

Maximum transmit power of HH is 5W, with the harmonics suppression at the transmit side over -50 dBc. Superheterodyne receiver has specified image rejection better than -58 dBc. The receiver is fully digital; in VHF, 12-bit 250 MHz analog to digital (AD) converters perform the conversion directly at RF, while in UHF, AD conversion is performed at intermediate frequency (IF). No selective filtering is applied before ADC. Broadband digitized signal is then issued to the FPGA, where it undergoes digital down conversion, matched filtering and demodulation.

HH has an integrated commercial Global Positioning System (GPS) receiver, but also provides the interface for the external GPS receiver. GPS data is available in National Marine Electronics Association (NMEA) format and may be outputted to the Ethernet port.

Radio is powered by Li-ion rechargeable batteries, however may also be externally powered through a 12.6V direct current (DC) source. Relatively small physical dimensions (80x220x50 mm), long battery life (8 hours at the maximum transmission power for a standard 8:1:1 duty cycle), and acceptable weight (960g with battery) allow for portability and untethered mobile operation of the device.

Hypertach expansion at the bottom of HH provides several interfaces, namely: 10/100 Ethernet; USB 2.0; RS-485 serial, DC power interface (max 12.7V), and PTT.

The radio provides operability in both Very High Frequency - VHF (30 - 88 MHz), and Ultra High Frequency - UHF (225 - 512 MHz) band. The software architecture of the radio is compliant with the Software Communications Architecture (SCA) 2.2.2 standard. Following that, HH provides support for both legacy and new waveform types. Currently, two functional waveforms are installed on the radio: SelfNET Soldier Broadband Waveform (SBW) and VHF/UHF Line Of Sight (VULOS), as well as the waveform providing support for the Internet Protocol (IP) communication in accordance with MIL-STD-188-220C specification. Currently installed waveforms are described and analysed in more details in section 2.1.4.

The considered power node – OMBRA v2 platform – is composed of a small form factor System-on-Module (SOM) with high computational power - developed by Selex ES - and the corresponding carrier board. It is based on an ARM Cortex A8 processor running at 1GHz, encompassed with powerful programmable Xilinx Spartan 6 FPGA and Texas Instruments TMS320C64+ DSP. It can be embodied with up to 1 GB LPDDR RAM, has support for microSD card up to 32 GB, and provides interfaces for

different RF front ends. Support for IEEE 802.11 b/g/n and ANT protocol standards are proffered. Furthermore, several other external interfaces are provided, i.e. 16 bit VGA interface; Mic-in, line-in and line-out audio interfaces; USB 2.0; Ethernet; and RS-232 serial. The node is DC-powered, and has Windows CE and Linux distribution running on it. System architecture of the Power node is shown in Figure 2-1: STL - OMBRA v2 nSHIELD Power node - system architecture

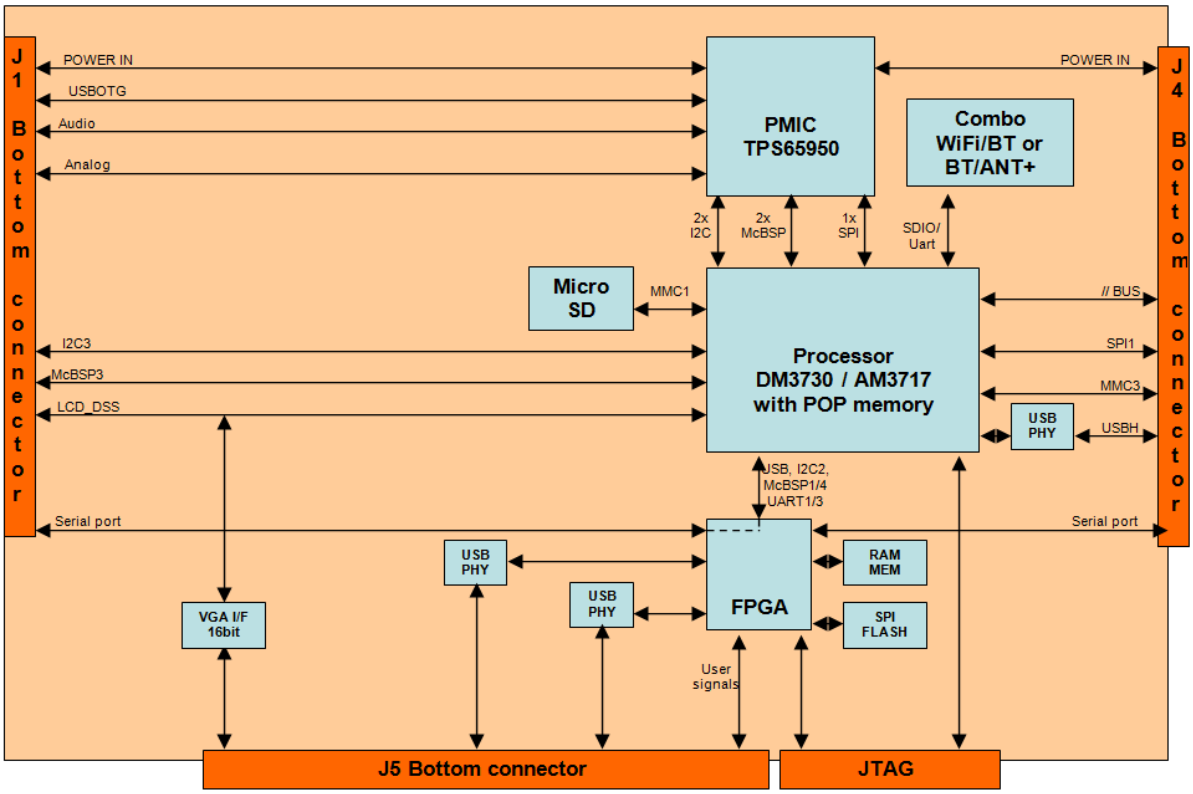


Figure 2-1: STL - OMBRA v2 nSHIELD Power node - system architecture

Connection to HH is achieved through Ethernet, as well as serial port. Ethernet is used for the remote control of the HH, using SNMP. For the serial connection, due to different serial interfaces - RS-232 and. RS-485, a RS-232-to-RS-485 converter is needed. Serial connection is used for transferring the spectrum snapshots from HH to Power node. More details on remote control and spectrum sensing are given in sections 2.1.3 and 2.1.6.

Figure 2-2 shows the implementations of HH and Power node which, once interconnected, are referred to as SDR-capable Power node.

Figure 2-2: STL - Implementations of SWAWE HH and the nSHIELD Power node

The current test bed prototype is composed of two SDR-capable Power nodes. A coaxial RF bench was implemented for the frequency range of interest. Because of the high output power of the radios, two programmable attenuators had to be included in the coaxial path, and were programmed to their maximum attenuation value - 30dB. Agilent 778D 100 MHz - 2GHz dual directional coupler with 20dB nominal coupling was placed between the attenuators, allowing for sampling and monitoring the signal of interest. Agilent E4438C vector signal generator was connected to incident port of the coupler, with the purpose of injecting noise/interference signal to the network. Agilent E4440A spectrum analyser was connected to the coupler's reflected port, facilitating the possibility of monitoring the RF activity.

Implementation of the test bed is shown in Figure 2-3.

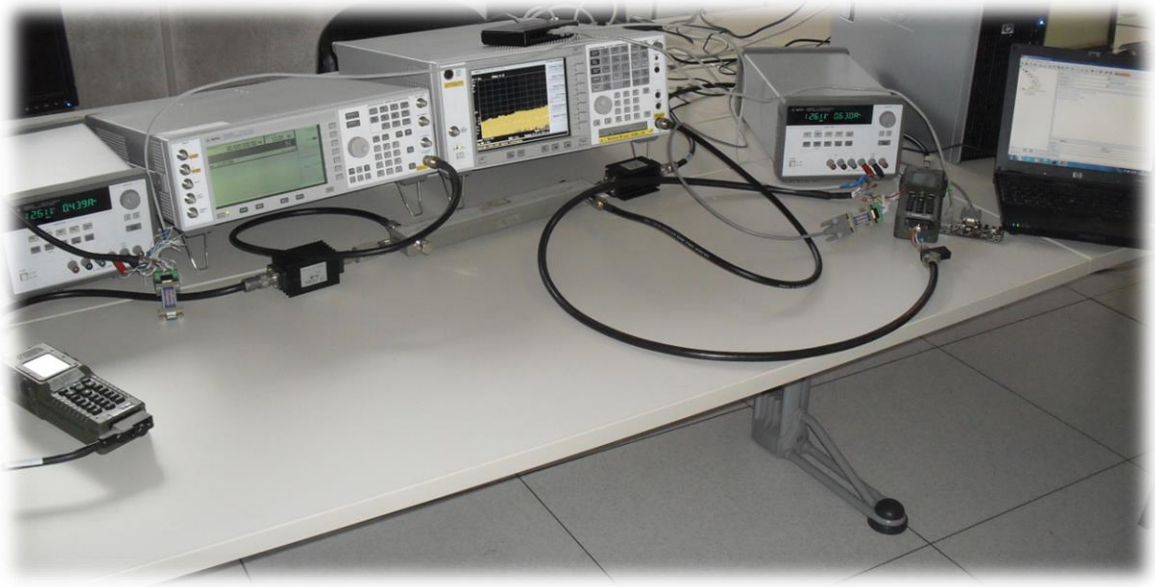


Figure 2-3: STL - Smart Transmission Layer test bed implementation

2.1.3 Remote control of the radio

Using Simple Network Management Protocol v3 (SNMP v3), several parameters of the HH radio may be externally controlled. For achieving this, SNMP manager has to be installed and running on the Power node. The host (Power node) and the agents (HHs in the network) are connected through an Ethernet hub, and need to be on the same domain.

By utilizing three basic SNMP commands: GET, SET and TRAP, it is possible to: read the current value of the parameter, set a new value, or issue a message/warning if the current value satisfies a condition, respectively.

The controllable parameters and their corresponding features are stored in a Management Information Base (MIB), which is loaded into the host's SNMP manager. MIB table contains all the definitions that define properties of the controllable parameters, and describes each object identifier (OID), which is a sequence of integers, with a more easily understandable (from a human operator's perspective) string.

The list of the parameters that may be controlled externally, with the corresponding input data types and the SNMP commands that may be invoked is given in Table 2-1. ManageEngine MibBrowser Free Tool was used as the SNMP manager running on the Power node.

Table 2-1: STL - HH's Parameters that may be remotely controlled via SNMP

Parameter	Type	SNMP commands
File Transfer Activation	string	SET/GET
File Transfer Type	string	SET/GET
FTP User Name	string	SET/GET
FTP Password	string	SET/GET
FTP Address	string	SET/GET
Login Username	string	SET/GET
Login Password	string	SET/GET
Transmit Power	integer	SET/GET
Transmitter On/Off	integer	SET/GET
Currently Installed Waveform	string seq	GET
Waveform’s MIB Root	string	GET
Waveform Status ION/OFFI	integer	SET/GET
Audio Message I n	string	SET/GET
Create New Waveform	string	SET/GET
Activate Preset	string	SET/GET
Activate Mission File	string	SET/GET
Audio Output Gain	float	SET/GET
Battery Charge Percentage	integer	GET
File Download Status	integer	GET
Trap Receiver's IF Address	string	SET/GET
Zeroize All Crypto Keys	integer	SET/GET
Crypto Key Loaded	integer	GET
System End Boot [failed / succeeded / in progress]	integer	GET

Accordingly, Table 2-2 provides list of the parameters that may be TRAPped, with the short description of the conditions under which TRAPping messages are issued.

Table 2-2: STL - HH's Parameters that may be TRAPped via SNMP

Parameter	Description
NET Radio OK	The notification is triggered when the visibility of the radio network is acquired
NET Radio FAIL	The notification is triggered when the visibility of the radio network is lost
Critical Alarm	The notification is triggered when the 1111 has sustained a critical operational error
End Boot	The notification is triggered when successful boot-up of the HU has been verified
End File Download	The trap notifies end of the procedure of file download. indicating whether it was successful
Low Power	The notification is triggered when the battery charge falls below a pre-declined limit
Create Waveform OK	The notification is triggered when the waveform is successfully created
Create Waveform FAIL	The notification is triggered when the waveform creation has failed

The process of turning on the HHs with respective IP addresses 10.31.44.210 and 10.31.44.201, logging in, changing the waveform type (SBW is automatically loaded on radios upon booting) to VULOS, and then changing it back to SBW, results in triggering the sequence of TRAP commands denoted in Figure 2-4.

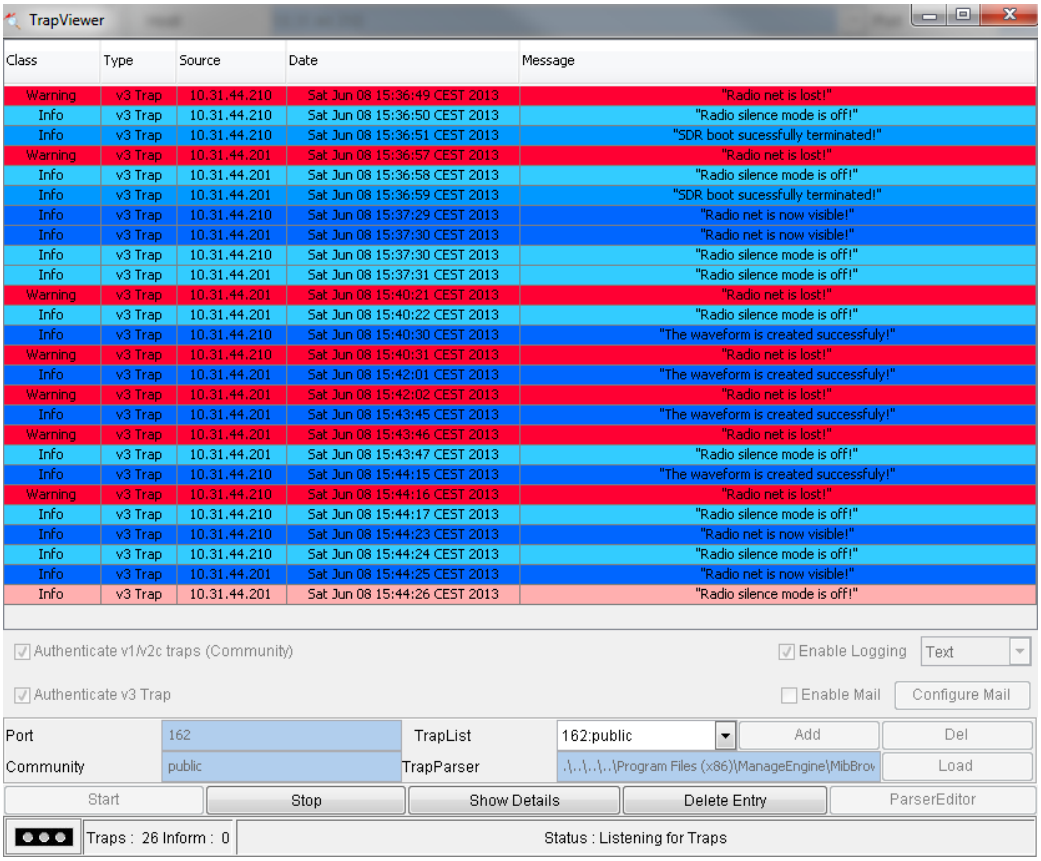


Figure 2-4: STL - Triggered TRAP messages for a turn on - log on - change waveform sequence on HHs

2.1.4 Waveform analysis

As previously stated, there are currently two functional waveforms installed on SWAVE HHs: SBW and VULOS. Having a wideband spectrum analyser allows for monitoring the waveforms and analysing their parameters.

SBW is a wideband multi-hop Mobile Ad-hoc NETwork (MANET) waveform, supporting operation in the 225 - 512 MHz part of the UHF band. The waveform provides self-(re)configurability and self-awareness of the network structure and topology, for up to 50 nodes and up to 5 hops. Furthermore, possibility of simultaneous streaming of voice and data services is provided, with prioritization for voice streaming (in case of exceeded bandwidth). Allocated channel bandwidth is adjustable - up to 5 MHz - with channel spacing of up to 2 MHz SBW uses a fixed digital modulation technique.

Self-awareness is exercised by monitoring the network topology for changes every *n* seconds (monitor interval is adjustable). Two Quality of Service (QoS) monitoring mechanisms are provided: Bit Error Rate (BER) Test, and the statistics data for the transmitting/receiving side. These mechanisms provide means for analysing and comparing the quality of communication in regular and impaired channel conditions. More in-depth analysis of these features is presented in section 2.1.5.

Figure 2-5 shows envelope shape and properties of the SBW waveform, for the maximum signal bandwidth (5 MHz) and 1/10th of the maximum transmit power (-3 dBW), in frequency domain.

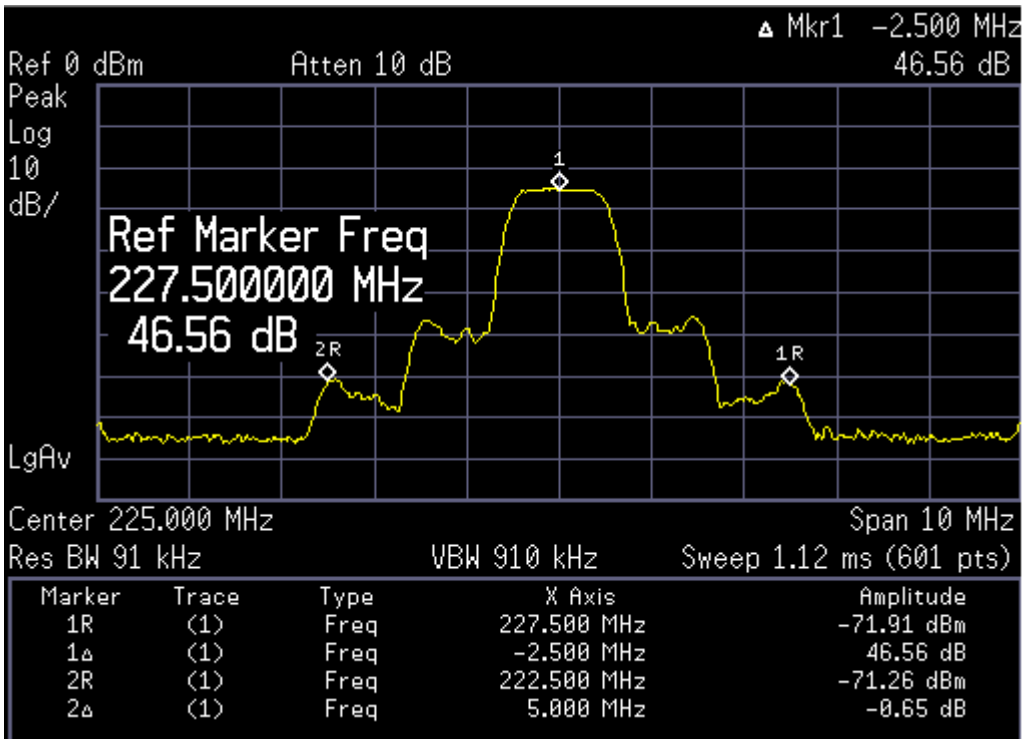


Figure 2-5: STL - SBW waveform in the frequency domain - max hold

VULOS is a narrowband single-hop waveform designed for short-distance voice or data communication. It supports operation in both VHF (30-88 MHz) and UHF (225-512 MHz) frequency bands. The waveform allows for choosing between two analog modulation techniques: Amplitude Modulation (AM) and Frequency Modulation (FM), which may be configured on-the-fly, alongside with the modulation index. Channel bandwidth is adjustable up to 25 kHz, with channel spacing also adjustable up to 25 kHz. Furthermore, the VULOS waveform is able to utilize both digital and analog voice Coder-Decoders (CODECs) installed on the radio.

Figure 2-6 shows envelope shape and properties of FM-modulated VULOS waveform with the 25 kHz bandwidth, transmitted at 1 dBW in VHF band (30 MHz).

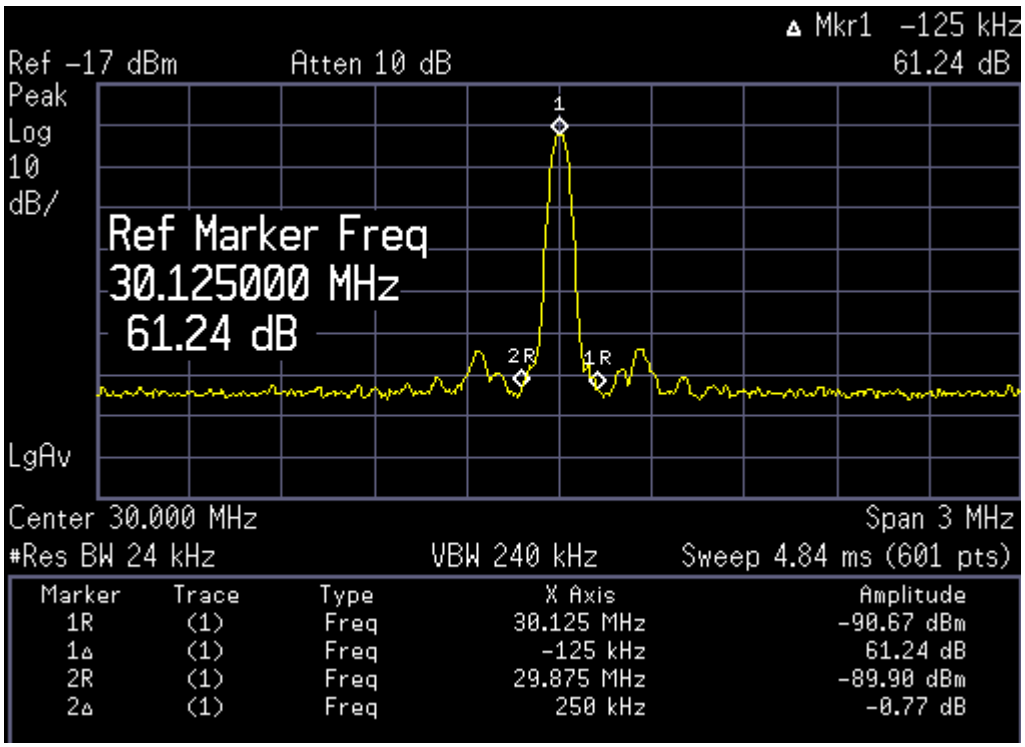


Figure 2-6: STL - VULOS waveform in the frequency domain - max hold

Waveform analysis will have an important SPD application - by creating a database of waveform types that are occurring in the system, it will be possible to identify potentially malicious or misbehaving users.

2.1.5 Interference detection

Various Denial of Service (DoS) attacks - and in particular jamming attacks - have for a long time been posing - and continue to pose - significant security threats to radio networks. Radiofrequency (RF) jamming attacks refer to the illicit transmissions of RF signals with the intention of disrupting the normal communication on the targeted channels. RF jamming is a known problem in modern wireless networks, and not an easy one to counter using traditional “hardware-based” equipment. Additionally, Software Defined Radios and Cognitive Radios bring the prospect for further improvement of the jamming capabilities of the malicious users. They also offer the possibility of developing advanced protection- and counter-mechanisms.

One of the main focuses of the SPD-driven Smart Transmission Layer is precisely providing safe and reliable communication in jamming-polluted environments. Momentarily, advanced jamming and anti-jamming algorithms are studied separately, and simulated using the proprietary simulator presented in section 2.2. As the prototype matures, these strategies will also be demonstrated on the real-life prototype described in this section.

The vector signal generator is presently used as means for creating disturbances in the communication channel, emulating a simple RF jammer. A set of measurements demonstrating how different types of created interfering signals influence the performance of the communication on the channel was done.

In the first set of measurements, aim is at showing the correlation between Bit Error Rate (BER) and the radio's built-in Link Quality metric. Link quality is HH's built-in QoS feature, and is represented by an integer in the range of [0-200]. The measurements are done with HHs having their signal bandwidths set to the maximum value (5 MHz), and repeated for two transmitting powers: -12dBW and 4 dBW. Created

interfering signal is a pulse signal, created at the same frequency as the frequency of the channel used for communication between radios (225 MHz). Amplitude of the created interfering signal varies. The results are presented in Figure 2-7.

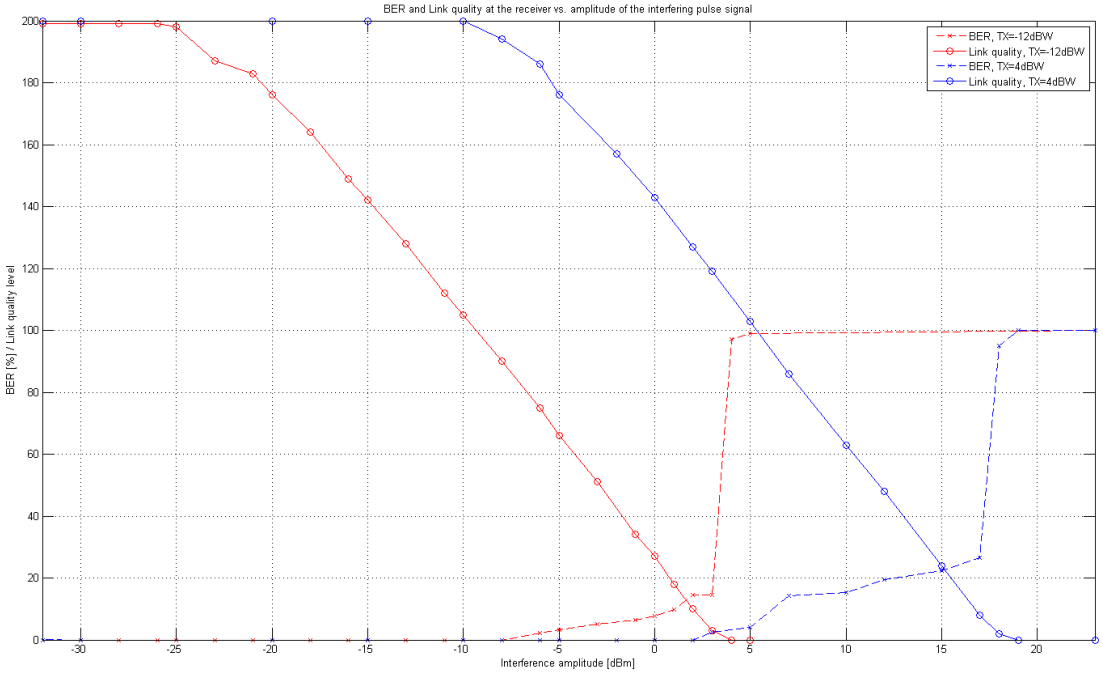


Figure 2-7: STL - BER and Link quality level vs. interference amplitude of interfering pulse signal

BER percentage is shown in the first half of the Y-axis (0-100), whereas Link quality level stretches throughout the whole Y-axis (0-200). The BER curves are mutually similarly shaped, with the expected offset due to differing transmission powers of the radio. The same goes for the link quality curve shapes. As can be seen, occurrence of errors at the receiving side (area where BER > 0) corresponds to Link quality levels in the range of [90-120]. As expected, 100% BER corresponds to the link quality of 0, meaning the communication has become impossible.

In the second set of measurements, different types of interfering signals are created by the signal generator, namely: pulse signal as in the first measurement set; Real Time I/Q Baseband Additive White Gaussian Noise (AWGN) with the effective bandwidth of 5 MHz; Real Time I/Q Baseband AWGN with the effective bandwidth of 1 MHz, and a GSM signal. Once again, central frequency of all of the interfering sources is the same as the frequency of the channel that the radios use for communication (225 MHz). The results are shown in Figure 2-8.

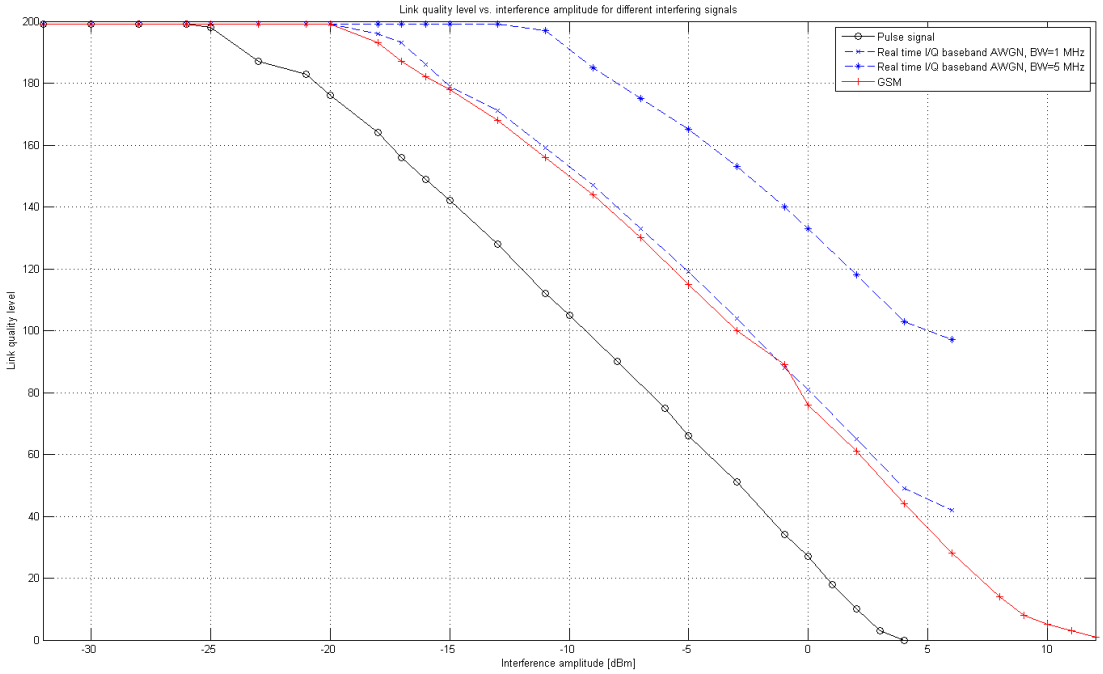


Figure 2-8: Link quality level vs. interference amplitude for different interfering signals

As expected, pulse signal has the best interfering capabilities, due to the fact that it has the most concentrated power, and importantly that it has been created at the exact frequency as the main carrier frequency of the transmitted signal. Even with small frequency offsets, interfering impact of the pulse signal would drop significantly. For the same reason, addition of AWGN results in higher link degradation in cases of smaller allocated bandwidth, due to the higher power density. The vector signal generator is only able to produce an AWGN signal of amplitude up to 20 dBm, hence the measurements for the higher values were not done.

It should be noted that the results presented in this subsection are of a reference, instead of an absolute value at this stage, the intention was not placed upon emulating real-life interferers, but rather at performing the initial study of the interference detection functionalities of the SWAVE HHs.

2.1.6 Energy detection spectrum sensing

Obtaining information of the current spectrum occupancy is paramount for the Cognitive Radios to be able to opportunistically access spectrum, but may also aid them in recognizing anomalous or malicious activity by comparing the current state to those stored in their databases. There are three established methods for CRs to acquire knowledge of the spectrum occupancy: spectrum sensing, geolocation/database, and beacon transmission. HH has a capability of performing energy detection spectrum sensing.

Every 20 seconds, 8192 samples from the ADC are transmitted over the RS-485 port this is functionality hard-coded in the HH's FPGA. Each sample is transmitted in two bytes: first byte containing the 6 most significant bits (MSBs), with 2 bits sign extension on the left. Second byte contains the 8 LSBs. In total, 16384 characters are transmitted, making up for the interpretation of a 16-bit word. Currently, there is not a synchronization pattern however the idle interval between the two transmissions may be used to e.g. perform analysis of the received data. Transmission of a full window takes approximately 2 minutes.

The signal at the HH's FPGA input is a sample of raw spectrum. Raw samples are stored in a RAM buffer internal to the FPGA, and output through HH's fast serial port to the Power node, where they can be processed.

Due to the high speed of the ADC (250 MHz), serial port speed (114200 bit/s is supported in the asynchronous mode) is not sufficient for the true real-time transfer; in addition processing capabilities of the Power node would be completely devoted to the processing of received signal, leaving no room for higher level applications. Power consumption would be heavily affected, too.

Adopted solution is to perform a quasi-real-time acquisition, i.e. to collect a large “snapshot” of incoming spectrum, i.e. tens of kilo-samples, and to transfer the snapshot to the Power node. When the snapshot has been transferred, a new collection may start. This is sufficient for proper analysis of the majority of RF scenarios: in practice, only fast pulsed signals might be completely missed.

Future hardware enhancements are expected to make real-time spectrum acquisition possible.

2.2 Algorithm for countering Smart Jamming Attacks in centralized networks

2.2.1 Description

Improved radio capabilities of Cognitive Radios also bring advanced possibilities with respect to attackers’ actions and complexity levels, starting with the possibility of jamming multiple frequencies and larger frequency bands by self-reconfiguring their transmission parameters ”on-the-fly”. Such advanced jammers, operating in SDR Networks and CR Networks are from now on referred to as “Smart” or “Intelligent” jammers [1].

The jamming strategies and the proposed counter-measures have been described in Section 2.2 of D4.3. Here, the most important corresponding parts of the simulator (C++ code) are provided:

The basic jamming algorithm for two jammer types: naïve (motion model “MB_DEFAULT”) and tracking (motion model “MB_FOLLOW) is given as follows:

```
double jammer::Jam(double idealRSS, isip::ipoint nodePos, double nodeFrequency)
{
	double jammedRSS = 0.0;
	bool isJammed = IsJammed(idealRSS, nodePos, nodeFrequency, jammedRSS);
	isip::ipoint position_JAM = GetCurrentPosition();
	double d_RN_JAMMER = position_JAM.DistanceTo(nodePos);
	if (d_RN_JAMMER<GetSensingRadius() && (m_pMotionModel->GetMotionBehaviour() ==
motionModel::MB_FOLLOW))
	{
		isip::ipoint newSpeed = nodePos - position_JAM;
		double newSpeedX = (double)newSpeed.x/sqrt(double(newSpeed.AbsSqr()));
		double newSpeedY = (double)newSpeed.y/sqrt(double(newSpeed.AbsSqr()));
		m_pMotionModel->SetSpeed(isip::ipoint((int)(newSpeedX+0.5*(newSpeedX>0?1:-1)),
			(int)(newSpeedY+0.5*(newSpeedY>0?1:-1))));
	}
	else if (m_pMotionModel->GetMotionBehaviour() == motionModel::MB_FOLLOW)
	{
		m_pMotionModel->ResetSpeed();
		m_pMotionModel->SetMotionBehaviour(motionModel::MB_DEFAULT);
	}
	return jammedRSS;
}
```

Program listing 1: SJA – basic jamming algorithm for naïve and tracking jammers

Jamming occurs whenever the node gets within the pre-defined jamming radius of the jamming entity:

```
bool jammer::IsJammed(double idealRSS, isip::ipoint nodePos, double nodeFrequency,
double &jammedRSS)
{
	bool isJammed = false;
	jammedRSS = idealRSS;
	isip::ipoint position_JAM = GetCurrentPosition();
	double d_RN_JAMMER = position_JAM.DistanceTo(nodePos);
	if (d_RN_JAMMER<GetSensingRadius())
	{
		if (GetJamFrequency() == nodeFrequency)
		{
			jammedRSS = 0;
			isJammed = true;
		}
	}
	return isJammed;
}
```

Program listing 2: SJA - Jamming implementation

Higher-order adaptive frequency jammer is able to perform spectrum sensing in order to selectively jam frequencies of interest. Its jamming algorithm is denoted as follows:

```
bool adaptiveFreqJammer::IsJammed(double idealRSS, isip::ipoint nodePos,
double nodeFrequency, double &jammedRSS)
{
	bool isJammed = false;
	jammedRSS = idealRSS;
	isip::ipoint position_JAM = GetCurrentPosition();
	double d_RN_JAMMER = position_JAM.DistanceTo(nodePos);
	if (d_RN_JAMMER<GetSensingRadius())
	{
		if (GetJamFrequency() != nodeFrequency)
		{
			SetJamFrequency(nodeFrequency);
		}
		if (GetJamFrequency() == nodeFrequency)
		{
			jammedRSS = 0;
			isJammed = true;
		}
	}
	return isJammed;
}
```

Program listing 3: SJA - adaptive frequency jammer

Finally, reputation-attacking jammer as the most sophisticated considered jamming entity, inherits the characteristics of “tracking” and “adaptive frequency” jammer, and is also able to deceive the reputation algorithm with a certain probability:

```
bool reputationAttackingJammer::IsJammed(double idealRSS, isip::ipoint nodePos,
double nodeFrequency, double &jammedRSS, double Deceivesuccess)
{
	bool isJammed = false;
	jammedRSS = idealRSS;
	isip::ipoint position_JAM = GetCurrentPosition();
	double d_RN_JAMMER = position_JAM.DistanceTo(nodePos);
	if (d_RN_JAMMER<GetSensingRadius())
	{
		if (GetJamFrequency() != nodeFrequency)
		{
			SetJamFrequency(nodeFrequency);
		}
		if (GetJamFrequency() == nodeFrequency)
		{
			jammedRSS = 0;
			isJammed = true;
		}
		if ((rand()%Deceivesuccess)==0)
			SetTXPower(0);
	}
	return isJammed;
}
```

Program listing 4: SJA - Reputation attacking jammer

The three considered (collaborating) anti-jamming algorithms – frequency switching [2]; reputation mechanism and trajectory altering – are presented as follows:

```
double node::ChooseRandAmongst(std::vector<double> AvailableNumbers, double CurrentNumber)
{
	std::vector<double> temp;
	for (size_t i=0; i<AvailableNumbers.size(); i++)
	{
		if (AvailableNumbers[i] != CurrentNumber)
		temp.push_back(AvailableNumbers[i]);
	}
	return theRNG.Generate(temp);
}

void node::ChangeTXFrequency()
{
	SetTXFrequency(ChooseRandAmongst(m_vAvailTxFrequencies, m_dTxFrequency));
}
```

Program listing 5: SJA - frequency switching algorithm

```
void radioSimulator::ReputationUpdate(std::map<std::string, bool> mNodesLost)
{
    for ( std::map<std::string, bool>::iterator it = mNodesLost.begin();
         it != mNodesLost.end(); ++it )
    {
       if (!it->second)
              continue;
              const isip::radio::node* pNode = GetNode(it->first);
              std::vector<isip::radio::node*> vNeighbour = GetNeighbouringNodes(
                                    pNode->GetUUID(), pNode->GetSensingRadius());
              for (unsigned int i=0; i<vNeighbour.size(); i++)
              {
                     vNeighbour[i]->SetReputation(vNeighbour[i]->GetReputation()-1);
              }
    }
}
```

Program listing 6: SJA - reputation mechanism

```
void radioSimulator::AvoidJammer(std::vector<isip::radio::node*> listofidentifiedjammers)
{
   for ( std::vector<isip::radio::node*>::iterator Iterator = m_vRadioNodes.begin();
              Iterator != m_vRadioNodes.end(); ++Iterator)
       for (int i=0; i<listofidentifiedjammers.size(); i++)
       {
              if ((*Iterator)->GetUUID()==listofidentifiedjammers[0]->GetUUID())
                     continue;
              isip::ipoint jammercoordinates = listofidentifiedjammers[0]
                                          ->GetCurrentPosition();
              isip::ipoint nodescoordinates = (*Iterator)->GetCurrentPosition();
              std::vector<isip::ipoint> vJamPos;
              vJamPos.push_back(jammercoordinates);
              (*Iterator)->SetJammersPosition(vJamPos);
       }
   return ;
}
```

Program listing 7: SJA - trajectory altering mechanism

The AvoidJammer algorithm is triggered when the cognitive entity decides, based on the reputation level, that a certain node exhibits malicious behaviour. Then, it stores its ID into a list, used for keeping track of its future behaviour and monitoring its position:

```
std::vector<std::string> radioSimulator::ReturnListOfIdentifiedJammers()
{
       std::vector<std::string> retvector;
       for (unsigned int i=0; i<m_vListofidentifiedjammers.size(); i++)
       {
              retvector.push_back(m_vListofidentifiedjammers[0]->GetUUID());
       }
       return retvector;
}
```

Program listing 8: SJA - list of identified jammers

Performances of the included algorithms are presented in deliverable D4.3.

3 Distributed self-x models

Distributed self-x models provide means for reducing vulnerabilities present in unmanaged and hybrid managed/unmanaged networks.

3.1 Recognizing & modelling of denial-of-service attacks

3.1.1 Basic description of the DoS scheme operation

Denial of Service (DoS) attacks aim to deplete resources of the target, either in physical or computational resources (node) or capacity, bandwidth and normal operation (network). An overview of DoS attacks and the theoretical approach that was followed to detect them has been given in the related section of deliverable D4.3. In the following section, more technical details will be given related to the DoS detection mechanism that has been designed as well as its ongoing implementation.

3.1.2 Architecture

The DoS attack detection mechanism involves cooperation between components belonging to all three layers on the nSHIELD architecture. However, the principal algorithmic operation is considered to be a network process and is therefore described in the network related documents.

The scheme can be seen as an algorithmic operation which is fed with inputs from various components and provides a set of results relating to the identification of a DoS attack that is underway.

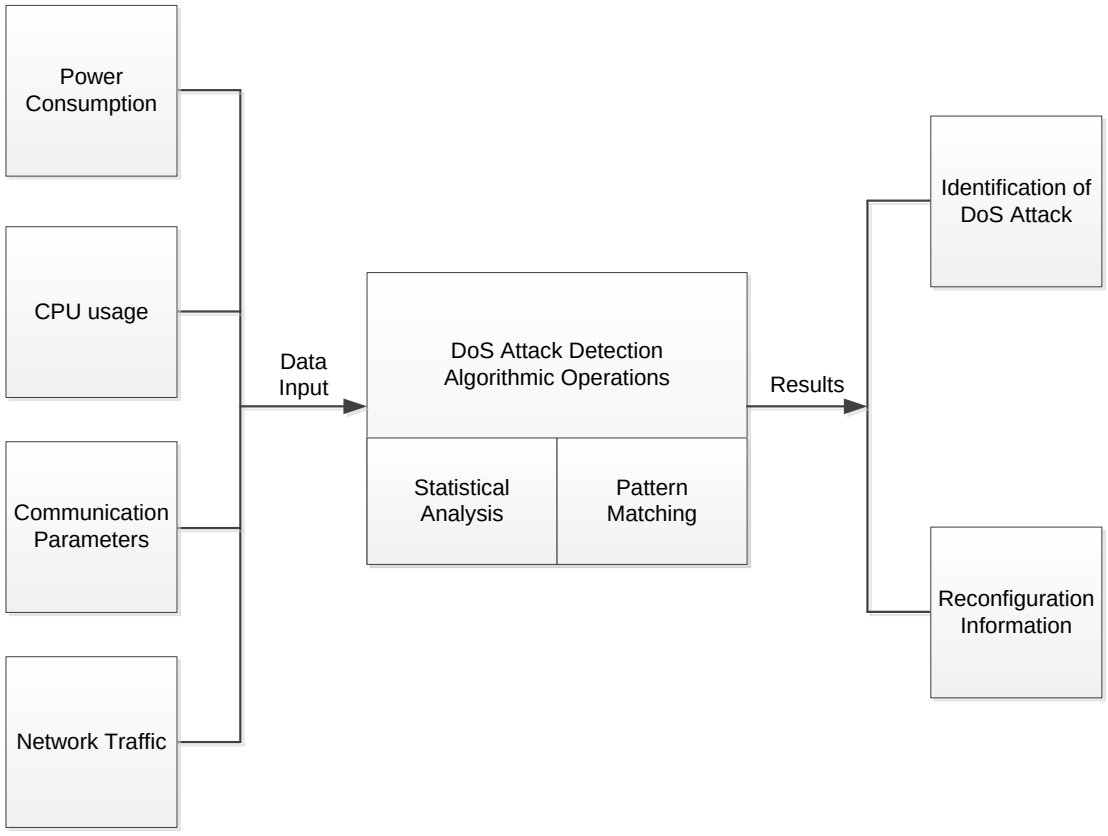


Figure 3-1: DoS Attack Detection Algorithm

As is evident from the figure, the algorithmic operations are fed data input which comes from different components, described below. The data input is analysed and used in order to provide the result of the algorithm which leads to attack identification and issue of reconfiguration commands.

3.1.2.1 Components

The components of the system correspond to software processes in the simulation architecture. In order to simulate the algorithms it was important to identify the basic system components which would be implemented as discrete functions. A proper organization of components leads to efficient and simple operation. The components identified in the system are the following, organized in two types:

3.1.2.2 Input modules

a. Power unit monitoring module. This is a process which runs in the power unit module and provides information related to power consumption.

b. CPU monitoring module. This is a process which provides CPU usage information and can typically be considered a service of the operating system.

c. Communication monitoring module. This is a process which provides information from the physical transmission such as signal strength.

d. Network traffic module. This module has access to the data packets exchanged and can sample either complete packets or specific parts in the content or header.

3.1.2.3 Algorithmic modules

There are two algorithmic software processes in the system, the Statistical analysis algorithm and the Pattern matching algorithm.

a. Statistical analysis algorithm

   This software process communicates with all four input modules, reads and processes that information.

b. Pattern matching algorithm

   This software process communicates with the network traffic module. Its task is to sample network packets and compare them with the signature database.

3.1.3 Interface between modules

Communication between the input modules and the algorithmic modules is made with the exchange of messages. A typical message is composed of the following three fields: Node ID, Timestamp and Payload. The Node ID identifies the node that is reporting the information and the Timestamp identifies the exact timing of the sampled information. These two fields are common for all messages originating from all modules. The third field, called the payload, contains the actual information being reported and is different for each of the modules.

Figure 3-2: DoS attacks – fields composing the message

The payload is as follows for the four different types of messages.

- Power consumption payload: contains the power consumption wattage at the timestamp sampling time in mW.
- CPU usage payload: contains the current CPU usage percentage and number of running processes at the timestamp sampling time.
- Communication parameters payload: contains the incoming and outgoing bitrate, the number of established connections as well as the number of other nodes to which there are established connections at the timestamp sampling time.
- Network traffic payload: contains complete data packets (typically TCP/IP) or headers of the packets, depending on mode of operation, which can be fed to the pattern matching algorithm.

3.1.4 Algorithmic operation

The statistical analysis algorithm receives messages from input modules and correlates inputs in order to detect whether an anomaly is taking place. The algorithm can produce an output by examining any number of input message types from 1 to 4. The algorithm is based on simple but multi-parameter statistical analysis.

Typical communication patterns produce highly predictable hardware and software response. For example, a typical communication scenario of a certain number of interacting nodes and amount of exchanged traffic corresponds to specific amount of CPU usage and power consumption. If there is a deviation from this correlation, a DoS attack may be under way.

A score is calculated by making each correlation, after which the scores of all correlations are added. If the total score is above the predetermined threshold, then a DoS attack warning is issued.

The pattern matching algorithm receives messages from the network traffic module which contains either header of data packages or complete data packages. The algorithm analyses the data and tries to detect patterns in the provided traffic according to a pre-determined attack pattern database. The database is composed of known risks such as malicious software or known network protocol vulnerabilities. Whether the operation is in header of full payload analysis depends on the mode of operation which in turn depends on the available node resources.

A code example can be given as follows. The most important part in the algorithmic operation is the pre-calculation of cofactors that can correlate the various parameters. These cofactors can be calculated by measurements (or simulation) in typical networking scenarios. As seen in the example, the generateCommStats method generates a snapshot of the current network environment by relating bitrates, number of connections and number of peers. This information, processed with the help of the calculated cofactors, is fed to the statAnalysis method which checks to see if the measured power and processing loads correspond to the current network status snapshot.

```
float generateCommStats (int nodesNum, int connectionsNum,  float inBitrate, float outBitrate)
{
   bitrateScore = (inBitrate * inCofactor) + (outBitrate * outCofactor);
   nodeScore = nodesNum * nodeCofactor1;
   connectionScore = connectionsNum * connectionsCofactor2;
   commTotalTrafficScore = bitrateScore + nodeScore + connectionScore;
   commNetworkComplexityScore = (nodesNum * nodeCofactor2) + (connectionsNum *
                                connectionsCofactor2);
}


int statAnalysis (float powerConsumption, float CpuUsage  struct commStats *)
{
   powerScore = powerconsumption * powerCofactor;
   cpuScore = CpuUsage * cpuCofactor;
   commStat1 = commStats.commTotalTrafficScore;
   commStat2 = commStats.commNetworkComplexityScore;

   if ((powerScore + cpuScore) > (commStat1 * threshold1))
   {
```

```
            printf ("Node load is above expected for current traffic amount");
        }

        if ((powerScore + cpuScore) > (commStat2 * threshold2))
        {
            printf ("Node load is above expected for current number of connected peers");
        }
    }
```

Program listing 9: DoS attacks - code example

3.1.5 Operation of the DoS attack detection scheme in the simulator

The chosen simulation environment for the development of the system is OMNet++. OMNeT++ is a modular, component-based C++ simulation library and framework, for building network simulators. It was chosen because it allows for creation of network models but also for inserting full custom code to modify operation.

The network simulation is set up as a discrete time event simulation. This allows simulating network events with their corresponding timestamps as the algorithms operate. We build the whole scheme as an OMNET++ model that is composed of various components, corresponding to the components described in the architecture above. For example, the messages exchanged between models have been mapped to the message structure contained within OMNeT++

```
message DoSInputPower
{
    fields:
    int nodeID;
    int timestamp;
    int powercon;
}
```

Program listing 10: DoS attacks - structure of DoS InputPower

OMNeT++ also supports compound models which are components containing other components. Compound models were used for the algorithmic operation composing of two distinct models.

In order to test the operation of the system, network traffic needs to be generated. In OMNeT++ this can be easily performed using the highly customizable internal functions to generate traffic. These include predetermined traffic patterns but also random patterns. This is a code example of how we generate traffic on the simulator with many of the parameters being controlled by defined parameters on the project properties.

```
#include "NetworkStackTrafficGen.h"
#include <cassert>
#include "Packet.h"
#include "BaseMacLayer.h"
#include "FindModule.h"
#include "NetwToMacControlInfo.h"
#include "AddressingInterface.h"
Define_Module(NetworkStackTrafficGen);

void NetworkStackTrafficGen::initialize(int stage)
{
    BaseLayer::initialize(stage);

    if(stage == 0)
    {
          world = FindModule<BaseWorldUtility*>::findGlobalModule();
          delayTimer   = new cMessage("delay-timer", TRAFFIC_TIMER);
          arp = FindModule<ArpInterface*>::findSubModule(findHost());
          packetLength = par("packetLength");
          packetTime = par("packetTime");
          pppt = par("packetsPerPacketTime");
          burstSize = par("burstSize");
```

```
            destination = LAddress::L3Type(par("destination").longValue());

            nbPacketDropped = 0;
            BaseMacLayer::catPacketSignal.initialize();
        }
        else if (stage == 1)
        {
            AddressingInterface* addrScheme =
                        FindModule<AddressingInterface*>::findSubModule(findHost());

            if(addrScheme)
            {
               myNetwAddr = addrScheme->myNetwAddr(this);
            }
            else
            {
               myNetwAddr = LAddress::L3Type( getId() );
            }

            if(burstSize > 0)
            {
                remainingBurst = burstSize;
                scheduleAt(dblrand() * packetTime * burstSize / pppt, delayTimer);
            }
        }
        else
        {
        }
    }

    NetworkStackTrafficGen::~NetworkStackTrafficGen()
    {
        cancelAndDelete(delayTimer);
    }


    void NetworkStackTrafficGen::finish()
    {
        recordScalar("dropped", nbPacketDropped);
    }

    void NetworkStackTrafficGen::handleSelfMsg(cMessage *msg)
    {
        switch( msg->getKind() )
        {
            case TRAFFIC_TIMER:
              assert(msg == delayTimer);
              sendBroadcast();
              remainingBurst--;

             if(remainingBurst == 0)
              {
                remainingBurst = burstSize;
                scheduleAt(simTime() + (dblrand()*1.4+0.3)*packetTime*burstSize/pppt, msg);
              }
              else
              {
                          scheduleAt(simTime() + packetTime * 2, msg);
                       }
                       break;
              default:
                    EV << "Unkown selfmessage! -> delete, kind: "<<msg->getKind() <<endl;
                    delete msg;
                      break;
          }
    }

    void NetworkStackTrafficGen::handleLowerMsg(cMessage *msg)
    {
        Packet p(packetLength, 1, 0);
```

```
    emit(BaseMacLayer::catPacketSignal, &p);
    delete msg;
    msg = NULL;
}

void NetworkStackTrafficGen::handleLowerControl(cMessage *msg)
{
    if(msg->getKind() == BaseMacLayer::PACKET_DROPPED)
    {
        nbPacketDropped++;
    }

    delete msg;
    msg = NULL;
}

void NetworkStackTrafficGen::sendBroadcast()
{
    LAddress::L2Type macAddr;
    LAddress::L3Type netwAddr = destination;

    netwpkt_ptr_t pkt = new netwpkt_t(LAddress::isL3Broadcast( netwAddr ) ? "TRAFFIC->ALL" :
"TRAFFIC->TO", LAddress::isL3Broadcast( netwAddr ) ? BROADCAST_MESSAGE : TARGET_MESSAGE);

    pkt->setBitLength(packetLength);
    Packet appPkt(packetLength, 0, 1);
    emit(BaseMacLayer::catPacketSignal, &appPkt);
    pkt->setSrcAddr(myNetwAddr);
    pkt->setDestAddr(netwAddr);

    if(LAddress::isL3Broadcast( netwAddr ))
    {
        macAddr = LAddress::L2BROADCAST;
    }
    else
    {
        macAddr = arp->getMacAddr(netwAddr);
    }

    NetwToMacControlInfo::setControlInfo(pkt, macAddr);
    sendDown(pkt);
}
```

Program listing 11: DoS Attacks - generating traffic

In order to simulate parameters specific to wireless embedded systems, MiXiM models were used. MiXiM is an OMNeT++ modelling framework created for mobile and fixed wireless networks offering detailed models of radio wave propagation, interference estimation, radio transceiver power consumption and wireless MAC protocols.

As seen in the following picture, a wireless MiXiM network can be setup using the graphical user interface of the simulator that contains a certain amount of nodes, a traffic generation module as well as a module that records statistics of all the operations.

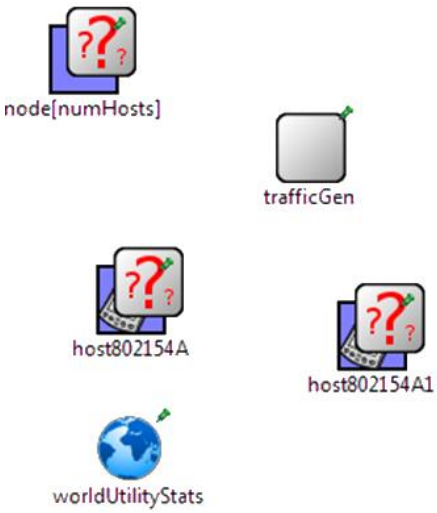


Figure 3-3: DoS Attacks - wireless MiXim network

Simulation runs have been made for a number of traffic patterns and configurations. The evaluation of results is difficult to be made before a working hardware prototype is created. The following average values of detection accuracy were derived after a significant number of runs.

Table 3-1: DoS Attacks – simulation results

	CPU abnormalities	Power abnormalities	Traffic abnormalities
Detection Accuracy	70%	60%	50%
False positives	15%	20%	15%

The detection accuracy refers to the percentage of simulation runs (with different parameters) where the inconsistent parameter was correctly detected. The False positive refers to the percentage of simulation runs where an alarm was issued without any real abnormality present.

These numbers, because of simulation confinements, do not correspond necessarily to real world data and were all caused by programmed conditions. The significant next step will be the creation of a real hardware prototype which will be based on the BeagleBone family of platforms. This will allow for real-world calculation of needed cofactors as well as real-world fluctuation parameters which will further test and evaluate the algorithm and the implementation.

3.2 Model-based framework for dependable distributed computation

Computation in a distributed environment introduces several complications compared to a “centralized” approach. In particular, it becomes necessary to account for faults within the array of resources allocated to the execution of a given application. Sometimes this requirement translates into the need to reconfigure the flow of execution, so to guarantee that a task will correctly run to completion.

Our model-based framework, called *Atta* (after the ant genus) addresses this problem and several others in the design and execution of distributed applications. A general overview of the framework is provided in D4.3; in this document we cover some details, both internal and related to the interfacing. More in-depth documentation can also be found within the public repository of the framework.

3.2.1 Artifact descriptors

An application in Atta is written using the dataflow paradigm, expressed as a graph comprising edges and vertices: edges represent data, and vertices represent implementations. Implementations may be just code sections to be executed, or they may represent a sub graph. Therefore we can see that the description of an application may become rather complex. To tackle such complexity and also to simplify abstraction/refinement development approaches, the descriptor of an application is hierarchical. This means that we do not store the entire application graph within one descriptor file, but we rather let descriptors reference other descriptors.

We generically define as an artifact an entity with information useful to define an Atta application. Artifacts are declared using descriptor files, which provide the basic information that Atta [3] needs to deploy, build and run applications. The two most important artifacts are data types and vertex implementations. For example, the descriptor file of a vertex implementation may specify the script that must be run to build the library (more on that in the following).

Practically, artifact descriptors are text files that use the YAML language [4]. YAML has been chosen because it offers a natural syntax with almost no overhead, since it relies on indentation: this is important to keep descriptor files readable, compared for example to XML or even JSON. Readability is not to be taken lightly, if we consider that designers must write their own descriptors to create an Atta-compliant distributed application; while IDE tools may mitigate the complexity of maintaining descriptors, manual editing always offers the maximum insight and control over the design process. Software libraries exist for several programming languages to automatically parse and produce YAML documents.

```
repositories
- name: myartifactname
  control: git
  entry: 'git@mysite.com:myself/myproject.git'
  path: myproject/path/to/myartifactname
  version: 477125dbe78fe0a51be2486d8902b49ec2161450
  mirrors:
  - 'git@thirdsite.com:myname/myproject.git'
  - 'git@fourthsite.com:othername/someproject.git'
```

Program listing 12: Repositories fragment of a YAML artifact descriptor

In Program listing 12, an example of a fragment of a descriptor is shown, which only lists the repositories element for an artifact. This example also illustrates our concept of repository, i.e., a remote location where we can find a versioned repository containing an artifact. The fragment shows the coordinates of the repository, that includes a version control system (Atta supports Git/Subversion/Mercurial repositories), an entry URL, a path within the repository and the version of the content. In addition, optional mirror entries are provided for redundancy purposes.

By design, a repository identifies one specific artifact; the descriptor for the remote artifact can be found as an *artifact.yaml* file at the root of the repository path. In other terms, it is possible to have one versioned repository with all the required artifacts organized into directories, and declare repository entries for each remote artifact of interest.

3.2.2 Data types

As explained before, the two main artifacts in Atta are data types and vertex implementations, since their composition allows defining a dataflow application graph. Data types are the most delicate aspect of a distributed computation framework, since they represent the “glue” that connects the different parts of an application. Our ultimate goal is to define complex types through which implementations can communicate in a safe and practical way.

```
artifact: datatype
name: town
fields:
- name: name
  type: text
- name: loc
  fields:
  - {name: province, type: text}
  - {name: region, type: text}
  - {name: state, type: text}
  - {name: coords, type: real, array:
'2:3'}
- name: props
  type: text
array: ':,2'
```

Program listing 13: DDC - YAML artifact descriptor of a type

An artifact definition of a data type is provided in Program listing 13. Here we see that there exist some *built-in* types, namely boolean, byte, real and text; these types are sufficient to describe simple types, but to do more we must compose them. Consequently the town data type is a composite type featuring both scalars and multi-dimensional arrays. More than that, it is *hierarchically* composite: the loc field is itself a composite type and it is declared *inline*.

When we reference a repository pointing to this artifact as in Program listing 12, i.e., when we have an *external* declaration of a type, we provide a custom name that may override the name of the artifact. In fact, the name field in the artifact descriptor is optional: this choice is due to the fact that externally declared types are reusable and may have a different name in different domains; also, it is preferable to have all type names explicit within a descriptor. If we consider the descriptor fragment of Program listing 14, where the xy field references a custom type, the type name corresponds to a repository name within the repositories element of the same descriptor file. The types element is where *internal* declaration of types are provided: when the validity of a type is restricted to the artifact declaring it, we can save ourselves the burden of setting up and referencing a repository.

```
repositories:
- name: 2Dcoordinates
  control: svn
  entry: 'http://mysite.com/someproject'
  path: types/xy
  version: 19
types:
- name: 3Dcoordinates
  fields:
  - {name: xy, type: 2Dcoordinates}
  - {name: z, type: real}
```

Program listing 14: DDC - Types and repositories fragment of a YAML artifact descriptor

```
artifact: datatype
fields:
- {name: x, type: real}
- {name: y, type: real}
```

Program listing 15: DDC - YAML artifact descriptor of a type for an x-y couple of reals

3.2.3 Data communication

The Atta architecture, as explained in D4.3, consists of six different kinds of nodes, also called *roles*. Those roles that directly involve the communication of application data are worker nodes (*wnodes*), synchronization nodes (*snodes*), and persistence nodes (*pnodes*).

The array of snodes is currently implemented using Apache ZooKeeper [5]. ZooKeeper is a replicated synchronization service with eventual consistency. It is robust, since the persisted data is distributed between multiple nodes (this set of nodes is called an ensemble) and one client connects to any of them (i.e., a specific server), migrating if one node fails; as long as a strict majority of nodes are working, the ensemble of ZooKeeper nodes is alive.

More in detail, a master node is dynamically chosen by consensus within the ensemble; if the master node fails, the role of master migrates to another node. The master is the authority for writes: in this way writes can be guaranteed to be persisted in-order, i.e., writes are linear. Each time a client writes to the ensemble, a majority of nodes persist the information: these nodes include the server for the client, and obviously the master. This means that each write makes the server up-to-date with the master. It also means, however, that you cannot have concurrent writes. As for reads, they are concurrent since they are handled by the specific server, hence the eventual consistency: the view of a client is outdated, since the master updates the corresponding server with a bounded but undefined delay.

The guarantee of linear writes is the reason for the fact that ZooKeeper does not perform well for write-dominant workloads. In particular, it should not be used for interchange of large data, such as media. The advantage that ZooKeeper brings to Atta is to be able to robustly listen for events and to issue them to all listeners. In addition, it can be used as an arbiter for consensus algorithms, like those related to scheduling and load balancing. For example, a very important event to listen to is the completion of a transaction to the pnodes, signalling the availability of data returned by a vertex implementation.

While a ZooKeeper ensemble could also be used as a set of pnodes, linear writes would be detrimental for heavy data streams. For the maximum generality, we consequently envision pnodes to be nodes of a distributed (replicated) database. The current DBMS of choice is MySQL Cluster, which offers both SQL and “NoSQL” APIs; the second one is particularly useful for the persistence of custom data structures such as all Atta data types. It is important to say that the framework may accommodate a different persistence technology with a rather simple change of driver library. On the contrary, Apache Zookeeper is to be considered a consolidated choice that will be replaced only if a more efficient and versatile solution is identified. Both snodes and pnodes are not by themselves “Atta-aware”, in the sense that they operate as generic services with which wnodes can interact; all the logic required to interface with snodes/pnodes resides within wnodes.

In terms of reliability, we purposefully avoided any single-point-of-failure situation that may arise in a distributed environment. Wnodes “push” the data they produce to the pnodes, and “pull” the data they need to consume from the pnodes. This decoupling shields from node failures and allows saving data as soon as it is produced. Compare this approach to the opposite one, where data is simply transferred from producers to consumers: as soon as one link of the chain breaks, the application needs to be restarted from scratch since we have not saved any “snapshot” of the application state.

It could be argued that this methodology introduces inefficiencies, and it is certainly such a case. However, we believe that a paradigm shift is necessary to be able to address the intrinsic problems in a

distributed environment. This is especially true for mobile ad-hoc networks, where connectivity and autonomy are relevant obstacles to collaboration.

3.2.4 Data conversion

We remind here that Atta is a Java framework that supports applications written in different languages. This generality implies the following: interprocess communication is required to transfer data between the middleware and any process that runs the code for a vertex. Also, since vertex implementations run on platforms with different data type precisions; even the “same” data types may have different representations among the wnodes. While a common representation is available at middleware level due to the platform-independent type system of Java, it is still necessary to account for the specific platform.

To account for these problems, Atta employs Apache Thrift [6], a Remote Procedure Call framework for heterogeneous services. Given a generic representation of a service and the data types of its arguments/return, Apache Thrift produces source files for client and server implementations. These source files can be used to perform cross-language remote communication.

Our current use of Apache Thrift within Atta is for interprocess communication: all the vertexes that are executed on a wnode have their own process, and each process exchanges data with the Atta middleware using Thrift. It must be noted that we can extend this approach for inter-wnode data communication with no effort. This would allow the exchange of application data between wnodes directly, bypassing pnodes. While such a choice would decrease the communication latency, we would break our guarantees of data persistence against node failures. It is still being investigated how to properly offer the best of the two worlds in an autonomic way.

4 Reputation-based resource management technologies

4.1 Reputation based Secure Routing

In distributed systems, each entity must depend on its neighbours to carry out a transaction and accomplish full communication among all participants. Routing protocols [7] are implemented for this purpose. Their basic functions are routing and forwarding. Routing is the process of establishing a communication path between two end nodes. Forwarding is the transmission of the traffic through the selected path. Most routing protocols base the routing process on distance metrics among a path's nodes. The protocol selects the shortest path.

Reputation-based schemes [8] are used in wireless networking to provide secure routing functionality. Due to the open medium and the dynamic entrance of new nodes to such networks there must be a way to establish trust relationships to avoid malicious entities. Reputation is formed by a node's past behaviour and reveals its cooperativeness. A node with high reputation can be considered as trustworthy. Legitimate nodes depend mostly on trustworthy entities to accomplish communication tasks, like routing and forwarding. Also low reputation can reveal selfish or malicious entities and is used for intrusion detection. Legitimate nodes try to avoid such entities and not serve their traffic.

A common approach for implementing secure routing functionality is the integration of a routing protocol with a reputation scheme. Reputation and trust information are included in the decision making process along with the distance metrics. For the routing process, the goal is the selection of short paths with well-reputed nodes. Thus, legitimate entities avoid malicious ones. For the forwarding process, the goal is to serve only legitimate entities and isolate the malicious or selfish ones.

Many reputation-based schemes for secure routing have been proposed and each one provides protection against a set of security attacks and vulnerabilities. The schemes embody features to form reputation and trust. These features add complexity and process overhead to the pure routing protocol. The basic trade-off that is encountered is between security and performance. As security goes high, the overhead to support the level of security goes also high and the system becomes more complex. The selection of the proper scheme depends on the application properties. Networks with ultra-constraint devices cannot support heavy reputation-based schemes that offer high levels of security. A network manager has to apply one of the proposed implementations without having the ability to adapt the scheme to its application's needs.

4.2 nSHIELD Reputation scheme

For nSHIELD network layer, we implement a novel module reputation-based scheme that can act as a general purpose scheme for a wide range of applications. The basic idea is to identify the common components of reputation-based schemes and provide an abstract framework. We identify eleven components where each one of them serves a specific functionality. For every component we propose a set of features that implements the component's functionality. The segmentation of the scheme into components enables the dynamic deployment and extension of the scheme. As new features and trends are proposed in the field of reputation-based schemes, we can simple implement these features in the components container.

The network manager selects which components are active and the exact set of features that implements them. During the selection process, a designer could model the combination of more than one feature for some components. The designing options can range from ultra-lightweight schemes to heavily secure ones. The selection decision will be either static at deployment time or dynamic at run time, if such operation is supported. Also, heterogeneous nodes could utilize different features for some components.

The eleven components are:

1. Knowledge type

2. Transaction evaluation
3. Transaction grading
4. Reputation evaluation scope
5. Reputation calculation
6. Notification strategy
7. Notification scope
8. Indirect trust evaluation
9. Punishment strategy
10. Initialization and re-entrance strategy
11. Path selection

Knowledge type is categorized as direct and indirect. Direct knowledge is the direct opinion that an entity possesses about other entities and is determined by their previous transactions or observations of certain factors. Indirect knowledge is the opinion that other entities possess about the investigated entity.

Transaction evaluation defines the result of a transaction and decides what will happen next. Simple evaluation denotes the transaction result (success or fail) and proceeds to the next step (transaction grading component). Congestion windows and communication channel observation can be used as a tolerance mechanism if failures occur during periods of traffic congestion and bad channel conditions respectively. Another option is the rerouting of a failed transaction.

The transaction grading defines the exact value that is applied for the examined transaction, after the transaction evaluation component execution. Simple and gradual grading is implemented.

The reputation evaluation scope indicates which parts of the network are about to be examined by the reputation scheme. Three categories are considered: node, path and community of nodes.

The reputation calculation determines the current reputation value of the examined entity. Four formulas are supported. The simple summation is the summation of the transaction grading values. The reputation fading stores a small history of grading values. The values are weighted according to time and reputation fades – indicating that most recent values are considered more important. The reputation normalization defines a statistical normalization of the reputation history, where the extreme values are ignored. The reputation fading of the normalized history combines the two approaches to achieve higher level of security.

If indirect knowledge is supported, the trust calculation can categorize nodes as trusted, legitimate, suspicious or malicious, based on their reputation value. Three types of notifications are supported: positive, negative, and positive/negative.

The scope of the notification determines which nodes are about to receive a notification. Three scopes are considered: broadcast, trusted/friends, and the misbehaving node (for negative notifications).

Indirect trust evaluation defines three formulas for evaluating the notifications that are received by other nodes. With simple evaluation, the node processes all notifications the same. With deviation test, the node checks if the notification it receives, deviates significantly from its direct knowledge. With the weighted evaluation, the notifications that are sent by trusted nodes gain higher weight.

A punishment strategy defines the thresholds for marking suspicious and malicious nodes as well as the type of punishment. Punishment types include the discarding from the routing process, the termination of packet forwarding for the punished nodes, the combination of routing and forwarding punishment, and warning messages to inform the other nodes (if indirect knowledge is supported).

A re-entrance strategy may allow a punished node to re-enter the network with a default reputation value under some conditions. The strategies that are currently supported are the periodic re-entrance after T minutes, redemption, and no re-entrance.

Path selection indicates the criteria for deciding which path to choose during the routing process. We support the shortest path, the most well reputed path and the shortest well reputed path.

Table 4-1: Reputation technologies - Features of the reputation & trust scheme and their supported implementations

Parameters	Implementations
Knowledge type	Direct (default)
	Indirect
Evaluation scope	Node (default)
	Path
	Community
Indirect notification type	No notification (default)
	Positive
	Negative
	Positive/Negative
Transaction evaluation	Simple evaluation (default)
	Rerouting
	Congestion windows
	Channel observation
Transaction grading	Simple grading (default)
	Gradual grading
Reputation calculation	Simple summation (default)
	Reputation Fading mechanism (Bayes or beta distribution)
	Reputation normalization mechanism
	Fading of the normalized reputation
Punishment strategy	Forwarding (default): stop forwarding packets for punished nodes
	Routing: stop including paths with punished nodes in the routing process
	Routing & forwarding: stop both including paths with punished nodes in the routing process & forwarding their packets
	Send a warning message to inform the other nodes
Initialization & re-entrance strategy	No re-entrance is allowed (default)
	Periodic: allow a punished node to re-enter the network with a default reputation after T minutes
	Redemption: all bad ratings are recalled to neutral ones after Rt minutes
Path selection	Reputed path (default)
	Shortest path
	Shortest reputed path

The most important operation of a reputation-based scheme is the calculation of reputation and trust. A node continuously receives new pieces of knowledge both from its direct interaction with its neighbours and the notifications from other nodes. There are two evaluation operations for direct and indirect

knowledge respectively. When new knowledge has been evaluated, the reputation and trust values are updated. If the trust level of the node has changed, notifications can be sent.

The above sample of code implements the main interactions between the 11 components and reasoning process for evaluating new pieces of knowledge.

For direct knowledge:

```
// FOR DIRECT KNOWLEDGE : receive new direct knowledge
void DSRAgent::receiveKnowledge(Path route, ID target, bool success) {
    double weight = 1.0, grade;
    bool WasTrustWorthy = route_cache->TRS.IsTrustworthy(target);
    bool WasPunished = route_cache->TRS.IsPunished(target);

    if(!success)    // if the examined transaction has been failed --> Report target node
        weight = transactionEvaluation(target);
    grade = transactionGrading(weight);
    reputationTrustCalculation(route,target,grade);
    sendNotification(target,WasTrustWorthy,WasPunished);
}
```

Program listing 16: Reputation technologies – receiving new direct knowledge (DK)

```
// Return the weight of the transaction result to the 'TransactionGrading'. [Weight from -1 to 0]
double DSRAgent::transactionEvaluation(ID id) {
    double weight = -1.0;   // Simple Evaluation (default option) => [weight=-1.0, tolerance=1.0]

    // Set weight tolerance acording to specified criteria [Re-route capability, Congestion window,
    // Channel observation]
    if(transaction_evaluation==EVALUATION_REROUTE)
        weight = reroute(id);
        // Don't report an error message
    if(transaction_evaluation==EVALUATION_CONGESTION_WINDOW)
        weight = CongestionWindow(id);
    if(transaction_evaluation==EVALUATION_CHANNEL_OBSERVATION)
        weight = ChannelObservation(id);
    return(weight);
}
```

Program listing 17: Reputation technologies – returning weight of transaction result (DK)

```
// Get the weight from the 'TransactionEvaluation' and return the final grade for the examined transaction
double DSRAgent::transactionGrading(double weight) {
        // Simple Grading (default option)  => [positive_value=1, negative_value=-1]
    double positive_value = 1.0;
    double negative_value = 1.0;

    if(transaction_grading==GRADING_GRADUAL) {  // Gradual Grading => [positive_value=1, negative_value=-2]
        positive_value = 1.0;
        negative_value = 2.0;
    }

        // Transaction Grading
    if(weight>=0.0) // CASE REWARD -> a target node for a successful transaction
        return(positive_value * weight);
    else    // CASE REPORT -> a target node for an unsuccessful transaction
        return(negative_value * weight);
}
```

Program listing 18: Reputation technologies – transaction grading (DK)

```
// FOR DIRECT KNOWLEDGE : Calculate the new reputation & trust after this transaction
void DSRAgent::reputationTrustCalculation(Path route, ID target, int grade) {
    // Calculate Reputation & Trust
    if(direct_knowledge_scope==SCOPE_NODE)
        route_cache->TRS.Rank(0, target,grade);     // Rank a Node  (DEFAULT)
    else if(direct_knowledge_scope==SCOPE_PATH || direct_knowledge_scope==SCOPE_COMMUNITY)
        route_cache->TRS.Rank(0, route,grade);      // Rank a Path or a Community
    else
        cout << "WARNING (reputationTrustCalculation): the direct_knowledge_scope isn't declared properly"
             << endl << flush;
}
```

Program listing 19: Reputation technologies – calculating new trust and reputation (DK)

For indirect knowledge:

```
// FOR INDIRECT KNOWLEDGE : receive new indirect knowledge
void DSRAgent::receiveKnowledge(Path route, ID target, ID notifier, InDirectTrust IT) {
    bool WasTrustWorthy = route_cache->TRS.IsTrustworthy(target);
    bool WasPunished = route_cache->TRS.IsPunished(target);

    reputationTrustCalculation(route, target, notifier, IT);
    sendNotification(target,WasTrustWorthy,WasPunished);
}
```

Program listing 20: Reputation technologies – receiving new indirect knowledge (IK)

```
// FOR INDIRECT KNOWLEDGE : Calculate the new trust & reputation after this transaction
void DSRAgent::reputationTrustCalculation(Path route, ID target, ID notifier, InDirectTrust IT) {

    // Calculate Reputation & Trust
    if(indirect_knowledge_scope==SCOPE_NODE)
        route_cache->TRS.Rank(0,target,notifier,IT);    // Rank Node    (DEFAULT)
    else if(indirect_knowledge_scope==SCOPE_PATH || indirect_knowledge_scope==SCOPE_COMMUNITY)
        route_cache->TRS.Rank(0,route,notifier,IT);     // Rank a Path or a Community
    else
        cout << "WARNING (reputationTrustCalculation): the indirect_knowledge_scope isn't declared properly
             << endl << flush;
}
```

Program listing 21: Reputation technologies – calculating new trust and reputation (IK)

```
// Send notifications according to the notificatino strategy when a node's status changes
void DSRAgent::sendNotification(ID target, bool WasTrustWorthy, bool WasPunished) {
    // Create new indirect knowledge from this node
    if(knowledge_type==KNOWLEDGE_INDIRECT && indirect_notification_type!=INDIRECT_NOTIFICATION_NONE) {
        cout << "\t\t**** INDIRECT KNOWLEDGE: " << Scheduler::instance().clock() << endl << flush;
        bool send_new_indirect_knowledge = false;
        char mydata[20];

        if(indirect_notification_type==INDIRECT_NOTIFICATION_POS_NEG ||
           indirect_notification_type==INDIRECT_NOTIFICATION_POSITIVE) {
            if((!WasTrustWorthy && route_cache->TRS.IsTrustworthy(target))) {
                // make a positive notificaiton
                send_new_indirect_knowledge = true;
                strcpy(mydata, "TRUSTED ");
                strcat(mydata,target.dump());
                cout << "\t\t**** SEND POSITIVE NOTIFICATION [" << mydata << "]" << endl << flush;
            }
        }
```

Program listing 22: Reputation technologies – creating new indirect knowledge (IK)

```
        if(indirect_notification_type==INDIRECT_NOTIFICATION_POS_NEG ||
           indirect_notification_type==INDIRECT_NOTIFICATION_NEGATIVE){
            if(!WasPunished && route_cache->TRS.IsPunished(target)) {   // make a negative notificaiton
                send_new_indirect_knowledge = true;
                strcpy(mydata, "NEG ");
                strcat(mydata,target.dump());
                cout << "\t\t**** SEND NEGATIVE NOTIFICATION [" << mydata << "] " << endl << flush;
            }
        }

        if(indirect_notification_type==INDIRECT_NOTIFICATION_POS_NEG){  // back to normal trust state
            if(WasTrustWorthy && !route_cache->TRS.IsTrustworthy(target)){  // undo a positive notification
                send_new_indirect_knowledge = true;
                strcpy(mydata, "NORMAL ");
                strcat(mydata,target.dump());
                cout << "\t\t**** SEND NORMAL NOTIFICATION [" << mydata << "]" << endl << flush;
            }
            else if(WasPunished && !route_cache->TRS.IsPunished(target)){   // undo a negative notification
                send_new_indirect_knowledge = true;
                strcpy(mydata, "NORMAL ");
                strcat(mydata,target.dump());
                cout << "\t\t**** SEND NORMAL NOTIFICATION [" << mydata << "]" << endl << flush;
            }
        }

        if(send_new_indirect_knowledge == true){            // SEND INDIRECT KNOWLEDGE -- NOTIFICATION TYPE
            SRPacket p;
            AppData* pkd = new PacketData(strlen(mydata));
            ((PacketData*) pkd)->setmydata(mydata, strlen(mydata));

            p.src = net_id;
            p.pkt = allocpkt();

            hdr_sr *srh = hdr_sr::access(p.pkt);
            hdr_ip *iph = hdr_ip::access(p.pkt);
            hdr_cmn *cmnh = hdr_cmn::access(p.pkt);

            p.pkt->setdata(((PacketData*) pkd)->copy());

            sendIndirectNotification(target,p,indirect_notification_type);
        }
    }
}
```

Program listing 23: Reputation technologies – sending new indirect knowledge (IK)

Other important operations are the path selection and the entrance of new or previously punished nodes. When a new communication path is established, the scheme has to decide which nodes will be included. Through the path selection component we can denote the selection strategy of the scheme. Also, if the re-entrance strategy is enabled, it is periodically examined the case of permitting to punished nodes to re-enter the network.

We implement a GUI to ease the framework's configuration. All the aforementioned component and parameters are described and the user can select the most appropriate for his network. Before the configuration process is completed, all the parameters are presented to the user for a final confirmation. To make our proposal more applicable and acceptable, we have pre-set the configuration options for implementing the decision making process of well-known reputation and trust schemes for secure routing. These schemes are the Watchdog and Pathrater [9], CONFIDANT [10], Improved CONFIDANT [11], CORE [12], Reputated-ARAN [13], CSRAN [14], RFSN [15] and a Semi-distributed reputation-based Intrusion Detection System for mobile Ad-hoc Networks [16].

The network designer is able to add functionality, increase the level of security and adopt the final scheme to his needs. Moreover, the configuration parameters can be altered at runtime. Consider a scenario where we have set the Watchdog and Pathrater scheme in a WSN with nSHIELD nano nodes. Then, the overlay layer becomes aware of an emergence situation and informs the overlay security agents to increase the security level of the underling networks. The security agent, who manages the examined WSN, checks its policy and informs the sensor nodes to increase their security. The policy orders a specific set of actions, like lowering the threshold for malicious nodes and applying the routing and

forwarding punishment strategy. The WSN is conformed to the new policy, becomes stricter with misbehaving nodes and isolates the malicious ones. Similarly, when the emergence situation is over, the overlay security agents can set the underling networks back to their normal form.

We integrate the Dynamic source routing (DSR) protocol with our trust and reputation-based scheme. The DSR is implemented in NS2 and C++ and our scheme extends this implementation. DSR is designed for wireless mesh networks. Most of the trust and reputation systems that are examined in this paper extend this protocol. DSR performs well in static and low-mobility environments. The routing overhead is proportional to the length of the path. The network animator (NAM) is used for demonstrating different application scenarios.

The figure below, illustrates a demonstration scenario with NAM. The pre-defined parameters for implementing the decision making process of CONFIDANT are selected. Nodes 0 and 4 communicate through the secure routing protocol. Node 0 sends packets to node 4. Nodes 1 and 3 are legitimate intermediates. Node 2 is a malicious intermediate, which tries to perform a black hole attack to the routing protocol. The circles are route requests and their circumference reveals a node's transmission range. The route [0, 2, 4] is initially selected as the shortest path (all nodes start with a default neutral reputation value). As node 2 begins to drop packets, node 0 lowers node's 2 reputation value. When the reputation reaches the malicious threshold, node 0 denotes node 2 as malicious. Then selects the other path [0, 1, 3, 4], which doesn't' include the punished node 2, continues the communication with node 4 and counters the attack. The scheme improves the performance of DSR under attacks as fewer failures occur and the communication isn't obstructed.

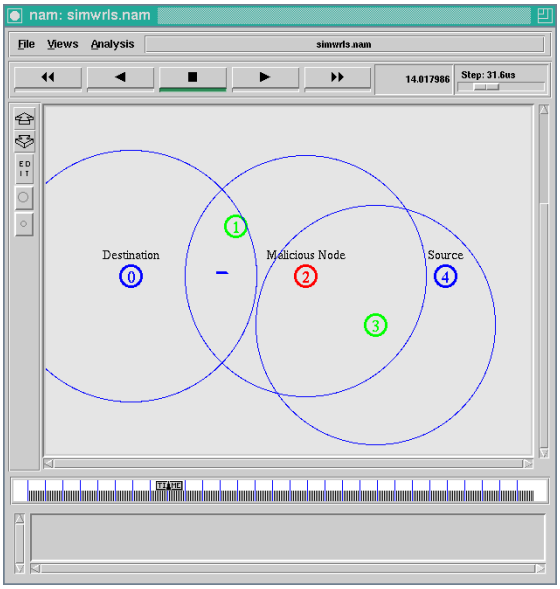


Figure 4-1: Reputation technologies - Animated example of the proposed reputation and trust scheme by NAM

From the pre-defined schemes, CONFIDANT implements the most robust reputation-only system, applicable to nano nodes. To even increase the security level for this type of devices, we can easily extend the scheme with gradual grading and negative/positive notifications (instead of simple grading and negative notifications). Furthermore, these configurations can take place at runtime to efficiently counter attacks that target the reputation scheme, like ballot- and topology-based attacks.

4.2.1 Trusted GPSR implementation

The purpose of the Reputation and Trust component is to guarantee a robust routing operation by bypassing or even isolating nodes with malicious behaviour. This section presents a more detailed investigation of the reputation and trust scheme prototype developed for TinyOS-based platforms. For clarity reasons an introduction of some TinyOS concepts is included prior to a more detailed analysis of

source code points of importance. Finally some tools used for debugging and validation of proper routing operation are presented at the end of this section.

4.2.1.1 TinyOS Components and Interfaces

TinyOS is an open-source small footprint operating system targeting networked sensors developed at Berkeley University and used in hundreds of research projects up until now. TinyOS applications are written in nesC (network embedded system C), which is C with some additional language features for components and concurrency. A nesC application consists of one or more **components** assembled, or wired, to form an application executable. A sensor node (mote) runs only one executable at a time which consists of all components needed for this application. Components define two scopes: one for their specification which contains the names of their interfaces and one for their implementation. Specification is a code block that declares the **interfaces** the component **provides** and **uses**. The provided interfaces are intended to represent the functionality that the component provides to its user in its specification while the used interfaces represent the functionality the component needs to perform its task in its implementation. For every function that a component provides in its specification an implementation must be defined, so other components can call it. Conversely, every used function depends on some other components which provide their implementation. TinyOS applications are constructed on two types of components: **modules** and **configurations**. Modules provide the implementations of one or more interfaces. Configurations are used to assemble other components together by connecting interfaces used by components to interfaces provided by others. Every nesC application is described by a top-level configuration that wires together all the components inside.

The set of interfaces a component uses and provides defines its signature. Interfaces are bidirectional: they specify a set of **commands** which are functions to be implemented by the interface's provider, and a set of **events** which are functions to be implemented by interface's user. A single component may use or provide multiple interfaces and multiple instances of the same interface.

A component can only reference variables from its own, local namespace and can't access variables in any other components. However a component can declare that it uses a function defined by another component. The composition of nesC programs involves writing components and wiring users to providers. As this occurs at compile time, runtime allocation or storing function pointers in RAM is not required. A nesC program knows the complete call graph at compile time.

TinyOS applications are built on static resource allocation, meaning that memory allocation for the network, sensors, UART and other OS services is done at compile-time. This also helps in better composition since components reserve the amount of memory they need, making total memory requirements checkable at compile-time. Although there are situations where this can lead to RAM waste it offers another level of protection against bad use of dynamic resource allocation.

4.2.1.2 Tasks and Scheduler

TinyOS has two basic computational instructions: asynchronous events and tasks. Tasks in TinyOS are a form of deferred procedure calls that enable components to perform general purpose background processing in an application. A task is a piece of code the execution of which will start later from the TinyOS scheduler. The **post** operation places the task on the internal task queue which is processed in FIFO order. A task cannot be preempted from another task and completes before the next task starts running. Tasks are allowed to be preempted only by hardware interrupts. For this reason, lengthy operations must be dispatched to a series of separate tasks which execute a part of the whole operation. Every long-running application can be written as **split-phase** operation. In a split-phase system when a program calls a long-running operation the call returns immediately and the called abstraction issues a callback when it completes. Two separate phases of execution are present under this scheme, execution invocation and completion. For example:

```
//start phase           (e.g. send();)
//completion phase      (e.g. sendDone();)
```

Although split-phase code is more complex than sequential code it offers certain advantages in memory usage and system responsiveness. Situations where an application needs to take some action and the system is blocked by a lengthy task are avoided. Split-phase interfaces are a mechanism to achieve a form of execution parallelism in TinyOS.

The TinyOS 2.x scheduler is implemented as a component that provides the *Scheduler* interface. The Scheduler interface has commands for initialization and running tasks and is used by the operating system to execute tasks in the appropriate order. The scheduler follows a simple FIFO policy but this can change by replacing the default scheduler with one that implements a different scheduling policy like Earliest Deadline First or Rate Monotonic scheduling. Otherwise if two tasks run in a particular order, it can be enforced by the earlier task posting the later task. The default TinyOS scheduler implementation is the module *SchedulerBasicP*. *McuSleep* function called by this module can be used to put microcontroller in low-power mode when no tasks are waiting in the queue.

TinyOS functions that can preemptively run are labeled with the **async** keyword: they run asynchronously with regards to tasks. A function that isn't asynchronous is synchronous (or sync). By default, commands and events are sync. Interface definitions specify whether their commands and events are async or sync. On the other hand all interrupt handlers are async and they cannot include any sync functions in their call graphs. The only way to execute a sync function within an interrupt handler is to post a task.

The main benefit of using tasks is the prevention of race conditions. Preemptive execution can modify the state of an underneath ongoing calculation which can cause a system to enter an inconsistent state. For this reason, TinyOS code must be kept synchronous whenever possible and async code should be used only if time is very important or if it might be used by something whose timing is important. NesC provides a mechanism which prevents preemption of the executed code through the use of **atomic** statements. Atomic statements are used to implement mutual exclusion, e.g. updating concurrent data structures. Atomic sections are implemented by disabling interrupts.

4.2.1.3 TinyOS Communication Interfaces

Being concentrated in network-centric devices TinyOS provides structures and interfaces to abstract the underlying communication services such as sending and receiving packets. In TinyOS, the basic network abstraction is active message a single hop unreliable packet. Active messages have a destination address, can provide synchronous acknowledgements and can be of variable length up to a fixed maximum size. A type field is also included which is essentially a protocol identifier for components built on top of this abstraction. It must be noted that active messages is an abstraction that can be used with different standard or proprietary MAC protocols. In case of IEEE 802.15.4 MAC, TinyOS Frames by default have the format of Figure 4-2 where 6lowpan is the NALP code to identify that this is a TinyOS packet (value 0x3F) and AM Type is a single byte field which indicates of active message type of the packet.

Figure 4-2: Trusted GPSR - Active Message type position in 802.15.4 Frames

Packet level communication in TinyOS has three basic classes of interfaces. *Packet* interfaces for accessing message fields and payloads, *Send* interfaces for transmitting packets and *Receive* interface for handling packet reception events. Depending on whether the protocol has a dispatch identifier field the *Receive* and *Send* interfaces may be parameterized in order to support multiple higher-level clients. *Packet* and *AMPacket* are the two basic interfaces of the first class. The *Packet* interface provides access to data payload. *AMPacket* provides additional handling such as setting and reading source and destination addresses and active message type. For active messages communication *AMSend* is the interface used for packet transmission to a specific destination AM address (with 0xFFFF denoting broadcast). TinyOS components that use the *AMSend* interface can simply send network packets using the command:

```
command error_t send(am_addr_t addr, message_t* msg, uint8_t len);
```

after setting source and destination addresses in *am_addr_t* structure, filling transmission buffer payload in *message_t* structure and setting packet length. Similarly a user of Receive interface has to write handling code to the event

```
event message_t* receive(message_t* msg, void* payload, uint8_t len);
```

which denotes the reception of a new frame. The user can handle all received packet header and payload information using *msg* and *payload* structures (usually working on a copy of payload).

4.2.1.4 Implementation of GPSR prototype with Trust and Reputation enhancements

During source code prototype development a number of new TinyOS interfaces where implemented to enable trust-aware multi-hop communication for sensor nodes (Figure 10). *GPSRNeighborhood* provides a number of commands for handling neighbouring nodes. Apart from *start* command to enable split-phase operation *startDiscovery* is used to enable periodic beacon messages broadcasting. *UpdateNeighborList* is called every Beacon period to remove from my neighbours list the nodes from which beacon messages haven't received after a number of periods. *AddNeighbor* is responsible for adding a node in the structure of one node's neighbours if there is enough room to do so and if the address of this node isn't already stored in the list. There are also a number of functions for accessing the neighbours list (*GetNeighbor*, *GetNeighborById*, *GetSpecNeighbor*) and for getting and setting parameters of operation like the total number of packets received from a node, the number of them that has been acknowledged, the number of packets that this node has successfully forwarded, the remaining energy and RSSI level of this node and its involvement in the reputation scheme.

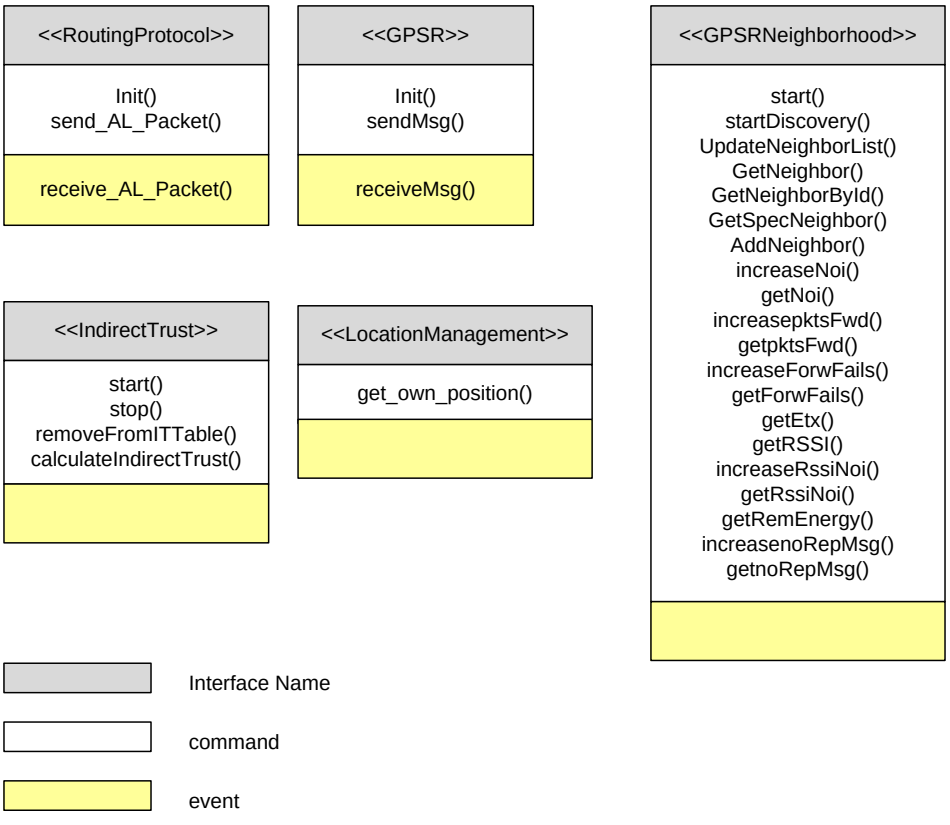


Figure 4-3: Trusted GPSR - TinyOS interfaces

RoutingProtocol and *GPSR* interfaces provide commands for sending routing and GPSR packets and callback functions that must be implemented when receiving the corresponding event [17]. The distinguish

between these two interfaces has to do with the fact that *RoutingProtocol* is a higher level component used by top level components which need multi-hop routing functionality whereas *GPSR* is the component providing a specific routing behaviour based on Greedy Perimeter Stateless Routing [18] enhanced with trust. In the future other routing algorithms could be used and configured to be active instead of GPSR. Functionality related to reputation based scheme is provided from *IndirectTrust* interface while *LocationManagement* interface can give the current position of the node which in the case of current implementation is a fixed value but in future versions could be the output of a GPS device for mobile nodes.

For the Trusted-GPSR (T-GPSR) prototype presented in this section the interactions between the different TinyOS components is presented in Figure 4-3. Apart from the new interfaces described above, standard TinyOS interfaces for sending, receiving and handling packets, queuing, timer functions and voltage reading have been used.

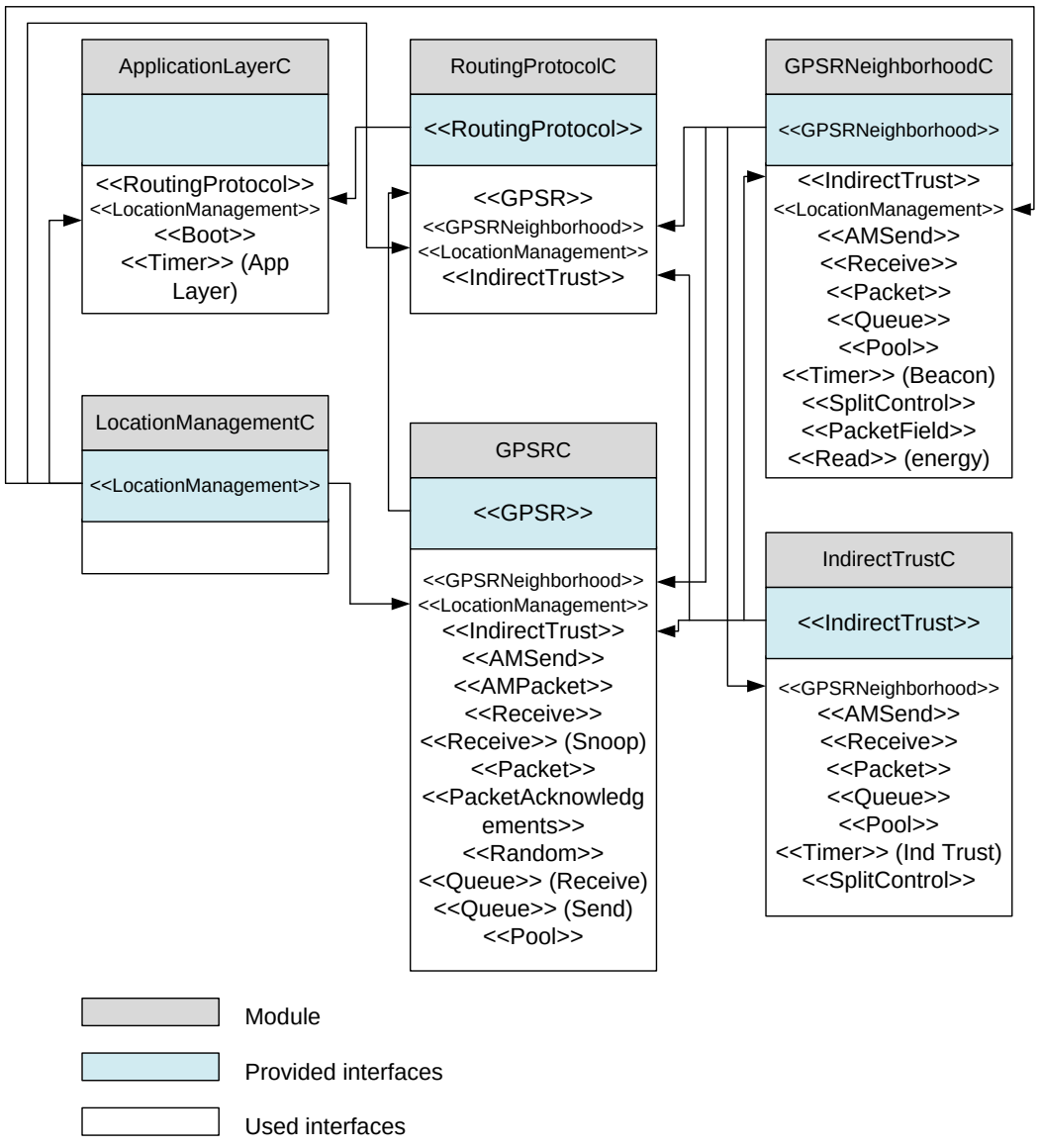


Figure 4-4: Trusted GPSR - Components wiring with provided and used interfaces in T-GPSR implementation

4.2.1.5 Frame formats

1. *Beacon Frame*

GPSR is based on proactive beaconing broadcasted at periodic intervals with the purpose for each node to advertise its existence and its coordinates in its neighbourhood. On receiving a beacon frame from a neighbour, node stores this information (address and coordinates of the neighbouring node) in a local table. In greedy mode of GPSR operation the node that is geographically closest to the destination will be chosen from the set stored in the neighbours table. An extension to standard GPSR included in this implementation is the transmission of the current energy level of the node which can help in routing decisions by avoiding nodes whose energy has dropped below a certain threshold.

The frame format of the beacon message is presented in Figure 4-5.

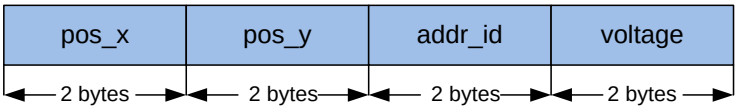


Figure 4-5: Trusted-GPSR - Beacon Frame format

where:

- pos_x: X coordinate of the node expressed in 2-bytes range
- pos_y: Y coordinate of the node expressed in 2-bytes range
- addr_id: unique identity of the node
- voltage: current voltage value of the node

2. *Network Layer Header of data frames*

Each time a node wants to send information about sensor values, services available or configuration parameters a data packet is created which besides the payload information must have filled the network layer header which in the case of T-GPSR presented in this section has the format of Figure 4-6. Destination identity and (x, y) coordinates are of primary importance as are used as means of identification of final destination arrival and for geographic forwarding decisions. A significant number of network layer header fields is occupied from various perimeter mode location coordinates (a detailed description of perimeter mode operation in GPSR is presented in [18]) as this mode of operation is more complicated and more information for geographically select the next neighbour is needed. It must be noted also that all the fields after the Len field presented in Figure 4-6 are optional and their primary use in this implementation is the increase of debugging and observation capabilities during network operation.

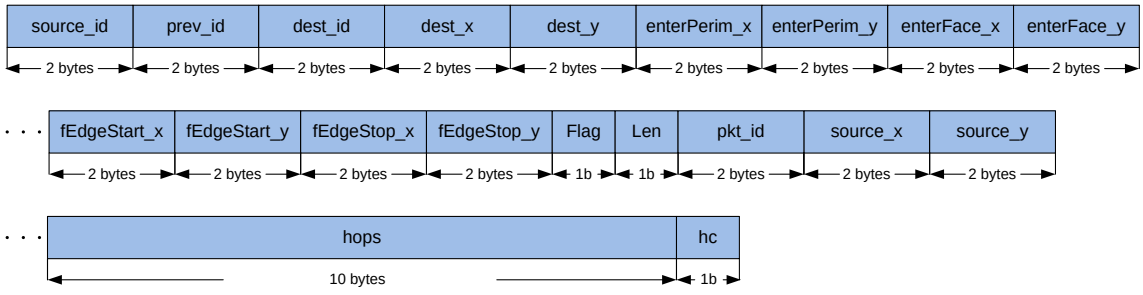


Figure 4-6: Trusted-GPSR - Network layer header

where:

- source_id: identity of the node created the packet
- prev_id: Identity of previous node
- dest_id: Identity of the final destination node
- dest_x: X coordinate of the final destination node
- dest_y: Y coordinate of the final destination node
- enterPerim_x: X coordinate when first time entering in perimeter mode
- enterPerim_y: Y coordinate when first time entering in perimeter mode
- enterFace_x: X coordinate when entering in a new face in the planar graph in perimeter mode
- enterFace_y: Y coordinate when entering in a new face in the planar graph in the perimeter mode
- fEdgeStart_x: X coordinate of the starting point of the current edge in perimeter mode
- fEdgeStart_y: Y coordinate of the starting point of the current edge in perimeter mode
- fEdgeStop_x: X coordinate of the end point of the current edge in perimeter mode
- fEdgeStop_y: Y coordinate of the end point of the current edge in perimeter mode
- Flag: Flag bits for mode of operation (greedy or perimeter) and content type
- Len: payload length
- pkt_id: packet sequence number
- source_x: X coordinate of the node created the packet
- source_y: Y coordinate of the node created the packet
- hops: 10 bytes space for storing identities that this node has traversed
- hc: hop count indicating the number of hops so far

3. *Reputation frame*

In order to include the opinions of other nodes about the trustworthiness of a certain node a reputation frame is broadcasted at periodic intervals with the structure and fields presented in Figure 14. The frame has a fixed length presenting information from seven neighbouring nodes. If more neighbours exist information will be split with neighbours after this value included in every second transmission. The number of entries for which valid information exists is included at the beginning of the reputation frame as well as an increasing counter field. Only first-hand evidence is included containing information about the number of interaction with this neighbouring node, its forwarding behaviour, the link quality between this node and the neighbour expressed in packet reception ratio and received signal strength, the remaining energy of the neighbour and its experience in reputation messages. After reception of a reputation frame a node will examine the identities advertised from this neighbour and will store information only for nodes that are also neighbours with the node receiving the frame (1 hop neighbours). This is due to the nature of GPSR in which data messages are forwarded to the neighbouring node which is closest to the destination and when trust is included to the trusted node that is closest to the destination.

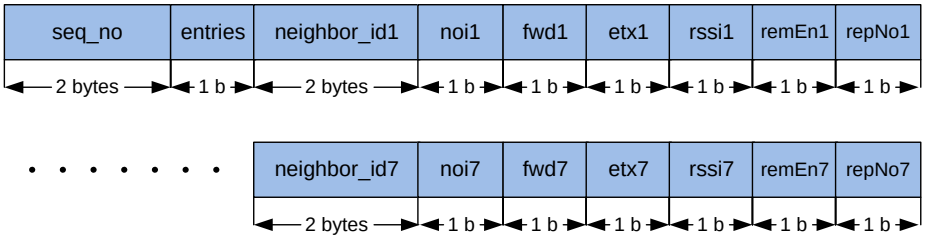


Figure 4-7: Trusted GPSR - Reputation frame

where:

- seq_no: increasing counter of reputation messages
- entries: number of valid entries for this reputation frame
- neighbor_id1: identity of the node for which transmitting node has first-hand evidence
- noi1: number of interactions between transmitting node and neighbour 1 expressed in the range 0 – 100.
- fwd1: forwarding behaviour of neighbour 1 measured from transmitting node expressed in the range 0 – 100.
- etx1: packet reception ratio measured from packets acknowledged from transmitting node when interacting with neighbour 1 expressed in the range 0 – 100.
- rssi1: received signal strength Indication for neighbour 1 as measured in transmitting node expressed in the range 0 – 100.
- remEn1: remaining energy of Neighbour 1 known in transmitting node expressed in the range 0 -100
- repNo1: number of reputation messages the transmitting node has received from neighbour 1 expressed in the range 0 - 100.

4.2.1.6 Storage data structures

Two main data structures are used for keeping information based on which routing decisions will be taken. The first one is an array of MAX_NEIGHBORS size

```
// neighbor information table
neighbor_t m_neighbors[MAX_NEIGHBORS];
```

where:

```
//define location_t
typedef struct location {
    uint16_t    x;
    uint16_t    y;
} location_t;

// format of keep alive messages that are broadcast periodically in order to inform neighbors
// about the node's existence

typedef nx_struct BeaconMsg
{
    // location of the node
    nx_location_t m_location;
    // id of the node
    nx_uint16_t m_id;
    // voltage reading
    nx_uint16_t m_voltage;
} BeaconMsg;

// information about a particular neighbor
typedef struct _neighbor
{
    BeaconMsg m_neighborInfo;

    // is the node neighbor in the planarized connectivity graph?
    nx_uint8_t m_isPlanar;
    // age of the entry
    nx_uint16_t m_age;

    nx_uint16_t noi;
    nx_uint16_t pktsFwd;
    nx_uint16_t forwFails;
    nx_uint16_t etx;
    nx_uint8_t  rssi;
    nx_uint16_t rssi_noi;
    nx_uint8_t rssi_t[5];
    nx_uint8_t table_pos;
    nx_uint16_t noRepMsg;
} neighbor_t;
```

Program listing 24: Trusted GPSR - Max neighbour's array

Fields of the m_neighbors array are updated in every beacon reception and can also be updated from all modules that use GPSRNeighborhood interface using accessor functions defined in it (Figure 10). Two fields deserve further analysis: Firstly forwFails field which indicates that transmitting node has received a MAC acknowledgement indicating successful delivery. This is achieved using TinyOS PacketAcknowledgements interface. In every unicast transmission requestAck command is used:

```
error = call PacketAcknowledgements.requestAck(msg);
```

After completing the transmission reception of acknowledgement in due time is examined using wasAcked command:

```
if(call PacketAcknowledgements.wasAcked(p_bufPtr))
```

If packet acknowledgement fails forwFails field is increased and the node reattempts to transmit the same frame up to a maximum value of MAC retries. In this way a node has a direct estimate of link quality which can be used alone or in conjunction with RSSI estimation to select a neighbouring node with good link quality.

Secondly the packet forwarding field (pktsFwd) which is crucial for the identification of undisrupted routing operation and detection of black-hole and grey-hole routing attacks. For this functionality the TinyOS AMSnoopingReceiverC.Receive interface is used in which the receive event is signalled whenever the packet layer receives an active message of the corresponding AM type regardless of destination address. Inside the event the prev_id field of the T-GPSR header is examined if equals the address of the node that received the frame and in case of equality there is evidence that the next node has performed the forwarding functionality appropriately. pktsFwd field of this neighbour is increased in m_neighbors table.

```
// Snooping function, used for the forwarding and integrity metric.
event message_t* Snoop.receive(message_t* p_bufPtr, void* p_payload, uint8_t p_len)
{
    message_t* new_msg = p_bufPtr;
    uint16_t id = 0;
    gpsr_header_t* header;
    header = (gpsr_header_t*) call Packet.getPayload(p_bufPtr, sizeof(gpsr_header_t));
    if (header->prev_id == TOS_NODE_ID) {
      fwd_metric++;
      id = call AMPacket.source(p_bufPtr);
      //      dbg("GPSR_RESULT", "The forwarding metric has the value of = %d.\n", fwd_metric);
      call GPSRNeighborhood.increasepktsFwd(id);
      Uprintf("SNOOP: Source ID = %d packets forwarded %d \n", id, call GPSRNeighborhood.getpktsFwd(id));
    }
    return new_msg;

}
```

Program listing 25: Trusted GPSR - TinyOS AMSnoopingReceiver interface

The second important data structure defined in *IndirectTrust* component is an array of MAX_IT_ENTRIES for storing reputation information

```
// indirect trust information table
it_t it_values[MAX_IT_ENTRIES];
```

where

```
typedef struct _it
{
    uint16_t target_node_id;
    uint16_t neighbor_id;
    uint8_t noi;
    uint8_t pktsfwd;
    uint8_t etx;
    uint8_t rssi;
    uint8_t remEnergy;
    uint16_t noITMsgRcv;
} it_t;
```

Program listing 26: Trusted GPSR - Indirect trust array

Each time a reputation frame is received the opinion of neighbouring node (identified by *neighbor_id*) for each one of its entries (*target_node_id*) is updated (or added if does not previously exist).

4.2.1.7 SPD Levels and Trust calculation

When programming a sensor node a parameter specifying the SPD level of the node is set according to the values of Table 4-2.

Table 4-2: Trusted GPSR - SPD Level and implemented algorithms in the trust module

SPD Level	Implemented Algorithms
1 (lowest)	-
2 (low)	Direct Trust [Algorithm A4 in D4.3]
3 (medium)	Weighted DT (Direct Trust) + ID (Indirect Trust) [A4 + A5 in D4.3]
4 (high)	Reputation based IDS algorithm, [A6 + A7 in D4.3]

Setting the SPD level affects the decision taken for the selection of the next node. Looking in more detail inside the code every time a data packet is received from a node using the standard *Receive* TinyOS interface the node puts the frame in the receive queue and calls the *Process* task where processing in the frame is performed to determine if final destination has been reached. In the case of final destination

arrival the node must pass the packet payload to the upper layer otherwise it has to select the next neighbour to forward this packet. *Forward* task is responsible to perform this functionality.

When SPD_LEVEL=0 next node will be selected purely geographically. The node will traverse all its neighbours stored in the m_neighbors table and will compare their distance to the final destination taken from the *dest_x*, *dest_y* parameters of the routing header of the packet received. The neighbour node whose distance to the final destination is shorter will be selected as the next node. The following code snippet from *Forward* task highlights this operation.

```
// Greedy forwarding: we seek a neighbor who is closer to the destination
    call GPSRNeighborhood.GetNeighbor(INIT);  //Neighborhood.GetNeighbor(INIT);
    while ( (neighbor = call GPSRNeighborhood.GetNeighbor(NEXT)) != NULL)
    {

        uint32_t neighborDist = call GPSR.Distance[1](ToLocation(neighbor->m_neighborInfo.m_location),
                                ToLocation(header->m_destLocation));

        if (SPD_LEVEL > 0)
        {
          .............
        }
        else
        {

            if (GET_MODE(header->m_controlFlag) == GREEDY &&
                neighborDist < call GPSR.Distance[1](myLocation,
                  ToLocation(header->m_destLocation))
                && neighborDist < candidateNodeDist)
            {
                candidateNode = neighbor->m_neighborInfo.m_id;
                candidateNodeDist = neighborDist;
                isNextNode = TRUE;
            }
            // If the packet was in perimeter mode and we find a closer neighbor,
            // we return to greedy mode immediately
            else if (GET_MODE(header->m_controlFlag) == PERIMETER)
            {
                if (neighborDist < call GPSR.Distance[1](ToLocation(header->m_enterPerimLocation),
                                    ToLocation(header->m_destLocation)) &&
                    neighborDist < candidateNodeDist)
                {
                    SET_MODE(header->m_controlFlag, GREEDY);

                    candidateNode = neighbor->m_neighborInfo.m_id;
                    candidateNodeDist = neighborDist;
                    header->prev_id = call AMPacket.source(msg);
                    isNextNode = TRUE;
                }
            }
        }
    }
```

Program listing 27: Trusted GPSR - Greedy forwarding

In this mode of operation the node is unable to avoid malicious nodes that disrupt routing like black-hole and grey-hole attackers.

When SPD_LEVEL=1 next node is selected using a weighted sum of distance and direct trust. Firstly the neighbours table is traversed to find the neighbour whose distance is closer to the destination (minimum distance). In the next iteration of neighbours table a distance metric is calculated as the fraction of minimum distance to current node distance:

$$D^{A,B} = \frac{d^{A,B}_{\min}}{d^{(A,B)}}$$

with D taking values in the range [0, 1].

Direct trust is calculated using a weighted sum of packet forwarding, link quality and remaining energy. Packet forwarding value has been stored in my neighbours table using the *AMSnoopingReceiverC.Receive* interface as analysed in the previous section. Direct trust is calculated as:

$$DT = W_f \cdot F + W_{etx} \cdot ETX + W_{rssi} \cdot RSSI + W_E \cdot E$$

where:

- F is packet forwarding
- ETX is packet delivery ratio
- $RSSI$ is the received signal strength indicator
- E is the remaining energy of the neighbour

and W_f , W_{etx} , W_{rssi} , W_E weighting factors with $W_f + W_{etx} + W_{rssi} + W_E = 1$

Finally, next node is selected by calculating neighbour value $NV^{A,B}$ using the formula:

$$NV^{A,B} = W_t \cdot DT + W_d \cdot D^{A,B} \quad (1)$$

with $W_t + W_d = 1$

The code snippet below represents the actions described above in source code format

```
if (SPD_LEVEL > 0)
{
    distanceMetric = (float) (minDistance) / (float) (neighborDist);

    if (neighbor->noi == 0)
    {
        totalTrustMetric = w_trust * DT_INIT;
    } else {
        if (SPD_LEVEL > 1) //total trust is the weighted sum of Direct & Indirect Trust
        {
            ------------------
        }
        else //total trust is only Direct Trust
        {
            energy100 = EnergyPerCent(neighbor->m_neighborInfo.m_voltage);
            pktFwd = ((w_frw * (float)(neighbor->pktsFwd))/(float)(neighbor->noi));
            totalTrustMetric =   (pktFwd +
                                  ((w_etx * neighbor->etx)/100) +
                                  ((w_rsi * neighbor->rssi)/100) +
                                  (((uint32_t)((uint32_t)w_nrg * (uint32_t)energy100))/100));
        }
    }
    routingFunction = (uint32_t)((w_trust *totalTrustMetric) + (w_dist * (distanceMetric*100)));
    Uprintf("ROUTING FUNCTION %d \n", routingFunction);
    if (GET_MODE(header->m_controlFlag) == GREEDY && routingFunction > routingFunction_temp)
    {
        candidateNode = neighbor->m_neighborInfo.m_id;
        candidateNodeDist = neighborDist;
        routingFunction_temp = routingFunction;
        isNextNode = TRUE;
    }
}
```

Program listing 28: Trusted GPSR - Next node selection mechanism for SPD level = 1

Next node selection is performed according to the *routingFunction* value which is equivalent to formula (1) above. Using this SPD_LEVEL if the node closer to the destination is an attacker will initially have the greatest value of routingFunction due to distanceMetric value which is equal to 1, but as packet forwarding fails the total value of *routingFunction* variable will become less compared to more distant nodes. At this point a trusted node will be selected as next hop, bypassing the routing attack.

In the case of SPD_LEVEL=2 the opinion of other nodes that interact with my neighbours will be taken into account. This is helpful in cases when my previous experience with the node under examination is limited. In this case Total Trust will be calculated as the weighting sum of Direct and Indirect Trust:

$$TT = W_{DT} \cdot DT + W_{IT} \cdot IT$$

and next node is selected by calculating neighbour value using the formula:

$$NV^{A,B} = W_t \cdot TT + W_d \cdot D^{A,B}$$

For the Indirect Trust calculation the table it_values will be traversed to extract the reputation values that neighbours have transmitted for the node under examination.

```
if (SPD_LEVEL > 1) //total trust is the weighted sum of Direct & Indirect Trust
{
    energy100 = EnergyPerCent(neighbor->m_neighborInfo.m_voltage);
    pktFwd = ((w_trust * w_frw * (float)(neighbor->pktsFwd))/(float)(neighbor->noi));
    directTrustMetric =   (pktFwd +
                           ((w_trust * w_etx * neighbor->etx)/100) +
                           ((w_trust * w_rsi * neighbor->rssi)/100) +
                           (((uint32_t)((uint32_t)w_trust * (uint32_t)w_nrg * (uint32_t)energy100))/100));

    indirectTrustMetric = call IndirectTrust.calculateIndirectTrust(neighbor->m_neighborInfo.m_id,
                                                                    w_frw, w_etx, w_rsi, w_nrg, w_rep);
    totalTrustMetric = w_direct*directTrustMetric + w_indirect*indirectTrustMetric;
}
```

Program listing 29: Trusted GPSR – Next node selection mechanism for SPD level=2, 3

Where *calculateIndirectTrust* is responsible for *it_values* table traversal, the finding all the entries for the node under examination and the calculation of the weighted sum using the weights passed as parameters to the function.

```
//calculate indirect trust by extracting all values for node_id fron indirect trust table
command uint32_t IndirectTrust.calculateIndirectTrust(uint16_t node_id,
                                                      uint8_t w_frw, uint8_t w_etx, uint8_t w_rsi, uint8_t w_nrg, uint8_t w_rep)
    {
        uint8_t i;
        uint32_t indirectTrustValue;
        uint8_t samples;

        i=0;
        indirectTrustValue = 0;
        samples = 0;
        while (i < MAX_IT_ENTRIES)
        {
            if (it_values[i].target_node_id == 0xffff)
                break;
            if (it_values[i].target_node_id == node_id)
            {
                if(it_values[i].noi == 0)
                {
                    indirectTrustValue = IT_INIT;
                }
                else {
                    indirectTrustValue += (uint32_t)(w_frw*it_values[i].pktsfwd) +
                                          (uint32_t)(w_etx*it_values[i].etx) +
                                          (uint32_t)(w_rsi*it_values[i].rssi) +
                                          (uint32_t)(w_nrg*it_values[i].remEnergy) +
                                          (uint32_t)(w_rep*it_values[i].noITMsgRcv);
                    samples++;
                }
            }
            i++;
        }
        if (samples > 1) {
            indirectTrustValue = indirectTrustValue/samples;
        }
        if (samples > 0) {
            indirectTrustValue = indirectTrustValue/100;
            Uprintf("IT from %d samples:   val %d \n", samples, indirectTrustValue);
        }
        return indirectTrustValue;
    }
```

Program listing 30: Trusted GPSR - Indirect trust calculation

Finally when SPD_LEVEL=4 only reputation messages that don't deviate from a beta distribution function are accepted as valid reputation messages. This way every time neighbours advertise false reputation messages either by praising or accusing the behaviour of the node under examination they will be isolated from participating in indirect trust calculation and the network will be robust to bad-mouthing attacks.

4.2.1.8 Fabrication of Routing Attacks

When programming a node the ATTACKER command line parameter is specified indicating the behaviour of the node. ATTACKER=0 indicates a normal non-malicious node. ATTACKER=1 indicates black-hole node, while ATTACKER=2 indicates a grey-hole node. The code snippet (from GPSR component which is responsible for packet forwarding when destination hasn't yet been reached) presented below presents the section that suspends the proper packet forwarding operation, withholding the packets every time in the case of black-hole attacker and according to a random pattern in the case of grey-hole attacker.

```
// Signalling reception
if ((pkt_destination.x == mylocation.x) && (pkt_destination.y == mylocation.y))       //If pkt arrived
{
    initPosition.x=header->m_iniLocation.x;
    initPosition.y=header->m_iniLocation.y;
    if (GET_TYPE(header->m_controlFlag) == TYPE_GPSR)
      signal GPSR.receiveMsg[handlerID](data, header->m_dataLen, initPosition, header->ini_id);
    call MessagePool.put(msg);
}
else    //forward it !!!!
{
    // If the packet should be forwarded, we attach that to the sending queue
    if (ATTACKER == BLACK_HOLE_ATTACKER) {
       call MessagePool.put(msg);
       return;
    }
    else if (ATTACKER == GRAY_HOLE_ATTACKER) {
       if (!((call Random.rand16()) & 0x01)) {
          call MessagePool.put(msg);
          return;
       }
    }

        dbg("GPSR_RESULT", "Forwarding message in inter-routing, no local processing needed.\n");
        atomic
        {
            if ( call QueueSend.enqueue(msg) == FAIL)
            {
                dbg("GPSR_RESULT", "GPSR: Receive queue is full! Message is dropped!\n");
                call MessagePool.put(msg);
                return;
            }
        }

    // and forward that
    post Forward();
}
```

Program listing 31: Trusted GPSR - Routing attacks

Programming a node with parameter ATTACKER=3 leads to a node that praise all its neighbours giving the maximum values for all the parameters in the reputation frame while parameter ATTACK=4 leads to a node that transmits false accusations for all its neighbours by minimizing the values for all the parameters in the reputation frame. These represent two kinds of bad-mouthing attacks that the reputation scheme should be able to counteract.

4.2.1.9 Trusted Routing Testing and Debugging Tools

T-GPSR TinyOS source code was tested using different topologies of sensor nodes. Crossbow IRIS mote [19] was the platform used in the experiments. The mote is able to be customized for a particular application with the use of the appropriate sensor board (i.e.: temperature, barometric, pressure, acceleration, acoustic, magnetic, etc.) but in the case of this section routing behaviour was the first priority of experimentation. A first tool where network behaviour can examined in TinyOS environment is the simulator provided, TOSSIM (TinyOS SIMulator). In the experiments of T-GPSR both TOSSM simulations and real hardware nodes were used. In order to be able to examine multi-hop routing in a small area using real hardware a controlled topology was used where programmatically only certain nodes from the sum of all nodes from which beacon messages are available are considered as neighbours. A short description of the tools used that helped in the verification of T-GPSR correct behaviour follows.

1. *TOSSIM simulation*

   During T-GPSR development extended simulations conducted in TOSSIM in order to validate the reliability and successful implementation of the trust-aware routing solution in TinyOS. TOSSIM provides flexibility during the TinyOS application development and debugging, as TinyOS code can be compiled into the TOSSIM framework on a PC, instead of compiling it on a sensor mote. Thus, algorithm testing and debugging is easily conducted in a controlled and repeatable way, isolated and relieved from external factors, which may affect the smooth algorithm operation. TOSSIM simulates entire TinyOS applications by replacing components with simulation implementations in several levels. For example, TOSSIM offers packet-level or low-level communication simulation by replacing a packet-level communication component in the former case, or replacing a low-level radio chip component in the latter case.

TOSSIM network simulation is based on the network topology provided. Moreover, TOSSIM radio simulation can be configured through a data set providing the propagation signal strengths between any possibly communicating nodes, the noise floor and receiver sensitivity. Thus, TOSSIM does not simulate only specific radio propagation models, but it can simulate a wide range of radios and behaviours through a few low-level primitives. In addition, TOSSIM also simulates the RF noise and, both self- and external, interference the WSN experiences implementing the Closest Pattern Matching (CPM) algorithm. A noise trace is provided as input to CPM, which then extracts a statistical model. This model can capture bursts of interference and other correlated phenomena, such that it greatly improves the quality of the RF simulation.

TOSSIM is a discrete event simulator. Events are kept in an event queue, sorted by time, which are pulled by TOSSIM and executed. Such simulation events can be hardware interrupts, high-level system events or even tasks, so that task posting results in the task running in the near future.

TOSSIM supports two programming interfaces, namely Python and C++.During T-GPSR testing Python was used.

The scope of the validation of the TinyOS source code is summarised in the following:

- Verification of the routing protocol: The routing solution should be capable of selecting the optimal path from the source to the destination node.
- Verification of the trust-aware solution: The trust-aware routing solution should be able to detect and avoid any malicious nodes on the shortest path by traversing longer, yet trusted, paths.
- Verification of the routing metrics enabled: T-GPSR implementation offers the ability to include energy awareness and link quality observations in the routing module. Correct behaviour of the routing module should be examined when these parameters are taken into consideration.

Compiling a TinyOS application with the sim parameter ($ make iris sim) will build all the libraries needed by TOSSIM as well as all the Python interfaces that interact with the library. A Python script used for TOSSIM simulations has the format of the program listing below. The details of using the simulator can be found in [20]. Network topology is loaded from a file which contains lines of the format:

```
gain src dst g
gain  1   2   -54.0
```

which is translated as when node 1 transmits node 2 hears it at -54.0 dBm.

Using dbg inside TinyOS application source code enables the programmer to print debug statements in the simulation environment, including inspection of current variables values. In dbg statements an output channel has been defined as the first argument before the message that will be presented in the output:

dbg("NeighborManagementC", "Node %d registered as a neighbour.\n", pBeaconMsg->m_id);

dbg statements proved to be a valuable tool during T-GPSR debugging providing useful information about various variables values and revealing internal nodes' behaviour.

```
from TOSSIM import *
import sys

t = Tossim([])
r = t.radio()
f = open("topo8.txt", "r")

lines = f.readlines()
for line in lines:
 s = line.split()
 if (len(s) > 0):
  if (s[0] == "gain"):
   print " ", s[1], " ", s[2], " ", s[3];
   r.add(int(s[1]), int(s[2]), float(s[3]))

t.addChannel("NeighborList", sys.stdout);
t.addChannel("NeighborManagementC", sys.stdout);
t.addChannel("GPSRC", sys.stdout);
t.addChannel("GPSR_RESULT", sys.stdout);
t.addChannel("APPLICATIONC", sys.stdout);

noise = open("meyer-heavy.txt", "r")
lines = noise.readlines()
for line in lines:
 str = line.strip()
 if (str != ""):
  val = int(str)
  for i in range(0, 10):
    t.getNode(i).addNoiseTraceReading(val)

for i in range(0, 10):
   print "Creating noise model for ",i;
   t.getNode(i).createNoiseModel()

t.getNode(0).bootAtTime(100001);
t.getNode(1).bootAtTime(5000000100);
t.getNode(2).bootAtTime(10000000000);
t.getNode(3).bootAtTime(15000000000);
t.getNode(4).bootAtTime(20000000100);
t.getNode(5).bootAtTime(22000000100);
t.getNode(6).bootAtTime(24000000100);
t.getNode(7).bootAtTime(26000000100);

t.runNextEvent();
time = t.time()

while (time + 400000000000 > t.time()):
 t.runNextEvent()
```

Program listing 32: Trusted GPSR - Python script used in TOSSIM simulations (simtest.py)

Redirecting python script to a text file

```
$python simtest.py > simresults.txt
```

gives the programmer the ability to inspect network internals, nodes' interactions and routing correctness over a time interval of interest. Part of the output dbg statements after running the python script above in TOSSIM simulator are presented in Figure 4-8.

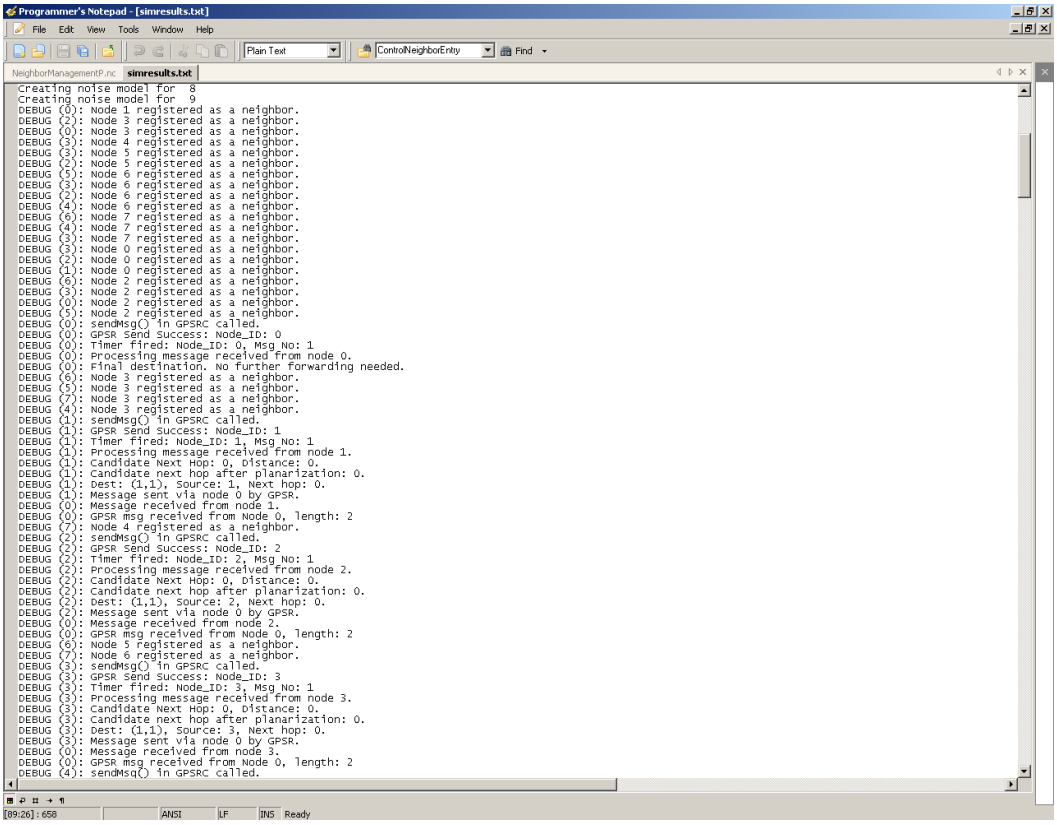


Figure 4-8: Trusted GPRS - Debug statements in the output file of a TOSSIM simulation.

2. Daintree SNA Packet Sniffer

Daintree Networks SNA kit [21] is an IEEE 802.15.4 packet sniffer that includes the Sensor Network Analyser (SNA) which is a packet filtering software application and the ATMEL-based capturing device in hardware (Atmel STK 541) which is connected through USB, as depicted in Figure 4-9.

The SNA combines a powerful protocol analyser with network visualization, measurements and diagnostics for wireless embedded networks. It provides automatic display of network formation, topology changes, and router and coordinator state changes allowing rapid detection of incorrect network behaviour and identification of device or network failures. Furthermore, this tool provides a powerful protocol decoder that allows drilling down to the packet, field and byte level, as well as customizing options including filtering, labelling and color-coding for locating packets of interest. The program was used to capture all the transmitted packets (beacon, data and reputation packets, MAC acknowledgements) and their exact content was examined at byte level to validate their correctness. Moreover multi-hop routing operation and paths followed can easily be monitored with Daintree SNA. Additional parameters of monitoring provided by the tool include observation of the timing behaviour of the network, packet forwarding delay and end-to-end latency. A snapshot of the visualization window with 9 nodes running T-GPSR is depicted in Figure 4-10.

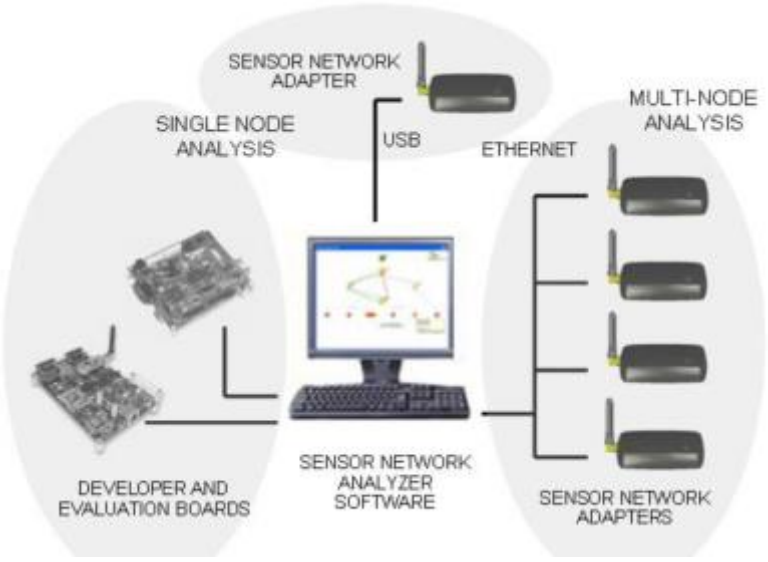


Figure 4-9: Trusted GPSR - Daintree sensor network analyser

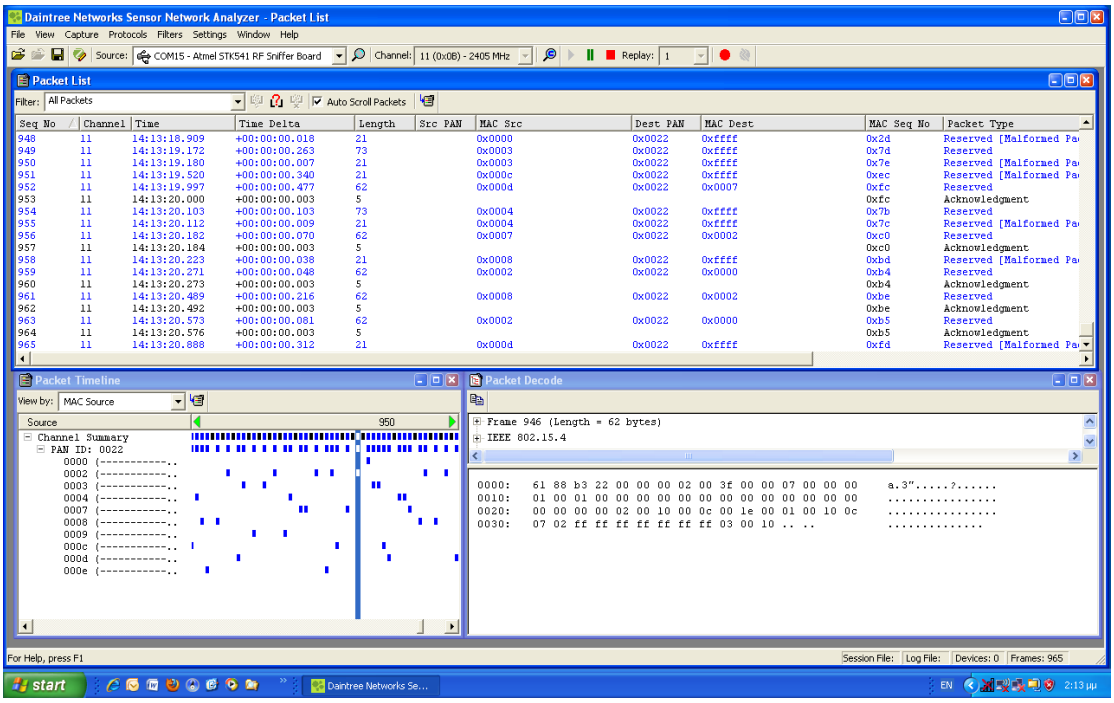


Figure 4-10: Trusted GPSR - frames capture from Daintree SNA

3. *Serial Communication*

Serial communication between IRIS mote and PC is feasible using a 3.3V RS232 level shifter. From the software point of view, *Uprintf* (defined in Uprintf.h) is a utility function that can be used to print statements to a PC serial port terminal application in a similar way in which the well-known C printf function prints statements in the PC standard output. *Uprintf* utilizes the UART0 port of the Atmel ATmega1281 microcontroller to communicate with a terminal application set to 57600, 8, N, 1 parameters. This function is based on the source code of an Atmel AVR utility library which was adopted and modified to be included into TinyOS to overcome difficulties encountered with the original *printfUART* function of TinyOS source code tree. *Uprintf* was a valuable tool for monitoring execution flow and print variables from inside the source code of the

microcontroller besides the burden of extra instructions added and the increased utilization of the limited amount of RAM memory of the ATmega1281 microcontroller. A snapshot of the terminal presenting values under examination in T-GPSR is depicted in Figure 4-11.

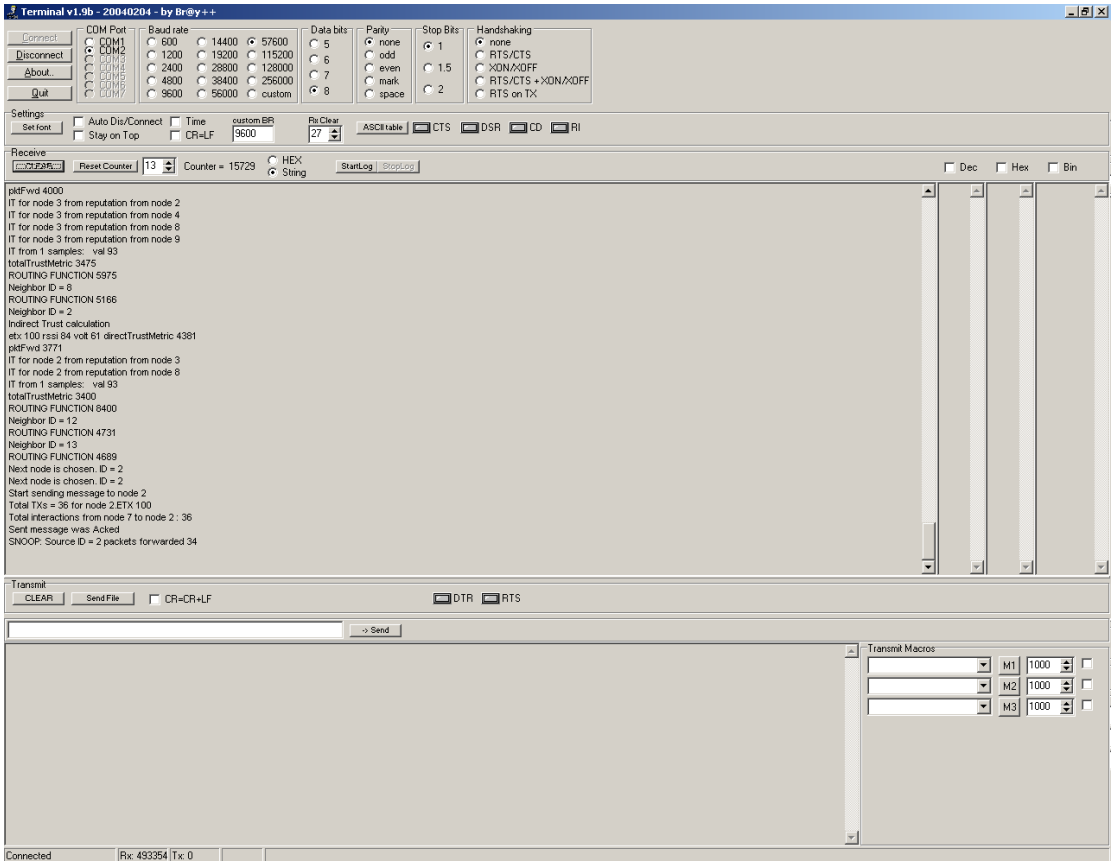


Figure 4-11: Trusted GPSR - Sensor node monitoring using terminal application and Uprintf

4.2.2 Intrusion Detection in Wireless Sensor Networks

4.2.2.1 Brief description

The proposed Intrusion Detection prototype consists of a number of wireless sensors running on Zolertia Z1 mote (which is briefly described in section 5.1.2 of D4.3 [22]), using a 802.15.4 radio interfaces. For our prototype, we have selected one of the available open source operating systems for wireless sensor networks, namely Contiki-OS 2.6 (also briefly described in the same section mentioned before).

The algorithms used for the Bayesian reputation system should be easily ported to other hardware and software platforms.

The Bayesian reputation Intrusion Detection strategies, and the proposed algorithms, have been described in Section 4.3.3 of D4.3 [23]. Here, the most important corresponding parts of algorithm (C code) for the preliminary prototype are provided.

4.2.2.2 Algorithms implementation

Given the hardware platform we have used to implement the prototype on it does not have a floating point unit, floating point operations are not available by default. Even though software implementation can be linked in, floating operations are very slow. For this reason, it has been decided to use fixed point arithmetic using long (32 bit) integers. This allows for 2 decimal places (which is good enough for the algorithms used, as it does not make them unstable due to lack of precision) and big enough values for

the α_j and β_j parameters, as these can grow very large for very active nodes. In order to avoid mistakes using fixed point arithmetic, several C pre-processor macros have been defined to hide all the details. These are the FLOATVAL_XXX macros that appear in the code below.

The algorithm uses three different sets of values for each node it knows about:

- First-hand information: which consists of the Id of the node in question (we assume it is an integer value in the demonstrator), plus the α and β parameters of the Bayesian distribution for first-hand information.
- Reputation information: which consists of the Id of the node in question (we assume it is an integer value in the demonstrator), plus the α and β parameters of the Bayesian distribution for reputation information.
- Trust information: which consists of the Id of the node in question (we assume it is an integer value in the demonstrator), plus the α and β parameters of the Bayesian distribution for trust information.

The values for the nodes are kept in tables, addressed by the node ID value. As the hardware for the demonstrator has rather tight memory constraints, it has been decided to use size-bounded tables for those values. The maximum size of those tables is configurable via a constant in the source code, so we can use bigger tables when running on more powerful hardware.

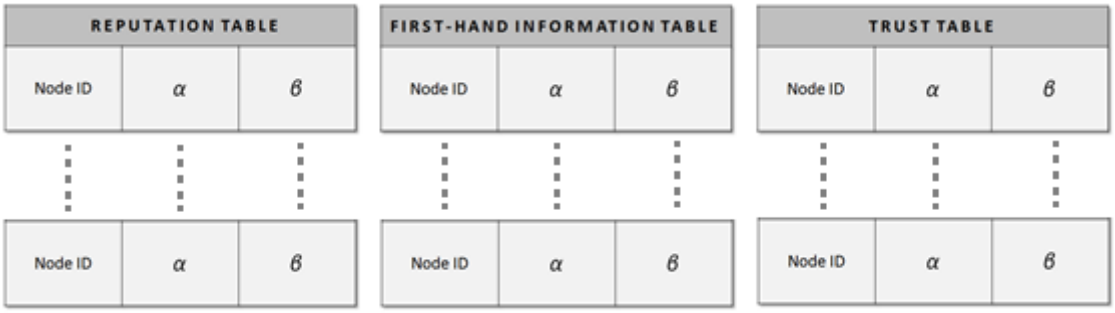


Figure 4-12: IDS - schematic

Contiki-OS already implements a linked list library that provides a set of functions for manipulating size-bounded linked lists in a memory efficient way. As Contiki-OS source code is released under a 3-clause BSD-style license, the code from the project can be used freely in both commercial and non-commercial systems as long as the copyright header in the source code files is retained. This will allow us to port the algorithm to other platforms with minimal effort by simply using the same linked list library.

Using that library, we will keep other nodes' first-hand information, reputation and trust tables. As the lists have a maximum fixed size, we will manage the entries in the tables using a simplified LRU (Least Recently Used) strategy. Also, to simplify the management of the values and keep memory used low, instead of using separate tables for each of the sets, we conflate all the values in a single table.

Also, since we need to decay first-hand information, reputation and trust on inactivity periods (as explained in D4.3), we need to also keep the time of the last interaction with a given node. In this case we do not need very precise timing for the inactivity periods (they are usually specified in the range of several seconds or even minutes), so we use a value measured in seconds.

The first thing we need to do is initialize some global constants used by the algorithms, like the ID of the node itself (to ignore second hand information data that is supposed to be sent by the node itself), the various fading and merging factors, and the decision thresholds. These values can be changed at runtime using several available functions. The values chosen initially (shown in the following piece of code) come from [24].

```
Void brt_init (nodeId_t ownId, interval_t inactivityPeriod)
{
        brt_myId = ownId;
        brt_reputationFading = brt_floatValMakeConst(0.9);
        brt_trustFading = brt_floatValMakeConst(0.9);
        brt_reputationMerging = brt_floatValMakeConst(0.1);
        brt_reputationThreshold = brt_floatValMakeConst(0.5);
        brt_trustThreshold = brt_floatValMakeConst(0.85);
        brt_devTestThreshold = brt_floatValMakeConst(0.5);
        brt_inactivityPeriod = inactivityPeriod;
        memb_init(&brt_nodeTableMem);
        list_init(brt_nodeTable);
}
```

Program listing 33: IDS - initialization of the global constants

Once the reputation system is initialized, each time a local (first hand) interaction occurs with a given node (or set of nodes), we call the brt_updateEntries() function that implements the algorithm 7 described in section 4.3.3 “Obtaining α and β for the reputation table and γ and δ for confidence table”.

We check whether this particular update is for a first-hand interaction, and if so, we update the first-hand information table and the reputation table for that node. And then test the reputation for the node. In case that the reputation is outside of the threshold, we store the node’s ID in the misbehaved array the caller has passed.

```
uint16_t brt_updateEntries(const nodeUpdateEntries_t *nodeUpdates,
                           nodeId_t *misbehavedNodes, uint16_t entriesMax)
{
  uint16_t i, misbehaved, trustworthy;
  floatVal_t deviationTest;
  nodeExchangedData_t nodeData;
  nodeEntry_t *node, reportingNode;
  misbehaved = 0;
  if (nodeUpdates->updateType == brt_firsthand)
  {
      for (i = 0; i < nodeUpdates->count; i++)
      {
        nodeData = nodeUpdates->nodeData[i];
        brt_updateFirsthandDataTable(nodeData.nodeId, nodeData.alpha, nodeData.beta);
        node = brt_updateReputationTable(nodeData.nodeId, nodeData.alpha,
                                         nodeData.beta, brt_firsthand);

        if (!brt_testReputationThreshold(node))
        {
            if (misbehaved < entriesMax)
            {
                  misbehavedNodes[misbehaved] = nodeData.nodeId;
                  misbehaved++;
            }
        }
      }
   }
}
```

Program listing 34: IDS - obtaining α and β for the reputation table and γ and δ for confidence table

The reason for doing so is that instead of triggering the alert process from inside the reputation algorithm (which may know nothing about the framework it is running under), we simply hand the information back to the caller which can better decide how to handle the alerting process.

If on the other hand the updates are from second hand information [25], we first check that the updates are not supposed to be from ourselves. This can happen if either our first hand data broadcasts are re-broadcasted in our neighbourhood (to reach our two-hop neigh boughs), or a malicious node is trying to impersonate us. In both cases, we can simply ignore the updates.

```
else if (nodeUpdates->updateType == brt_reported)
{
     if (nodeUpdates->reporterId == brt_myId)
     {
          return 0;
     }
}
```

Program listing 35: IDS - updates from second hand information

In other case, we iterate over the nodes' values that are contained in the updates. For each node's values we need to perform the deviation test for the reporting node (and update its trust accordingly), to see if we will integrate the second hand information with our local values or not. If we do, we update the reputation for the reported node and check whether the node is misbehaved or not (to report it back to the caller):

```
reportingNode = brt_nodeLookup(brt_nodeTable, nodeUpdates->reporterId);
if (reportingNode == NULL)
{
     reportingNode = brt_nodeAdd(brt_nodeTable, &brt_nodeTableMem,
                                 nodeUpdates->reporterId, floatValOne,
                                 floatValOne, floatValOne, floatValOne,
                                 floatValOne, floatValOne);
}
for (i = 0; i < nodeUpdates->count; i++)
{
     nodeData = nodeUpdates->nodeData[i];
     deviationTest = brt_deviationTest(reportingNode, nodeData.alpha, nodeData.beta);
     if (deviationTest)
     {
          brt_updateTrustTable(reportingNode, floatValOne, floatValZero);
     } else {
          brt_updateTrustTable(reportingNode, floatValZero, floatValOne);
     }

     trustworthy = brt_testTrustThreshold(reportingNode);
     if ((!deviationTest) || trustworthy)
     {
          node = brt_updateReputationTable(nodeData.nodeId, nodeData.alpha,
                                           nodeData.beta, brt_reported);
          if (!brt_testReputationThreshold(node))
          {
               if (misbehaved < entriesMax)
               {
                    misbehavedNodes[misbehaved] = nodeData.nodeId;
                    misbehaved++;
               }
          }
     }
}
} /* END: second hand information updates */
/* Tell the caller how many misbehaved nodes we are returning back */
return misbehaved;
```

Program listing 36: IDS - node lookup

The code to update the first-hand information table, the reputation table and the trust (confidence) table are shown below:

```
nodeEntry_t* brt_updateFirsthandDataTable(nodeId_t nodeId, floatVal_t alphaFirst, floatVal_t
betaFirst)
{
      nodeEntry_t *node;
      node = brt_nodeLookup(brt_nodeTable, nodeId);
      if (node == NULL)
      {
          node = brt_nodeAdd(brt_nodeTable, &brt_nodeTableMem, nodeId, floatValOne,
                              floatValOne, floatValOne, floatValOne, floatValOne, floatValOne);
      }
      node->data.alphaFirst = FLOATVAL_ADD(FLOATVAL_MUL(brt_reputationFading,
                                          node->data.alphaFirst), alphaFirst);
      node->data.betaFirst = FLOATVAL_ADD(FLOATVAL_MUL(brt_reputationFading,
                                          node->data.betaFirst), betaFirst);
      SET_CURRENT_TIME_SEC(node->lastUpdate);
      brt_lruNode(brt_nodeTable, node);
      return node;
}
```

Program listing 37: IDS - updating first hand data table

```
nodeEntry_t* brt_updateReputationTable(nodeId_t nodeId, floatVal_t alphaRep, floatVal_t
betaRep, updateType_t update)
{
      nodeEntry_t *node;
      node = brt_nodeLookup(brt_nodeTable, nodeId);
      if (node == NULL)
      {
            node = brt_nodeAdd(brt_nodeTable, &brt_nodeTableMem, nodeId,
                        floatValOne, floatValOne, floatValOne,
                        floatValOne, floatValOne, floatValOne);
      }

      switch (update)
      {
            case brt_firsthand:
                  node->data.alphaRep = FLOATVAL_ADD(FLOATVAL_MUL(brt_reputationFading,
                                                node->data.alphaRep), alphaRep);
                  node->data.betaRep = FLOATVAL_ADD(FLOATVAL_MUL(brt_reputationFading,
                                                node->data.betaRep), betaRep);
                  break;
            case brt_reported:
                  node->data.alphaRep = FLOATVAL_ADD(node->data.alphaRep, FLOAT
                                          VAL_MUL(brt_reputationMerging, alphaRep));
                  node->data.betaRep = FLOATVAL_ADD(node->data.betaRep, FLOAT
                                          VAL_MUL(brt_reputationMerging, betaRep));
                  break;
            default:
                  break;
      }
      SET_CURRENT_TIME_SEC(node->lastUpdate);
      brt_lruNode(brt_nodeTable, node);
      return node;
}
```

Program listing 38: IDS - updating reputation table

```
nodeEntry_t* brt_updateTrustTable(nodeEntry_t *node, floatVal_t gamma, floatVal_t delta)
{
    node->data.gamma = FLOATVAL_ADD(FLOATVAL_MUL(brt_trustFading, node->data.gamma), gamma);
    node->data.delta = FLOATVAL_ADD(FLOATVAL_MUL(brt_trustFading, node->data.delta), delta);
    SET_CURRENT_TIME_SEC(node->lastUpdate);
    brt_lruNode(brt_nodeTable, node);
    return node;
}
```

Program listing 39: IDS - updating trust table

The code for the deviation test is given as follows:

```
    uint16_t brt_deviationTest(nodeEntry_t *reportingNode, floatVal_t alphaReported,
    floatVal_t betaReported)
    {

        floatVal_t reportingNodeExpectation;
        floatVal_t secondhandExpectation;
        uint16_t ret;
        reportingNodeExpectation = FLOATVAL_DIV(reportingNode->data.alphaRep,
               FLOATVAL_ADD(reportingNode->data.alphaRep, reportingNode->data.betaRep));
        secondhandExpectation = FLOATVAL_DIV(alphaReported, FLOATVAL_ADD(alphaReported,
               betaReported));
        if (reportingNodeExpectation < secondhandExpectation)
        {
               ret = (FLOATVAL_SUB(secondhandExpectation, reportingNodeExpectation) >=
                       brt_devTestThreshold);
        }
        else
        {
               ret = (FLOATVAL_SUB(reportingNodeExpectation, secondhandExpectation) >=
                      brt_devTestThreshold);
        }
        return ret;
    }
```

Program listing 40: IDS - deviation test

As we have explained before we need to decay first-hand information, reputation and trust on inactivity periods. That is why on every update of the first-hand information table, reputation table or trust table we update the lastUpdate value of the node. We set a periodic inactivity timer, and when the timer expires we call the brt_updateOnInactivityTimer() function:

```
void brt_updateOnInactivityTimer(void)
{
    nodeEntry_t *entry;
    interval_t now, expiry;
    SET_CURRENT_TIME_SEC(now);
    if (now < brt_inactivityPeriod) {
     /* Prevent integer overflowing at system start */
     expiry = 0;
    } else {
     expiry = now - brt_inactivityPeriod;
    }
    for (entry = list_head(brt_nodeTable); entry != NULL; entry = entry->next)
    {
     if (entry->lastUpdate < expiry)
     {
        entry->data.alphaFirst = FLOATVAL_MUL(brt_reputationFading, entry->data.alphaFirst);
        entry->data.betaFirst = FLOATVAL_MUL(brt_reputationFading, entry->data.betaFirst);
        entry->data.alphaRep = FLOATVAL_MUL(brt_reputationFading,   entry->data.alphaRep);
        entry->data.betaRep = FLOATVAL_MUL(brt_reputationFading, entry->data.betaRep);
        entry->data.gamma = FLOATVAL_MUL(brt_trustFading, entry->data.gamma);
        entry->data.delta = FLOATVAL_MUL(brt_trustFading, entry->data.delta);
     }
    }
}
```

Program listing 41: IDS - update on inactivity timer

There are also several helper functions to get the reputation or trust of a given node (expressed in the sense defined by the Bayesian underlying approach), and the contents of the whole first-hand information table. As stated in D4.3, we publish the first-hand information when at least one of the updated nodes is considered misbehaved.

5 Trusted and dependable connectivity

5.1 Link layer security

Network Access Control (NAC) refers to methods used to authorize or deny network communications to particular systems or users. In other words, our system before transmitting data through the network has to ensure that new nodes can join to and leave from the network in a secure way.

As described in Deliverable D4.3 one of the protocols to manage NAC is EAP which uses certificates to ensure the security and confiability of the system.

Firstly we are going to explain how to create a root CA for the whole WSN and after that we will analyse the algorithms and results of the proposed solution.

5.1.1 Creating a root CA for the whole WSN (nSHIELD)

To perform the creation, request, issuing and signing certificates will be performed with openSSL library installed on a Linux (Ubuntu) operating system.

First of all we have to establish the OpenSSL Environment for creating and issuing certificates.

```
# Set up the relevant directories
mkdir -p ~/etc
mkdir -p ~/etc/ssl
mkdir -p ~/etc/ssl/private
chmod og-rwx ~/etc/ssl/private
mkdir -p ~/etc/ssl/certs
mkdir -p ~/etc/ssl/crl
mkdir -p ~/etc/ssl/newcerts
mkdir -p ~/tmp

# Set up the location of your OpenSSL configuration file
export OPENSSL_CONF="/etc/ssl/openssl.cnf"

# Add the location of your OpenSSL configuration file to your .bashrc
echo "/etc/ssl/openssl.cnf" >> ~/.bashrc
echo "export OPENSSL_CONF=\"/etc/ssl/openssl.cnf\"" >> ~/.bashrc

# Create the random file
openssl rand -out ~/etc/ssl/private/.rand 1024
chmod og-rwx ~/etc/ssl/private/.rand
```

Program listing 42: Link layer security - establishing OpenSSL environment

After this set of commands, we have to modify the openSSL config file (/etc/ssl/openssl.cnf) and change the default directory from "./demoCA" to "~/etc/ssl".

1. **Generating the private (and public) key**

```
# Create an RSA private key
openssl genrsa -des3 -out /etc/ssl/private/nshield.key 4096
openssl ec -des3 -out /etc/ssl/private/nshield.key 4096
chmod og-rwx /etc/ssl/private/nshield.key
```

Program listing 43: Link layer security - generating private key

2. **Fill in the certificate request**
   A certificate request is a combination of your private key and your self-information in reality, in order for the CA to verify its correctness and sign on it later. For this reason, you will be asked

several questions relevant to this key, including your country, city, organization, department, the certificate name, contact e-mail, and your applying valid period. Fill in them one by one.

If you want to use your root CA as the same server certificate for your server, fill in your server's full qualified domain name (FQDN, i.e. www.newshield.eu) as the certificate's common name here.

```
# Fill in the certificate request
openssl req -new -key /etc/ssl/private/nshield.key -out /tmp/nshield.req
```

Program listing 44: Link layer security - filling in the certificate request

3. **Issue the certificate**

Root CA is the topmost CA. No further higher CA can issue a root CA. It has to be signed by itself.

Root CA should never expire. Otherwise, all the certificates it had issued will need to be reissued, and all of the relevant SSL programs will need to be reconfigured. So we set its valid period to 7305 days (20 years). If we do not set the valid period, it will be set to its default as 30 days (1 month).

The certificate request is not required after it is signed. We can safely delete it.

```
# Sign its own certificate request
openssl x509 -req -days 7305 -sha1
 -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
 -signkey /etc/ssl/private/nshield.key \
 -in /tmp/myrootca.req -out /etc/ssl/certs/nshield.crt
# Delete the certificate request
rm -f /tmp/nshield.req
```

Program listing 45: Link layer security - certificate request

To visualize in details the generated certificate you have to introduce the following sentence in the command-line interpreter:

```
openssl x509 -in /etc/ssl/certs/nshield.crt -noout -text
```

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            ab:95:f3:88:30:04:5a:58
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=ES, ST=MADRID, O=INDRA, OU=WP4, CN=nSHIELD/emailAddress=ibarriv@indra.es
        Validity
            Not Before: Feb 12 15:48:27 2013 GMT
            Not After : Feb 12 15:48:27 2033 GMT
        Subject: C=ES, ST=MADRID, O=INDRA, OU=WP4, CN=nSHIELD/emailAddress=ibarriv@indra.es
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ce:f8:d2:83:06:30:9e:bc:cf:f2:c1:9d:bc:23:
                    45:fa:6a:c2:6e:01:f6:6e:bf:14:58:b3:fa:96:16:
                    6c:f3:55:42:48:53:f0:b7:5b:34:cd:cc:6b:45:e8:
                    ca:0f:5d:5d:13:cd:df:9a:47:9c:19:05:d7:6d:a0:
                    59:92:99:a7:10:20:65:d6:59:0f:af:24:1f:d1:d9:
                    2b:eb:68:cd:e4:50:8c:dd:62:02:1a:28:82:d2:62:
                    5e:9c:66:af:60:a2:2b:4d:1c:4d:94:09:8b:fa:26:
                    87:9b:6a:fe:7d:a1:4b:78:46:4e:9b:fb:0a:88:76:
                    78:39:75:bf:f3:db:4d:68:ca:92:14:9a:3b:aa:3f:
```

```
            07:f8:37:63:fb:67:94:bf:6f:b0:1a:8a:13:9c:82:
            33:4e:44:47:65:19:a3:b8:bd:9c:19:78:1f:20:f0:
            11:63:e1:99:66:4a:6c:48:f3:6f:20:17:4c:7c:35:
            f5:39:5e:a2:4d:93:d9:bc:56:e8:52:1f:2f:06:6c:
            c3:99:b6:d4:a0:f6:25:c8:15:b2:5b:6e:25:16:f3:
            8b:35:d6:8a:ed:1d:cb:d8:44:42:05:63:f9:da:97:
            92:d0:82:43:e1:58:a0:be:a8:85:a8:da:04:5c:b5:
            cd:85:54:79:90:13:25:20:b6:92:7a:a4:1f:2a:55:
            a1:df
        Exponent: 65537 (0x10001)
    X509v3 extensions:
      X509v3 Subject Key Identifier:
        1B:9C:1E:C6:EC:F5:FC:37:3D:4E:40:96:CB:9F:B3:A0:BA:D1:6C:AB
      X509v3 Authority Key Identifier:
        keyid:1B:9C:1E:C6:EC:F5:FC:37:3D:4E:40:96:CB:9F:B3:A0:BA:D1:6C:AB

      X509v3 Basic Constraints:
        CA:TRUE
  Signature Algorithm: sha1WithRSAEncryption
    37:39:41:8b:9e:36:fd:7e:7f:25:e6:30:3e:a1:aa:a4:53:26:
    be:2e:ee:b6:e3:d0:fb:5c:a1:e9:c3:d3:cc:a4:95:94:ac:ad:
    35:70:75:6e:c3:ad:54:a9:95:18:4e:76:e6:90:17:ec:3f:6b:
    c5:11:6a:bf:0d:40:18:b5:44:2f:b5:92:ee:3d:e5:7a:f4:d7:
    de:c1:de:4e:ca:03:7e:31:5a:1f:1b:46:55:47:f1:ec:cd:e5:
    bf:f2:5f:ec:c1:c0:8f:7a:3d:26:d6:0b:f9:4c:71:47:a9:ae:
    0b:d8:d9:ef:b7:66:02:b9:d7:8f:26:f5:e5:b3:40:c2:c9:f8:
    cf:2b:2b:b3:6a:a9:d7:75:d6:70:ed:ce:15:af:c3:2e:de:6c:
    c1:27:42:3a:1d:13:72:d4:6a:cf:32:5d:15:ae:a0:90:a4:1f:
    c9:c8:9b:75:7c:b4:9c:a2:f5:b3:89:34:22:58:ac:3a:b5:bd:
    a4:7f:91:1d:e8:b7:5b:4c:e2:17:24:14:90:de:b4:ba:d5:ed:
    0e:9c:82:7f:11:4b:60:ef:4c:3a:96:36:92:f8:74:34:04:8b:
    30:ed:56:46:64:ef:54:a4:9a:a5:18:0a:d5:24:f8:8c:18:0b:
    de:a1:1f:f7:7e:51:96:23:02:16:81:23:79:63:dd:86:d7:90:
    86:19:1a:e6
```

Figure 5-1: Link layer security - Generated certificate

Creating certificates for nSHIELD' nodes and signing them with the CA

1. **Generating the private (and public) key**

```
# Create an RSA private key
openssl genrsa -out /etc/ssl/private/node1.key 2048
chmod og-rwx /etc/ssl/private/node1.key
```

Program listing 46: Link layer security - creating RSA private key

2. **Fill in the certificate request**

```
# Fill in the certificate request
openssl req -new -key /etc/ssl/private/node1.key -out /tmp/node1.req
```

Program listing 47: Link layer security - filling in certificate request

3. **Issue the certificate**

```
# Sign the certificate request
openssl x509 -req -days 3650 -sha1 \
 -extfile /etc/ssl/openssl.cnf -extensions v3_req \
 -CA /etc/ssl/certs/nshield.crt -CAkey /etc/ssl/private/nshield.key \
 -CAserial /etc/ssl/nshield.srl -CAcreateserial \
 -in /tmp/node1.req -out /etc/ssl/certs/node1.crt

# Delete the certificate request
rm -f /tmp/node1.req
```

Program listing 48: Link layer security - issuing certificate

To visualize in details the generated certificate you have to introduce the following sentence in the command-line interpreter:

```
openssl x509 -in /etc/ssl/certs/node1.crt -noout -text
```

Certificate:
Data:
Version: *3 (0x2)*
Serial Number:
9d:3e:1f:b8:de:b9:07:94
Signature Algorithm: *sha1WithRSAEncryption*
Issuer: *C=ES, ST=MADRID, O=INDRA, OU=WP4, CN=nSHIELD/emailAddress=ibarriv@indra.es*
Validity
Not Before: *Feb 13 12:19:08 2013 GMT*
Not After : *Feb 13 12:19:08 2014 GMT*
Subject: *C=ES, ST=MADRID, O=INDRA,* ***OU=FFD/RFD***, ***CN=MACADDRESS/64-31-50-96-7A-16***
Subject Public Key Info:
Public Key Algorithm: *rsaEncryption*
Public-Key: *(2048 bit)*
Modulus:
00:db:c8:84:4b:42:15:5c:a8:a7:fd:6c:99:42:6f:
73:02:47:93:2d:77:79:7a:64:40:57:d0:d1:bd:9f:
0c:81:ba:cb:23:7b:75:c2:0a:d9:a7:7e:24:ff:ba:
5c:0e:57:ed:ca:3a:db:a5:ff:48:a3:1c:9c:5f:cb:
f5:c0:7f:ec:4e:ae:03:e3:07:05:55:7f:c5:ba:71:
70:bb:f0:a6:97:14:91:6d:b2:69:92:9c:31:f5:de:
7b:25:5d:1b:14:5f:92:59:63:fa:52:b0:70:db:8f:
61:50:55:b6:18:c5:86:f3:98:d8:94:57:d8:a5:95:
7e:f1:6b:c4:4f:1e:94:ae:84:51:fd:d1:8c:76:4a:
a1:91:a5:8c:44:e7:07:13:81:7e:0b:06:52:d3:1e:
5b:c8:88:da:10:31:1c:7b:02:0d:ee:e0:1a:36:7b:
19:f5:3e:3e:41:c0:97:48:e0:73:a4:d5:e9:7e:4f:
7d:74:af:f5:f8:d8:79:b3:cc:88:5c:b3:10:cd:b1:
61:fe:42:52:c1:f6:fd:9a:5e:f7:49:8c:09:b5:fd:
e5:67:a4:4f:5e:85:02:89:bc:e5:0a:85:38:77:80:
2a:53:50:8c:6f:ce:ac:e6:db:c3:d8:f1:1b:02:9e:
7f:e1:71:d6:40:a8:c3:73:2b:9b:c7:08:e5:72:b1:
bb:c1
Exponent: *65537 (0x10001)*
X509v3 extensions:
X509v3 Basic Constraints:
CA:*FALSE*
X509v3 Key Usage:
Digital Signature, Non Repudiation, Key Encipherment
Signature Algorithm: *sha1WithRSAEncryption*
81:da:10:fa:5c:fd:fc:8b:e1:29:08:18:92:34:6a:7c:6f:a2:
ad:37:5e:1c:da:8a:ec:40:81:4d:ed:51:31:56:b2:de:de:51:
7e:d5:9b:83:88:a4:22:16:99:cc:bb:b7:e6:de:25:f1:ef:fb:
19:ab:b6:48:64:99:07:04:db:64:8c:ce:bf:57:d8:3c:10:47:
4b:76:36:0c:3b:a1:27:d7:ce:6d:02:95:db:3d:c2:7c:cb:e5:
a2:22:1f:6b:2f:63:df:e9:8d:4c:79:7d:a6:76:48:32:08:e7:
53:86:e4:1a:63:d7:e7:de:17:c8:7b:46:e8:9c:65:43:8d:db:
43:58:98:74:f6:75:80:f1:a1:9c:48:e3:88:77:f5:7d:16:54:
95:80:dd:35:68:c0:fd:84:73:91:ab:f3:d1:50:75:56:9b:59:
5c:7a:55:bb:88:95:22:3e:05:21:4c:c4:00:a4:fb:ca:49:e3:
d4:78:b7:61:19:a6:df:43:16:53:0b:bc:83:6b:14:7f:0d:85:
3b:29:03:91:35:ff:9b:e3:74:d2:fd:6a:22:6a:37:90:60:b3:
0d:2a:a5:bd:4c:6a:b1:62:a0:64:6f:c2:6a:87:46:a8:36:cc:
f1:cd:36:25:fb:f2:e0:76:a3:b4:00:93:64:ff:d9:1d:37:e7:
58:30:f9:ae

Figure 5-2: Link layer security - generated certificate

Note that the node certificate is issued by the CA (created before). Moreover, the full qualified domain name (FQDN) for the node1 certificate, highlighted in red, corresponds to the MAC address of the node. Moreover, in order to determine the roles of the nodes belonging to the network, we have to specify the Organizational Unit of each node (highlighted also in red). Thanks to this field, for instance, will be allowed the management of node policies in higher layers of nSHIELD project (middleware or application layers).

When a node has been compromised, we will have to revoke its certificate. But this can't be done automatically, should be performed by the administrator of the network.

```
# Setting up the certificate revocation list

touch crlnumber && echo 01 > crlnumber
openssl ca -gencrl -out crl.pem

# Revocating a node certificate
openssl ca -revoke certs/node2.crt
Using configuration from /etc/ssl/openssl.cnf
Enter pass phrase for /etc/ssl/private/nshield.key:
Adding Entry with serial number 9D3E1FB8DEB90795 to DB for
/C=ES/ST=MADRID/O=INDRA/OU=FFD/RFD/CN=MACADDRESS46-13-05-96-B7-61
Revoking Certificate 9D3E1FB8DEB90795.
Data Base Updated

# Printing the revoked certificates (just 1 in that case)
openssl crl -in crl.pem -noout -text
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer:
/C=ES/ST=MADRID/O=INDRA/OU=WP4/CN=nSHIELD/emailAddress=ibarriv@indra.es
        Last Update: Feb 13 17:28:53 2013 GMT
        Next Update: Mar 15 17:28:53 2013 GMT
        CRL extensions:
            X509v3 CRL Number:
                1
No Revoked Certificates.
    Signature Algorithm: sha1WithRSAEncryption
         90:de:c3:a3:0d:d5:4c:5d:c2:8d:a0:1f:09:49:77:8e:50:66:
         5c:50:38:3a:e9:cb:5a:14:b1:65:41:ec:4c:43:06:f5:82:80:
         ef:1b:bf:3f:80:92:37:4b:38:a2:d0:02:6f:df:f9:dd:f9:9d:
         98:30:8d:73:ca:bc:df:1a:1e:c9:0e:b2:a4:7a:50:12:81:37:
         dc:2b:46:30:8f:51:28:6e:fe:ad:1d:bb:c9:05:b7:48:4f:d3:
         c1:81:75:49:71:42:13:31:1d:d7:60:a8:fa:92:64:a9:f8:70:
         96:5c:dc:9a:9e:00:dd:f4:cb:1b:1d:f4:34:fc:3c:40:d8:de:
         b0:71:3c:c7:29:48:58:6b:90:52:38:2f:44:cb:4d:fd:5a:28:
         59:22:87:a7:d6:5b:4b:78:7f:98:1d:82:d5:50:87:6b:e6:89:
         fd:94:48:47:de:bd:1f:08:a3:86:79:aa:93:bf:42:9c:23:f2:
         c8:c7:36:8a:02:f2:b8:7c:d9:8c:43:1a:a4:26:9a:1c:2f:f6:
         25:c7:ac:14:69:b1:a4:96:b3:94:53:f1:47:e1:76:bd:45:9b:
         e2:89:7d:67:d1:06:cc:90:d7:78:ac:08:88:07:e9:77:10:f3:
         32:2b:ca:60:20:7b:5c:b1:6c:33:51:d3:77:68:42:b7:77:d4:
         1c:b3:25:a2
```

Program listing 49: Link layer security - revoking the node's certificate

5.1.2 Proposed solution

IEEE standard 802.15.4 intends to offer the fundamental lower network layers of a type of wireless personal area network (WPAN) which focuses on low-cost, low-speed ubiquitous communication between devices. The emphasis is on very low cost communication of nearby devices with little to no underlying infrastructure, intending to exploit this to lower power consumption even more.

Following D4.3, the proposed solution to give link layer security to the network is to use CTR, CBC-MAC and CCM algorithms, which will be implemented over one of the most extended operating systems, TinyOS 2.x. Concretely, the device to make the test will be a Zolertia Z1 mote which has a CC2420 chip.

If we analyse the characteristics of CC2420 we can check that it supports security operations such ciphering, authentication and integrity and it is capable of performing these functions at the MAC level, among which are included CTR (encryption), CBC-MAC (authentication and integrity) and CCM (encryption + authentication and integrity). Each one of these based on AES (Advanced Encryption Standard) 128 bit key.

Inside the RAM there is space to save two keys to make the operations but it will be throw the application the way to select which one will be used.

5.1.3 Algorithms implementation

As mentioned on the previous paragraph, TinyOS is one of the most well-known operative systems for motes. After the first version where TinySec was used to provide the link layer security, appeared the IEEE 802.15.4 standard and was necessary to implement different algorithms to provide the security level demanded.

Taking into account that the mote used to do test is a Zolertia Z1 with a CC2420 radio chip, the algorithms on the tinyOS layer will be as follows, to be 802.15.4 compliant.

5.1.3.1 CTR

This algorithm is provided of the message to send, the identifier of the register where the key is stored on the chip and the bytes it have to skip to do the ctr operation.

```
command error_t CC2420SecurityMode.setCtr(message_t* msg, uint8_t setKey, uint8_t setSkip)
{
    cc2420_header_t* hdr = (cc2420_header_t*)msg->header;
    security_header_t* secHdr = (security_header_t*) & hdr->secHdr;

    #if ! defined(TFRAMES_ENABLED)
      (uint8_t*)secHdr += 1;
    #endif

    if (setKey > 1 || setSkip > 7)
    {
      return FAIL;
    }

    secLevel = CTR;
    keyIndex = setKey;
    reserved = setSkip;
    nonceCounter++;

    secHdr->secLevel = secLevel;
    secHdr->keyMode = 1; // Fixed to 1 for now
    secHdr->reserved = reserved; //skip in cc2420
    secHdr->frameCounter = nonceCounter;
    secHdr->keyID[0] = keyIndex; // Always first position for now due to fixed key Mode
    hdr->fcf |= 1 << IEEE154_FCF_SECURITY_ENABLED;

    return SUCCESS;
}
```

Program listing 50: Link layer security - CTR algorithm

5.1.3.2 CBC-MAC

This algorithm is provided of the message to send, the identifier of the register where the key is stored on the chip and the bytes it have to skip to do the CBC-MAC operation and the size of the message authentication code

```
command error_t CC2420SecurityMode.setCbcMac(message_t* msg, uint8_t setKey,
                                             uint8_t setSkip, uint8_t size)
{
   cc2420_header_t* hdr = (cc2420_header_t*)msg->header;
   security_header_t* secHdr = (security_header_t*)&hdr->secHdr;

   #if ! defined(TFRAMES_ENABLED)
         (uint8_t*)secHdr += 1;
   #endif

   if (setKey > 1 || (size != 4 && size != 8 && size != 16) || (setSkip > 7))
   {
         return FAIL;
   }

   if(size == 4)
         secLevel = CBC_MAC_4;
   else if (size == 8)
         secLevel = CBC_MAC_8;
   else if (size == 16)
         secLevel = CBC_MAC_16;
   else
         return FAIL;

   keyIndex = setKey;
   reserved = setSkip;
   nonceCounter++;

   secHdr->secLevel = secLevel;
   secHdr->keyMode = 1; // Fixed to 1 for now
   secHdr->reserved = reserved; //skip in cc2420
   secHdr->frameCounter = nonceCounter;
   secHdr->keyID[0] = keyIndex; // Always first position for now due to fixed key Mode
   hdr->fcf |= 1 << IEEE154_FCF_SECURITY_ENABLED;

   return SUCCESS;
}
```

Program listing 51: Link layer security - CBC-MAC algorithm

5.1.3.3 CCM

This algorithm is provided of the message to send, the identifier of the register where the key is stored on the chip and the bytes it have to skip to do the CCM operation and the size of the message authentication code

```
command error_t CC2420SecurityMode.setCcm(message_t* msg, uint8_t setKey,
                                          uint8_t setSkip, uint8_t size)
{
   cc2420_header_t* hdr = (cc2420_header_t*)msg->header;
   security_header_t* secHdr = (security_header_t*)&hdr->secHdr;

   #if ! defined(TFRAMES_ENABLED)
      (uint8_t*)secHdr += 1;
   #endif

   if (setKey > 1 || (size != 4 && size != 8 && size != 16) || (setSkip > 7))
   {
      return FAIL;
   }

   if(size == 4)
      secLevel = CCM_4;
   else if (size == 8)
      secLevel = CCM_8;
   else if (size == 16)
      secLevel = CCM_16;
   else
      return FAIL;

   keyIndex = setKey;
   reserved = setSkip;
   nonceCounter++;

   secHdr->secLevel = secLevel;
   secHdr->keyMode = 1; // Fixed to 1 for now
   secHdr->reserved = reserved; //skip in cc2420
   secHdr->frameCounter = nonceCounter;
   secHdr->keyID[0] = keyIndex; // Always first position for now due to fixed key Mode
   hdr->fcf |= 1 << IEEE154_FCF_SECURITY_ENABLED;

   return SUCCESS;
}
```

Program listing 52: Link layer security - CCM algorithm

5.1.4 Test programs

The first thing we have to do is to store the key value:

```
uint8_t key[16] =
{0x98,0x67,0x7F,0xAF,0xD6,0xAD,0xB7,0x0C,0x59,0xE8,0xD9,0x47,0xC9,0x71,0x15,0x0F};
```

After that we call to *setKey* to select the register where the key will be set.

```
call CC2420Keys.setKey(1, key);
```

To control when the *setKey* has finished we will use this event:

```
event void CC2420Keys.setKeyDone(uint8_t keyNo, uint8_t* skey)
{
    keyReady=1;
}
```

And when this event is signalled, we will call the instructions to encrypt and send the packet:

```
counter=1234; //Data to be transmitted
if(keyReady == 1)
{
    radio_count_msg_t* rcm = (radio_count_msg_t*)call Packet.getPayload(&packet,
                            sizeof(radio_count_msg_t));

    if (rcm == NULL)
    {
        return;
    }

    rcm->counter = counter;
    call CC2420Security.setCtr(&packet, 0, 0);
    //call CC2420Security.setCbcMac(&packet, 0, 0, 16);
    //call CC2420Security.setCcm(&packet, 1, 0, 16);
    call PacketLink.setRetries(&packet, 1);
    call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(radio_count_msg_t));
}
```

Program listing 53: Link layer security - encrypting and sending the packet

To check if the frames sent by the program are correct, we can sniff the radio channel and use a protocol analyser or take one of the programs implemented by TinyOS, like Base Station which reads the data received by radio and writes it on the serial port.

5.1.5 Analysis results

To make a correct analysis of the results, firstly we will focus on the frames obtained applying security algorithms and without applying it, and finally we will focus on energy consumption.

Here we can see the same frame transmitted without security:

```
69 88 e3 22 00 ff ff 01 00 3f 00 00 0a e4 01 00 06 00 01 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00
```

applying CTR algorithm

```
69 88 e3 22 00 ff ff 01 00 3f 00 00 0a e4 01 00 06 23 0d 15 6c ca cd 0c d4 ba 7e 17 a0
72 b6 ca 38 99 fa 1b 73 13 41 4e 92 6c 96 54 48 6e 20 c6 b4 09 f6 5c 98 ef a1 58 6d 61
fb e2 70 b1 5b 6b dc 85 1e f3 d3 eb 7c
```

applying CBC-MAC-4 algorithm

```
69 88 e3 22 00 ff ff 01 00 3f 00 00 0a e4 01 00 06 00 01 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 A2 80 87 59
```

and applying CCM-4 algorithm

```
69 88 e3 22 00 ff ff 01 00 3f 00 00 0a e4 01 00 06 23 0d 15 6c ca cd 0c d4 ba 7e 17 a0
72 b6 ca 38 99 fa 1b 73 13 41 4e 92 6c 96 54 48 6e 20 c6 b4 09 f6 5c 98 ef a1 58 6d 61
fb e2 70 b1 5b 6b dc 85 1e f3 d3 eb 7c B4 81 37 86
```

and it's shown that the CTR algorithm ciphers the data, the second generates a MAC code and adds it at the end of the frame maintaining the original data on the previous bytes and finally the CCM encrypts and generates a MAC code.

Turning now to measured energy consumption on a transmitting device and on a receiver one and using the most powerful method of security analysed in the study (CCM-16) and sending the same messages using no security methods we have obtained the following results:

Table 5-1: Link layer security - Energy consumption

	Without security		CCM16	
data (bytes)	TX energy(mJ)	RX energy(mJ)	TX energy(mJ)	RX energy(mJ)
60	0,6701	0,7259	0,7463	0,8819
70	0,7493	0,8117	0,8057	0,9521
80	0,8069	0,8741	0,8585	1,0145
90	0,8717	0,9443	0,9245	1,0925

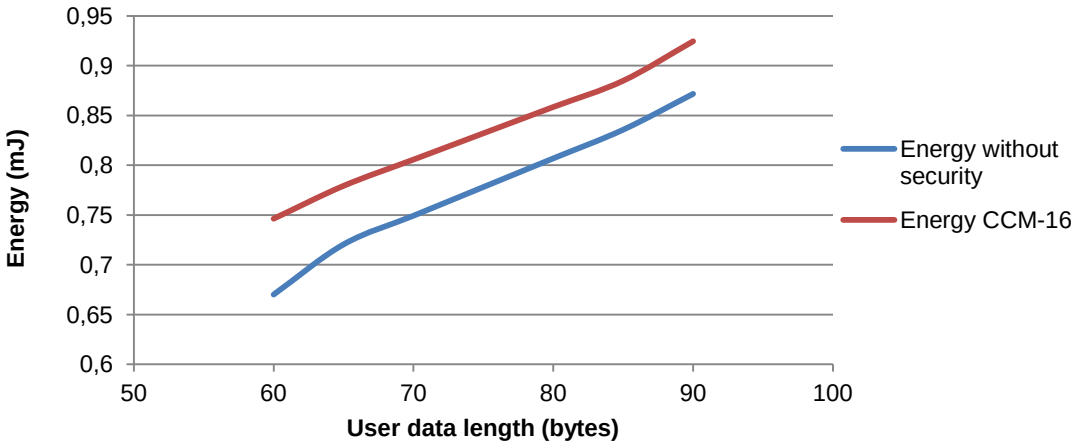


Figure 5-3: Link layer security - Energy consumption on transmission

As we can see in the figure below, the energy needed to transmit the same quantity of data using security is higher than if we don't use security. This is because if we use security the sending and receiving times are longer and with security is needed a processing time to transform plain data into secure data.

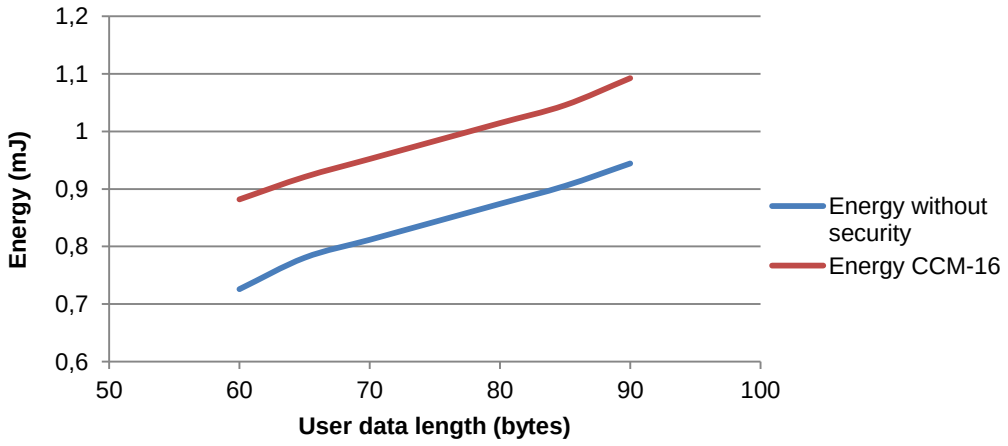


Figure 5-4: Link layer security - Energy consumption on reception

Reception results are more or less the same than in transmission mode because the quantity of data to receive is higher on a secure mode due to the headers and process time.

According to the results we can conclude saying that the use of security modes on both, transmission and reception processes are more expensive in the use of energy than programs that don't use security to send messages. But this cost is acceptable because the system protection is a great advantage on this kind of communication.

5.2 Secure communication protocols on the network layer

5.2.1 Scheme prerequisites

Security at the network layer for networks adopting the TCP/IP stack is provided by the standardized IPsec protocol, which typically inherits all IP's characteristics and requirements. Deploying IPv6 in the restricted environment of nSHIELD nodes using IEEE802.15.4 as the data link layer protocol for transferring messages poses a major challenge. This is due to the very limited size of IEEE802.15.4 frames which is restricted to 127 octets[1], hence not satisfying IPv6's requirement for an MTU of at least 1280 octets. Such a limited frame length requires special handling to accommodate IPv6 datagrams. As a result, the 6LoWPAN adaptation layer was introduced to act as a bridge between these two protocols and reduce the large IPv6 header while considering restrictions in terms of computation power, memory, bandwidth and energy. The solution is header compression which is defined in RFC 6282 [26].

More specifically, RFC 6282 defines an encoding format, namely LOWPAN_IPHC shown in the following figure, for effective compression of IPv6 header fields. LOWPAN_IPHC consists of 2 or 3 octets where the first three bits are the dispatch value, as it is defined in RFC 4944. The same RFC also defines a format, namely LOWPAN_NHC, for next headers, while dedicated bits in LOWPAN_IPHC indicate whether the next header is encoded using LOWPAN_NHC. In this case the encoded LOWPAN_NHC immediately follows the compressed IPv6 header.

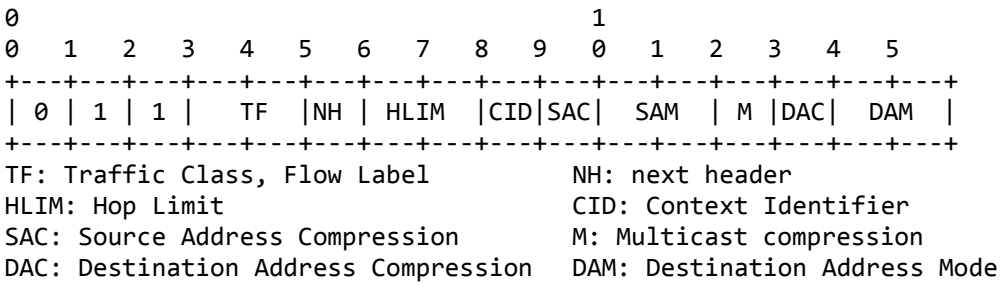


Figure 5-5: Network layer security - LOWPAN_IPHC base format

Following this compressed format, IPv6 header fields can be fully elided or placed immediately after the LOWPAN_IPHC, either in a compressed form if the field is partially elided or literally as shown in the figure below.

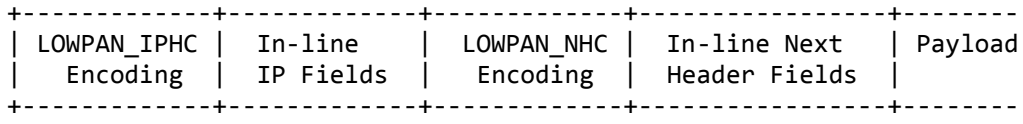


Figure 5-6: Network layer security - IPv6 Compressed Datagram

[1] The maximum MAC layer header size is 25 octets, hence leaving only 102 octets for the payload. If AES-CCM-128 is also used for protecting these messages, this leaves only 81 octets for upper layers. If no compression is used for the IP and UDP headers, hence another 40 plus 8 bytes are needed respectively, only 33 bytes remain for the actual data.

The proposed format can be used to compress header fields. Among them, the two addresses which "are formed with an IID derived directly from either the 64-bit extended or the 16-bit short IEEE 802.15.4 addresses" [26]. Header compression is most effective when communicating with link-local addresses where IPv6 address field can be reduced from 16 bytes down to 16 bits. If SAC is set, i.e. stateful, context-based compression is used, and the SAM field has the binary value of 0b10, the first 112 bits of the source address are elided and only 16 bits, which are carried in line, are used for the address, while if SAM = 0b11, the address is fully elided. Therefore, the approach taken here to secure messages using IPSEC is most applicable within the 6LoWPANnetwork. If a message crosses the borders of this network, through a gateway, routable addresses have to be used instead[2].

The NH field in LOWPAN_IPHC denotes whether the full 8 bits of the Next Header field are carried in line (value 0), or the Next Header field is compressed and therefore the next header is encoded using LOWPAN_NHC. LOWPAN_IPHC elides the IPv6 Next Header field when the NH bit is set to 1. The value of IPv6 Next Header is recovered from the first bits in the LOWPAN_NHC encoding. As a result, the structure of an IPv6 datagram compressed using LOWPAN_IPHC and LOWPAN_NHC is as shown in the following figure (note that the "In-line IP fields" are uncompressed IP headers that follow the LOWPAN_IPHC Encoding).

The encoding of LOWPAN_NHC is as shown in the figure below (note that according to IPv6, next header can either be a Transport Layer protocol header (e.g. UDP) or an extension header (e.g. IPSEC) :

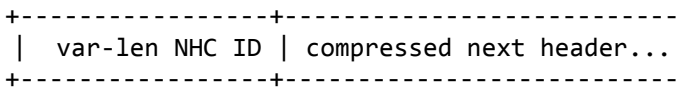


Figure 5-7: Network layer security - LOWPAN_NHC encoding

Looking specifically at IPv6 extension headers "the LOWPAN_NHC encodings" for IPv6 Extension Headers are composed of a single LOWPAN_NHC octet followed by the IPv6 Extension Header [26].

The format of the LOWPAN_NHC octet for IPv6 extension header is shown in the figure below. Note that the first 4 bits have the value of "1110", according to the IANA registry created by RFC 6282:

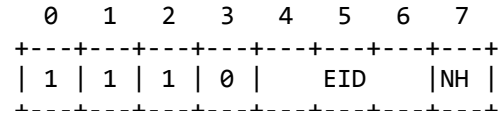


Figure 5-8: Network layer security - LOWPAN_NHC format for IPv6 Extension header

The EID field identifies the IPv6 Extension Header that follows the LOWPAN_NHC byte. In the first solution described below, both these values are used while the second only one is required leaving the other for future use. NH has the same role as previously mentioned and is used to denote whether the Full 8 bits of the Next Header, i.e. the extension header, are carried in-line (NH=0) or the Next Header field is elided and the next header is encoded using LOWPAN_NHC (NH=1).

There are typically three options (initially proposed in [27] and [28]) to encode a new IPSEC header using LOWPAN_IPHC:

1. One reserved EID slot is used to denote that an IPsec protocol header is to follow while the ID bits of the encoded extension header (NHC ID) define whether next header refers to AH or ESP.

[2] IEEE802.15.4 devices may use either IEEE 64-bit extended addresses or 16-bit addresses that are unique within a PAN.

2. Two LOWPAN_NHC encodings are introduced for AH and ESP respectively. This is the approach taken in [27] where the authors used both reserved EID values for the new headers. In this case the NHC ID bits are redundant.
3. A LOWPAN_NHC_IPSEC encoding which will be used to further introduce another LOWPAN_NHC encoding, one for AH and one for ESP.

5.2.2 Compressed IPsec ESP and AH

Raza et. al. propose in [27] and [28] extension header encodings for AH and ESP. Although all the aforementioned options regarding the use of the EID value are defined the authors preferred using both reserved values to encode AH (EID=101) and ESP (EID=110). The format of the two header encodings is as follows:

- **LOWPAN_NHC_AH**. The encoded header for AH is shown below.

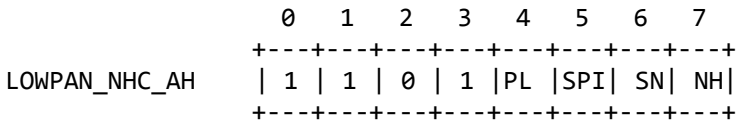


Figure 5-9: Network layer security - LOWPAN_NHC_AH header encoding

The corresponding fields can take the following values:

- The first four bits in the NHC AH represent the NHC ID for AH, and are set to 1101.
- **PL (Payload Length)**: If 0, the payload length is omitted. This length can be obtained from the SPI value because the length of the authenticating data depends on the algorithm used and are fixed for any input size. If 1, the length is carried in line after the NHC AH header.
- **SPI (security Parameter Index)**: If 0, the default SPI for the sensor network is used and the SPI field is omitted. The default SPI value is set to 1. This does not mean that all nodes use the same security association (SA), but that every node has its own preferred SA, identified by SPI 1. If 1, the SPI is carried in line.
- **SN (Sequence Number)**: If 0, a 16 bit sequence number is used and the leftmost 16 bits are assumed to be zero. If 1, all 32 bits of the sequence number are carried inline.
- **NH (Next Header)**: If 0, the next header field in AH will be used to specify the next header and it is carried inline. If 1, the next header field in AH is skipped. The next header will be encoded using NHC.

- **LOWPAN_NHC_ESP**. The encoded header for ESP is shown in below.

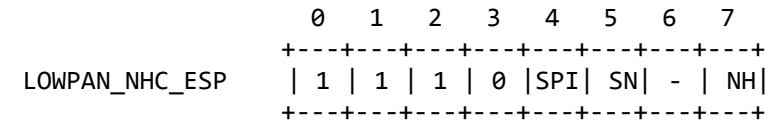


Figure 5-10: Network layer security - LOWPAN_NHC_ESP header encoding

The corresponding fields can take the following values.

- The first 4 bits in the NHC ESP represent the NHC ID we define for ESP. These are set to 1110.

- **SPI**: If SPI = 0: The default SPI for the sensor network is used and the SPI field is omitted. We set the default SPI value to 0.If SPI = 1: All 32 bits indicating the SPI are carried in line after the NHC ESP header.
- **SN**: If SN = 0: A 16 bit sequence number is used. The leftmost 16 bits are assumed to be zero. If SN = 1: All 32 bits of the sequence number are carried in line after the NHC ESP header.
- **NH**: If NH = 0: The next header field in ESP will be used to specify the next header and it is carried inline. If NH = 1: The next header field in ESP is skipped. The next header will be encoded using NHC. This is only possible if hosts are able to execute 6LoWPAN compression/decompression and encryption/decryption jointly.

5.2.3 Compressed IPsec ESP with AES in CCM* mode

This scheme focuses on the use of Advanced Encryption Standard (AES) algorithm in Counter with CBC-MAC (CCM) mode on IPSEC ESP protocol to protect messages on the network layer and provide confidentiality, integrity and data origin authentication. CCM mode is also the choice of preference for IEEE802.15.4 message protection while a scheme to use it with IPSec Encapsulating Security Payload (ESP) is defined in RFC 4309 [29].

AES-CCM has two parameters:

- **M**: the length in octets of the authentication data also known as Integrity Check Value - ICV. The ICV is computed for the ESP header, Payload, and ESP trailer fields. Although *M* can take the values of 4, 6, 8, 10, 12, 14, and 16 [30], RFC 4309 accepts the values of 8 and 16 and optionally 12 octets. IEEE 802.15.4 uses a value of 0, i.e. authentication is not used 4, 8 and 16. The scheme described here adopts the same values as 802.15.4 as possible lengths of the authentication data, i.e. 0, 4, 8, and 16 octets.
- **L**: The size of the length field in octets, where the length includes all of the encrypted data, which also includes the ESP Padding, Pad Length, and Next Header Fields. Although CCM defines values of L between 2 and 8 octets, RFC 4309 accepts only the value of 4. This value exceeds the needs of 802.15.4 where the value of L is set to 2.

For AES_CCM mode the Payload found in a typical ESP header consists of the Initialization Vector, followed by the Encrypted Payload and the Authentication Data, i.e. an encrypted ICV, also shown in the figure below:

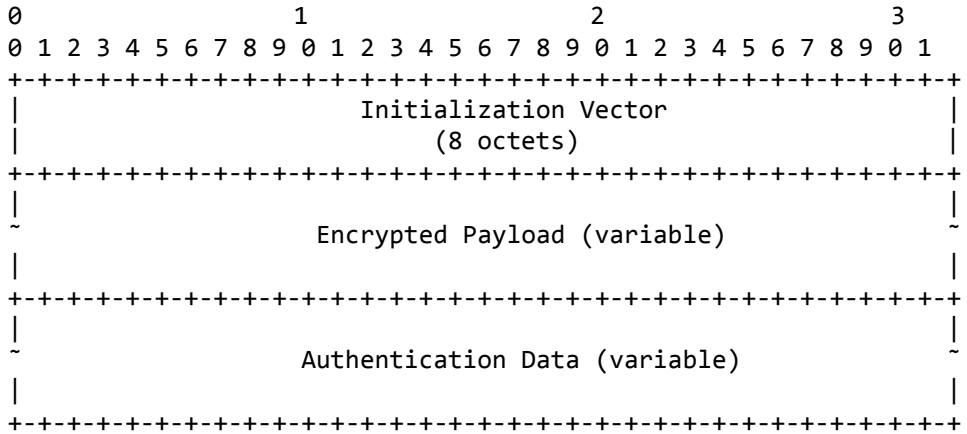


Figure 5-11: Network layer security - ESP payload

In this scheme for the encoding of the aforementioned values using LOWPAN_NHC only a single EID value, i.e. value 101 is reserved for IPSEC, thus leaving the other available EID value for future use. The NHC ID bits of the following LOWPAN_NHC are used to further distinguish among the different flavours.

Therefore, a LOWPAN_NHC_IPSEC header is proposed with the format shown in the following figure.

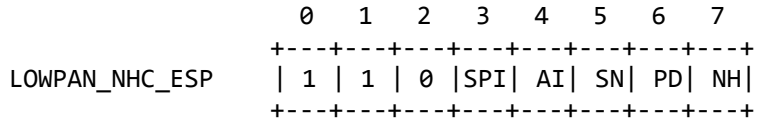


Figure 5-12: Network layer security - LOWPAN_NHC_ESP Header format

The above fields can convey within the NHC_ESP header all the necessary information for AES_CCM:

- **SPI (Security Parameter Index):** If SPI=0 the default SPI is used and the SPI is omitted. If SPI=1 all 32 bits are carried inline, following the ESP header (LOWPAN NHC ESP).
- **AI (Address Inclusion):** AES-CCM* offers the capability to include headers in the computation of the authentication field, without encrypting them. This typically allows the inclusion of extra fields to the payload header in the computation of the ICV. Such fields are the nodes' addresses. If AI=1 the addresses are included in the computation of the authentication code, while if AI=0 they are omitted.
- **SN (Sequence Number):** If SN=0 then the sequence number required to construct the 13-byte Nonce field (see below), is inline and consists of 2 bytes, while the left most 16 bits are assumed to be zero. If SN=1 all 32 bits (4 octets) are carried inline after the ESP header (LOWPAN NHC ESP).
- **PD (Padding):** This field is used to denote whether padding is added to the data prior to being encrypted, according to the ESP specifications [31] and RFC 4309 [29]. In contrast to ESP specifications the Pad Length field is optional and must only be present if PD=1. In this case the padding data must also be present while Padding, Pad Length and Next Header fields must be concatenated prior to being encrypted, according to RFC 4309 [29].
- **NH (Next Header):** If NH=0, the Next Header field in ESP will be used to specify the next header and it is carried inline. If NH=1, the Next Header field in ESP is skipped. The next header will be encoded using NHC. This is only possible if hosts are able to execute 6LoWPAN compression/decompression and encryption/decryption jointly.

Using AES in counter mode requires generating a sequence of counter blocks, based on an IV carried in each packet [29]. These counter blocks are in turn used to generate the key stream. The AES counter block has a length of 16 octets and comprises of a 1-byte Flags field, a 13-byte Nonce and a 2-byte Length field (shown in

Figure 5-13) is the first block that has to be constructed for authentication purposes.

The counter block has a length of 16 octets and comprises of a 1-byte Flags field, a 13-byte Nonce and a 2-byte Length field (shown here is the first block that has to be constructed for authentication purposes).

BYTES	0	1-13	14-15
	Flags	Nonce	Length field

Figure 5-13: Network layer security - 1st Block

The Flags field is in turn formatted as shown in the following figure:

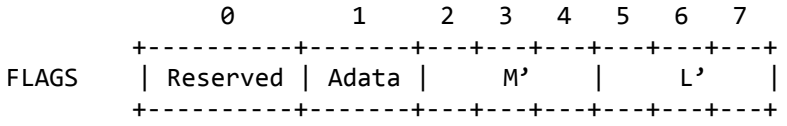


Figure 5-14: Network layer security - Flags Byte

The 1-bit reserved field is reserved for future expansions and shall be set to '0'. The 1-bitAdata field is set to '0' if *l*(*a*) = 0 and set to '1' if *l*(*a*) gt 0. The M field is the 3-bit representation of the integer (*M* - 2)/2 if *M* gt 0 and of the integer 0 if *M* = 0, in most-significant-bit-first order. The L field is the 3-bit representation of the integer *L* - 1, in most-significant-bit-first order.

The13-byte Nonce field consists of the following:

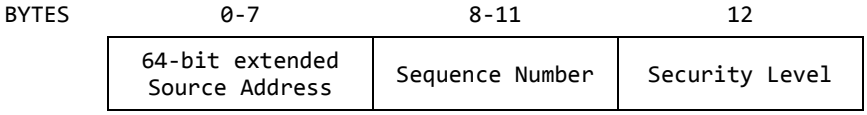


Figure 5-15: Network layer security - 13 byte nonce filed

while the Security Level byte comprises of the following fields:

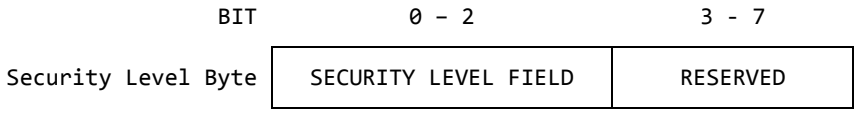


Figure 5-16: Network layer security – security level byte structure

Finally the Security Level Field can take the following values:

Table 5-2: Network layer security - security level field values

Security Level	Security level field b_2 b_1 b_0	Security Attributes	Data Confidentiality	Data Authenticity	Encrypted authentication tag length, M octets
0	000	None	OFF	NO	0
1	001	MIC-32	OFF	YES	4
2	010	MIC-64	OFF	YES	8
3	011	MIC-128	OFF	YES	16
4	100	ENC	ON	NO	0
5	101	ENC-MIC-32	ON	YES	4
6	110	ENC-MIC-64	ON	YES	8
7	111	ENC-MIC-128	ON	YES	16

On top of the above, the IP Addresses Flag is used to denote whether the IP Addresses found in the IP Header are included in the computation of the Authentication Data. If “IP Addresses Flag=0” the IP

Addresses are not included while if “IP Addresses Flag=1” IP Addresses are part of the Authentication Data, hence providing integrity to the IP addresses in a similar to AH approach. Obviously, the flag cannot have the value of 1 if Data Authenticity is not used.

5.2.4 Experimental results

An implementation of IPsec with AES-CCM* was performed using the Contiki operating system and the COOJA simulator, using Tmote Sky motes for the simulation setup. The IPsec headers were compressed for 6LoWPAN in the manner presented above and Contiki’s µIP stack was modified accordingly. The AES implementation used was the one provided by the MIRACL library [32].

Tmote Sky motes have a Texas Instruments MSP430 microcontroller, with 10 KB RAM and 48 KB Flash memory [33]. Therefore, the msp430-size utility of the respective compiler toolchain can be used for providing an estimate of the expected memory usage, both in terms of flash memory and stack (RAM) size. The obtained values were approximately 45.5 KB for flash and 8 KB for RAM.

It is also worth emphasizing that COOJA is unable to simulate code that utilizes any AES implementation on the mote’s hardware (in the case of the Tmote Sky, such functionality is provided by the CC2420 chip), thus requiring a software implementation as well. By compiling the code twice (once using MIRACL’s AES implementation and once using only the necessary statements that utilize the provided functionality of the CC2420 chip) and comparing the two results, it was concluded that the one with the software AES implementation is larger by approximately 2.6 KB and requires an additional 0.2 KB of RAM.

In order to get an idea of the imposed packet overhead, the proposed AES-CCM* scheme is compared to compressed IPsec that uses the “traditional” approach of AH and ESP, as well as to 802.15.4. It should be emphasized that 802.15.4 supports only link-layer security, which has the advantage of lower packet overhead, at the expense of increased power consumption. The main difference among these three schemes is that the ones using IPsec are able to offer end-to-end security, whereas the 802.15.4 inherent link-layer security can only offer node-to-node security. The comparison results are summarized in Table 5-3

Table 5-3: Network layer security - Comparison of packet overhead

Security Service	AES-CCM*		Compressed IPsec		802.15.4	
	Attributes	Overhead	Attributes	Overhead	Attributes	Overhead
Authentication	MIC-32 MIC-64 MIC-128	10 14 24	AH with HMAC-SHA1-96	16	AES-CBC-MAC-96	12
Encryption	ENC	12	AES-CBC	12	AES-CTR	5
Both	ENC-MIC-32 ENC-MIC-64 ENC-MIC-128	10 14 24	AH with HMAC-SHA1-96 and ESP with AES-CBC	24	AES-CCM-128	21

Finally, some processing speed and energy consumption measurements are presented in Figure 5-17 and Figure 5-18, respectively.

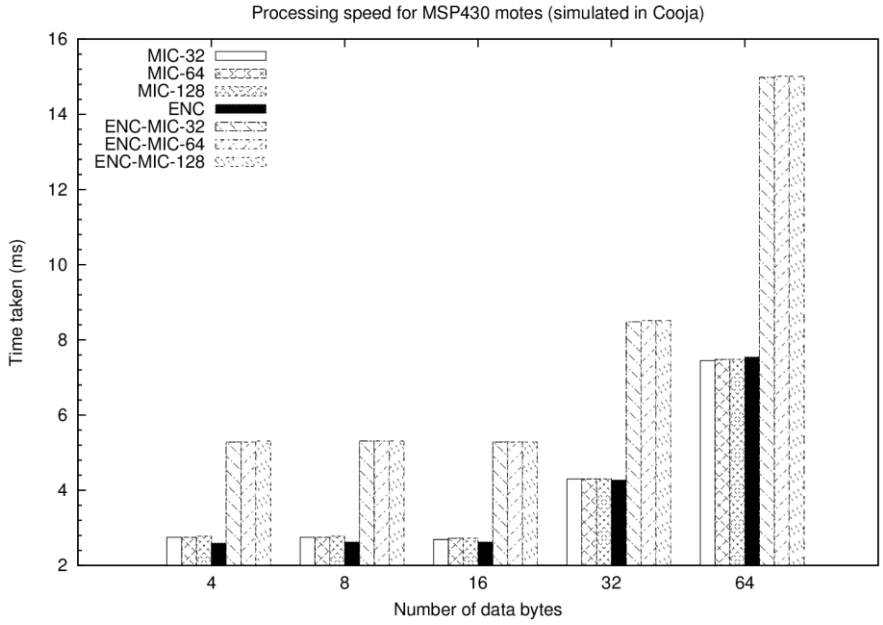


Figure 5-17: Network Layer Security - processing speed measurements

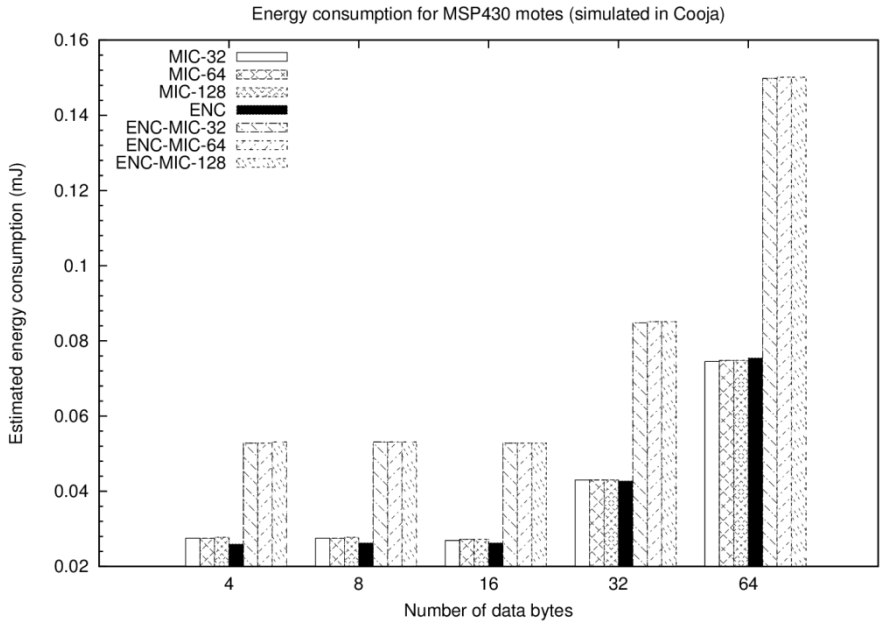


Figure 5-18: Network layer security - Energy consumption measurements

5.3 Access control in Smart Grid networks

Due to many circumstances, power industry faces crucial irreversible changes. Classic schematics, i.e. tariffs, begins to meet its limits, modern ecological power supplies have high rate of unpredictable supply with changes in almost second's intervals. Defined energy flow from large producer towards consumer is significantly changed as distributed power supplies play more important role and production is moved on the both, low and high voltage levels. Last, but not least, the idea of load characteristics, where basic harmonics belong to small consumers and higher harmonics are related to high load is also obsolete. Currently, the modern households use low input appliances, and we can rarely find appliance that

wouldn't distort the energy network. At this place, it is necessary to mention this situation could remain (in the best case) the same, but this is very optimistic view.

Communication channel is one of the most crucial elements of whatever solution. It serves for the data exchange between single devices of Smart Grid. We are able to define two types of transmitted data: i) data with need of defined latency (they are very short and their number is relatively low) ii) other data of higher volume with noncritical delivery time in hours. In order to ensure functionality, it is necessary to respect communication channel characteristics. It is not possible to extend the volume of transmitted data. It is inevitable (and responsible) to consider necessity of transmitted data. Wherever, we can generate whatever data volume, but we have to strictly ask: Cui bono (to whose benefit?)?

Realized pilot projects have a significant sense for development of modern communication modem that use power lines for communication. Requirements for such modem were defined: i) reliability and robustness of the transmitted data, and ii) communication rate and large volume data transmission. Interoperability is definitely obligatory aim. Whatever interoperable protocol has to respect physical characteristics of communication channel, the reverse process is excluded.

New device has to be also resistible against cyber criminality. The only encryption is not sufficient, it is necessary to build the whole secured system with complete key management. There will be systems of primary acquired data, subsequent processing systems, and control systems, as parts of the whole system. Credible data transfer from meter to billing is crucial from the legal point of view. The implementation of electronic signature is the only relevant tool how to solve this request. Therefore, such a solution has to be find that brings satisfactory cryptographic robustness whilst not to burden original message (and not decrease data channel throughput) too much.

New emerging technologies and devices for tele-controlling energy consumption across the Smart Grid carry new threats and vulnerabilities that could be exploited by both internal and external agents. The Smart Meter is a clear example of this: this device requires different network protocols (such as M&M, PRIME and DLMS) for communicating towards the concentrator, and security is not being held as a primary approach, but as an add-on solution.

Therefore, Smart Grid operators are including different devices across the network without being able to manage security for preserving privacy and confidentiality in low power lines and sabotages in medium power line. Operators not only need to protect their networks but also to know how security is being transferred.

The prototypes will bring two scenarios (but just one of them will be analysed): the first one focuses on low power line and is concerned with privacy and integrity mechanisms. The second one (out of scope but considered for future work) focuses on medium power line and aims to protect the availability and integrity for defending against cyber-sabotage.

Low voltage domain

TECNALIA certifies functionality in DLMS network layer for connection between devices in last mile. Implementation of security in this layer is not yet established. Some control access mechanisms have been specified by CENELEC in blue book for DLMS as referenced in the following picture by CLASS security:

Security setup		0...n	class_id = 64, version = 0			
Attributes		*Data type*	*Min.*	*Max.*	*Def.*	Short name
1. logical_name	(static)	octet-string				x
2. security_policy	(static)	enum				x + 0x08
3. security_suite	(static)	enum				x + 0x10
4. client_system_title	(dyn.)	octet-string				x + 0x18
5. server_system_title	(static)	octet-string				x + 0x20
Specific methods		*m/o*				
1. security_activate		o				x + 0x28
2. global_key_transfer		o				x + 0x30

logical_name	Identifies the "Security setup" object instance. See 4.12.2.25.
security_policy	Enforces authentication and/or encryption algorithm provided with security_suite. enum: (0) nothing, (1) all messages to be authenticated, (2) all messages to be encrypted, (3) all messages to be authenticated and encrypted
security_suite	Specifies authentication, encryption and key wrapping algorithm. enum: (0) AES-GCM-128 for authenticated encryption and AES-128 for key wrapping

Figure 5-19: Smart Grids - Security Setup class for DLMS Cosem

This class specifies both control access and encryption for assuring security in network layer within low voltage domain in Smart Grid area. The objective of TECNALIA will be to analyse that this implementation could be linked to nSHIELD focus. In Deliverable 4.3 these classes are specified more detailed.

Second analysis will not be carried out in nSHIELD project. It is focused on the IEC 60870-5-104 Transmission Protocols, Network access for IEC 60870-5-101 using standard transport profiles (Future Work).

6 References

[1] K. Dabcevic, L. Marcenaro and C. Regazzoni, “Reputation-based frequency switching algorithm for defense against intelligent jamming attacks in centralized Cognitive Radio Networks,” in *submitted for IEEE International Conference on Sensing, Communication, and Networking 2013 (SECON 2013)*, 2013.

[2] P. Morerio, K. Dabcevic, L. Marcenaro and C. Regazzoni, “Distributed cognitive radio architecture with automatic frequency switching,” in *Complexity in Engineering (COMPENG) pp. 1 –4*, 2012.

[3] Atta distribution, “Atta distribution and docs,” [Online]. Available: https://bitbucket.org/lgeretti/atta. [Accessed 2013].

[4] “YAML Official Web Site,” [Online]. Available: http://yaml.org/. [Accessed 2013].

[5] “Apache ZooKeeper Official Documentation,” [Online]. Available: http://zookeeper.apache.org/doc/current/. [Accessed 2013].

[6] “Apache Thrift Official Web Site,” [Online]. Available: http://thrift.apache.org/. [Accessed 2013].

[7] S. K. Singh, M. P. Singh and D. K. Singh, “Routing Protocols in Wireless Sensor Netwroks – A Survey,” *In International Journal of Computer Science & Engineering Survey (IJCSES),* vol. 1, no. 2, pp. 63-83, November 2010.

[8] H. G. and M. C., “Building Trust in Ad hoc Distributed Resource-sharing Networks Using Reputation-based Systems,” in *In 16th Panhellenic Conference on Onformatics PCI - pp. 416-421*, Available on-line via http://doi.ieeecomputersociety.org/10.1109/PCi.2012.28, 2012.

[9] S. Marti, T. J. Giul, K. Lai and M. Baker, “Mitigating Routing Misbehavior in Mobile Ad Hoc Networks,” in *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom’00)*, 2000.

[10] S. Buchegger and J. L. Boudec, “Performance Analysis of the CONFIDANT Protocol (Cooperation Of Nodes - Fairness In Dynamic Ad-hoc NeTworks),” *Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc’02),* pp. 226-336, 2002.

[11] S. Ganeriwal, L. Balzano and M. Srivastava, “Reputation-based framework for high integrity sensor networks,” *In Proceedings of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks (SAN ’04),* pp. 66-77, 2004.

[12] P. Michiardi and R. Molva, “Core: A Collaborative Reputation mechanism to enforce node cooperation in Mobile Ad Hoc Networks,” in *Communication and Multimedia Security Conference (CMS’02)*, 2002.

[13] A. K. Trivedi, R. Arora, R. Kapoor, S. Sanyal and S. Sanyal, “A Semi-distributed Reputation-based Intrusion Detection System for Mobile Adhoc Networks,” in *arXiv:1006.1956v2*, 2010.

[14] A. T. Rahem and H. K. Sawant, “Collaborative Trust-based Secure Routing based Ad-hoc Routing Protocol,” *International Journal of Modern Engineering Research (IJMER),* vol. 2, no. 2, pp. 95-101, Mar-Apr 2012.

[15] Y. Zhang, L. Xu and X. Wang, “A Cooperative Secure Routing Protocol based on Reputation System

for Ad Hoc Networks," *Journal of Communications,* vol. 3, no. 6, pp. 43-50, November 2008.

[16] S. Madhavi and T. H. Kim, "An Intelligent Distributed Reputation Based Mobile Intrusion Detection System," *International Journal of Computer Science and Telecommunications,* vol. 2, no. 7, October 2011.

[17] A. Pirzada and C. McDonald, "Trust Establishment in Pure Ad-hoc Networks," *Wireless Personal Communications,* vol. 37, pp. 139-163, 2006.

[18] B. Karp and H. T. Kung, "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks," in *In Proceedings of the 6th ACM/IEEE Annual International Conference on Mobile Computing and Networking (MoniCom'00) - pp. 243-254*, 2000.

[19] MEMSIC Inc, "IRIS wireless measurement system," [Online]. Available: http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf.

[20] TinyOS, "TOSSIM simulator," 10 05 2013. [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/TOSSIM.

[21] Daintree Networks, "Sensor Network Analysez (SNA)," [Online]. Available: http://www.daintree.net/sna/sna.php.

[22] K. Gerrigagoitia, R. Uribeetxeberria, U. Zurutuza and I. Arenaza, "Reputation-based Intrusion Detection System for wireless sensor networks," in *Complexity in Engineering (COMPENG)*, 2012.

[23] A. Jsang and R. Ismail, "The beta reputation system," in *Proceedings of the 15th Bled Electronic Commerce Conference (pp. 41-55)*, 2002.

[24] S. Buchegger and J. Y. L. Boudec, "A robust reputation system for mobile ad-hoc networks," in *in Proceedings of P2PEcon*, 2003.

[25] S. Ganeriwal, L. K. Balzano y M. B. Srivastava, «Reputation-based framework for high integrity sensor networks,» *ACM Transactions on Sensor Networks (TOSN),* vol. 4, nº 3, p. 15, 2008.

[26] J. Hui, Ed. and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," Internet Engineering Task Force (IETF), RFC 6282.

[27] S. Raza, S. Duquennoy, T. Chung, D. Yazar, T. Voigt and U. Roedig, "Securing Communication in 6LoWPAN with Compressed IPsec," in *7th IEEE International Conference on Distributed Computing in Sensor Systems*, Barcelona, Spain, 27-29 June 2011.

[28] T. V. a. U. R. S. Raza, "6LoWPAN Extension for IPsec," in *Interconnecting Smart Objects with the Internet Workshop*, Prague, Czech Republic, 25 March 2011.

[29] R. Housley, "Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)," RFC 4309.

[30] D. Whiting, R. Housley and N. Ferguson, "Counter with CBC-MAC (CCM), RFC 3610," September 2003.

[31] S. Kent, "IP Encapsulating Security Payload (ESP)," RFC 4303, 2005.

[32] CertiVox, "Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL)," [Online]. Available: https://github.com/CertiVox/MIRACL. [Accessed 06 06 2013].

[33] Moteiv Corporation, “Tmote Sky – Ultra low power IEEE 802.15.4 compliant wireless sensor module (datasheet),” 13 11 2006. [Online]. Available: http://www.snm.ethz.ch/ snmwiki/pub/uploads/Projects/tmote sky schematic.pdf.

[34] S. Buchegger and J.-Y. L. Boudec, “A Robust Reputation System for Mobile Ad-hoc Networks,” EPFL IC Technical Report IC, 2003.

[35] J. O. Berger, “Statistical Decision Theory and Bayesian Analysis,” in *Springer, second edition edition*, 1985.

[36] A. Srinivasan, J. Teitelbaum and J. Wu, “Drbts: Distributed reputation based beacon trust system,” in *2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing [pp 277-283]*, 2006.

[37] S. Buchegger and J.-Y. L. Boudec, “A Robust Reputation System for Peer-to-Peer and Mobile Ad Hoc Networks,” in *In Proceedings of P2Pecon 2004*, Harvard University, Cambridge MA, USA, June 2004.

[38] C. Tripp, “Impact of security mechanisms in IEEE 802.15.4 sensor operation,” 2009.