

## WSAN4CIP

### Deliverable D3.3

#### **Implementation and testing of dependable networking mechanisms for the WSA4CIP application scenarios**

Editor:	Levente Buttyan, BME
Deliverable nature:	Prototype (P)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	30/04/2011
Actual delivery date:	24/05/2011
Suggested readers:	Experts interested in dependable networking mechanisms
Version:	1.0
Total number of pages:	51
Keywords:	Dependable Protocol Stack, Transport, Routing, MAC, Implementation, Software modules, Component testing

---

#### *Abstract*

This document presents the implementation and component testing results of the WSA4CIP dependable protocol stack including the transport, the routing, and the MAC protocols. For each protocol of the protocol stack, we present a short overview of the operation of the protocol, the software architecture of its implementation, the description of the main modules and interfaces, and examples for usage and configuration of the implementation. Moreover, we present the results of the in-lab tests that we performed independently for the protocols in order to check that they function correctly. The implementations themselves are available for on-line download from the project's SVN repository.

---

---

**Disclaimer**

---

This document contains material, which is the copyright of certain WSAN4CIP consortium parties, and may not be reproduced or copied without permission.

All WSAN4CIP consortium parties have agreed to full publication of this document.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the WSAN4CIP consortium as a whole, nor a certain party of the WSAN4CIP consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

**Impressum**

[Full project title] Wireless Sensor Networks for the Protection of Critical Infrastructures

[Short project title] WSAN4CIP

[Number and title of work-package] WP3 Dependable networking

[Document title] Implementation and testing of dependable networking mechanisms for the WSAN4CIP application scenarios

[Editor: Name, company] Levente Buttyan, BME

[Work-package leader: Name, company] Levente Buttyan, BME

[Estimation of PM spent on the Deliverable] 26 PM

**Copyright notice**

© 2011 Participants in project WSAN4CIP

## Executive summary

This document presents the implementation and component testing results of the WSAN4CIP dependable protocol stack including the transport, the routing, and the MAC protocols. For each protocol of the protocol stack, we present a short overview of the operation of the protocol, the software architecture of its implementation, the description of the main modules and interfaces, and examples for usage and configuration of the implementation. Moreover, we present the results of the in-lab tests that we performed independently for the protocols in order to check that they function correctly. The implementations themselves are available for on-line download from the project's SVN repository.

The requirement to transmit video over multiple hops in the EDP demo scenario led to the need of a suitable reliable transport protocol. The DTSN protocol was selected for this purpose, whose basic service had been developed in a precursor project (UbiSec&Sens). Within the context of WSAN4CIP, a multimedia extension was added and implemented, so that DTSN can now cope with delay-sensitive data. The multimedia extension allows DTSN to reset a multimedia session at any time, purging the intermediate node caches of outdated packets. This is useful guarantee that video frame delay bounds are respected at the expense of decreased reliability. However, it is expected that this decrease in reliability will result on a graceful degradation of video quality, and can be used by the application to differentiate between frames with different degrees of importance. Moreover, security mechanisms were also added and implemented in the context of the WSAN4CIP Project. These mechanisms provide authentication and integrity guarantees, especially for the control packets that are need to be processed by intermediate nodes (e.g. ACK and NACK packets). The target operating system for the DTSN implementation was Linux, and the implementation was tested in an Ethernet LAN network.

In WSAN4CIP, we decided to adopt the IPv6 Routing Protocol for Low-power and Lossy Networks (RPL), which was designed to work in environments where the links may be unstable and the participating nodes may have power, computational, and storage constraints. The operation of RPL is based on the principles of distance vector routing and on the notion Directed Acyclic Graphs (DAG). In particular, through the exchange of distance information between neighbouring nodes, the network maintains one or more Destination Oriented DAGs (DODAGs), where the destination is the DODAG root, which is typically a base station. Upward and downward routes are selected along the edges of these DODAGs. Point to point communication is also possible in RPL using the upward and downward routes. In addition to the core protocol, we added and implemented two security extensions: DODAG version number authentication and local key exchange. The RPL protocol was implemented in standard C in order to use one implementation for both demo environment (the EDP demo uses Linux while the FWA demo uses TinyOS). The implementation was tested in both environments; on Linux, we used a test network of 4 nodes including the target hardware, while on TinyOS, we used a 5 node network within the TOSSIM simulation environment.

The implemented MAC protocol is a completely distributed multichannel TDMA scheme. The packet transmission between neighboring nodes happens during superframes of predefined length. Each superframe contains one contention-based period and several assured collision free slots. The transmissions in each part of the superframe happen on different radio channels. The channel hopping pattern is different between subsequent superframes. The protocol is fully distributed and does not require either centralized or manual configuration. The transmitted messages are cryptographically authenticated. Neither the establishment of the channel-hopping schedule nor the establishment of the time division schedule requires an exchange of control information except for the transmission request from a particular node in the time division schedule case. The target operating system for the MAC protocol implementation was TinyOS. As the hardware to be used in the FWA demonstrator was not available at the time of the implementation, we performed the testing on the hardware that was available at the partner developing the protocol (i.e., LTU), and we measured processing time and energy consumption.

## List of authors

<b>Company</b>	<b>Author</b>
INOV	Antonio Grilo
BME	Levente Buttyan
BME	Boldizsar Bencsath
BME	Tamas Holczer
BME	Laszlo Dora
LTU	Evgeny Osipov
LTU	Laurynas Riliskis

## Table of Contents

Executive summary .....	3
List of authors.....	4
Table of Contents .....	5
List of figures .....	6
List of tables .....	7
Abbreviations .....	8
1 Implementation and testing of the transport layer protocol.....	9
1.1 Implementation .....	9
1.1.1 Software architecture .....	9
1.1.2 Modules and interfaces .....	11
1.1.3 Configuration and usage examples .....	11
1.1.4 Access to the implementation .....	11
1.2 Component testing .....	11
1.2.1 Performance Tests with Enabled Cache .....	11
1.2.2 Performance Tests with Disabled Cache .....	12
1.2.3 Performance Tests with Enabled Security .....	13
1.2.4 Throughput Limit Tests .....	15
2 Implementation and testing of the routing protocol .....	17
2.1 Implementation .....	17
2.1.1 Software architecture .....	17
2.1.2 Modules and interfaces .....	18
2.1.3 Operation .....	21
2.1.4 Configuration and usage examples .....	21
2.1.5 Access to the implementation .....	24
2.2 Component testing .....	24
2.2.1 Performance testing in Linux.....	24
2.2.2 Performance testing in TinyOS (TOSSIM) .....	25
3 Implementation and testing of the MAC protocol.....	27
3.1 Implementation .....	28
3.1.1 Software architecture .....	29
3.1.2 Modules and interfaces .....	30
3.1.3 Operation .....	31
3.1.4 Configuration and usage examples .....	33
3.1.5 Access to the implementation .....	36
3.2 Component testing .....	36
3.2.1 Testing on own hardware.....	36
3.2.2 Lessons Learned .....	39
3.2.3 On Assessing the Performance Of Hardware and Software Components .....	39
4 References .....	40
Annex A DTSN Implementation Examples .....	41
A.1 DTSN configuration file (dtsn.conf).....	41
A.2 DTSN Header File with Primitives (dtsn.h).....	42
A.3 DTSN Type Definition file (dtsn_types.h).....	43
A.4 DTSN Receiver (dtsn_recv.c) .....	46
A.5 DTSN Sender (dtsn_send.c).....	49

## List of figures

Figure 1. Software architecture of the DTSN implementation.....	10
Figure 2. DTSN shared library architecture. ....	11
Figure 3. Average number of transmitted data packets per segment, with caching enabled.....	11
Figure 4. Overhead ratio with caching enabled.....	12
Figure 5. Average number of transmissions per segment and average number of links crossed by each data packet, when caching is disabled. ....	12
Figure 6. Data packets lost in each link, with caching disabled. ....	13
Figure 7. Number of data packets lost, normalized with the total number of transmitted segments.....	13
Figure 8. Average number of data packet transmission per transmitted segment. ....	14
Figure 9. Total transmitted bytes in all computers, during the five tests.....	14
Figure 10. Data volume distribution, with security enabled and 512 bytes payload.....	15
Figure 11. DTSN overhead ratio. ....	15
Figure 12: Software architecture of the RPL implementation.....	18
Figure 13: Encapsulation of different protocol messages in the Linux based implementation.....	20
Figure 14: Test topology for the Linux based implementation.....	24
Figure 15: Test topology for the TinyOS based implementation.....	26
Figure 16. Establishment of the channel hopping and the time division patterns at a glance. The case when a source has only one packet to transmit to the destination. ....	27
Figure 17: Software architecture of LTU-MAC.....	30
Figure 18: Software architecture of the ActiveMessageLayerP component. ....	30
Figure 19: Architecture of the core functionality of the LTU-MAC.....	31
Figure 20: Structure of the superframe.....	31
Figure 21: Example of the broadcast subframe after having sent one RTS and received two RTS messages.....	32
Figure 22: Example of the unicast subframe after having sent an RTS and received two RTS addressed to this node. ....	32
Figure 23: Synchronization on the sending node. ....	32
Figure 24: Synchronization on the receiving node. ....	32
Figure 25: Current consumption including times before and after bootstrapping. ....	37
Figure 26: Current consumption of one idle superframe.....	37
Figure 27: Current consumption of one superframe when sending (2). ....	38

## List of tables

Table 1. Data rates and transfer times for the throughput limit tests.....	15
Table 2: Success delivery ratio between the nodes .....	26
Table 3: Summary of the properties of the implemented protocol that differ from those of the original design. .....	33
Table 4: Execution time of different phases in LTU-MAC protocol. ....	36
Table 5: Execution time of the <i>Alarm</i> functions.....	36
Table 6: Summary of the expected and achieved performance. ....	38

## Abbreviations

ACK	Acknowledgement
API	Application Programming Interface
DAG	Directed Acyclic Graph
DAO	Destination Advertisement Object
DIO	DODAG Information Object
DIS	DODAG Information Solicitation
DODAG	Destination Oriented DAG
DTSN	MAC Acknowledgement
EAR	Explicit ACK Request
FTP	File Transfer Protocol
GCC	Gnu C Compiler
ICMP	Internet Control Message Protocol
IP	Internet Protocol
NACK	Negative ACK
MAC	Medium Access Control
MPEG	Moving Picture Experts Group
OF0	Objective Function 0
RPC	Remote Procedure Call
RPL	IPv6 Routing Protocol for Low-power and Lossy Networks
RTS	Request To Send
TCP	Transport Control Protocol
TDMA	Time Division Multiple Access
TFTP	Trivial File Transfer Protocol
TOSSIM	TinyOS SIMulation environment
UDP	User Datagram Protocol
WSN	Wireless Sensor Network

# 1 Implementation and testing of the transport layer protocol

The requirement to transmit video over multiple hops in the EDP demo scenario led to the need of a suitable reliable transport protocol. The selected transport protocol was DTSN [1], whose basic service had been developed in project FP6 UbiSec&Sens. Within the context of WSAN4CIP a multimedia extension was added, so that DTSN can now cope with delay-sensitive data. Security mechanisms were also added to the protocol in the context of the WSAN4CIP Project.

DTSN is a reliable generic upstream transport protocol designed specifically for WSNs. Its objective is to provide reliable data delivery while minimizing the energy consumption. Data is transmitted in the context of sessions that are explicitly created by the application. These sessions are internally implicitly created and are univocally identified by a session number, an application id, the source address and the destination address. Every packet carries this identification in its header, along with a sequence number that orders the packet in the sequence, allowing the reception of out of order packets. Recovery of lost packets is achieved by a selective repeat mechanism. DTSN uses both accumulative ACKs and selective NACKs which also act as an accumulative ACK since they include the sequence number of the last packet successfully received in order. These packets are sent by the receiver only when explicitly requested by the source, through an explicit acknowledgment request (EAR). This request is a bit that is included in data packets or it is sent as a control packet. The EAR is sent in many situations, for example, after a certain number of new packets have been sent. A timer is used to detect the loss of an EAR or its answer, triggering another request.

Even though only the source and the destination create control packets, the intermediate nodes also participate in the recovery of lost packets. These nodes cache data packets that cross them, so that later, if needed, they can retransmit them. To accomplish this, they intercept NACK packets and retransmit the packets requested in the bitmap that are found in cache. The bitmap of the NACK is modified before forwarding it so that the packets sent are not requested anymore. In this way, the packets can be retransmitted from a point closer to the link where it was previously lost, reducing the retransmission distance.

The multimedia extension allows DTSN to reset a multimedia session at any time, purging the intermediate node caches of outdated packets. This is useful guarantee that video frame delay bounds are respected at the expense of decreased reliability. However, it is expected that this decrease in reliability will result on a graceful degradation of video quality, and can be used by the application to differentiate between frames with different degrees of importance (e.g. 'I', 'P' and 'B' frames in an MPEG stream).

Regarding the security extensions, they provide authentication and integrity guarantees, especially for the control packets on which the source-received synchronization relies, which must be processed by intermediate nodes (e.g. ACK and NACK packets).

A detailed description of the multimedia and security extensions of DTSN can be found in Deliverable 3.2 [2].

## 1.1 Implementation

This section describes the implementation of the DTSN protocol in the context of WSAN4CIP. The target OS was LINUX, since a light version of the latter can be run by the Silex SX-560 node [3] used in the EDP demo. The used C library, *uClibc*, is much smaller than the traditional *glibc* and was developed specifically for embedded systems. It supports all POSIX functions (including threads) and shared libraries, which turned out to be useful for this implementation.

### 1.1.1 Software architecture

The general architecture of the DTSN implementation is depicted in Figure 1. The core of the protocol runs in a daemon process. This process implements all the functionalities of the protocol since all the nodes – source, receiver and intermediate ones - run the same process. The communication between nodes is done through UDP sockets. The daemon sets up, on start up, a UDP port through which all exchanges with the client applications take place.

The daemon is constituted by a single thread, running synchronously. It has to handle incoming packets, incoming local connections, incoming RPC requests and timeouts. The sending and receiving windows of sessions are mapped to circular buffers. Both the sender and the receiver have the windows synchronized through the control packets. NACK and ACK allow the sender's window to advance and the EAR packet specifies the last sent packet and might correct the receiver's window. This mechanism implements a simple flow control. Also a simple routing module was developed for testing purposes, to simulate packet loss and to perform arbitrary packet routing not dependent on the IP addresses. Many protocol's parameters, as timeouts, cache sizes and windows sizes are configurable through a configuration file that is loaded on startup.

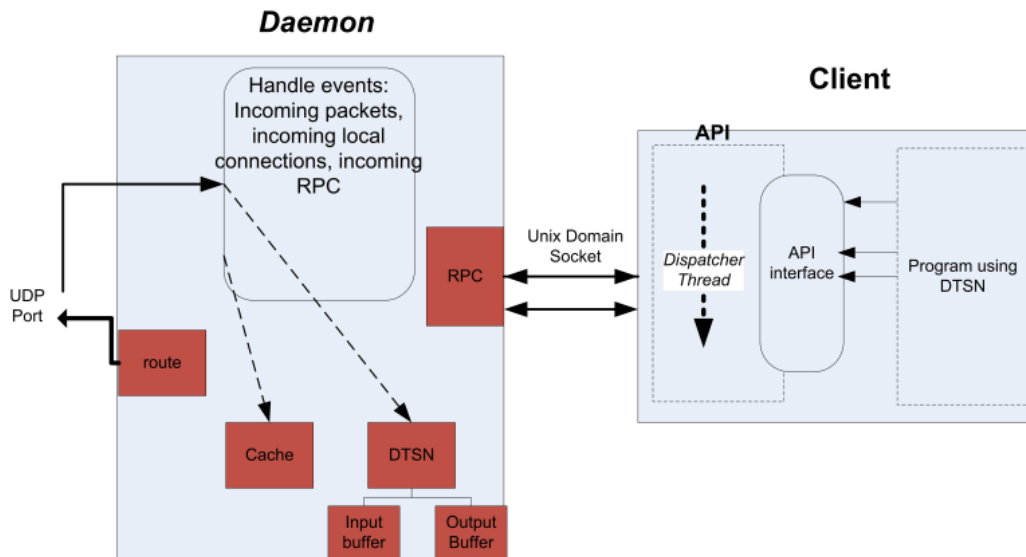
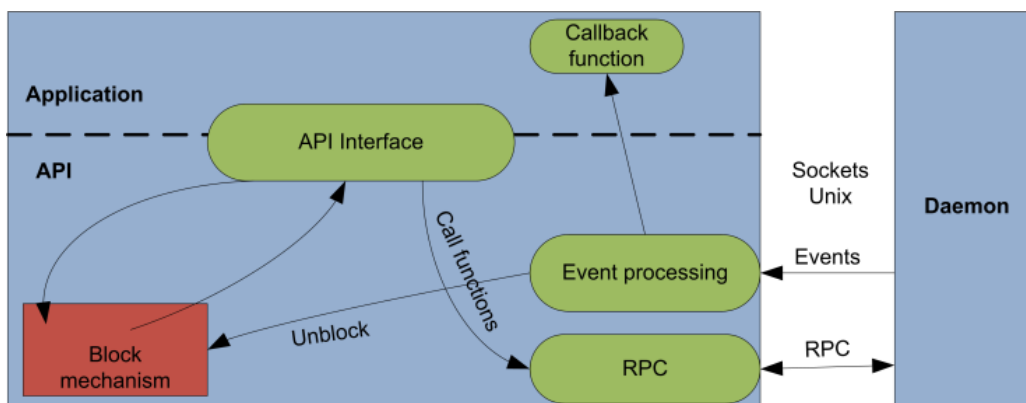


Figure 1. Software architecture of the DTSN implementation.

The functions performed by the different modules are the following:

- **DTSN:** Implements the source and destination DTSN protocol and Finite State Machine.
- **Cache:** Implements the DTSN intermediate node functionality.
- **RPC:** Implements the Remote Procedure Call (RPC) communication between the DTSN client application and the DTSN *daemon*. This RPC module is build on top of a UNIX domain socket.
- **Route:** Static routing module embedded for testing purposes.

An API shared library was developed to create a simple interface for applications that want to use DTSN. This interface is described in detail in Deliverable 3.2 [2]. The library communicates with the daemon through two UNIX sockets: one for remote procedure calls (RPC) and another for transmission of asynchronous events to the application. This model requires having a thread continuously waiting for events, to transmit them to the application and to release blocked threads based on them (see Figure 2).



**Figure 2. DTSN shared library architecture.**

### 1.1.2 Modules and interfaces

The API of the DTSN implementation is described in detail in Deliverable 3.2 [2].

### 1.1.3 Configuration and usage examples

Annex A presents example configuration and application files, together with the required header files that define the API interface. All DTSN primitives invoked in the applications bear the prefix 'DTSN\_'. For an exhaustive description of the primitives and parameters, the reader is referred to Deliverable 3.2 [2].

### 1.1.4 Access to the implementation

The DTSN implementation is available from the WSAN4CIP project SVN server<sup>1</sup>.

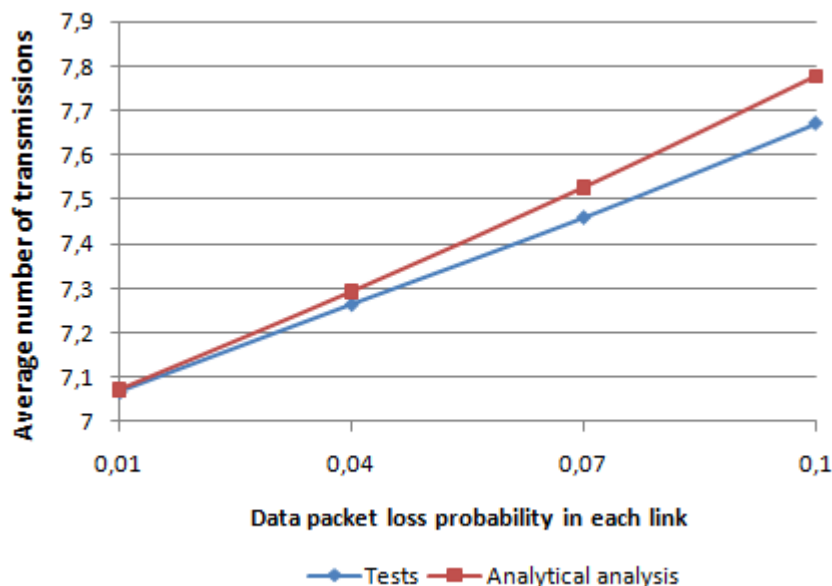
## 1.2 Component testing

The DTSN implementation was tested in an Ethernet LAN network. A routing module was added to the program, which routes packets through 7 hops and simulates packet loss in each link, based on a bit error probability. A 1,95 MB file was transferred in this simulated WSN and statistics were saved in each node, allowing us to draw many conclusions about the protocol's performance. The performance of the *daemon* and shared library were also evaluated in a scenario of communication between a PC and a Silex node.

Additional test procedures and results are to be presented in Deliverable 5.1.

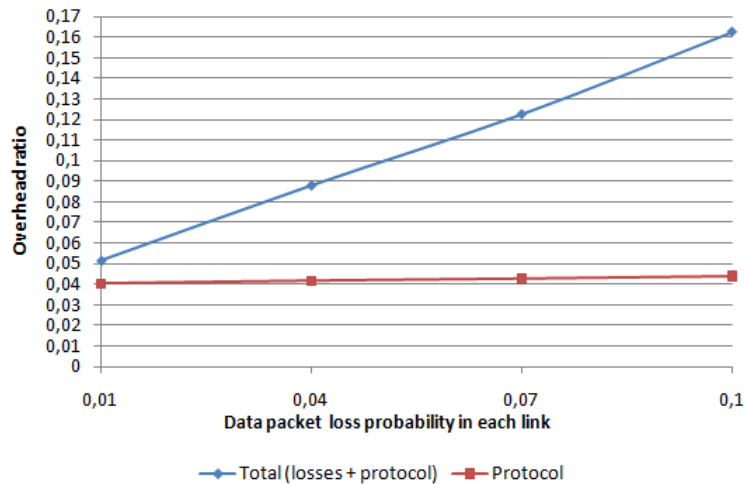
### 1.2.1 Performance Tests with Enabled Cache

Figure 3 shows the average data packet transmissions made by all the nodes to send each data segment from the source to the final destination. It also compares the values acquired in the test with those calculated using the analytical model described in Deliverable 3.2 [2]. As can be seen, the analytical model constitutes a fairly good prediction for the real performance.

**Figure 3. Average number of transmitted data packets per segment, with caching enabled.**

<sup>1</sup><https://svn.wsan4cip.eu/>

Figure 4 shows the impact of the packet loss and of the protocol overhead in the transmitted data volume. These values, especially the protocol overhead, are strongly dependent on the data packets' payload. With the used payload (512 bytes) DTSN overhead is very reasonable, staying below 5%. It increases slightly when the loss probability increases because the number of control packets transmitted also increases.

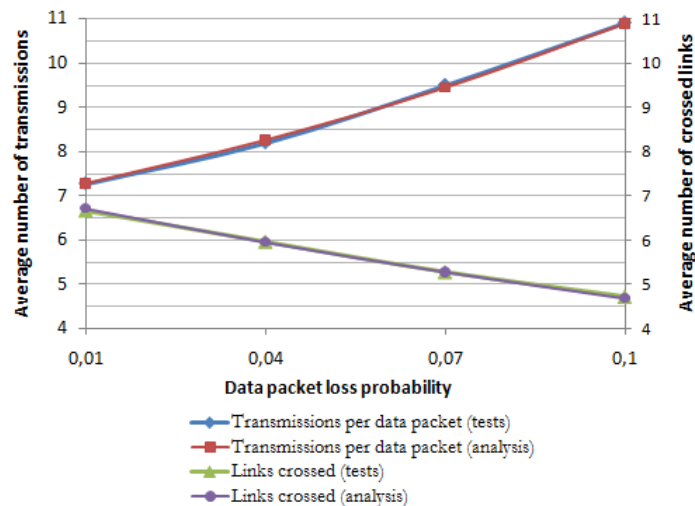


**Figure 4. Overhead ratio with caching enabled.**

### 1.2.2 Performance Tests with Disabled Cache

The same tests were repeated, but this time with caching disabled in the intermediate nodes.

Figure 5 illustrates the average number of data packets transmissions per transmitted segment and the average number of traversed links by the data packets before being lost or successfully received by the destination, attained both using the model and by the tests. Both results are, once again, very close.



**Figure 5. Average number of transmissions per segment and average number of links crossed by each data packet, when caching is disabled.**

In Figure 6 we can observe more details regarding the data packets lost in each link. Unlike the case where caching is enabled, nodes closer to the source are crossed by more packets, and as a result we can observe a higher packet loss in links closer to the source.

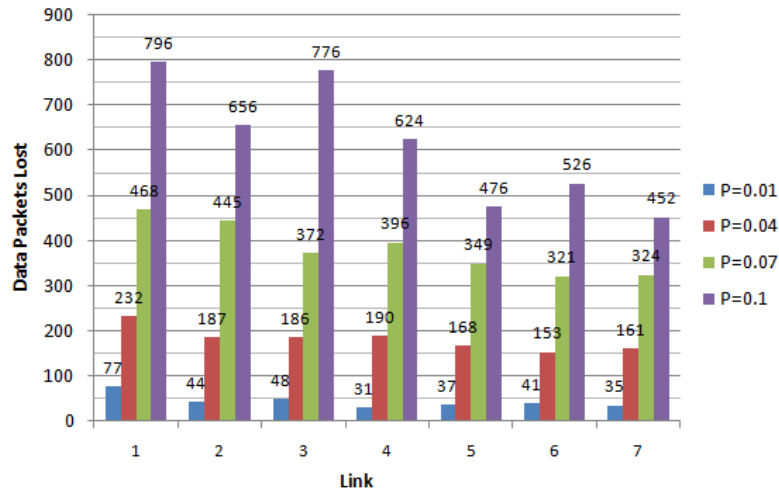


Figure 6. Data packets lost in each link, with caching disabled.

### 1.2.3 Performance Tests with Enabled Security

The same file transfer test was performed with security and caching enabled. The bit error probabilities simulated were the same as the last sections, but this time three different payloads were used (200, 350 and 512 bytes) to better understand its effect in the protocol’s performance.

Figure 7 shows the total data packet loss for the 5 different tested situations, normalized by the number of segments sent (tests with smaller payloads have more segments). The smaller the payload is, the smaller is the number of losses per segment, since the probability of losing each packet decreases. Note that with security there are more losses as the packets are larger. Without caching, there is also more packet loss because the retransmitting distance is larger, increasing the number of links where the retransmitted packet can be lost. Therefore, without caching, not only it is more expensive energetically to retransmit packets, but there are also more packets lost.

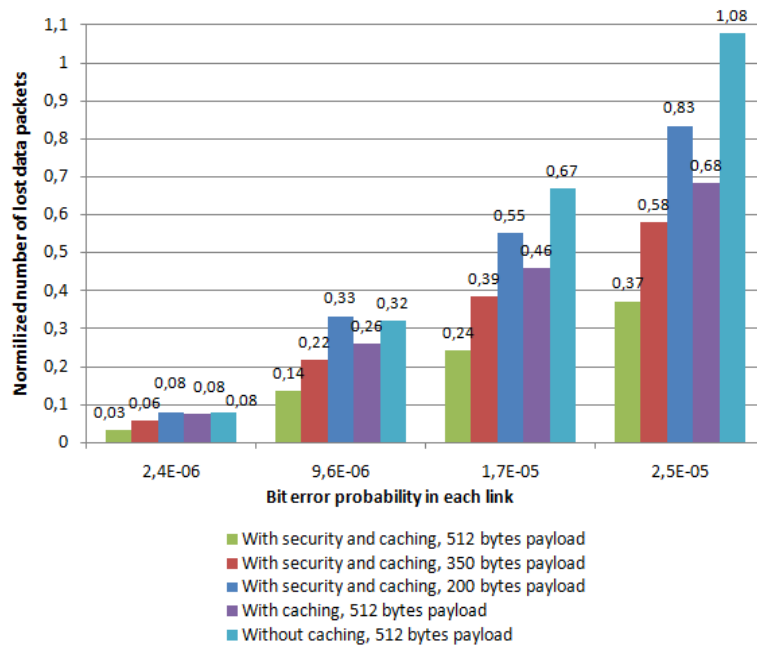
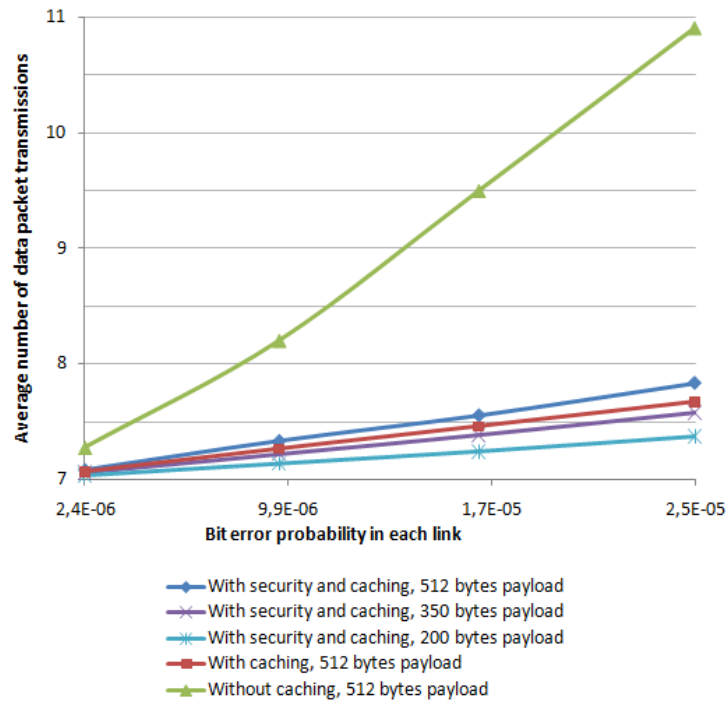


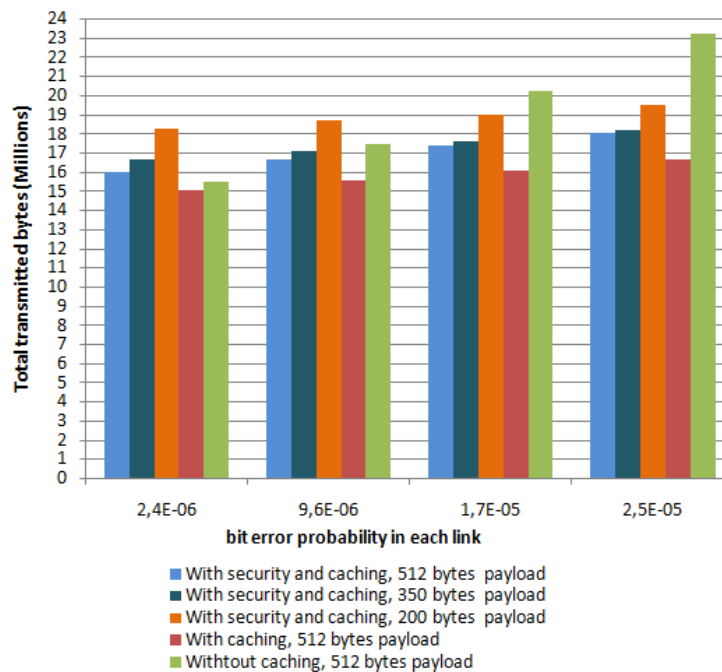
Figure 7. Number of data packets lost, normalized with the total number of transmitted segments.

Figure 8 compares the average number of necessary data packet transmissions to successfully send a segment to the final destination. The use of caching greatly reduces transmissions, saving energy.



**Figure 8. Average number of data packet transmission per transmitted segment.**

Figure 9 shows the total data volume for all 5 tested situations. This is the full amount of data transmitted, including control packets. Essentially, this is the value we are interested in minimizing to reduce energy consumption. Enabling security, not only increases packet size, but also, for the same reason, turns packet loss more likely. For low error probability, using security has lower performance than the situation with cache and security disabled. With cache, the total amount of bytes transmitted is much less dependent on the networks conditions. From the 3 situations with cache and security enabled, the most efficient way to transmit data is using 512 bytes payload. This result matches the one obtained from the analysis.



**Figure 9. Total transmitted bytes in all computers, during the five tests.**

The data volume corresponding to control packets is relatively small, as it can be observed in Figure 10. On the other hand, when security is enabled, the total protocol overhead is high. Figure 11 shows the overhead ratio for the tested situations. Once again, the values are highly dependent on the data packets payload size.

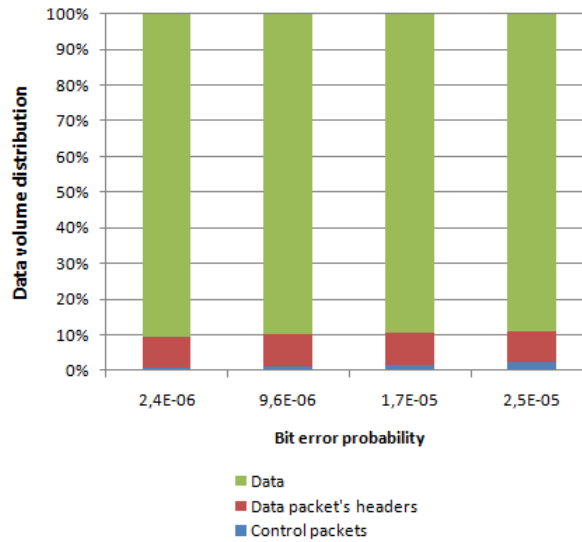


Figure 10. Data volume distribution, with security enabled and 512 bytes payload.

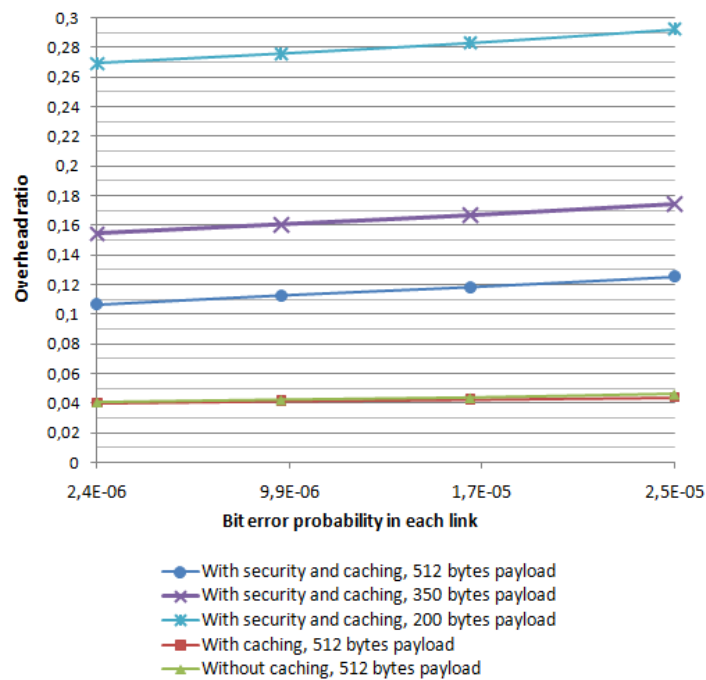


Figure 11. DTSN overhead ratio.

### 1.2.4 Throughput Limit Tests

The data transfer rate between a PC and a Silex Sx-560 was tested using the implemented transport protocol. Both computers were connected through a 1-hop Ethernet connection and with the appropriate configuration, there is no buffer overflow and hence no packet loss occurs. In this situation, the maximum data rate is limited by the processing capacity of the Silex SX-560. A 1,95MB file transfer was used for these tests.

Table 1 compares the performance of the implemented DTSN with FTP and TFTP in transferring the file in both ways.

Table 1. Data rates and transfer times for the throughput limit tests.

	<b>PC → Silex</b>	<b>Silex → PC</b>
<b>DTSN</b>	10,36 s (1,51 Mbps)	8,03 s (1,94 Mbps)
<b>DTSN with security</b>	10,97 s (1,42 Mbps)	8,22 s (1,90 Mbps)
<b>DTSN without delivery to app.</b>	4,69 s (3,33 Mbps)	8,03 s (1,94 Mbps)
<b>TFTP (over UDP)</b>	12,0 s (1,30 Mbps)	79.9 s (0,20 Mbps)
<b>FTP (over TCP)</b>	3,0 s (5,20 Mbps)	5,17 s (3,02 Mbps)

Despite TCP having better transfer times in these tests, they do not reflect the performance in high loss multi-hop networks with large delays as WSNs, since the results only depend on the processing speed.

Benchmarking DTSN, we found out that the bottleneck is the communication between the API and daemon, which creates a considerable delay because of the use of a *mutex* to guarantee exclusive access, synchronization and data transfers. In table 2 is also included the data rate achieved when no data is delivered to the API. As expected, security does affect the data rate, however the performance hit is very low.

## 2 Implementation and testing of the routing protocol

The Routing Protocol for Low-power and Lossy Networks (RPL) [4] is designed for wireless sensor networks. In such networks, the links may be unstable and the participating nodes may have power, computational, and storage constraints.

Usually in sensor networks, there is one or a few special nodes called base stations. A base station is responsible for collecting the data measured by the sensor nodes and to control these nodes. Therefore, the destination or the source of most of the flows is a base station. RPL is designed according to this philosophy. There are upward routes directed from sensor nodes to the base station and there are downward routes directed from the base station to the sensor nodes. The upward routes and downward routes are established independently. The support for downward routes is optional, thus, a network can operate according to the RPL standard even if only sensors can send messages to the base station. RPL also supports the communication between two regular sensor nodes by combining upward and downward routes.

The operation of RPL is based on the principles of distance vector routing and on the notion Directed Acyclic Graphs (DAG). In particular, through the exchange of distance information between neighbouring nodes, the network maintains one or more Destination Oriented DAGs (DODAGs), where the destination is the DODAG root, which is typically a base station. Upward and downward routes are selected along the edges of these DODAGs. Point to point communication is also possible in RPL using the upward and downward routes. For a more detailed description of the RPL protocol and the extensions we suggested, see Deliverable 3.2 [2] and our Internet Draft [5].

### 2.1 Implementation

The RPL core protocol was implemented in standard C in order to use one implementation for both demos. The interfaces are different for the Linux based demo and the TinyOS based demo. The interfaces will be discussed after introducing the main software components.

#### 2.1.1 Software architecture

The software architecture is introduced in Figure 12.

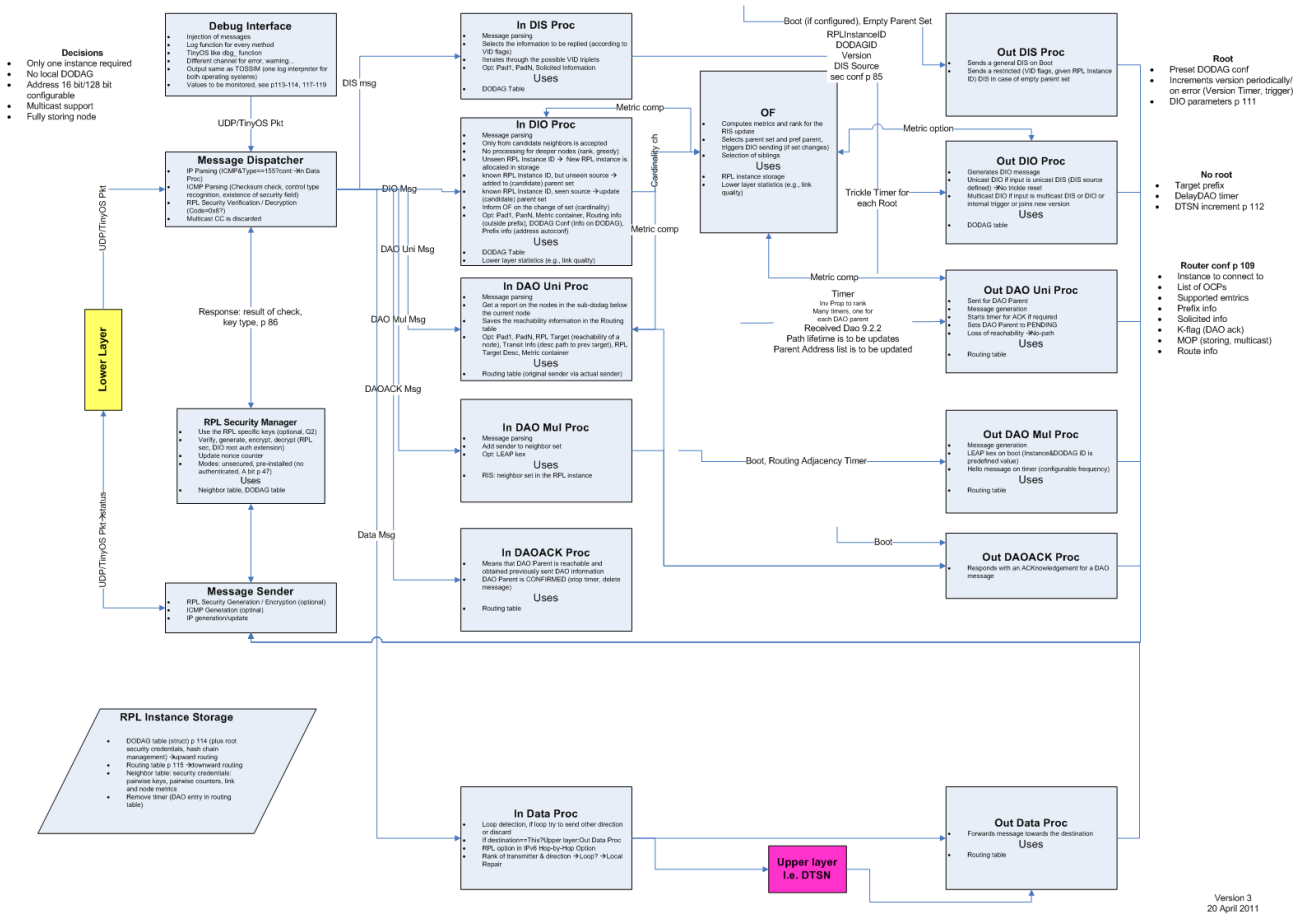


Figure 12: Software architecture of the RPL implementation

Most of the modules can be categorized into three main parts (three columns on the figure): general message handling (sending and receiving), incoming control and data message handling, outgoing control and data message creating. The modules and the interfaces are introduced in the next section.

### 2.1.2 Modules and interfaces

In this section, the modules are briefly described and the interfaces for the two operating systems are described as well:

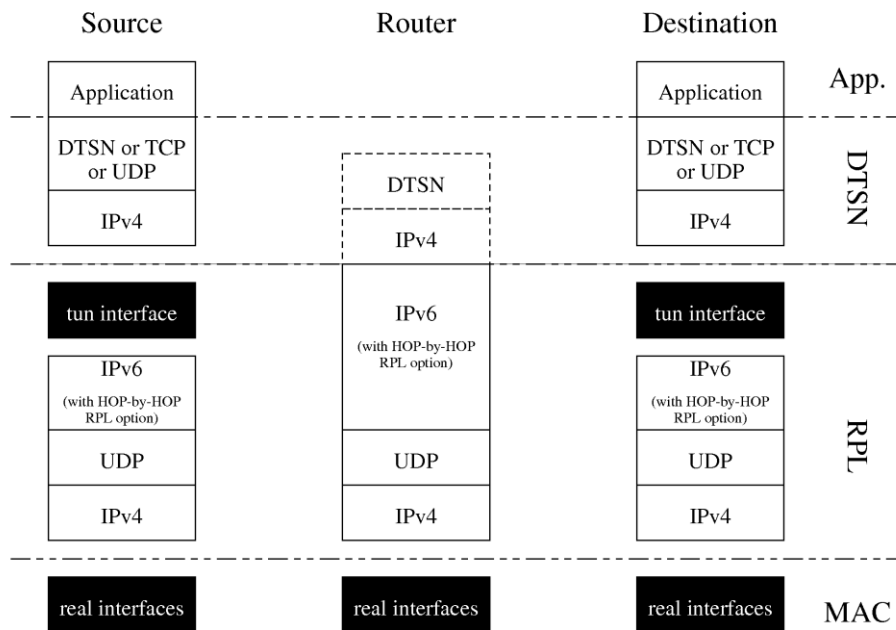
- **MSG\_DISPATCHER:** This module is responsible for receiving all the incoming messages. It checks if the packet is really sent to this node and the message format is correct. Correct messages are dispatched to the appropriate module.
- **MSG\_SENDER:** This module is responsible for sending messages. It creates the correct IP header and asks the ICMP module to create the ICMP header if needed. The message is then forwarded to the IP/TINYOSM module to send it. In TinyOS, this module implements a sending buffer as the standard AMSender module in TinyOS can only handle one message at a time.
- **DTSN:** This module is responsible for giving an interface for upper layer applications for message manipulation, when the data message is only forwarded by the node. This manipulation can be simple inspection (IDS), or modification (DTSN).
- **IP:** This module is for Linux use only. The module provides a socket interface for upper layer applications, and encapsulates the messages into UDP packets for delivery through the lower layer. The interfaces are further described below.
- **TINYOSM:** This module is for TinyOS use only. The module provides AMSend and Receive interfaces for the upper layer application and uses AMSend and Receive interfaces for sending and receiving messages through the lower layer. The module also provides an interface for the lower layer for getting the keys this layer exchanged. The interfaces are further described below.

- **ICMP:** This module creates and verifies the ICMP headers of the RPL control messages.
- **LOGGER:** This module provides a convenient logging service for TOSSIM, Linux and TinyOS.
- **ROOT:** This module is only used, if the node is a DODAg Root. It initializes the Root specific values at initialization, and is responsible for global repairs.
- **SECURITY:** This module is responsible for the DIO authentication and the key exchange between the nodes. In DIO authentication it creates and handles the authentication options added to DIO messages. In key exchange, it creates and handles the key exchange options added to DIO and DIS messages.
- **DIS (DIS\_IN, DIS\_OUT):** This module is responsible for incoming and outgoing DIS messages. DIS messages are sent if the node does not know the configuration of the network. On reception of a DIS message, the node answers with a DIO message containing the DODAG configuration option. If a cluster key is received in an option, then this cluster key is checked and stored.
- **DIO (DIO\_IN, DIO\_OUT):** This module is responsible for generating and processing DIO messages. The DIO messages generate the upward routing entries in the nodes. The module is also responsible for the trickle timer which controls the frequency of DIO message generation. If the incoming message contains some security related options, then the options are handled by the SECURITY module.
- **DAO (DAO\_IN, DAO\_OUT):** This module is responsible for generating and processing DAO messages. The DAO messages generate the downward routing entries in the nodes. This module is also responsible for the route lifetime of the node. If the route lifetime is close to its end, the module can send a refreshing DAO message.
- **DAO\_ACK (DAO\_ACK\_IN, DAO\_ACK\_OUT):** This module is closely related to the DAO module. It sends back DAO-Ack messages if required. On reception of a correct DAO-Ack message, it informs the DAO module, that the message is acknowledged (so it should not be repeated).
- **DATA\_PROC (DATA\_PROC\_IN, DATA\_PROC\_OUT):** This module is responsible for forwarding the data packets to the destination. The source and destination of the data packet can be the node itself or other nodes. Multicast packets are also handled by the node.
- **OF0:** This module realizes the objective function used by the routing. The objective function is a hop count based metric. This module can be replaced by another module realizing a more sophisticated metric.

The interfaces are different for the two operating systems, so they are described separately.

### 2.1.2.1 Interfaces for Linux

The encapsulation of the different protocol messages are showed in Figure 13.



**Figure 13: Encapsulation of different protocol messages in the Linux based implementation**

RPL has been implemented such that it can be executed in user space similarly to the olsrd implementation (<http://www.olsr.org/>). Therefore, we used the tun interface to intercept the IPv4 packets and embed them into RPL specific IPv6 packets. In order to transport the IPv6 packets, neighboring nodes are connected through (IPv4) UDP sockets which means that IPv6 packets are further embedded into UDP which is embedded in IPv4 packets.

Due to this architecture, the original (upper layer) IPv4 message contains the real source and destination addresses in IPv4 format. These are the so called main addresses of the RPL nodes. The IPv6 packet contains the same addresses, but in IPv6 format, which is the original IPv4 address filled with zeros in the beginning (our IPv4-RPL IPv6 conversion). Finally, the lower IPv4 layer contains the source and destination addresses of the neighbouring nodes (real IPv4 interface addresses), which changes hop by hop. They are called local address. The main and local addresses are independent; they do not need to be on the same subnet. (e.g. one node might have multiple local addresses if using multiple interfaces)

The RPL nodes receive information from each other through link local broadcast messages (UDP broadcasts). This is utilized to learn the bindings between the main (RPL node address) and local addresses (IPv4 interface address) of neighbouring nodes and the interface where the neighbours can be reached. The implementation handles multiple interfaces.

The above described situation might look complicated, but it provides a flexible approach to deal with IP numbering, routing, etc.

The implementation requires the tun interfaces to be used. Tun interface is a kernel module which must be loaded to use it. If this is not compiled into the kernel, it has to be loaded after the device has been booted, before the first run of the RPL.

The Netlink/Tun interface is a virtual network interface. After proper setting of our RPL implementation, each packet going (routed) to the tun interface is received by the RPL daemon and transmitted accordingly, while at the endpoint, the RPL daemon puts the payload IP packets to the destination tun interface enabling any user space program to use RPL transparently.

Tun interface can be loaded by the following command:

```
insmod tun
```

One can check if it has been already loaded by the following command:

```
lsmod | grep tun
```

Note, that the SX-560 mote default environment does not have a compiled tun.ko module on the device, so it has to be compiled in the buildroot environment with the appropriate GCC version.

Our binary tun.ko compiled version for the “standard” SX-560 firmware can be installed in binary form.

During the startup, the RPL daemon initializes a new tun device, it turns it to “UP” mode, and sets it’s IP address to the RPL node address. However, the RPL daemon does not route any subnets to this tun interface, therefore, before communicating with other nodes, the appropriate routing should be set up while the RPL daemon is running.

### 2.1.2.2 Interfaces for TinyOS

The TinyOS implementation of the RPL protocol uses the standard AMSend and Receive interfaces. It also provides the same interfaces for the applications, but unlike the normal operation, here not only neighbouring nodes can be addressed, but distant nodes as well. The implementation also provides an interface for the lower layer to access the keys it created. The interface provides the following commands:

```
// getPairwiseKey Returns a pointer to a key shared with a neighbor
with the given address.
// NULL if the key does not exist.
command void* getPairwiseKey(am_addr_t address);

// getClusterKey Returns a pointer to a key which is the
cluster/broadcast key of a neighbor with the given address.
// NULL if the key does not exist.
command void* getClusterKey(am_addr_t address);

// getMyClusterKey Returns my cluster/broadcast key
// NULL if the key does not exist.
command void* getMyClusterKey();
```

### 2.1.3 Operation

To have an operational network, the network must contain at least one root node. This node starts the route discovery by starting the trickle timer which leads to sending DIO messages. An ordinary node receiving a DIO message joins the network (if the DIO is correct) and starts its trickle timer which leads to sending DIO messages. A node joined to the network can send a DAO message. A node receiving a DAO message from a node below can populate its routing table with the sender. When the routing tables are populated correctly, data packets can be sent to unicast and multicast addresses.

### 2.1.4 Configuration and usage examples

The usage and configuration is different for Linux and TinyOS so they are described separately.

#### 2.1.4.1 Configuration and usage for Linux

Example configuration for a root node:

```
verbose_mode = false;
log_filename = "rpl.log";

tun_interface = "tun10";
tun_address = "1.2.3.254";
#lower_layer_interface = "eth1:3";
lower_layer_interfaces = ("eth1:3", "eth1:13");

rpl_instance_id = 0;
```

```
dodag_root_mode = true;
dodag_root_settings:
{
    rpl_instance_id = 0;
    rank = 256;
    mode_of_operation = 2; /* STORING_WITHOUT_MULTICAST */
    grounded = true;
    preference = 0;
    authentication_enabled = false;
    path_control_size = 0;
    DIO_interval_doublings = 20;
    DIO_interval_minimum = 3;
    DIO_redundancy_constant = 10;
    max_rank_increase = 0;
    min_hop_rank_increase = 256;
    objective_code_point = 0;
    default_lifetime = 24;
    lifetime_unit = 3600;
    dioauth_key = "ec.key";
    dioauth_key_password = "";
    hash_chain_length = 250;
    hash_chain_initvalue = "kiskacsá";
};
```

The usage is quite simple:

```
ifconfig eth1:13 10.11.1.254 netmask 255.255.255.0
./rpl_router -c conf_shamir.cfg &
sleep 1
ip route add 1.2.3.0/24 dev tun10
tail -f rpl.log
```

After starting the above script, the node can be accessed using the 1.2.3.254 IP address.

The configuration for a router node is quite similar to the root configuration:

```
verbose_mode = false;
log_filename = "/tmp/rpl.log";

tun_interface = "tun0";
tun_address = "1.2.3.97";
lower_layer_interfaces = ("eth0:1", "eth0:2");

rpl_instance_id = 0;

dodag_root_dioauth_settings = (
    {dodagid      = "1.2.3.254";   dioauth_key    = "ec_pub.key";
    hash_chain_rootvalue = "kiskacsá";   hash_chain_length = 250;
    version_number_root = 240;}
);
```

The usage is quite similar as well:

```
ifconfig eth0:1 10.11.1.97 netmask 255.255.255.0
ifconfig eth0:2 10.11.2.97 netmask 255.255.255.0
./rpl_router -c conf_silex.cfg &
sleep 1
ip route add 1.2.3.0/24 dev tun0
tail -f /tmp/rpl.log
```

After starting the above router, each node can ping each other on 1.2.3.254 and 1.2.3.97.

### 2.1.4.2 Configuration and usage for TinyOS

In TinyOS configuration files cannot be used, because files are not used in TinyOS at all. To have some kind of flexibility, a dedicated c file stores the configuration parameters. Obviously these parameters can only be changed in compilation time.

```

error_t CONFIG_set_tinyos_configure() {
#ifdef RPL_NO_DIOAUTH_HASHCHAIN_OPTION
    RPL_DODAGROOT_ENTRY dodagroot;
#endif /* RPL_NO_DIOAUTH_HASHCHAIN_OPTION */

    RPLCONFIG_rpl_instance_id = RPL_DEFAULT_INSTANCE;
    RPLCONFIG_root_rank = ROOT_RANK;
    RPLCONFIG_mop = STORING_WITHOUT_MULTICAST;
    RPLCONFIG_grounded = TRUE;
    RPLCONFIG_preference = 0;
    RPLCONFIG_authentication_enabled = FALSE;
    RPLCONFIG_path_control_size = DEFAULT_PATH_CONTROL_SIZE;
    RPLCONFIG_DIO_interval_doublings = DEFAULT_DIO_INTERVAL_DOUBLINGS;
    RPLCONFIG_DIO_interval_minimum = DEFAULT_DIO_INTERVAL_MIN;
    RPLCONFIG_DIO_redundancy_constant = DEFAULT_DIO_REDUNDANCY_CONSTANT;
    RPLCONFIG_max_rank_increase = 0; //No local repair <= Rank cannot be changed
    RPLCONFIG_min_hop_rank_increase = DEFAULT_MIN_HOP_RANK_INCREASE;
    RPLCONFIG_objective_code_point = OCP_OF0;
    RPLCONFIG_default_lifetime = 24;
    RPLCONFIG_lifetime_unit = 3600;

#ifdef RPL_NO_DIOAUTH_HASHCHAIN_OPTION
    RPLCONFIG_hash_chain_length = 250;
    RPLCONFIG_hash_chain_initvalue = "kiskacska";

    memset(DIOAUTH_SETTINGS[0].DODAGID.Addr, '\0', IPV6_ADDRESS_LENGTH);
    DIOAUTH_SETTINGS[0].RootVersionNumber = 240;
    SECURITY_HASH_CHAIN_initiate(RPLCONFIG_hash_chain_initvalue,
DIOAUTH_SETTINGS[0].RootVersionNumber, &dodagroot);
    DIOAUTH_SETTINGS[0].RootHashChainValue = dodagroot.RootHashChainValue;
    DIOAUTH_SETTING_SIZE = 1;
#endif /* RPL_NO_DIOAUTH_HASHCHAIN_OPTION */

#ifdef RPL_NO_LEAP_KEY_AGREEMENT
    RPLCONFIG_leap_init_key_seed = "almafa";
#endif /* RPL_NO_LEAP_KEY_AGREEMENT */

    switch (TOS_NODE_ID) {
        case 0:
ADDRESS_generate_multicast_group_address(&(MULTICASTGROUPS[0].address), 1);
            NUMBERMULTICASTGROUPS = 1;
            break;
        case 1:
ADDRESS_generate_multicast_group_address(&(MULTICASTGROUPS[0].address), 1);
ADDRESS_generate_multicast_group_address(&(MULTICASTGROUPS[1].address), 0);
            NUMBERMULTICASTGROUPS = 2;
            break;
        case 2:
ADDRESS_generate_multicast_group_address(&(MULTICASTGROUPS[0].address), 1);
            NUMBERMULTICASTGROUPS = 1;

```

```

        break;
    case 3:

ADDRESS_generate_multicast_group_address(&(MULTICASTGROUPS[0].address), 1);

ADDRESS_generate_multicast_group_address(&(MULTICASTGROUPS[1].address), 0);
    NUMBERMULTICASTGROUPS = 2;
        break;
    case 4:

ADDRESS_generate_multicast_group_address(&(MULTICASTGROUPS[0].address), 1);
    NUMBERMULTICASTGROUPS = 1;
        break;
    }
    return ERROR_NO;
}

```

To have the same configuration file for each node, the root node is not defined in the common configuration file. In TinyOS, the node with address 0 is always the root node.

### 2.1.5 Access to the implementation

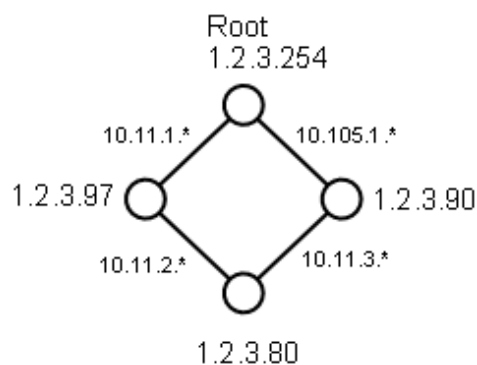
The RPL implementation is available from the WSAN4CIP project SVN server<sup>2</sup>.

## 2.2 Component testing

The modules were tested separately by analyzing manually and automatically the generated logs. The detailed description of test results will be included in Deliverable 5.1. Here we only give some short performance testing results in terms of packet loss and delivery ratio.

### 2.2.1 Performance testing in Linux

Topology used in this test is shown in Figure 14.



**Figure 14: Test topology for the Linux based implementation**

The nodes were connected by wire; the connections were defined by subnets. On the figure, the addresses close to the nodes are their RPL addresses. The connectivity was tested by simple pinging from the root. Ping is a practical test as both directions must be set correctly to have a successful ping. The results are the following:

```

holczer@shamir:~/cryrpl/crysysrpl $ ping 1.2.3.254
PING 1.2.3.254 (1.2.3.254) 56(84) bytes of data:
64 bytes from 1.2.3.254: icmp_req=1 ttl=64 time=0.063 ms

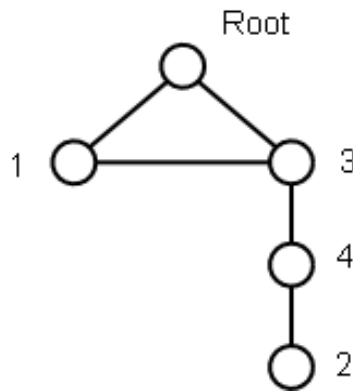
```

<sup>2</sup><https://svn.wsan4cip.eu/>

```
64 bytes from 1.2.3.254: icmp_req=2 ttl=64 time=0.049 ms
64 bytes from 1.2.3.254: icmp_req=3 ttl=64 time=0.041 ms
64 bytes from 1.2.3.254: icmp_req=4 ttl=64 time=0.041 ms
^C
--- 1.2.3.254 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3033ms
rtt min/avg/max/mdev = 0.041/0.048/0.063/0.011 ms
holczer@shamir:~/cryrpl/crysysrpl $ ping 1.2.3.90
PING 1.2.3.90 (1.2.3.90) 56(84) bytes of data.
64 bytes from 1.2.3.90: icmp_req=1 ttl=64 time=3.93 ms
64 bytes from 1.2.3.90: icmp_req=2 ttl=64 time=3.72 ms
64 bytes from 1.2.3.90: icmp_req=3 ttl=64 time=3.65 ms
64 bytes from 1.2.3.90: icmp_req=4 ttl=64 time=3.71 ms
64 bytes from 1.2.3.90: icmp_req=5 ttl=64 time=3.80 ms
64 bytes from 1.2.3.90: icmp_req=6 ttl=64 time=3.72 ms
^C
--- 1.2.3.90 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5029ms
rtt min/avg/max/mdev = 3.659/3.762/3.939/0.096 ms
holczer@shamir:~/cryrpl/crysysrpl $ ping 1.2.3.97
PING 1.2.3.97 (1.2.3.97) 56(84) bytes of data.
64 bytes from 1.2.3.97: icmp_req=1 ttl=64 time=54.5 ms
64 bytes from 1.2.3.97: icmp_req=2 ttl=64 time=52.1 ms
64 bytes from 1.2.3.97: icmp_req=3 ttl=64 time=53.1 ms
64 bytes from 1.2.3.97: icmp_req=4 ttl=64 time=52.4 ms
64 bytes from 1.2.3.97: icmp_req=5 ttl=64 time=53.5 ms
^C
--- 1.2.3.97 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4020ms
rtt min/avg/max/mdev = 52.112/53.190/54.577/0.901 ms
holczer@shamir:~/cryrpl/crysysrpl $ ping 1.2.3.80
PING 1.2.3.80 (1.2.3.80) 56(84) bytes of data.
64 bytes from 1.2.3.80: icmp_req=1 ttl=64 time=59.7 ms
64 bytes from 1.2.3.80: icmp_req=2 ttl=64 time=55.6 ms
64 bytes from 1.2.3.80: icmp_req=3 ttl=64 time=56.3 ms
64 bytes from 1.2.3.80: icmp_req=4 ttl=64 time=55.2 ms
64 bytes from 1.2.3.80: icmp_req=5 ttl=64 time=55.9 ms
64 bytes from 1.2.3.80: icmp_req=6 ttl=64 time=58.6 ms
^C
--- 1.2.3.80 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5026ms
rtt min/avg/max/mdev = 55.246/56.928/59.794/1.694 ms
```

### 2.2.2 Performance testing in TinyOS (TOSSIM)

Topology used in this test is shown in Figure 15.



**Figure 15: Test topology for the TinyOS based implementation**

TOSSIM parametes: Meyer-Heavy noise model, -60 dB

Results:

```

Number of messages sent: 1079
Number of messages received: 1025
Success ratio (received/sent): .9499
Could not forward the message (routing problem): 0
Forward error ratio: 0
Could not forward the message (send buffer problem): 0
Forward error ratio: 0
Number of multicast messages sent: 100
Number of multicast messages received: 484 (all nodes are in the destination
multicast group, max: 500)
Success ratio (received/sent): 4.8400
    
```

The success delivery ratio between the nodes is shown in Table 2: Success delivery ratio between the nodes.

From To	Root	1	2	3	4
Root	1.0000	.9583	.9361	.9166	.8604
1	.9830	1.0000	.7187	.9347	.9387
2	.9666	.9166	1.0000	.8846	1.0000
3	.9655	.9583	.9218	1.0000	.9803
4	.8888	.9636	1.0000	1.0000	1.0000
Multicast message	.9500	.9300	1.0000	.9700	.9900

**Table 2: Success delivery ratio between the nodes**

It can be seen that in general the delivery ratio is good, and no packets are lost because of forwarding errors or send buffer problems. The multicast messages are also delivered with high success (source is Node 2).

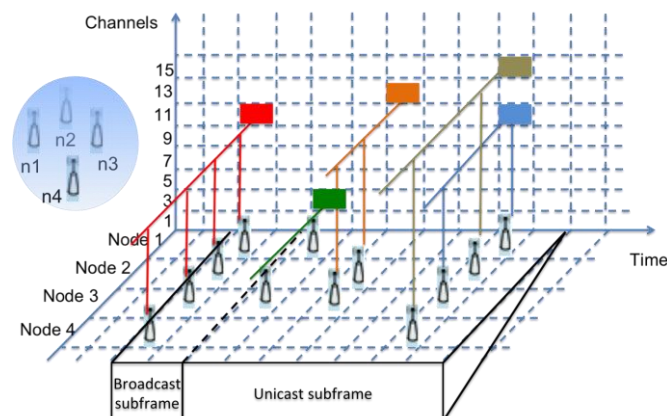
### 3 Implementation and testing of the MAC protocol

The implemented MAC protocol (further on referred to as *LTU-MAC*) is a completely distributed multichannel TDMA scheme. The packet transmission between neighboring nodes happens during superframes of predefined length. Each superframe contains one contention-based period and several assured collision free slots. The transmissions in each part of the superframe happen on different radio channels. The channel hopping pattern is different between subsequent superframes. The protocol is fully distributed and does not require either centralized or manual configuration. The transmitted messages are authenticated. Neither the establishment of the channel-hopping schedule nor the establishment of the time division schedule requires an exchange of control information except for the transmission request from a particular node in the time division schedule case.

The time line for the protocol is divided into epochs. The duration of one epoch is a configurable parameter and may span over the duration of one or several superframes. A superframe contains a broadcast subframe during which transceivers of all nodes are on and a slotted unicast subframe during which only communicating nodes are on. The length of the broadcast subframe is a configurable parameter. The duration of a unicast slot equals the time to transmit one maximum size data unit plus a short acknowledgement generated by the recipient of the data packet. This is a configurable parameter of the protocol. The epochs are enumerated and all sensor nodes have a unified understanding of the current epoch number.

At the beginning of each epoch each node independently computes the channel hopping and time division patterns *unique with respect to the particular destination node*. The computation of the channel hopping and the time division schedules is completely distributed. The derivation of both the channel hopping and the time division patterns is based on computation of a hash function taking identifiers of the communicating nodes and the epoch number as its arguments. These patterns are computed locally by each node and do not require exchange of control messages.

Figure 16 presents the overview of separating the concurrent transmissions both in frequency and the time domains. On the figure nodes *n1*, *n2*, *n3* and *n4* are physically located in the same communication range. This figure also illustrates an important property of the proposed scheme obtained as the result of the analysis presented below. Although there is a non-zero probability for the hash function computing collided slot numbers, the joint probability of collision in time and in frequency domains could be negligible with proper parameterization of the protocol. The right most communication pattern in the figure shows the case when two four nodes select the same time slot, however, their transmission happens over different channels.



**Figure 16. Establishment of the channel hopping and the time division patterns at a glance. The case when a source has only one packet to transmit to the destination.**

The details of protocol's operations are described in Deliverable 3.2 [2].

### 3.1 Implementation

The MAC protocol overviewed above and specified in Deliverable D3.2 [2] was implemented on two different hardware platforms. The initial development and debugging was performed on the Mulle platform. In order to simplify the process of cross-platform porting, our goal was to implement the proposed MAC protocol in a transceiver-independent way and also to keep the functionality of the TinyOS network stack intact as far as possible. In all releases of TinyOS, the implementation of the transceiver's drivers contains the implementation of the MAC functionality. Only unified interfaces (such as ActiveMessage, AMSend, AMReceive etc.) are provided. This is the major obstacle to easy porting of MAC protocols. The RF2XX network stack has two target transceivers, namely RF230 and RF212. The implementation of the RF2XX network stack is relatively hardware-independent compared to other implementations. This property of RF2XX, combined with the fact that one of our target platforms uses the RF212 transceiver, influenced the decision to base our implementation on the RF2XX implementation. In order to completely decouple the MAC protocol from transceiver-specific dependencies, we re-factored the RF2XX network stack. We implemented DriverLayerC (see Listing 1) which performs the decoupling and the wiring of the particular transceiver's driver to the independent driver layer used by the MAC protocol (see line 18-34 in the Listing). In this section, we describe the details of the implementation and the lessons learned while conducting it. Particular attention is paid to those instances in which it was necessary to introduce changes between the implementation and the specification, rather than focusing uniformly on all aspects of the implementation.

```

1 configuration DriverLayerC {
2     provides {
3         interface RadioState ;
4         interface RadioSend ;
5         interface RadioReceive ;
6         interface RadioCCA ;
7         interface RadioPacket ;
8         interface PacketField<uint8_t> as PacketTransmitPower ;
9         interface PacketField<uint8_t> as PacketRSSI ;
10        interface PacketField<uint8_t> as PacketTimeSyncOffset ;
11        interface PacketField<uint8_t> as PacketLinkQuality ;
12        interface LocalTime<TRadio> as LocalTimeRadio ;
13    }
14    uses { interface PacketTimeStamp<TRadio , uint32_t >; }
15 }
16 implementation {
17     //wi r r i n g concrete t r a n s c e i v e r d r i v e r i m p l e m e n t a t i o n
18     components RF212DriverLayerC as Driver ;
19     components RF212DriverConfigC as Config ;
20     Driver -> Config . RF212DriverConfig ;
21     components Ieee154PacketLayerC ;
22     Config . Ieee154PacketLayer -> Ieee154PacketLayerC ;
23     Driver = PacketTimeStamp ;
24     // Provides
25     RadioState = Driver ;
26     RadioSend = Driver ;
27     RadioReceive = Driver ;
28     RadioCCA = Driver ;
29     RadioPacket = Driver ;
30     PacketTransmitPower = Driver . PacketTransmitPower ;
31     PacketRSSI = Driver . PacketRSSI ;
32     PacketTimeSyncOffset = Driver . PacketTimeSyncOffset ;
33     PacketLinkQuality = Driver . PacketLinkQuality ;
34     LocalTimeRadio = Driver ;
35 }

```

**Listing 1: DriverLayerC**

### 3.1.1 Software architecture

The software architecture is shown in Figure 17, where the figures the gray colored ellipses denote the interfaces and the white squares show components. An arrow from a gray ellipse to a square indicates a component using the interface and an arrow from a white square to an ellipse indicates a component providing the interface.

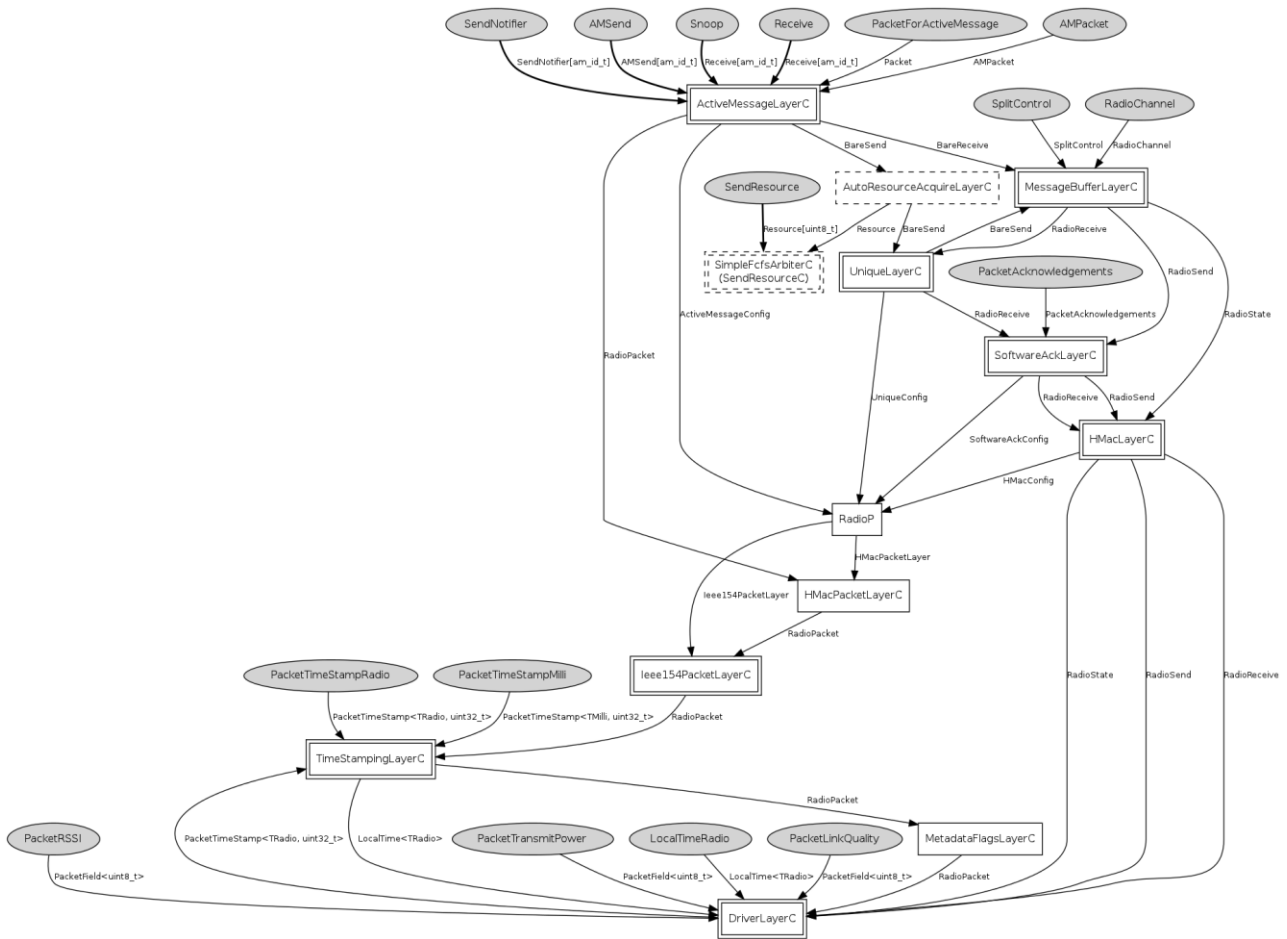


Figure 17: Software architecture of LTU-MAC.

### 3.1.2 Modules and interfaces

Figure 18 shows the provided interfaces of the MAC protocol. The provided and used interfaces are the same as in the original TinyOS stack.

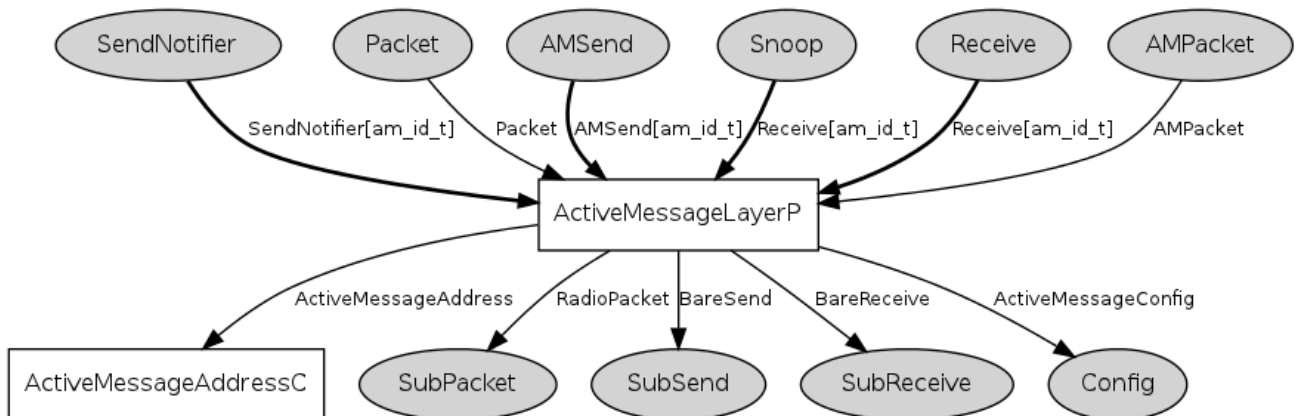
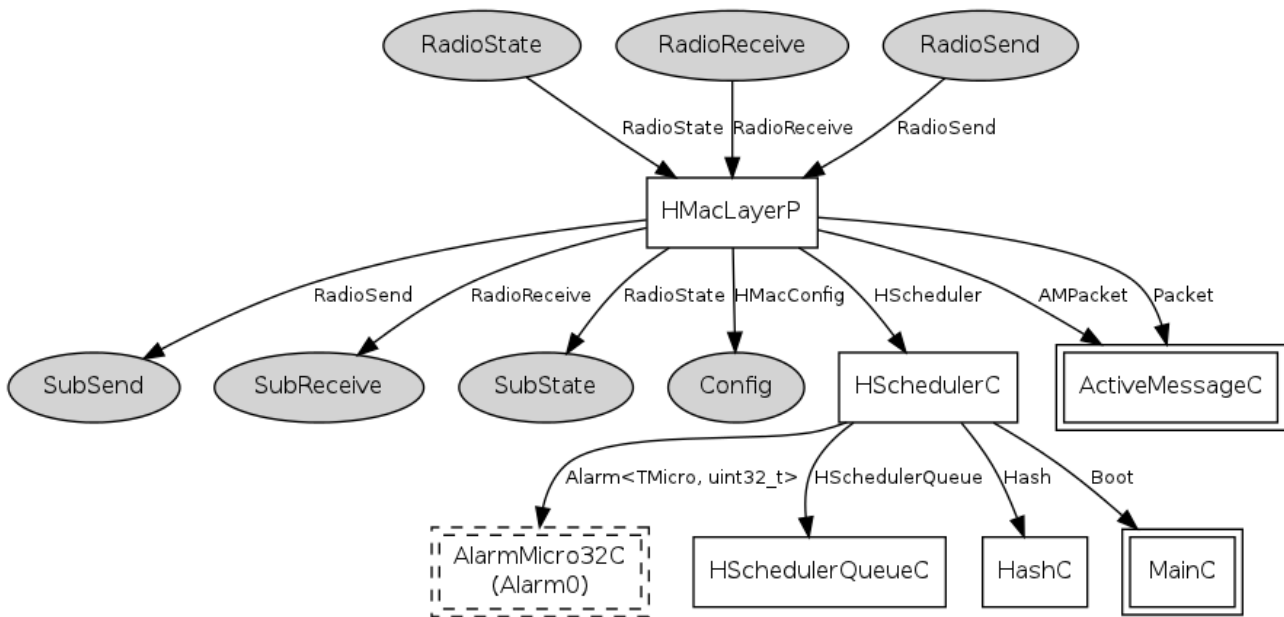


Figure 18: Software architecture of the ActiveMessageLayerP component.



**Figure 19: Architecture of the core functionality of the LTU-MAC.**

Figure 19 shows the core component implementation of LTU-MAC namely LTU-MACLayerP. The main part of it the HSchedulerC component, which operation is described in the following section. The HashC component provides the interface to the calculations of hash values used to determine the time and the channel slots.

### 3.1.3 Operation

This section provides details on implemented protocol operations.

#### 3.1.3.1 Scheduler Phases and Frames

The core of the LTU-MAC implementation is a scheduler, which executes the phases of the protocol described below at the appropriate times. Essentially, the scheduler is a state machine and uses the hardware-independent Alarm component of TinyOS to schedule the timing of the different phases. Once the alarm is set, the Alarm component waits for the particular timer to expire and then executes the Alarm.fired() event. The accuracy and the consistency of the Alarm component on all nodes is an essential property that affects the performance of many parts of the system.



**Figure 20: Structure of the superframe.**

The timing properties of all operational phases of LTU-MAC are the most critical for the overall functionality of the protocol. During the initial stages of the implementation, it became apparent that the time delays caused by switching between different states of the transceiver, computing channel and the slot numbers, and performing other operating system tasks introduced an unacceptable shift in the protocol’s timeline. We therefore added an additional start frame into the superframe in order to compensate for the delays listed above, as shown in Figure 20.

The start subframe is used to perform tasks such as switching to the current broadcast channel, scheduling a packet for transmission, and calculating the channel and time slot numbers. The benefit of the extra subframe is that additional computationally heavy operations such as encryption and signing can be performed here without affecting the protocol’s functionality.

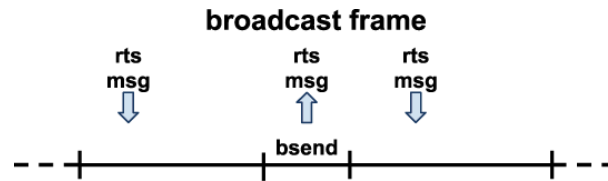


Figure 21: Example of the broadcast subframe after having sent one RTS and received two RTS messages.

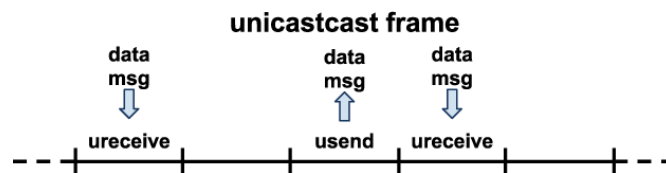


Figure 22: Example of the unicast subframe after having sent an RTS and received two RTS addressed to this node.

The next phase of the LTU-MAC protocol is the broadcast send phase as illustrated in Figure 21. During this phase, an RTS message is broadcasted to the network by the node that has a data packet to send. During this phase, the node also listens to the medium for incoming RTS messages addressed to itself. If the node does not receive an RTS message, which is taken to mean there will be no packets to receive during the unicast subframe, the radio is turned off. The unicast send phase follows the broadcast phase. During this phase the data packet is sent or received in the pre-computed time slot over the pre-computed radio channel (Figure 22).

### 3.1.3.2 Time synchronization

The medium access control techniques used in LTU-MAC require precise time synchronization for collision-free scheduling of data transmission. The development of a new time synchronization scheme is beyond the task of the MAC protocol design, therefore, an existing solution was used in this implementation.

For the sake of simplicity and in order to reduce the burden of implementation, we created a simple synchronization bootstrap routine. We define a synchronization epoch with a configurable number of epochs for LTU-MAC operations in between. During the synchronization epoch, a user-selected node sends a synchronization packet at the start of the broadcast subframe. The receiver nodes will then let their current superframe elapse and save their queued send jobs until after the synchronization has finished.



Figure 23: Synchronization on the sending node.



Figure 24: Synchronization on the receiving node.

Currently, only one node in the network is allowed to synchronize all of the others. This was done to simplify the debugging process. Figure 23 shows the time synchronization process in terms of frames for the sender node. The receiver node on the other hand has a synchronization period whose duration varies depending on the time offset between the nodes, as shown in Figure 24.

### 3.1.3.3 Summary of the implemented parts of the protocol's specification

The differences between the specification and the implementation of LTU-MAC are summarized in Table 3. The API deviates in the following respects. First, the defined interface was changed to comply with the default TinyOS interfaces provided by the *ActiveMessage* component. The status of the *send* command, for instance, is implicitly given by the absence of asynchronous *sendDone* (i.e. packet pending) events and returned error codes (which indicate sending failure). This is due to the design of the network stack in TinyOS (to which we conformed). Secondly, the priority queuing was not implemented in the MAC layer because it must be implemented on top of *ActiveMessage*, which is the interface to the application layer. *ActiveMessage* operates on only one packet at a time. Queuing the packets within that environment is not recommended because each send request from the application layer is followed by a *sendDone* event before the application is supposed to perform another send request.

**Table 3: Summary of the properties of the implemented protocol that differ from those of the original design.**

Property	Implementation	Specification
centralized	no	no
API	different	yes
boot in sync	no	yes
priority queues	no	yes
configuration	yes+	yes
frames	yes+	yes
mac messages	yes+	yes
bootstrap phase	yes	yes
channel hopping	yes	yes
time division	yes	yes
security	no	yes

### 3.1.4 Configuration and usage examples

This section shows example of the test application. In order to use *ActiveMessage* component provided by LTU-MAC the Makefile needs to include the directories of its location as shown in Listing 2.

```

1 COMPONENT=RfTestAppC
2 CFLAGS+=-I../..\
    i. -I../include \
    ii. -I../include/rf-indep-hmac \
    iii. -I../include/rf-indep \
    iv. -I../include/rf-indep/rf212 \
    v. -I../include/rf-indep/rf212/patches
3 CFLAGS+=-DDEBUG1
4 CFLAGS+=-I$(TOSDIR)/lib/printf/
5 CFLAGS+=-DRF212_DEF_CHANNEL=1
6 include $(MAKERULES)

```

**Listing 2: Makefile example for inclusion of the LTU-MAC in the compilation.**

Further the Active message can be used as in standard TinyOS network stack. See Listing 3 for main component configuration and Listing 4 for the test application.

```

1 configuration RfTestAppC {
2 }
3 implementation {
4 components MainC, LedsC, RfTestC;
5 components new TimerMilliC() as Timer0;
6 RfTestC.Boot -> MainC.Boot;
7 RfTestC.Leds -> LedsC;
8 RfTestC.SendTimer -> Timer0;
9 enum {
    a. AM_MSG = 7
10 };
11 components ActiveMessageC as Radio;
12 components new AMSenderC(AM_MSG);
13 components new AMReceiverC(AM_MSG);
14 RfTestC.RadioControl -> Radio;
15 RfTestC.Packet -> Radio;
16 RfTestC.AMPacket -> Radio;
17 RfTestC.AMSend -> AMSenderC;
18 RfTestC.Receive -> AMReceiverC;
19 }

```

**Listing 3: Main component configuration.**

```

module RfTestC {
    uses {

        interface Boot;
        interface Leds;
        interface Timer<TMilli> as SendTimer;

        /* Radio */
        interface SplitControl as RadioControl;
        interface Packet;
        interface AMPacket;
        interface AMSend;
        interface Receive;

    }
}
implementation {

    void sendPingPong();

    bool send = TRUE;
    bool radioBusy = TRUE;    // all purpose, error => HALT
    message_t packet;
    am_addr_t player2;
    uint16_t pingPongCounter = 0;

    enum {

        PING_NODE = 1,
        PONG_NODE = 2,

    };

    typedef nx_struct PingPongMsg {
        nx_uint16_t counter;
    } PingPongMsg;

    event void Boot.booted() {
        player2 = (TOS_NODE_ID == PING_NODE) ? PONG_NODE : PING_NODE;
        call RadioControl.start();
    }

    event void SendTimer.fired() {
        call Leds.led2Toggle();

        if (send) {

            sendPingPong();
            //call Leds.led1Off();
            call SendTimer.startOneShot(3000);

        }

    }

    /* Radio */
    event void RadioControl.startDone(error_t error) {
        printf("RadioControl.startDone> error_t: %d\n", error);
        printfflush();
        radioBusy = FALSE;
        // Master node starts
    }
}

```

### Listing 4: Test application example.

#### 3.1.5 Access to the implementation

The implementation as well as the complete description of the API is accessible on the WSAN4CIP project SVN<sup>3</sup>.

## 3.2 Component testing

### 3.2.1 Testing on own hardware

For the time of the implementation the hardware to be used in the demonstrator was not available. Therefore, we in this section follow the tests performed on the available hardware at the LTU.

**Processing time overhead:** When an alarm is triggered, the operations of the current phase are executed and the hardware clock is configured for the next phase. Table 4 shows the execution time of the different phases in LTU-MAC. Table 5 shows the measured overhead for getting the current time and setting the alarm.

**Table 4: Execution time of different phases in LTU-MAC protocol.**

State of LTU-MAC	Execution time
Start	100
Broadcast send	440
Unicast idle	130
Unicast active	10
Unicast send	570
Unicast receive	90

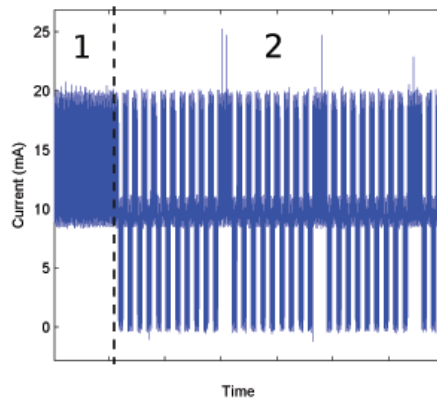
**Table 5: Execution time of the *Alarm* functions.**

Alarm function	Execution time
100	15

Having profiled the execution time for the different states of the MAC protocol and the Alarm functions, we were able to achieve a timing precision of a few microseconds. It was thus possible to maintain the variation in the superframe length within a range of +/-1 microsecond. Because of the inaccuracy of the timing operations and the non-deterministic hardware behaviour, we extended the length of the unicast slots; as currently implemented, their duration in debugging mode is twice that originally specified.

**Energy consumption:** Figure 25 shows the energy consumption both during (1) and after (2) the *bootstrap* phase. Note that the frequent spikes that can be seen in the plots are due to the TinyOS scheduler that runs in the background.

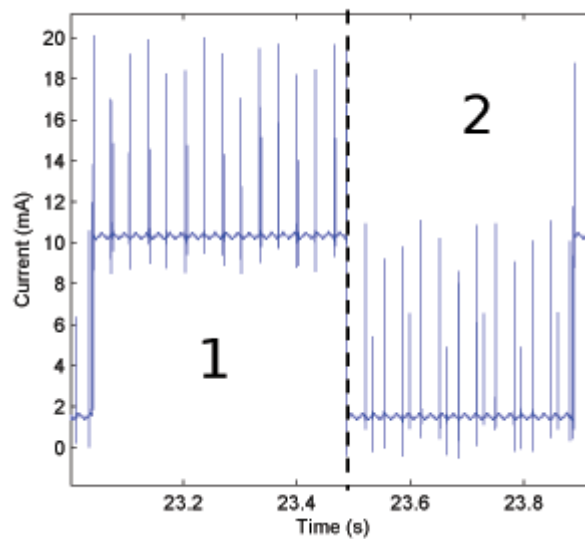
<sup>3</sup><https://svn.wsan4cip.eu/>



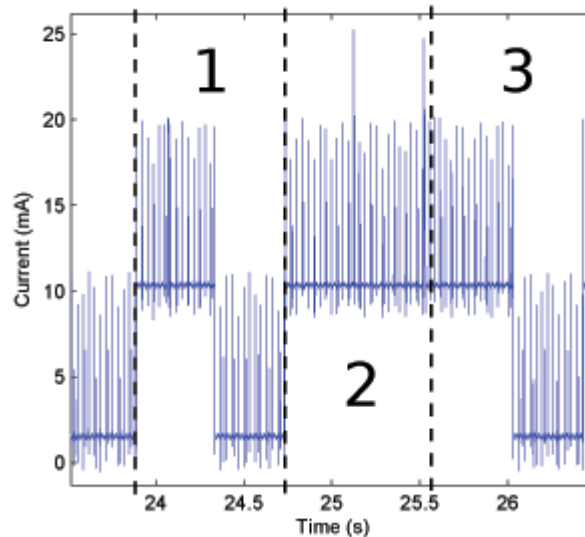
**Figure 25: Current consumption including times before and after bootstrapping.**

By the end of the broadcast frame, LTU-MAC can determine whether it will be in a receive or a send state. We can therefore turn off the transceiver during the idle unicast subframe (provided that the node has finished executing the *bootstrap* process) to save energy.

Figure 26 shows the measured current level during the *broadcast subframe* (1). The current level is significantly higher than during the idle state or the *unicast subframe* (2). Obviously, the smaller current consumption is achieved by turning radio off when idle.



**Figure 26: Current consumption of one idle superframe.**



**Figure 27: Current consumption of one superframe when sending (2).**

Figure 27 shows the current consumption during an active superframe (2). To improve energy conservation, the radio could be turned off during individual unicast slots. However, our measurements show that the time consumed by switching the radio between the On and Off states is not deterministic. The reason for this behaviour may be due to the drivers or hardware problems, and needs further investigation. Further optimization of the energy consumption profile by deactivating the radio on a slot-based scale has therefore been deferred until this issue is resolved.

**Network performance:** The current implementation of LTU-MAC was tested with only one packet transmission per superframe to simplify the debugging process. The maximum throughput is obtained by dividing the MTU size by the duration of the superframe as shown in Equation 3.3. The duration of the superframe is the sum of the durations of the start, broadcast and unicast subframes.

**Equation 1: Network performance equations.**

$$F_{\text{slot}} = \text{MTU}/r_{\text{phy}} \tag{3.1}$$

$$F_{\text{super}} = F_{\text{start}} + F_{\text{slot}} \cdot N_{\text{nodes}} \cdot 2 \tag{3.2}$$

$$r_{\text{mac}} = \text{MTU}/F_{\text{super}} \tag{3.3}$$

**Table 6: Summary of the expected and achieved performance.**

	<i>Specified</i>	<i>Implemented</i>
<b>Unicast slot length</b>	12ms	24ms
<b>Unicast frame length</b>	876ms	1752ms
<b>Broadcast frame length</b>	5000ms	5000ms
<b>Start frame length</b>	--	100ms
<b>Super frame length</b>	5876ms	6852ms
<b>Throughput</b>	500 b/s	35 b/s
<b>Delay</b>	12000ms	47964ms

Table 6 summarizes the analytical performance estimates and the extrapolated performance from the debug configuration of the LTU-MAC protocol. The most important performance metric for our target application was the packet loss rate, which was calculated analytically in previous section. Thus, the parameters of the implemented protocol were assigned values that were intended to ensure it would achieve a packet loss rate of  $\leq 1$ . However, we observed that with the current settings, this requirement was not satisfied. This deficiency will be addressed in the deployment version of MAC protocol on target hardware.

### 3.2.2 Lessons Learned

During the implementation phase, we discovered that imposing strict time requirements on the protocol is problematic for the following reasons. First, achieving time synchronization between nodes is a hard task. Currently, it is not possible to synchronize nodes with microsecond precision. This means that the time slots, subframes and superframes have to be extended by the amount of time that clocks can drift. The impact of extending the slot duration to compensate for hardware state transition delays (the duration is currently doubled) is apparent in the lower-than-expected performance figures for the MAC protocol and the higher energy consumption during communication. The time consumed by hardware and software operations needs to be taken into account when implementing a TDMA-based MAC protocol. For example, it can take between 200  $\mu$ s and 800  $\mu$ s to change channel in this system. Setting a hardware alarm consumes 90-120  $\mu$ s. Interestingly, we also discovered that the time consumed by the transceiver's hardware operations is not deterministic; nodes from the same batch exhibited varying performance.

### 3.2.3 On Assessing the Performance Of Hardware and Software Components

In order to satisfy dependability requirements, the performance of the hardware platform and software components should be assessed experimentally *before* initiating the process of protocol design. Different operations should be profiled on a set of nodes in order to determine the timing range of the target hardware. It is important to note that the time profile of the hardware should be established while it is running the operating system that is to be used. Even a lightweight OS such as TinyOS has some computational overhead. Further, in most cases, it is difficult or impossible to determine whether a given delay in a specific operation has its origin in hardware or in software. Ideally, a general purpose test suite should be developed in order to establish a profile of the target hardware platform when running the chosen operating system. Profiling should be done on several different nodes in order to accurately estimate the performance boundaries.

## 4 References

- [1] Rocha, F.; Grilo, A.; Pereira, P.; “Performance Evaluation of DTSN in Wireless Sensor Networks”, Proceedings of the 4th EuroNGI Workshop on Wireless and Mobility, Barcelona, Spain, January 2008. In Springer-Verlag Lecture Notes in Computer Science, vol. 5122, 2008.
- [2] L. Buttyán (Ed.) et al., “WSAN4CIP Deliverable 3.2 – Specification and analysis of dependable networking mechanisms for the WSAN4CIP application scenarios”, June 2010.
- [3] Silex Technology, “SX-560 Intelligent Programmable WLAN Module”, SX-5602009EN, Internet: [http://www.silxeurope.com/media/datasheets/SX-560-ds\\_en\\_0902.pdf](http://www.silxeurope.com/media/datasheets/SX-560-ds_en_0902.pdf), 2009 [Dec 2010].
- [4] T. Winter (ed) et al., RPL: IPv6 Routing Protocol for Low power and Lossy Networks, Internet-Draft draft-ietf-roll-rpl-19, March 13, 2011.
- [5] A. Dvir, T. Holczer, L. Dora, L. Buttyan, Version Number Authentication and Local Key Agreement, Internet-Draft draft-dvir-roll-security-extensions-00.txt, January 14, 2011.

## Annex A DTSN Implementation Examples

This annex presents examples of DTSN configuration and C application files (sender and receiver). For the meaning of the primitives invoked by the latter as well as the respective parameters, the reader should refer to Deliverable 3.2 [2].

### A.1 DTSN configuration file (dtsn.conf)

```
[security]

password = passtdsn      # Password if using security. Security has to be
                        # enabled at compile time defining 'USE_SECURITY'.

[cache]

enable = 1
max_packets = 1000      # Global maximum

[dtsn]

forward = 1             # Forward packets to other nodes.
                        # Only applies if cache is disabled.

port = 56001            # UDP port used by DTSN.

aw = 6                  # Acknowledge window size (in packets).

no_aw = 2               # Number of acknowledge windows per sending window.

sequence_size = 65535  # UINT16_MAX

payload_size = 1453     # Maximum payload size (DTSN MTU).
                        # To avoid IP fragmentation use;
                        # IF_MTU - HEADER_IP - HEADER_UDP - HEADER_DTSN
                        # HEADER_DTSN = 19 (w/o security) or 51 (w/ security)

max_ear_attempts = 15

[timeout]
# All values in miliseconds

timeout_ear = 75        # Set after sending a EAR. Time before re-tx the EAR.

timeout_cg = 50         # Timeout in congestion situation (Window is full and
                        # can't advance). When this timeout occurs a EAR is
                        # sent.

timeout_act = 100      # Timer set when the window is neither full or empty.
                        # Prevents the situation in which no EAR is sent because
                        # a AW is not completed. After this time a EAR is sent.

timeout_idle = 3000    # Window is empty (nothing to send). If after this time
                        # the application still didn't send anything, the session
                        # is destroyed.

timeout_rcv = 4000     # Activity timeout in a session receiving data. If no
                        # data is received after this time, the session is
                        # destroyed.

timeout_cache = 4000   # How long a session remains in cache without activity.
                        # Only applies if cache is enabled.
```

## A.2 DTSN Header File with Primitives (dtsn.h)

```

/*+-----
 * INESC Inovacao (INOV)
 * Rua Alves Redol no. 9
 * 1000-029 Lisboa
 * Portugal
 *-----
 * Copyright (C) 2011 INESC Inovacao
 * All rights reserved.
 *
 * THE AUTHORIZING COMPANY DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT
 * SHALL THE AUTHORIZING COMPANY BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUEN-
 * TIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
 * PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS AC-
 * TION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
 * SOFTWARE.
 *-----
 * $Id: dtsn.h 390 2011-04-13 18:16:02Z jose.goncalves $
 *-----+*/

#ifndef _DTSN_H
#define _DTSN_H

#include <stdint.h>
#include "dtsn_types.h"

DTSN_handler_t *DTSN_AppAssociate(DTSN_return_t * Ret, int (*callback_func) (uint16_t AppH,
DTSN_event_t EventId));

DTSN_return_t DTSN_AppDissociate(DTSN_handler_t * AppH);

DTSN_session_t *DTSN_AppCreateSession(DTSN_return_t * Ret, DTSN_handler_t * AppH, char
*SenderAddr, char *DestinationAddr, uint16_t AppId);

DTSN_session_t *DTSN_AppAcceptSession(DTSN_return_t * Ret, DTSN_handler_t * AppH, uint16_t
AppId, uint32_t Timeout, DTSN_recv_mode_t Mode);

DTSN_return_t DTSN_AppTerminateSession(DTSN_session_t * SessionH);

DTSN_return_t DTSN_AppResetSession(DTSN_session_t * SessionH, DTSN_packet_t ** Packets);

DTSN_return_t DTSN_AppResetStats(DTSN_session_t * SessionH);

DTSN_return_t DTSN_AppGetStats(DTSN_session_t * SessionH, DTSN_stats_t * Stats);

DTSN_return_t DTSN_AppSendData(DTSN_session_t * SessionH, uint16_t BuffLen, void *Buffer,
int32_t Timeout, uint16_t Tag);

DTSN_return_t DTSN_AppReceiveData(DTSN_session_t * SessionH, uint16_t BuffLen, void *Buffer,
int32_t Timeout, uint16_t * RecvdLen);

const char *DTSN_ErrorStr(DTSN_return_t Code);

#endif

```

### A.3 DTSN Type Definition file (dtsn\_types.h)

```

/*+-----
 * INESC Inovacao (INOV)
 * Rua Alves Redol no. 9
 * 1000-029 Lisboa
 * Portugal
 *-----
 * Copyright (C) 2011 INESC Inovacao
 * All rights reserved.
 *
 * THE AUTHORIZING COMPANY DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT
 * SHALL THE AUTHORIZING COMPANY BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUEN-
 * TIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
 * PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS AC-
 * TION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
 * SOFTWARE.
 *-----
 * $Id: dtsn_types.h 390 2011-04-13 18:16:02Z jose.goncalves $
 *-----+*/

/**@file
 * Estruturas de dados partilhadas entre a API e o daemon do DTSN.
 */

#ifndef _DTSN_TYPES_H
#define _DTSN_TYPES_H

#include <stdint.h>

/*-- API Data Structures -----*/

#define ADDR_SIZE 128

typedef struct _DTSN_session_t DTSN_session_t;

typedef struct _DTSN_handler_t DTSN_handler_t;

typedef struct _dtsn_stats {
    //packets
    uint32_t    packets_tx;
    uint32_t    packets_rx;
    uint32_t    packets_retx;
    uint32_t    packets_fwd;
    uint32_t    packets_data_fwd;
    uint32_t    packets_dropped;

    //bytes out
    uint32_t    bytes_data_tx;
    uint32_t    bytes_data_retx;
    uint32_t    bytes_tx;
    uint32_t    bytes_fwd;
    uint32_t    bytes_data_fwd;

    //bytes in
    uint32_t    bytes_data_rx;
    uint32_t    bytes_rx;

    //errors

```

```

    uint32_t      proterr;
    uint32_t      err;
    uint32_t      memerr;
} DTSN_stats_t;

typedef struct _DTSN_packet {
    uint16_t      tag;
    struct _DTSN_packet *next;
} DTSN_packet_t;

typedef enum {
    DTSN_SUCCESS = 0,
    ER_INVARG = -1,
    ER_INVSESSION = -2,
    ER_APPINUSE = -3,
    ER_ENDSESSION = -4,
    ER_TIMEOUT = -5,
    ER_DAEMON_UNAVAIL = -6,
    ER_UNAVAIL = -7,
    ER_NOBUFS = -8,
    ER_MEM = -9,
    ER_NORES = -10,
    ER_UNKNOWN = -11,
    ER_INVADDR = -12,
    ER_MTU = -13,
    ER_RESETSESSION = -14,
    ER_BUFSIZE = -15,
    ER_DAEMON_CON = -16
} DTSN_return_t;

typedef enum {
    EV_DATAAVAIL = 1,
    EV_SENDAVAIL,
    EV_ENDSESSION,
    EV_NEWSESSION,
    EV_APPERROR,
    EV_RESETSESSION,
    EV_ABORTSESSION
} DTSN_event_t;

typedef enum {
    /*! Datagram reception mode */
    DGRAM = 1,
    /*! Stream reception mode */
    STREAM
} DTSN_rcv_mode_t;

/*-- RPC Data Structures -----*/

#ifdef DIR_VAR_PREFIX
#define DIR_RUN_PREFIX DIR_VAR_PREFIX "run/"
#else
#define DIR_RUN_PREFIX
#endif

#define UNIX SOCK_PATH DIR_RUN_PREFIX "dtsn.socket"

enum CON_TYPE {
    CON_RPC,
    CON_EVENT
};

```

```
enum RPC_PROT {
    //Daemon to API
    RPC_OK = DTSN_SUCCESS, //should be the same as DTSN_SUCCESS!

    //from API to Daemon
    RPC_SENDDATA = 11,
    RPC_NEWSESSION = 12,
    RPC_ERROR = 13,
    RPC_ACCEPTSESSION = 14,
    RPC_ENDSESSION = 15,
    RPC_DATAAV = 16,
    RPC_RECVDATA = 17,
    RPC_GETSTATS = 18,
    RPC_RESETSESSION = 19,
    RPC_RESETSTATS = 20
};

typedef struct {
    uint16_t AppId;
    uint8_t recv_mode;
    char SenderAddr[ADDR_SIZE];
    char DestinationAddr[ADDR_SIZE];
} rpc_session_req_t;

typedef struct {
    uint16_t AppId;
    uint16_t size;
    uint16_t tag;
} rpc_data_req_t;

#endif
```

## A.4 DTSN Receiver (dtsn\_recv.c)

```

/*+-----+
 * INESC Inovacao (INOV)
 * Rua Alves Redol no. 9
 * 1000-029 Lisboa
 * Portugal
 *-----+
 * Copyright (C) 2011 INESC Inovacao
 * All rights reserved.
 *
 * THE AUTHORIZING COMPANY DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT
 * SHALL THE AUTHORIZING COMPANY BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUEN-
 * TIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
 * PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS AC-
 * TION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
 * SOFTWARE.
 *-----+
 * $Id: dtsn_recv.c 391 2011-04-14 15:29:23Z jose.goncalves $
 *-----+*/

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <stdbool.h>
#include "dtsn.h"

#define BUFSIZE 500

volatile bool  exit_flag = false;

static int
callback(uint16_t app, DTSN_event_t event)
{
#ifdef DEBUG
    printf("Callback: AppId=%u EventId=%d\n", app, event);
#else
    (void) app;
    (void) event;
#endif
    return 0;
}

static void
sig_handler(int sig)
{
    (void) sig;
    exit_flag = true;
}

int
main(int argc, char **argv)
{
    DTSN_return_t  ret;
    int            res;
    uint16_t       len;
    uint32_t       recvd;
    uint32_t       size;
    char           buffer[BUFSIZE];
    FILE           *fp;

```

```

DTSN_handler_t *AppH;
DTSN_session_t *SessionH;
uint16_t      LinkId;

if (argc != 3) {
    fprintf(stderr, "Usage: %s file id\n", argv[0]);
    return EXIT_FAILURE;
}

signal(SIGINT, sig_handler);
signal(SIGTERM, sig_handler);

AppH = DTSN_AppAssociate(&ret, &callback);
if (AppH == NULL) {
    fprintf(stderr, "Error %d @ Associate: %s\n", ret,
DTSN_ErrorStr(ret));
    return EXIT_FAILURE;
}

/*
 * Wait for session
 */
LinkId = atoi(argv[2]);
printf("Waiting for session with ID %u...\n", LinkId);
SessionH = DTSN_AppAcceptSession(&ret, AppH, LinkId, 0, STREAM);
if (SessionH == NULL) {
    fprintf(stderr, "Error %d @ AcceptSession: %s\n", ret,
DTSN_ErrorStr(ret));
    DTSN_AppDissociate(AppH);
    return EXIT_FAILURE;
}
printf("Session created\n");

/*
 * Receive file size
 */
ret = DTSN_AppReceiveData(SessionH, sizeof(uint32_t), &size, 0, &len);
if (ret < 0) {
    fprintf(stderr, "Error %d @ ReceiveData: %s\n", ret,
DTSN_ErrorStr(ret));
    DTSN_AppDissociate(AppH);
    return EXIT_FAILURE;
}
printf("Received file size: %u\n", size);

/*
 * Create output file
 */
fp = fopen(argv[1], "w");
if (fp == NULL) {
    perror("Error creating output file");
    DTSN_AppDissociate(AppH);
    return EXIT_FAILURE;
}

/*
 * Receive file
 */
printf("Receiving file...\n");
for (recvd = 0; recvd < size && !exit_flag; recvd += len) {
    ret = DTSN_AppReceiveData(SessionH, sizeof(buffer), buffer, 0,
&len);
    if (ret < 0 && ret != ER_RESETSESSION) {

```

```
        fprintf(stderr, "Error %d @ ReceiveData: %s\n", ret,
                DTSN_ErrorStr(ret));
        break;
    }

    res = fwrite(buffer, 1, len, fp);
    if (res != len && ferror(fp)) {
        perror("Error writing to file");
        break;
    }
}

/*
 * Close output file
 */
fclose(fp);

res = DTSN_AppDissociate(AppH);
if (res < 0)
    fprintf(stderr, "Error %d @ Dissociate: %s\n", res,
DTSN_ErrorStr(res));

if (recvd == size)
    printf("DONE!\n");
else
    printf("Failed! Received: %d\n", recvd);

return ret;
}
```

## A.5 DTSN Sender (dtsn\_send.c)

```

/*+-----+
 * INESC Inovacao (INOV)
 * Rua Alves Redol no. 9
 * 1000-029 Lisboa
 * Portugal
 *-----+
 * Copyright (C) 2011 INESC Inovacao
 * All rights reserved.
 *
 * THE AUTHORIZING COMPANY DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT
 * SHALL THE AUTHORIZING COMPANY BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUEN-
 * TIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
 * PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS AC-
 * TION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
 * SOFTWARE.
 *-----+
 * $Id: dtsn_send.c 391 2011-04-14 15:29:23Z jose.goncalves $
 *-----+*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <stdbool.h>
#include "dtsn.h"

#define BUFSIZE 1453

pid_t      pid = -1;
volatile bool  ack = false;
volatile bool  exit_flag = false;

static int
callback(uint16_t app, DTSN_event_t event)
{
#ifdef DEBUG
    printf("Callback: AppId=%u Event=%d\n", app, event);
#else
    (void) app;
#endif
    switch (event) {
        case EV_ENDSESSION:
            kill(pid, SIGINT);
            ack = true;
            break;
        case EV_ABORTSESSION:
            kill(pid, SIGINT);
            ack = false;
            break;
        default:
            ;
    }
    return 0;
}

static void
sig_handler(int sig)
{
    (void) sig;
}

```

```

    exit_flag = true;
}

int
main(int argc, char **argv)
{
    DTSN_return_t    ret;
    int              res;
    uint32_t         sent;
    char             buffer[BUFSIZE];
    uint32_t         size;
    FILE             *fp;
    DTSN_handler_t  *AppH;
    DTSN_session_t  *SessionH;
    uint16_t         LinkId;
    sigset_t         mask, oldmask;

    if (argc != 5) {
        fprintf(stderr, "Usage: %s file src_ip dst_ip id\n", argv[0]);
        return EXIT_FAILURE;
    }

    pid = getpid();
    signal(SIGINT, sig_handler);
    signal(SIGTERM, sig_handler);

    AppH = DTSN_AppAssociate(&ret, &callback);
    if (AppH == NULL) {
        fprintf(stderr, "Error %d @ Associate: %s\n", ret,
DTSN_ErrorStr(ret));
        return EXIT_FAILURE;
    }

    /*
     * Create session
     */
    LinkId = atoi(argv[4]);
    printf("Creating session with ID %u\n", LinkId);
    SessionH = DTSN_AppCreateSession(&ret, AppH, argv[2], argv[3], LinkId);
    if (SessionH == NULL) {
        fprintf(stderr, "Error %d @ CreateSession: %s\n", ret,
                DTSN_ErrorStr(ret));
        DTSN_AppDissociate(AppH);
        return EXIT_FAILURE;
    }

    /*
     * Open input file
     */
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        perror("Error opening input file");
        DTSN_AppDissociate(AppH);
        return EXIT_FAILURE;
    }

    /*
     * Get file size and send it
     */
    fseek(fp, 0L, SEEK_END);
    size = ftell(fp);
    fseek(fp, 0L, SEEK_SET);
    printf("Sending size: %u\n", size);
}

```

```

    ret = DTSN_AppSendData(SessionH, sizeof(uint32_t), &size, 0, 0);
    if (ret < 0) {
        fprintf(stderr, "Error %d @ SendData: %s\n", ret,
DTSN_ErrorStr(ret));
        fclose(fp);
        DTSN_AppDissociate(AppH);
        return EXIT_FAILURE;
    }

    /*
     * Send file
     */
    printf("Sending file...\n");
    for (sent = 0; !feof(fp) && !exit_flag; sent += res) {
        res = fread(buffer, 1, sizeof(buffer), fp);
        if (ferror(fp)) {
            perror("Error reading input file");
            break;
        }
        /*
         * if (sent == 2000) {
         *     printf("Resetting session\n");
         *     DTSN_AppResetSession(SessionH, NULL);
         * }
         */
        ret = DTSN_AppSendData(SessionH, res, buffer, 0, 0);
        if (ret < 0) {
            fprintf(stderr, "Error %d @ SendData: %s\n", ret,
DTSN_ErrorStr(ret));
            break;
        }
    }

    /*
     * Close input file
     */
    fclose(fp);

    if (sent == size) {
        printf("Waiting for ACK...\n");
        sigemptyset(&mask);
        sigaddset(&mask, SIGINT);
        sigaddset(&mask, SIGTERM);
        sigprocmask(SIG_BLOCK, &mask, &oldmask);
        while (!exit_flag)
            sigsuspend(&oldmask);
        sigprocmask(SIG_SETMASK, &oldmask, NULL);
    }

    res = DTSN_AppDissociate(AppH);
    if (res < 0)
        fprintf(stderr, "Error %d @ Dissociate: %s\n", res,
DTSN_ErrorStr(res));

    if (ack)
        printf("DONE!\n");
    else if (sent == size)
        printf("Failed! No ACK received.\n");
    else
        printf("Failed! Sent %d\n", sent);

    return ret;
}

```