FAKULTÄT FÜR !INFORMATIK

Simulation of a KNX network with EIBsec protocol extensions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

ausgeführt von

Wolfgang Köhler
Matrikelnummer 8925973

am:
Institut für Rechnergestütze Automation
Arbeitsbereich Automatisierungssysteme

Betreuung:
Betreuer: Ao. Univ.-Prof. Dr. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Wolfgang Granzer

Wien, 01.08.2008 _____ _____

(Unterschrift Verfasser) (Unterschrift Betreuer)

Kurzfassung

Systeme der Gebäudeautomation beschäftigen sich mit der automatischen Steuerung von Heizungsanlagen, Belüftungssystemen, Licht und Beschattung. KNX/EIB ist ein verbreiteter Standard in der Gebäudeautomation. EIBsec ist eine Erweiterung dieses Standards um Sicherheitskonzepte wie sichere Kommunikation, Schlüsselverwaltung, Datenintegrität, Vertraulichkeit, Aktualität und Authentifizierung sicherzustellen. Diese Konzepte sind notwendig um KNX/EIB in sicherheitskritischen Aufgabengebieten wie Zutrittskontrollen und Alarmierungssystemen einsetzen zu können.

In der vorliegenden Diplomarbeit werden die Konzepte von KNX/EIB vorgestellt. Das Verhalten von KNX/EIB und dessen Erweiterung EIBsec wird an Hand einer Simulation untersucht und ein Vergleich von für diesen Zweck verfügbaren Frameworks erstellt.

In weiterer Folge wird das Simulations-Framework OMNeT++ im Detail präsentiert, das die Simulation eines KNX/EIB basierten Netzwerkes erlaubt. Bei der Modellierung der Objektstrukturen wurde auf eine flexible Erweiterung des KNX/EIB Netzwerks besonders Wert gelegt. Die Implementierung untersucht im Speziellen, wie sich das Netzwerkverhalten ändert, wenn eine bestehende KNX/EIB Installation um das Protokoll EIBsec erweitert wird. Die durch die Simulation gewonnenen Daten sind in beliebigen Applikationen auswertbar.

Abstract

Building automation systems deal with the control of heating, ventilation, air conditioning, lighting and shading. KNX/EIB is a popular standard in building automation. EIBsec is an extension to this standard and supports secure communication, key management, data integrity, data confidentiality, data freshness as well as authentication. These concepts are required to implement KNX/EIB in security critical applications like access control and security alarm systems.

In this thesis first the concepts of KNX/EIB will be introduced. The performance of KNX/EIB and its extension EIBsec will be analysed by building a simulation. Frameworks for such a simulation will be compared to each other.

Furthermore, the simulation framework OMNeT++, which allows the building of a KNX/EIB based network, will be presented in detail. The modelling of the object structure takes special care of the extensibility of the KNX/EIB network. In particular, the implementation analyses the change in network behaviour when an existing KNX/EIB installation is extended with EIBsec. The results generated when running the simulation can be evaluated with an arbitrary application.

Contents

1 Why simulating a KNX/EIB network?

The Automation System Group at the Vienna University of Technology is home to research in building automation systems (BAS), focusing on security extensions to the KNX/EIB protocol. One such protocol security extension that has been developed is *EIBsec*, which needs to be evaluated in regard to its performance.

In general, the following methods can be used to evaluate protocol extensions [01]:

Analytical methods allow to formalise the protocol in an exact, predictable and provable form. More specific model types would be *mathematical models* or *stochastic models*. Traditionally these are being used in economic simulations, climate forecasts, chemical process simulation. *Formal and functional verification* is used for e.g. protocol verification, verification of algorithms and simulation of digital circuits. However, the interaction between devices and their inter-dependencies make it difficult if not impossible to formalise their behaviour and analyse their complexity.

Prototyping can be used e.g. for proof-of-concept designs. It is somewhat limited to small systems as it is often not possible to implement more than several devices in a laboratory environment. It is best suited for experimental designs, examine their behaviour in a defined environment and possibly extrapolate this behaviour to larger systems. Large scale evaluation cannot be carried out easily as the number of prototypes – as the name implies – is limited. In the proposed case a prototype of a security enhanced device may prove the concept of the device and the interaction with a subset of other devices. Also the implementation of the prototype allows the observation of unplanned side effects, e.g. environmental influences or electrical effects. But the prototyping method will not yield a result when evaluating the effect on a large scale network where all devices interact with each other.

Simulation allows the design and implementation of literally thousands of devices. Simulation can combine the use of mathematical models with models of “real” events, e.g. interacting with equipment. A category of simulators that is widely used is the *discrete event simulator* which represents a chronological sequence of events. The granularity of the simulation can be adjusted to the focus of what needs to be shown. It might not make sense to simulate every aspect of the behaviour of a system but concentrate on the ones that are key to the results. However, not too many parameters may be omitted, otherwise the result of the simulation may not reflect the expected behaviour in reality.

The trigger for a simulation is manifold: the network behaviour needs to be tested but the real network cannot be built. This might be because of its dimensions or affordability. Another example is that a new, possibly not yet implemented, component needs to be tested against a known environment. In the present case *EIBsec* has been defined and a prototype device developed to implement this enhancement. The requirement is to analyse the effect of those extensions on a large network.

Considering the complexity of interaction between the network devices, the desired flexibility in designing and modifying the network and the existence of a prototype device, the *simulation* deems the most appropriate approach to simulate the effects on a large scale network. The available simulators must be evaluated on how fit they are for the purpose of simulating a KNX/EIB network protocol and network structure.

This thesis has been carried out within the project *Security in Building Automation* (FWF Österreichischer Fonds zur Förderung der wissenschaftlichen Forschung; Projekt P19673).

2 Introduction to KNX/EIB

2.1 Purpose of KNX/EIB

A *Building Automation System* (*BAS*) [02] is a network of (programmable) devices that control the environmental condition of a building like lighting and shading, heating, ventilation and air conditioning (HVAC). It aims at improving control, monitoring and administration of technical building subsystems to gain cost efficiency and building control and to improve comfort for the occupants. Management and configuration of an integrated BAS becomes easier and allows reduction of management tools.

Security critical subsystems like access control and security alarm systems have been implemented as stand-alone systems. This is due to the fact that they depend on the underlying control systems to be reliable and robust to avoid malicious manipulation of devices and traffic. However, today the demand for a tighter integration of the traditional BAS and security control systems exists.

The EIB (European Installation Bus) [04] is a fieldbus designed to enhance electrical installations in homes and buildings by separating the transmission of control information from traditional electrical wiring. Its main applications are solutions in lighting, window blinds control and HVAC systems. EIB is based on an open specification, maintained until recently by the EIBA (EIB Association, [05]). The newly emerged KNX standard [06] is a combination of EIB, Batibus and EHS (European Home System), combining their best aspects. EIBA, EHS Association and Batibus Club International formed the Konnex Association, accordingly.

KNX/EIB installations are hierarchically structured, end devices are topologically arranged in *lines* and *areas*. Lines are interconnected with each other by *line couplers* (LC). Up to 15 lines can be combined to an *area*, *backbone couplers* (BC) can combine up to another 15 areas. A maximum of 256 devices can be addressed in a line. Thus, a completely extended KNX/EIB system can accommodate up to 57600 devices.

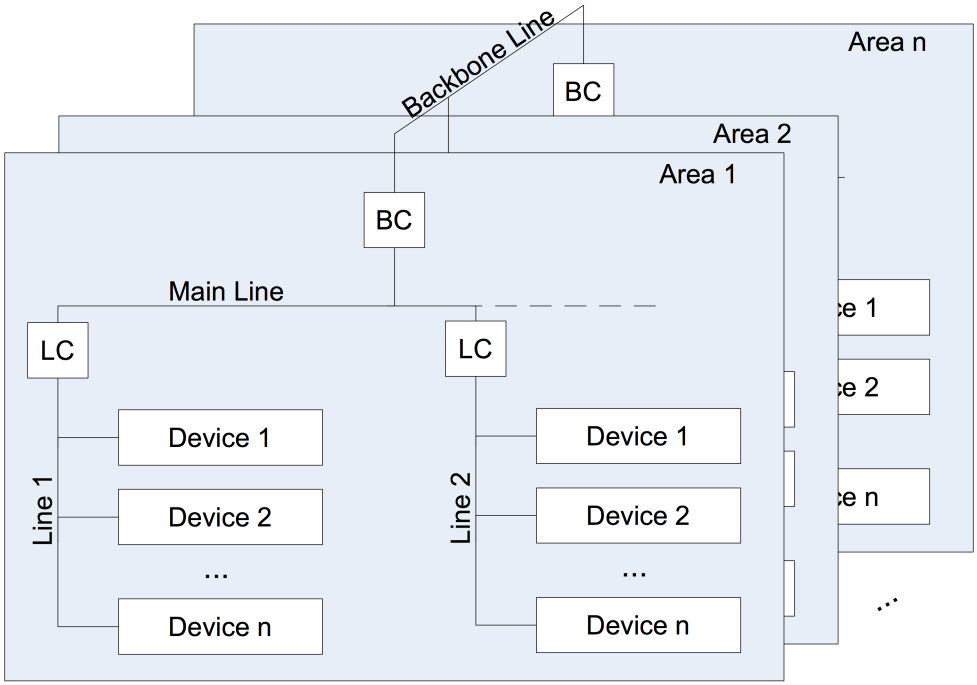


Figure 2.1: KNX/EIB network with different areas and backbone line

Logically, KNX/EIB is a peer-to-peer system. Devices communicate with each other without the presence of a dedicated master. In general, two types of communication can be distinguished: *management communication* using unicast and broadcast and *process data communication* using multicast communication.

2.2 Introduction to the KNX/EIB Protocol

2.2.1 KNX/EIB and the OSI reference model

The *Open Systems Interconnection* (OSI) model splits the complex tasks of data communication into 7 defined sub-areas, referred to as *layers* [07]. Each layer interacts with the layer above and below. A layer, the *service provider,* provides a *service* to the layer immediately above, the *service user*. The *interface* between both layers defines how the service user can access the service of the service provider, specifies the parameters and the result to be expected. A *protocol* defines a set of rules and conventions that is being used by layers of the same level, allowing communication between devices.

Communication between layer N and layer (N-1) via its services or its interfaces respectively occurs via a *Service Data Unit* (SDU). Communication between two peer layers is done via a *Protocol Data Unit* (PDU) which consists of the user data, the *Interface Control Information* (ICI) and the layer specific *Protocol Control Information* (PCI).

Examining the KNX protocol, not all layers of the OSI protocol are necessary. Only 5 out of 7 layers are being used by the KNX standard. These are:

- Physical Layer
- Data Link Layer
- Network Layer
- Transport Layer
- Application Layer

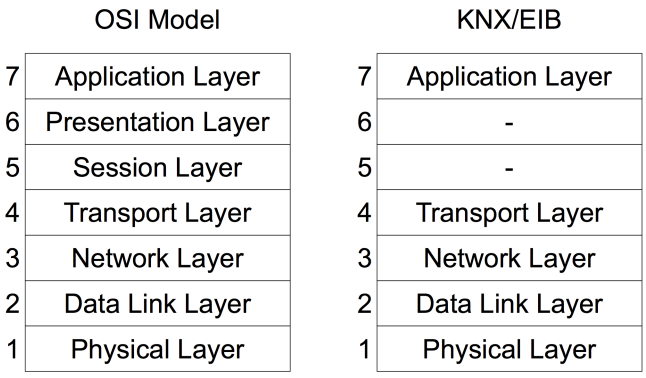


Figure 2.2: KNX/EIB layers ↔ OSI layers

Focusing on the communication of peer layers, there are 4 different *service primitives*: *request* (req), *indication* (ind), *confirmation* (con) and *response* (res). Services need not always to make use of each of the service primitives and can be classified as follows:

Locally confirmed services comprise of a request, an indication and a confirmation. The local service user calls the layer N service provider. A request and the corresponding PDU is being generated and passed on to the layer (N-1) until it is given over to the physical medium. On the receiver side, the peer layer N is activated with an indication, the enclosed PDU is decoded and the data passed to the layer above.

Layer N on the sender side receives a confirmation from the local layer (N-1) telling whether the underlying layer was able to process the request accordingly.

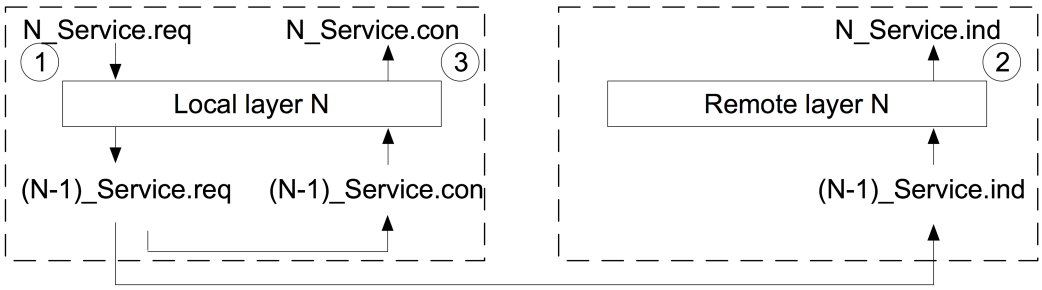


Figure 2.3: Layer N and locally confirmed services

Confirmed services also consist of a request, an indication and a confirmation. With a confirmed service, the peer remote layer generates an acknowledgement immediately after receiving the indication. At the sender side the received acknowledgement is passed on to the local layer N as a confirmation.

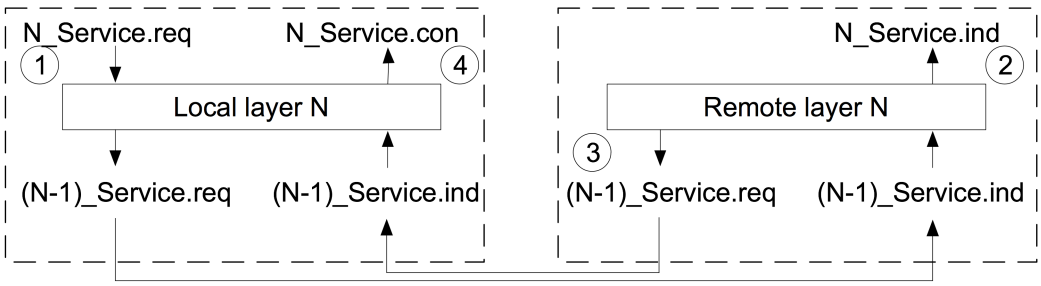


Figure 2.4: Layer N and confirmed services

Answered services always consist of a request, an indication, a response and a confirmation. The confirmation response is being generated at the remote service level provider as a response which is received at the local layer N as a confirmation.

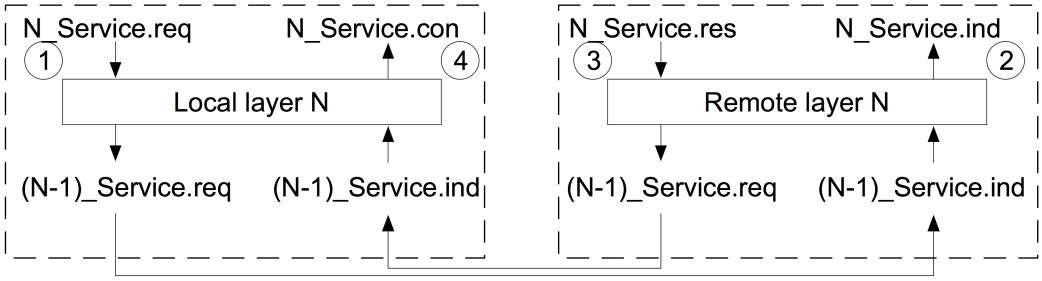


Figure 2.5: Layer N and answered services

2.2.2 Physical Layer

This layer describes the mechanical, physical, electrical and logical properties of the used network medium. Examples are the physical connector type, cabling type, cable impedance and transmissions frequency.

KNX/EIB includes several communication media [08]:

- TP-0 (twisted pair, type 0): this communication medium has been taken over from BatiBUS. It features a transmission rate of 4800 bit/s. KNX TP-0 certified products will operate on the same bus as BatiBUS certified components but will not be able to exchange information with BatiBUS devices.
- TP-1 (twisted pair, type 1): this communication medium has been taken over from EIB. Its transmission rate is 9600 bit/s. EIB and KNX certified products will operate and communicate with each other on the same bus they use.
- PL-110 (power-line, 110 kHz): this medium, too, has been taken over from EIB. Its transmission rate is 1200 bit/s. The EIB and KNX PL-110 certified products will operate and communicate with each other on the same electrical distribution network.
- PL-132 (power-line, 132 kHz): this communication medium has been taken over from EHS. Its transmission rate is 2400 bit/s. KNX PL-132 certified components and EHS 1.3a certified products will operate together on the same distribution network but will not communicate with each other without a dedicated protocol converter.
- RF (radio frequency, 868 MHz): this communication medium, radio frequency with a

bit rate of 38.4 kbit/s, has been developed directly within the framework of the KNX wireless standard.

- Ethernet: this medium can be used in conjunction with the *KNX over IP* specifications which allow the tunnelling and routing of KNX frames encapsulated in UDP/IP frames.

The traditional medium of the former EIB bus is a twisted pair (TP) line, today known as KNX TP-1. The TP-1 cable carries 29V DC voltage as well as the communication signalling. TP-1 allows free topology, up to 1000m length per physical segment. Up to 4 segments can be concatenated with *line repeaters* forming a *line* of maximum 4000 meters.

Only the twisted pair cabling method will be covered in this thesis as the other available media (power line, radio, etc.) have different frame formats but the simulation results are expected not to differ significantly.

The topology of the bus can be freely chosen, loops are not allowed. Junctions in the cabling are allowed and also used in real-world implementations. However, for the simulation model the topology will in principal (but is not limited to) resemble a bus line.

2.2.3 Data Link Layer

This layer ensures the transmission between two devices. There are two basic functions of the layer: first the composition of the data frame based on the information from the above layer and error checking *(LLC, Logical Link Control)* and secondly the coding of the frame, ensuring that the data is transmitted correctly (*MAC, Medium Access Control*).

2.2.3.1 Logical Link Control (LLC)

The LLC provides flow control and error control to the network layer.

A *datagram* is a data unit that can be transmitted from a sender to one or more receivers. Before sending the datagram it must be converted to a *data frame* that must include the full addressing information.

2.2.3.2 Medium Access Control (MAC)

The MAC provides channel access control mechanisms to the underlying physical layer to which it is closely linked.

The MAC access method for KNX/EIB is *Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)*. Collisions are avoided by writing and listening to the bus at the same time. If a device discovers that the signal it listens to is different from what it is sending it ceases communication immediately. This might be the case when another device is sending a packet with higher priority in parallel.

In addition to the bus access method, the MAC also implements some timing considerations specific to the TP-1 specification of the KNX/EIB bus: at the sender side the data to be sent is being received from the LLC sub-layer and sent via a transceiver to the physical medium. However, before sending, the bus must be *idle* for 50 bits time, where 1 bit time is defined as 104µs (9600 bits per second). Repeated frames can be sent after that time period. If a frame is sent for the first time the MAC needs to wait for another 3 bits time. The receiving station has – after receipt of the datagram – to wait for 15 bits time to return the acknowledgement packet. If the acknowledgement datagram needs to be sent by multiple stations, these stations send their responses in parallel, where a *negative acknowledgement* signal prevails a positive one [09].

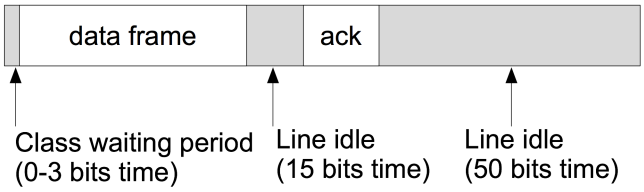


Figure 2.6: Protocol of the MAC layer

2.2.3.3 Addressing Types

There are two ways of addressing at the KNX/EIB protocol:

1) *Individual addressing*: during the engineering process at the installation time each device needs to be configured with a unique individual address. The uniqueness must be ensured within the whole KNX/EIB installation. In KNX/EIB the individual address consists of

   a) the device number (0-255) which uniquely indicates the device within its line,

   b) the line number (0-15) which specifies the line of the device and

   c) the area number (0-15) which specifies the area of the device.

Alternatively, the line number bits and area number bits can be regarded combined as a *subnetwork address*. In that case individual address consists of

a) the device number (0-255) which uniquely indicates the device within its line,

b) the subnetwork address (0-255).

2) *Group addressing*: this logical address can be assigned to one or more devices which will form a functional group. The group address is not bound to the physical location of the device which means that devices within the same group can be spread over several lines. Two structures of a group address are available that differ in their representation:

a) similar to the individual address the group address consists of a triplet of the range (0-31), (0-7) and (0-255) that forms the group address

b) a sub-group number (0-2047) denotes the group's function and the main group number (1-31) specifies the task area of the group.

The naming convention used for individual addresses is *area.line.device* where "." is being used as separator. For group addressing the character "/" is being used as separator in both addressing formats. So 1.4.6 represents an individual address of a device, whereas 2/4/1 represents a group address that may contain one or more devices. During this work we will use only the group notation in triplet form of a/b/c.

2.2.3.4 Frame Formats

As we are aware of different addressing formats, the structure of a standard KNX/EIB TP-1 data frame is defined as follows:

Data Frame L_Data:

C field (8 bit)	Source field (16 bit)	Destination field (17 bit)	RL field (7 bit)	Data field (1-16 Byte)	P field (8 bit)

Figure 2.7: KNX/EIB Data Frame

Figure 2.7 shows the structure of a standard KNX/EIB data frame. Alternatively, an *extended frame* format exists that allows to transport up to 254 bytes in the *data field*. In this case, the structure of the *Control Field* differs and a new *Extended Control Field* has been introduced to

allow encoding of the larger frame. Since the extended frame is not subject of the protocol investigation in this work, it will not be used in further definitions and descriptions.

Control field (C field):

Octet 0: Control Field							
7	6	5	4	3	2	1	0
1	0	R	1	P1	P0	0	0

Figure 2.8: Control Field

Flag R: the *Repeat* flag is being used in a scenario where no or a negative acknowledgement frame has been previously sent back to the sender because one receiver did not positively receive the data frame. To distinguish resent data frames, the *repeat* flag has to be set. This way the receiving stations know that the frame has already been sent and it is the responsibility of each receiver to correctly determine if the information is being passed on to the higher levels. Bits P0 and P1 determine the *priority* of the frame. 4 priorities exist: *system*, *urgent*, *normal* and *low*.

Source field:

Octet 1: Source Field (high)								Octet 2: Source Field (low)							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Z3	Z2	Z1	Z0	L3	L2	L1	L0	D7	D6	D5	D4	D3	D2	D1	D0

Figure 2.9: Source Field

The *source address* is always the individual address of the sending station. Every KNX/EIB device must be assigned a unique address during the installation phase.

Destination field:

Octet 3: Destination Field (high)								Octet 4: Destination Field (low)								Octet 5: Dest. Addr. Flag
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7
Z3	Z2	Z1	Z0	L3	L2	L1	L0	D7	D6	D5	D4	D3	D2	D1	D0	DAF

Figure 2.10: Destination Address Field + DAF

The *destination address field* is of the same format as the *source address field* but with the extension of the *destination address flag (DAF)*. A frame addressed to one physical device (unicast message) must use the individual address of the device and needs to have the DAF set to 0. This way only the addressed device will act upon the received frame. If one or multiple devices are addressed with the frame (multicast or broadcast message), the destination field holds the logical address of the recipients and the DAF is set to 1.

Note that unicast messages are sent by management communication only, e.g. setting sensor parameters by a central monitoring or control station. Normal process communication, e.g. switching a light actuator on or off, is always done with multicast communication to a corresponding group address. The group address can be used by only a single device or by a group of devices, also located on different lines and areas.

R/L field:

Octet 5: R/L Field							
7	6	5	4	3	2	1	0
DAF	R2	R1	R0	L3	L2	L1	L0

Figure 2.11: R/L Field

L0-L3 specify how many data bytes are following in the *data field*, the range is from 0 to 15. The length of the *data field* is 1-16 bytes. The value of zero in the fields L0 to L3 represents a data field length of 1 byte, a value of 15 represents a *data field* length of 16 bytes.

Data field:

The *data field* contains 1 to 16 bytes of data. The data is being forwarded to the network layer for further processing and form the actual data transferred between devices. Note that the data field must contain at least one byte – the protocol information from the transport layer.

P field:

Last Octet: P Field							
7	6	5	4	3	2	1	0
C7	C6	C5	C4	C3	C2	C1	C0

Figure 2.12: P Field

The *parity field* contains 8 check bits. Its purpose is to detect transmission errors. The sender creates the checksum for the frame and adds it to the frame. The receiving device checks the frame consistency by comparing the frame contents against the checksum. When a modification of the frame is detected, the frame is rejected and requested again by the LLC layer.

Acknowledgment Frame L_Ack:

Acknowledgement Frame							
7	6	5	4	3	2	1	0
N1	N0	0	0	B1	B0	0	0

Figure 2.13: Acknowledgement Frame

As illustrated at the description of the MAC layer, an *acknowledgement frame* is being sent upon receipt of a frame to indicate its correct or incorrect receipt. The *acknowledgement frame* consists of one byte and has no additional payload. The format is:

N1	N0	0	0	B1	B0	0	0	
1	1			1	1			positive acknowledgement frame
1	1			0	0			busy frame
0	0			1	1			negative acknowledgement frame

Table 2.1: Table of acknowledgement frame bits and their meaning

- If a frame with a group address (DAF=1) is received incorrectly by one device, the *negative acknowledgement frame* wins against probable *positive acknowledgement frames* from devices which received the frame correctly.
- If a device cannot process the frame, the bits B0 and B1 are set to 0. This signalling is dominant to other devices that send a positive acknowledgement.
- If the frame has been destroyed or is not recognised by a device, the line stays “idle”.

In all cases the frame is transmitted again by the LLC layer's protocol, with the *repeat* flag set.

2.2.4 Network Layer

The network layer sets up end-to-end connections between devices and provides services to transport packets from the source to sink. Target destinations can be either logically or individually addressed. The packets need to traverse different network lines that are linked together with *routers*. The network layer provides the above lying transport layer with a connectionless packet exchange.

Depending on the destination network address the packet is sent to either one receiver (*unicast*), multiple receivers (*multicast*) or all receivers *(broadcast*) on the network.

2.2.4.1 Unicast Communication

If the network layer sends a packet to a single receiver, it uses the individual address of the target device. Additionally, the *destination address flag (DAF)* is set to 0 to indicate an individual address.

2.2.4.2 Multicast / Broadcast Communication

Multicast and broadcast messages are very similar. When addressing multiple devices, these

devices are addressed with a *group address* that has been defined for these devices during the installation phase. To distinguish the logical address from the individual address, the DAF needs to be set to 1.

A broadcast message also has its DAF set to 1 but the destination address is set to 0/0/0.

2.2.4.3 Bridges and Routers

There are 2 device classes that allow to extend the KNX/EIB network:

A *bridge* regenerates the line signal and allows the KNX/EIB line to be extended in length. Basically this is the function of a *repeater* but in KNX/EIB they also acknowledge packets on the data link layer. Bridges do not need an individual address assigned. The reason for the *bridge* – normally a layer 2 device - to be defined in layer 3 is because it decrements the *routing counter* when forwarding a frame, as described below in detail.

A *router* is a device that connects separate network lines and acts on layer 3 of the OSI model. In contrast to a *bridge* it does not only refresh the network signal but the router also decides on basis of the *routing table* whether a frame is forwarded on to another network segment.

In KNX/EIB the *routing table* is being defined at the installation phase of the network. Its entries are static and are not self-modified during runtime. When receiving a frame, the router decides upon the routing table if the frame will be forwarded to the next network segment or not. The decision if a router acts as a *line coupler* or as a *backbone coupler* is based on the individual address of the router which determines its location on the network [10].

Coupler type	Individual address		
	Area number	Line number	Device number
Line coupler	1 - 15	1 - 15	0
Backbone coupler	1 - 15	0	0

Table 2.2: Individual addresses of routers and associated functions

Only the group addresses need to be defined in the routing table as the individual address of the router also denotes the routing of individual addresses. Routers have in general 2 ports. In case of a *line coupler* that is one towards the line and one towards the main line, in case of a *backbone coupler* that is one towards the main line and one towards the backbone line.

Octet 5: R/L Field							
7	6	5	4	3	2	1	0
DAF	R2	R1	R0	L3	L2	L1	L0

Figure 2.14: Routing Counter in R/L Field

When a frame is forwarded by a router – and the same is also valid for a bridge - to another network segment, the router decrements the *routing counter* (contained in the *R/L Field*) if its value is less than or equal to 6. A *routing counter* value of 7 means that the counter will not be decremented by the router, a value of 0 lets the router or bridge discard the frame as it has reached its maximum routing hops.

2.2.5 Transport Layer

The KNX/EIB transport layer offers two methods of transferring data to the application layer:

1. *Connection oriented communication*: this communication type establishes a reliable connection between sender and receiver, transports data and end communication. If a frame is lost during communication it is being retransmitted. The communication can also end if e.g. the receiver does not acknowledge the received frames in time.
2. *Connectionless communication*: this communication type sends a frame from the sender to the receiver without prior establishment of the communication "channel". The sender has no control or receives feedback on whether the frame has reached its target(s).

Octet 6: Transport Control Field							
7	6	5	4	3	2	1	0
T3	T2	S3	S2	S1	S0	T1	T0

Figure 2.15: Data Field, 1st byte = Transport Control Field

Transport Control Field								
7	6	5	4	3	2	1	0	
0	0	0	0	0	0	x	x	Data Unack. PDU, Group PDU, Broadcast PDU
0	1	S3	S2	S1	S0	x	x	Data PDU
1	0	0	0	0	0	0	0	Connect PDU
1	0	0	0	0	0	0	1	Disconnect PDU
1	1	S3	S2	S1	S0	1	0	Acknowledge PDU
1	1	S3	S2	S1	S0	1	1	Negative Acknowledge PDU

Table 2.3: Transport Control Field values

For connection oriented communication only the bits S0-S3 are used as *send counter* of the PDU. The *send counter* is incremented each time a frame is sent from the sender to the receiver and the frame is positively acknowledged. This way the receiver can keep track of received and expected frames.

2.2.6 Application Layer

The layers that have been discussed up to now do not provide any real functionality to the user. The application layer implements services for e.g. process data communication, device management and network management in utilising the layers underneath.

2.2.6.1 Process Data Communication

To implement a function like “light on” or “read temperature”, KNX/EIB devices must implement means of process data exchange. *Communication objects* are being used on the sender and receiver application side to exchange this data. Basically it is a memory area of the device that can be addressed using standardised application layer functions and be read or written to.

As communication objects might be “subscribed” by multiple devices, a means to reach all devices is required. For this purpose *group addressing* is being used. A sending device can address multiple receiving devices when they are listening to the same group address. However, the sending device can only use one group address to send its data. The association between group address and communication object is being defined in the *association table*. When data is being received by a device, the application layer uses its association table to find out which

communication objects are targeted by a group address.

Association Table	
Group Address	Communication Object (ASAP)
0/0/1	0
0/1/2	1
0/0/2	0
0/2/1	2

Figure 2.16: Association table with group address and communication object relation

2.2.6.2 Management Communication

Individual Address Configuration

The *device configuration* is being used at the application layer when an individual address of a device needs to be set or modified. For this purpose the device must already be set to *programming mode* which is invoked normally at the device by pressing the programming button. The device then waits for a broadcast frame which contains the device's individual address.

Another possibility to set a device's individual address is a service that will address the device by its unique serial number and set the individual address accordingly.

Other Services

Some other examples of services that are based on an acknowledged, connection-oriented communication of layer 4 include

- *Memory access* service to read or write up to 12 bytes of memory of a device in one transferred frame.
- *Analogue/digital converter access* service where the requester specifies the channel number of the A/D converter and the number of conversions. The receiver returns the value of the carried out conversion.

- *Routing table access* service to read and write routing tables of line couplers and backbone couplers.

3 Comparison of Protocol Simulators

The following protocol simulators have been considered and compared for the implementation of the KNX/EIB protocol, especially with regard to their effectiveness to simulate the required functionality.

3.1 GloMoSim

GloMoSim (Global Mobile Information Systems Simulation Library, [11]) is being used to build wired and wireless simulations. Its basis is ParSec, a discrete event-based simulation system. GloMoSim strives to implement a layered model similar to the OSI model, providing APIs for the different levels and protocols. Measures can be applied at each level of abstraction. GloMoSim features many ready-to-use wired and wireless protocols like CSMA, 802.11, TCP, UDP, routing protocols like OSPF and wireless propagation models and according device mobility simulations. It is extensible with modules written in C language. Documentation is available online in PDF format.

Initially a DARPA project, it has been abandoned around the year 2000 as no new update is available since that date. There has been a commercial spin-off called QualNet.

3.2 QualNet

QualNet [12] derived from GloMoSim as a commercial branch of the former DARPA project. It has the same underlying ParSec concept of an event-based simulation system. Wired and wireless systems can be simulated and parameterised with configuration files for the relevant simulation layers [13]. Simulated protocols include CSMA, 802.11, 802.16, GSM, TCP, UDP, many routing protocols as well as many protocols of the application layer (e.g. FTP, HTTP, VoIP, GSM). It features commercial support, online documentation, extensibility in C++ (but not for all available versions) and an active user community. Very specific for QualNet is its extensive implementation of wireless protocols, mobility simulation and available terrain simulation partially designed for military usage.

3.3 ns-2

ns-2 [14] is a GPL-licensed simulation framework, mainly used at universities. It includes wired and wireless simulation models that can be extended in C++ and OTcl, an object oriented variant of Tcl. While C++ is more used to extend the system's functional capability, OTcl is being used to define and extend the behaviour of the simulation. An active, mostly academic community is present where the software is constantly developed. Like GloMoSim and QualNet, ns-2 is also very data network centric as it supports a lot of LAN/WAN and wireless based protocols. Extensions are mostly limited to extend routing capabilities in implementing new protocols and to build test networks. Implementation of a completely different network like KNX/EIB, LON, BACnet, etc. would mean a major change of the software.

3.4 OMNeT++

OMNeT++ [15, 16] is a discrete event based simulation framework and can be used for traffic modelling of communication networks, protocol modelling, modelling queueing networks and others. The licensing has a free-of-charge option for non-profit and academic use as well as a commercial license (OMNEST, [17]). In contrast to the simulators discussed above, OMNeT++ provides easy ways to implement any protocol or system that can be handled as based on events. It does not follow an IP-centric approach in its basic simulation models although TCP/IP and wireless implementations are available and widely used. Extensions to existing protocols and new functionality is being added in C++ modules. The network layout is being defined in network configuration files. An active community is available that develops solutions mainly in the 'classic' data network areas.

3.5 Summary

All simulators are capable to a different extent to simulate classic IP based networks and wireless networks. Especially QualNet has a wealth of supported protocols and terrain simulation for its wireless networks part. However, simulating a KNX/EIB network is not directly supported by any simulator. The focus on the decision for a simulator was to find one that was capable of modelling the KNX/EIB protocol to some extent that was usable for the intended work.

The table below illustrates aspects that have been considered when choosing the most appropriate simulator.

	GloMoSim	*QualNet*	*ns-2*	*OMNeT++*
Link	http://pcl.cs.ucla.edu/projects/glomosim/	http://www.qualnet.com/	http://www.isi.edu/nsnam/ns/	http://www.omnetpp.org/
Extensibility	Yes, in C	Yes, in C++ for customisations. Not all versions provide source code to write extensions	Yes, in OTcl scripts and C++ mechanism implementation	Yes, C++ interface definition
Event based	Yes	Yes	Yes	Yes
Layers simulated	L2+, lines / radio can be parameterised	L2+, lines / radio can be parameterised	L2+, lines / radio can be parameterised	L2+ (partially depending on simple module implementation), lines can be parameterised
Line & radio propagation models	Radio and line based protocols	Many radio protocols, includes military specific ones, GSM, WiMax, line based protocols	Radio and line based protocols; focus on Internet technology	Radio and line based protocols available through specific simulation models (INET, MF)

	GloMoSim	*QualNet*	*ns-2*	*OMNeT++*
Additional modules & libraries	-	(Military) radio models, urban terrain modelling including buildings	Satellite communication modules, many TCP based contributed models	Community provided models, mainly IP based routing models but also more generic algorithm and protocol simulations
License terms	Proprietary; academic use only. Commercially available via QualNet	Commercial license	GPL v2	Proprietary; free academic license, commercial license
Community	Initially a DARPA project, abandoned 2000, turned to commercial product QualNet by SNT	Professional community, support provided by supplier	Active community	Active community

Table 3.1: Comparison of protocol simulators

OMNeT++ has been chosen for its versatility. Not only classical networks can be simulated but also queueing problems, file system interaction and more generic protocols like SCSI and HTML.

4 Introduction to OMNeT++

4.1 Overview

OMNeT++ is a discrete event simulation framework [18]. Its primary application area is the simulation of communication networks. Several models for wired and wireless networks exist and some new network routing algorithms are also being implemented for scientific research. Although OMNeT++ is not a network simulator itself, it has widespread popularity as a network simulation platform in the scientific community. The simulation environment is not limited to network models but can simulate a range of problems that can be broken down to passing discrete messages between objects, e.g. pipelining models, process communication, queueing networks or hardware architectures as well.

OMNeT++ provides a component architecture for models. Components (modules) are programmed in C++, then assembled into larger components and models using a high-level language called NED. OMNeT++ comes also with GUI support which enables easy simulation visualisation and interpretation of results during runtime.

4.2 Modelling Concepts

An OMNeT++ model consists of hierarchically nested modules which communicate by passing messages through gates that connect those modules.

The top level *module* (system module) contains sub-modules which can also contain sub-modules. There is no limitation to the nesting level of modules (*compound modules*) making it possible to construct the simulation model according to the logical structure of the actual model to be simulated.

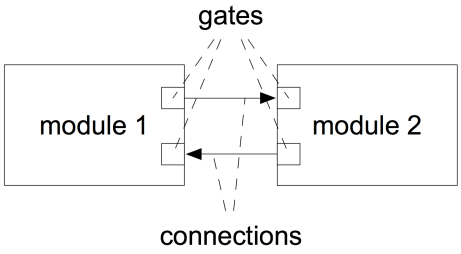


Figure 4.1: Module with gates and connections

The modules are connected to each other via *gates*, which are the interfaces of the modules and *connections*, that link the gates together. Modules can be connected within the same hierarchy level in the compound module or towards the parent compound module. The gate is a unidirectional pipe for *messages*. When designing bidirectional communication between modules, gates and connections in both directions need to be defined.

Connections can be optionally assigned parameters for their speed, error rate and propagation delay. The parameters are later used to calculate the message delays in the simulation and to statistically provoke errors in the packets transmitted. These errors can be queried upon receipt of a message.

Module *parameters* are used to pass specific information to the individual objects of the module and are defined in the NED file or in an `omnet.ini` configuration file. The parameter can be a numeric or string value as well as an XML file.

The *NED file* is the network definition file for an OMNeT++ simulation model. It contains the simple and compound module definitions, the definitions of the gates of a module and the connections between modules.

The functionality of the defined simple modules is implemented in C++. The flexibility of C++ can be used, supported by the functionality of the OMNeT++ class library. The OMNeT++ library contains e.g. the simulation objects of modules, gates, connections, parameters, messages, container classes and data collection classes. Not only messages can be created at runtime within the C++ code, also connections to other modules can be created dynamically at runtime.

4.3 The NED Language

The NED language describes the relation between *simple modules*, *compound modules*, *gates* and *connections* as well as a module's *parameters*.

The following NED file extract is taken from the simulation:

```
module KNXdev
      parameters:
            address: string,
            table: string,
            application: string;
      gates:
            in: in_bus, in_user;
            out: out_bus;
      submodules:
            stack: KNXstack;
                  parameters:
                        address = address,
                        table = table;
            appl: KNXappl;
                  parameters:
                        table = table,
                        application = application;
            gui: KNXappl_gui;
                  parameters:
                        address = address,
                        application = application;
      connections:
            // module gates to external
            out_bus <-- stack.out_bus;
            in_bus --> stack.in_bus;
            in_user --> appl.in_user;
            // internal module gates
```

```
            stack.out_appl --> appl.in_appl;
            stack.in_appl <-- appl.out_appl;
endmodule
```

The example above shows a network device definition, *parameters*, *gates*, *submodules* and *connections*. The *parameter* section defines the parameters that are required to operate the device like the hardware address, the application association table and the application type. The *gates* section defines the way how the device communicates with other objects and represents the interface of the device. The *submodules* section defines which submodules the object is comprised of and also allows to parametrise the submodule's parameters. The *connection* section defines which in and out gates are connected to each other, especially in regard to the used submodules.

4.4 Running a Simulation in OMNeT++

The simulation consists of the

- NED file (`.ned`), containing the simple and compound modules' definitions and the gates and connections describing the structure of the simulation,
- optional message definitions (`.msg`) files, which can contain various data types to add data fields to the messages passed between modules. Message definition files are translated to C++ classes and integrated into the simulation.
- simple module source files as C++ files (`.h` and `.cpp`) that implement the functionality of the modules.

The OMNeT++ simulation system provides the

- simulation kernel, containing the code that manages the simulation and OMNeT's C++ class library.
- user interface to allow interaction with the defined modules and visualisation of the simulation model.

4.5 Collection and Representation of Simulation Results

Output vectors are time series data, i.e. values with time stamps. They can be used to record delays, round trip times of packets, queue lengths, etc. Basically it is a good representation of what happens to certain values over the time of the simulation run. Output vectors are recorded during the simulation from simple modules with the `cOutVector()` method.

The format of a (named by default) `omnet.vec` output vector file is demonstrated in an example:

```
vector 4  "knx_3_line.line_1.bus_1"  "bus_1"  1
4     0     0.0927083333333
vector 5  "knx_3_line.line_2.bus_2"  "bus_2"  1
5     0     0.0927083333333
vector 6  "knx_3_line.line_3.bus_3"  "bus_3"  1
6     0     0.0927083333333
vector 7  "knx_3_line.line_backbone_1.bus_b_1"  "bus_b_1"  1
7     0     0.278125
4     1     0.00208333333333
5     1     0.00208333333333
6     1     0.00208333333333
7     1     0.00625
4     2     0.00208333333333
5     2     0.00208333333333
6     2     0.00208333333333
7     2     0.00625
4     3     0.00208333333333
5     3     0.00208333333333
6     3     0.00208333333333
7     3     0.00625
4     4     0.00208333333333
5     4     0.00208333333333
6     4     0.00208333333333
7     4     0.00625
```

```
4     5     0.00104166666667
5     5     0.00104166666667
6     5     0.00104166666667
7     5     0.003125
4     6     0.00104166666667
5     6     0.00104166666667
6     6     0.00104166666667
7     6     0.003125
```

There are two types of lines: declaration lines and data lines.

The *vector declaration line* looks like:

```
vector 4  "knx_3_line.line_1.bus_1"  "bus_1"  1
```

and shows the keyword “vector”, the vector id, module of creation (as referenced in the NED file), name of the *cOutVector* object and multiplicity (single numbers or pairs will be written; usually 1).

The contents of a *data line* as an example is:

```
4     1     0.00208333333333
```

and shows the vector id, simulation time and recorded value.

Scalar statistics are mainly used to compare model behaviour under various parameter settings called “runs”. The `recordScalar()` method is mostly called at the end of the simulation from within the `finish()` method of the simple module. It records in a (named by default) `omnetpp.sca` file data like gathered information over the simulation run like bytes sent / received, channel utilisation, channel idle time, etc. `run 1` and `run 2` refer to two simulation runs with different start parameters, e.g. different line speeds or queue lengths:

```
run 1 "knx"
scalar "knx_3_line.line_1.bus_1" "simulated time" 34.756
scalar "knx_3_line.line_1.bus_1" "frames sent" 99
scalar "knx_3_line.line_1.bus_1" "frames rcvd" 3088
scalar "knx_3_line.line_1.bus_1" "bytes sent" 64869
scalar "knx_3_line.line_1.bus_1" "bytes rcvd" 3529448
```

```
scalar "knx_3_line.line_2.bus_1" "simulated time" 34.756
scalar "knx_3_line.line_2.bus_1" "frames sent" 99
scalar "knx_3_line.line_2.bus_1" "frames rcvd" 3088
scalar "knx_3_line.line_2.bus_1" "bytes sent" 64869
scalar "knx_3_line.line_2.bus_1" "bytes rcvd" 3529448
scalar "knx_3_line.line_3.bus_1" "simulated time" 34.756
[...]

run 2 "knx"
scalar "knx_3_line.line_1.bus_1" "simulated time" 71.926
[...]
```

Different tools can be used to visualise the gathered data. OMNeT++ provides the following tools:

Plove can be used to load created `.vec` files. In an interactive user interface, the vectors can be narrowed down to examine and individually plot them. However, for larger amounts of vectors one might want to use external tools to separate the vector data from each other and use them optionally in other applications for further analysis.

Scalars visualises the generated `.sca` file in a block diagram and scatter plots. The scatter plot can be used to conveniently compare data results of different simulation runs. Also here it might be useful to extract the data with help of tools for further comparison to external applications.

5 OMNeT++ Module Design

The simulation of the KNX/EIB network consists of the NED file, various C++ files representing the modules defined in the OMNeT++ hierarchy and files such as `UserSim.txt` that is being used to simulate the user behaviour.

The generated output files are `<module_name>.log` files that protocol the received data per KNX/EIB network segment, e.g. `bus_1.log`. Also the line couplers (LC) and backbone couplers (BC) protocol their filtering activity in the appropriate `<coupler_name>.log` files, e.g. `line_backbone_1.log`. These files can be used for further analysis after the simulation run.

To simplify the illustration of *connectors* in the following descriptions, a graphical simplification will be made in the illustrations to denote a bidirectional connection between two modules: the drawing of the incoming connection (connection towards the *in* gate) and corresponding outgoing connection (connection from the *out* gate) will be described as:

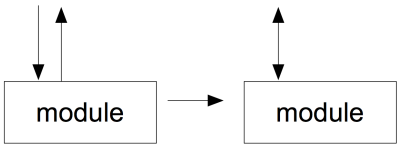


Figure 5.1: Abbreviation of connection when using incoming and outgoing connections that correspond

Technically still both connections – the `in` and the `out` *connection* from the `in` and the `out` *gate* - must be defined in the NED file as OMNeT++ is not aware of bidirectional connections.

5.1 Module Overview

The Fig. 5.2 provides a high level overview of the network model, showing the structure of the network with its backbone line, main lines and lines, coupling devices, devices and the messages exchanged over the bus.

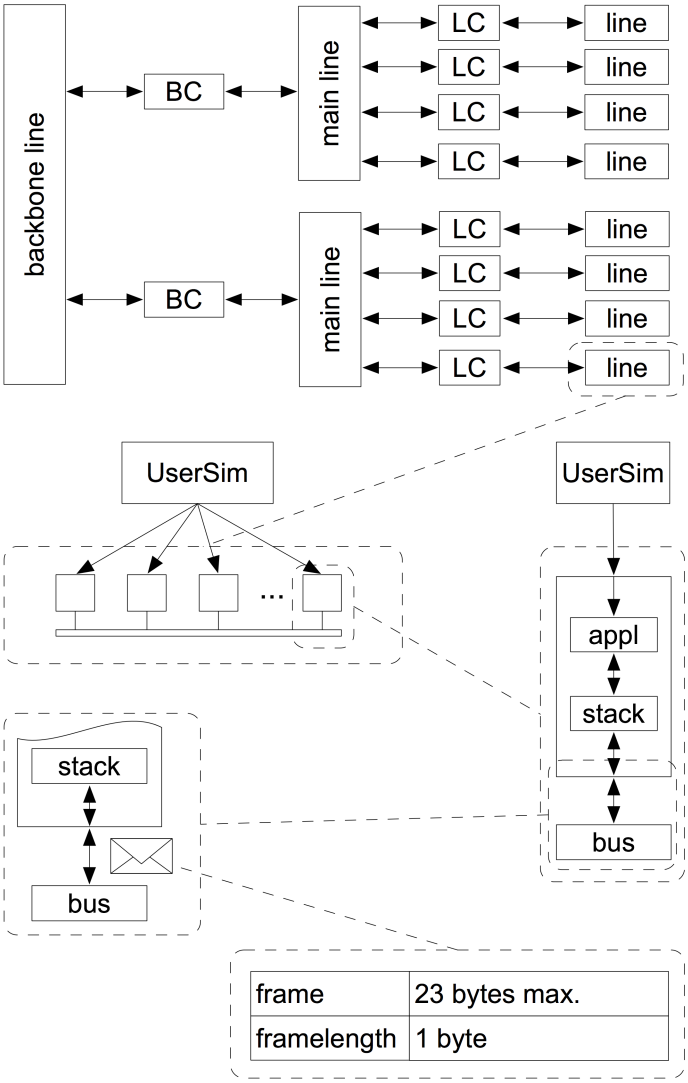


Figure 5.2: OMNeT++ module overview

5.2 Basic C++ Objects

The *KNXPacket* class provides the functionality that represents a message (or frame) passed on through the KNX/EIB network on layer 2. This class is generated out of the message definition outlined in the `KNXmsg.msg` file using the OMNeT++ tool called *opp_msgc*.

Contents of the `KNXmsg.msg` file:

```
message KNXPacket
{
      fields:
            unsigned char message[23];
            unsigned int messagelength;
};
```

This simple definition reflects the KNX/EIB frame to be transported as an embedded message in an OMNeT++ message. `message` is the array that holds the frame content, `messagelength` denotes the valid length of the message. The `KNXmsg.msg` file is processed before compilation in the following way with OMNeT++:

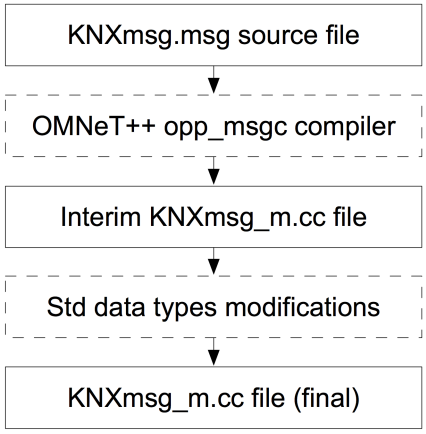


Figure 5.3: `KNXmsg.msg` processing

The `KNXmsg.msg` file is processed by the *opp_msgc* compiler which generates the file `KNXmsg_m.cc` in C++ code. This file uses standard C++ data types like char, unsigned char, int, etc. To uniformly use the UINT8 and UINT16 data type, the file is manually processed to align the data types with the ones used throughout the other C++ code.

The resulting `KNXmsg_m.cc` file is part of the source files for compilation. The file is the basis for defining the *cMessage* embedded object structure in OMNeT++. The *cMessage* embedded object's binary data is encapsulated with OMNeT++ `encapsulate()` method and retrieved from every *KNXstack* module which receives a message with OMNeT++ `decapsulate()`

method. The binary data is loaded with the `setMsgFromBin(UINT8 *msg, UINT8 msglength)` to the *KNXmessage* object which provides various methods to access different elements of the KNX/EIB frame.

Methods of KNXmessage class:

This class contains the get and set methods to get respectively set the object's values. The basic data type used for many methods are strings as this data type allows easy interaction with the OMNeT++ runtime environment, e.g. allows inspection and modification of objects and parameters at runtime. A typical implementation would be that after receiving the OMNeT++ message and setting the object's values with `setMsgFromBin(...)`, the set and get methods can be used on the message.

`KNXmessage(void)` : class constructor. Initialises the message content to be empty (which is not equal to set it to "0" as the KNX/EIB frame structure, empty message content and checksum must be set correctly. The data length of the KNX/EIB frame is set to 0).

`~KNXmessage(void)` : class destructor. It has no added functionality.

`string msg2hexstring(void)` : converts the whole KNX/EIB frame (not only the contents of the message) from binary values to a hexadecimal string. The string conversion is useful for runtime output and allows to inspect the OMNeT++ message content when a message is being transferred within the OMNeT++ simulation environment.

`void hexstring2msg(string hexstring)` : converts the string to a KNX/EIB frame. All data elements of a KNX/EIB frame are required and will be converted to a KNXmessage object. The frame is not checked for consistency, e.g. correct P-field, which allows insertion of "malformed" frames to the network.

`void setMsgFromBin(UINT8 *msg, UINT8 msglength)` : copies the binary information from a pointer to the message. This is being used when e.g. receiving a message from OMNeT++ simulation to populate the corresponding KNXmessage object with the content. The P-field is not being recalculated.

`void updateChecksum(void)` : updates the checksum of the frame. This function is

invoked by all `set_*()` functions (except method `setMsgFromBin()`) to ensure frame integrity after setting frame values. This function does not calculate a correct checksum, it sets a predefined value instead, see Section 5.6.

`void set_L_CField(UINT8 field)` : sets all *control field* bits of a KNX/EIB frame.

`void set_L_CFFrame(UINT8 frametype)` : sets the *control field* value of a *repeated frame*. Correct values are `FrameNormal` and `FrameRepeat`.

`void set_L_SourceAddr(string address)` : sets the source address of the frame.

`void set_L_TargetAddr(string address)` : set the destination address of the frame. Depending on the format of the address string - "." of "/" used as separators – the *destination address flag* (*DAF*) will be set accordingly to 0 or 1.

`void set_L_Class(UINT8 msg_class)` : sets the *frame priority* of the *control field*. Correct values are `PrioLow`, `PrioNormal`, `PrioSystem` and `PrioUrgent`.

`void set_L_DAF(string daf)` : sets the *destination address flag* of the frame to 0 or 1. Note that the DAF is set automatically when setting the target address depending on its format string.

`void set_L_RouterCounter(UINT8 rc)` : sets the *router counter* bits.

`void set_L_LengthField(UINT8 lengthField)` : sets the *length* field of the frame.

`void set_L_SDU(string pdu)` : sets the SDU (basically the data section of the KNX/EIB frame) with data – a hexadecimal string - without modifying other frame information.

`void setMsgPtr(void * msgptr, unsigned int length)` : sets the object's message pointer and length. This is used to interact with the OMNeT++ objects.

`UINT8 get_L_CField(void)` : returns the *control field* of the frame.

`UINT8 get_L_CFFrame(void)` : returns the *frame type* of the *control field*. Values are `FrameNormal` and `FrameRepeat`.

`string get_L_SourceAddr(void)` : returns the *source address* of the frame in the format "x.y.z"

`string get_L_TargetAddr(void)` : returns the *destination address* of the frame in the

format “x.y.z” for individual addresses and format “x/y/z” for group addresses.

`UINT8 get_L_Class(void)` : returns the *priority* of the frame (`PrioLow`, `PrioNormal`, `PrioSystem` and `PrioUrgent`).

`string get_L_DAF(void)` : returns the *destination address flag* value of the frame. Values are 0 or 1.

`UINT8 get_L_RouterCounter(void)` : returns the *router counter* of the frame.

`UINT8 get_L_LengthField(void)` : returns the length of the frame field, denoting the size of the data portion of the frame.

`string get_L_SDU(void)` : returns the data portion of the frame as a hexadecimal string.

`UINT8 get_L_MsgLength(void)` : returns the message length from the message object. Contrary to `get_L_length_Field()` which reads this value from the frame data, this value is from the object variables. Its purpose is to be able to verify any differences between the object's and frame's data.

`UINT8 * getMessagePtr(void)` : returns a pointer to the message.

`void print(ostream &os)` : prints a human readable output of the frame's key values to the OMNeT++ standard output stream.

5.3 Basic Modules

The following modules are defined in OMNeT++ and form the basis of the simulation of the KNX/EIB network. First the simple modules are presented, followed by a description of their interaction and inter-linkage.

Every OMNeT++ module has a corresponding C++ implementation file which defines the functionality of the module. At least the following methods must be defined in the module's C++ sources:

`initialize()`**:** initialises the module. It is a common place to read the parameters defined in the NED file to determine the object's functionality.

`handleMessage()`**:** this method is called every time the module receives a message on any incoming gate. The method needs to check from which gate the message is being received if

there is more than one incoming gate defined. Depending on the incoming gate, the function initiates appropriate calls to handle the message.

Optionally, the module can implement the `finish()` function. This function is called at the end of the simulation run and should terminate the object properly from a simulation point of view. It is not the destructor of the module. Within `finish()` certain statistical values collected during the simulation can be written with the `recordScalar()` function, log files can be finalised and closed.

5.3.1 KNX/EIB Device's Network Stack – KNXstack

The *KNXstack* resembles the part of a KNX/EIB device that covers the communication layers 2 to 7. All communication activity is handed in the *KNXstack* implementation.

The *KNXstack* in the NED file is defined as:

```
simple KNXstack
      parameters:
            address: string,
            table:string;
      gates:
            in: in_bus;
            out: out_bus;
            in: in_appl;
            out: out_appl;
endsimple
```

The *KNXstack* requires 2 parameters:

- *address*: sets the individual address of the device. The address must be defined in the individual address notation format, e.g. 1.2.6 which addresses the area 1, line 2, device 6. Each device requires a unique individual address.
- *table*: the *association table* of the application layer sets the relation between *communication objects* and *group addresses*. The contents of the application layer's association table is shared via the *KNXdev* module with the *KNXstack* module. A valid

entry is e.g. `"1/1/2=lightA;1/2/3=lightB"`. More details are outlined describing *KNXappl* in Section 5.3.2

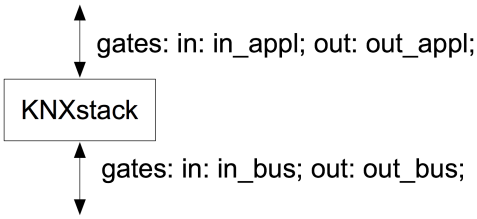


Figure 5.4: KNXstack

Once a received message has been successfully decoded as a valid KNX/EIB frame, the message is verified if it is addressed to the receiving stack and the corresponding *device* as outlined later. Messages that are not addressed to the receiving stack are ignored and deleted from the system. This corresponds to the behaviour of a KNX/EIB device that listens to the medium, reads the frame and its addressing information and – when not being addressed – ignores the received frame.

The KNXstack sends a KNX/EIB frame to the *KNXbus* for transmission. If the bus is currently busy by another frame, the *KNXbus* sends back a *busy* indication (see *collisions* remark in the *KNXbus* Section 5.3.4). When receiving the *busy* signal, the *KNXstack* retries to send the frame after a defined waiting period.

5.3.2 KNX/EIB Device's Application – KNXappl

KNXappl is the application module of a KNX/EIB device. It communicates with the network stack *KNXstack* via messages. OMNeT++ messages between *KNXstack* and *KNXappl* are not binary encoded and encapsulated like the communication over the *KNXbus* but are sent in strings that contain the message content hexadecimal encoded with the OMNeT++ message for easier message tracking during the simulation.

The structure is very similar to the *KNXstack* simple module. It communicates with the stack with a bidirectional gate and the corresponding connection. More details of this relationship are described at the *KNXdev* module.

The *KNXappl* in the NED file is defined as:

```
simple KNXappl
      parameters:
            table: string,
            application: string;
      gates:
            in: in_appl, in_user;
            out: out_appl;
endsimple
```

KNXappl requires 2 parameters, which are outlined by an example:

```
table = "1/1/2=lightA;0/0/0=lightALL", application = "switch1";
```

- *table*: this is the *association table* of the application layer. It is key to the configuration of the device as it defines at which addresses the device is reacting as a communication partner and which actions are associated with the different *group addresses*. The structure of the association table as defined in Section 2.2.6.1 is reflected in the *table* parameter in the following format: “group address=application object”. Multiple entries are separated by “;”.
- *application*: is the application ID which is required by *UserSim* to send the application the correct simulation triggers. The application ID together with the application object ID is used to identify the corresponding application object. *Application* is recommended to be unique within the network. If a specific *application* ID is assigned multiple times, these applications (or their associated devices respectively) will receive the same set of simulation event triggers. For further information how simulation events are triggered, see *UserSim* Section 5.3.6.

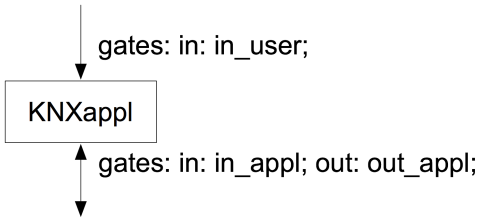


Figure 5.5: KNXappl

Upon receipt of a trigger message from *UserSim* via the gate *in_user, KNXappl* starts sending a message to the stack with the appropriate group address that is looked up from the *association table*.

Depending on the *application* parameter, different functionality is defined in *KNXappl*. Currently the functionality of a light, switch and an ACU (see Section 6.1) is defined.

5.3.3 KNX/EIB Device – KNXdev

KNXdev resembles a KNX/EIB device. It is a compound module of OMNeT++ that consists of two simple modules: *KNXstack* and *KNXappl*. Both communicate via the appropriate gates and connections with each other. *KNXdev* exchanges frames with the *KNXbus* on one hand and decoded frames with *KNXappl* on the other hand. *UserSim* gives "instructions" to initiate actions on the device that are passed to the *KNXappl* module.

The *KNXdev* in the NED file is defined as:

```
module KNXdev
    parameters:
        // address -> forwarder for stack address
        // application -> forwarder for application type
        address: string,
        table: string,
        application: string;
    gates:
        in: in_bus, in_user;
        out: out_bus;
    submodules:
```

```
            stack: KNXstack;
                  parameters:
                        address = address,
                        table = table;
                  display: "p=55,125;i=block/layer,cyan";
            appl: KNXappl;
                  parameters:
                        table = table,
                        application = application;
                  display: "p=55,55;i=block/process,grey";
            gui: KNXappl_gui;
                  parameters:
                        address = address,
                        application = application;
                  display: "p=90,45;b=20,20,rect";
      connections:
            // module gates to external
            out_bus <-- stack.out_bus display "m=s";
            in_bus --> stack.in_bus display "m=s";
            in_user --> appl.in_user;
            // internal module gates
            stack.out_appl --> appl.in_appl display "m=n";
            stack.in_appl <-- appl.out_appl display "m=n";
      display: "b=90,170;i=block/process,black";
endmodule
```

Note that the gates of *KNXappl* and *KNXstack* transparently extend their gates to *KNXdev* and that these gates are used by *KNXdev* as incoming and bidirectional gates for communicating with other modules: `out_bus <-- stack.out_bus`, `in_bus --> stack.in_bus` and `in_user --> appl.in_user` are the respective definitions. Gate names can be identical, they are specified uniquely with their `<module>.<gate>` name.

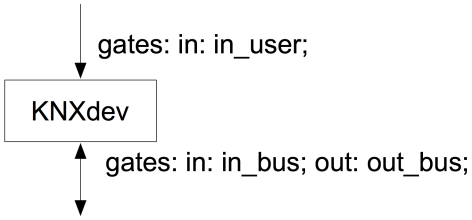


Figure 5.6: KNXdev

The inner structure of *KNXdev* can be illustrated as below:

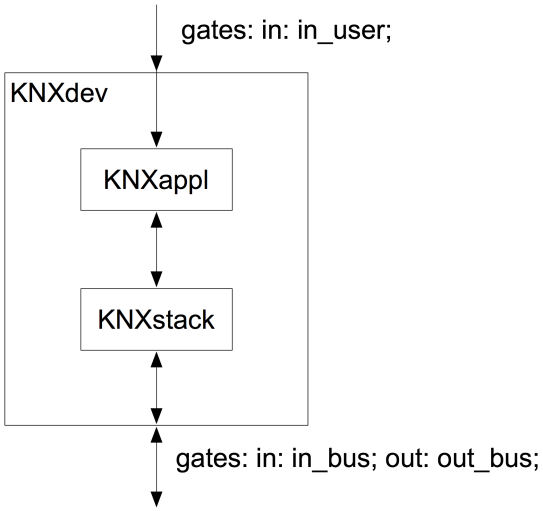


*Figure 5.7:*KNXdev detailed

The structure is also resembled in the OMNeT++ application's graphical representation of the modules when loading the simulation model:

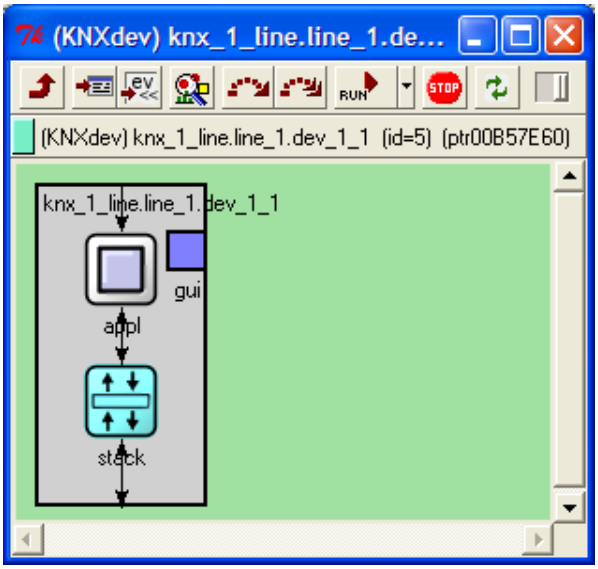


Figure 5.8: KNXdev OMNeT++ representation

The OMNeT++ screenshot shows the same structure as the NED file and Figure 5.7. The objects in OMNeT++ can be opened and displayed in more detail – up to simple module level. Module parameters that are set or defined in the NED file can be viewed and also modified at runtime by opening the module properties. There is one additional module *gui* of type *KNXappl_gui* which is described below in Section 5.3.7.1.

During the simulation, parameters are also used as a means to exchange data between modules that share a common parent module. The dynamically created parameter *group_target_addrs* is a list of addresses that are target for the application layer. The *KNXdev* parameter is added from within the *KNXappl* module at runtime and its value is set according to the data that is read by the *KNXappl* at initialisation. This enables the *KNXstack* simple module to use the same set of parameters for deciding whether a packet is addressed for this device or not.

5.3.4 KNX/EIB Bus – KNXbus

The simple module *KNXbus* is defined in the NED file is as:

```
simple KNXbus
    parameters:
        txRate: string;
    gates:
        in: in[];
```

```
        out: out[];
endsimple
```

The parameter *txRate* sets the bus transfer speed measured in bits per second. When left empty (`txRate = ""`), the default value of 9600 bits/second is set.

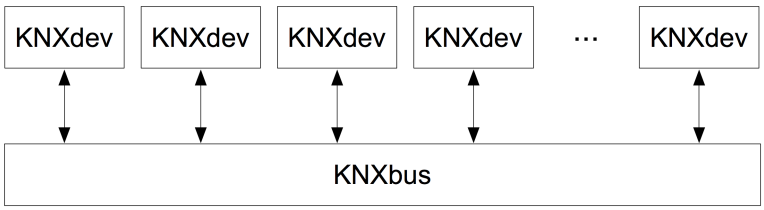


Figure 5.9: KNXbus

The *KNXbus* is an artificially created bus that connects all devices of a KNX/EIB line together. Although the twisted pair implementation of a KNX/EIB network does not necessarily resemble the topology of a bus, it can be abstracted to one considering that every device on the physical line needs to be connected to the twisted pair cable. Branches in the cable can be disregarded to a certain extent when abstracting the model.

The *KNXbus* features an unlimited number of input gates `in[i]` and output gates `out[i]` that are dynamically created when defining the devices on the bus in the NED file. From a simulation point of view, the bus is implemented as a hub that connects all defined devices, including the line coupling devices. The gates in OMNeT++ are unidirectional which requires that a connection to the bus needs two components defined: an *in* gate and an *out* gate that represent one connection. An OMNeT++ message that is being received on a specific input gate *i* is being forwarded to all output gates except the output gate *i* that corresponds to the input gate *i*. Thus, all devices connected to the bus are receiving the message except the sending one which reflects the behaviour in a real KNX/EIB line.

Not all events that might happen in reality can be simulated with this bus abstraction. Events like a broken twisted pair connection, a line segment that imposes a higher error rate or an increased message passing delay cannot be easily simulated. However, the bus abstraction allows that specific parameters can be applied to all connected links.

To create a certain amount of gates, the gate must be defined with the *increment operator* ++:

```
module KNX_line
      parameters:
            [...]
      gates:
            [...]
      submodules:
            bus_1: KNXbus;
                  [...]
            dev_1_1: KNXdev;
                  [...]
            dev_1_2: KNXdev;
                  [...]
            [...]
      connections:
            dev_1_1.out_bus --> bus_1.in++;
            dev_1_1.in_bus <-- bus_1.out++;
            dev_1_2.out_bus --> bus_1.in++;
            dev_1_2.in_bus <-- bus_1.out++;
      [...]
endmodule
```

In the `connections` section the statement `dev_1_1.out_bus --> bus_1.in++` assigns a connection from the outgoing gate of device `dev_1_1` named `out_bus` to the incoming gate of device `bus_1` named `in`. `in++` denotes that the number of available gates of `in` are incremented by one, the symbolised arrow denoted by `-->` specifies the direction of the connection, in this case from `out_bus` to `in`. The corresponding connection in the opposite direction is declared one line below. The gate names are accordingly changed from `in` to `out`, the connection direction arrow changed from `-->` to `<--` to signal that the message flow goes in the other direction.

One important function of *KNXbus* is to collect data on the bus data transfer capacity. When receiving a frame, the bus load is calculated according to the following scheme:

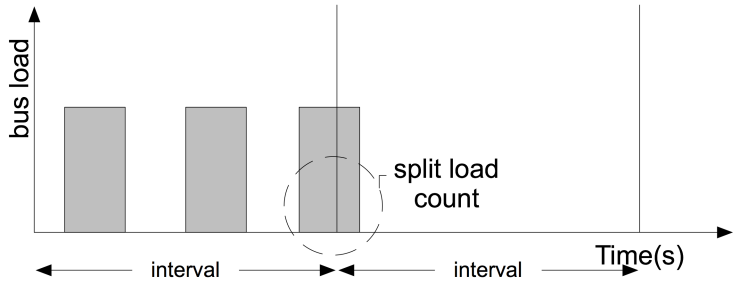


Figure 5.10: KNXbus network load calculation

Figure 5.10 shows two frames within the monitoring interval and one frame that is in between two interval boundaries. For the two frames the time busy on the bus will be added to the bus' network load calculation. As shown below, the frame that crosses the interval boundary will be split into two parts (i_a and i_b) which account in the respective intervals for bus load calculation. Note that the *timestamp* parameter for the function `addStats(simtime_t timestamp, UINT16 packetlength, UINT16 collision)` expresses the start time of the frame.

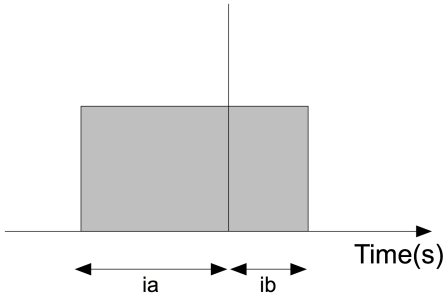


Figure 5.11: KNXbus network split load

The simulated load is calculated by simulating the

- frame transmission time
- time lag between frame transmission and acknowledgement frame and the bus idle time
- acknowledgement frame transmission time

where the time lag and the acknowledgement frame is already added when the function is called:

```
addStats(now, msg_length + frame_overhead_bits, 0);
```

The parameters `msg_length` and `frame_overhead_bits` are in unit *bits*.

The calculation is performed as: (excerpt from the code segment)

```
// total_packets: for object parameter display [#frames]
// total_packet_length: for object parameter display [bits]
// txRate: transfer rate parameter, set per bus [bits/second]
// stat_interval: monitoring interval for bus utilisation (=1s)
// stat_last_interval: last interval that has been
//     recorded [timestamp in seconds]
// last_packetlength: sum of packetlength in last
//     interval [bits]
packettime = ((double) packetlength) / ((double) txRate);
// time to transmit the packet
while ((stat_last_interval + stat_interval) <= now)
{     // one or more full intervals have passed. Update.
      writeStats(stat_last_interval, stat_last_busytime,
                  last_collisions);
      stat_last_busytime = 0.0f;
      stat_load = 0.0f;
      last_collisions = 0;
      stat_last_interval += stat_interval;
}

if ((stat_last_interval + stat_interval) >= (now + packettime))
{     // the interval is still the current one
      stat_last_busytime += packettime;
      // add packettime to busy
      last_collisions += collision;
      // add collision to collisions for interval
}

if ((stat_last_interval + stat_interval) < (now + packettime))
{     // the last interval is finished and needs to be recorded
```

```
      double t1, t2;
      t1 = (stat_last_interval + stat_interval) - now;
      // 1st packettime part before interval boundary
      t2 = packettime - t1;
      // 2nd packettime part after interval boundary
      stat_last_busytime += t1;
      // add to busytime for utilisation
      stat_load = stat_last_busytime / stat_interval;
      // compute utilisation [%] for interval
      writeStats(stat_last_interval, stat_load, last_collisions);
      stat_last_busytime = t2;
      // new stat_load is remainder of packet
      stat_last_interval += stat_interval;
      // set new stat_last_interval timestamp
}
```

In parallel, also the *collisions* on the bus are registered. The term *collision* needs to be interpreted as an indication that the bus is busy (occupied by another device on the same bus) when a device tries to send a frame over the bus. KNXbus allocates the bus during the transmission time of a packet, calculating the transmission time depending on the bus transfer rate *txRate* and the packet length (including the KNX specific idle times and acknowledgements) and blocking the bus for this time period. If a device wants to send data during the bus is blocked, the KNXbus sends back a *collision* indication to the sending device's KNXstack that indicates that the bus is busy. It is calculated as: (excerpt from the code segment)

```
// check for collision on the bus
// the bus is busy in the interval
// [simtime .. simtime+transfer_time+wait_time+ack_time].
// check if the bus is free
if (now < coll_last_busy_timestamp)
{     // BUS BUSY
      // send a message back to receiving port to
      // indicate collision to device
```

```
        cMessage *msg_coll;
        msg_coll = new cMessage("COLL");
        msg_coll->setLength(0); // status message, no length
        send(msg_coll, "out", recv_gateindex);
        addStats(now, 0, 1); // add the collision to stats
}
else
{       // BUS FREE
        addStats(now, (UINT16)msg->length() + frame_overhead_bits,
                 0); // add package to stats [bits]
        // add the overhead from the frame as busy bus time
        coll_last_busy_timestamp = now + ((double)(msg->length() +
           frame_overhead_bits)) / ((double)txRate);
        // calculate message delay for forwarding to out ports
        // due to frame length
        // delay = message length [bits] / transfer speed [bits/s]
        delayTime = ((double)(msg->length())) / ((double)txRate);
        // send message to all out ports but receiving one
        for (int g = 0; g < this->gateSize("in"); g++)
        {
                if (g != recv_gateindex)
                {       // copy of message to be sent to other port
                        cMessage *msg_cpy;
                        msg_cpy = (cMessage *) msg->dup();
                        sendDelayed(msg_cpy, delayTime, "out", g);
                }
        }
}
```

It is the device's responsibility to act upon this signal and resend the packet. The *collision* events are also measured and are indicating the load on the bus, representing the pending transmissions of devices. A corresponding call of the function `addStats()` to record such an event is done by setting the `collision` parameter to 1:

```
addStats(now, 0, 1);
```

The default transmission rate for the *KNXbus* is assumed to be 9600 bits/second. This can be changed by applying the *txRate* parameter. Changing the bus transmission rate directly affects the bus load, allowing simulation of a high speed backbone bus and measure its load during the simulation. Also the transmission delay – the time from receiving the frame at one of the `in` gates and forwarding it to `out` gates – is calculated depending on the transmission rate *txRate* and the length of the frame. The forwarded frames are delayed by the transmission time and scheduled in the OMNeT++ event queue.

In the simulation, parameters are also used as a means to feed back data of the module to the user. The parameter *counters* is created dynamically at runtime (thus it is not represented in the NED file) and shows the current statistics for packets transferred over the bus. The parameter can be watched from within the simulation run by opening the corresponding OMNeT++ object's property.

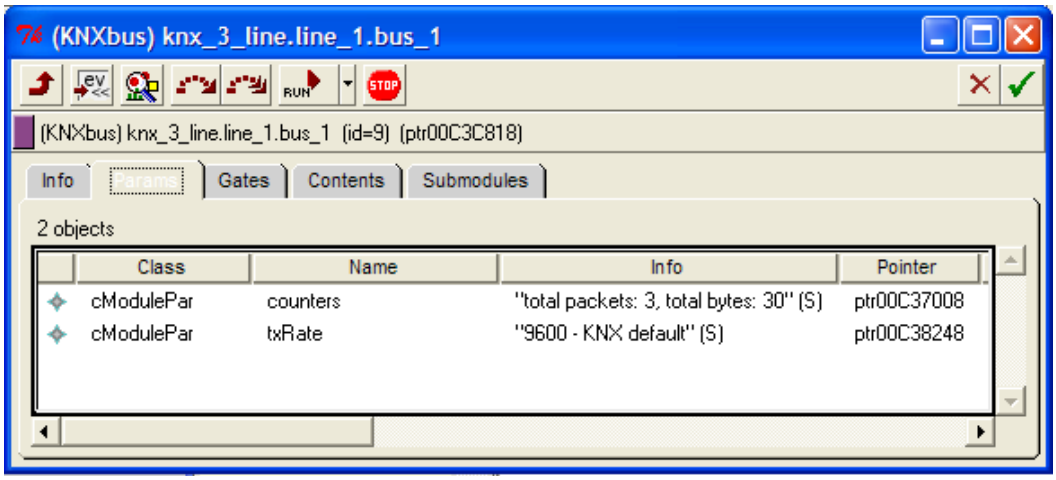


Figure 5.12: OMNeT++ screenshot showing *counters* runtime value

5.3.5 KNX/EIB Line Coupler - KNXlinecoupler

The *KNXlinecoupler* module simulates the functionality of a KNX/EIB line coupling device. Typically it has two bidirectional ports: one towards the line, one towards the backbone.

The *KNXlinecoupler* in the NED file is defined as:

```
simple KNXlinecoupler
```

```
    parameters:
        address: string,
        routing: string;
    gates:
        in: in_line, in_backbone;
        out: out_line, out_backbone;
endsimple
```

The *KNXlinecoupler* requires 2 parameters:

- *address*: the individual KNX/EIB address of the line coupler device
- *routing*: the routing table. It defines which group addresses will be passed on from the line to the backbone and from the backbone to the line. A sample entry for the *routing* parameter would be: `"1/3/5;2/1/4"` which is interpreted as the group addresses 1/3/5 and 2/1/4 will be routed through the coupling device.

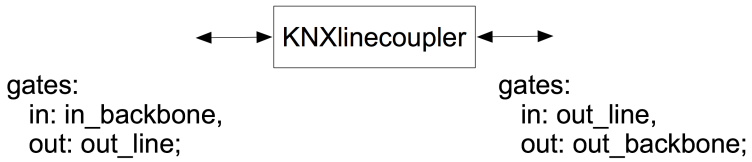


Figure 5.13: KNXlinecoupler

Depending of the frame's destination address the frame will be passed to the backbone by the *backbone coupler* or the frame will be discarded and thus be kept within the main line attached.

The functionality of the *line coupler* is identical to the *backbone coupler*. The individual address of the coupling device determines its exact behaviour. A device address of x.0.0 implies that the device is being located at the backbone line – the *line coupler* acts as a *backbone coupler*. A device address of x.y.0 locates the coupling device to be on a main line, acting as a *line coupler*. The individual address of a coupling device needs always to be set to 0.

5.3.6 Device's User Simulation – UserSim

The simple module *UserSim* does not represent any KNX/EIB specific function. Its purpose is

to simulate user interaction with the various KNX/EIB devices defined in the simulation. A defined OMNeT++ network is a set of modules, sub-modules and connections between these modules. To actually simulate events, messages must be created at the start of the simulation. In fact a simulation ends when no more messages are in the queue to be processed by OMNeT++.

The *UserSim* in the NED file is defined as:

```
simple UserSim
      gates:
            out: out[];
endsimple
```

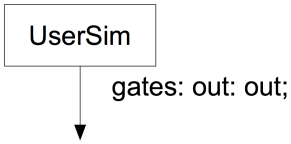


Figure 5.14: UserSim

UserSim is a means to initialise those messages used for the simulation. It parses the file `UserSim.txt` when the module is initialised by OMNeT++ with the `initialize()` function. After parsing the file, OMNeT++ has a set of messages in its event queue. During the simulation, additional messages are created dynamically by modules like *KNXbus*, *KNXstack* and others.

The NED definition shows that *UserSim* has only output gates. These are connected to the various *KNXdev* modules. Similar to *KNXbus* the `out[]` gates are created with the increment operator ++ when designing the NED file as shown in the example:

```
dev_1_1.in_user <-- user.out++.
```

The *UserSim* is connected to every *KNXdev* and sends events to the application module *KNXappl* of the device.

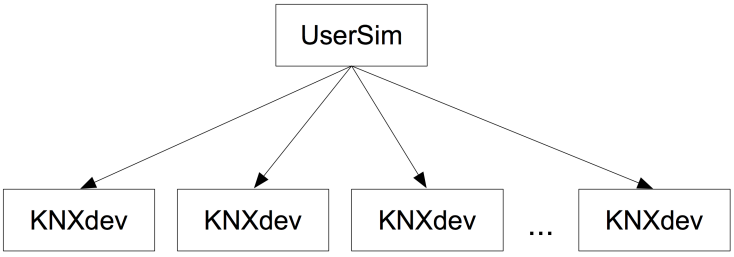


Figure 5.15: UserSim linked to multiple KNXdevs

The connection *UserSim* to *KNXdev* is ideally across line and backbone line boundaries. Technically, the lines and backbone lines do have a separate instance of *UserSim* defined but logically connect all to one module, sharing one configuration.

The structure of the `UserSim.txt` file is outlined in the following example:

```
0/0.01/80, switch2, lightB
1,         switch2, lightB:00
2,         switch2, lightB:01
3.1,       switch2, lightB:00
4,         switch2, lightB:01
5.1,       switch2, lightB:00
6,         switch2, lightB:01
0/~1.5/60, switch5, lightD:02
```

The format of a line is: <timestamp[s][/interval[s]/count[#]]>,<application>,<data>. Every line describes either a single event or a series of events that will be sent to *KNXdev* modules. The events are scheduled in the event queue of OMNeT++.

The parameters *timestamp* and *interval* are in seconds, *count* is a number, *application* and *data* are strings. The detailed parameters are:

- *timestamp*: is the simulation time in seconds at which the event will be triggered. Every simulation run starts with simulation time 0.
- *interval*: for *count* > 0 the event will be triggered again after *interval* seconds for *count* times. Note that a point has to be used to describe the decimal sign, e.g. `0.5`. There will be a uniformly distributed spread *s* defined around timestamp T_n within which the event

will be triggered. The modifier "~" can be used to broaden the spread *s* even more, e.g. `~1.5`. The spread *s* is set depending on *interval*'s *t* value and the timestamp T_n, the triggered event is uniformly distributed within that interval.

The behaviour is illustrated in Fig. 5.16 showing the interval *t,* the spread *s,* the timestamp *T* and count *x:*

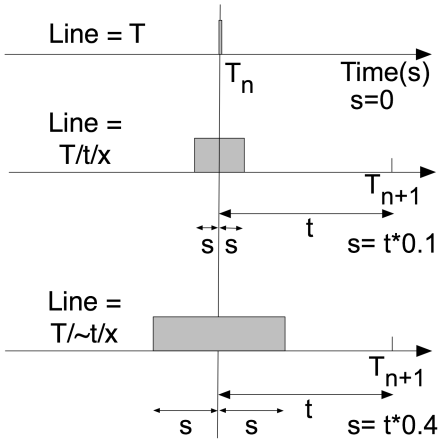


Figure 5.16: UserSim timings

- *count*: denotes how often an event will be repeated.
- *application*: describes the application of the KNXdev module as defined in the NED file. An example is "switchX" which describes the KNXdev module to work as a light switch. To differentiate the various switches in the simulation this switch has the (not necessarily unique) description "switchX". If more than one *KNXdev* module has the same application description, the event will be set for all such modules.
- *data*: data that will be passed to the application. It consists of the *communication object ID* and optionally, separated with a ":" character, the application data. For example the entry "`lightD:02`" addresses communication object "lightD" and sets the application data to the hexadecimal value 02. The application addressed will look up the communication object and create a message with the respective target group address and use the provided application data as the frame's data field.

The example of the `UserSim.txt` text file from above shows line `0/0.01/80, switch2, lightB` which will be interpreted as follows: starting at simulation time 0 (at the

beginning of the simulation) the parameter “lightB” will be sent to all devices configured as “switch2”. This will be repeated every 0.01 second (with a light variation as outlined in Fig. 5.16) for 80 times. The scheduled events that will be set in the OMNeT++ event queue are timed as:

time[i] = 0 + i*0.01 (+ spreading) where i = [0..79].

The extract from the code sequence in UserSim that schedules the events is:

```
for (unsigned int i=0; i <= repeat_time; i++)
{     // "every_time" denotes the interval
      double limit_lo = 0.0f;
      double limit_hi = 0.0f;
      double sched_time_2 = 0.0f;

      if (modifier)  // ~ modifier used
      {
            limit_lo = sched_time + ((double)i * every_time) -
               0.4*every_time;
            limit_hi = sched_time + ((double)i * every_time) +
               0.4*every_time;
      }
      else
      {
            limit_lo = sched_time + ((double)i * every_time) -
               0.1*every_time;
            limit_hi = sched_time + ((double)i * every_time) +
               0.1*every_time;
      }
      if (limit_lo < 0.0f) limit_lo = 0.0f; // boundary check
      sched_time_2 = uniform(limit_lo, limit_hi);
      msg = new cMessage((cMessage) sched_appl_par.c_str());
      sendDelayed(msg, sched_time_2, "out", getGate(sched_appl));
}
```

The line `1, switch2, lightB` will be interpreted as a one time event that will be scheduled at simulation time 1s with no repeating pattern.

Both, the one shot events and the repeated events allow a set of actions on the KNX/EIB network. Working with the same set of parameters in the `UserSim.txt` file allows to run identical simulations with comparable results. Note that when using the modifier "~", the timed events will be set within a larger time interval rather than one exact point in time. When introducing new functionality into the KNX/EIB network, the additionally scheduled events can easily be added to the `UserSim.txt` file. Due to the nature of how events are scheduled in OMNeT++, there is no need to put the events in the `UserSim.txt` file into any sorted or structured timeline.

5.3.7 Helper modules

Several modules are defined that do not have an immediate counterpart in KNX/EIB. They are being used for helping with visual representations, modelling the topology or by helping in standardising the general structure and capture the situations in which otherwise exceptions to the design of other modules would be necessary.

5.3.7.1 KNXappl_gui

The simple module *KNXappl_gui* is a helper module for the visual representation of *KNXdev* and *KNXappl*. It is implemented for a small subset of applications and allows to represent the state of an application. As an example, a light that is turned on or off can be represented by a lightened or darkened box. *KNXappl_gui* sets the specified icon according to the application state.

The definition of *KNXappl_gui* in the NED file:

```
simple KNXappl_gui
      parameters:
            address: string,
            application: string;
endsimple
```

No gates are required to communicate with the module. Instead the module is addressing other modules via OMNeT++'s `parentModule()` function, which allows access to parameters and functions of the objects' parent module, directly.

5.3.7.2 KNXterminator

The KNX/EIB topology does not require a network termination. Any device on the network can be the last device on the network. However, for some situations, e.g. when designing a single line network to be simulated it might be required or favourable to terminate a bus or other components. Some modules are defined with a fixed set of gates (e.g. line couplers) that – by OMNeT++ definition – require to be connected to another module's gate. The *KNXterminator* module can satisfy the formal requirement of connecting to the gates. It will receive the packets from the gate and discard them.

The *KNXterminator* in the NED file is defined as:

```
simple KNXterminator
      gates:
            in: in_term;
            out: out_term;
endsimple
```

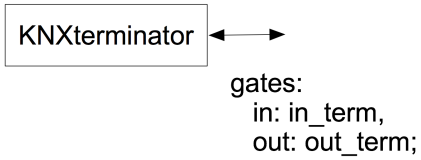


Figure 5.17: KNXterminator

A typical example for the usage of *KNXterminator* is when simulating or debugging a small bus with a line coupler attached. When creating a line, the gates of the line coupler are defined as to connect to a backbone to form a larger network. Terminating the *KNXlinecoupler* with the module *KNXterminator* allows simulating the bus and routing functions without connecting it to a backbone bus.

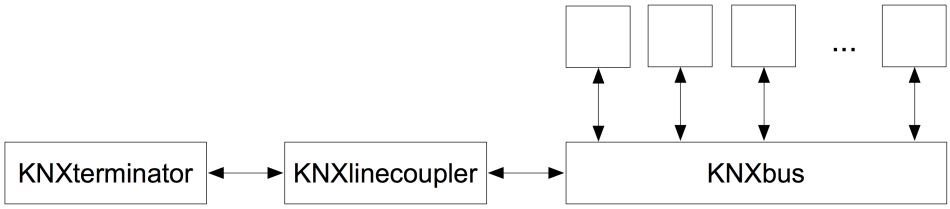


Figure 5.18: KNXterminator connected to KNXlinecoupler

5.4 KNX/EIB Lines

A KNX/EIB line is defined as a compound module in the NED file which consists of a KNXbus and typically of one or multiple KNXdev modules and a UserSim connected to the KNXdev modules:

```
module KNX_line_1
      parameters:
            numPorts: numeric const;
      gates:
            // connectors to the Line Coupler (LC)
            in: line_in[];
            out: line_out[];
      submodules:
            bus_1: KNXbus;
                  parameters:
                        txRate = "9600";
            dev_1_1: KNXdev;
                  parameters:
                        address = "1.1.1",
                        table = "1/1/2=lightA;0/0/0=lightALL",
                        application = "switch1";
            dev_1_2: KNXdev;
                  parameters:
                        address = "1.1.2",
```

```
                table = "1/1/2=light;1/1/7=light",
                application = "light1";
        dev_1_3: KNXdev;
            parameters:
                address = "1.1.3",
                table = "1/1/7=light",
                application = "light2";
        dev_1_4: KNXdev;
            parameters:
                address = "1.1.4",
                table = "1/1/7=lightB",
                application = "switch2";
        user: UserSim;
    connections:
        dev_1_1.out_bus --> bus_1.in++;
        dev_1_1.in_bus <-- bus_1.out++;
        dev_1_2.out_bus --> bus_1.in++;
        dev_1_2.in_bus <-- bus_1.out++;
        dev_1_3.out_bus --> bus_1.in++;
        dev_1_3.in_bus <-- bus_1.out++;
        dev_1_4.out_bus --> bus_1.in++;
        dev_1_4.in_bus <-- bus_1.out++;
        // Line Coupler connection
        line_out++ <-- bus_1.out++;
        line_in++ --> bus_1.in++;
        //User simulation layer
        dev_1_1.in_user <-- user.out++;
        dev_1_2.in_user <-- user.out++;
        dev_1_3.in_user <-- user.out++;
        dev_1_4.in_user <-- user.out++;
endmodule
```

The example describes a line with 4 devices, `dev_1_1` to `dev_1_4`. Each of the devices needs

to be configured with the appropriate set of parameters *address*, *table* and *application* as defined in their respective definitions. The defined gate `line_in[]` and `line_out[]` is used to connect to the line coupler. Although a line only connects to a single line coupler, the gate is set as to be of variable size due to enhanced flexibility when using the same definition to describe a main line or a backbone line. A main line and a backbone line will connect to multiple line couplers. The parameter *numPorts* describes how many line couplers are being used in the respective segment where *numPorts* = 0 describes a connection to one line coupler. In the `connections` section all devices need to get connected to the bus, the line couplers are transparently connected to the gate of the line and the devices are connected to the *UserSim* module.

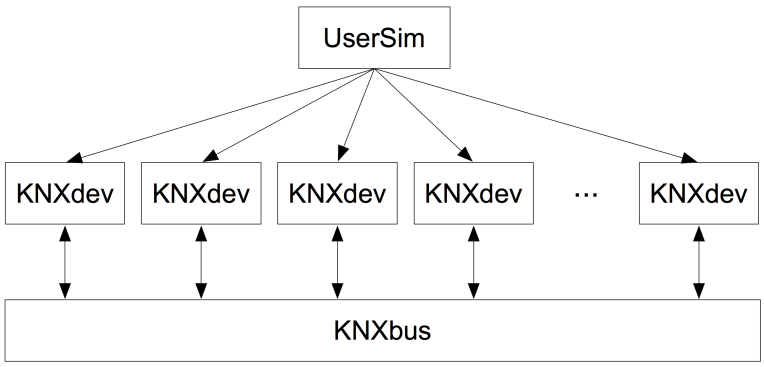


Figure 5.19: KNX/EIB line module structure

The representation of the KNX/EIB line in OMNeT++:

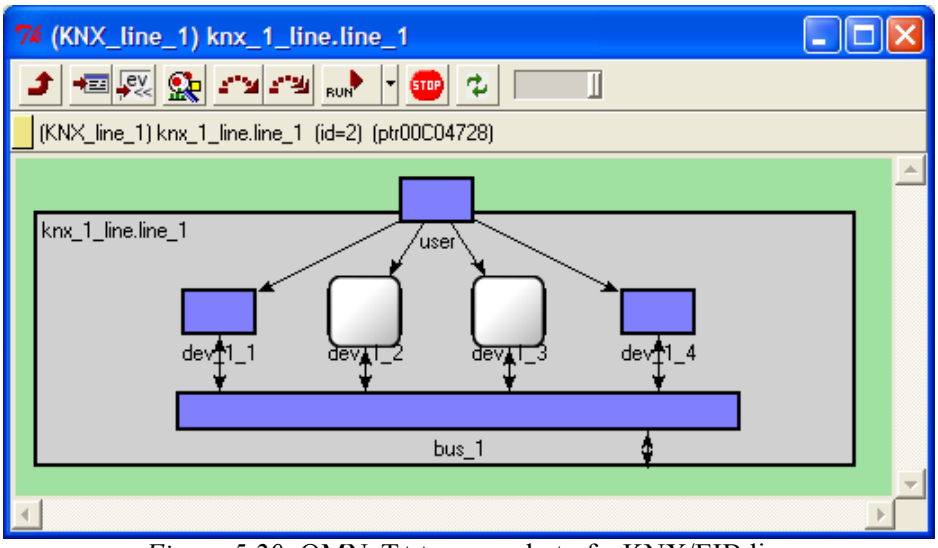


Figure 5.20: OMNeT++ screenshot of a KNX/EIB line

5.5 KNX/EIB Network

Similar to KNX/EIB *line*, the KNX/EIB *network* is only defined in NED language:

```
module KNX_network
      submodules:
            line_1: KNX_line_1;
                  parameters:
                        numPorts = 1;
            lc_1: KNXlinecoupler;
                  parameters:
                        address = "1.1.0", routing = "1/1/2";
            line_2: KNX_line_2;
                  parameters:
                        numPorts = 1;
            lc_2: KNXlinecoupler;
                  parameters:
                        address = "1.2.0", routing="1/1/7";
```

```
        line_3: KNX_line_3;
            parameters:
                numPorts = 1;
        lc_3: KNXlinecoupler;
            parameters:
                address = "1.3.0", routing="";
        line_backbone_1: KNX_backbone_1;
            parameters:
                numPorts = 3;
    connections:
        // Connect line couplers to the lines
        line_1.line_in++ <-- lc_1.out_line;
        line_1.line_out++ --> lc_1.in_line;
        line_2.line_in++ <-- lc_2.out_line;
        line_2.line_out++ --> lc_2.in_line;
        line_3.line_in++ <-- lc_3.out_line;
        line_3.line_out++ --> lc_3.in_line;
        // Connect line couplers to backbone
        lc_1.in_backbone <-- line_backbone_1.line_out++;
        lc_1.out_backbone --> line_backbone_1.line_in++;
        lc_2.in_backbone <-- line_backbone_1.line_out++;
        lc_2.out_backbone --> line_backbone_1.line_in++;
        lc_3.in_backbone <-- line_backbone_1.line_out++;
        lc_3.out_backbone --> line_backbone_1.line_in++;
endmodule
```

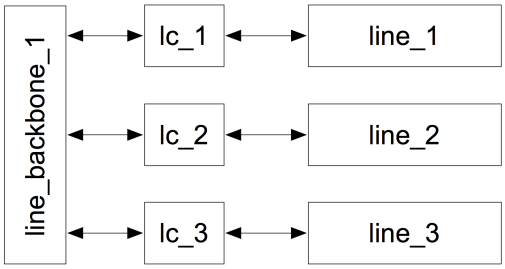


Figure 5.21: KNX/EIB backbone with attached line couplers and corresponding lines

The complete KNX/EIB network consists of 3 lines `line_X` and the corresponding 3 line couplers `lc_X` and the backbone line that connects the 3 lines together. Note that as described in Section 5.4 the *numPorts* parameter needs to be set correctly to the number of line couplers a line connects to. In the `connections` section the lines are connected with their line couplers and all line couplers are connected to the backbone.

5.6 What needs to be simulated

The simulation shall cover the following KNX/EIB characteristics:

- KNX/EIB lines and devices are resembling a real-word scenario and implement functionality that represents existing applications (switches, light actuators) and applications that are subject to current research.
- The KNX/EIB network must be freely designable within the KNX/EIB specifications to build test scenarios that range from a simple single line to a multi-area network that resembles a building automation network. Devices can be added to the lines as well as to the main lines and backbone lines. The line couplers and backbone couplers can be either configured as non-routing (with an empty routing table) or configured with routing functionality.
- KNX/EIB frames are routed through the network depending on the routing tables.
- The generated KNX/EIB layer 2 frame's parameters and data are set at the sender's side and are interpreted correctly at the receiver side.
- The load on a KNX/EIB line can be measured over time by observing the generated

frames on the bus.

- The routing function of the line couplers and backbone couplers can be monitored to investigate the network behaviour.
- The KNX/EIB devices can be activated by means from outside the network to trigger device activity and e.g. simulate a user interacting with the device. The user simulation can trigger events on
    - a one-time basis or
    - by setting a time interval and repeat count, where, optionally, an exact time or a broader time interval can be set.
- Application data can be parameterised to be sent to the device to trigger different device application functionality.

Not all protocol details of KNX/EIB will be covered by the simulation. There are some limitations on the implementation of the simulation of the KNX/EIB protocol:

- Acknowledgement frames are not sent back to the sender. Instead the time the acknowledgement frame is consuming the bus is considered in the bus load calculations. This is not necessarily a limitation as KNX/EIB is very strict in the timing on sending back acknowledgement frames.
- Checksums for frames are not being calculated with their real checksum. Instead a standard value is set for the checksum value when the member function is called.
- When a device sends a frame, it will receive a *busy* signal from the bus if the bus is currently occupied. The device will then retry to send the frame after a fixed waiting period of 125 bits time.

6 Purpose of the Simulation Model

6.1 EIBsec Protocol Extension

Building the structure of the OMNeT++ model allows to run simulations of a KNX/EIB based network, measure the load of KNX/EIB lines, measure the throughput at the line couplers and bus couplers and conclude on the performance behaviour of such a network.

As mentioned in the introductory chapters, KNX/EIB does not feature a secure transmission of frames in regard to data confidentiality, data integrity or data freshness, where, shortly described,

- *data confidentiality* describes the avoidance of disclosure of confidential information,
- *data integrity* means the prevention of sending data by unauthorised devices and detection of sending modified data and
- *data freshness* guarantees that the data sent by a device is valid at the current point in time, preventing the re-insertion of data at a later point in time.

EIBsec [03, 19], a security model to enhance KNX/EIB building automation systems, has been proposed which enhances KNX/EIB data communication in the security aspects mentioned above. It introduces an *Advanced Coupler Unit* (ACU) which is contained in each network segment that features secure communication. ACUs replace the standard KNX/EIB line couplers or backbone couplers or are realised as additional stand-alone devices.

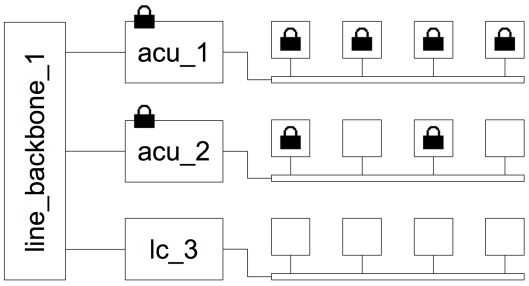


Figure 6.1: Secure line, mixed line and insecure line

The OMNeT++ model will be used to investigate how much additional traffic is being introduced to the data communication when implementing secure KNX/EIB lines. Causes for

additional traffic are:

- KNX/EIB frames that utilise the maximum frame size for secure communication,
- traffic introduced by key management, i.e. key revocation from the ACU to the devices, but also
- management communication during the start-up phase of the network increases because of the key distribution mechanism.

Specifically the *key revocation* mechanism [03] is implemented in OMNeT++. The protocol can be outlined as:

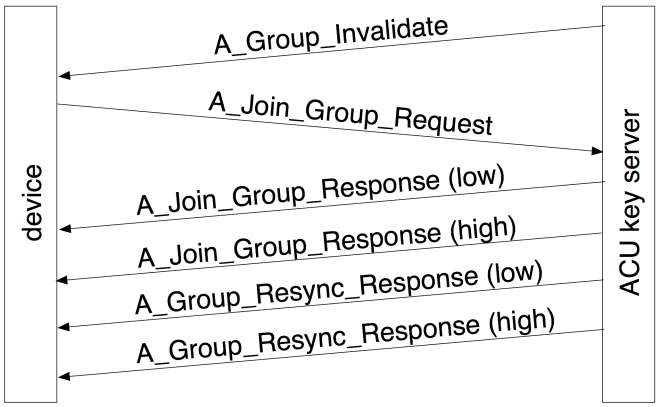


Figure 6.2: ACU key revocation

The ACU key server sends a key revocation message *A_Group_Invalidate* to a group address. The devices individually contact the ACU to request the new group key and group counter value by sending an *A_Join_Group_Request* to the ACU. The ACU is responding with 4 frames containing the high and low value of the group key (*A_Join_Group_Response*) and the high and low value of the initial group counter value (*A_Group_Resync_Response*).

The means available to measure KNX/EIB bus load (see *KNXbus Section 5.3.4*) will be used to compare the network load of insecure and secure lines. The following topics will be measured by the simulation model:

- The additional network load due to secure communication,
- the additional network traffic caused by the ACU's key management when operating the network communication.

The expected results should show an increased network traffic at the model that has the EIBsec protocol extension applied. As a second data set we monitor the attempt to write to the bus when the bus is busy. The number of busy waits when accessing the bus is also a good indicator on the load of the network. The results of both simulation runs – classic KNX/EIB and KNX/EIB with EIBsec - are compared to each other and the overhead traffic will be shown.

6.2 Configuring the Simulation

For the intended simulation a large scale network needs to be defined to measure the network load on the segments under different conditions and configurations.

The defined network consists of 10 devices per line, 4 lines per main line and 2 main lines, connected via a backbone line. This results in 80 devices which will be configured with 3 KNX/EIB groups per line with 5-10 members per group.

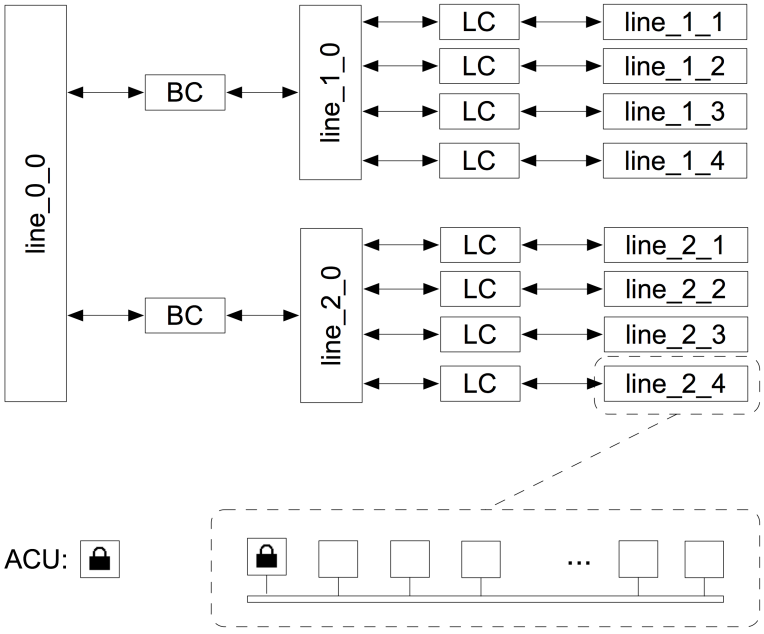


Figure 6.3: The test network

Note that the devices need to be configured individually, similar to the configuration done when installing a real KNX/EIB network. A tool could be created that supports the automatic creation of the framework of the NED file but details like the KNX/EIB individual addresses, group addresses and application association tables need to be configured on a per-device basis.

To simulate some network activity, traffic will be induced via the *UserSim* module. By configuring the *UserSim's* configuration file, devices will put constant load on the network that is

- bound to the line where the communication is being initiated, not passing the line coupler,
- bound to the main line so frames traverse through the line couplers but will not pass the backbone couplers,
- traversing the backbone line to its destination on other main lines, traversing line couplers and backbone couplers,
- broadcasted to all devices, thus being distributed throughout the whole network.

To gather representative data the network must be loaded to a certain extent. The load that is imposed should be high enough to be recognised in the measurements but not over-utilising the bus. For sure a few simulated activations of a light switch will not load the bus, even if 20 switches are present on the line. The UserSim needs to cover also periodically talking devices that communicate regularly like e.g. a temperature sensor.

Some lines will be configured to have a more constant load on the bus, other lines will be stressed with transmission peaks where the peaks can be sourced by either a single or multiple devices. Looking at the configuration possibilities, the constant load is easy to apply as only a single entry in *UserSim* allows constant load defines: `0/0.1/100` puts every 0.1 seconds a frame on the bus. 100 frames will be created, resulting in a bus that will be loaded with frames for 10 seconds. To increase the network load, simply the parameter *interval* needs to be adjusted to a shorter time frequency, e.g. 0.05. This doubles the creation of frames. To load the bus for a longer period of time the *repeat* parameter can be increased. The total time during which packets are being created is *interval* * *repeat*. To run a simulation for 15 minutes, the `UserSim.txt` entries look like:

```
0/1.0/900; switch1; lightA
```

To stress the network with transmission peaks, the following entries can be used, also considering 90 seconds of simulation time for a single device:

```
0/0.1/300    ; switch1; lightA  // 30 seconds load
```

```
120/0.05/240 ; switch1; lightA  // 12 seconds high load
240/0.025/240; switch1; lightA  //  6 seconds peak load
120/0.05/240 ; switch1; lightA  // 12 seconds high load
600/0.1/300  ; switch1; lightA  // 30 seconds load
```

The line segments will measure their network load by using the *KNXbus* functionality. Each *KNXbus* tracks its load in the OMNeT++ data file `omnet.vec` collectively (which contains all bus data) and in `<bus_name>.log` and `<bus_name>.csv` text files individually. The `.log` file allows for easier investigation on the bus activities and the `.csv` file allows easier data representation with external programs as it contains the bus load pre-processed as comma separated values.

6.3 Simulation Test Cases

To test the behaviour of the KNX/EIB network, 3 test cases are set up:

- insecure communication: this represents a traditional KNX/EIB network with devices that communicate without encryption and without a key server,
- secure communication with line ACUs: each KNX/EIB line features an ACU serving as a key server. The devices communicate securely by encrypting their messages. The key server will revoke the encryption key for the devices on the same line and handle the requests of those devices for requesting new keys.
- secure communication with central ACU: one central key server is placed on the backbone line. The devices communicate securely by encrypting their messages. The central key server will revoke the encryption key for all devices on the network and handle requests of the devices for requesting new keys.

The above mentioned cases are set by using three distinct UserSim files which are outlined in Appendix 9.2. The `UserSim.txt` files have one common section for the activity on the network and one separate section for the secure communication which defines the periodic key revocation interval.

OMNeT++ is run by setting the corresponding `UserSim.txt` file as the input parameter for

the simulation. The result of the simulation are the `.csv` files created by the *KNXbus* modules showing the network load of the respective line. The network load graphical representation of some example lines (line, main line and backbone line) will be shown as the result of the simulation.

7 Results of the Simulation

As a result of the simulation, the following graphs will be presented that show the

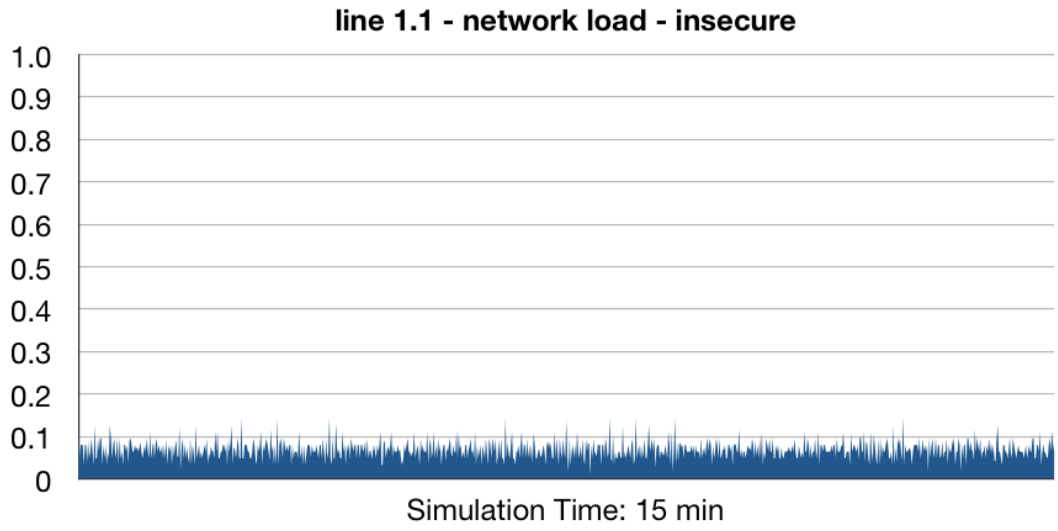
- *network load*: the load of the network in percent. Theoretically a 100% network load can be achieved. Practically timing constraints like wait times depending on the protocol will limit the achievable network load.
- *busy response*: when the KNXstack tries to send a frame on the bus, the bus may be busy sending another frame. The *busy response* on send indicates how many times during the one second interval the bus was busy when a device tried to send data. It is an indication on the amount of frames that are in the backlog to be sent.

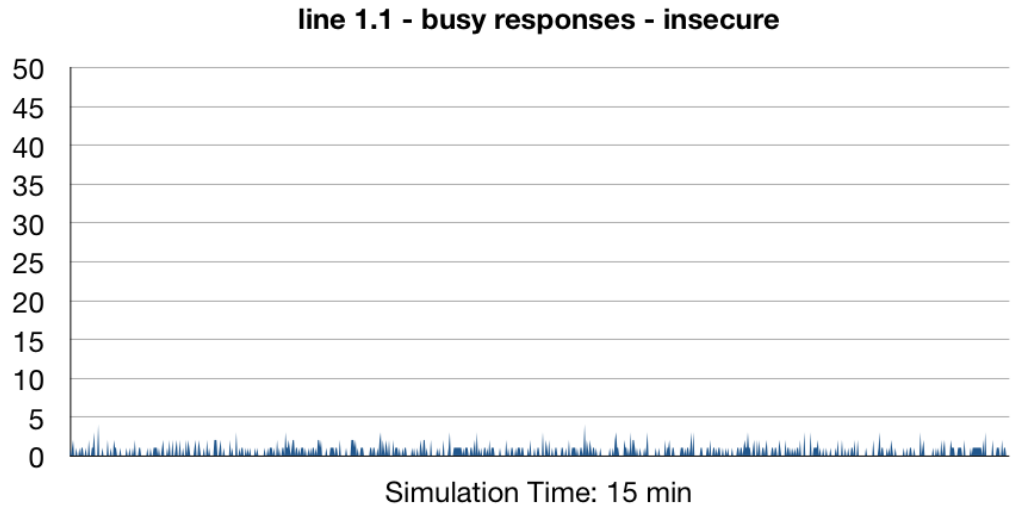
To compare the insecure and the secure network, the delta graph shows the additional load that is put on the network due to secure data communication.

7.1 Simulation Results for Insecure Communication

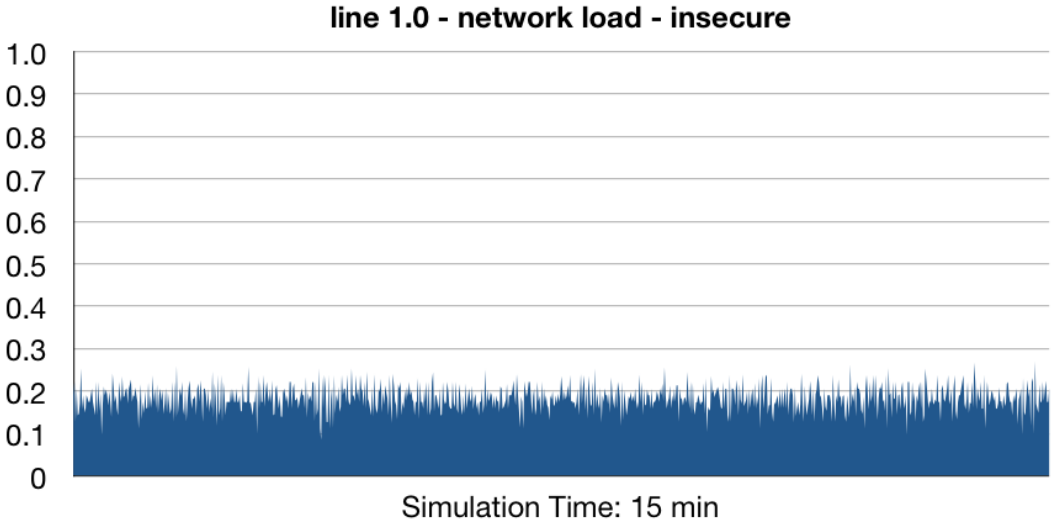
This network configuration features classic KNX/EIB configuration with no security.

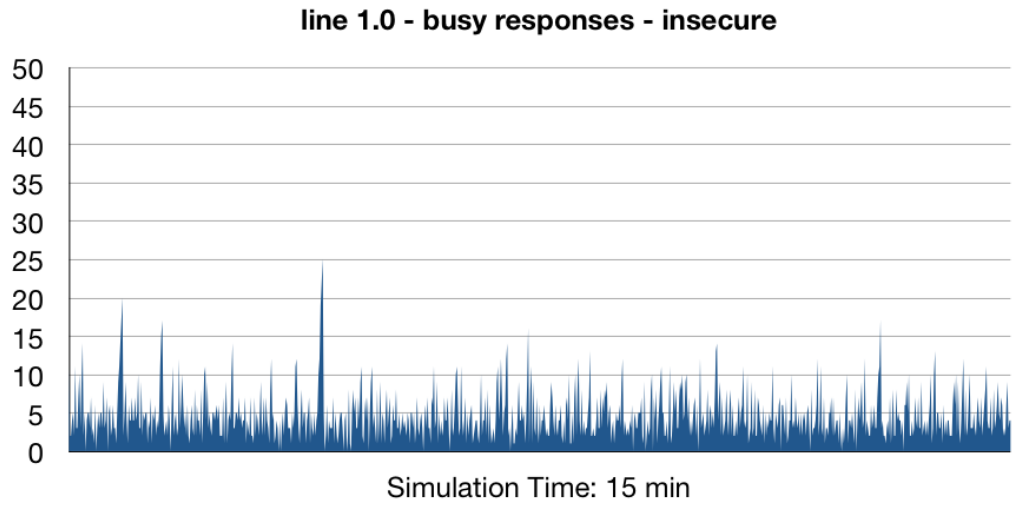
Results for line line_1_1:



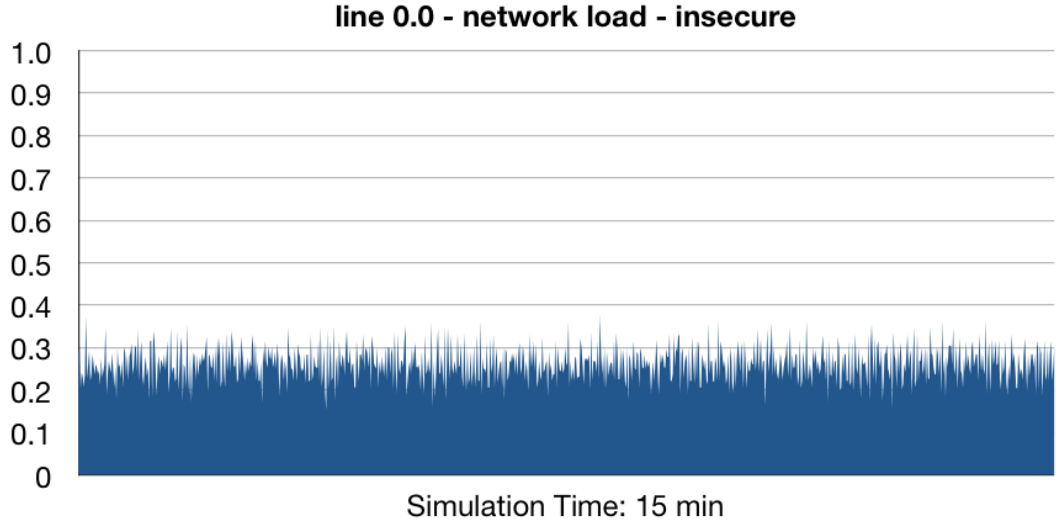


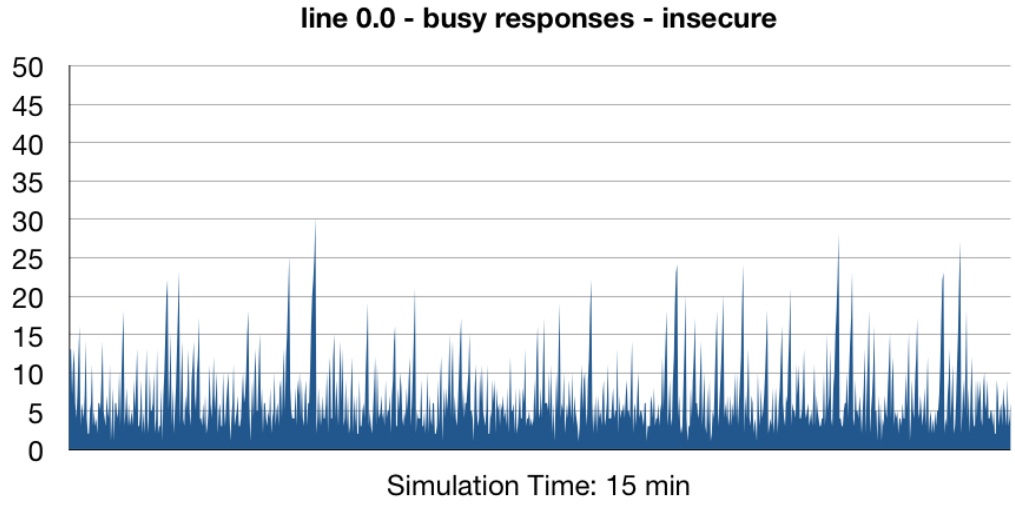
Results for main line line_1_0:





Results for the backbone line:

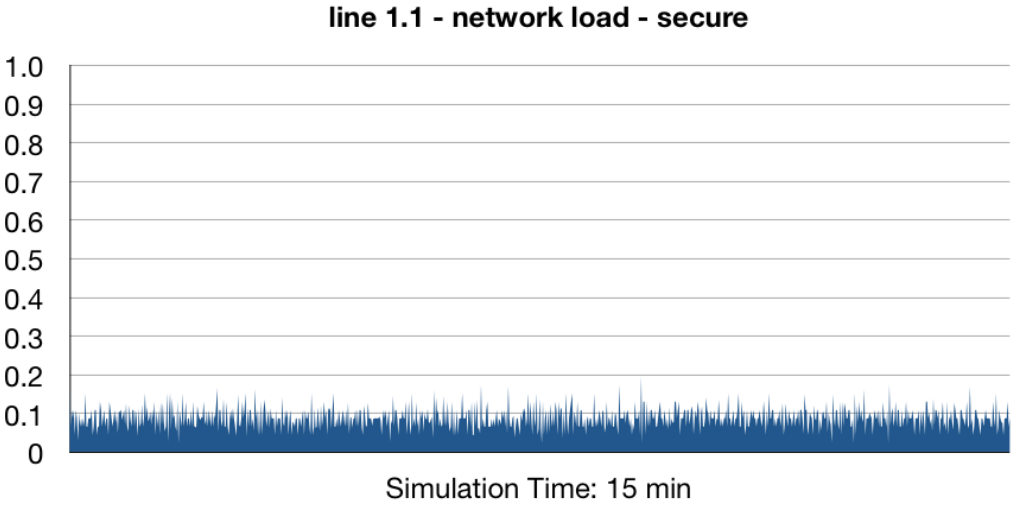


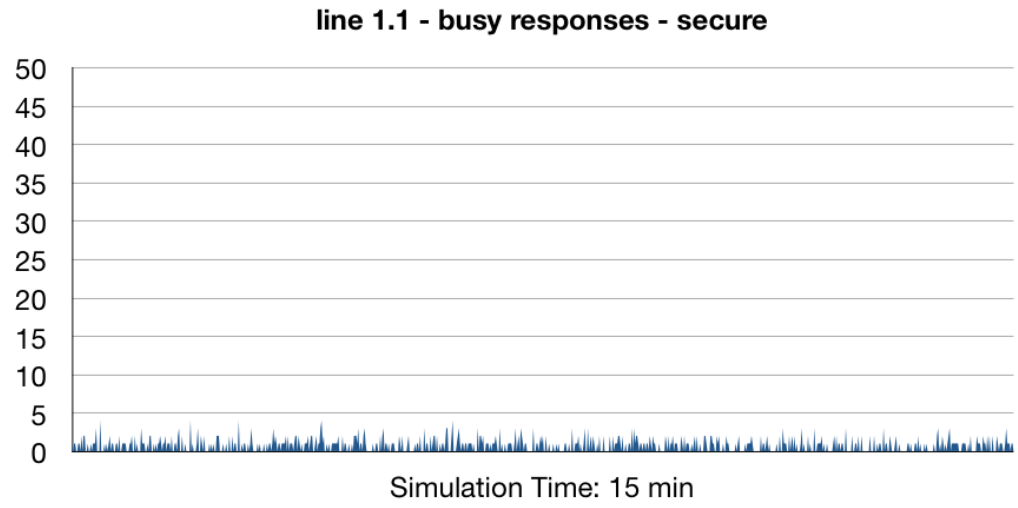


7.2 Simulation Results for Secure Communication with Line ACUs

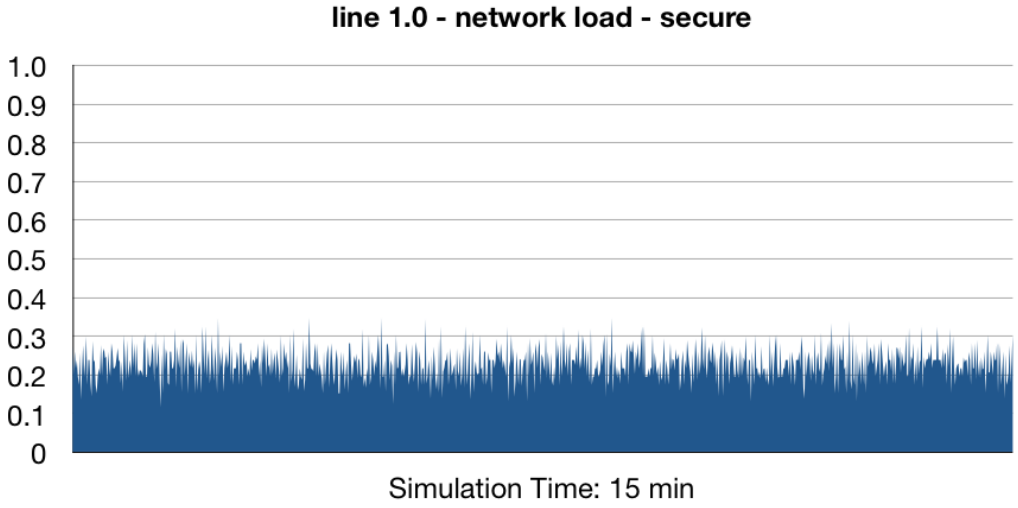
This network configuration implements the EIBsec proposed ACUs, one ACU per KNX/EIB line. The devices on the line have their associated ACU configured.

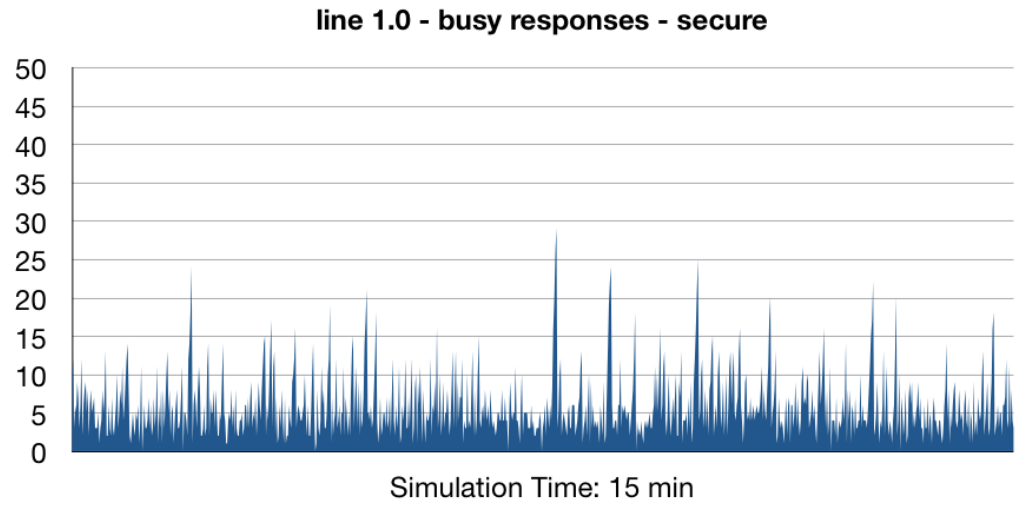
Results for line line_1_1:



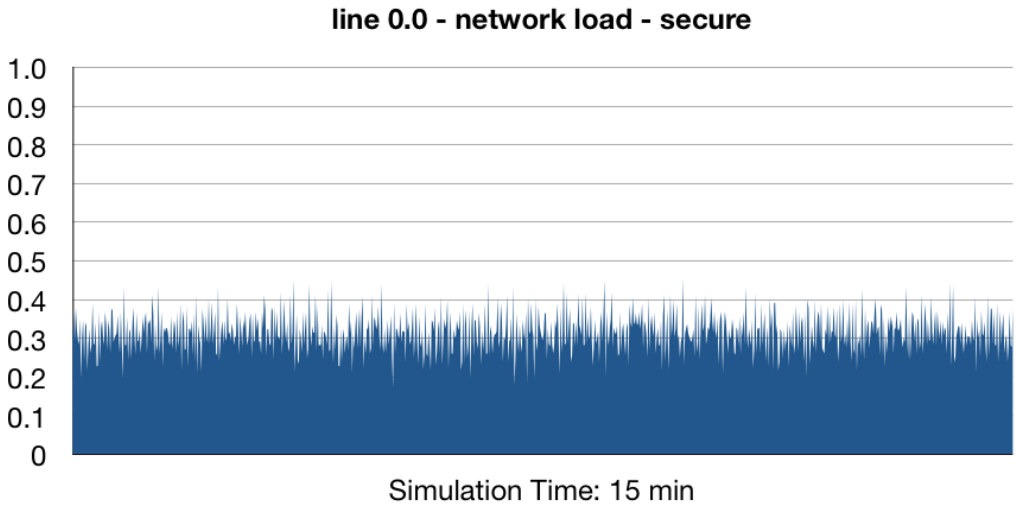


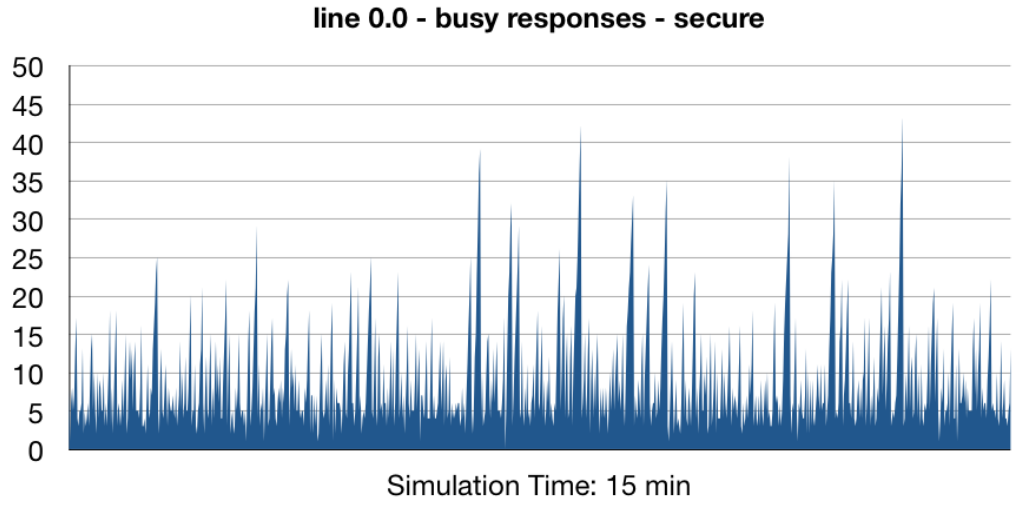
Results for main line line_1_0:





Results for the backbone line:

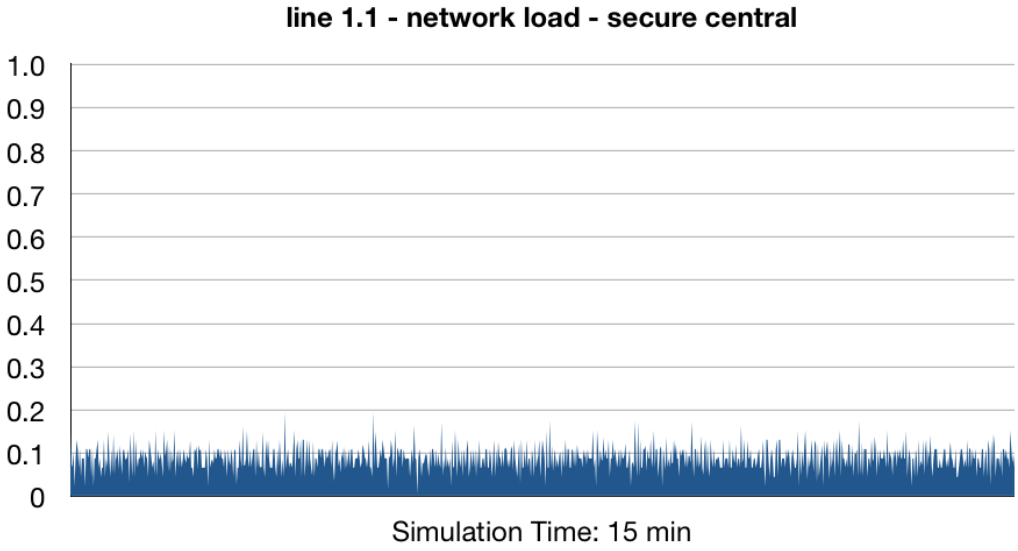


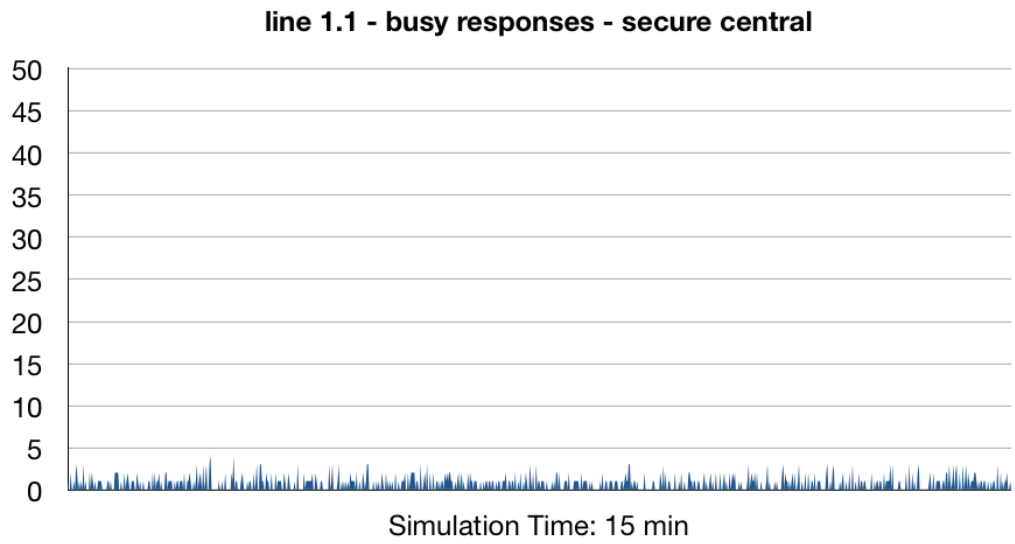


7.3 Simulation Results for Secure Communication with a Backbone ACU

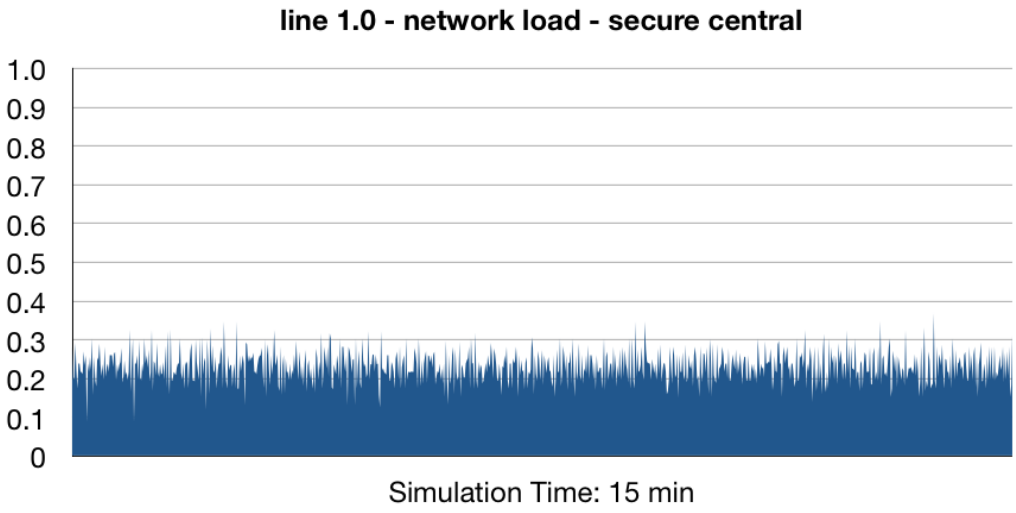
In contrast to the previous configuration with one ACU per KNX/EIB line, this approach configures one single central ACU on the backbone line of the KNX/EIB bus.

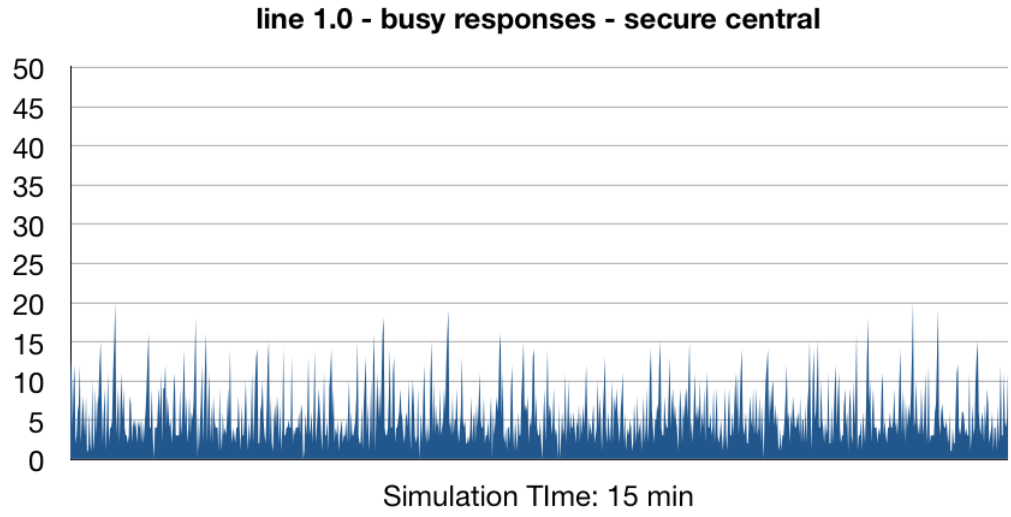
Results for the line line_1_1:



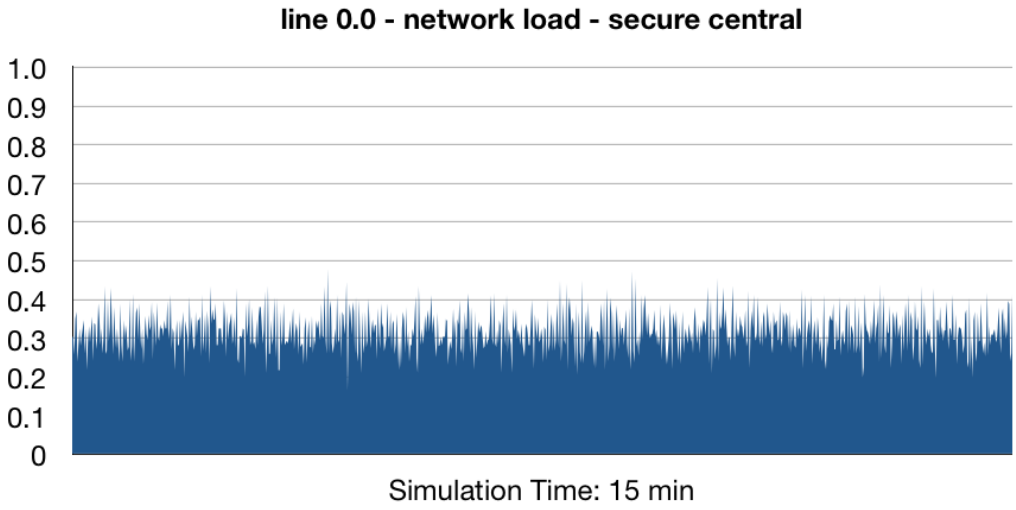


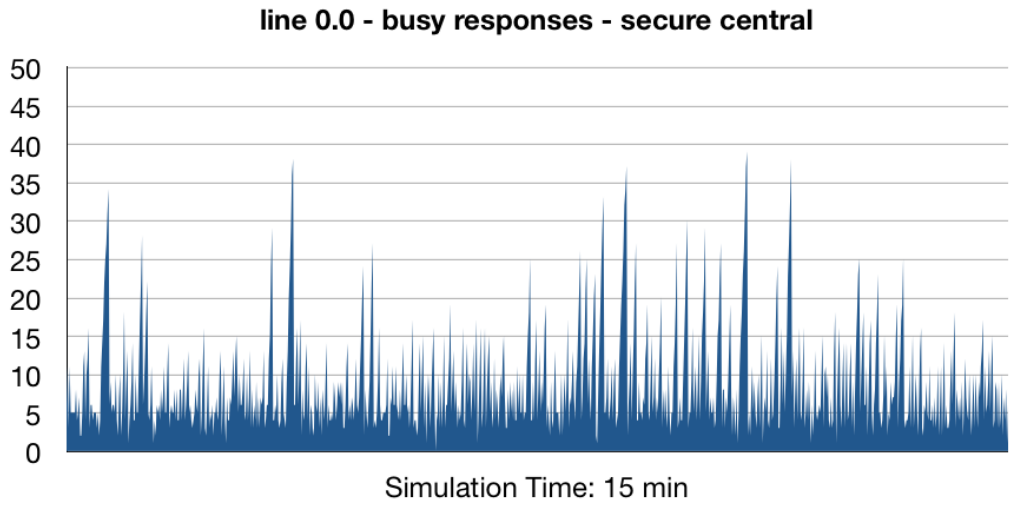
Results for the main line line_1_0:





Results for the backbone line:





7.4 Summary of Simulation Results

The results of the different simulation configurations can be summarised by showing the average and maximum *network load* of the lines and the average and maximum *busy responses* of the lines.

network load		insecure	secure	secure central
line 1.1	average	0.0677	0.0874	0.0877
	maximum	0.1453	0.1941	0.1941
line 1.0	average	0.1821	0.2276	0.2291
	maximum	0.2683	0.3450	0.3666
line 0.0	average	0.2608	0.3131	0.3151
	maximum	0.3745	0.4517	0.4754

busy responses		insecure	secure	secure central
line 1.1	average	0.5617	0.7164	0.6963
	maximum	4	4	4
line 1.0	average	4.6485	5.8420	5.6874
	maximum	25	29	20
line 0.0	average	6.8776	8.9166	8.6685
	maximum	30	43	39

7.5 Analysis of the Results

Looking at the simulation results and comparing the different network security configurations, the following comparison tables show the growth of increased network communication. The *growth per interval* shows the growth (average and maximum) on a per-interval basis, taking the same interval t_i of the simulation time for comparison. The *growth total* shows average and maximum growth over the full simulation time of 15 minutes. *Network load* has a value range of either [0..1] or [0%..100%], depending on the representation. *Busy responses*' unit is responses/second, the growth is indicated in percent.

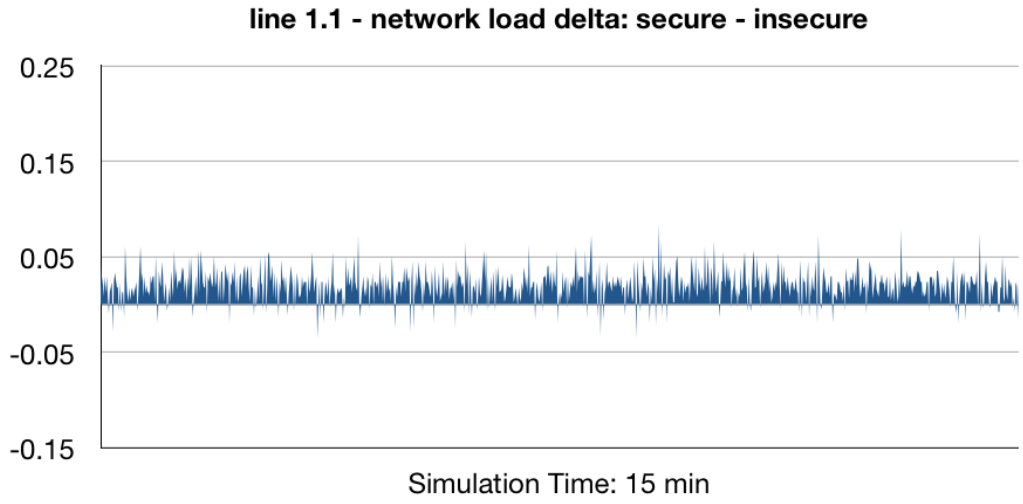
network load		growth insecure → secure (per interval)	growth secure → secure central (per interval)	growth insecure → secure (total)	growth secure → secure central (total)
line 1.1	average	2.0%	0.0%	29.0%	0.4%
	maximum	8.4%	12.9%	33.5%	0.0%
line 1.0	average	4.6%	0.1%	25.0%	0.6%
	maximum	17.8%	15.1%	28.6%	6.2%
line 0.0	average	5.2%	0.2%	20.1%	0.6%
	maximum	21.5%	16.6%	20.6%	5.2%

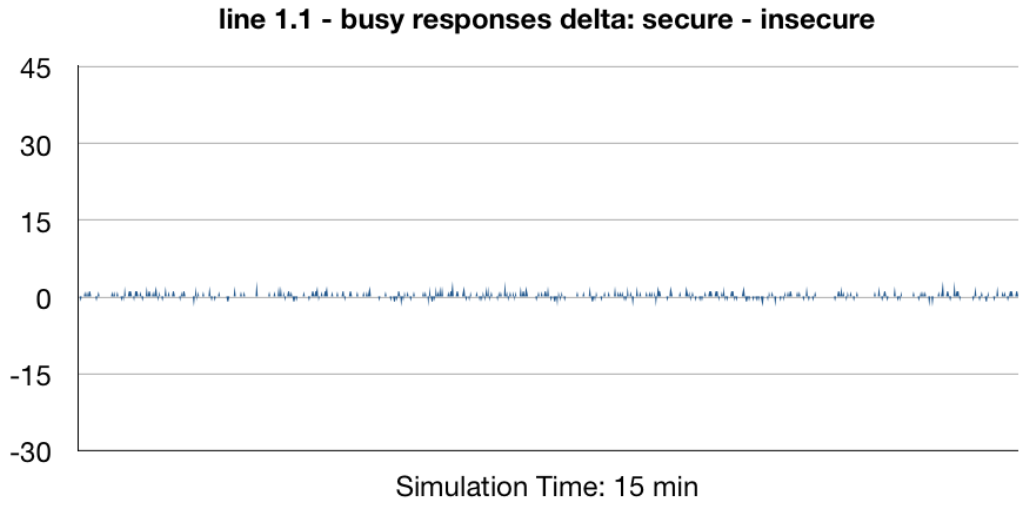
busy responses		growth insecure → secure (per interval)	growth secure → secure central (per interval)	growth insecure → secure (total)	growth secure → secure central (total)
line 1.1	average	0.15	-0.02	27.52%	-2.80%
	maximum	3	4	0.00%	0.00%
line 1.0	average	1.19	-0.15	25.68%	-2.65%
	maximum	25	16	16.00%	-31.03%
line 0.0	average	2.04	-0.25	29.65%	-2.78%
	maximum	37	33	43.33%	-9.30%

7.5.1 Comparison of Insecure and Secure Communication

Looking at the insecure and secure communication, we can see a general increase of the average and maximum *bus load*. That is due to the increased frame length of secure communication and the periodic key distribution mechanism of the ACUs.

On line level (e.g. line 1.1) *the bus load* is increasing by 2.0% on average, the maximum load increased by 8.4%. *Busy responses* from the bus are steady at a low level, the frames can be sent with little or no delay from the devices in all security modes.

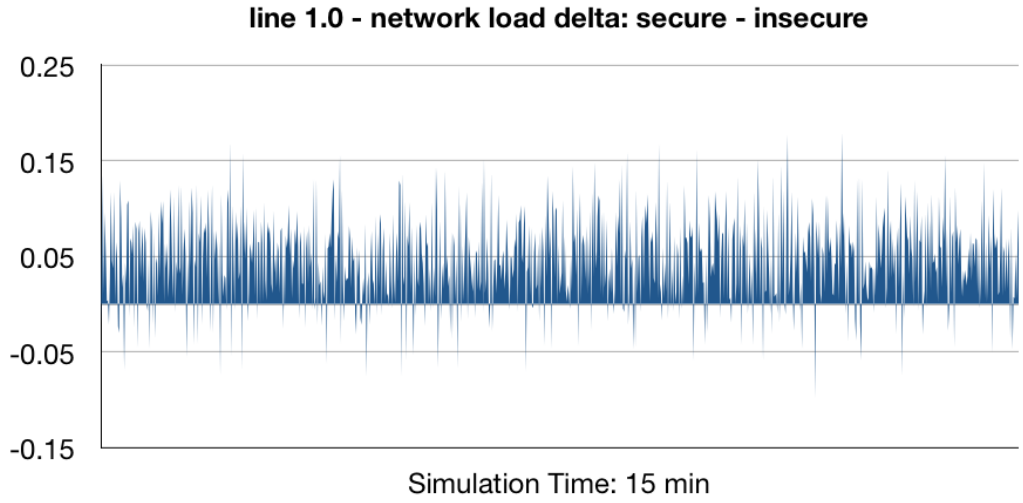


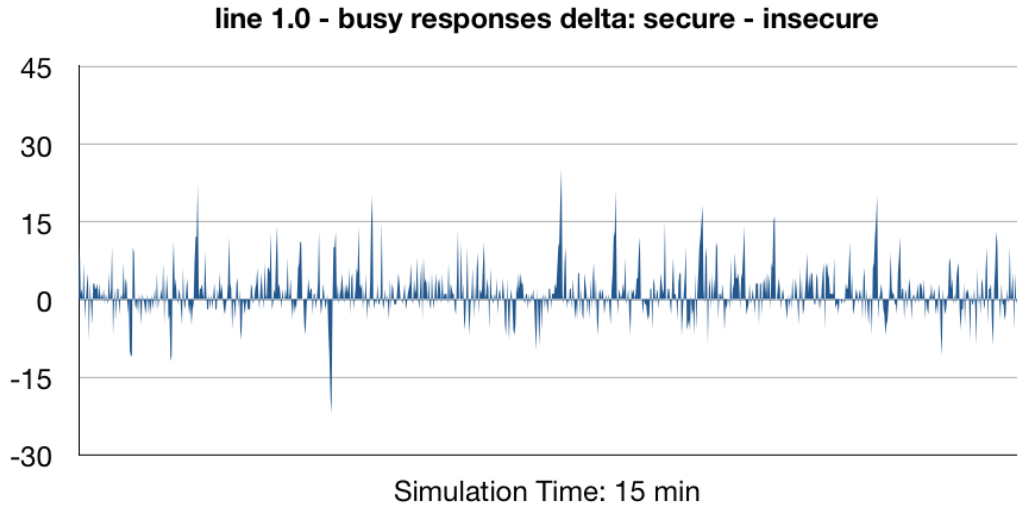


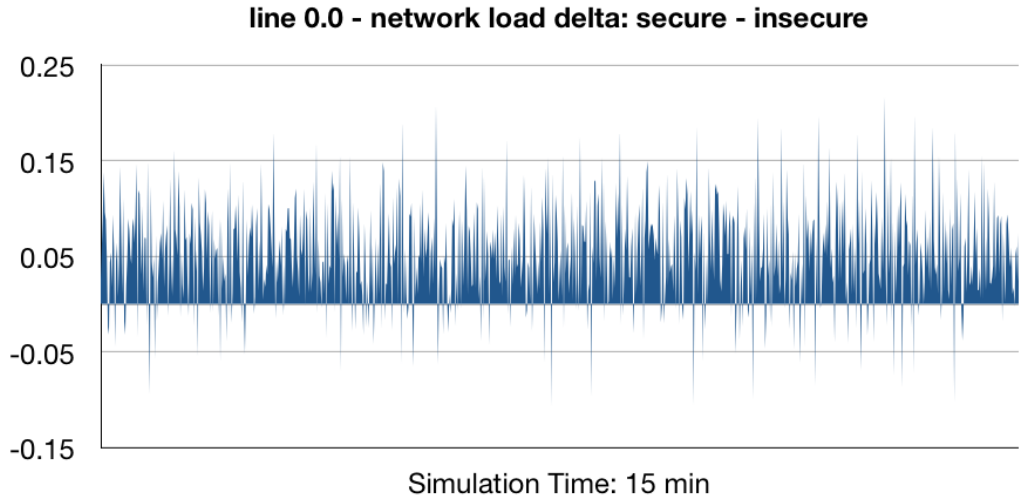
At main line level (line 1.0) the increase of the *bus load* is significantly higher (4.6% on average

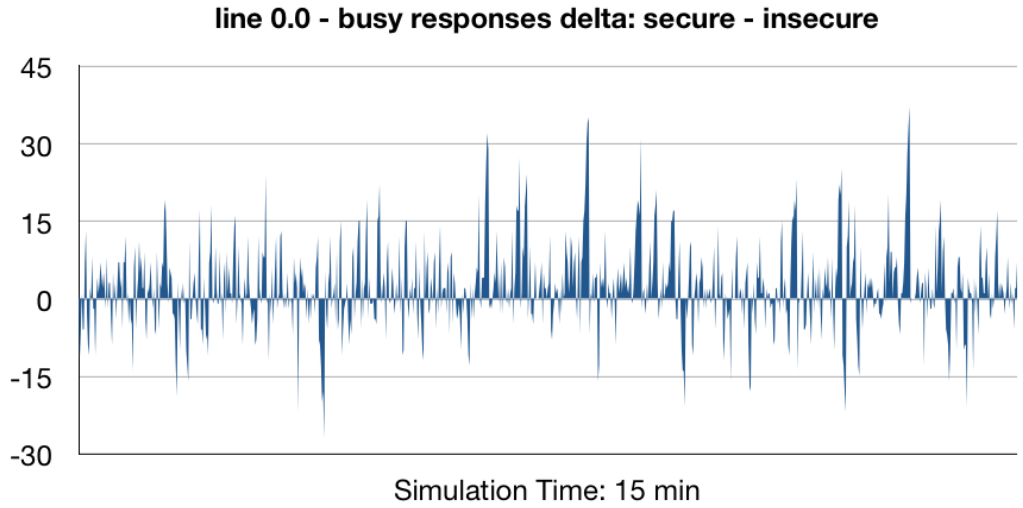
load, 17.8% on maximum load) than the increase on the line level.

This increase continues on the backbone line 0.0: average load per interval increases by 5.2%, the maximum load per interval by 21.5%. For *busy responses* the increase needs to be highlighted: 29.65% growth from insecure to secure mode counting the total *busy responses* over the simulation time indicate a oncoming bottleneck on the backbone. This is accompanied by a peak of *busy responses* growth of 43.33% compared to insecure network configuration.



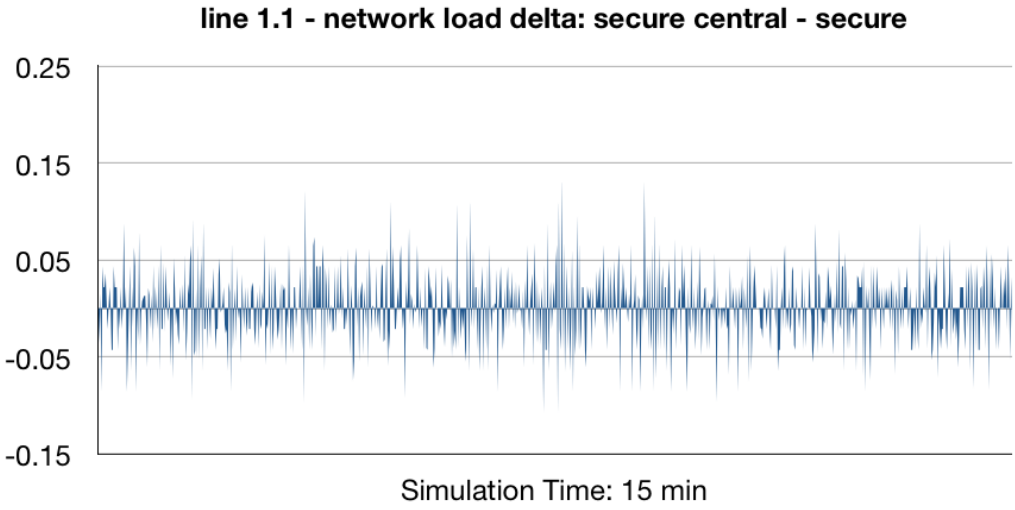


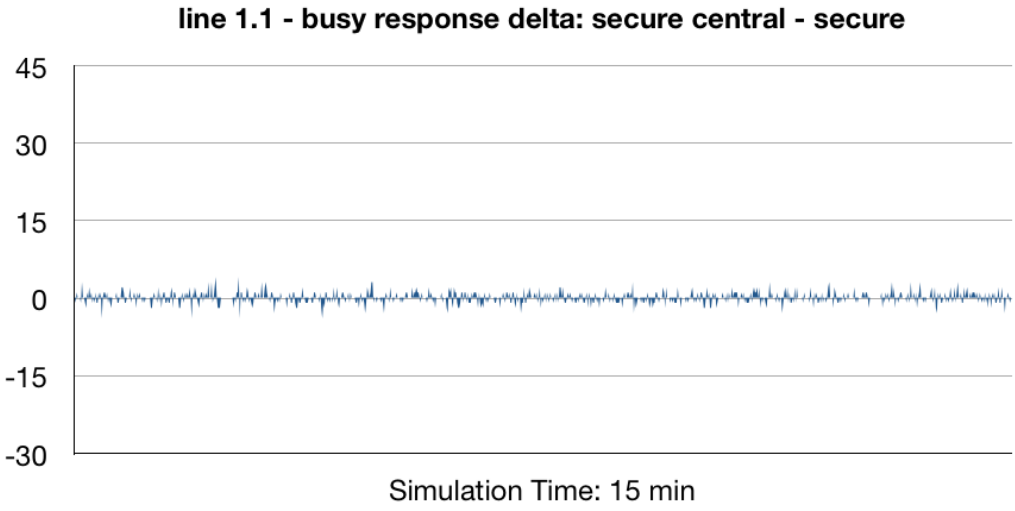




7.5.2 Comparison of Secure Communication with Line ACUs and Secure Communication with a Backbone ACU

For all lines – lines, main lines and the backbone line – a negative growth can be measured for *busy responses* when replacing the line ACUs with a backbone line ACU. The decrease is between -2.65% and -2.80% although the *bus load* is slightly increasing between 0.4% and 0.6% for the average load and 0.0% and 6.2% for the peak *bus load*.





As an example, the graphs for line 1.1 illustrate that the total growth of *network load* is near zero (0.4%) but that the difference per interval is varying much more. However, the *busy response* is lower in total growth (-2.8%) but shows very little variance.

7.6 Conclusion on EIBsec Implementation

The implementation of EIBsec puts significantly more load on the KNX/EIB network, especially the backbone line is prone to delays if it is operated at the same speed as the lines,

typically 9600 bits/second at KNX/EIB. The simulation environment has been defined with devices which are very frequently sending frames over the network. Any KNX/EIB implementation that does not require high volume communication at all times but needs to secure its communication will not encounter this possible bottleneck situation.

The decision whether to implement the ACUs in the lines or put the ACU on the backbone line is a matter of which security requirements need to be implemented. According to the simulation environment's results, the network load does not differ significantly when comparing placing the ACU on the backbone line or placing multiple ACUs within the lines. If functionality like network segment isolation is requested, a line based ACU is required. In cases where secure communication by implementing EIBsec's encryption is required, an ACU placed on the backbone line offers a more centralised approach without a significant effect on the network load. However, the single ACU does not provide redundancy and is a single point of failure.

8 Pitfalls in the Modelling

Several decisions had to be made during the design of the OMNeT++ modules and their interaction. Some models are reflecting reality more precisely and allow more granular design of a network. However, it needs also to be considered that when designing a large KNX/EIB network, the amount of configuration work gets tremendous with all configuration details necessary.

As an example the KNX/EIB bus can be taken. A first module design defined a *KNXnode* module that – translated to reality – would represent the connector where a KNX/EIB device is attached to the bus. The definition is more granular than the introduced *KNXbus*: *KNXnodes* are connected to each other bidirectionally, each connection from module to module can be separately configured (see Figure 8.2).

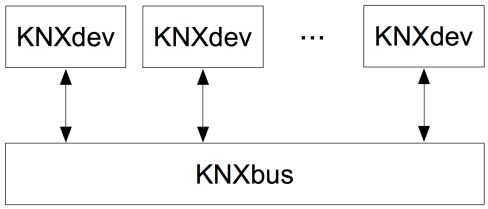


Figure 8.1: KNXbus with connected KNXdevs

In this case, the *KNXbus* is transformed in a double-linked chain of *KNXnodes*. The end nodes can be either be terminated with a *KNXterminator* simple module or, alternatively the node could be connected to itself to indicate a termination. This self-reference could be investigated at runtime by the module.

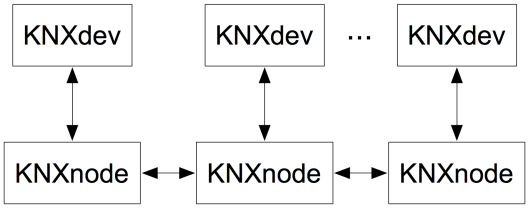


Figure 8.2: KNXnodes with connected KNXdevs

A detailed view on the *KNXnode* based structure, focusing on the device view, reveals that the module structure of the *KNXdev* is not changed, compared to the implemented structure. The *KNXappl* module connects as well to the *UserSim* module for receiving simulation events.

Similar to the *KNXbus*, the *KNXnode* does not require the KNX/EIB hardware device address as the frames are simply distributed to the next *KNXnode* and the *KNXstack* attached.

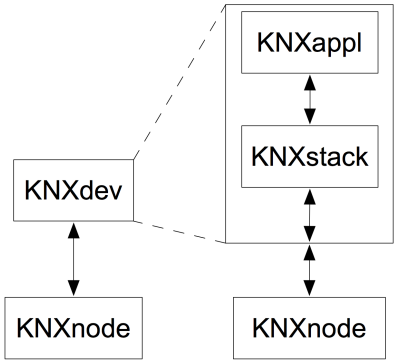


Figure 8.3: KNXnode connectivity detail

The final decision criteria on preferring the *KNXbus* structure to the *KNXnode* structure was the more flexible way how to attach devices to the bus. The reduced configuration requirements in the NED file allow a better overview of the line structure when looking at the OMNeT++ structure during simulation. Also evaluating how much traffic is transported over the line is easier to manage and calculate with the *KNXbus* structure.

9 Conclusion and Future Work

In this thesis, a generic simulation framework that can be used to simulate KNX/EIB based building automation networks has been presented. As a proof-of-concept, the KNX/EIB security extension EIBsec has been tested for its effect on the network.

However, not limited to these investigations the model can be used to verify other KNX/EIB protocol extensions. The generic structure of the simulation framework allows to extend it to other building automation network technologies like LonWorks [20] or ZigBee [21].

10 Bibliography

Chapter 1

[01] A Simulation Framework for Fault-Tolerant Clock Synchronization in Industrial Automation Networks; Fritz Praus, Wolfgang Granzer, Georg Gaderer and Thilo Sauter; in Proc. of 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA '07), pages 1465-1472, September 2007

Chapter 2

[02] Communication Systems for Building Automation Systems; Wolfgang Kastner, Georg Neugschwandtner, Stefan Soucek and H. Michael Newman; Proceedings of the IEEE, Vol. 93, No.6, June 2005

[03] Security in Networked Building Automation Systems; Wolfgang Granzer, Wolfgang Kastner, Georg Neugschwandtner and Fritz Praus; in Proc. 6th IEEE International Workshop on Factory Communication Systems (WFCS '06), pages 283-292, June 2006

[04] EIB: European Installation Bus; Wolfgang Kastner, Georg Neugschwandtner; Vienna University of Technology; CRC Press; Vienna, Austria, 2005

[05] EIBA, EIB Association; http://www.eiba.com/

[06] KNX Specifications, Version 1.1; Konnex Association, Diegem, Belgium, 2004

[07] EIB – Installation Bus System; Thilo Sauter, Dietmar Dietrich, Wolfgang Kastner (editors); Publicis, Munich, Germany, 2001

[08] Introduction to KNX and Konnex; Konnex Association, http://www.knx.org/, Diegem, Belgium, 2004

[09] Gebäudeautomation – Kommunikationssysteme mit EIB/KNX, LON und BACnet; Hermann Merz, Thomas Hansemann, Christof Hübner; Carl Hanser Verlag, Munich, Germany, 2007

[10] KNX Standard, Volume 3 – Systems Specification, Part 3 - Communication, Version 1.0; Konnex Association, Diegem, Belgium, 2001

Chapter 3

[11] http://pcl.cs.ucla.edu/projects/glomosim/ ; UCLA Computer Science Department,

University of California, Los Angeles, U.S.A.

[12] http://www.qualnet.com/ ; Scaleable Network Technologies Inc.

[13] QualNet Model Libraries and QualNet Product Family; Scaleable Network Technologies Inc., http://www.qualnet.com/

[14] http://www.isi.edu/nsnam/ns/ ; ns-2 Network Simulator; Information Sciences Institute, University of Southern California

[15] http://www.omnetpp.org/ ; OMNeT++ Discrete Event Simulation System, András Varga

[16] Using the OMNeT++ discrete event simulation system in education; András Varga; IEEE Transactions on Education, 1999

[17] http://www.omnest.com/ ; Simulcraft Inc.

Chapter 4

[18] OMNeT++ Discrete Event Simulation System User Manual; András Varga, http://www.omnetpp.org/, 2005

Chapter 6

[19] Security in Networked Building Automation Systems – Master's Thesis; Wolfgang Granzer, Vienna University of Technology, Vienna, Austria, 2005

Chapter 9

[20] Control Network Protocol Specification; ANSI/EIA/CEA 709.1, 1999

[21] ZigBee Specification; ZigBee Alliance, 2007

Further reading:

[R1] Protocol verification as a hardware design aid; David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang; IEEE International Conference on Computer Design: VLSI in Computers and Processors, pages 522-525, 1992

[R2] Formal Verification Methods; Eduard Cerny, Xiaoyu Song; Département d´informatique et de recherche opérationnelle (IRO); Course lecture “IFT6222 Spécification et vérification formelle”; Université de Montréal, Canada, 1999

[R3] System-architecture; Konnex Association, http://www.knx.org/, Diegem, Belgium, 2004

[R4] NS Simulator for beginners; Eitan Altman, Tania Jiménez; Lecture Notes 2003-2004; Univ. de Los Andes, Mérida, Venezuela and ESSI, Sophia-Antipolis, France, 2003

[R5] The ns Manual; Kevin Fall, Kannan Varadhan (editors); A Collaboration between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC, 2008

A Appendices

A.1 NED file

```
// Network definition file

//
// Starting here the module structure is designed
//
simple KNXbus
      // the KNX bus
      // implemented as a hub that forwards an incoming msg
      // to all other gates
      parameters:
            txRate: string;
      gates:
            in: in[];
            out: out[];
endsimple

simple KNXterminator
      // Terminator to a KNX line when no LineCoupler is
      // being used
      gates:
            in: in_term;
            out: out_term;
endsimple

simple KNXstack
      // implements the KNX protocol
      parameters:
            address: string, // KNX hw address of the device,
                             // e.g. "0.0.1"
```

```
            table:string;    // table for appl. addressing
      gates:
            in: in_bus;
            out: out_bus;
            in: in_appl;
            out: out_appl;
endsimple

simple KNXappl
      // implements an KNX application
      parameters:
            table: string,
            application: string;
      gates:
            in: in_appl, in_user;
            out: out_appl;
endsimple

simple KNXappl_gui
      // GUI feedback for KNXappl
      parameters:
            address: string,
            application: string;
endsimple

simple KNXlinecoupler
      // KNX bus line coupler for linking 2 bus segments.
      parameters:
            address: string,
            routing: string;
      gates:
            in: in_line, in_backbone;
            out: out_line, out_backbone;
endsimple
```

```
simple UserSim
      // User simulation, sends messages to KNXappl
      gates:
            out: out[];
endsimple

module KNXdev
      parameters:
            // address -> forwarder for stack address
            // application -> forwarder for application type
            address: string,
            table: string,
            application: string;
      gates:
            in: in_bus, in_user;
            out: out_bus;
      submodules:
            stack: KNXstack;
                  parameters:
                        address = address,
                        table = table;
                  display: "p=55,125;i=block/layer,cyan";
            appl: KNXappl;
                  parameters:
                        table = table,
                        application = application;
                  display: "p=55,55;i=block/process,grey";
            gui: KNXappl_gui;
                  parameters:
                        address = address,
                        application = application;
                  display: "p=90,45;b=20,20,rect";
      connections:
```

```
            // module gates to external
            out_bus <-- stack.out_bus display "m=s";
            in_bus --> stack.in_bus display "m=s";
            in_user --> appl.in_user;
            // internal module gates
            stack.out_appl --> appl.in_appl display "m=n";
            stack.in_appl <-- appl.out_appl display "m=n";
      display: "b=90,170;i=block/process,black";
endmodule

//
// Starting here the network is designed
//
module KNX_line_1_1
      parameters:
            numPorts: numeric const;
      gates:
            // connectors to the Line Connector (LC)
            in: line_in[];
            out: line_out[];
      submodules:
            bus_1_1: KNXbus;
                  parameters:
                        txRate = "";
                  display: "p=335,140;b=600,20";
            dev_1_1_1: KNXdev;
                  parameters:
                        address = "1.1.1",
       table = "1/1/2=lightA;0/0/0=lightALL;1/1/100=acurange",
                        application = "switch1";
                  display: "p=60,80;b=40,24";
            dev_1_1_2: KNXdev;
                  parameters:
                        address = "1.1.2",
```

```
 table = "1/1/2=light;1/1/7=light;1/1/100=acurange",
          application = "light1";
     display: "p=120,80;b=40,24";
dev_1_1_3: KNXdev;
     parameters:
          address = "1.1.3",
          table = "1/1/7=light;1/1/100=acurange",
          application = "light2";
     display: "p=180,80;b=40,24";
dev_1_1_4: KNXdev;
     parameters:
          address = "1.1.4",
          table = "1/1/7=lightB;1/1/100=acurange",
          application = "switch2";
     display: "p=240,80;b=40,24";
dev_1_1_5: KNXdev;
     parameters:
          address = "1.1.5",
          table = "1/1/7=lightB;1/1/100=acurange",
          application = "switch3";
     display: "p=300,80;b=40,24";
dev_1_1_6: KNXdev;
     parameters:
          address = "1.1.6",
          table = "1/1/5=lightC;1/1/100=acurange",
          application = "switch4";
     display: "p=360,80;b=40,24";
dev_1_1_7: KNXdev;
     parameters:
          address = "1.1.7",
          table = "1/1/3=lightD;1/1/100=acurange",
          application = "switch5";
     display: "p=420,80;b=40,24";
dev_1_1_8: KNXdev;
```

```
            parameters:
                address = "1.1.8",
                table = "1/1/4=lightXE;1/1/100=acurange",
                application = "switch6";
            display: "p=480,80;b=40,24";
        dev_1_1_9: KNXdev;
            parameters:
                address = "1.1.9",
                table = "1/1/2=light;1/1/100=acurange",
                application = "light3";
            display: "p=540,80;b=40,24";
        dev_1_1_10: KNXdev;
            parameters:
                address = "1.1.10",
                table = "1/1/7=light;1/1/100=acurange",
                application = "light4";
            display: "p=600,80;b=40,24";
        dev_1_1_100: KNXdev;
            parameters:
                address = "1.1.100",
                table = "1/1/100=keyrange",
                application = "acu_1_1";
            display: "p=660,80;b=40,24";
        user: UserSim;
            display: "p=330,30;b=100,24";
    connections:
        dev_1_1_1.out_bus --> bus_1_1.in++;
        dev_1_1_1.in_bus <-- bus_1_1.out++;
        dev_1_1_2.out_bus --> bus_1_1.in++;
        dev_1_1_2.in_bus <-- bus_1_1.out++;
        dev_1_1_3.out_bus --> bus_1_1.in++;
        dev_1_1_3.in_bus <-- bus_1_1.out++;
        dev_1_1_4.out_bus --> bus_1_1.in++;
        dev_1_1_4.in_bus <-- bus_1_1.out++;
```

```
            dev_1_1_5.out_bus --> bus_1_1.in++;
            dev_1_1_5.in_bus <-- bus_1_1.out++;
            dev_1_1_6.out_bus --> bus_1_1.in++;
            dev_1_1_6.in_bus <-- bus_1_1.out++;
            dev_1_1_7.out_bus --> bus_1_1.in++;
            dev_1_1_7.in_bus <-- bus_1_1.out++;
            dev_1_1_8.out_bus --> bus_1_1.in++;
            dev_1_1_8.in_bus <-- bus_1_1.out++;
            dev_1_1_9.out_bus --> bus_1_1.in++;
            dev_1_1_9.in_bus <-- bus_1_1.out++;
            dev_1_1_10.out_bus --> bus_1_1.in++;
            dev_1_1_10.in_bus <-- bus_1_1.out++;
            dev_1_1_100.out_bus --> bus_1_1.in++;
            dev_1_1_100.in_bus <-- bus_1_1.out++;
            // Line Coupler connection lc_out = to line coupler,
            // lc_in = from line coupler
            line_out++ <-- bus_1_1.out++;
            line_in++ --> bus_1_1.in++;
            //User simulation layer
            dev_1_1_1.in_user <-- user.out++;
            dev_1_1_2.in_user <-- user.out++;
            dev_1_1_3.in_user <-- user.out++;
            dev_1_1_4.in_user <-- user.out++;
            dev_1_1_5.in_user <-- user.out++;
            dev_1_1_6.in_user <-- user.out++;
            dev_1_1_7.in_user <-- user.out++;
            dev_1_1_8.in_user <-- user.out++;
            dev_1_1_9.in_user <-- user.out++;
            dev_1_1_10.in_user <-- user.out++;
            dev_1_1_100.in_user <-- user.out++;
      display: "b=650,160;m=w";
endmodule
```

```
module KNX_line_1_2
      parameters:
            numPorts: numeric const;
      gates:
            // connectors to the Line Connector (LC)
            in: line_in[];
            out: line_out[];
      submodules:
            bus_1_2: KNXbus;
                  parameters:
                        txRate = "";
                  display: "p=335,140;b=600,20";
            dev_1_2_1: KNXdev;
                  parameters:
                        address = "1.2.1",
        table = "1/1/2=lightA;0/0/0=lightALL;1/2/100=acurange",
                        application = "switch1";
                  display: "p=60,80;b=40,24";
            dev_1_2_2: KNXdev;
                  parameters:
                        address = "1.2.2",
             table = "1/1/2=light;1/1/7=light;1/2/100=acurange",
                        application = "light1";
                  display: "p=120,80;b=40,24";
            dev_1_2_3: KNXdev;
                  parameters:
                        address = "1.2.3",
                        table = "1/1/7=light;1/2/100=acurange",
                        application = "light2";
                  display: "p=180,80;b=40,24";
            dev_1_2_4: KNXdev;
                  parameters:
                        address = "1.2.4",
                        table = "1/1/7=lightB;1/2/100=acurange",
```

```
            application = "switch2";
      display: "p=240,80;b=40,24";
dev_1_2_5: KNXdev;
      parameters:
            address = "1.2.5",
            table = "1/1/7=lightB;1/2/100=acurange",
            application = "switch3";
      display: "p=300,80;b=40,24";
dev_1_2_6: KNXdev;
      parameters:
            address = "1.2.6",
            table = "1/1/5=lightC;1/2/100=acurange",
            application = "switch4";
      display: "p=360,80;b=40,24";
dev_1_2_7: KNXdev;
      parameters:
            address = "1.2.7",
            table = "1/1/3=lightD;1/2/100=acurange",
            application = "switch5";
      display: "p=420,80;b=40,24";
dev_1_2_8: KNXdev;
      parameters:
            address = "1.2.8",
            table = "1/1/4=lightXE;1/2/100=acurange",
            application = "switch6";
      display: "p=480,80;b=40,24";
dev_1_2_9: KNXdev;
      parameters:
            address = "1.2.9",
            table = "1/1/2=light;1/2/100=acurange",
            application = "light3";
      display: "p=540,80;b=40,24";
dev_1_2_10: KNXdev;
      parameters:
```

```
                    address = "1.2.10",
                    table = "1/1/7=light",
                    application = "light4";
               display: "p=600,80;b=40,24";
          dev_1_2_100: KNXdev;
               parameters:
                    address = "1.2.100",
                    table = "1/2/100=keyrange",
                    application = "acu_1_2";
               display: "p=660,80;b=40,24";
          user: UserSim;
               display: "p=330,30;b=100,24";
     connections:
          dev_1_2_1.out_bus --> bus_1_2.in++;
          dev_1_2_1.in_bus <-- bus_1_2.out++;
          dev_1_2_2.out_bus --> bus_1_2.in++;
          dev_1_2_2.in_bus <-- bus_1_2.out++;
          dev_1_2_3.out_bus --> bus_1_2.in++;
          dev_1_2_3.in_bus <-- bus_1_2.out++;
          dev_1_2_4.out_bus --> bus_1_2.in++;
          dev_1_2_4.in_bus <-- bus_1_2.out++;
          dev_1_2_5.out_bus --> bus_1_2.in++;
          dev_1_2_5.in_bus <-- bus_1_2.out++;
          dev_1_2_6.out_bus --> bus_1_2.in++;
          dev_1_2_6.in_bus <-- bus_1_2.out++;
          dev_1_2_7.out_bus --> bus_1_2.in++;
          dev_1_2_7.in_bus <-- bus_1_2.out++;
          dev_1_2_8.out_bus --> bus_1_2.in++;
          dev_1_2_8.in_bus <-- bus_1_2.out++;
          dev_1_2_9.out_bus --> bus_1_2.in++;
          dev_1_2_9.in_bus <-- bus_1_2.out++;
          dev_1_2_10.out_bus --> bus_1_2.in++;
          dev_1_2_10.in_bus <-- bus_1_2.out++;
          dev_1_2_100.out_bus --> bus_1_2.in++;
```

```
            dev_1_2_100.in_bus <-- bus_1_2.out++;
            // Line Coupler connection lc_out = to line coupler,
            // lc_in = from line coupler
            line_out++ <-- bus_1_2.out++;
            line_in++ --> bus_1_2.in++;
            //User simulation layer
            dev_1_2_1.in_user <-- user.out++;
            dev_1_2_2.in_user <-- user.out++;
            dev_1_2_3.in_user <-- user.out++;
            dev_1_2_4.in_user <-- user.out++;
            dev_1_2_5.in_user <-- user.out++;
            dev_1_2_6.in_user <-- user.out++;
            dev_1_2_7.in_user <-- user.out++;
            dev_1_2_8.in_user <-- user.out++;
            dev_1_2_9.in_user <-- user.out++;
            dev_1_2_10.in_user <-- user.out++;
            dev_1_2_100.in_user <-- user.out++;
      display: "b=650,160;m=w";
endmodule
```

```
module KNX_line_1_0
      parameters:
            numPorts: numeric const; // = 4, [0..3]
      gates:
            // connectors to the Line Couplers (LC)
            in: line_in[];
            out: line_out[];
      submodules:
            bus_1_0: KNXbus;
                  parameters:
                        txRate = "";
                  display: "p=245,120;b=312,20";
```

```
            dev_1_0_1: KNXdev;
                  parameters:
                        address = "1.0.1",
                        table = "0/0/0=keyALL",
                        application = "keyserver1";
                  display: "p=112,65;b=40,24";
            user: UserSim;
      connections:
            dev_1_0_1.out_bus --> bus_1_0.in++;
            dev_1_0_1.in_bus <-- bus_1_0.out++;
            // Line Coupler connection lc_out = to line coupler,
            // lc_in = from line coupler
            for i=0..numPorts-1 do
                  line_out[i] <-- bus_1_0.out++;
                  line_in[i] --> bus_1_0.in++;
            endfor;
            //User simulation layer
            dev_1_0_1.in_user <-- user.out++;
      display: "b=455,140;m=w";
endmodule


module KNX_line_2_1
      parameters:
            numPorts: numeric const;
      gates:
            // connectors to the Line Connector (LC)
            in: line_in[];
            out: line_out[];
      submodules:
            bus_2_1: KNXbus;
                  parameters:
                        txRate = "";
                  display: "p=335,140;b=600,20";
```

```
        dev_2_1_1: KNXdev;
              parameters:
                    address = "2.1.1",
        table = "1/1/2=lightA;0/0/0=lightALL;2/1/100=acurange",
                    application = "switch1";
              display: "p=60,80;b=40,24";
        dev_2_1_2: KNXdev;
              parameters:
                    address = "2.1.2",
         table = "1/1/2=light;1/1/7=light;2/1/100=acurange",
                    application = "light1";
              display: "p=120,80;b=40,24";
        dev_2_1_3: KNXdev;
              parameters:
                    address = "2.1.3",
                    table = "1/1/7=light;2/1/100=acurange",
                    application = "light2";
              display: "p=180,80;b=40,24";
        dev_2_1_4: KNXdev;
              parameters:
                    address = "2.1.4",
                    table = "1/1/7=lightB;2/1/100=acurange",
                    application = "switch2";
              display: "p=240,80;b=40,24";
        dev_2_1_5: KNXdev;
              parameters:
                    address = "2.1.5",
                    table = "1/1/7=lightB;2/1/100=acurange",
                    application = "switch3";
              display: "p=300,80;b=40,24";
        dev_2_1_6: KNXdev;
              parameters:
                    address = "2.1.6",
                    table = "1/1/5=lightC;2/1/100=acurange",
```

```
            application = "switch4";
      display: "p=360,80;b=40,24";
dev_2_1_7: KNXdev;
      parameters:
            address = "2.1.7",
            table = "1/1/3=lightXD;2/1/100=acurange",
            application = "switch5";
      display: "p=420,80;b=40,24";
dev_2_1_8: KNXdev;
      parameters:
            address = "2.1.8",
            table = "1/1/4=lightE;2/1/100=acurange",
            application = "switch6";
      display: "p=480,80;b=40,24";
dev_2_1_9: KNXdev;
      parameters:
            address = "2.1.9",
            table = "1/1/2=light;2/1/100=acurange",
            application = "light3";
      display: "p=540,80;b=40,24";
dev_2_1_10: KNXdev;
      parameters:
            address = "2.1.10",
            table = "1/1/7=light;2/1/100=acurange",
            application = "light4";
      display: "p=600,80;b=40,24";
dev_2_1_100: KNXdev;
      parameters:
            address = "2.1.100",
            table = "2/1/100=keyrange",
            application = "acu_2_1";
      display: "p=660,80;b=40,24";
user: UserSim;
      display: "p=330,30;b=100,24";
```

```
connections:
        dev_2_1_1.out_bus --> bus_2_1.in++;
        dev_2_1_1.in_bus <-- bus_2_1.out++;
        dev_2_1_2.out_bus --> bus_2_1.in++;
        dev_2_1_2.in_bus <-- bus_2_1.out++;
        dev_2_1_3.out_bus --> bus_2_1.in++;
        dev_2_1_3.in_bus <-- bus_2_1.out++;
        dev_2_1_4.out_bus --> bus_2_1.in++;
        dev_2_1_4.in_bus <-- bus_2_1.out++;
        dev_2_1_5.out_bus --> bus_2_1.in++;
        dev_2_1_5.in_bus <-- bus_2_1.out++;
        dev_2_1_6.out_bus --> bus_2_1.in++;
        dev_2_1_6.in_bus <-- bus_2_1.out++;
        dev_2_1_7.out_bus --> bus_2_1.in++;
        dev_2_1_7.in_bus <-- bus_2_1.out++;
        dev_2_1_8.out_bus --> bus_2_1.in++;
        dev_2_1_8.in_bus <-- bus_2_1.out++;
        dev_2_1_9.out_bus --> bus_2_1.in++;
        dev_2_1_9.in_bus <-- bus_2_1.out++;
        dev_2_1_10.out_bus --> bus_2_1.in++;
        dev_2_1_10.in_bus <-- bus_2_1.out++;
        dev_2_1_100.out_bus --> bus_2_1.in++;
        dev_2_1_100.in_bus <-- bus_2_1.out++;
        // Line Coupler connection lc_out = to line coupler,
        // lc_in = from line coupler
        line_out++ <-- bus_2_1.out++;
        line_in++ --> bus_2_1.in++;
        //User simulation layer
        dev_2_1_1.in_user <-- user.out++;
        dev_2_1_2.in_user <-- user.out++;
        dev_2_1_3.in_user <-- user.out++;
        dev_2_1_4.in_user <-- user.out++;
        dev_2_1_5.in_user <-- user.out++;
        dev_2_1_6.in_user <-- user.out++;
```

```
            dev_2_1_7.in_user <-- user.out++;
            dev_2_1_8.in_user <-- user.out++;
            dev_2_1_9.in_user <-- user.out++;
            dev_2_1_10.in_user <-- user.out++;
            dev_2_1_100.in_user <-- user.out++;
      display: "b=650,160;m=w";
endmodule
```

```
module KNX_line_0_0  // root level backbone, connecting the
                     // other main lines
      parameters:
            numPorts: numeric const;
      gates:
            // connectors to the Backbone Couplers
            // (BCs equivalent to LCs)
            in: line_in[];
            out: line_out[];
      submodules:
            bus_backbone: KNXbus;
                  parameters:
                        txRate = "";
            dev_0_0_100: KNXdev;
                  parameters:
                        address = "0.0.100",
                        table = "0/0/0=keyALL",
                        application = "acu_0_0";
                  display: "p=112,65;b=40,24";
            user: UserSim;
      connections:
            dev_0_0_100.out_bus --> bus_backbone.in++;
            dev_0_0_100.in_bus <-- bus_backbone.out++;
            // Line Coupler connection lc_out = to line coupler,
            // lc_in = from line coupler
```

```
            for i=0..numPorts-1 do
                  line_out[i] <-- bus_backbone.out++;
                  line_in[i] --> bus_backbone.in++;
            endfor;
            //User simulation layer
            dev_0_0_100.in_user <-- user.out++;
      display: "b=455,140;m=w";
endmodule

//
// Connecting the lines together, set numPorts parameter
//

module KNX_network_1  // main line 1
      parameters:
            numPorts: numeric const;
      gates:
            // connectors to the Backbone Couplers (BC)
            in: line_in[];
            out: line_out[];
      submodules:
            line_1_0: KNX_line_1_0; // bus
                  parameters:
                        numPorts = 5; // LC=4 + BC=1 = 5
                        // connections exposed from bus to extern
            line_1_1: KNX_line_1_1; // "devices"
                  parameters:
                        numPorts = 1;
            line_1_2: KNX_line_1_2;
                  parameters:
                        numPorts = 1;
            line_1_3: KNX_line_1_3;
                  parameters:
                        numPorts = 1;
```

```
        line_1_4: KNX_line_1_4;
             parameters:
                  numPorts = 1;
        lc_1_1_0: KNXlinecoupler;
             parameters:
                  address = "1.1.0", routing = "";
        lc_1_2_0: KNXlinecoupler;
             parameters:
                  address = "1.2.0", routing = "";
        lc_1_3_0: KNXlinecoupler;
             parameters:
                  address = "1.3.0", routing = "";
        lc_1_4_0: KNXlinecoupler;
             parameters:
                  address = "1.4.0", routing = "";
   connections:
        // Connect line couplers to the lines
        line_1_1.line_in++ <-- lc_1_1_0.out_line;
        line_1_1.line_out++ --> lc_1_1_0.in_line;
        line_1_2.line_in++ <-- lc_1_2_0.out_line;
        line_1_2.line_out++ --> lc_1_2_0.in_line;
        line_1_3.line_in++ <-- lc_1_3_0.out_line;
        line_1_3.line_out++ --> lc_1_3_0.in_line;
        line_1_4.line_in++ <-- lc_1_4_0.out_line;
        line_1_4.line_out++ --> lc_1_4_0.in_line;
        // Connect line couplers to backbone
        lc_1_1_0.in_backbone <-- line_1_0.line_out++;
        lc_1_1_0.out_backbone --> line_1_0.line_in++;
        lc_1_2_0.in_backbone <-- line_1_0.line_out++;
        lc_1_2_0.out_backbone --> line_1_0.line_in++;
        lc_1_3_0.in_backbone <-- line_1_0.line_out++;
        lc_1_3_0.out_backbone --> line_1_0.line_in++;
        lc_1_4_0.in_backbone <-- line_1_0.line_out++;
        lc_1_4_0.out_backbone --> line_1_0.line_in++;
```

```
            for i=0..numPorts-1 do
                  line_out[i] <-- line_1_0.line_out++;
                  line_in[i] --> line_1_0.line_in++;
            endfor;
endmodule


module KNX_network_2  // main line 2
      parameters:
            numPorts: numeric const;
      gates:
            // connectors to the Backbone Couplers (BC)
            in: line_in[];
            out: line_out[];
      submodules:
            line_2_0: KNX_line_2_0; // bus
                  parameters:
                        numPorts = 5; // LC=4 + BC=1 = 2
                        // connections exposed from bus to extern
            line_2_1: KNX_line_2_1; // "devices"
                  parameters:
                        numPorts = 1;
            line_2_2: KNX_line_2_2;
                  parameters:
                        numPorts = 1;
            line_2_3: KNX_line_2_3;
                  parameters:
                        numPorts = 1;
            line_2_4: KNX_line_2_4;
                  parameters:
                        numPorts = 1;
            lc_2_1_0: KNXlinecoupler;
                  parameters:
                        address = "2.1.0", routing = "";
```

```
            lc_2_2_0: KNXlinecoupler;
                  parameters:
                        address = "2.2.0", routing = "";
            lc_2_3_0: KNXlinecoupler;
                  parameters:
                        address = "2.3.0", routing = "";
            lc_2_4_0: KNXlinecoupler;
                  parameters:
                        address = "2.4.0", routing = "";
      connections:
            // Connect line couplers to the lines
            line_2_1.line_in++ <-- lc_2_1_0.out_line;
            line_2_1.line_out++ --> lc_2_1_0.in_line;
            line_2_2.line_in++ <-- lc_2_2_0.out_line;
            line_2_2.line_out++ --> lc_2_2_0.in_line;
            line_2_3.line_in++ <-- lc_2_3_0.out_line;
            line_2_3.line_out++ --> lc_2_3_0.in_line;
            line_2_4.line_in++ <-- lc_2_4_0.out_line;
            line_2_4.line_out++ --> lc_2_4_0.in_line;
            // Connect line couplers to backbone
            lc_2_1_0.in_backbone <-- line_2_0.line_out++;
            lc_2_1_0.out_backbone --> line_2_0.line_in++;
            lc_2_2_0.in_backbone <-- line_2_0.line_out++;
            lc_2_2_0.out_backbone --> line_2_0.line_in++;
            lc_2_3_0.in_backbone <-- line_2_0.line_out++;
            lc_2_3_0.out_backbone --> line_2_0.line_in++;
            lc_2_4_0.in_backbone <-- line_2_0.line_out++;
            lc_2_4_0.out_backbone --> line_2_0.line_in++;
            for i=0..numPorts-1 do
                  line_out[i] <-- line_2_0.line_out++;
                  line_in[i] --> line_2_0.line_in++;
            endfor;
endmodule
```

```
module KNX_network_0
      // connect all KNX main lines together on root backbone
      parameters:
            numPorts: numeric const; // = 0;
            // change when main lines have been added
      gates:
            // connectors to the Backbone Couplers (BC)
            in: line_in[];
            out: line_out[];
      submodules:
            line_0_0: KNX_line_0_0;
                  parameters:
                        numPorts = 2; // BC=2
            network_1: KNX_network_1;
                  parameters:
                        numPorts = 1;
            network_2: KNX_network_2;
                  parameters:
                        numPorts = 1;
            lc_1_0_0: KNXlinecoupler;
                  parameters:
                        address = "1.0.0", routing = "";
            lc_2_0_0: KNXlinecoupler;
                  parameters:
                        address = "2.0.0", routing = "";
      connections:
            // connect line couplers to the lines
            network_1.line_in++ <-- lc_1_0_0.out_line;
            network_1.line_out++ --> lc_1_0_0.in_line;
            network_2.line_in++ <-- lc_2_0_0.out_line;
            network_2.line_out++ --> lc_2_0_0.in_line;
            // Connect line couplers to backbone
            lc_1_0_0.in_backbone <-- line_0_0.line_out++;
```

```
            lc_1_0_0.out_backbone --> line_0_0.line_in++;
            lc_2_0_0.in_backbone <-- line_0_0.line_out++;
            lc_2_0_0.out_backbone --> line_0_0.line_in++;
            for i=0..numPorts-1 do
                  line_out[i] <-- line_0_0.line_out++;
                  line_in[i] --> line_0_0.line_in++;
            endfor;
endmodule


network knx_network : KNX_network_0
      parameters:
            numPorts = 0;
endnetwork
```

A.2 UserSim Files

A.2.1 Insecure Communication

```
0/3.0/300, switch1, lightA:02ffffffffffff
0/3.0/300, switch1, lightALL:02ffffffffffff

0/1.0/900, switch2, lightB:01ffffffffffff
0/~1.0/900, switch2, lightB:02ffffffffffff

0/1.5/600, switch4, lightC:02ffffffffffff

0/~1.5/600, switch5, lightD:02ffffffffffff
```

A.2.2 Secure Communication with Line ACUs

```
0/3.0/300, switch1, lightA:02ffffffffffffffffffffffffff
0/3.0/300, switch1, lightALL:02ffffffffffffffffffffffffff
```

```
0/1.0/900, switch2, lightB:01ffffffffffffffffffffffffff
0/~1.0/900, switch2, lightB:02ffffffffffffffffffffffffff

0/1.5/600, switch4, lightC:02ffffffffffffffffffffffffff

0/~1.5/600, switch5, lightD:02ffffffffffffffffffffffffff

// ACU activities

// Initiate key distribution to devices
0/240/4, acu_1_1, acurange:10010101
0/240/4, acu_1_2, acurange:10010101
0/240/4, acu_1_3, acurange:10010101
0/240/4, acu_1_4, acurange:10010101
0/240/4, acu_2_1, acurange:10010101
0/240/4, acu_2_2, acurange:10010101
0/240/4, acu_2_3, acurange:10010101
0/240/4, acu_2_4, acurange:10010101
```

A.2.3 Secure Communication with Central ACU

```
0/3.0/300, switch1, lightA:02ffffffffffffffffffffffffff
0/3.0/300, switch1, lightALL:02ffffffffffffffffffffffffff

0/1.0/900, switch2, lightB:01ffffffffffffffffffffffffff
0/~1.0/900, switch2, lightB:02ffffffffffffffffffffffffff

0/1.5/600, switch4, lightC:02ffffffffffffffffffffffffff

0/~1.5/600, switch5, lightD:02ffffffffffffffffffffffffff

// ACU activities
```

```
// Initiate key distribution to devices
0/240/4, acu_0_0, keyALL:10010101
```