FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reliable Control Network Gateways

A Case Study for KNX and ZigBee

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Stefan Seifried
Matrikelnummer 0925401

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr. Lukas Krammer

Wien, TT.MM.JJJJ

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reliable Control Network Gateways

A Case Study for KNX and ZigBee

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Stefan Seifried
Registration Number 0925401

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Univ.Ass. Dipl.-Ing. Dr. Lukas Krammer

Vienna, TT.MM.JJJJ

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Stefan Seifried
Wiedner Gürtel 50/12, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all, I want to thank my advisor, ao. Univ. Prof. Dipl.-Ing. Dr. Wolfgang Kastner and Univ. Ass. Dipl.-Ing. Dr. Lukas Krammer for their ongoing support, encouragement and feedback throughout the whole progress of this thesis.

Further, I want to thank my parents, which were always there for me and provided me with their loving care and support through all those years.

Also, my dear friends throughout my study: Matthias Burgholzer, Djordje Slijepcevic, Christian Steinkress, Anelia Dincheva and Jürgen Pannosch. Thanks for your support and the funny time!

Additionally, I want to thank my dear colleagues at the Automation System Group: Andreas Fernbach, Thomas Frühwirth, Stefan Gaida, Daniel Schachinger, Jürgen Schober and Peter Hausberger. They made working there, an experience of a life time and provided me with additional feedback on my thesis.

Abstract

From the so-called fieldbus wars on, automations systems suffered from an increased heterogeneity [26]. Since then solutions for the interconnection and integration of different communication technologies throughout the automation pyramid (see Figure 1.1) are still high in demand. However, past efforts tended to introduce additional abstraction layers above the field level, and direct interconnection between field protocols was seldom considered. Despite the fact that more and more critical infrastructure is incorporated into automation systems, only little effort is put into proper reliability mechanisms throughout existing integration solutions.

Therefore, this thesis introduces a *gateway* solution targeting the problem of horizontal integration at the field level of the automation pyramid in a reliable way. A general applicable translation process between field networks has been advised. Field devices residing in one field network are mapped into another fieldbus by the means of a proposed *information model*. The *information model* has been inspired by modeling capabilities of well-established integration solutions like object linking and embedding (OLE) for process control (OPC) - unified architecture (OPC UA) [16] and open building information exchange (oBIX) [14].

The reliability concept has been based on the replication of multiple *gateway* devices to form a redundant compound. Therefore, synchronization and end-to-end communication monitoring mechanisms are proposed. Synchronization tasks are performed via existing fieldbus connections, which obviates the need for a separate backbone link between individual *gateway* devices. Furthermore, the fault hypothesis of the redundant compound and the overall interconnected automation systems are stated and analyzed.

Besides an analytic discussion of all fault scenarios and fault recovery processes, a proof-of-concept testbed for the technologies KNX and ZigBee has been developed to verify the results of the fault analysis.

In conclusion, the described *gateway* approach has been proven to be a feasible and viable solution to the integration problem of different field level protocols. However, there are still tasks for future work, that have not been addressed by this thesis. This includes several management operations for the redundant *gateway* compound and more sophisticated mechanisms for fault handling.

Kurzfassung

Durch die historisch gewachsene Vielfalt an Feldbuslösungen nutzen Automatisierungsysteme oftmals ein heterogenes Konglomerat unterschiedlichster Kommunikationsprotokolle [26]. Lösungen für die Vernetzung und Integration der verschiedenen Feldbusprotokolle sind noch immer sehr gefragt. Bisherige Anstrengungen in diese Richtung resultierten daher in Ansätzen die zusätzliche Abstraktionsebenen einführten. Auch die fortschreitende Automatisierung kritischer Systeme wurde nicht immer ausreichend mit Mechanismen zur Sicherung der Zuverlässigkeit berücksichtigt.

Die vorliegende Diplomarbeit führt eine *Gateway* Lösung ein, die sowohl auf das Problem der horizontalen Integration als auch auf die zuverlässige Anbindung kritischer Systeme abzielt. Dafür wurde ein allgemeines Informationsmodell entwickelt. Das Informationsmodell wurde inspiriert von etablierten Lösungen, wie OPC UA [16] und oBIX [14].

Die Zuverlässigkeit der eingeführten *Gateway* Lösung wird durch Replikation mehrerer *Gatewaygeräte* erreicht, die in einem redundanten Verbund zusammengeschlossen sind. Dabei müssen die einzelnen *Gatewaygeräte* untereinander synchronisiert werden, was über die bereits vorhandenen Feldbusverbindungen geschieht. Dies macht eine separate *Backbone* Verbindung überflüssig. Weiters werden zu besseren Verteilung der Synchronisationstelegramme Buslastmetriken erhoben und eine Ende-zu-Ende Verbindungssicherung zwischen den einzelnen *Gatewaygeräten* durchgeführt. Außerdem wurde eine Fehlerhypothese für den *Gatewayverbund* und für das gesamte, verbundene Automatisierungssystem aufgestellt und analysiert.

Eine detaillierte Fehleranalyse wurde mittels vollständiger Enumeration aller möglichen Fehlerfälle durchgeführt. Weiters wurden die Fehleranalyseergebnisse in einer Testumgebung mit einer prototypischen Gatewayimplementierung für die Technologien KNX und ZigBee auf ihre Richtigkeit hin überprüft.

Abschließend betrachtet ist der beschriebene *Gateway* Ansatz nicht nur eine mögliche, sondern auch eine sinnvolle Lösung für das Integrationsproblem verschiedener Feldbusse. Für zukünftige Arbeiten sind unter anderem Managementprozeduren für die einzelnen Teilnehmer eines Gatewayverbunds und differenzierte Methoden zur Fehlerbehandlung.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Automation systems use a large variety of communication mechanisms due to mostly historical reasons. At the advent of automation systems, well established standards from the world of office networking were not readily available or not appropriate for the task. Furthermore, automation was strongly driven by particular application domains with very own requirements in types of communication services, information modeling and processing. The emergence of the automation pyramid [32] and further on the wide-spread success of Ethernet/IP, led to a consolidation of communication protocols. Nowadays, the upper layers of the automation pyramid are mostly based on IP networks and use information modeling and process techniques present in modern information and communication technology (ICT) systems.

However, the field level remained a conglomerate of different protocols, despite standardization efforts and the introduction of Ethernet/IP. The diversity is driven by maintaining existing, well-established solutions and the exploitation of new technological advances that enable new approaches to application domains. Hence a lot of field protocols serving effectively the same application domains, share mostly semantically equivalent information models and processing mechanisms. Significant differences often arise at the physical transport medium, where individual protocols target a particular set of requirements. (e.g. long distance, reliability)

This gives rise to the integration problem at the vertical and the horizontal dimension of the automation pyramid. While the challenge of the vertical integration lies in information aggregation and filtering, the challenge of horizontal integration is finding a common information model between protocols. Furthermore in the scenario of automating critical infrastructure, reliability plays an important role. Different communication protocols along the pyramid implement their very own reliability mechanisms and single points of integration be it at the vertical or horizontal dimension would compromise a holistic reliability concept for the automation pyramid model. Therefore, a well defined integration concept must also consider multiple coupling devices between or in layers.

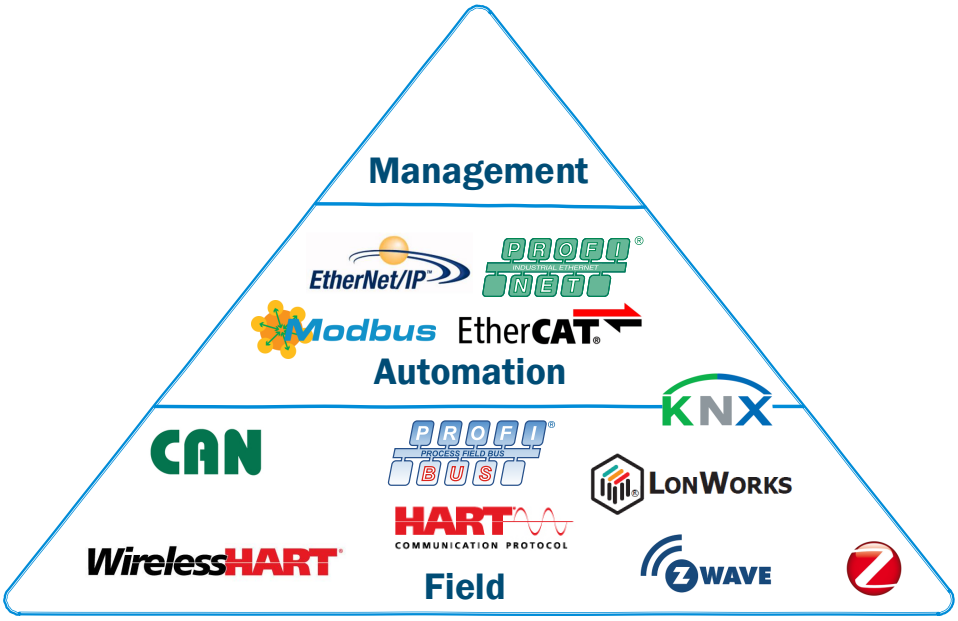

Figure 1.1: Automation pyramid for BACS according to *ISO 16484-2:2004* [11] illustrating the heterogeneity of the field layer

The continuing success of Ethernet/IP at the management and automation layer of the automation pyramid already solved the integration problem by unifying message transmission. Different information structures can be easily transformed by well-established means, like Webservices. All that remains is the integration problem at the horizontal dimension of the field level, which lacks ready to use approaches. [46]

1.2 Reliability

Reliability is an important factor in the design of dependable communication systems and describes the probability that a system will continue a correct service, given that it was working properly at the very beginning [36]. A correct service is delivered when a system works according to its specification, whereas an incorrect service delivers no results at all or even worse bogus information. The transition between a correct service and an incorrect service is characterized by three terms: *fault*, *error* and *failure*. A *fault* is the hypothetical origin of an error. It is called *inactive*, when the system has not triggered the *fault* and *active* when the system has produced an *error*. An *error* is then a part of the system that is affected by the fault and experiences a malfunction. Then, a *failure* occurs when an *error* is able to affect the delivered service [22].

There are several concepts to sustain reliability, either during the development or operation of a system. The first method is *fault prevention*, which subsumes quality control mechanisms during the development and manufacturing phase of a system. *Fault removal* aims to correct

faults during the development cycle and operation of a system. The process of *fault removal* is partitioned in three steps: *verification*, *diagnosis* and *correction*. The *verification* step describes the checking of properties and conditions according to the specification. In case the such a check fails, the root cause is determined by *diagnosis*. Finally the *fault* is fixed by a proper *correction*.

But the most important method regarding this work is *fault tolerance*. *Fault tolerance* maintains the correct service of a system even in if *faults* are active. This is achieved by first detecting such an *error* and then subsequently apply proper countermeasures to avoid a *failure*. *Error detection* can take place either during service delivery or while a system is in idle mode. Upon a detected *error*, an affected system has two options: Either it resides on *recovery* or *fault handling*. *Recovery* involves mechanisms that reverse to a known good previous state, recover due to the presence of redundancy information, or even switch to an entirely new state which is *error* free. *Fault handling* on the other hand tries to exclude the *faulted* component from the operation of the system, called *fault isolation*. Or even further, a rearrangement of tasks or swap-in of spare components is performed by *system reconfiguration*. Finally an active *fault* can also be dealt with by performing a *system reinitialization*. [22]

1.3 Problem statement

The focus of this work lies on the horizontal integration of field layer protocols that share similar information models i.e. targeting the same application domain. Furthermore, any proposed solution must retain the level of availability and reliability for the protocols involved. Additionally aggravating, data representation and communication mechanisms of those protocols are usually not stateless.

Two designated field level protocols from the domain of home and building automation (HBA), KNX [34] and ZigBee [15] are chosen to restrain the problem domain. Both protocols implement completely different communication stacks and even operate on different physical media, but define similar device, data and functional models. KNX [34] specifies its information model in *Volume 7 - Application Descriptions* and ZigBee provides a supplement standard, the *ZigBee Home Automation Profile* [17].

One of the objectives of this work is to strictly remain at the same layer of the automation pyramid. Hence, an information translation mechanisms needs to be advised, which transforms information between the models of the involved protocols. In case of KNX [34] and ZigBee [15], the problem is complicated by the fact that the communication is not stateless. Furthermore both also differ at the mode of communication. *ZigBee* only defines event based communication, whereas KNX [34] both allows polling and event driven communication. Thereby, a viable solution must be capable of storing the respective protocol state and enabling a native view of the whole field layer for each involved protocol.

Another important requirement is to maintain the same quality of service throughout the whole integrated network with respect to availability and reliability. ZigBee defines a mesh topology which is capable of tolerating a silent failure of a single node without affecting the remaining network ([15], [20]). Also KNX is able to cope with the failure of single bus devices and can be enhanced to even tolerate the failure of single network segments [40]. In conclusion, an acceptable solution then must also tolerate single silent failures.

There are in general two feasible approaches to interconnect arbitrary disjoint field networks with each other, a tunneling or a gateway solution. The tunneling approach is only suitable when all elements that shall be interconnected at the field level are utilizing the same field level protocol. Another field level protocol then serves as a mere medium of transportation. However, field level protocols may impose strict timing constraints, which another protocol might not be able to fulfill [41].

Finally, only the gateway approach remains and hence is further evaluated. The proposed gateway translates between the chosen field protocols KNX and ZigBee by storing its own information model. Furthermore, several gateways may be linked together to form a redundant compound gateway. A prototype realizing a lighting scenario is deployed. Then the approach is evaluated by the means of the prototype and a theoretical analysis.

1.4 Aim of the work

The major aim of the thesis is a concept on how to integrate disjoint field protocols with resembling information models in a reliable way. As a additional constraint, the integration shall be established within the limits of the field layer of the automation pyramid. Therefore two representative protocols, namely KNX [34] and ZigBee [15] are chosen. The main criteria supporting this choice are similar information models, but are sufficiently diverse. The chosen protocols are even implemented on two different physical media and require different modes of communication (polling and event driven).

A profound literature study reveals, that the most promising feasible solution for the integration problem at the field level is a gateway approach. It is shown, that the use of such a gateway device enables the use of the whole field network by a single protocol. Effectively hiding the needed complexity of the inter-working between field protocols in a gateway device. Thereby, the upper automation level does not need to be adapted and the deployment side effects of another field protocol is contained within the field layer of the automation protocol. Hence a gateway solution to the horizontal integration problem between KNX and ZigBee is modeled, implemented and analyzed such that it fulfills the previously stated requirements.

At first, an *information model* is proposed in order to manage the state of all network elements, regardless at which fieldbus they reside. Further, a *translation service* will be sketched, that is able to transform the *information model* to the respective representation of the integrated field protocols. Hence, the *information model* is effectively a *super*-model to the models of the chosen field protocols.

Next, a *redundancy service* is introduced that synchronizes the *information models* of disjoint gateway devices using a accompanying *redundancy protocol*. An arbitrary number of gateway devices can then be subsumed into a redundant compound, called virtual redundant gateway (VRG). One of the gateways will then assume the role of a *master* and the other ones will assume the role of *backup* devices. The *redundancy service* will also serve as the protocol stack and watchdog. Therefore, a detailed fault analysis is done, together with an analysis of critical timing paths.

Finally, the complete gateway device is implemented as a proof-of-concept prototype in order to show the feasibility and efficiency of the proposed approach. Furthermore. detailed

performance data is gathered and used to provide a detailed comparison to similar state-of-the-art methods.

1.5 Methodological approach

At first, a literature and Internet research on existing approaches to the integration problem at the field level is performed. Findings are classified either as horizontal solutions, when they remain within the limits of the field layer. Otherwise, they are classified as vertical solutions, in case they introduce additional layers of abstraction to the overall automation system. All found approaches are analyzed and described at the state-of-the-art chapter 2. Further, the gateway approach is sketched and illustrated by a block diagram. Furthermore, each building block is examined and feasible concepts and data models are discussed.

Then the *information model* building block is designed, tailored to the application domain of HBA, using suitable models. Furthermore, a mechanism for *information transformation* is advised, so the stored data can be converted into representations native to the chosen field protocols. Especially solutions that perform the integration on the vertical domain are investigated while modeling *information model* and the *translation service*, as they usually provide abstract data models of the advised problem domain using established concepts.

As a next step, the *redundancy service* and the accompanying *redundancy protocol* are modeled. Different approaches found during the literature research are discussed, compared, and finally a appropriate approach is chosen and implemented. The final outcome will be specified using *UML* and message sequence diagrams. Furthermore the timing, fault tolerance and communication overhead will be theoretically examined.

Finally the theoretical gateway approach is implemented in a prototype gateway device using suitable hardware and testbed. The *information model* and *translation service* will be evaluated in terms feasibility and performance. Especially CPU utilization and memory consumption are of importance. The *redundancy mechanisms* of the prototype are evaluated through careful chosen test cases and compared to existing approaches where applicable.

1.6 Structure of the work

At the beginning a short overview about the motivation, the integration problem and the overall goals of this work are stated. Further, the basic terms of reliability are introduced, emphasizing the importance of this topic (see Chapter 1). In Chapter 2, an overview of existing concepts and solutions tackling the integration problem is given. It is shown that the readily available solutions indeed solve the integration problem, but either lack *fault handling* mechanisms or introduce additional abstraction layers. No existing approach favors a *flat hierarchy* approach, like the *gateway* presented by this work. Chapter 3 then outlines the fundamental requirements of a *reliable gateway*. Then, possible fieldbus topologies are analyzed and an overall architecture of a generic *gateway device* is developed. The components of this architectural model are then described in detail and are evaluated regarding their *reliability* and fitness for the given task. Chapter 5 describes a proof-of-concept implementation using two concrete instances of the targeted fieldbus family, namely *KNX* and *ZigBee*. The implementation is discussed in detail and

the feasibility for real-world applications is shown. Finally, Chapter 6 concludes the thesis with a critical reflection of the presented approach. It further includes an outlook of possible furture enhancements.

CHAPTER 2

State of the art

Existing solutions and approaches throughout the history of the fieldbusses are discussed in Section 2.1. The analysis gives rise to a classification into *integration approaches* and *integration technologies*. The *integration approaches* section describes concepts deployed by most fieldbus protocols to achieve integration with disjoint instances. Whereas the *integration* technologies section presents *vertical integration approaches*, that provide a unified layer of abstraction for communication and data models over several fieldbus protocols. This additional layer is implemented in the bounds of the overlying *automation layer* regarding the *automation pyramid* [11]. A feasible *gateway* solution, as discussed by this work, may be directly compared and measured by the performance and scalability to such *vertical integration solutions*. Further, the unifying information models of such solutions procure proven approaches and practices that are considered designing the data model of the proposed gateway approach.

2.1 Related Work

In the very beginning, field level networks were particularly developed for automation purposes and hence quite different from traditional ICT networks. This led to a vast amount of different protocols each tailored to a specific automation or measurement task [45]. First attempts of integrating the automation communication busses were driven from the desire to access factory data from the office. Thus, in the 1980's the automation pyramid emerged to give a hierarchical model for Computer Integrated Manufacturing (CIM) [50]. Early attempts of a standardized horizontal integration of the individual layers of the automation pyramid are described in [28]. However, for most of the proprietary protocols this was never the intention and integration happened largely on an individual basis if ever.

Starting in the late 1980's, the so called fieldbus war [26] fully emerged and made the integration problem to one of the top issues of the domain. However, it was believed that a unified fieldbus protocol had to be found which could subsequently tackle all possible tasks that may occur in the automation domain. The most notable attempt for a holistic fieldbus approach was

the Interoperable System Project (ISP). Unfortunately, ISP was canceled before reaching a mature state in 1994 [44]. Another attempt was IEC 61158 [6] standard, where only the physical layer was finished and published. The main issue with the holistic fieldbus approaches was, that a full implementation of the standard was too expensive in terms of space and computing power. Further, only partial implementation would have resulted in widely incompatible devices.

In the early 1990's another horizontal integration approach emerged from Supervisory Control and Data Acquisition (SCADA) in conjunction with fieldbus technology and more sophisticated office grade ICT. As a result the OPC standard was published in 1996, which offered a unified interface to different attached fieldbusses [38]. However, direct communication between field devices originating in different field protocols were still unable to communicate. This was still not changed with later iterations like OPC UA (see Section 2.3 – OPC UA) or other incarnations like oBIX (see Section 2.3 – oBIX).

The rise of Ethernet and internet protocol (IP) based office networks enabled the advance of this protocol into the field layer. Recently, the *Industrial Ethernet* attempts revive the hopes for a holistic field protocol. However, also wireless field protocols gained a decent amount of momentum, which do not utilize the Ethernet and IP stack so far [45]. Further most fieldbusses that survived the fieldbus wars provide a Ethernet and IP tunneling solution (see Section 2.2 – Tunneling solutions), which enables to interconnect disjoint field networks. Also solutions that utilize other fieldbusses as such a backbone link have been considered [41]. However, all these approaches only support integration of networks that use the same field protocols.

Gateway approaches (see Section 2.2 – Gateway solutions) to the integration problem are mostly intended as Web interfaces to fieldbusses. On the other hand, a vertical integration solution enabling similar field level protocols to exchange information has rarely been considered [50]. An attempt has been made in ([56], [57]) to bridge specifically KNX and ZigBee. However, the presented approach only focus on pure message translation capabilities and force restrictions on the topology of the networks and their addressing scheme. Further, it lacks a reliability concept and capabilities to store information (see Figure 2.2).

2.2 Interconnection Approaches

Tunneling solutions

Tunneling connections over other protocols is a well-known mechanism in corporate ICT systems. Essentially, *tunneling* refers to the concept of wrapping data packets from one protocol in the packet payload of another protocol without performing any modification to the original data. In [50], two scenarios are listed: *IP Tunneling over Fieldbus Protocols* and *Fieldbus Data Tunneling over Backbone Networks*. Both scenarios are well-established use-cases int the automation system domain and are subject to active research and standardization attempts. However, *tunneling* solutions that extend the reach of one fieldbus by a *tunnel* through another fieldbus are rarely considered. One example of a fieldbus *tunneling* another fieldbus can be found in [41].

There are several obstacles *tunneling* solutions need to tackle. One of the most tempting problems is packet *latency*. Communication partners, residing in disjunctive segments just connected by such a *tunnel* are not aware of the tunnel itself. Hence communication partners that

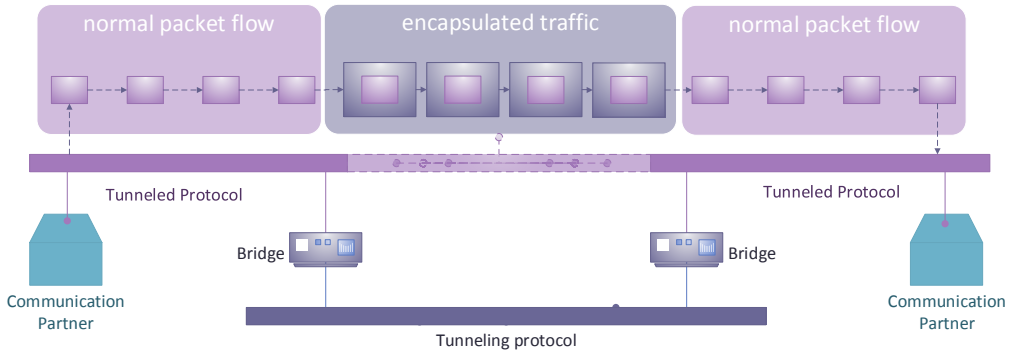

Figure 2.1: Tunneling concept network topology

are connected through a *tunnel* do not apply any special considerations or treatment to their data packets. Furthermore field level protocols require harsh communication timeouts due to their realtime characteristics. Hence, a feasible *tunneling* solution needs to meet the timing specifications of the *tunneled* protocol.

Another problem related to packet *latency* is packet *fragmentation*. First, consider the case where the payload of the *tunneling* protocol is smaller than the data frame size of the *tunneled* protocol. Then the *tunneled* packet needs to be split up on the sending endpoint of the *tunnel* and later joined on the receiving endpoint. This introduces additional computing and transmission overhead for the *tunneling* protocol. But on the other hand, a *tunneling* protocol more capable in payload size may combine several packets from the *tunneled protocol*. Subsequently, this can benefit field protocols that frequently send out significant amounts of small datagrams that may be aggregated in one packet for the *tunneling* protocol.

But the most important conceptual issue with field level *tunneling* solutions is that communication endpoints are only able to reach out to other endpoints within the limits of the same field protocol. Hence devices residing in the *tunneling* protocol segment and devices utilizing the *tunneled* protocol are unable to communicate and exchange information.

Reliability for *tunneling* solutions is rather hard to achieve due to the unawareness of the *tunneled* protocol about the *tunnel* itself. Possible single points of failure are the access points to the *tunnel* and the *tunneling* connection itself. A possible solution can be found at [41], but there is no applicable approach in general.

In conclusion, *tunneling* solutions are only feasible for field protocols with relaxed timing constraints. Furthermore a *tunneling* protocol is only capable of forwarding datagrams and cannot be incorporated in the *tunneled* communication infrastructure.

Gateway solutions

A gateway device is a full member of each connected fieldbus. Its main purpose is the exchange of data between different field level protocols independent of different communication modes or data encoding. But unlike the *tunneling* approach, communication is always targeted from and to the gateway device. There are three main components in a gateway device (see Figure 2.2). First, there are the individual protocol handlers for each supported fieldbus. Second, the core component implementing the gateway logic and providing additional services (e.g. diagnostic services). Finally, the database or information model which stores the bits of information necessary to process and forward data between the involved fieldbusses.

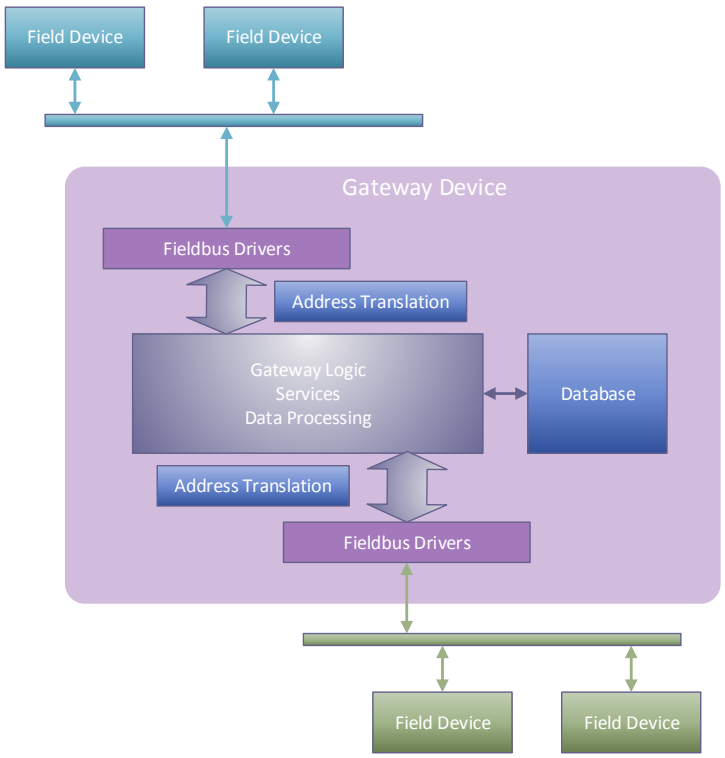

Figure 2.2: Gateway device and network topology [50]

Every information exchange between fieldbus devices, utilizing different field protocols involves the gateway device as a mediator. In general, the gateway needs to provide a *process image* [50] of each fieldbus segment of the automation system. Such a *process image* consists of the data points and functionality a gateway publicly exposes to the connected fieldbusses. Further, a *process image* is a concrete instance of the *information model* which characterizes each separate fieldbus for the gateway device. Two common approaches for the *information model*

are either a organization by logical functionality or by physical topology of the involved field devices.

The *latency* problem inherit to the *tunneling* solution is addressed by the gateway by acting as an intermediary communication partner. However, a gateway only tackles timing constraints on the field layer. End-to-end timing guarantees cannot be given, unless the gateway is able to synchronize (e.g. act as a master) both fieldbusses [50]. On the plus side, a gateway may act as a *buffer* which decouples fieldbusses and can deliver results stored in the *process image*.

Till today, the gateway approach was often considered, but never implemented as a product or even prototype device. The main reason relies in the increased heterogeneity and domain specific tailoring of fieldbusses. Hence, establishing a one-to-one entity mapping between the involved protocols is often not achievable.

Since the gateway approach was seldom implemented for the field-to-field communication scenario, there are no common approaches to provide fault-tolerant modes of operation. One solution used in corporate *ICT* environments is Virtual Router Redundancy Protocol (VRRP), which is standardized in [43]. Originally, VRRP was designed for network routers, but can easily be used for any compound of devices that need fail-over capabilities. Apart from the individual address, there is a common address that is assigned to the current *active* device. Communication from and to the VRRP compound is handled via the common address and hence through the *active* VRRP device. A failure of the *active* VRRP device is detected by frequent monitoring messages between devices. In case of a failure, a *backup* device assumes the role of the *active* device and is subsequently assigned the common address.

2.3 Integration Technologies

OPC UA

The roots of OPC UA date back as far as the year 1995. A new formed industry task force, the *OPC Foundation*, was formed to enable the exchange of real-time data between devices from different vendors. Effectively attacking the arisen integration problem for the numerous available field layer protocols (PROFIBUS, Modbus, ...). The focus was especially on developing a common abstraction model used to interconnect programmable logic controllers (PLCs).

During the mid 1990's, automation systems based on Windows powered PCs were gaining popularity. And so after only one year of development, a standard based on Windows device drivers and Microsofts OLE technology, hence the name OPC, was released. Later, Microsoft's succeeding technologies component object model (COM) and distributed component object model (DCOM) for distributed automation systems have been used. The usage of standard ICT technology at that time enabled a quick adoption and wide-spread acceptance throughout the industry. As of today, every important corporate player in the field of industrial automation is a member of the OPC foundation, that develops and maintains the standard.

The architecture of the OPC standard follows a strict *client/server* model. The *OPC server* hides all the communication details of the underlying fieldbusses and provides a platform independent service model. One or more *OPC clients* may connect to such an *OPC server* and consume those service. Different services are described in the initial standard and several com-

plementary standards that have been released shortly after. Nowadays, those standards are bundled together as the *OPC Classic* standard and consist of:

- *Data Access (OPC/DA)* [8] resembles the original OPC standard and comprises the definition of the client/server architecture. Furthermore the definition of an address space and general activities like reading, writing and data update notifications.
- *Historical Data Access (OPC/HDA)* [10] describes client access to data about past events, usually stored in some form of database.
- *Alarms and Events (OPC/A&E)* [5] describes the monitoring of areas and dispatching of notifications upon events, as opposed to the continuous communication used by the data access standard.
- *XML-Data Access (OPC/XML-DA)* [12] supplements the original data access standard by a Extensible Markup Language (XML) based format usable for Web services and simple object access protocol (SOAP) applications.
- *Data eXchange (OPC/DX)* [9] addresses the communication between different OPC servers. Moreover management, monitoring and diagnostic services.
- *Complex Data* [7] supplements the distribution of whole XML documents and arbitrary binary data.
- *Security* [3] offers an optional layer of security, tightly integrated with Windows NT domain services. Furthermore adding access control to applications and data.
- *Batch* [4] is an extension that addresses the execution of tasks without human interaction.

OPC Classic set of standards was and is still very successful, but was strongly linked to the Microsoft Windows eco-system. Furthermore, additional use cases emerged from traditional SCADA and Human-Machine Interface (HMI) tasks to Enterprise Resource Planning (ERP) and Manufacturing Execution Systems (MESs) tasks. Action was taken and in 2006, after more than three years of development, OPC UA was released as a successor. However there were several more reasons, why a simple update of the standard was no longer adequate [38]:

1. *Discontinuation of COM/DCOM* as of 2002, Microsoft declared COM/DCOM deprecated technologies in favor of their new *.NET* technology. Although future versions of *Windows* are not dropping support for COM/DCOM, it still means that COM/DCOM is not a actively developed technology. Quite recent, COM/DCOM was again resurrected as part of the new *Windows RT* in *Windows 8*.

2. *DCOM limitations*, especially cumbersome security settings and difficult access privilege administration. Moreover, communication timeouts were not adjustable, resulting in a delayed detection of communication problems. Tunneling mechanisms were deployed as a workaround and avoided the use of DCOM as a whole.

3. *OPC communication across firewalls* was limited by the underlying DCOM technology. In IP communication, a port is usually assigned to a fixed service, e.g. port 80 is assigned to Hypertext Transfer Protocol (HTTP). DCOM uses multiple ports, which vary

and change from instance to instance. Therefore, a wide range of ports needs to be opened at the firewall, where each open port means a potential security gap for the automation system.

4. *Use of OPC on non-Windows platforms* like *Linux* powered PCs or embedded devices became increasingly more important. Especially high performance system on chip (SOC) powered embedded devices gave rise to the demand of OPC running on platforms like *QNX*, *VxWorks* or *embedded Linux*.

5. *High-performance OPC communication via Web services* was also seeing more and more demand. The *OPC Foundation* addressed the problem by issuing the *OPC/XML-DA* standard. But operation was too slow, since it was constructed as a wrapper around traditional *OPC/DA* resulting in a performance degradation of five to seven times compared to *OPC/DA*.

6. *Unified data model* in contrast to the separated approach of *OPC Classic*, where three different *OPC Servers* were needed for *data access*, *alarms & events* and *historical data access*.

7. *Support of complex data structures* for configuration services was needed. While simple data types, like *byte*, *array* or *real* are sufficient for state and process data, more structured data is needed to configure a device through a *OPC Server*.

8. *Process data communication without data loss* is a key parameter for communication in automation systems. Originally, a cyclic data update mechanism was foreseen between an *OPC Server* and an *OPC Client*. Data was simply lost, in case the physical data changed more often than the cyclic update rate or the communication link broke down.

9. *Increased protection against unauthorized data access* was more and more an issue due to the use of *OPC Servers* for remote control and maintenance. As already mentioned, OPC suffered from the cumbersome security settings and firewall configuration natural to DCOM.

10. *Support of method calls* besides the *read* and *write* transactions, like *starting* or *stopping* an actor were not supported by the OPC specification.

OPC UA is again built on the *client-server* concept. But contrary to *OPC Classic* it utilizes well accepted Web-standards like *Web Services*, XML and HTTP for communication between *server* and *client*. Also a binary-optimized *TCP protocol* for high performance connections has been advised. However, the abstract communication model itself does not depend on any specific feature and allows the addition of future protocols.

Another fundamental building block of OPC UA is the data model (see Figure 2.3). The data model is a *meta-model* which defines building blocks and restrictions for the domain specific *information model*. The general architecture of OPC UA data model is very versatile and is designed to represent diverse automation systems. Further, it allows a client to access only parts of the exposed *information model*, without the need to gain knowledge of the whole system.

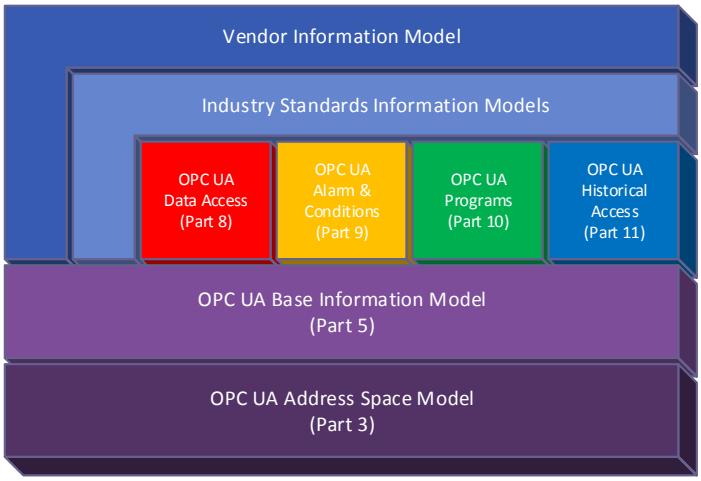

Figure 2.3: OPC UA information model architecture [54]

OPC UA provides analogous information model building blocks for every major standard in the set of *OPC Classic* in order to keep backwards compatibility.

Information Modeling

As depicted by Figure 2.3, information modeling in OPC UA can be separated into several layers. A vendor information model may depend on the foundation of models defined by industry standards and further parts of the OPC UA standard. But essentially, all domain specific models are based on the *base information model* defined in part 5 and further on the *address space model* defined in part 3 of the OPC UA standard [16].

The *address space model* is based on the idea that every entity of information is enumerated and can be accessed by an identifier. The concept very much resembles the idea of how a computer stores and manipulates data. Basically there are two types of elements in the *address space model*: *Nodes* and *References* (see Figure 2.4).

The OPC UA base information model builds on the *address space model* and is the foundation of domain specific information models (see Figure 2.3). OPC UA information models are used to define and describe domain specific node types and constraints. Consequently, information models can be derived from other information models and form a hierarchy.

- **Nodes** are the basic entity of information in OPC UA and are utilized to represent instances, types, and other modeling elements. Every *Node* is characterized by a set of *attributes* dependent on the respective *NodeClass* of a *Node*. *NodeClasses* cannot be customized or extended and stick to the standard node classes predefined by the OPC UA standard.

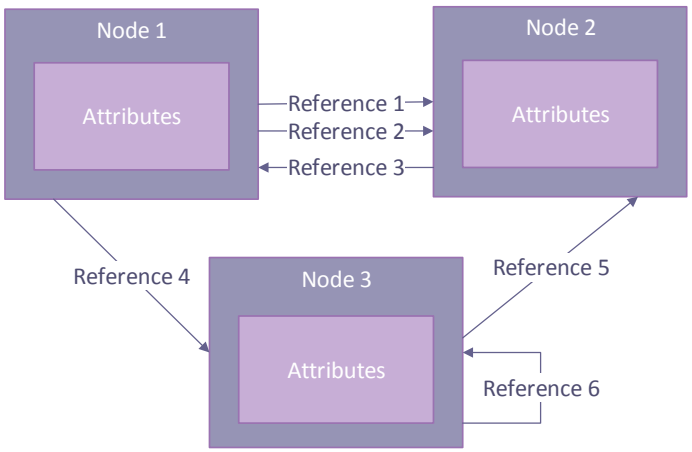

Figure 2.4: Nodes and references [42]

The definition of an *attribute* consist of an *id*, a *name*, a *description* and a *data type*. Access to *attributes* of arbitrary nodes is possible by the services *Read*, *Write*, *Query* and *Subscription / MonitoredItem*. An notable *attribute* is the *NodeId*, which serves as a unique address for a *node* in an OPC UA server.

- **References** are used to connect *Nodes* and are equivalent to directed edges in a graph theoretical model. However, *references* in OPC UA are represented as *Nodes* themselves and are simply characterized by their own *NodeClass*. Unlike other *NodeClasses*, *References* cannot be browsed directly, but only *followed*.

  Basically *references* are classified as *HierarchicalReferences* and *NonHierarchicalReferences*. The semantic of a *reference* is solely defined by its *ReferenceType*, since it does not posses any *attributes*. Again it is not possible to extend the meta-model with own *reference* types.

Standard Node Classes

- **Base** is the fundamental *NodeClass*. Every other *NodeClass* is derived from *Base*.
- **View** provides a subset of the available *NodeClasses* to a OPC UA client. It serves as a starting point for traversing the address space.
- **Object** is similar to the concept of an object instance as perceived from a modern programming languages. An *Object* is a group of contextual related methods and variables. Hence, *Nodes* derived from the *NodeClasses Method* or *Variable* always have an ownership relation to an *Object*.

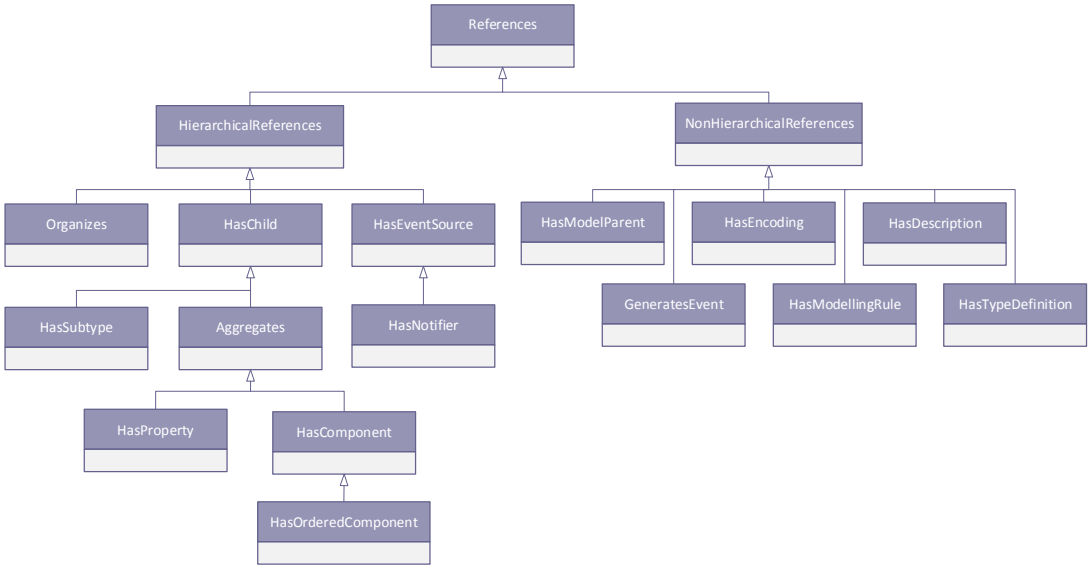

Figure 2.5: Reference types [16]

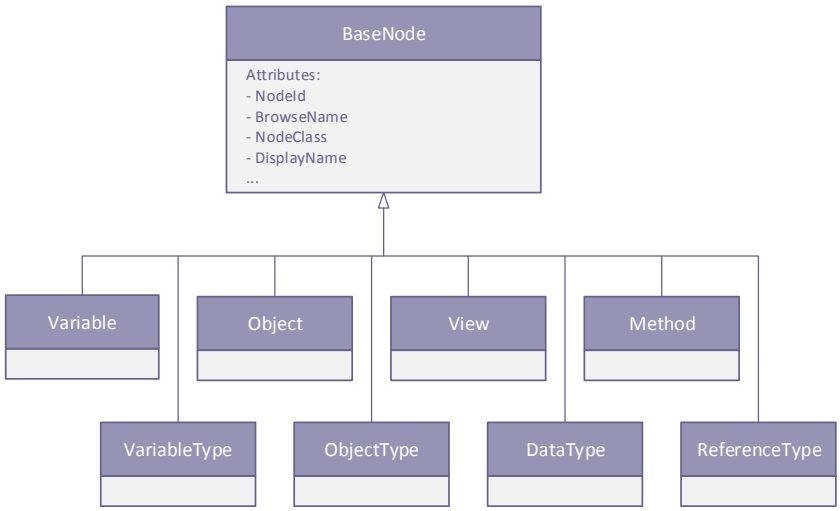

Figure 2.6: Standard node classes [16]

- **Variable** is a *NodeClass* that represents simple or complex data value.
- **Method** defines a *NodeClass* that is used in conjunction with the *Call* service to execute arbitrary functionality on the server.

- **ObjectType** provides type definitions for *Objects*. The *ObjectType NodeClass* resembles the class definition concept of an object oriented programming language.
- **VariableType** provides type definitions for *Variables*. Again, a *VariableType* resembles the concept of a type definition in object oriented programming.
- **DataType** is a *NodeClass* that describes the syntax of a *Variable* or *VariableType*.
- **ReferenceType** defines the semantics of a reference owned by a *Node*.

Reliability

OPC UA bases its reliability on a client and server redundancy concept. There are specialized data structures and services provided by OPC UA that support mirrored instances of both clients and servers.

OPC UA client redundancy is for example important for the continuous supervision and control of safety-critical automation systems. Usually, two or more OPC UA clients are deployed in such a redundancy scenario. One, which assumes the role of the *active* client and all other denoted as *backup* clients. Every client logged into an OPC UA server maintains a connection monitored *session*. In case, the *active* client fails to maintain the *session* due to either a device or connection fault, the *session* information is closed. The *backup* clients keep again a live *session* and monitor the *session* of the *active* client. When the *active* client *session* is closed, a *backup* client initiates a so called *TransferSubscriptions* service. This service transfers all data notification subscriptions form the previously *active* client to the *backup* client taking over. Any notification event that might occur during such a *TransferSubscription* service execution is queued on the server and then pushed to the newly *active* client.

OPC UA server redundancy is again split into the notion of *transparent* and *non-transparent* server redundancy. In the *transparent* redundancy case, all servers that are members of such a *transparent* redundant pool are mirrored. Hence, all participating servers in such a pool maintain exactly the same *session* and data information. When an *active* server ceases functioning, one of the *backup* servers takes over without interrupting any client operation.

In the *non-transparent* server redundancy case, client action is necessary for a failover. In general, the client needs to create a new *session* on the *backup* server once the *active* server has failed. Data notification subscriptions are either duplicated on all servers of a *redundant* pool or transferred by the *TransferSubscription* service. As for the *backup* servers, the OPC UA standard defines different *failure* modes:

- **Cold:** The *backup* server is running. Once the *active* server fails, the *backup* server becomes *active*. A client then needs to completely recreate the *session* on the new *active* server.
- **Warm:** The *backup* server is running and also accepts client connections. A client maintains both *sessions* on the *active* and on the *backup* server. Contrary to the *hot* case, the *backup* server does not communicate with actual field level devices, but rather mirrors information from the *active* server.

- **Hot:** The *backup* server is acting by no means different than the *active* server. Clients again connect to both the *active* and backup *server*.

oBIX

oBIX is an industry standard specifically designed to provide a common and platform independent interface for various building automation systems (BAS). The standard defines its own low level object model (see figure 2.7) and publishes it by the means of Web services and XML. Like OPC UA, oBIX is based on a client/server architecture, where a client can interact with the modeled BAS via the model exposed by the server. Version 1.0 of the standard was released in 2006 and a revised version 1.1 is in the works since 2009.

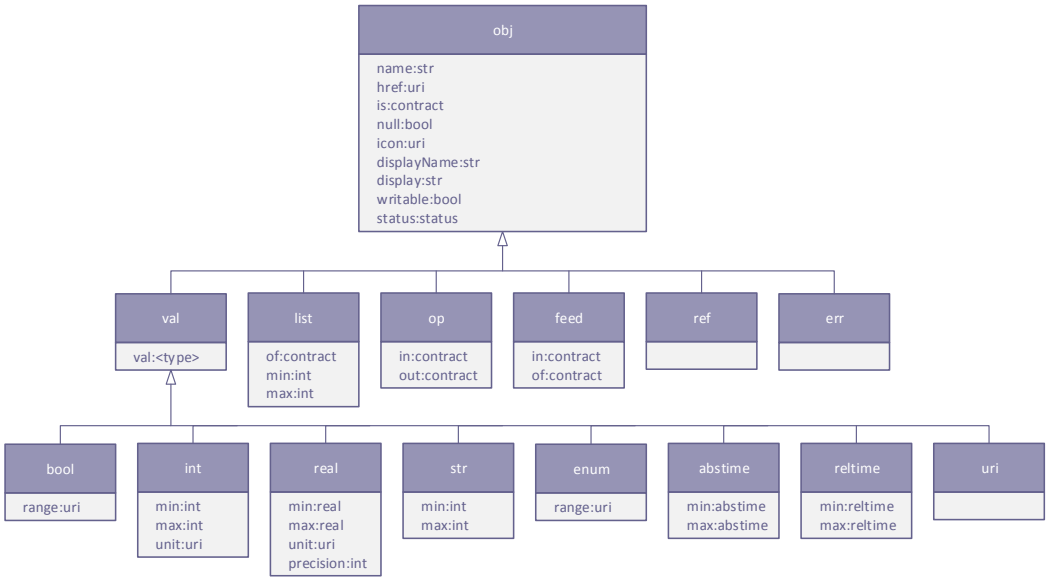

Figure 2.7: Object model [14]

The standard consequently follows an *object first* design approach, where everything is an object and is derived from an common root object (see Figure 2.7). This includes not only data types, but also operations. The objects themselves are represented utilizing the oBIX XML scheme. Hence allowing multiple inheritance and composition of objects resulting in a highly extensible data model.

Further, oBIX defines the concept of *contracts* in order to provide templates for recurring patterns throughout the data model. There is already a predefined set of *contracts* defined by the oBIX *core contract library* (e.g. *unit* defines a *object* resembling the physical SI-units). Also a number of specialized objects exist:

- **Watch:** Implements an *event queue* on the oBIX server, which is then continuously polled by clients for updated data.

- **Points:** Resembles the concept of a *data point* in automation systems by the means of a direct mapping of an object to a physical actuator or sensor. The goal of oBIX is to provide a normalized representation via this *object*.

- **History:** Provides access to persisted information from the past for trending or logging purposes.

- **Alarm:** Is an *object* that implement event notifications for either a user or an application.

The Web service part is implemented using the Representational State Transfer (REST) software architecture style. Therefore the set of operations is limited to three basic service calls [14]. The *Read* service returns the current state of an object in a XML format. The *Write* service takes an XML description, updates the state of an object and finally returns the updated state again as an XML document. The *Invoke* service takes the input parameter again via a XML description, executes the method specified by a Uniform Resource Identifier (URI) and returns the result again in a XML format.

Similar to the address space concept in OPC UA, oBIX uses a complementary concept of two naming systems. First of all every *object* that needs to be accessed via REST calls is identified via a URI. A URI is basically a string of characters incorporating the hierarchy layers and a common name that uniquely identifies an *object* residing in a oBIX server. Also URIs are used for referencing between *objects*. Second, naming of nested objects is done via a *name* attribute.

Since oBIX is considered a fairly low-level standard, it does not define any reliability measures. Although a redundancy concept similar to OPC UA is possible, oBIX does not define any services that might assist a controlled failover between clients or servers.

EEBus

The *EEBus* specification [19] is designed to interconnect BAS with metering and control networks deployed by power supply companies to enable the so-called *Smart grid*. The term *Smart grid* refers to technology that allows bidirectional communication of energy demand and supply between consumers and producers, leading to a more efficient utilization of available resources. The *EEBus* standard is still in draft stage, but its release of the final specification is soon to be announced.

The core element of the *EEBus* communication model is the *gateway* device (see Figure 2.8). Like *oBIX* (see Section 2.3 – oBIX), the *EEBus* specification is based around an XML data model located at the *Outbound Communication Layer*. The XML data model is designed to represent information in a technology neutral form and can be accessed in a REST like manner via Ethernet and IP. The *EEBus* data model is designed around a *function-first* design approach, where everything is a callable function. Hence, access to various datapoints is achieved by calling the corresponding *read* and *write* access functions defined in the *EEBus* meta-model. The hierarchy of the data model deployed by the *EEBus* is divided into different levels:

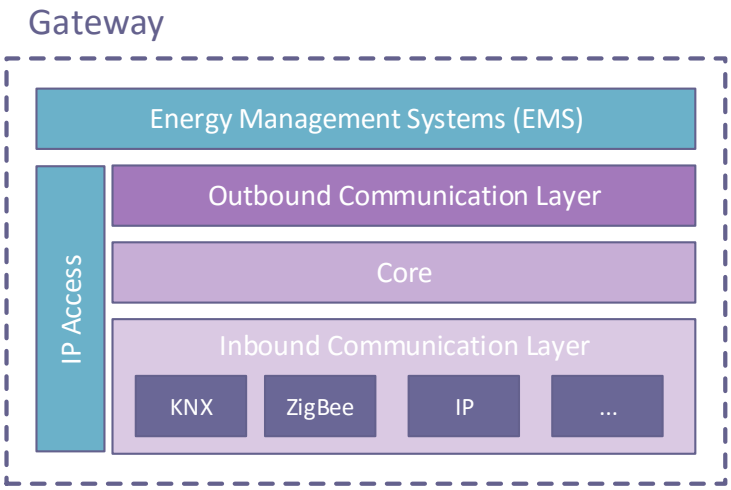

Figure 2.8: *EEBus* gateway architecture

- **Class:** Highest level of the *EEBus* defined problem domains. (E.g. *Metering*)
- **Sub-Class:** Is an optional hierarchy level and allows a fine grained differentiation between different functionality.
- **Function-Group:** Groups contextual related *functions* for access to capabilities of modeled entities.
- **Function:** Describes concrete interaction methods, like *read* and *write* functionality for arbitrary access to datapoints.

Different field level protocol mappings to the *EEBus* data model are foreseen at the *Inbound Communication Layer* (see Figure 2.8). However, the draft specification still lacks the description of detailed binding mechanism between the data model and different fieldbus technologies.

The current draft of the *EEBus* standard does not include any reliability measures. However, reliability concepts as deployed by the proposed *gateway* approach (see Section 3.1 – Reliability) are possible.

2.4 Conclusion

The current available solutions to the stated integration problem either introduce additional layers of abstraction to an automation system or lack a reliability concept. OPC UA (see Section 2.3 – OPC UA) implements a versatile information model and also leverages a reliability concept. However, due to the SCADA centric orientation, a noticable amount of overhead is introduced at the information model. Due to the verbosity of OPC UA telegrams a separate high bandwidth backbone network is needed between OPC UA clients and servers for synchronization and hence fails to achieve integration at the field level.

oBIX (see Section 2.3 – oBIX) is only focused on the domain of BAS and lacks a reliability concept. The rather new *EEBus* specification (see Section 2.3 – EEBus) resembles the architecture of a single *gateway* device of the proposed approach, but again does not define any reliability measures.

Hence, there is no available solution that is capable of integrating two different field level protocols at the field level and simultaneously deploys a reliable concept.

CHAPTER 3

Design and concept

As already elaborated in Chapter 2, there are two viable approaches to the integration problem at the field level. On the one hand the *tunneling* concept and on the other hand the *gateway* approach. The *tunneling* solution has been already extensively discussed in [41], whereas the *gateway* solution was only briefly analyzed in [50].

Hence, a *gateway* approach is designed and evaluated, that enables information exchange between fieldbusses. Further in this chapter, the requirements such a *gateway* shall fulfill are stated (see Section 3.1). Then, an analysis of different usage scenarios and possible network topologies is performed. A feasible *gateway* approach should be able to be connected with any of the presented communication structures (see Section 3.2).

Then, the overall architecture is advised, based on the *gateway* approach basics already discussed in the previous chapter (see Section 3.3). The architecture of the proposed *gateway* solution gives then rise to a detailed analysis of each major building block. At first, the information model is discussed (see Section 3.4), which describes a possible meta-model to structure the internal state of a *gateway* device. Further on, in Section 3.5 the *reliability* concept is introduced.

3.1 Requirements

A proper *gateway* solution should comprise three key properties. Namely, a feasible *masquerading* or *address translation* respectively, an appropriate *reliability* concept and to be able to *seamless integrate* into the chosen fieldbusses network topology.

Seamless integration

In order to promote rapid deployment and hassle free integration, a feasible *gateway* solution should blend into an existing network topology as natural as possible. Hence, devices from one fieldbus shall neither need any special knowledge about the *gateway* device itself, nor about other fieldbusses attached to it. *Seamless integration* is a major requirement, since fieldbus devices often only allow limited user-defined application layer programs and configuration of

their protocol stack behavior. Even worse, some field protocols allow no modification at all to their operational behavior.

A lot of field protocols promote various device classes with different implementation levels of the protocol stack functionality. Device classes that realize the role of mediators in a field network are good choices for a device class usable by a *gateway* device. Typical mediator device classes are either coupling devices between different network segments, physical media or devices with some kind of message routing and forwarding capabilities. A *gateway* solution, implementing a mediator like device class can then simply pretend that other attached field protocols are just different segments of the same network.

There are several topological network structures typically used throughout different fieldbus protocols. Namely, *star*, *ring*, *line/bus*, *tree* and *mesh*. But the most commonly used network topologies are the *line* or *bus* topology for wired networks and the *mesh* topology for wireless networks [51]. The available device classes, and therefore the possible *seamless integration* approaches are highly dependent on the network topology. As an example, device classes that implement routing capabilities are typically foreseen in field level protocols that are based on a *mesh* network. In order to give an example of the selection of a proper device class, two prominent representatives for the two most common fieldbus topologies are analyzed:

- The *ZigBee* standard serves as a good example for a wireless automation protocol utilizing a *mesh* network topology. There are three device classes defined in the *ZigBee* protocol: *End device*, *router*, and *coordinator* [15]. The *end device* class is intended for constrained devices that only need minimal networking functionality. From a graph-theoretical perspective, an *end device* is always a leaf. The *coordinator* class acts as a network *master* providing *routing* capabilities, *security* mechanisms and manages the network in general. There is always only one *coordinator* allowed in a *ZigBee* network. Finally, the purpose of the *router* device class is to relay and route messages throughout the *mesh* topology. In case of the *ZigBee* protocol, the only possible choices for a feasible *gateway* solution are the *router* or the *coordinator* device class due to their *routing* capabilities. Thus, messages to other fieldbusses can be redirected through the *gateway* by utilization of the inherent *routing* mechanisms of the *ZigBee* stack (see also Section 3.1 – Masquerading).

- An example for a well-established wired automation protocol based on a *line* topology is the *KNX* standard [34]. More specifically, a *line* topology comprising of three distinct stages is advised by *KNX*. The superordinate line is called the *backbone line*, which further connects down to the *main line* and finally to the plain *lines*. The connection between each stage of the *line* hierarchy is carried out by special devices. The connection from the *backbone line* to the *main line* is realized by a *zone coupler*, whereas the connection between the *main line* to the plain *line* is done by a *line coupler*. The usual *KNX* device is called *bus device* and resembles an arbitrary communication partner.

  Following the directive of choosing device classes that implement message relaying capabilities, one of the *coupler* devices would be a good candidate for a possible *gateway* approach. However, also a simple *bus device* may be used to realize a *gateway*. This is due to the shared medium nature of a *KNX* line, where a message transmitted on an arbitrary

line is visible to all communication partners connected to the very same line. Hence, a *gateway* device can listen and relay any telegram as long as it is forwarded to its *line*.

Another possibility for a *gateway* solution to seamlessly integrate with a field protocol is the *promiscuous mode* of communication. In this mode, a possible *gateway* implementation behaves like an *eaves dropper*. Meaning, that a *gateway* device listens to every message on a fieldbus and intercepts telegrams that need to be forwarded to another fieldbus. The other way round, the *gateway* injects information from a foreign fieldbus by packet spoofing mechanisms.

The biggest disadvantage of a *promiscuous* mode *gateway* is the missing separation between traffic destined in another field network and normal bus traffic. Thus, every *eaves dropped* packet needs to be forwarded to the *application layer* of a *gateway* device for inspection. Whereas, the utilization of mediator device classes benefits from the mechanisms present at the *protocol stack*.

Masquerading

In Ethernet and IP based ICT networks, *masquerading* [23] is a common mechanism to translate messages between different classes or hierarchy levels of networks. Actually there are two common techniques for the same purpose, one is Network and Address Translation (NAT) and the other *IP Masquerading* or Port and Address Translation (PAT).

Assume that there are two network segments, each assigned a different IP class. One network segment is named *private network* and the other one is denoted as *public network*. A router or gateway, implementing NAT, would then make use of a so called *address translation table*, where IP addresses from the *private network* are mapped to IP addresses from the *public network* and vice verse. Messages arriving at the NAT router or gateway are examined and the destination IP address is replaced by the equivalent counterpart of the other network segment stored in the *address translation table*.

On the other hand, *IP Masquerading* uses a different kind of mapping to translate messages between different network segments. A single IP address is used at one network segment, where communication partners can connect to different ports. Further, those different ports are mapped to different IP addresses, residing in the other network. Hence, an *IP Masquerading* gateway resembles the functionality of an aggregation device or proxy.

In case of the proposed field level *gateway* solution, *masquerading* refers to the concept of translation and forwarding of messages between field devices located in different disjoint fieldbusses. Like *IP Masquerading*, the goal is to allow interconnection of different fieldbusses without adding additional knowledge to the field devices. Given, that the *gateway* implements a stateful *translation mechanism*, like an *information model*.

Every device present in *field network A* that is to be visible to *field network B* and vice verse is modeled at the *gateway information model*. Hence, messages targeted to a field device originating in another fieldbus are translated and forwarded with the help of such a model, further called *virtual device*. Thus, a device residing in one *field network*, that wants to send information to a device residing in another *field network* does so by communicating with the *virtual device* at the *gateway*. Further, the information is forwarded to its final destination by the *virtual device*. Hence, the virtual device at the *gateway masquerades* the real device residing at the other field-

bus. Depending on whether the *gateway* information model is organized as a logical or physical topology, either NAT like behavior or *IP Masquerading* like behavior is resembled.

A similar classification has already been performed by the *INTERBUS Integration* working group, in order to map *Interbus* devices to the *Profinet IO device model*. Basically, there are three approaches how to organize such a *virtual devices* at the *gateway information model*:

- **Compact topology:** The compact topology simply concentrates all available bus devices in a single *virtual device* at the *gateway information model*. Although, the compact topology promotes a flat *information model* design, it also implies loss of any network structure. Hence it is only suitable for small instances of field networks.

- **Logical topology:** Suppose, that the *gateway* is constructed such that it facilitates a logical topology for its information model. Such a topology promotes the grouping of devices by their functionality in logical entities. Assume a simple heating system comprising of a heater and a temperature sensor as an example. Although both are independent bus devices, they can be presented as a single *virtual device* at the *gateway information model*. On the other hand, it is possible to represent the functionality of a single physical device in two separate logical *virtual devices*.

- **Physical Topology:** Assume the case of a physical topology based information model, a NAT like behavior is carried out. Like in NAT defined for Ethernet and IP based systems, the virtual device performs a direct translation of the address and content of a message into the native format of the other field protocol. In contrary to the logical topology, where an *IP masquerading* like behavior is applicable. In *IP masquerading* every port is redirected to a different computer. So to speak, the logical *virtual device* redirects calls to different services to possible different field devices. Whereas, a *physical topology* requires a strict bijective mapping between field devices and representing *virtual devices*.

In conclusion, the choice whether to favor a logical or a physical topology to implement a proper *masquerading* mechanism is arbitrary. Regarding the fact, that the proposed *gateway* is targeted at field level protocols approaching the same problem domain, a bijective mapping is most likely. In any case, a logical topology is a superset of a physical topology. Thus, a possible realization of the proposed *gateway approach* is more flexible by supporting a logical topology.

Further, a feasible *gateway* approach shall always obey the seamless integration requirement (see Section 3.1 – Seamless integration). Some field protocols, like *CANopen* [2], define different categories of field devices in order to enforce compatibility between different device vendors. Hence, also a possible *information model* implementation should comply to such categories.

Reliability

Fieldbus technology is already widely used throughout safety critical applications. Hence, *reliability* is an important requirement for a feasible *gateway* approach. Up till today, most existing *gateway* solution focused on the interconnection problem, but gave no considerations about *reliability* of the resulting system. A single *gateway* device connecting two different field protocols easily becomes a single point of failure, unless some kind of *reliability* mechanism is deployed.

A *gateway* should at least be able to handle as many different failures as an attached fieldbus in order to retain the same level of *reliability*. A well-established reliability mechanism leveraging the principle of *fault tolerance* is *gateway redundancy*. Assume two fieldbusses, which are both able to tolerate one failure each and are interconnected by a *gateway* device. Then, such a *gateway* needs to abide one failure without major interruption of operation. Hence, a second, *redundant* gateway needs to be deployed, which resumes operation in the event of an outage of the first *gateway* device. Therefore, a mechanism needs to be advised, which allows *gateway* devices to monitor each other, synchronizes their information model and coordinates the *fault isolation* procedure.

A possible synchronization protocol, originally designed for redundant router configurations in Ethernet and IP based ICT systems is VRRP [43]. This mechanism groups redundant routers in a single virtual device. Therefore a common IP address is assigned to that virtual device, apart from the individual IP addresses of the physical router devices. Hence, a communication partner can either address the virtual device by its common IP address or single devices by their individual IP address. One router, part of a virtual device, is assigned the role of a master and hence claims ownership of the common IP address. The master device is then responsible to handle all incoming traffic through the common IP address. All other routers assume the role of backup or hot-standby devices. In order to detect a fault, the master router sends out periodic *heart-beat* packets. In case the backup devices are unable to receive several *heart-beat* messages in a row, an election procedure is started and a new master device is chosen.

In conclusion, a proper *gateway* solution shall be able cooperate with similar devices and form a redundant compound. A valid *redundancy* mechanism takes care of synchronization and monitoring of all members of such a *gateway* compound. Monitoring messages have to be sent out on all available communication links, in order to detect link faults. Further, it is desirable that also synchronization traffic is distributed on all available fieldbus connection links. Thus sharing the load of the synchronization traffic on all available fieldbus connections and therefore minimizing the communication overhead from the *gateway* devices on the automation system as a whole. As an alternative, a separate backbone connection may be established between individual *gateways* to completely separate normal bus traffic and communication between *gateways*.

3.2 Communication structure

The *gateway* communication structure describes considerations regarding the requirements stated in Section 3.1 and their fitness for different fieldbus network topologies. As already stated in Section 3.1 – Seamless integration, individual fieldbusses are based on different network layout concepts, whereas *mesh* and *line/bus* are the most frequently used topologies. But for the design of a general applicable gateway approach, all possible topologies have to be considered, and a *gateway device* must be able to interconnect those. Aggravating, also network topology cases leveraging *redundancy* concepts need to be analyzed in conjunction with a feasible *gateway* approach.

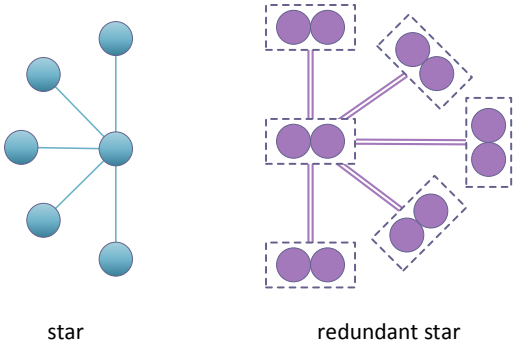

Figure 3.1: Star topology

Star

The *star* topology directly connects every fieldbus device to a central communication hub. Before the emergence of field protocols, such a hub was usually a PLC controlling directly attached sensor and actuator devices. After the advent of fieldbusses, the *star* topology nearly vanished from the industrial and building automation domain. Throughout the last decade however, the *star* topology celebrated a comeback by the increased usage of Ethernet and IP and Ethernet based fieldbusses. In the case of Ethernet and IP, the central node would be a switch, hub or router device. [51]

Given that the field devices in a *star topology* have no message forwarding capabilities, a *gateway* solution would naturally be best placed in the role of the central communication hub. In the other case, where field devices are able to redirect messages, a *gateway* device may also be placed at a leaf position of a *star* topology. If none of the prior mentioned conditions is applicable, meaning that field devices have neither message forwarding capabilities, nor that the *gateway* is allowed to be the central node, no general applicable *gateway* position can be found. Depending on the deployed field protocol and intended application, the *gateway* device can be attached to the central device (e.g. PLC) via internal mechanisms. Another possibility for the *gateway* is to behave like a bridging device acting as a transparent bridge placed at a leaf position of the *star* topology.

A star topology based fieldbus may also be organized in a redundant way, meaning that field devices and communication links are present multiple times. Thus, in the event of a failure of a redundant component, a functioning part replaces the faulty one. A single *gateway* device would then become a single point of failure. Hence, multiple gateways need to be deployed and combined to retain the desired level of fault tolerance. Suppose a scenario, where the *gateway* is positioned as the central element of the *star topology* with multiple links connecting to each field device. Further, it is assumed that the field devices are not able to redirect message traffic. Hence the *star* topology based fieldbus cannot be utilized to synchronize and monitor the individual devices of a redundant *gateway* compound. Then, a separate backbone link between the single elements of a *gateway* compound has to be implemented (see Section 3.1 – Reliability).

Furthermore, such a backbone link needs also to be laid out in a redundant way to ensure a sufficient level fault tolerance.

A fault analysis, given that all components emit a fail-silent behavior, is described next. As depicted in Figure 3.1, a redundant *star* topology based system may be designed such as that for every field level device originating at a leaf position of the *star* topology, there exists an equivalent backup device forming a redundant compound. Additionally there is also a backup communication link between the central element and the field devices of a redundant compound available. Thus, in the event of a link fault, the backup communication link resumes operation. Further, in case of a device fault, the backup field device continues service, resulting in an arrangement, where such a system is capable of tolerating one link fault and one device fault. More communication links or backup field devices can be added to enable immunity against even more failures. Also the *gateway* device has to be laid out multiple times, in order to avoid a single point of failure at the central node. Thus, in the event one of the central *gateway* devices ceases operation, one of the deployed backup *gateways* resumes operation.

Assume again a *gateway* device is part of a redundant *star* topology fieldbus. This time the *gateway* is not placed at the central hub position, but rather at one of the leaf positions (see Figure 3.1). Again, every field device is laid out in a redundant manner. Further, every communication link is present multiple times and shared among the individual devices of redundant field device compounds. Thus, the fault hypothesis, stating the tolerance of one device fault and one link fault remains the same for a system deploying double field devices and connections. In order to provide the same level of fault tolerance as the previously described system, individual *gateway* to the same extent as field devices of a redundant compound have to be present. Also the amount of shared communication links has to be the same as for redundant field device compounds. Thus, the fault hypothesis of the overall system is not compromised by the *gateway*. The fault analysis is similar to the scenario, where the *gateway* is placed at the central node. In case of a device fault at a *gateway* node, one of the backup *gateway* nodes resumes operation. In case of a link fault, one of the additional communication links from the central hub device continues service. Hence the proposed *gateway* solution retains the same level of fault tolerance as the rest of the field network.

Ring

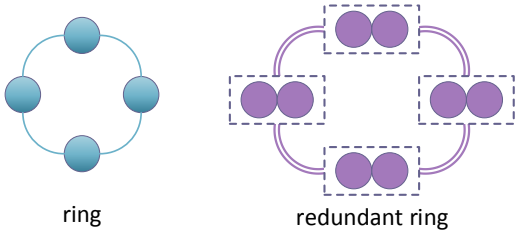

Figure 3.2: Ring topology

Another fieldbus topology variant is the *ring*. Each field device of a *ring* topology is connected to the next bus participant by the means of a disjoint communication channel. Hence a field device is connected by a separate input link from one neighboring device and output link to another neighboring device. A great advantage of the *ring* topology is its inherent immunity against a single link fault. Fieldbus participants sensing a faulty connection or device next to them, simply bridge their internal network input and output channels. A prominent example of a fieldbus utilizing the ring topology is *INTERBUS* [31].

All nodes of a *ring* have inherent message forwarding capabilities as required by the topology. Hence, a *gateway* device may be inserted as an arbitrary participant in a *ring* based field network. A *gateway* device that integrates in such a *ring* topology needs to mimic a complete segment of the *ring*. Therefore, virtual field devices modeled at the *gateway* behave like *ring* communication partners. Incoming messages at the physical input channel of the *gateway* device are therefore forwarded through all virtual field device models until finally forwarded back into the real *ring* network. A particularity of the *ring* network structure is that a device fault or link fault basically degenerates the *ring* to a *line* topology (see Section 3.2 – Line). A *gatway* device operates by no means other than other *field* devices part of a *ring* based field network. Fault tolerance against a single link fault is still valid. However, in case the *gateway* device fails, the whole *segment* of virtual field devices is unreachable.

A redundant *ring*, as depicted by Figure 3.2, leverages multiple connections between field devices compounds, which consist of several nodes as well. In case of n communication links, $2n - 1$ link faults are handled at worst. Further, assuming that a redundant node consists of m field devices, $m - 1$ device faults are tolerated. A redundant *gateway* node residing in a *ring* network and required to retain the same level of device fault tolerance needs also to consist of m individual devices. Also n communication channels are used to interconnect the *gateway* compound with the remaining *ring* in order to provide the same level of link fault tolerance.

In the event of a link failure at one of the *gateway* devices in a single, non-redundant *ring*, as depicted by Figure 3.2, the same mechanism is performed as for every other bus participant. The input communication channels and output communication channels of the devices neighboring the faulted connection are linked in such a way that messages are looped back through the still operating connection links. In case of a device fault at the *gateway*, the neighboring nodes will again shorten their communication channel and isolate the *gateway* device. Unfortunately, this effectively shuts down communication to any virtual field device modeled at the *gateway*.

Continuing with a fault analysis of the redundant *ring* topology scenario, the *gateway* nodes behave just like every other bus participant. Synchronization of individual *gateway* appliances forming a compound may be done either via the *ring* network or a separate backbone link. Like for the *star* topology (see Section 3.2 – Star), a separate backbone link must be implemented as such that it retains the same level of fault tolerance as the *ring* based field network. Assuming n individual connections between each disjoint communication partner in a *ring* topology, at least $2n - 1$ backbone channels interconnecting the *gateway* devices must be present.

Line

The *line* topology is the most frequently used network structure throughout all fieldbusses [51]. Its development was a direct response to the cabling overhead inherent to field protocols utilizing

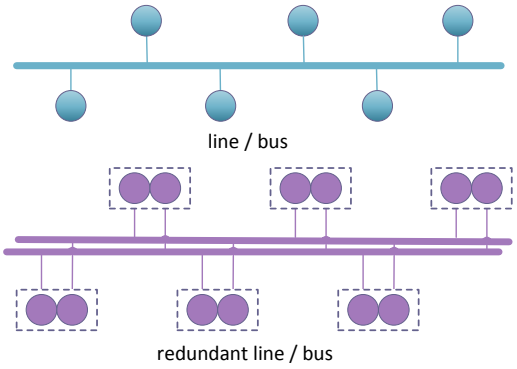

Figure 3.3: Line topology

the *star* topology. The *line* or *bus* topology consists of a single communication channel that is shared among all connected field devices. Prominent usages of the *bus* topology are *RS485* [47] or *CAN* [1]. Remarkably, the *line* topology is usually not deployed in its purest form. Often, a fieldbus network utilizing a *line* topology is partitioned, where separate *lines* are used for each level of a hierarchy. Such a network structure, consisting of multiple levels of *busses* is called a *tree* topology (see Section 3.2 – Tree).

A *gateway* device may be inserted instead of any arbitrary field node part of a *bus* topology. A *line* topology based field protocol always leverages an addressing scheme to direct messages to individual bus participants. Hence, virtual field devices exposed by a *gateway* appliance are each assigned an individual address. A *gateway* node then listens on the bus for messages on their behalf, facilitating a so-called *promiscious mode* of communication. Hence a *gateway* acts like a coupling device interconnecting different *lines* as defined for field protocols utilizing that implement a *tree* topology [34].

Given a redundant setup, like depicted in Figure 3.3, every field node consists of multiple independent devices grouped together in a redundant compound. In order to overcome link faults, the communication channels interconnecting such redundant compounds are also present multiple times. A *gateway* device placed in such a redundant *line* network must be arranged like the other bus participant. Hence, a redundant *gateway* compound is facilitated, which consists of multiple *gateway* devices and is redundantly connected to the available bus *lines* by several disjoint communication links.

In the event of a device fault, the affected field node is no longer reachable. If the device fault happens to be at an arbitrary *gateway* node, all of its modeled virtual field devices become unreachable. Considering a worst case link fault, occurring in a simple *line* topology, splits the network in half, meaning, that field nodes, including *gateway* nodes, completely lose all communication capabilities to all bus participants beyond the affected link segment. This usually renders the complete fieldbus unusable and effectively shuts down the whole network.

A redundant *bus* network, deploying n communication links, is able to handle up to $n-1$ link faults. It is also able to tolerate $m-1$ device faults, where a compound field node consists of m

individual devices. The same case is true if the field node is in fact representing a *gateway* node. Synchronization between different *gateway* devices, member to a redundant *gateway* compound can either be achieved by utilizing the available fieldbus links or separate backbone links. Again, to retain the same level of fault tolerance, the backbone links must be present in just the same amount as the redundant fieldbus *lines*.

Tree

The *tree* topology is a hybrid network structure composed of a mix of a *star* topology and *line* topologies. A *tree* topology in its simplest form consists of a root device or a root *bus* with possible several devices attached. The attached nodes are connected with disjoint point-to-point communication channels (*star* topology, see Section 3.2 – Star) to the root device or *bus*. The previously mentioned field devices can again service as a single connecting device for further field devices or *bus* communication channels (*line* topology, see Section 3.2 – Line). Important protocols that leverage a *tree* topology are *KNX* and *LONWorks*.

The *star* and *line* topology have been discussed in Section 3.2 – Star and Section 3.2 – Line respectively. In fact, aspects of a *tree* network structure, like interconnected *line* hierarchies or deeply nested *star* like structures have been dealt with in the accompanying sections. Since the *tree* topology does not display different behavior as the already discussed topologies, results gained from their respective analysis are also valid for the *tree* topology.

Mesh

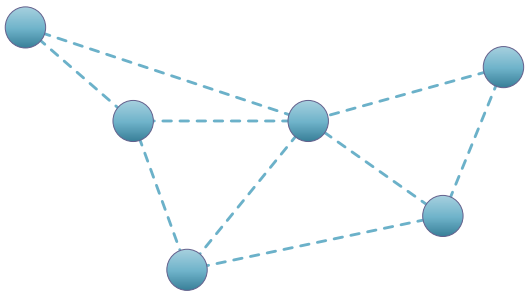

Figure 3.4: Mesh topology

Mesh topologies had little to no importance in the domain of traditional wired field protocols [51], because of the routing mechanisms needed to keep telegrams from circling. However, the *mesh* network structure is the most important topology for wireless field protocols, such as *ISA100.11a* [18] or *ZigBee* [15]. Although other topologies are implemented for the prior mentioned protocols, a *mesh* topology has the advantage of enabling arbitrary multiple links between field devices. Therefore, the fault tolerance regarding permanent and transient distortions inherent to the wireless communication medium is drastically increased by the available connections to other communication partners in the same network. Considering that *mesh* networks are not

used for wired field protocols, the *gateway* approach is discussed with a focus on wireless fieldbusses. However, any gained insights may be transferred to a possible wired fieldbus as well without any loss of generality.

Wireless field protocols usually introduce different device classes for their field devices. Each device class has a different level of communication capabilities. Taking the *ZigBee* protocol as a representative example: There are three device classes, namely *end device*, *router* and *coordinator*. An *end device* possesses only the very basic communication capabilities needed for a *ZigBee* device to function. This includes sending and receiving of telegrams, authentication and encryption. A *router* additionally defines message redirection and forwarding capabilities. Whereas a *coordinator* device is capable of additional administrative services like key distribution and acknowledging authentication requests. Further, a *coordinator* implements all capabilities of the *end device* and *router* communication stacks.

In other discussed network topologies, it was always emphasized that a possible *gateway* device needs to assume the role of a mediator to naturally blend into the communication infrastructure. For a *mesh* topology, a single *gateway* may also act as an *end device* without message redirect capabilities. The key for proper operation in such a scenario is the *promiscuous mode* of communication and *packet spoofing*. A wireless field network based on a *mesh* topology deploys an addressing scheme to uniquely identify devices. Each virtual field device residing inside a possible *gateway* device therefore also needs a valid address assigned to it by the *mesh* network. Messages directed to virtual field devices then need to be captured by the *gateway* with the help of the *promiscuous mode*. Furthermore, messages directed from virtual field devices need to be sent out by utilizing their very own address, but not the *gateway* devices own address.

However, the previously outlined solution is rather cumbersome and uses mechanisms like *message spoofing* that are usually not foreseen by the protocol specification. A much better approach is to facilitate the communication capabilities of defined mediator device classes like the *router* or *coordinator* class specified by the *ZigBee* protocol. A single *gateway* device would announce itself to the network as a field node with *routing* capabilities. Every virtual field device residing in the gateway is then modeled like an *end device* exclusively attached to the *gateway* node.

A redundant *gateway* compound facilitates two or more individual *gateway* devices with *routing* capabilities. Hence, messages directed to a virtual field device have the option of traveling along two communication paths. In order to prevent messages from circling, protocols utilizing a *mesh* topology, must advise some mechanism to select and determine a single communication path without loops between arbitrary communication partners. Especially wireless protocols consult connection metrics like *packet drop*, *signal strength* or *bit error rate* to choose a path that is most likely to deliver a message without retransmits. Hence a *gateway* acting as a backup device ought to artificially weaken the metrics for its virtual field devices. Even further, a backup *gateway* states that virtual field devices are unreachable, until it assumes the role of the master *gateway*. Thus, a communication path along the *gateway* master device is chosen by the means of the field protocols own routing mechanism.

The impact of a node fault in a *mesh* topology scenario depends on the device class of the faulty node. In case the affected node resembles an *end device*, the node simply becomes unavailable, but the rest of the network remains intact. If the prior mentioned node happens

to be a *gateway* device utilizing *promiscuous mode* of communication, all virtual field devices are unreachable. Further, if the faulty node implemented *routing* capabilities, a device fault foremost results in the fact, that the affected node is removed from the network. Furthermore, field devices which are solely connected by a single communication path via the concerned node are isolated from the rest of the network. Thus, it is beneficial that *mesh* topologies are laid out in such a way that every node part of a network is connected by several communication paths. Finally, if a single *gateway* device implementing the *router* or *coordinator* device class is faulty, all virtual field devices residing in the *gateway* vanquish from the network. Additionally, field nodes which are solely connected via the concerned *gateway* node to the remaining network are unable to exchange messages anymore.

In case of a link fault at an arbitrary field node, the device tries to redirect traffic via an alternative route. If there is no alternative route, the device becomes isolated and is no longer able to participate in communication with other network nodes. Assuming that the affected node is a *gateway* device and no other communication path exists, all messages from and to the virtual field devices are dropped. Hence, all devices modeled by the *gateway* vanquish at once from the *mesh* network.

Suppose a redundant *mesh* network scenario, where every field device is present at least twice. Furthermore, each field device is connected via two disjoint communication paths to each other. In case of an arbitrary field device fault, an associated backup field device is available to resume operation. A device fault may also result in several broken communication paths. As stated in our assumption about the redundant *mesh* network under test, every field device has an alternative route that does not involve the faulty device. Hence message exchange is still possible throughout the whole *mesh* network. The same holds true in case the concerned node is a *gateway* device. Additionally the backup device resumes operation of all virtual field devices. In fact, the backup *gateway* works in such a way, that it appears as an additional route to any arbitrary virtual field device modeled at the individual *gateway* devices of a redundant compound. The routing mechanisms of the field protocol have to adapt the communication paths to the virtual field devices, since a backup *gateway* device most likely maintains a different set of communication links.

Synchronization traffic between *gateway* devices of a redundant compound is transacted either by means of the fieldbus, or a separate backbone link. Like for other topologies, a possible separate backbone connection has to be laid out multiple times to meet the desired fault tolerance against link faults. Hence, in case n link faults are handled by a *mesh* network, at least $n + 1$ separate backbone links must be present.

3.3 Architecture

Considering all the possible communication structures, an internal structure for the *gateway* can be advised. Essentially there are two main tasks that a *gateway* device part of a logical redundant compound needs to fulfill. First, it needs to forward and translate the communication between the attached field networks. Second, a *gateway* has to coordinate its operation with all the peer *gateways* in a logical redundant compound.

In Section 2.2 – Gateway solutions of Chapter 2, a general *gateway* approach is described and pictured by Figure 2.2, which serves as a well-established basic model for a feasible *gateway* approach.

As depicted by Figure 2.2, each *gateway* device must be capable to communicate with the attached fieldbusses. Thus, *fieldbus drivers* must be available that handle the operation of individual attached field networks. Such *protocol stacks* must implement at least the *Physical*, *DataLink*, *Network* and *Transport Layer* of the International Organization for Standardization (ISO)/Open Systems Interconnection (OSI) model [23] (See Figure 3.5).

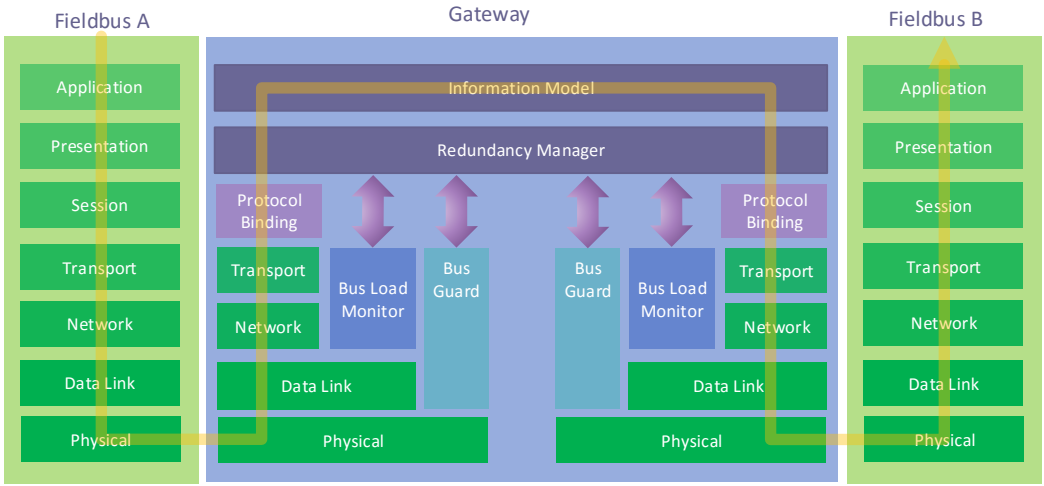

Figure 3.5: ISO/OSI 7 layer model incorporating a gateway

Field level protocols usually also differ in their timing requirements, mode of communication and message formats. Hence, a feasible *gateway* device solving the interconnection problem for diverse field level protocols cannot be stateless. As an example consider a *fieldbus A* that transmits a data frame, which contains the current room temperature and a heating setpoint. However, *fieldbus B* interconnecting with *fieldbus A* cannot express both user data variables in a single telegram. Thus, a feasible *gateway* solution must be capable of at least temporary information storage. The information model is intended as such an information storage. Its main purpose is to provide an independent model of the data required for interconnection of two or more field networks (see Section 3.4). It further resembles the *database*, pictured in Figure 2.2.

Further, an additional *protocol binding* component has to be foreseen, which handles the translation from raw field message frames into a format the *information model* is able to understand. Such a *protocol binding* must be specifically designed for each individual field level protocol. Thus, it is best suited as the *presenter* or *session* layer atop of the protocol stack. Hence, it is not viewed as an independent component, but further an integral part of the *protocol stack*.

In order to provide the required grade of reliability, a component which coordinates the

members of a redundant *gateway* compound must be advised. Since the proposed *gateway* approach is stateful, each peer of a redundant *gateway* compound must also replicate its inner state. Therefore, the *redundancy manager* as depicted in Figure 3.6 is foreseen to handle those coordination and synchronization tasks.

Furthermore, the *redundancy manager* component must sense failures of other *gateway* devices part of a logical compound and react accordingly. Hence, the *bus guard* is advised, which targets the monitoring of other *gateway devices*. Another auxiliary component, aiding the *redundancy manager*, is the *bus load monitor*. Given, the synchronization and coordination traffic, necessary in a redundant *gateway* scenario, is directed via the attached field networks and not via a separate backbone link. The *bus load monitor* then provides load metrics to the *redundancy manager*. Such metrics enable the *redundancy manager* component to choose less frequented fieldbus links. Thus, field networks already experiencing peak load situation are relieved and less busy connections are utilized.

A block diagram of the overall proposed *gateway* architecture is depcited by Figure 3.6. It resembles the basic model from the previous chapter (see Figure 2.2). To further illustrate the intended functionality of each characterized building block, a layer diagram according to the ISO/OSI standard model is illustrated by Figure 3.5. The next sections refine the design and concepts of each presented building block.

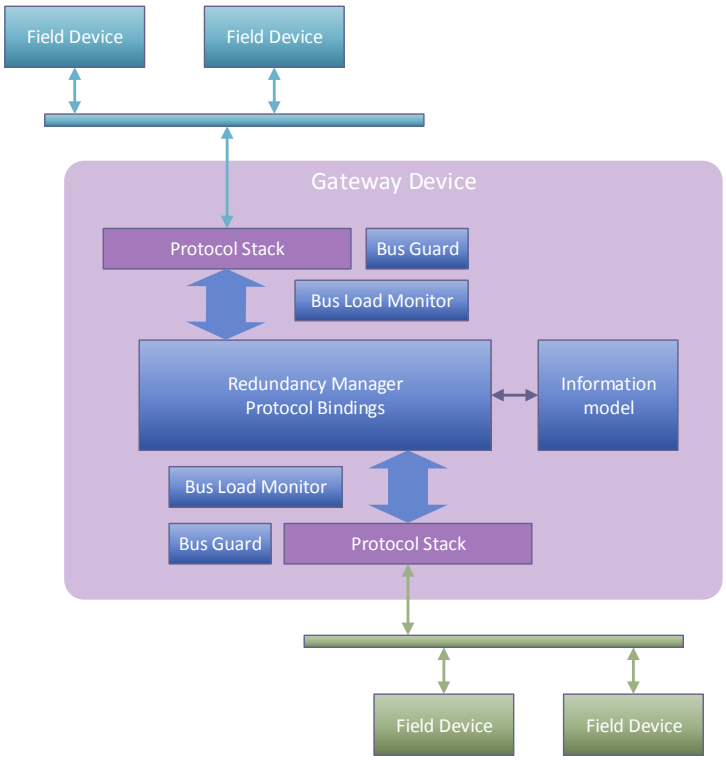

Figure 3.6: Gateway core components

3.4 Information model

The main purpose of a *gateway* device is the translation of messages between all attached field networks. As already outlined in Section 3.1 – Masquerading, fieldbus protocols make use of various different forms of network topologies, information representation and addressing schemes. Hence, a feasible *gateway* solution must provide mechanisms to transform addresses and data between the native representation formats of arbitrary attached field level protocols. Furthermore, different modes of communication like polling or event triggered message transmission also requires additional attention.

As long as all field level protocols considered by a *gateway* device follow the same mode of communication a stateless translation is possible. But even subtle differences in the transmission mode, like diverging message frame intervals of time triggered field protocols [37] or varying telegram content, make stateless communication infeasible. Hence, information exchange at a *gateway* device can only be achieved in a stateful manner for a general applicable solution.

Like for other approaches to the integration problem (see Section 2.2) a model for fieldbus devices, the *information model* needs to be advised. The *information model* is designed to represent logical entities of information in a field protocol independent way and enables the storage of the current state of an attached field network. A model of an arbitrary field network designed by the means of the *information model* must be transformable such that it is accessible for other field networks without additional arrangements or knowledge. Therefore, logical entities existing in such a field network model contain all necessary information to perform exactly like similar, native field devices of another fieldbus attached to the *gateway*.

Concentrating every bit of information in a single entity at the *gateway* tends to become cumbersome and inflexible. As an example, a virtual field device entity needs to store address information for every field protocol supported by the *gateway*. Hence, only a subset of the information modeled is needed for a single supported fieldbus, whereas the rest is unused. Furthermore, one classification into logical entities may be suitable for a single fieldbus, but may be infeasible for another one. Hence a separation of concerns [52] between the raw user data and the protocol specific data needs to be realized.

A possible solution is the deployment and adaption of the Model View Presenter (MVP) software architectural design pattern [49]. The MVP pattern originates from the more popular Model View Controller (MVC) design pattern [49] and is usually used for building and designing user interfaces. As depicted by Figure 3.7, the MVP pattern consists of three interconnected components. The *model* defines the available user data and offers information update mechanisms and state-change notification services. The *view* is the front-end presented to the user and is responsible for forwarding user commands to the *presenter*. The *presenter* acts as *glue*, interconnecting *model* and *view* and is responsible for the formatting and filtering of information.

Applying the MVP design pattern to the information model of the proposed *gateway* approach, the model is partitioned into a general segment and a field protocol specific segment. The general part resembles the *model* introduced by the MVP design pattern and stores user data. On the other hand, the protocol specific part, resembling the *presenter* and describes information only relevant to a single protocol. Therefore, a *gateway* device information model contains one *model* and *presenters* for each attached field protocol. Finally, there is the *view*

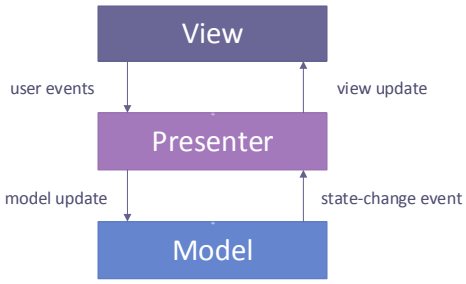

Figure 3.7: MVP design pattern [49]

component from the MVP pattern, which defines a unique access point of the the *gateway* devices field protocol stacks.

Model

As already suggested, a *model* represents the user data of arbitrary field devices, which are members of field networks attached to a *gateway* device. Considering a simple temperature sensor, possible user data is a floating point decimal expressing the current ambient temperature, whereas device addresses and other protocol specific information modeled at the *presenter*.

A feasible meta-model framework describing the *model* must provide a the necessary means and flexibility to model even complex user data structures. On the other hand, it should be as simple and fast as possible to take resource constraints inherent to embedded devices into account. When considering a design for the meta-model, it is beneficial to refer to existing abstractions due to the vast amount of available field protocols. A good orientation about needed primitives and complex data types needed, is given by OPC UA (see Section 2.3 – OPC UA), which itself is designed to be an abstraction layer above multiple fieldbusses. However, only a narrow subset of the available OPC UA features is needed for the *gateway* approach. Hence, a stripped down version of the OPC UA information meta-model is advised.

At first, a *BaseNode* entity type is defined which is the fundamental entity of the proposed type hierarchy. Every, from *BaseNode* derived element can be directly addressed using the *NodeID* attribute. The value of the *NodeID* must hence be unique throughout the whole gateway device. *Attributes* are simple key-value pairs that model essential information regarding other elements of the hierarchy. Further, *Attributes* cannot be directly addressed and always belong to a *BaseNode* derived type. An *Attribute* consists of two elements, a *Name* and a *Value*. The *Name* identifies an *Attribute* uniquely in the context of an arbitrary *BaseNode* derived entity type. Thus, no two *Attributes* with the same *Name* are allowed to exist throughout a derivation chain. The *Value* part of an *Attribute* stores its current state and must be encoded according to one of the defined *ValueTypes* (See Table 3.1). The relationship from an *Attribute* to a *ValueType* is modeled as a directed *reference* with a $1 : *$ cardinality. Thus, an *Attribute* can be part of exactly one *ValueType*.

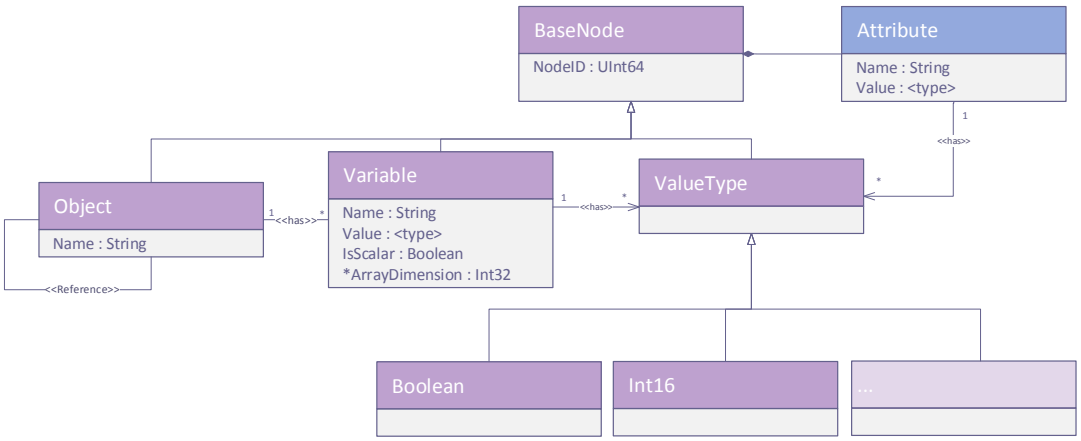

Figure 3.8: Meta-model for *model*

A *Variable* entity type resembles arbitrary data values, and is similar to the concept of a variable in an arbitrary programming language. Its set of *attributes* consists of the mandatory properties *Name*, *Value*, *IsScalar* and an optional *ArrayDimension* (See Figure 3.8). *Variable* types are intended to be put in the context of an *Object* entity type. But also *global Variables* are allowed to exist, meaning that no *Object* has a *reference* to the prior mentioned *Variable*. Furthermore, a *Variable* with a reference to an Object may be addressed by its *Name* attribute or the *NodeID* unique in a *gateway* device, whereas a *global Variable* can only be accessed via the *NodeID*. The modeled *Value* of a *Variable* is either a scalar or array type, determined by the *IsScalar* attribute. In case the *IsScalar* property is set to *FALSE*, the *ArrayDimension* attribute has to be present. The *ArrayDimension* defines the fixed number of different values this *Variable* may hold.

The purpose of an *Object* element is to group contextual related *Variable* types in a single entity and to allow the modeling of deeply nested data structures. An *Object* is allowed to have an arbitrary number of *references* to other *Objects* or *Variables*. The only mandatory *Attribute* for an *Object* is the *Name*, so it may be identified in the context of another *Object* without knowing its *gateway* wide unique *NodeID*.

The *VariableType* element represents the supported data encoding formats of the *Value* entries for both *Attribute* and *Variable* entity types. There is an exhaustive amount of commonly used data encodings predefined in Table 3.1. An extension of the meta-model with custom built *VariableTypes* is not yet foreseen, but part of the considerations for future work (see Chapter 6).

In comparison to OPC UA, most of the mandatory *attributes* of a *BaseNode* have been discarded in favor of a lean meta-model. The only remaining mandatory *Attribute* of a *BaseNode* is the *NodeID*. The *NodeID* property is foreseen as a 64-bit wide integer value, which limits the maximal number of available elements to 2^{64}. The available *NodeID* address space should suffice to model even huge systems.

Value Type	Description
Boolean	represents a boolean variable, which can either assume the value *true* or *false*
Byte	unsigned single byte value with 8-bit width
ByteString	32-bit wide length plus additional number of length bytes for arbitrary storage
DateTime	UTC time format encoded time stamp
Double	IEEE double precision 64-bit floating point number
Float	IEEE single precision 32-bit floating point number
Int16	signed integer with 16-bit width
Int32	signed integer with 32-bit width
Int64	signed integer with 64-bit width
String	represents a string of unicode characters
UInt16	unsigned integer with 16-bit width
UInt32	unsigned integer with 32-bit width
UInt64	unsigned integer with 64-bit width

Table 3.1: Basic value types

Also the complex *reference hierarchy* present in OPC UA has been abandoned (see Chapter 2). Only a single reference type is foreseen, with no additional semantic, apart from *direction* and *endpoint cardinality*. Thus, a flat hierarchy is encouraged, which should benefit resource constrained devices in both storage space and reduced navigation times throughout the *model*. The different sized integer and unsigned integer types are utilized. Although a single integer type with 64-bit width would suffice, the more complex arrangement was chosen to again allow modeling for resource constrained devices.

Presenter

The purpose of the *presenter* is the representation of the various *virtual field devices* to an individual field network. Thus, the *presenter* serves as a *glue logic* between the *model* and an arbitrary field level protocol. Further, it allows the modeling of fieldbus devices specific to a single field level protocol without inducing additional information into the *model*.

As depicted by Figure 3.9, the *presenter* part of the information model utilizes the same *BaseNode*, *Variable*, *ValueType* and *Attribute* entity type as the *model*. The *NodeID* attribute of the *BaseNode* must be again unique throughout the whole gateway device. Furthermore, this effectively means that the *model* and the *presenter* entity types share the same *NodeID* address space. On the positive side the binding between the *model* and *presenter* is simplified by unified addressing mechanisms and a common system of value types. This even allows the combination of *model* and *presenter*. On the other hand, there is no sharp distinction between the *model* and the *presenter*.

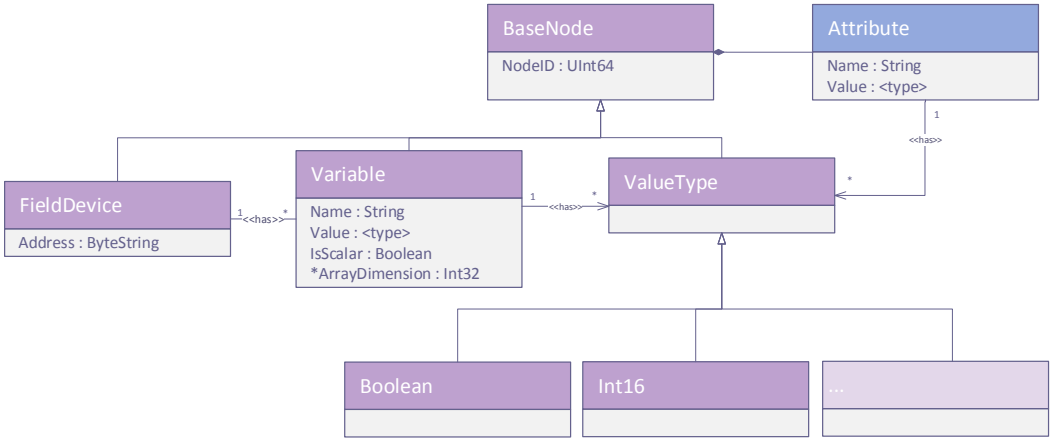

Figure 3.9: Meta-model for *presenter*

The new entity type, introduced especially for the *presenter* part of the information model, is called *FieldDevice*. It is designed to enable the modeling of virtual field devices presented to arbitrary *gateway* attached busses. Such a modeled virtual field device is exclusive to one field protocol and cannot be shared among different *views*. The only mandatory attribute, *Address*, represents the native address for a single field level protocol. It is encoded in the targeted field protocols native encoding and is stored as an arbitrary *ByteString*. Additional Information, like special status flags or device class indication, can be linked to *FieldDevices* in form of *Variable* entities. Further, *FieldDevices* make user data available to the attached field network, by binding to *Objects* and *Variables* from the *model* via their respective *NodeID*.

There is no official equivalent model or binding mechanism defined in OPC UA. However, *OPC UA information models* for specific field level protocols are discussed in several papers, like [54]. Such documents usually specify a binding layer, which complements the *OPC UA information model*, and is exclusively designed to interact with a single field level protocol. Hence, such an OPC UA binding mechanism, is able to provide a more detailed modeling abstraction than the proposed *presenter* model. Hence, Comparison between a specific OPC UA binding layer and the proposed *presenter* model is meaningless, since the proposed *presenter* model is designed to be as general as possible.

View

The *view* part of the proposed information model resembles a presentation layer for a concrete field network protocol stack. The *View* entity type depicted in Figure 3.10 serves as a single entry point for a protocol stack, and therefore has references to all *FieldDevice* entities designated to a certain field network. Hence, an arbitrary protocol stack bound to a *View* entity must be able to parse and interpret the data stored in the information model. Furthermore, services handling data updates and notifications on changing data are provided (see Section 3.4 – Information Model

Transactions).

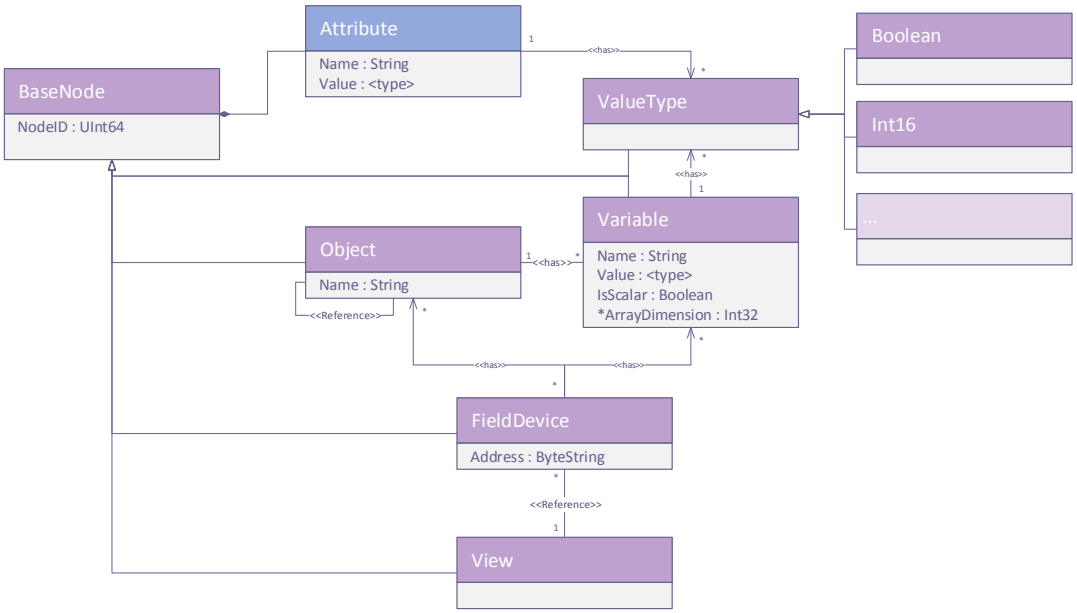

Figure 3.10: Overall meta-model

Figure 3.10 depicts the overall meta-model describing the proposed information model, while Figure 3.11 shows an example of a concrete information model connecting two different field networks. Virtual field devices, accessible through *View FieldBus A* are visible to one network, while virtual field devices connected to *View Fieldbus B* are seen by *View FieldBus B*. Information is then passed via the respective virtual field devices to the *Variables* and *Objects*.

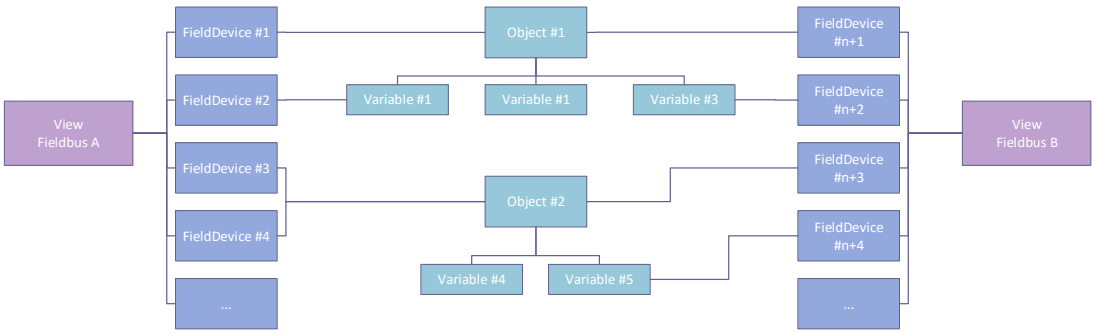

Figure 3.11: Example information model

Information Model Transactions

Three distinguished transactions are foreseen to interact with the information model: an *update* transaction, a *poll* transaction and a *notify* transaction. Each described transaction is self-contained and either succeeds as a whole or needs to be re-initiated. The information model does not define any fault recovery mechanisms. Hence, a failed transaction must be triggered again by its respective caller.

Update

The *update* transaction, as depicted by Figure 3.12, is triggered by an incoming message at the *protocol stack*. The message is then interpreted by the *protocol stack*. At first, all available virtual field devices at the *gateway* are enumerated in order to find the responsible *FieldDevice* entity. Therefore, an *enumerate_field_device* service message call is issued to the *View* linked to the *protocol stack*. As shown in Figure 3.12, the *View* needs to create a list of all the virtual device addresses from each associated *FieldBus* entity. Thus, each *FieldDevice* entity associated to the respective View has been queried for its mandatory *address* attribute. Further, the *address* and the *FieldDevice* node identifier are combined in form of a map data structure, where the *address* serves as a key and the node identifier *NodeID* as the value.

The prior mentioned map, consisting of the native field device addresses and the *NodeIDs* is transferred to the *protocol stack* as a reply to the *enumerate_field_devices* service call. The *protocol stack* compares the destination address from the received message to the individual device address from each virtual field device *address* attribute received in the map. Upon finding a matching address, the native fieldbus message payload is decoded. Assuming, that the payload decodes to a write call to the virtual field device, an *update_field_device* service message is transmitted to the concerned *FieldDevice* entity.

Next, the destination *Variable*, respectively *Object* has to be found. Hence, the *protocol stack* must derive the proper naming convention from a possible *device class* defined by the field protocol or from information stored in the received fieldbus message. Determining the right *Variable* or *Object* target to update, given a received message can only be done without ambiguity if the concrete model is properly designed. Hence, in conjunction with individual protocol stacks, a proper naming convention for references at the *presenter* model has to be advised. The concerned *FieldDevice* entities then need to implement the convention or design rule. This illustrates yet again the necessity of a layered information model and emphasizes the role of the *presenter* layer as independent binding mechanism.

After the correct *Object* or *Variable* entity has been found, an *update_object* and consecutive an *update_variable* service message call is issued. The *protocol stack* is responsible to transfer the new data value in the encoding of the destined *Variable* entity (see Section 3.1). Therefore, a valid *protocol stack* implementation provides all possible encoding conversions services between the native format of the represented fieldbus and the formats defined by the *ValueType* entity derived types.

Finally, the transaction is closed, by returning an *update_done* reply message from the *Object* to the *FieldDevice* and consecutively to the *protocol stack*. Further a *data_change_notification*

is issued to notify other *FieldDevice* entities and furthermore other *protocol stack* implementations about changed data values (See section 3.4 – Notify).

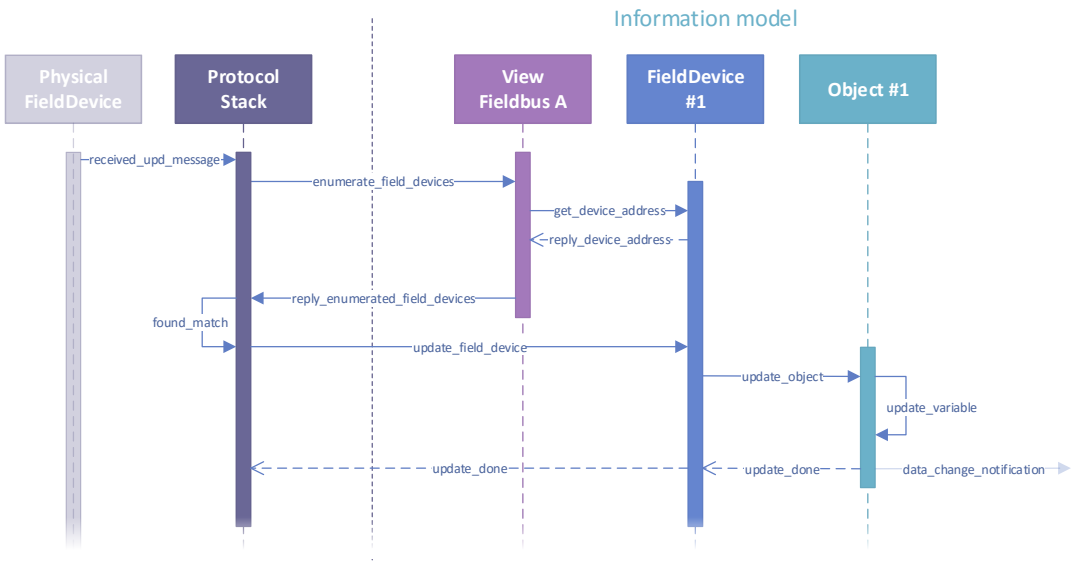

Figure 3.12: Information model update transaction

Poll

The *poll* transaction, as shown in Figure 3.13, is especially intended for field level protocols which are based on a *poll mode* of communication. The transaction is triggered by an arbitrary fieldbus message at the *protocol stack* which requests data from a *virtual field device* stored at the information model.

In order to identify the *virtual field device*, an *enumerate_field_device* service message call is issued to the *View* entity. This is in fact the very same procedure as already described in Section 3.4 – Update. Again, after the *View* has successfully queried all associated *FieldDevice* entities for their *address* attribute, a map data structure is returned. The map is designed as such that the *key* is the fieldbus native device address and the *value* contains the *NodeID* attribute.

After a matching *FieldDevice* has been found, by comparison of the destination address, part of the field message received by *protocol stack* and the *address* attribute from the map, a *poll_field_device* service message call is issued. Again the same thoughts about design rules that enable the linkage of the *presenter* layer to the *protocol stack* apply as discussed in the previous Section 3.4 – Update. The concerned *FieldDevice* entity will again query the referenced *Object* entity and further the concerned *Variable* entity. Finally, the requested data value is returned by the means of a *poll_reply* service message. The protocol stack is then responsible to encode the received data value in the native format of the associated fieldbus. Further, a response message

to the original request is created and a reply on behalf of the *virtual field device* is sent out on the field network. This successfully completes the transaction.

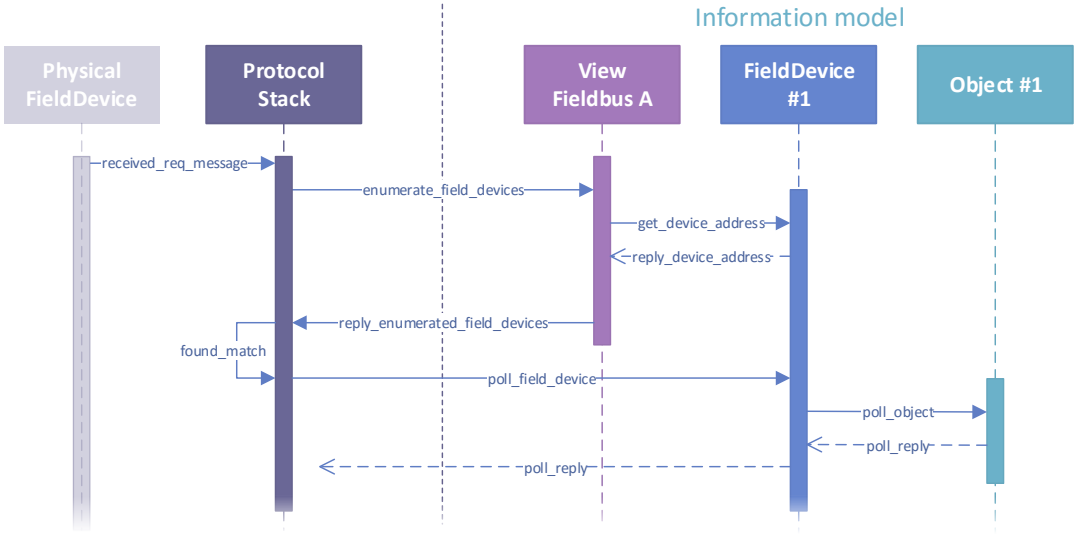

Figure 3.13: Information model poll transaction

Notify

The *notify* transaction, as depicted in Figure 3.14, is triggered by a successful *update* transaction (see Section 3.4 – Update). Its main purpose is to inform other *FieldDevice* entities about changed data values at their associated *Object* and *Variable* entities. Further, the respective *View* entities are notified and finally the concerned protocol stacks. Thus, the *notify* transaction is especially useful for field protocols utilizing an event based mode of communication.

A *data_changed_notification* caused by a successful *update* transaction, triggers the *notify* transaction. At first, all *FieldDevice* entities linked to the affected *Object* or *Variable* relay the notification to all linked *View* elements. Further, the *View* entities call the *protocol stack* with a *notify_stack* service message call. The information enclosed by this message contains the *FieldDevice* reference, the affected *Object* or *Variable* reference.

In case an element of the attached fieldbus serviced by such a notified *protocol stack* requires a report of the changed data value, a native fieldbus message is generated. The transaction is successfully completed upon notification of the *protocol stack*.

3.5 Reliability

Another important building block regarding the design of a feasible *gateway* approach is the linkage of several individual *gateway* devices to one logical compound. Such a *gateway* compound is further called VRG. The VRG concept and design is heavily influenced by the Common

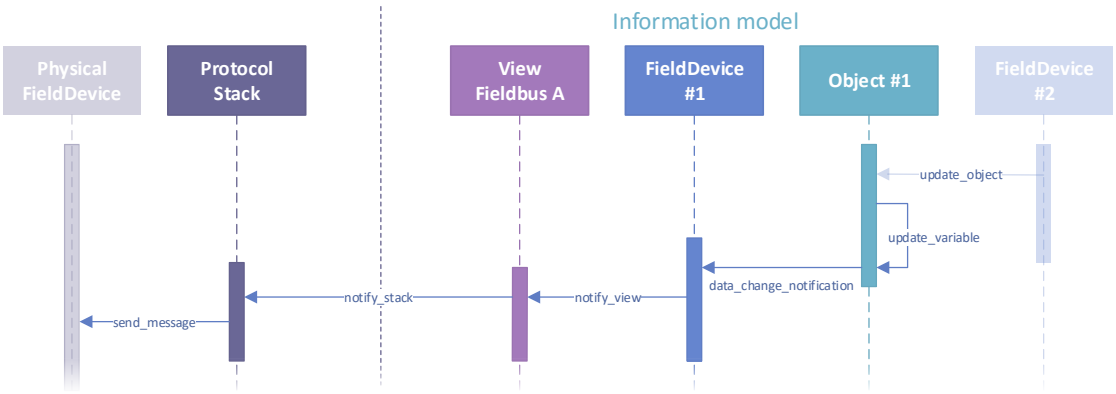

Figure 3.14: Information model notify transaction

Address Redundancy Protocol (CARP) [21] approach and its proprietary predecessor VRRP, originally intended for routers and firewalls of Ethernet and IP based ICT systems.

The reliability concept of the proposed *gateway* approach is based around a feasible *fault hypothesis*. The *fault hypothesis* is the number and type of faults that a system is able to tolerate [35]. As already mentioned in several previous sections, the fault hypothesis of the VRG should be at least as strong as the weakest fault hypothesis among the attached fieldbusses. As depicted in Figure 3.15, several areas can be defined where individual *fault hypothesis's* hold. Faults that occur in such an area are contained within the previously mentioned area, unless the *fault hypothesis* is violated. Hence, the areas shown in Figure 3.15 form fault-containment regions (FCRs) [35]. For an interconnected system of FCRs, the FCR with the least amount of tolerable failures dictates the *fault hypothesis* for the whole system, since it is the most likely subsystem where the *fault hypothesis* might be violated. It is required, that a VRG is by no means the FCR with the weakest *hypothesis* alone. Hence, the *fault hypothesis* of a VRG must be at least as strong as a *fault hypothesis* of one of the attached fieldbusses.

The implementation of the *reliability* concept is that a designated *gateway* device, member of a VRG, handles all the work, while the other members are in a *hot-standby* mode. The designated *gateway* device is called the *active gateway*, whereas the others are referred to as *inactive* or *standby gateways*. Since *gateway* devices are stateful, apart from *coordination* tasks, a *synchronization* mechanisms needs to be advised. As already mentioned throughout Section 3.2, there are two viable approaches interconnecting single *gateway* devices to a VRG.

The first option is a separate backbone link, completely disjoint from the tethered fieldbus networks of an arbitrary *gateway* device. One advantage of such an approach is the complete separation of the synchronization and coordination traffic on the hand and normal fieldbus operation traffic on the other hand. As a disadvantage, additional wiring effort is needed. Also an additional *protocol stack* might be needed for a *gateway* implementation, unless the *backbone* network utilizes a protocol stack of one of the *gateways* already implemented field level protocols.

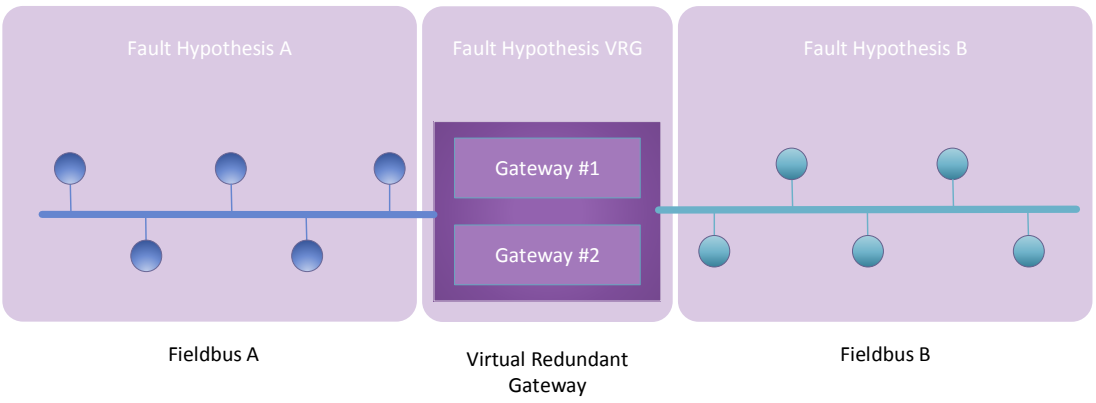

Figure 3.15: FCRs of an interconnected system

The second option utilizes the available fieldbusses already attached to a *gateway* device for coordination and synchronization traffic. As a benefit, no additional wiring effort is needed and existing *protocl stacks* readily available at the *gateway* are utilized. On the other hand, the traffic overhead induced by the *gateway* devices may interfere with normal fieldbus operation. This is especially true, when field level protocols with low data rate are interconnected by the *gateway* devices. An improvement to a high load situation at one particular fieldbus, a *bus load monitor* is introduced in Section 3.5 – Bus load monitor. The *bus load monitor* aids the *gateway*, so that coordination and synchronization traffic is redirected via less frequented fieldbusses. Another detail is, that a single *gateway* device itself needs to be assigned a native fieldbus address for every attached fieldbus. Whereas with a backbone network, the *gateway* device does not need to be addressed directly.

The second option is chosen, since the additional wiring effort inherent to the first proposed option is opposing the *seamless integration* requirement stated in Section 3.1 – Seamless integration. The synchronization and coordination tasks of a single *gateway* part of a VRG are handled by three main building blocks: The *bus guard*, the *bus load monitor* and the *redundancy manager*.

Bus guard

The *bus guard* component is responsible for the detection and reporting of link faults at an arbitrary fieldbus interface of a *gateway* device. Therefore, an *end-to-end connection monitoring* is deployed between *active* and *inactive* members of a VRG. Also mechanisms specific to individual field protocols, which are in general not available are utilized. An example of such a fieldbus feature is *carrier sense*, which monitors if a base bus voltage level is present at the physical interface, and hence detects a broken cable by the absence of such a voltage level.

The *end-to-end connection monitoring* is realized by sending a *heart-beat* message between the *active* and *inactive gateway*. The *inactive gateway* is always the initiator of a *heart-beat* transaction, whereas the *active gateway* solely acknowledges such incoming messages. The

transaction is successfully finished, as soon as the initiator of the *heart-beat* message receives a valid acknowledge telegram.

The *heart-beat* transaction is repeated every *HeartBeatInterval* seconds. In case of a failed *heart-beat* transaction, a *grace counter* is foreseen, which is incremented for each subsequently failed *heart-beat* transaction. Thus, the notification of the *redundancy manager* is delayed until an upper limit, named *GraceLimit*, is hit by the *grace counter*. In case a successful *heart-beat* transaction is encountered after a previously failed *heart-beat* transaction, the *grace counter* is reset.

$$GracePeriod = GraceLimit \cdot HeartBeatInterval \tag{3.1}$$

The time span until a connection failure is signaled by the *bus guard*, the *GracePeriod* is calculated by the Equation 3.1. In the event, that the *GracePeriod* is exceeded without interruption of a successful *heart-beat* transaction, the redundancy manager is notified and a *fail-over* procedure is triggered (see Section 3.5 – Redundancy manager).

The prior stated *end-to-end connection monitoring* method reliably detects link faults between two different *gateway* devices. However, it is not capable of making a statement which of the involved fieldbus interfaces is faulty. It is even possible that both interfaces work perfectly and the fault is located somewhere between two routing nodes of the interconnecting field network. Hence, a method needs to be advised that narrows the location of a link fault down to a certain fieldbus interface.

The most simple solution to this issue, is the usage of a third *gateway* device approving a failed *heart-beat* transaction. Like in a *triple modular redundant system* [24], the failed *heart-beat* transaction has to be approved by the third *gateway*. Otherwise, the reporting *gateway* is marked as faulty and subsequently removed from the VRG. However, this approach has not been further considered since it would increase the economical costs and complexity of a VRG, but not add any additional value compared to the *ping* solution presented next.

A better approach to verify an arbitrary fieldbus interface of a *gateway* device is a third party communication partner also connected to the network interface under test. A service similar to the *ping* command used in Ethernet and IP is advised, which verifies correct communication between the *gateway* and another fieldbus participant. The *ping* service itself can be implemented by an arbitrary message with a defined payload, which gets acknowledged by a communication partner upon successful retrieval. The only constraint is, that the message sent out by the *ping* service must not modify the state of the receiver, or trigger any action that will affect other fieldbus participants.

In case of field level protocols which do not leverage any fault-tolerance mechanisms, a single arbitrary fieldbus node is fit to serve as *ping* reference node. As soon as the attached fieldbus implements fault-tolerance, a reference node according to the *fault-hypothesis* of the network has to be found. Therefore, the concept of a virtual reference node is proposed. A virtual reference node is a compound of otherwise disjoint bus participants. The number of virtual reference node members is solely determined by the *fault hypothesis* of the fieldbus. Hence, if it is stated that the field network is able to tolerate up to one device or link faults, then the virtual reference node consists of two bus participants. Prior to the activation of a field

network, an order has to be defined for the members of a virtual reference node at the VRG. Thus, if a fault occurs, the members of a virtual reference node are contacted by the means of the *ping* service in the previously given order from the members of the VRG. This ensures a consistent result of the *ping* service as long as the fieldbus itself operates according to its *fault hypothesis*.

Bus load monitor

The *bus load monitor* component passively listens on a fieldbus and provides the *redundancy manager* with load metrics regarding network operation. Such load metrics are then used to choose the least frequented fieldbus link to transmit synchronization data frames. Otherwise, additional synchronization traffic induced by the individual *gateway* devices may put a fieldbus in an overload situation, which is undesirable for both the *gateway* and normal bus operation.

The network load is defined as the overall amount of data injected by the fieldbus participants into the network, divided by the maximum amount of data which can be transacted [30]. An overload situation is encountered when the ratio of the injected data amount and maximum possible exchange rate of a network segment approaches 1.0. This usually results in higher message jitter, increased packet latency and even lost telegrams.

Not all discussed fieldbus topologies from Section 3.2 are equally vulnerable to network overload situations or traffic congestion of isolated network segments. The *star topology* (see Section 3.2 – Star) deploys exclusive links between each communication partner. Hence, an overload situation is contained within a network segment, connecting an exposed fieldbus node with the central node. The *ring topology* (see Section 3.2 – Ring) and *line topology* (see Section 3.2 – Line) however are prone to such traffic congestion, which then may render the whole fieldbus unusable. Therefore, a *tree topology* is more desirable, which partitions a single fieldbus *line* into different hierarchical level of *lines* and is able to contains an overload situation in one such network segment. A *mesh topology* is most fit to tolerate overload situations, since there is usually more than one communication path available, which connect different nodes. But a single network segment, interconnecting otherwise disjoint partitions of a network, experiencing a traffic congestion will cause the *mesh network* to disintegrate into two separate areas.

A possible implementation of a *bus load monitor* has several different options to measure the *load* of a network. In case of a *line* or *ring* topology, an eaves-dropping interface is able to sample the number of telegrams per an arbitrary time unit. However, in a partitioned network like a *tree topology* or multi-path network like a *mesh topology*, this would be rather inaccurate. A mechanism, sampling only messages passing the network interface at the *gateway*, measures only the load regarding the local *line* segment. Hence, it is not capable to indicate the traffic load situation at another partition of the same field network.

Another possible load metric indicator, is the measurement of a time-stamped *bus guard* message. The round-trip delay of such a telegram is then a possible indicator of the network load along its communication path. Such a load metric mechanism effectively measures the time it takes to grab a free communication time slot at the fieldbus for data transmission and incorporates delays due to message queuing at forwarding nodes. As stated in [27], such an indicator is only valid in networks which are not delay-tolerant. However, round-trip delay measurement for *gateway devices* is a valid load metric, taking into consideration that it is designed

to operate on field level protocols which are most likely real-time capable and not delay-tolerant. As a disadvantage, at least one sample round-trip measurement at a fieldbus idle situation must be taken as a comparison reference for future measurements.

Other metrics available to the *bus load monitor* are dependent on the actual implementation of individual field protocols, like connection quality and signal strength of a wireless field protocol. An ideal implementation of a *bus load monitor* regarding a certain field level protocol would consider all applicable metrics and combine them in a single numerical *score* value. Utterly important, a *score* of one fieldbus must be comparable to the *score* of another fieldbus, despite different metrics used for each individual network. Therefore, the *score* relies mostly on the two general-applicable indicators described previously. Other metrics may be additionally implemented, but are only intended to marginally alter the *score* value.

Redundancy manager

The *redundancy manager* is responsible for both synchronization and coordination of individual *gateways*. As long as no separate backbone link is used, the *redundancy manager* must be assigned an individual address for each attached field network. Otherwise, it would not be possible to directly address and send synchronization and coordination telegrams to each *gateway* device. Furthermore, every attached field level protocol must allow some field message type which carries an arbitrary payload or at least tolerate messages with an arbitrary payload. The worst case would be a field protocol, that does not only forbid, but actively opposes such messages. This could be done by deep packet inspection and further dumping of such packets at mediator fieldbus node along a communication path between two *gateway* devices. Fortunately, there is no known field protocol which deploys such rigorous restrictions. Otherwise, the only solution is the deployment of a separate backbone link between the members of a VRG.

As already indicated, a *synchronization message* is triggered by the *data_change_ notification* emitted by an *Object* or *Variable* from the *information model*. The updated data from the *Variable*, including its *NodeID* identifier are packed into a *synchronization frame*. Once an attached field network has been selected by evaluating recent metrics provided by the *bus load monitors*, a native telegram is generated. The native message contains the *synchronization frame* as payload and is sent out via the interface to the chosen fieldbus. The receiving *gateway device* simply recognizes the arriving message as synchronization or coordination message, because it is addressed directly to its individual field protocol address. Hence, the payload is extracted from the native message and its content is interpreted accordingly. The unpacked frame is perceived as *synchronization frame* by evaluating a message type data field. Further, the received *NodeID* attribute identifies the *Object* or *Variable* and its content is updated with the received data by an *information model* update transaction.

In a compound of redundant *gateway* devices, one unique *gateway* is *active* and assumes the role of a *master*. All other gateways part of a compound are *inactive*. Hence, the *master gateway* is responsible for the synchronization effort, while the other *gateways* are monitoring the *active* device. The designated member of a VRG, which assumes the role of the *master gateway* is determined by an election algorithm. The algorithm is designed in such a way, that each peer *gateway* of a VRG reaches the same conclusion with a minimal amount of coordination traffic. This is especially important in a fail-over scenario, where a failed *active gateway* must be

replaced by one from the *inactive gateway* pool. A minimal traffic amount results in the shortest possible time, where communication between attached fieldbusses is not possible.

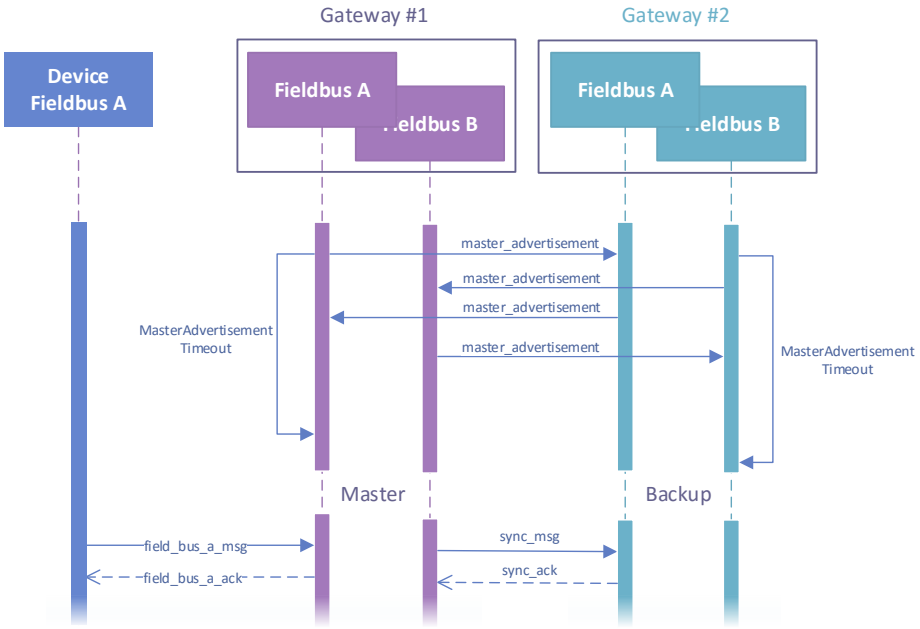

Figure 3.16: Election algorithm

A minimal implementation of such an election is based on an arbitrary fixed order between every member of a VRG. Such an order is achieved by assigning a fixed unique number to every *gateway* within a redundant compound, like a *gateway* identifier or priority value. When the election process is triggered, every functioning *gateway*, member of the VRG, participates by sending out its own *master* priority value via multicast messages. Such a message is called *master_advertisement* message. After a certain amount of time called *MasterAdvertisement-Timeout*, no new priority values are accepted. Each *gateway device* participating in the election then chooses the *gateway device* with the highest priority value as the new *master*.

The prior paragraph already outlines the main prerequisite for a reliable and consistent election mechanism, a *reliable multicast*. Meaning, that either a *master_advertisement* message is delivered to every participating, fully functional *gateway device* or to none. Otherwise, the list of priority values may differ from *gateway* to *gateway* and the election process is inconsistent.

The election process itself is triggered either by the *bus guard* and subsequently the *redundancy manager*, a received *master_advertisement* message or in case the VRG does not have a current *master* set. When a *master gateway* itself receives a *master_advertisement* message, it assumes that at least one communication link does not work. Thus, the failed *master* assumes the role of a *fail-silent* device and does not participate in any communication.

The election algorithm advised is rather simple and more sophisticated approaches are part of future work that is considered. As an example, a possible item for improvement is the recovery or join of a previously failed *gateway*, which has not been addressed. Another possibility for an improvement is the fault handling during either synchronization or *information model* transactions.

CHAPTER 4

Fault Analysis

The fault analysis of a *gateway* compound is carried out by evaluating a scenario consisting of two individual *gateways* without loss of generality. Such a VRG is capable of tolerating one link fault at an arbitrary network interface of a *gateway* and one *gateway* device fault. The fault analysis of the proposed *gateway* approach proceeds by a case distinction on all the different situations a fault may occur. The following cases are considered:

- **Link fault at *master gateway*.** Given a link fault at an arbitrary field device interface of the current *active gateway*, the monitoring *inactive gateways* will recognize the failure through their respective *bus guard* components. *Heart-beat* messages remain unacknowledged and the *GracePeriod* is started at the monitoring devices. The time span before the current *master gateway* is considered failed is, as stated in Section 3.5 – Bus guard, the *GracePeriod* plus a possible jitter of one *Heartbeat* interval. After the expiration of the *GracePeriod*, the *inactive gateway* performs a *self-test* using the *ping* service on its interfaces to avoid erroneous triggering of the election process. Therefore, a *backup gateway* reaches out to a virtual reference node via each fieldbus interface. In case such *ping* messages are acknowledged on every available field network interface, the *inactive gateway* starts to multicast the *master_advertisement* messages simultaneously on all available interfaces. The *master gateway* also senses the *master_advertisement* messages, since one fieldbus link is still operational. As an effect, the failed *master gateway* shuts down any operation and removes itself from the VRG. The *backup gateways* wait for *MasterAdvertisementTimeout* time for incoming *master_advertisement_messages*. After the timeout has expired, the *inactive gateway* with the highest priority value is selected among all *gateway* devices that participated in the master advertisement. Thus, the highest priority *gateway* assumes the role of the new *master* device and continues normal operation. All remaining *inactive gateways*, part of the VRG compound start to monitor the new *master*.

- **Device fault at *master gateway*.** Assume a device fault at the current *master gateway*. Hence the current *active gateway* ceases to respond on all its fieldbus interfaces. The

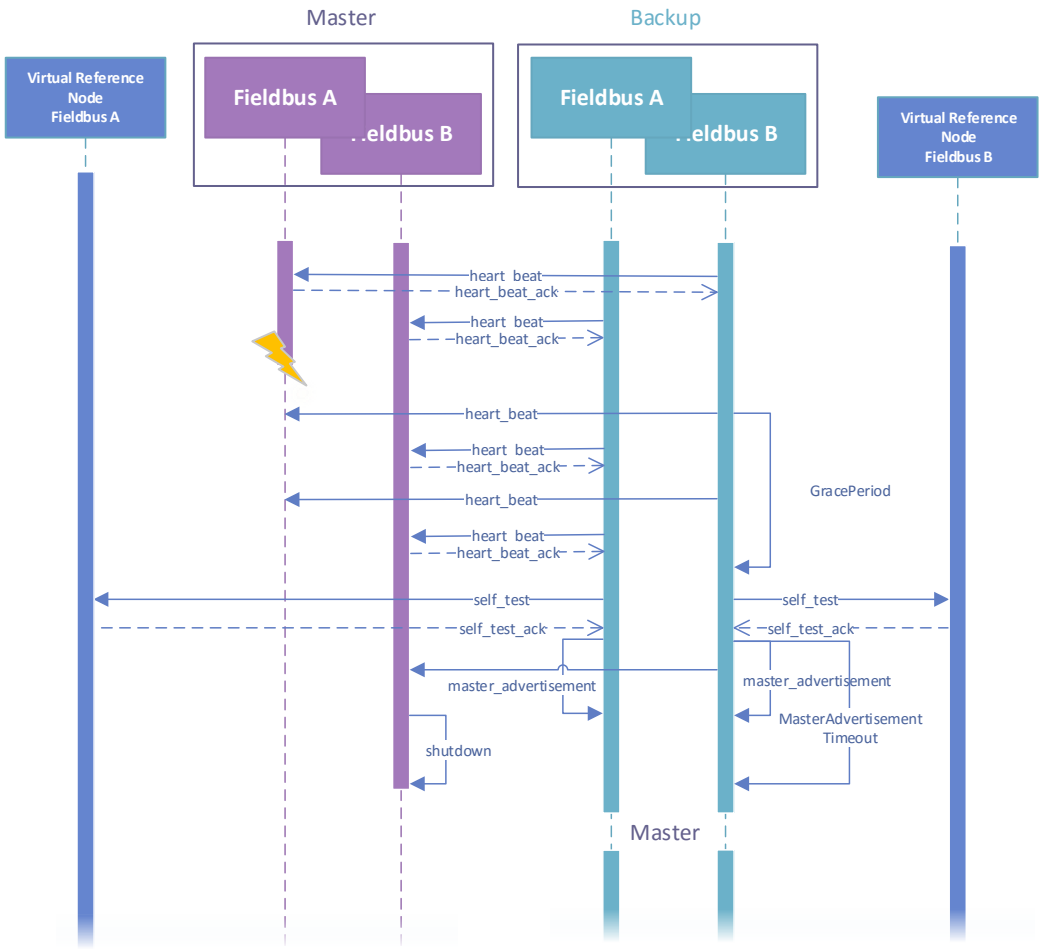

Figure 4.1: Link fault at *master gateway*

monitoring *backup gateways* again recognize the outage of the *master* via their respective *bus guard* components. *Heart-beat* messages remain unacknowledged. After the expiration of the *GracePeriod* plus a possible jitter of one *HeartBeat* interval the *inactive gateways* start a *self-test* of its fieldbus interfaces using the *ping service*. Again, a *backup device* reaches out to the virtual reference node defined for each single operational field network interface. After the *self-test* is successfully completed, the *backup* devices start to send out *master_advertisement_messages* on all available field network interfaces, utilizing reliable multicasts. Different to the previous *link fault at master gateway* case, the current *master* device, experiencing a fault is already *fail-silent* and removed from the VRG. Thus, it cannot receive or even respond to a *master_advertisement_message*. But, since the failed device emits a *fail-silent* behavior in the first place, it already stopped

interacting with the remaining *gateways* and attached *field networks*. After the *MasterAdvertisementTimeout* has expired, all *backup gateways* choose the new *master* by comparing the received priority values from the advertisement phase. The device with the highest priority value is selected as the new *master gateway* and continues normal operation. All remaining functional *inactive gateways* part of the virtual *gateway* compound switch their *bus guard* monitoring to the new *master* device.

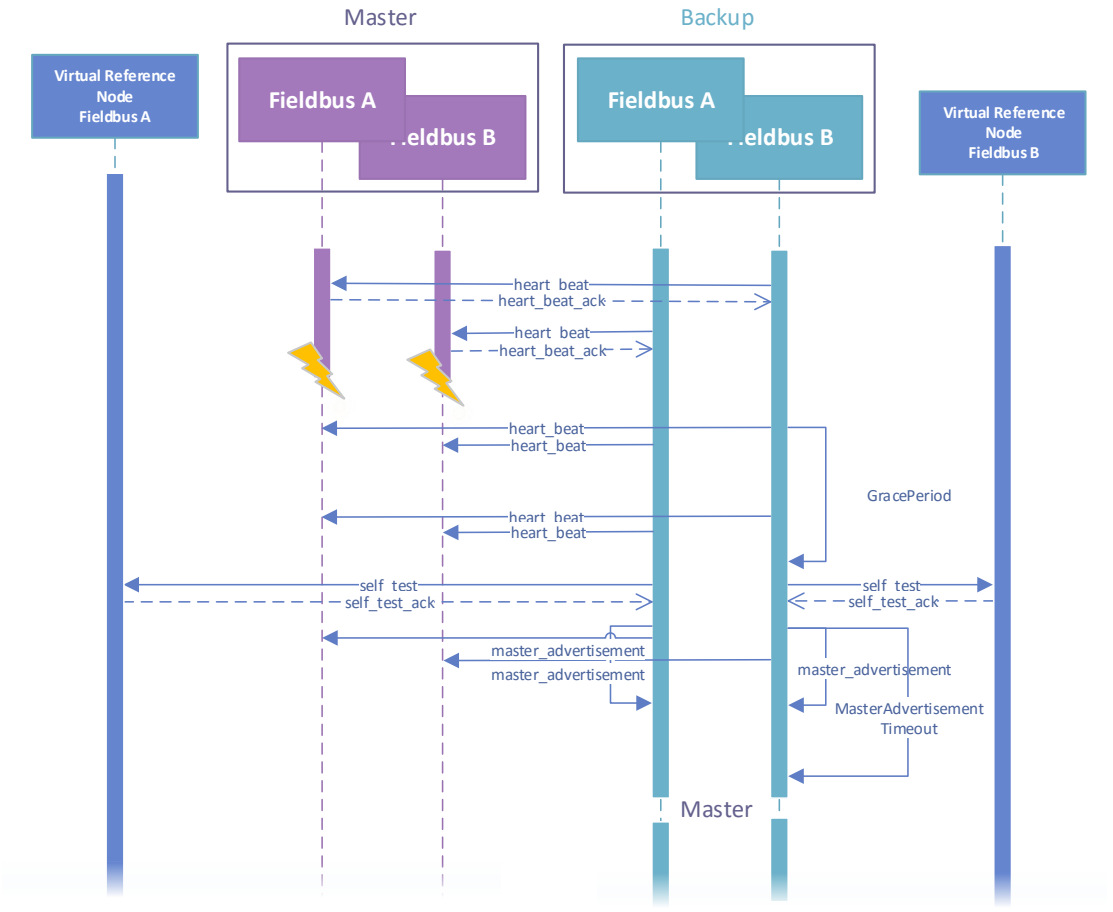

Figure 4.2: Device fault at *master gateway*

- **Link fault at *backup gateway*.** Suppose a link fault at an arbitrary *backup gateway* and an arbitrary field device interface thereof. The affected *inactive gateway* is then unable to send out *heart-beat* messages or receive synchronization telegrams at the failed interface. Hence, the *bus guard* of the failed interface starts the *GracePeriod*. After the *GracePeriod* plus a jitter of one *HeartBeat* interval has expired, the *ping service* is used to check the function of its own interfaces. Thus, the failed interface on the *inactive gateway* is unable

to contact the virtual reference node and detects itself as faulted. Furthermore, the *inactive gateway* does not send out any *master_advertisement_message* contrary to previous cases, but shuts down any communication and effectively removes itself from the logical *gateway* compound.

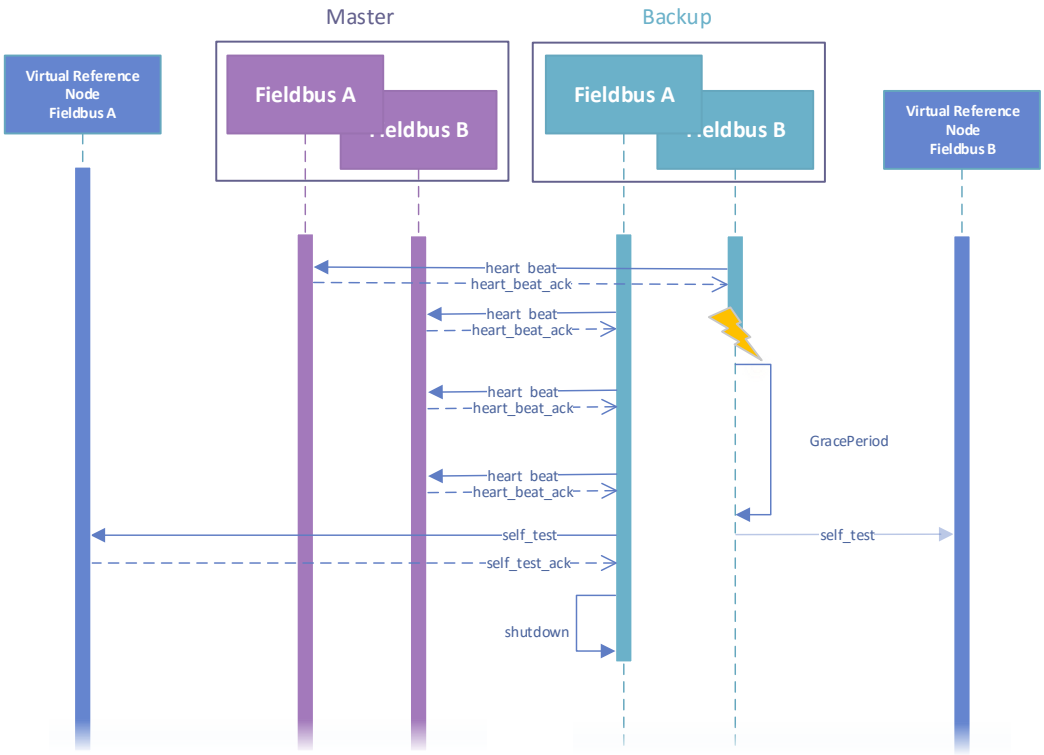

Figure 4.3: Link fault at *backup gateway*

- **device fault at *backup gateway*.** Assume a device fault at an arbitrary *inactive gateway* of the logical *gateway* compound. Thus, such a failed *gateway* ceases to communicate with any of its peers, member of a VRG. Synchronization attempts from the *master* device and possible *master_advertisement_messages* remain unacknowledged. Hence, the failed *gateway* is effectively removed from a logical *gateway* compound.

- **Link fault at *master gateway* with pending transaction.** Suppose a link fault at the current *master gateway*, while the master is busy with a synchronization or information model transaction. In such a case, a distinction must be made, whether the faulty interface is involved in a pending transaction or not.

  At first, assume that none of the regarded transactions is linked to the faulted field network interface. Hence, the transaction is completed, despite any incoming *master_ advertise-*

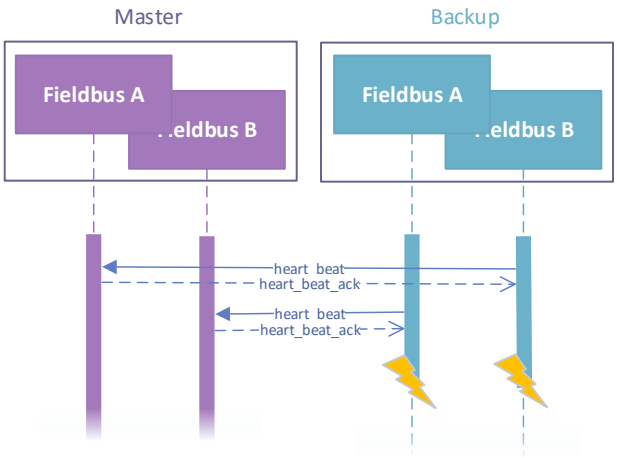

Figure 4.4: Device fault at *backup gateway*

ment_messages. In detail, for a synchronization transaction, the *master gateway* device finishes the transaction and then shuts down any operation. In case of an information model transaction at the *master gateway*, the *poll* and *update* transactions are aborted. The *notify* transaction is finished according to possibility, so any updated data is still pushed out to attached field devices. In the meanwhile, after the *GracePeriod* plus one *HeartBeatInterval* jitter has expired, all functional *inactive gateways* send out their *master_advertisement_ messages*. When furthermore the *MasterAdvertisementTimeout* has passed, the *inactive gateways* agree on a new *master* device. The aborted *information model* transaction has been lost and must be retriggered by the retransmission capabilities of the attached field networks.

Next it is considered, that the faulted interface interferes with a pending synchronization or information model transaction. In case of an information model transaction there is nothing which can be done to recover it. Thus, the transaction is unfortunately lost. On the other hand, a synchronization transaction will timeout and remain unacknowledged, but is retried on one of the remaining operable fieldbus interfaces. The retried synchronization transaction then succeeds and will leave the information model in a consistent state. In the meanwhile, the *inactive gateways* have gone through the *GracePeriod* plus *HeartBeatInterval* jitter. Then again, after the *self-test* of the *backup* devices, the *master_advertisement_messages* are sent out via multicast messages. After the *MasterAdvertisementTimeout* has expired, the fully operable *inactive gateways* agree on a new *master* and normal operation of the logical *gateway* compound is resumed.

- **Device fault at *master gateway* with pending transaction.** Assume a device fault at a *master gateway* with an ongoing synchronization or information model transaction. In either case, the transaction is aborted and inadvertently lost. The *inactive gateway* will sense the failure of the *master* device due to unacknowledged *heart_beat* messages. After the

GracePeriod plus a jitter of one *HeartBeatInterval* has passed, a *self-test* utilizing the *ping service* is performed on all network interfaces. Then the *master_advertisement_messages* are sent out by the *inactive* members of the VRG via multicast messages. The *backup gateways* wait for the *MasterAdvertisementTimout* to expire. After the *MasterAdvertisementTimeout* has finally passed, the *inactive gateways* choose the new *master gateway* by comparing the received *priority values*. Finally, the newly selected *master* device continues operation.

- **Link fault at *backup gateway* with pending transaction.** Given is a link fault at an arbitrary *inactive gateway* of a logical *gateway* compound, while a synchronization transaction is ongoing. A distinction has to be made, whether the synchronization transaction is processed via the faulted network interface or not.

  In case the synchronization transaction is affected by the faulty network interface, it is aborted and inadvertently lost. The *inactive gateway* experiencing the link fault encounters unacknowledged *heart-beat* messages. Hence, the *GracePeriod* is started at the affected member of the VRG. After the *GracePeriod* plus one *HeartBeatInterval* jitter has expired, the *backup gateway* performs a *self-test* by contacting the virtual reference node on each of its network interfaces. Since the message to the virtual reference node via the faulted network interface also remains unacknowledged, the *backup gateway* assumes the link fault is located on its own field network interface. Thus, the concerned *inactive gateway* stops operation and effectively removes itself from the logical *gateway* compound.

  The other case is that the synchronization transaction is not affected by the faulty network interface and therefore completed successfully. But the *inactive gateway* with the faulty network interface still encounters unacknowledged *heart_beat* telegrams. Thus, after the *GracePeriod* plus one *HeartBeatInterval* jitter has expired, the affected *backup gateway* will start a *self-test*. Again, a virtual reference node is contacted on each field network interface, and the message sent out via the faulty interface remains unacknowledged. Hence, the *inactive gateway* experiencing the link fault shuts down its operation and effectively removes itself from the logical *gateway* compound.

- **Device fault at *backup gateway* with pending transaction.** Assume a device fault at an arbitrary *gateway device* part of the group of *inactive gateways* of a VRG. All synchronization attempts from the *master gateway* will then fail and remain unacknowledged, meaning that the *backup* device has removed itself from the logical *gateway* compound.

CHAPTER 5

Implementation

In this chapter, a proof-of-concept gateway implementation is presented, in order to demonstrate the feasibility of the described concepts. The evaluation scenario is designed to be as simple as possible, but is able to demonstrate the main features without any restrictions. Therefore, a testbed interconnecting two field networks with the means of a VRG, consisting of two individual *gateways* is considered.

A schematic overview of the proposed testbed is shown in Figure 5.1. There are several criteria for choosing two suitable field level protocols. First, each chosen field level protocol must utilize a different network topology. Further, the topology must be one of the most widespread used, either *line/bus* or *mesh* topology. Additionally, both field level protocols need to target the same problem domain. Hence, *KNX* (see Section 5.1 – KNX) and *ZigBee* (see Section 5.1 – ZigBee) have been selected. First and foremost due to their corresponding network topology and their application domain of BAS. A detailed statement, why those field level protocols are chosen is outlined in Section 5.1.

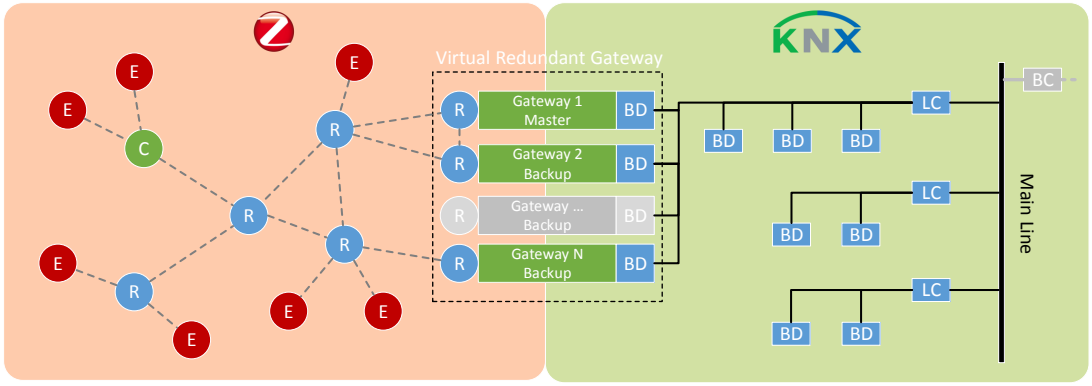

Figure 5.1: KNX-ZigBee gateway schematic overview

Next, the structure of the testbed is discussed in Section 5.2. The test scenario is a lighting scenario, consisting of a switch and a lighting actuator in each field network. The switches shall be able to interact with the actuator, located in the very other network. A description of the used hardware platform for the *gateway* proof-of-concept and field devices is given.

In Section 5.3, the software architecture and overall *gateway* setup is specified. All used building blocks, implemented according to Chapter 3 are described briefly. Furthermore, configuration and initial deployment of the *gateway devices* are outlined. A detailed look on one of the most important components of the translation process between the field networks, the *information model*, is given in Section 5.4. Finally, test cases carried out on the prior mentioned testbed and their results are delineated in Section 5.5.

5.1 Field protocols

There is a wide range of available field protocols from where to choose candidates for a *proof-of-concept* implementation. Each fieldbus protocol leverages different, sometime unique properties that makes them stand-out in their respective target domain. In order to implement a demonstration of the proposed *gateway*, two protocols have been selected.

Several criteria have been applied throughout the selection process. First, both field protocols have to serve the same problem domain. Second, the utilized physical media must be different. Ideally, each chosen field protocol implements one of the major physical media domains, namely wireless or wired. Further, the fieldbusses should either rely on a *line/bus* or *mesh* topology.

Therefore, *KNX* and *ZigBee/Home Automation (HA)* have been selected as interconnected protocols for the *gateway* prototype. Both fieldbus protocols are targeted for the domain of home and building automation. A *KNX* solution, given it utilizes the *TP1* physical layer, relies on a wired medium and implements a hierarchical *line* topology. Whereas, a *ZigBee/HA* solution is realized on top of the 802.15.4 wireless standard [13] and leverages a *mesh* network topology.

KNX

KNX is a well-established standard for *Home and Building Automation*, with core application fields of Heating, Ventilating, and Air Conditioning (HVAC) and electrical installation. It is operable on a wide spectrum of physical media, including powerline, twisted pair and radio (see Section 5.1 – Topology and Physical Media). A special focus has been set on the interoperability of devices from different vendors. This includes an exhaustive interworking model and a rigorous certification process.

The roots of the *KNX* standard are originated in the fieldbus wars [26] of the early 1990's. Three fieldbusses were competing for supremacy of the electrical installation market, namely the European Installation Bus (EIB), European Home Systems Protocol (EHS) and *BatiBus*. At about 1996, the need for a more streamlined and integrated standard was recognized to compete and promote each of the mentioned BAS standards on an international level. Therefore, in 1999 the *Konnex Association* was founded with the aim to merge EIB, EHS and *Batibus*. Later in

2004, the *KNX* protocol was standardized as norm *EN 50090* [34]. Further, in November 2006, *KNX* was recognized as the international *ISO/IEC 14543-3* standard.

A key feature of the *KNX* message protocol is its *observer* pattern based mechanism of information exchange. Multiple *observing* bus devices are notified via a single multicast message of changes to data on a single *source*. This allows the easy creation of *m* - *n* relations, since the *source* is not a fixed device. Hence normal bus traffic is not transacted via point-to-point messages, but so called *group* communication. Therefore, special *group* addresses are foreseen, which enable a receiving device to decide if it is member of such a *group*. Thus, a received message can either be ignored or processed.

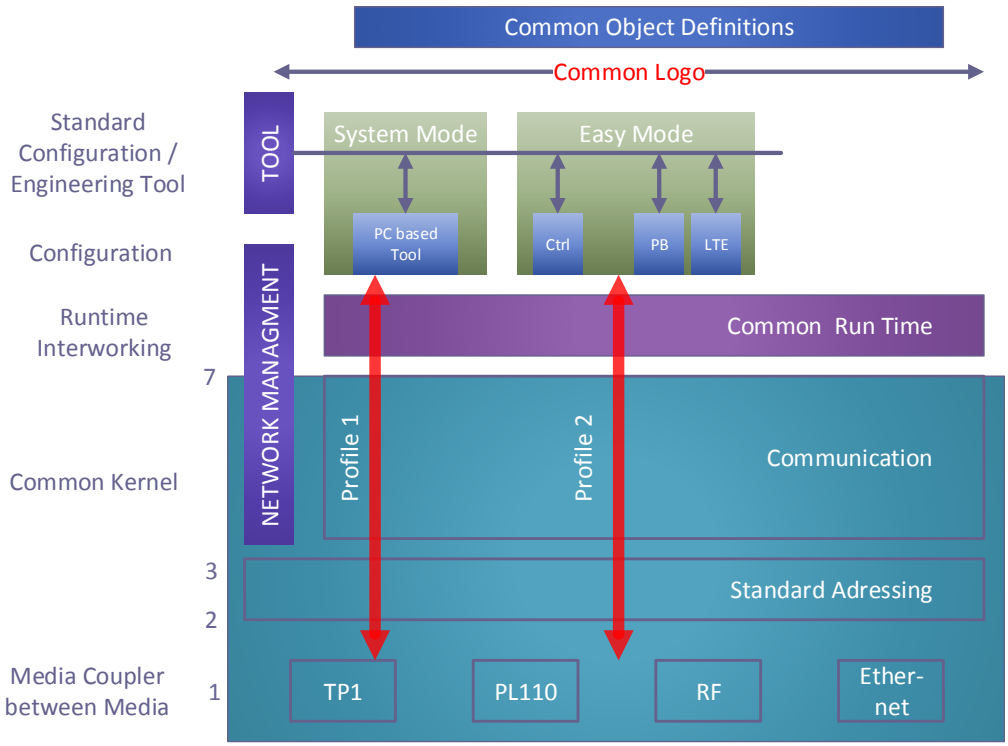

Figure 5.2: KNX Model [34]

Apart from *group communication*, *KNX* [34] defines four key aspects of its overall network model as shown in Figure 5.2:

- **Interworking and Distributed Application Models** are the core component of *KNX* and its intended application domain BAS. Therefore, *KNX* defines the idea of *datapoints* (see Section 5.1 – Datapoints), which represent the process and control variables of a system. A reduced instructions set with *read* and *write* operations is provided to interact with those *datapoints*. Further, *functional blocks* (see Section 5.1 – Functional block) are defined, which group *datapoints* into standard building blocks for applications.

- **Schemes for Configuration Management** to configure and manage various elements of the network and their logical binding. This can be achieved on two levels, first the *easy mode* leverages manual interaction on the single *KNX* bus components, like pressing a button or setting a jumper. Or second, the *system mode*, which utilizes a PC based tool to configure the *KNX* network via point-to-point messages.

- **Communication System.** This includes several different physical media like *twisted pair*, *powerline*, *radio*, and *Ethernet* (see Section 5.1 – Topology and Physical Media). On top of the physical layer and a medium dependent data link layer, a common communication stack is defined with a compliant ISO/OSI model covering layers 3,4 and 7. Thus, bus devices using different physical media can easily interact along *coupling* devices with each other.

- **Device Models**, combine all elements listed above in ready to use device profiles. Apart from functionality defined by *functional blocks*, capabilities of bus devices also varies with their intended *role* in a *KNX* network, like *application end device* or *configuration master*. All those different operational behaviors are defined in the device profiles.

Topology and Physical Media

The *KNX* protocol leverages a logical hierarchical *line* topology with a 16-bit wide address space. The address space is divided into several subnetwork areas, called *line*, which are able to connect up to 256 devices. 16 lines may be grouped together to *areas*, whereas 15 *areas* may again form a *domain* connected by a *backbone line*. Different levels of lines are interconnected by *coupler* devices, the *line coupler* and the *area coupler*.

- **Twisted Pair (TP-1)** is considered the main physical transportation medium and has a data rate of 9600 bits/s data signal. Also a 29V DC supply voltage signal is carried by the *twisted pair* medium, in order to power *KNX* devices without additional wiring. Data frames are encoded in a balanced baseband signal encoding [34]. Further, *TP-1* physical topology can either be a *line*, *tree*, *star* or any mix of those topologies. A logical *KNX* line can be comprised of up to four physical segments connected by special repeater devices, called *line couplers*. Such a *line* can extend up to 1000 m and is able to address a maximum of 255 end devices. *TP-1* allows devices to concurrently access the fieldbus and therefore deploys Carrier Sense / Multiple Access (CSMA) with bitwise arbitration as a control mechanism. [33]

- **Powerline (PL-110)** allows the usage of the standard electrical wiring in a building as a physical transport medium for communication. The data signal is modulated using *Spread Frequency Shift Keying* on a frequency range of 95 kHz to 125 kHz according to norm EN 50065-1. The physical topology follows the layout of the electrical wiring and is not restricted by the specification. The data rate, achieved by *PL-110* is 1200 bits per second, and a maximum of 255 devices can be addressed in a single *PL-110* domain. In order to allow multiple access to the power line, a priority based time-slot mechanism is

deployed. Each priority level defines different back-off times after a previous transmission to minimize bus collisions.

- **KNX-RF** enables communication via radio, utilizing the publicly available Industrial-Scientific-Medical (ISM) frequency band at 868 MHz. *KNX-RF* devices are able to communicate at speeds up to 16384 bits per seconds. The data signal is modulated by Frequency Shift Keying (FSK), and medium access is carried out by a CSMA algorithm. Further, bidirectional and unidirectional communication are foreseen, whereas the last mentioned mode of communication saves on battery life, because no receiver needs to be powered. A speciality of *KNX-RF*, is its unique addressing scheme compared to the other *KNX* media. In order to distinguish different *KNX* installations from their neighboring installations, the serial number of the device is added to the traditional *KNX* address. Further, the multicast group address of a sender contains its serial number, so no two senders are able to address the same *KNX-RF* group address. Hence, the original *m* - *n* relation, is reduced to a *1* - *n* relation for *KNX-RF*.

- **KNX IP** is the fourth physical medium and is not to be confused with *KNXnet/IP*. *KNX IP* devices are directly connected to *Ethernet*, whereas *KNXnet/IP* allows only tunneling between different segments of the *KNX* network. As a benefit, existing home and office IP installations can be used jointly. Moreover, the high bandwidth inherent to this medium allows new usage scenarios which were unthinkable with *TP-1*, *PL-110* or *KNX-RF*. *KNX IP* uses a User Datagram Protocol (UDP) to encapsulate the dataframes of the *KNX* protocol.

Functional block

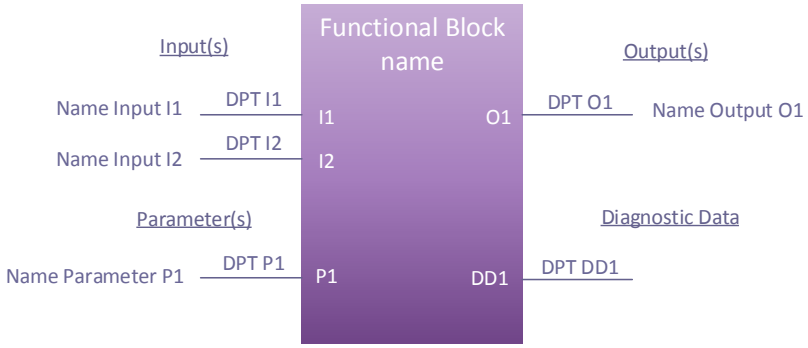

Figure 5.3: Functional block [34]

The *KNX* interworking principles rely on the various application models defined by the standard [34]. A *functional block* is a standardized *datapoint* interface description of a component utilized in such an application model. Each *datapoint*, part of a *functional block*, is assigned a

descriptive name and a *datapoint* type determining the format and encoding of such a process data variable. A *KNX* bus device implements at least one *functional block*, but is also able to house multiple different *functional blocks*.

As shown in Figure 5.3, a *functional block* consists of four categories of *datapoints*:

- **Outputs** are *datapoints* that publish data, and subsequently send out *write* messages to connected *input datapoints*. An example, is a *push button* that makes its *on/off* state available as a binary *datapoint*.
- **Inputs** are *datapoints* that are able to subscribe to compatible *output* data types. Given a *light bulb functional block*, the state of the light bulb, *on/off* can be provided as a binary *datapoint*. Regarding the example of the *output category*, the *on/off* state of the *push button* can be bound to the *on/off* state of the *light bulb*. Hence, a simple *light switch* application is created.
- **Parameters** are *datapoints* that define the operational behavior of the underlying physical sensor or actuator. Given a temperature sensor, a possible *parameter datapoint* would be the sampling time of the individual measurements.
- **Diagnostic Data** are special *output* like *datapoints* which are only used during debugging or maintenance.

In Figure 5.4, an example of a *functional block* description is shown. The fields shaded in dark gray are mandatory *datapoints*, whereas all other fields are optional *datapoints*. Obviously, a valid realization of the *sunblind actuator basic (SAB)* may only implement the mandatory *datapoints*.

Datapoints

KNX datapoints resemble the concept of communication endpoints or ports of a *functional block*. The information flow direction of a *datapoint* is either *out* (transmit, write) or *in* (receive, read). The semantics of such a *datapoint* is defined by its *datapoint type*. Hence, an *outbound datapoint* can only be connected to a *inbound datapoint* of the same *datapoint type*. A *datapoint type* provides the following information about the content of a *datapoint* variable:

- **Format**, firstly indicates the overall length of the *datapoint* field in bits. Furthermore, the *datatype* of a field is specified, like *boolean*, *string* and so on.
- **Encoding** describes the concrete representation and possible valid values, given the *format*. Given an enumeration *format*, the *encoding* contains the valid states of the enumeration.
- **Range** defines the numerical limits of a *datapoint*. Considering a *format* that stores a percentage, *range* defines a minimum of 0% and a maximum of 100%.
- **Unit** specifies the used engineering unit.

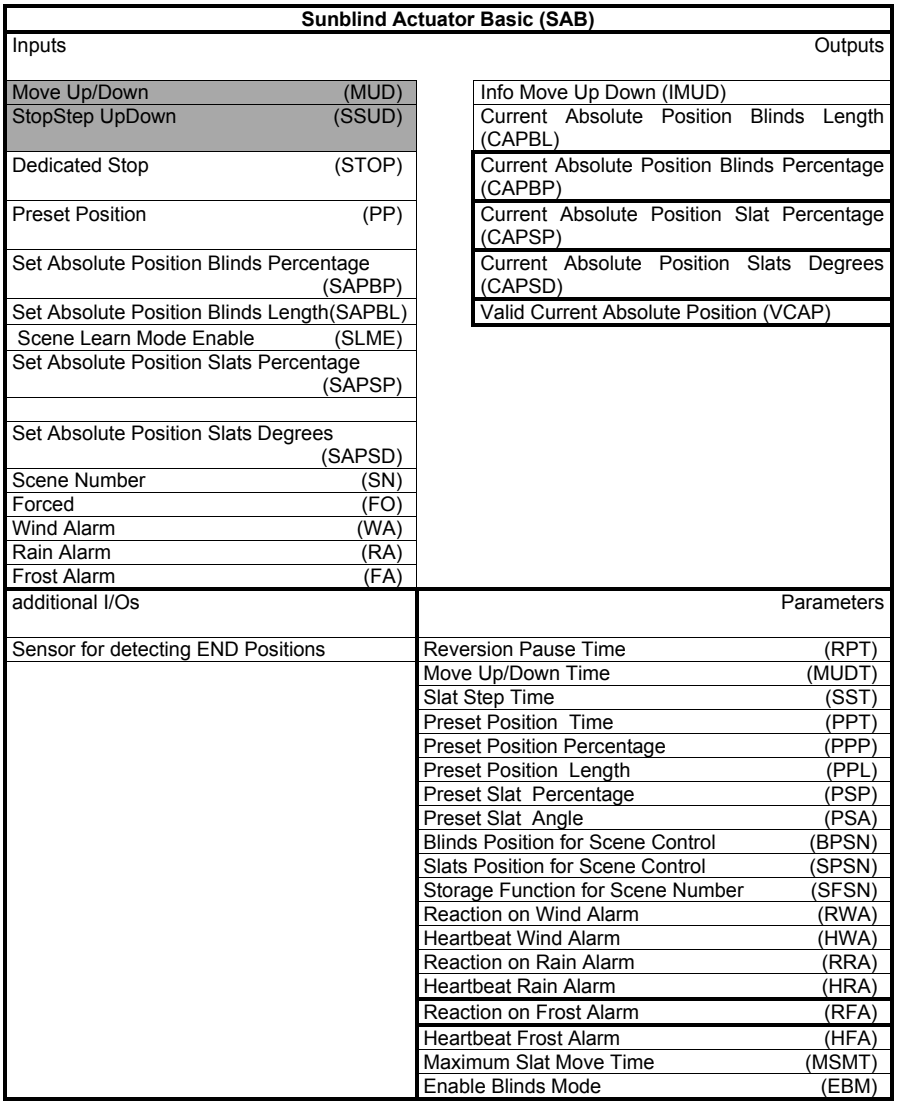

Figure 5.4: Functional block diagram example [34]

ZigBee

The *ZigBee* protocol is a well-established standard for low-cost, low-power, low-rate, wireless personal area networks. The main idea behind the *ZigBee* specification, is the reliable interconnection of exposed sensors or actuators that cannot be interconnected by wire for economical reasons. So far, multiple application domains have been foreseen by the standard in form of application profiles on top of the *cluster library* (see Section 5.1 – Application profiles and cluster library). Examples for specified profiles are *Industrial Plant Monitoring (IPM)*, *Home Automation (HA)*, or *Advanced Metering Initiative (AMI)* [29].

Similar protocols to *ZigBee* began to emerge in 1999. Since existing protocols, like *WLAN* –

IEEE 802.11 and *Bluetooth*, were unsuitable to form wireless sensor and control networks, due to the lack of various features, like radio range or power consumption. Different vendors, realized that such a new wireless standard can only succeed as a single world-wide standard. Hence, in 2002 the *ZigBee Alliance* was formed. But it took two more years until the first version of the standard was released to the public in 2004. In 2006, a new revision of the standard was published, replacing the *message / key value pair* structure of the 2004 version with the more versatile *cluster library* (see Section 5.1 – Application profiles and cluster library). The current revision of the standard was published in 2007, which defines two different stack profiles. First, the *stack profile 1* or simply *ZigBee*, intended to be used for home and light commercial applications. Second, the *stack profile 2* or *ZigBee PRO*, which offers more capabilities, like multicast messaging and many-to-one routing [53].

Figure 5.5 shows the architecture of the *ZigBee* stack resembling the ISO/OSI 7 layer model. The *ZigBee* stack consists of several independent layers, which are interconnected by so called Service Access Points (SAPs).

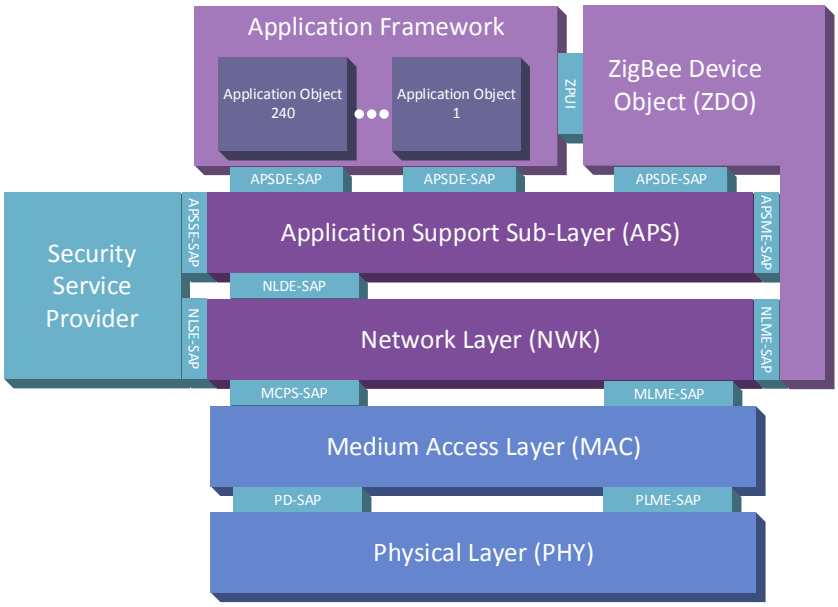

Figure 5.5: *ZigBee* architecture [29]

The physical and data link layer of the *ZigBee* network model rely on the IEEE 802.15.4 standard [13] in the form of 2003. Although newer releases of the IEEE 802.15.4 norm are readily available, they have never been adopted into the official *ZigBee* specification. There are three different radio bands available. One, utilizing the ISM band at 868 MHz available in Europe and operating on a bit rate of 20 Kbps. Second, the 915 MHz ISM band available in North America, which provides a bandwidth of 40 Kbps. And finally, the 2.4 GHz band, which is available world-wide and is able to sustain a bitrate of 250 Kbps [25].

The network layer is responsible for all tasks concerning message transmission and network management. Hence, it provides facilities for broadcast, multicast, and unicast messaging. Further, together with the *Security Service Provider*, secure joining, rejoining and packet encryption are provided.

The main idea behind the application support layer is to simplify message handling for application objects. Therefore, filtering for duplicate messages and a binding table for communication partners is maintained by this layer. The application support layer also validates endpoints used by application objects against application profiles (see Section 5.1 – Application profiles and cluster library).

The *ZigBee* device object component represents and manages the current state of the node inside a network. Thus, it provides mechanisms for discovering other devices and services within a *ZigBee* network.

Finally, the application framework hosts different, independent application objects and connects those application objects via individual endpoints to the remaining *ZigBee* stack. Different endpoint numbers can then be used to redirect messages to different services or application programs. The application framework also accommodates the *ZigBee Cluster Library*, which provides ready-to-use application building blocks (see Section 5.1 – Application profiles and cluster library) [29].

Device types

IEEE 802.15.4 defines two different device types, namely the Full Function Device (FFD) and the Reduced Function Device (RFD). A *FFD* implements the full communication stack, defined by the IEEE 802.15.4 norm and therefore is able to perform all foreseen task. Whereas the RFD lays its focus on battery life, memory, and computation consumption. Hence, the capabilities of the RFD are limited, and a RFD is only able to communicate with a FFD. On top of the device types defined by IEEE 802.15.4, *ZigBee* deploys a more fine grained differentiation of types:

- **ZigBee Coordinator (ZC)** is the most capable device type in a *ZigBee* network. There is exactly one ZC in an individual network, since it is responsible for starting the network in the first place. Furthermore, this device type is responsible for the joining of new devices, security and other management tasks. Since the tasks of a ZC are quite power hungry, it is advised to choose an application that already requires mains connection for this role. In terms of the IEEE 802.15.4 standard, this is a FFD device.

- **ZigBee Router (ZR)** is able to forward and redirect messages from other *ZigBee* devices. It is also capable of hosting and serving arbitrary application. A ZR, again resembles the functionality of a FFD in IEEE 802.15.4.

- **ZigBee End Device (ZED)** are the most limited devices, residing in a *ZigBee* network. It only has basic communication mechanisms for transmitting and receiving data concerning its own hosted application. Therefore, a ZED is the most power effective device type from the *ZigBee* perspective, as it is able to stay asleep for a significant amount of time. Thus, a ZED corresponds to a RFD device in terms of the IEEE 802.15.4 norm.

Topologies

Network topologies in *ZigBee* must assume one of the two communication structures defined in the IEEE 802.15.4 specification:

- **Star topology.** In a *star topology*, every device in a *ZigBee* network is only able to communicate with the ZC that originated such a network. Therefore, this topology is only suitable for small instances.

- **Peer-to-peer topology.** Every participant in a *ZigBee* network is allowed to communicate with every other participant, as long as the device type sustains the necessary capabilities of the communication stack. A *peer-to-peer topology* can assume different manifestations. Ideally, a full *mesh topology* is formed, since multiple communication paths provide a higher level of fault tolerance. But also *tree topology* like forms are possible, where ZED devices are grouped around ZR devices which are then interconnected with the ZC.

Application profiles and cluster library

Every communication transaction fulfilled in a *ZigBee* network is carried out upon an *application profile*. Such an *application profile* describes a certain problem domain, and hence groups related devices into a single application model. The *ZigBee* specification differentiates between two different classes of application profiles, namely *public profiles* and *private profiles*. *Public profiles* are defined by the *ZigBee Alliance* and aim at interoperability between different device manufacturers, whereas *private profiles* are specified by individual vendors.

Public profiles are built on the foundation of the ZigBee Cluster Library (ZCL), which provides predefined application objects or *clusters*. Therefore, clusters encapsulate *attributes* as the current state and *commands* as actions that may be taken. Given the *on/off cluster* as an example, it defines the necessary *attributes* and *commands* to model an arbitrary object that has an *on* and *off* state. However, no additional semantic information is foreseen, which identifies a concrete instance, like a pump or a light. Hence, with the ZCL alone, it is not possible to determine if a pump, a light or something else is operated.

The ZCL utilizes the *client/server* principle. One endpoint is the *client*, which has a predefined pool of transactions, dependent on the cluster, at its disposal. Another endpoint is the *server*, which can execute a certain set of functionality based on the implemented cluster. *Server* and *client*, implementing the same cluster are then able to interconnect. Figure 5.6 shows an example of *ZigBee* devices, interconnected by the means of the ZCL. Thus, the *On/Off Switch* is able to control the *On/Off Light* by the transactions defined by the *On/Off Cluster*.

Finally, Figure 5.7, depicts the specification of a *shade actuator* device profile from the *Home Automation Profile*. There is a clear distinction between the *client* functionality and *server* functionality of the ZCL. Additionally, not all suggested clusters are *mandatory*, and also *optional* clusters are foreseen. This opens up the possibility to implement different set of features, and still serving same device profile, while obeying the *Home Automation Profile* specification.

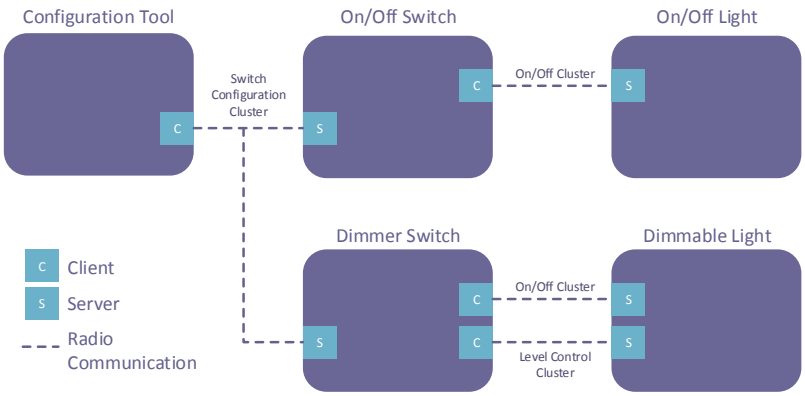

Figure 5.6: Cluster client/server example [15]

Server Side	**Client Side**
Mandatory	
Shade Configuration	*None*
On/Off	
Level Control	
Scenes	
Groups	
Optional	
None	*None*

Figure 5.7: *ZigBee* shade actuator [17]

5.2 Testbed

A schematic overview of the testbed for the proof-of-concept *gateway* implementation is depicted by Figure 5.8. Two field level protocols were chosen, that are representatives of the most commonly used network topologies. As determined by Section 3.2, these are *line/bus* and *mesh* topology. Therefore, as depicted in Figure 5.8, the green area represents the *KNX* network, which leverages a *line* protocol. Whereas the red area illustrates the *ZigBee* network, which deploys a *mesh* communication structure. Furthermore, both considered field protocols do not only differ in their network topology. *KNX* is built upon wired connections and *ZigBee* communicates via radio. But, both field level protocols target the same application domain of BAS and therefore model similar application tasks and functionality. Another nicety, are the extensive interoperability models, provided in form of functional specifications for different devices and

scenarios. Hence, various ready-to-use field devices are available for the testbed.

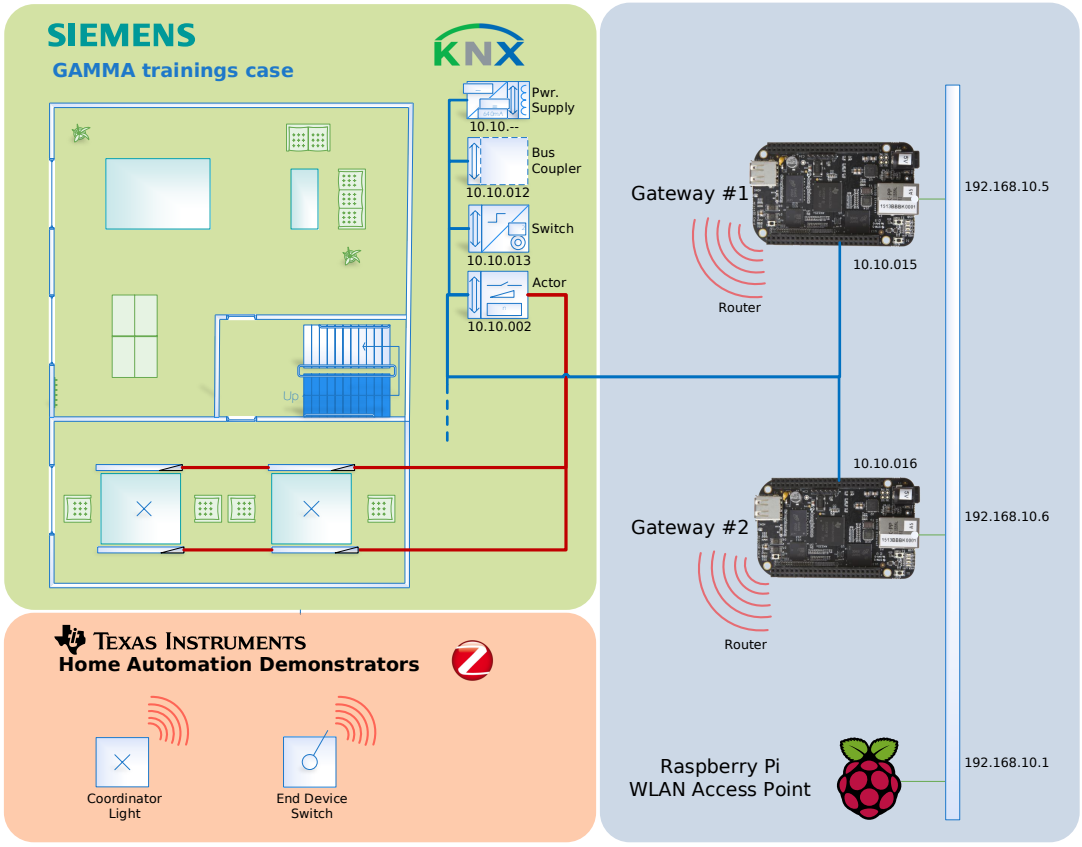

Figure 5.8: Topology of evaluation testbed

The application domain of BAS gives rise to several possible test scenarios, like heating, lighting and venting. In order to keep the proof-of-concept *gateway* implementation as simple as possible, a lighting scenario has been chosen. Each field network consists of a binary switch field device and a lighting actuator field device. The switches of each network shall be enabled to operate the lights by utilizing a VRG consisting of two disjoint *gateway* devices. Hence, the testbed outlined by Figure 5.8 is advised.

The hardware platform for the two *gateway* devices in the depicted scenario are based on *beaglebone black* development boards. The *beaglebone black* development boards consist of a powerful *ARMv7 CPU* clocked at 1 GHz and 1 GB of RAM, running a customized *Debian Linux* platform. Thus, there is plenty of processing power available to fulfill the tasks. The interface devices, a *TPUART* USB module for *KNX*, and a *CC2531* USB dongle for *ZigBee*, are connected via USB. Further, there is a backbone *Ethernet* connection available which is used for engineering purposes, but may also be used as a backbone link for *gateway* synchronization in future work.

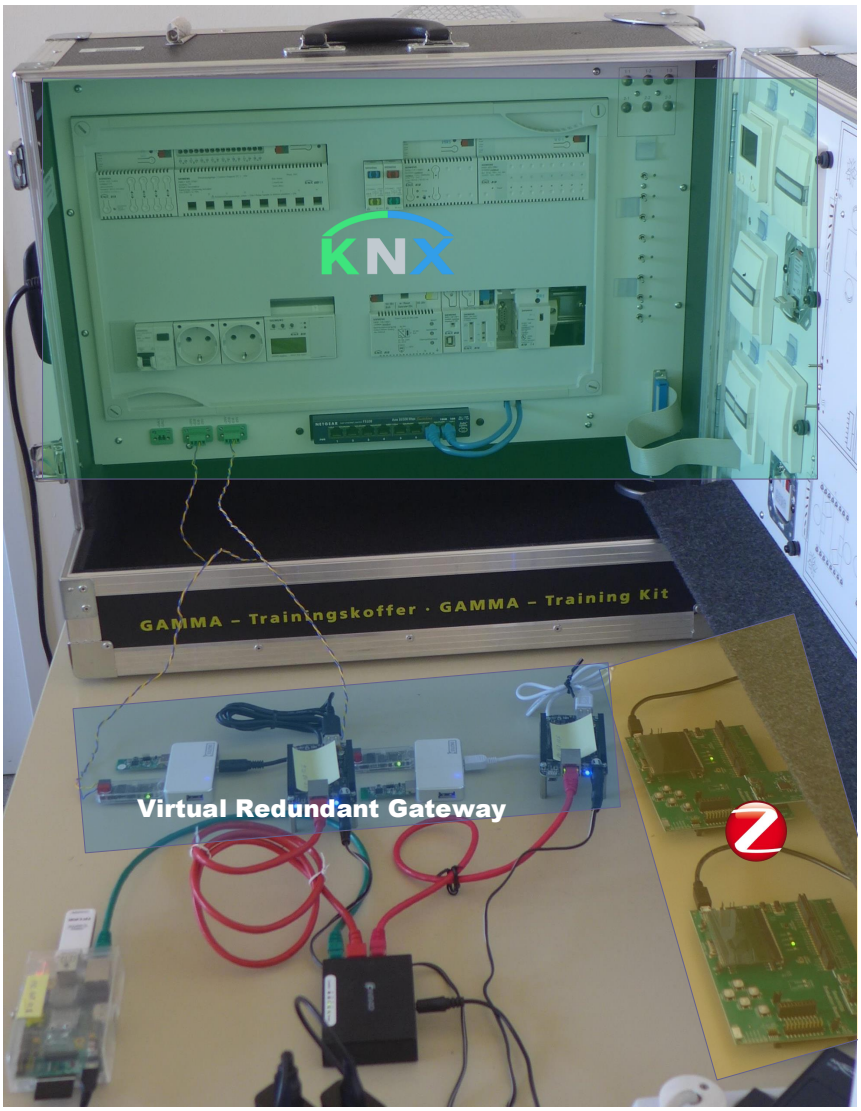

Figure 5.9: Testbed

The *KNX* network is realized via a *Siemens GAMMA training kit*, which offers a base platform for various HBA scenarios. However, only the field devices relevant to the proposed lighting scenario are considered. Therefore, a *KNX* switch and lighting actuator operating on two light bulbs is selected among the available field devices. The base configuration already *wires* the switch and lighting actuator, so existing group communication (see Section 5.1 – KNX) can be reused without reconfiguration. This is done by simply binding the concerned virtual field device statically to the respective group communication at the proof-of-concept *gateway* device.

The *ZigBee* network is implemented by two *CC2538* development boards provided by *Texas*

Instruments. As part of the promotion of the *ZigBee* protocol, and the HA profile especially, *Texas Instruments* offers demonstration software projects for each field device defined in the context of the HA profile. Thus, one *CC2538* development board is programmed as a *ZigBee coordinator*, resembling a lighting actuator. As already mentioned in Section 5.1 – ZigBee, a device with a permanent connection to a power supply is most suitable for the role of a *coordinator*. Finally, another *CC2538* development board implements the lighting switch in the role of a *ZigBee end device*. Figure 5.9, shows a picture of the actual testbed used for testing of the proof-of-concept *gateway* implementation.

5.3 Architecture

The internal architecture of the proof-of-concept *gateway* device follows the structure outlined in Section 3.3. Therefore, the internal structure of the *gateway* software is divided into several disjoint building blocks according to Figure 3.5. First and foremost, the coupling to the two considered fieldbusses, *KNX* and *ZigBee*, and a communication stack is implemented. The next vital building block is the *information model*, which powers the translation process between the fieldbusses. The communication stack and the information model are interconnected by the *protocol binding* layer, which consists of the necessary functionality to redirect fieldbus telegrams and provides virtual field devices. Also, the *information model* itself is modeled (see Section 5.4). Finally, components, part of the redundancy concept, the bus guard, bus load monitor and the redundancy manager are created and integrated.

As an operating system platform for the software implementation of the proof-of-concept *gateway* device, a *Debian Linux* has been chosen. Main reasons were the large amount of ready-to-use software packages and rapid development cycle possible due to the familiarity with *Linux* in general. Readily available software packages also included fieldbus drivers and communication stacks, while most of the remaining functionality had to be implemented. Notable, a more sophisticated implementation of a proof-of-concept *gateway* device might choose a real-time capable software platform, in order to satisfy certain latency guarantees. However, this proof-of-concept implementation limits itself to demonstrate the feasibility and viability of the proposed fieldbus translation mechanism and the reliability concept.

The following enumeration describes all the building blocks according to Figure 3.5, that were created to form a working proof-of-concept *gateway* device:

- **KNX communication stack:** The *knxd* project provides the communication stack connecting the *KNX* network to the *protocol binding* layer and further to the *information model*. *knxd* is a fork of the *eibd* project which has been advised in [55]. The communication stack provides a complete *API*, including message monitoring, filtering and forwarding. The physical connection is done via a *USB* attached bus coupler device that forwards incoming messages via a *tty* software interface.

- **ZigBee communication stack:** The interface to the *ZigBee* network is achieved via the *Texas Instruments Z-Stack Linux Gateway*. This *gateway* driver solution was intended as a demonstration project for a *gateway* between Ethernet and IP based applications and *ZigBee/HA*. However, the original implementation has been alienated by the proof-of-concept

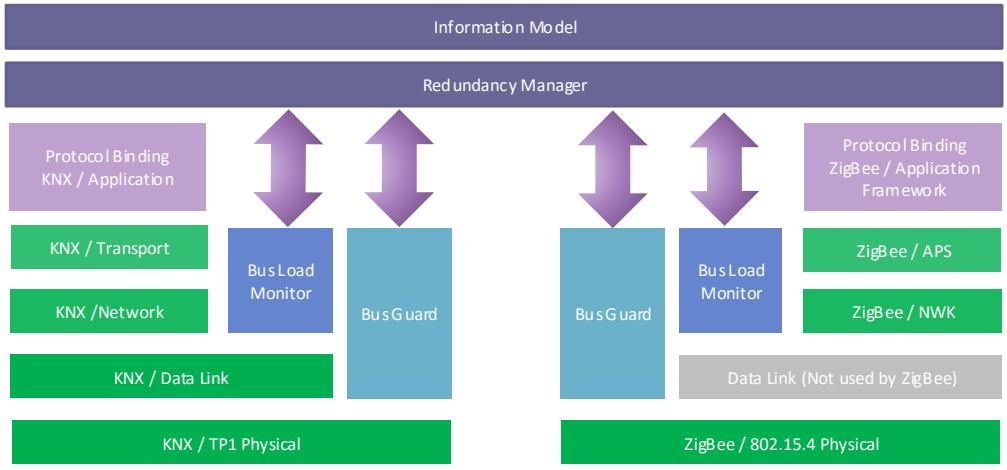

Figure 5.10: Gateway component overview

implementation and was integrated via local *unix* sockets into the *gateway* software architecture.

- **Information Model:** The *information model* is implemented with the help of a *mysql* relational database. A simple database schema directly tailored to the lighting test scenario was created and used for the storage of the different entities of the *information model*.

- **Protocol Binding:** The *protocol binding* layer utilizes the stored entities at the *information model* via atomic database transactions. Thus, the *mysql* database already provides the necessary synchronization to avoid concurrent access to stored information. Virtual field devices are directly implemented at the protocol binding layer of the corresponding fieldbus. The address field of each message retrieved at the fieldbus interface of the proof-of-concept *gateway* device is matched against the list of virtual field devices. In case a match has been found, the message is processed by the responsible virtual field device. In the other case, the message is simply dropped.

- **Bus Guard:** For each supported fieldbus, a *bus guard* component instance is created. The *bus guard* then transmits periodic messages to the field network address, which are automatically acknowledged by the communication stacks of the target *gateway* devices. The configuration parameters of the proof-of-concept *gateway* device are hardcoded. The *heart-beat* interval is set to one second, whereas the *grace period* is set to three consecutive failed transmission attempts.

- **Bus Load Monitor:** The instances of the *bus load monitor* building block attach themselves to the field protocol stacks as passive listeners in promiscuous mode. Thus, simply

all incoming and outgoing messages are counted and a load indicator is built. Such an indicator is calculated by the means of a *moving average* algorithm. More sophisticated influence factors regarding the load indicator are not incorporated into the *bus load monitor*, like *signal strength* or *link quality index* for wireless fieldbus networks. This is mainly to keep the code simple and the load indicators from different field networks comparable without any tweaking or balancing mechanisms or factors.

- **Redundancy Manager:** The *redundancy manager* utilizes the information from the *bus guard* and the *bus load monitor* components to implement the reliability concept. Synchronization between individual *gateways* is achieved by subscribing to change notifications from the *information model*. Furthermore, any message directly addressed to the *gateway* device is interpreted as *gateway* backbone communication. Hence, such telegrams are always directed to the *redundancy manager*, while messages with differing addresses are matched against the virtual field devices residing at the *information model*.

Finally, all outlined software components are interconnected via *Unix* sockets and the *ZeroMQ* message queue library. *ZeroMQ* provides a high performance message passing system with *publish/subscriber* mechanisms and is able to serialize arbitrary data structures. There is also support for various programming languages. All custom software core components of the proof-of-concept gateway device have been written in C, with some minor binding written in Python.

5.4 Information model

The *information model* describes the proposed lighting scenario built by the testbed. It consists of a switch and a lighting actuator in each field network that is interconnect by the VRG, formed by the proof-of-concept *gateway* devices. Thereby, the *information model* is created by the means of the proposed meta-model in Section 3.4 and hence follows the MVP paradigm. Figure 5.11 illustrates the *information model* and its interconnection with the attached fieldbusses as given by the testbed lighting scenario.

The dashed line, in Figure 5.11, shows the boundary of a single proof-of-concept *gateway* device. The highlighted areas in red and blue illustrate the areas that interact with the field level protocols. Therefore, the *presenter* layer and *view* layer are located in the afore mentioned areas. Finally, the blue highlighted area contains the entities of the *information model*, that are dedicated to the *model layer*.

- **Model:** The *model* part consists of two *Object* entities. A *Object*, represents the state of one lighting application task. The *Object* instance, named *Light1*, depicts the *KNX* switch field device interacting with the *ZigBee* lighting actuator. Whereas the other *Object*, named *Light2*, resembles the binding of the *ZigBee* switch device to the *KNX* lighting device. Each *Object* has a reference to a *Variable* entity holding the *on/off* state as a boolean value.

- **ZigBee Presenter:** The *presenter* layer part, serving the *ZigBee* network, maps the two *KNX* fieldbus devices considered by the lighting scenario into the *ZigBee* domain. Two

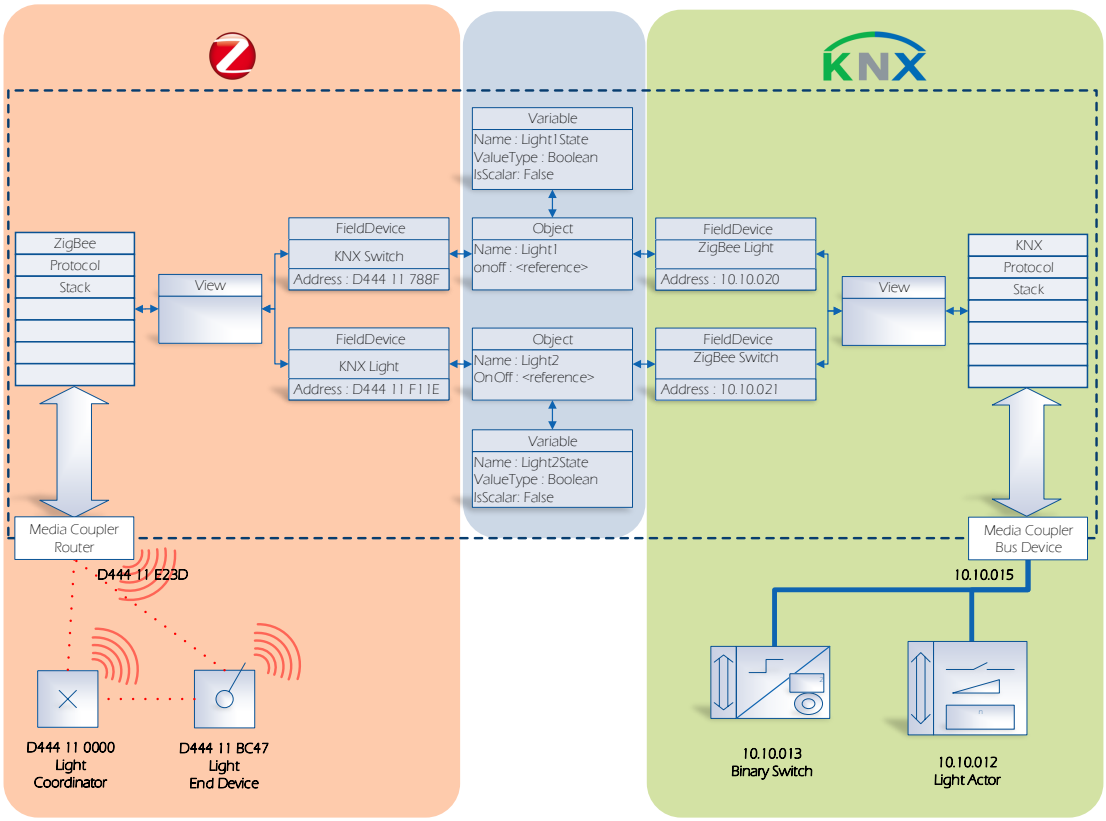

Figure 5.11: Information model of evaluation testbed

FieldDevice entities are modeled, where one represents the *KNX* switch and the other one the *KNX* lighting actuator. Both *FieldDevice* instances comprise of an address attribute, storing the 64 Bit *ZigBee* address. For each *FieldDevice* instance, the protocol binding layer creates a virtual field device and forwards changes via the proposed information model transactions.

- **KNX Presenter:** The *KNX* part of the *presenter* layer consists again of two *FieldDevice* entities. Both entities model the representation of the *ZigBee* field devices in the domain of *KNX*. Therefore, each *FieldDevice* instance comprises of an address attribute, that stores the *KNX* address. Again, the respective protocol binding layer creates a virtual field device for each single *FieldDevice* entity and redirects incoming messages accordingly.

- **View:** As proposed in Section 3.4, a individual *View* entity provides a single entry point for each protocol binding layer.

5.5 Test cases and results

Besides the intended operation of the proof-of-concept, each case from the fault analysis in Chapter 4 was considered. The *gateway devices* were preconfigured by configuration files. One *gateway* was initially set to assume the *master* role, while the other assumed the *backup* role. The VRG and the rest of the field network were set up according to the testbed description in Section 5.2. The *virtual reference node*, according to Section 3.5 – Bus guard, was manually set to the single switch field device, present in both field network segments. Results were obtained by connecting via remote shell access to the individual gateways and evaluated via console outputs.

- **Intended operation of VRG.** The normal operation of the overall system was tested by pressing one of the switch field devices. It is then observed that with respect to the prior state of the lighting actuators, either all lights turn on or off. Different sequences of button presses while varying the pressed switch did not show any unintended behavior. Furthermore, proper synchronization from the *master* to the *backup* gateway was observed by comparing the database dumps after a test sequence and by evaluation of the debug console outputs on the individual *gateway* devices.

- **Link fault at *master gateway*.** The link fault was triggered by simply disconnecting the wires at the *KNX USB dongle* of the *master gateway*. As expected the *backup gateway* recognized the interface outage due to failed *heart-beat* messages. Thus, the *bus guard* notified the *redundancy manager* after the expiration of the *grace period*. The hand-over procedure between the old *master* gateway and the *backup* gateway was accomplished immediately after the notification. The whole procedure was verified by evaluation of the console output. Furthermore, continued operation of the new *master* was verified by another sequence of switch presses.

- **Device fault at *master gateway*.** The device fault was triggered by interrupting the power supply of the *master gateway*. The *backup gateway* immediately recognized the failed *heart-beat* messages and notified the *redundancy manager* after the expiration of the *grace period*. The *backup gateway* then assumed the role of the *master* and continued operation. This was again verified by evaluation of console output at the *backup gateway*. Also, continued operation of the system as a whole was verified by a sequence of switch presses.

- **link fault at *backup gateway*.** The link fault was again triggered by disconnecting the cable between the *backup gateway* and the rest of the *KNX* bus. Synchronization transaction at the *master gateway* timed out as expected according to console log messages. The *backup gateway* sensed the link fault. After the *grace period* expired, the *backup gateway* operation was shutdown. Again, a sequence of switch presses verified the continued operation of the overall system.

- **Device fault at *backup gateway*.** The device fault was again simulated by detaching the power supply from the *backup gateway*. Again, the synchronization attempt from the *master gateway* failed, as observed at the console log. Operation of the residual system continued as expected and was verified by a sequence of switch presses.

- **link fault at *master gateway* with pending transaction.** The *master gateway* was stopped by a breakpoint, set by a software debugging tool during a pending *information model* transaction initiated by a switch press. Next, the cable to the *KNX* bus was disconnected and the transaction was resumed. The *backup gateway* sensed the disconnected link and initiated the hand-over procedure. In the meanwhile, the *master gateway* tried to reach out via both the *ZigBee* network and the *KNX* network to the *backup gateway*. As expected, the synchronization attempt via *KNX* failed, whereas the synchronization attempt via the *ZigBee* interface succeeded. Finally, the *backup gateway* assumed the role of the new *master*. Operation of the residual system was continued and a sequence of switch operation, turning the lighting actuators on and off, confirmed the working condition.

- **device fault at *master gateway* with pending transaction.** This test case was not carried out, as it is essentially the same procedure as described in test case *device fault at master gateway*. The transaction concerned by the device fault at the *master gateway* is inevitable lost and has to be recovered by the upperlying layer.

- **link fault at *backup gateway* with pending transaction.** A pending transaction involving the *backup gateway* is in any case a synchronization transaction. A test program was deployed at the *ZigBee* switch field device, which automatically toggles its on/off state every 200 ms. Hence the VRG has to process constantly changing data and synchronize its internal state, stored in the *information model*. The link fault was again simulated by disconnecting the cable between the *KNX* network and the *backup gateway*. As observed by the console output at the *master gateway*, a synchronization attempt via *KNX* was carried out. The synchronization message timed out and a successful retry via the *ZigBee* link was carried out. As observed at the console output, the *backup gateway* detected the link fault and removed itself from the VRG. Flawless operation was verified by the before mentioned test program and was observed at the lighting actuators.

- **device fault at *backup gateway* with pending transaction.** This test case is essentially the same as case *device fault at backup gateway*. It was observed at the *master gateway*, that synchronization attempts time out. However, continued operation was monitored and additionally tested by a sequence of switch presses.

CHAPTER 6

Conclusion

The goal of this work was the design of a reliable *gateway* approach solving the horizontal integration problem at the field level of the automation pyramid (see Figure 1.1), and a first approach has been published in [48]. As stated by [51], horizontal integration of field networks never reached significant relevance due to the heterogeneity of the mostly application driven design of field level protocols. However, latest protocol consolidations at the field level, and the emergence of standardized device profiles for field devices modify the prior statement. As described in Chapter 2, existing approaches target mostly the vertical integration problem, with a special focus on Ethernet and IP interconnection towards upperlying layers of the automation pyramid. Further, the few approaches, that are conceived as horizontal integration solutions, lack a reliability concept and are therefore not fit for critical systems like personal life safety systems.

The proposed design especially focuses on the reliability aspect of such a *gateway* approach. Therefore, possible network topologies of different field networks are analyzed at first in Chapter 3. The evaluation yields, that a proper *gateway* implementation must not compromise the fault hypothesis of an attached fieldbus. However, the fault hypothesis of the whole interconnected system is ideally determined by the field network with the weakest fault hypothesis and not the interconnecting *gateway*. As already mentioned by the prior statement, the integrity of the fault hypothesis of an arbitrary *gateway* attached fieldbus is not only influenced by the design of the *gateway* device. Also, the fault hypothesis of another fieldbus must be taken into account. This is due to the mapping of field devices, performed at the proposed *gateway* device, and the fact that the *gateway* is not able to strengthen the fault hypothesis of an attached network.

For instance, consider two field devices, interconnected by the proposed *gateway* solution. One such field device resides in a field network with a strong fault hypothesis, and the other one resides in a network with a weak fault hypothesis. A mapped field device follows the fault hypothesis of its originating field network. Hence, the virtual field device mapped from the network with the weaker fault hypothesis into the field network with the stronger fault hypothesis weakens the stronger fault hypothesis, while the mapping of the virtual field device with the strong fault hypothesis does not strengthen the weak fault hypothesis. Hence, a general relia-

bility concept needs to be deployed to truly enable reliability, not only for the interconnecting device, but also for the whole interconnected system. An exhaustive discussion on this topic in the domain of BAS has been performed in [39]. A general approach to this problem is a highly anticipated topic for future work.

The translation process between different field networks has been designed using a stateful approach in this work, resting upon a common *information model*. A stateful approach allows a decoupling of field protocol timing requirements, aggregation and dissipation of telegrams, and a more flexible handling of information in general. But, a stateless translation process enables direct end-to-end delivery of data, and therefore end-to-end acknowledgments. End-to-end acknowledged messages eliminate the problem of lost information at a *gateway* during a fault, whereas the proposed solution redirects the problem to the upperlying application. Furthermore, in order to deploy a reliability concept in a stateless approach, every attached field protocol needs to allow multiple links between communication partners. Additionally, proper routing mechanisms need to be advised to select a single communication path and to provide rerouting in case of a *gateway* outage. In conclusion, a direct stateless translation is superior in very particular scenarios, but there is a vast amount of constraints to be fulfilled in order to be feasible. Hence, a stateful approach, as chosen for the proposed *gateway*, which is more flexible and allows the interconnection of arbitrary field protocols.

The *information model* was highly influenced by the model and concepts deployed by OPC UA. However, considering resource constraints inherent to a typical embedded device, a stripped down version has been advised. Furthermore, a separation of concerns between a field protocol specific model and a core model has been realized by the means of the MVP paradigm. The current design of the *gateway* approach delegates all functional tasks, like data format conversion, to the protocol binding layer. A possible enhancement for future refinements of the proposed concept is the integration of *Function* entities, which are able to model arbitrary operations on the *information model*. OPC UA already implements such a concept, but it was chosen to drop those features to use the *information model* only for data storage.

The protocol binding layer, interconnecting the individual fieldbus communication stacks with the *information model*, is currently designed as a completely custom component. Among its many tasks are data format conversion, virtual field device implementation and provisioning. As part of a future work, the protocol binding layer may be analyzed and general mechanisms extracted, so that a more sophisticated solution is derived. A first step, already outlined in the previous paragraph, was the extraction of certain functionality into reusable elements as part of the *information model*.

A topic, that remained entirely undiscussed by this work so far, is *deployment* and *configuration* of the VRG. The *gateway* approach described, lacks of a proper management service for configuration and formation of a VRG. Hence, a *gateway* management service has to be advised, including important operations like *joining*, *leaving* and initial *creation* of a VRG.

The *join* operation must be capable of adding a new *backup gateway* to the VRG without interrupting any ongoing tasks. As soon as the *join* is completed, the *master gateway* must start synchronizing its internal state via the available fieldbus connections. In order to avoid peak load situations, this can only be done over a longer period of time. Thus, in case of a failure at the *master gateway*, an incomplete synchronized *backup gateway* shall not be able to participate in

the election process and assume the role of the new *master* device. The *leave* operation is much simpler. Essentially, only a notification message has to be sent to all other *gateway* devices to cancel the VRG membership of the concerned device. After this message, the leaving *gateway* is no longer part of the VRG, and hence it is not synchronized or allowed to take part in *master* elections. A possible way of creating a VRG has been already proposed by this work. A single *gateway* device is powered up and subsequently elects itself as *master*. Then it is possible to expand the VRG by the means of the *join* operation.

Another open item for future work, is the procedure, how a faulted gateway leaves the VRG. The synchronization approach proposed by this work is rather simple and a more sophisticated mechanism could be taken into account. For the proof-of-concept *gateway* implementation, synchronization of an *Object* or *Variable* entity is only tried once via each available fieldbus interface. The conceptual *gateway* approach presented in this work proposes an unlimited amount of retries, not counting any retry mechanism deployed by the fieldbusses themselves. A more practical approach would be a kind of orphanage algorithm deployed at the *master gateway*. *Heart-beat* messages sent out by the *backup gateways* are not only used as an end-to-end communication channel monitoring, but also as keep-alive signals for the individual *backup gateways*. As soon as a single *backup gateway* device does not send a *heart-beat* message after a certain amount of time, it is kicked out from the VRG compound. Hence, an orphaned *backup gateway* needs to rejoin and trigger a full synchronization in case it wants to be again part of the VRG. A suitable sequence of messages and notifications need to be designed. Firstly, to update the membership list of the VRG at the *backup gateways*, and secondly to notifiy an orphaned *backup gateway*, when it starts to send *heart-beat* messages again.

Furthermore, a tool needs to be designed that allows the creation of the *information model* in a feasible way. The *information model* of the proof-of-concept information was manually filled with all necessary information. But this is a rather crude approach, and a more sophisticated tooling is strongly advised.

The presented work was able to show, that a horizontal integration between different fieldbusses utilizing the *gateway* approach is feasible and viable. Other solutions presented in Chapter 2 need either to deploy additional layers on top of the field network or lack a reliability concept. Furthermore, it is even possible to retain the level of fault tolerance for the field network with the weakest fault hypothesis by utilizing a replication approach. These open topics give rise to future work or refinement of the presented *gateway* approach.

List of Abbreviations

BACS building automation and control systems

BAS building automation systems

CARP Common Address Redundancy Protocol

CIM Computer Integrated Manufacturing

COM component object model

CSMA Carrier Sense / Multiple Access

DCOM distributed component object model

EHS European Home Systems Protocol

EIB European Installation Bus

ERP Enterprise Resource Planning

FCR fault-containment region

FFD Full Function Device

FSK Frequency Shift Keying

HA Home Automation

HBA home and building automation

HMI Human-Machine Interface

HTTP hypertext transfer protocol

HTTP Hypertext Transfer Protocol

HVAC Heating, Ventilating, and Air Conditioning

ICT information and communication technology

IP internet protocol

ISM Industrial-Scientific-Medical

ISO International Organization for Standardization

ISP Interoperable System Project

MES Manufacturing Execution System

MVC Model View Controller

MVP Model View Presenter

NAT Network and Address Translation

oBIX open building information exchange

OLE object linking and embedding

OPC OLE for process control

OPC UA OPC - unified architecture

OSI Open Systems Interconnection

PAT Port and Address Translation

PLC programmable logic controller

REST Representational State Transfer

RFD Reduced Function Device

SAP Service Access Point

SCADA Supervisory Control and Data Acquisition

SCADA Supervisory Control and Data Acquisition

SOAP simple object access protocol

SOC system on chip

UDP User Datagram Protocol

URI Uniform Resource Identifier

VRG virtual redundant gateway

VRRP Virtual Router Redundancy Protocol

XML extensible markup language

XML Extensible Markup Language

ZCL ZigBee Cluster Library

ZC ZigBee Coordinator

ZED ZigBee End Device

ZR ZigBee Router

Bibliography

[1] CAN Specification Version 2.0. Bosch, 1991.

[2] CANopen Application Layer and Communication Profile, CiA/DS301. CAN In Automation, 2000.

[3] OPC Classic - Security. , OPC Foundation, 2000.

[4] OPC Classic - Batch. , OPC Foundation, 2001.

[5] OPC Classic - Alarms and Events. , OPC Foundation, 2002.

[6] Digital data communications for measurement and control Fieldbus for use in industrial control systems. International Electrotechnical Commission, 2003.

[7] OPC Classic - Complex Data. , OPC Foundation, 2003.

[8] OPC Classic - Data Access. , OPC Foundation, 2003.

[9] OPC Classic - Data Exchange. , OPC Foundation, 2003.

[10] OPC Classic - Historical Data Access. , OPC Foundation, 2003.

[11] Building automation and control systems (BACS) - Part 2: Hardware. ISO, 2004.

[12] OPC Classic - XML Data Access. , OPC Foundation, 2004.

[13] IEEE 802.15.4 – Standard for Information technology – Local and metropolitan area networks– Specific requirements– Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs). IEEE, 2006.

[14] oBIX 1.0 - Committee Specification 01. OASIS, 2006.

[15] ZigBee Specification. ZigBee Alliance, Jan 2008.

[16] IEC 62541 - OPC Unified Architecture Specification. IEC, 2010.

[17] ZigBee Home Automation Public Application Profile. ZigBee Alliance, Feb 2010.

[18] ISA100.11a-2011. Wireless systems for industrial automation: Process control and related applications, 2011.

[19] EEBus Specification 1.0 beta 4. EEBus Initiative, 2015.

[20] Alena, R. and Gilstrap, R. and Baldwin, J. and Stone, T. and Wilson, P. Fault tolerance in ZigBee wireless sensor networks. In *Aerospace Conference, 2011 IEEE*, pages 1–15, March 2011.

[21] Attebury, G. and Ramamurthy, B. Router and Firewall Redundancy with OpenBSD and CARP. In *Communications, 2006. ICC '06. IEEE International Conference on*, volume 1, pages 146–151, June 2006.

[22] Avižienis, A. and Laprie, J.C. and Randell, B. and University of Newcastle upon Tyne. Computing Science. *Fundamental Concepts of Dependability*. Technical report series. University of Newcastle upon Tyne, Computing Science, 2001.

[23] Bautts, T. and Dawson, T. and Purdy, G. *Linux Network Administrator's Guide*. O'Reilly Series. O'Reilly Media, 2005.

[24] Dhillon, B.S. *Reliability in Computer System Design*. Ablex series in software engineering. Ablex Publishing Corporation, 1987.

[25] Farahani, S. *ZigBee Wireless Networks and Transceivers*. Elsevier Science, 2011.

[26] Felser, M. and Sauter, T. The fieldbus war: history or short break between battles? In *Factory Communication Systems, 2002. 4th IEEE International Workshop on*, pages 73–80, 2002.

[27] Galati, A. and Greenhalgh, C. A New Metric for Network Load and Equation of State. In *Networking and Services, 2009. ICNS '09. Fifth International Conference on*, pages 309–313, April 2009.

[28] G.G Wood. Survey of LANs and standards. *Computer Standards and Interfaces*, 6(1):27 – 36, 1987. Real-time Distributed Software.

[29] Gislason, D. *ZigBee Wireless Networking*. Elsevier Science, 2008.

[30] Jain, Sushant and Fall, Kevin and Patra, Rabin. Routing in a Delay Tolerant Network. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 145–158, New York, NY, USA, 2004. ACM.

[31] Jürgen Jasperneite. INTERBUS. In Richard Zurawski, editor, *Industrial Communication Technology Handbook, Second Edition*, chapter 15. CRC Press, Inc., 2014.

[32] Wolfgang Kastner, Stefan Soucek, Christian Reinisch, and Alexander Klapproth. State of the Art in Smart Homes and Buildings. In Richard Zurawski, editor, *Industrial Communication Technology Handbook, Second Edition*, chapter 59. CRC Press, Inc., 2014.

[33] Kastner, Wolfgang and Neugschwandtner, Georg. Data Communications for Distributed Building Automation. In Zurawski, Richard, editor, *Networked Embedded Systems*, page 29. CRC Press, 2009.

[34] KNX Association. EN 50090-1:2011: Home and Building Electronic Systems (HBES). Cenelec, 2011.

[35] Kopetz, H. On the Fault Hypothesis for a Safety-Critical Real-Time System. In Broy, Manfred and Krüger, IngolfH. and Meisinger, Michael, editor, *Automotive Software – Connected Services in Mobile Networks*, volume 4147 of *Lecture Notes in Computer Science*, pages 31–42. 2006.

[36] Kopetz, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011.

[37] Kopetz, Hermann and Ademaj, A. and Grillinger, P. and Steinhammer, K. The time-triggered Ethernet (TTE) design. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 22–33, May 2005.

[38] Lange, J. and Iwanitz, F. and Burke, T.J. *OPC: From Data Access to Unified Architecture*. VDE-Verlag, 2010.

[39] Lukas Krammer. *Dependability in building automation networks*. PhD thesis, Vienna University of Technology, December 2014.

[40] Lukas Krammer and Markus Klein and Johannes Kasberger and Wolfgang Kastner. A fault-tolerant Communication Scheme for Fire Alarm Systems based on KNX. In *KNX Scientific Conference*, November 2012.

[41] Lukas Krammer and Stefan Seifried and Wolfgang Kastner. A fault-tolerant Backbone for IEEE 802.15.4 based Networks. In *Proc. of the IEEE International Conference on Industrial Technology (ICIT 2014)*, February 2014.

[42] Mahnke, W. and Leitner, S.H. and Damm, M. *OPC Unified Architecture*. SpringerLink: Springer e-Books. Springer, 2009.

[43] R. Hinden. RFC3768 – Virtual Router Redundancy Protocol. URL: `http://www.ietf.org/rfc/rfc3768.txt`, Apr 2004.

[44] Rathje J. Der Feldbus zwischen Wunsch und Wirklichkeit. In *Automatisierungstechnische Praxis*, volume 39, pages 52–57, 1997.

[45] Sauter, T. The Three Generations of Field-Level Networks – Evolution and Compatibility Issues. *Industrial Electronics, IEEE Transactions on*, 57(11):3585–3595, Nov 2010.

[46] Sauter, T. and Soucek, S. and Kastner, W. and Dietrich, D. The Evolution of Factory and Building Automation. *Industrial Electronics Magazine, IEEE*, 5(3):35–48, Sept 2011.

[47] Schnell, G. *Bussysteme in der Automatisierungs- und Prozesstechnik: Grundlagen und Systeme der industriellen Kommunikation*. Praxis der Automatisierungstechnik. Vieweg+Teubner Verlag, 2013.

[48] Stefan Seifried, Lukas Krammer, and Wolfgang Kastner. A reliable and flexible KNX Gateway. In *KNX Scientific Conference*, October 2014.

[49] Syromiatnikov, A. and Weyns, D. A Journey through the Land of Model-View-Design Patterns. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 21–30, April 2014.

[50] Thilo Sauter. Fieldbus Systems: Embedded Networks for Automation. In Richard Zurawski, editor, *Embedded Systems Handbook, Second Edition: Networked Embedded Systems*, chapter 20. CRC Press, 2009.

[51] Thilo Sauter. Fieldbus System Fundamentals. In Richard Zurawski, editor, *Industrial Communication Technology Handbook, Second Edition*, chapter 1. CRC Press, Inc., 2014.

[52] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. College of Computer Science, Northeastern University Boston, 1995.

[53] Wang, C. and Jiang, T. and Zhang, Q. *ZigBee® Network Protocols and Applications*. CRC Press, 2014.

[54] Wolfgang Granzer and Wolfgang Kastner and Paul Furtak. KNX and OPC UA. In *Konnex Scientific Conference*, Nov 2010.

[55] Wolfgang Kastner and Georg Neugschwandtner and Martin Kögler. An open approach to EIB/KNX software development. In *Proc. 6th IFAC Intl. Conference on Fieldbus Systems and their Applications (FeT '05)*, pages 255–262 (preprints volume), nov 2005.

[56] Woo Suk Lee and Seung-Ho Hong. KNX-ZigBee gateway for home automation. In *Automation Science and Engineering, 2008. CASE 2008. IEEE International Conference*, pages 750–755, Aug 2008.

[57] Woo Suk Lee and Seung-Ho Hong. Implementation of a KNX-ZigBee gateway for home automation. In *Consumer Electronics, 2009. ISCE '09. IEEE 13th International Symposium*, pages 545–549, May 2009.