Efficient Communication in Control-Oriented Embedded Networks

Andreas Scholz, Irina Gaponova, Stephan Sommer, Alfons Kemper, Alois Knoll
Technische Universität München
{scholza,gaponova,sommerst,kemper,knoll}@in.tum.de

Christian Buckl
fortiss GmbH
buckl@in.tum.de

Jörg Heuer, Anton Schmitt
Siemens AG, Corporate Technology, Multimedia and Network Communication
{joerg.heuer,anton.schmitt}@siemens.com

Abstract—**In the recent years, wireless sensor networks (WSNs) have drawn a lot of attention and a lot of work has been done to provide an efficient communication infrastructure for these systems. This paper focuses on another -not so well studied- class of embedded networks: embedded networks used for control and automation purposes. In contrast to WSNs, these networks have a comparably stable infrastructure, with a low probability of node or link failure. The main challenge for the communication in control oriented networks is the heterogeneity of the underlying infrastructure and the resource constraints already known from WSNs. We propose an adaptable communication layer that leverages existing network protocols and at the same time provides a seamless communication over heterogeneous networks and an efficient and scalable network stack for individual nodes. We show the feasibility of this approach with a demonstrator for the smart energy management in a future home automation scenario.**

I. Introduction

Embedded networks (often also called sensor or sensor-actor networks) containing a multitude of networked nodes with varying sensing, acting, and processing capabilities are emerging in many application areas such as the automotive, building management, or factory automation sector. Due to differences in the underlying hardware (wired/wireless links, limited/unlimited energy resources, mobile/stationary nodes) different network architectures have been developed. Especially the area of wireless sensor networks (WSNs) has drawn a lot of attention in the recent years. We want to focus on another area of embedded networks: embedded networks for control and automation purposes.

The resource limitations known from WSNs also apply for control oriented networks, requiring mechanisms for a resource efficient execution of applications. Compared to WSNs, control oriented networks typically possess a well planned structure and nodes with a comparably high reliability. These networks therefore are rather stable. Node or link failures, either because of hardware failure or energy depletion, may still occur and have to be compensated, but their likelihood is rather low. A new challenge encountered in control oriented networks is heterogeneity. The increasing convergence of embedded networks and the heterogeneity of the used hardware and communication media, such as wireless links, Ethernet connections or bus systems, has led to the situation that different communication protocols are used in a single network. Often these protocols use different addressing schemes and have different characteristics. Given the resource constraints and the stable but heterogeneous network infrastructure, a developer is faced with a new challenge: the requirement to build an application that is capable of running on a heterogeneous infrastructure and at the same time is capable of adapting its execution to the characteristics of the hardware in order to provide an as efficient execution as possible. Because the development of individual solutions for every application field is too costly and time consuming, middleware solutions, which ease these optimizations, have to be developed.

In this paper we will focus on a fundamental part of these middleware solutions: the communication layer used to facilitate message exchange between different nodes in the network. We propose a communication architecture that allows leveraging existing communication protocols and provides a unified addressing scheme, a seamless communication over heterogeneous networks, and a plug-and-play like support for communication protocols that allows to choose the protocol best suited for the intended application. The proposed architecture is based on a modular design that allows tailoring the communication stack to an application's needs, leading to an efficient and scalable solution that is suitable for a broad range of devices. Additionally, the modularization allows exploiting features provided by the underlying network protocols or communication media, resulting in an implementation that is resource efficient and avoids redundant work.

The contributions of this paper are:

- an analysis of the communication requirements in heterogeneous embedded networks used for control and automation purposes;
- the development of adaptable and modular communication layer fulfilling these requirements.

In the following two sections, we take a closer look at the communication in control oriented networks and the resulting requirements for the communication infrastructure. In Sections 4 and 5, we propose a communication architecture, which is able to meet these requirements. In Section 6 we outline the key features of an efficient implementation of this architecture and describe a demonstrator that comprises a heterogeneous embedded network. In Section 7 we give an overview of related work and conclude the paper in Section 8.

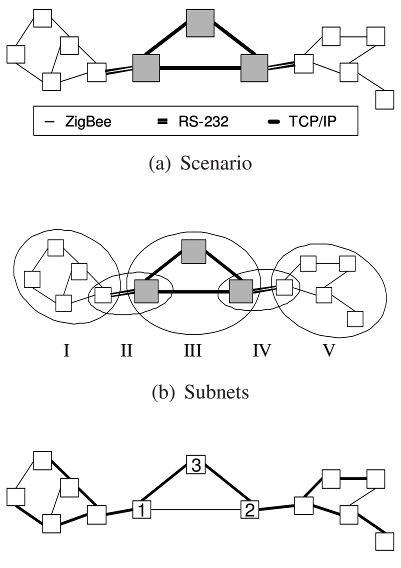
II. Communication in Control-Oriented Embedded Networks

Analogous to other distributed systems, the development of individual solutions for every single embedded network is too costly and time consuming. This has led to the development of several generic approaches targeting different application scenarios. A common denominator of these approaches is the optimized execution of applications w.r.t. the underlying network structure: data is not routed to a central component which does all the processing, but instead the application logic is distributed in the network, avoiding the creation of bottlenecks and single points of failure. This distributed data processing is based on a data stream oriented processing paradigm, i.e., data is generated by sources (typically sensor devices), processed and transformed by intermediate components (logic components) and ultimately consumed by data sinks (actors, end-user devices, databases, etc).

One class of systems are database-like query interfaces for embedded networks, such as TinyDB[1] or Cougar[2]. In these systems a declarative query (often formulated in an extended SQL dialect) is decomposed into operators which are spread throughout the network to perform the acquisition, processing and dissemination of data. Because embedded networks are mainly deployed for monitoring or control purposes, ad-hoc requests are rare. Instead continuous monitoring and operation based on the measured data is important. As a consequence, the operators generated by the declarative queries often operate on window based parts of the data streams produced by the measurement devices in the network. These systems can therefore be seen as data stream processing systems, containing operators which perform in-network data processing.

Another class of systems are service oriented middleware architectures for embedded networks, such as the ϵSOA[3] project, the DPWS[4] based SIRENA[5] and SOCRADES[6] projects, or the OASiS[7], MORE[8], or RUNES[9] projects. A driving factor for the development of these systems is the complexity of the application development for embedded networks. Often a multitude of heterogeneous devices has to be integrated into a platform that allows an efficient and distributed execution of applications under the presence of resource limitations, real-time requirements or limited energy resources. To cope with this complexity and to provide reusable and interoperable solutions, a decomposition of applications into small encapsulated building blocks (services) is employed by these systems. In order to provide an efficient data processing and to allow energy saving options, these services typically operate on an event based processing paradigm, i.e., the execution of the service logic is triggered by incoming data which may produce new data that is submitted to the next service in the chain. Just like the query system, these systems can be seen as data stream processing systems, too.

The non-functional requirements for individual data streams can be very different. For some data streams, such as periodic measurements, reliable data transfer is not required or even not useful, because re-transmitted packets are outdated when they arrive at the receiver. Other data, such as an alarm event generated by fire sensor, have to be transmitted reliably and perhaps w.r.t. some maximum transmission delays.



(a) Scenario

(b) Subnets

(c) Data Flows

Fig. 1. Heterogeneous Embedded Network

The communication layer provided by an embedded network middleware should therefore be able to efficiently support the transmission of this streaming data and at the same time be scalable to the non-functional requirements of individual data streams.

III. Requirements for Efficient Data Exchange in Embedded Networks

Given the characteristics described in the previous section, the communication layer in an embedded network has to fulfil several requirements. First the network protocol used for communication in the embedded network has to be exchangeable. If the intended application scenario changes or a better suited protocol becomes available, the communication layer has to be re-configurable in order to adapt to these new circumstances.

The second requirement is the support of heterogeneous networks comprising several different protocols. A typical example for this scenario is depicted in Figure 1(a). It comprises a backbone of nodes connected via Ethernet and multiple sensor nodes connected via ZigBee to the backbone. In this scenario, three network types with different protocols are used, as shown in Figure 1(b): ZigBee for the communication between the sensor nodes (Subnets I and V), a RS-232 interface for the communication between the sensor nodes and the backbone nodes (Subnets II and IV), and TCP/IP for the communication between the backbone nodes (Subnet III). We will use the term "subnet" to refer to such clusters of nodes that communicate with the same network protocol. Assume the developer wants to run a simple monitoring application that calculates average values over the measurements of all sensor nodes. A possible solution is illustrated in Figure 1(c): Node 1 is used to calculate the average for the left ZigBee subnet, Node 2 for the right ZigBee subnet. Node 3 finally is used to calculate the average of the intermediate results produced by Node 1 and Node 2. The corresponding data flows are shown as thick lines in Figure 1(c). During application development the developer should not have to worry about the message conversions required to facilitate a communication between the individual nodes, but should be able to work on an abstract view of the network that hides the underlying protocols and communication media. Nevertheless a majority of the communication will occur between nodes in the same subnet and should not be hindered by a too complex abstraction layer with high overhead. Finally, the special characteristics of the underlying networks, such as bandwidth restrictions, have to be passed on to upper layers to allow an optimization of the placement of services and to avoid overload situations.

A third challenge is the heterogeneity of the involved networks. Some of the used networks may already offer services such as reliable transmissions, either through a reliable communication medium or a reliable transport protocol, whereas others do not. We will refer to these capabilities as "features". A possible solution is to use an overlay network that offers all required features. The major drawback of this approach is the inefficient resource usage. If some features are already implemented by an underlying protocol the overhead of a re-implementation should be avoided. Additionally, not all nodes may require the same features. Consider a network comprising sensors which are used for long-term monitoring and do not require a reliable data transport. At the same time the network contains a fire detector which has to reliably send a message to an alarm if a fire is detected. Given this scenario the network stack on the temperature sensing nodes does not have to provide reliable transport, but the stack on the alarm and the fire detector has to. To provide an efficient communication in this scenario, the network stack used on individual nodes has to be adaptable. It should provide the features required by the services running on the node, but no additional features if the provisioning of these incurs an overhead. Additionally, features already available in the underlying network should be exploited and not re-implemented by the network stack. An interesting area for future research is an energy-aware (re-)configuration of the network stack. If a node has low energy resources, it might be beneficial to disable some features, e.g., reliable transport, in order to increase the lifetime of the node.

The data transmitted by the communication layer can be grouped into two classes: á-priori known data transfer and ad-hoc communication. The latter category of transmissions is communication for which there is no further information available, often caused by administrative messages which cannot be predicted. The former is created by the data streams flowing between services on different nodes. An important property of data streams is that the source and sink of these streams is known á-priori. Often the data rates are known, too. Because data streams constitute a large amount of the overall transmitted data, the fourth requirement is to exploit this information to optimize the routing of data streams in the network. This optimization should also take into account the topology of the underlying network and the characteristics of nodes and links. With this information, many overload situations can be avoided through a topology aware placement of services and routing of data streams.

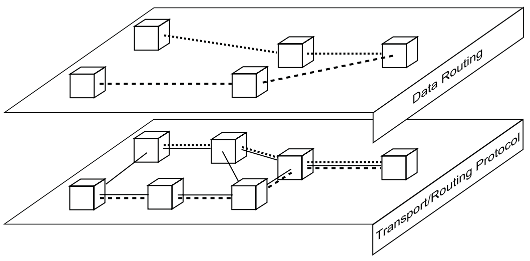


Fig. 2. Layered Routing

Another difficulty that has to be handled by the communication layer of an embedded network is fault tolerance. The link quality between nodes can vary over time and link or node failures can occur, especially if wireless communication and battery powered devices are used. The communication layer should support a graceful degradation in these scenarios, i.e., alternative routes between nodes should be used to ensure the application continues to work. This recovery has to be done fully automatically and in a very timely fashion. The alternatives chosen during this process may result in a reduced overall performance, e.g., because the backup routes are much longer than the original ones. If the error persists over a longer time, the communication layer should generate a notification that allows reorganizing the placement of services w.r.t. the new network topology.

Summing up, the communication layer for embedded networks should provide the following features:

- easy addition and removal of network protocols;
- support for communication over heterogeneous subnets;
- efficient communication within a subnet;
- an adaptable network stack that can be tailored to application needs and exploits features provided by underlying protocols;
- efficient transport of data streams;
- fault tolerance;
- notification of topology changes.

IV. Layered Routing

The routing in the embedded network has to be designed in a way that allows combining two contradicting goals mentioned in the previous section: a topology aware routing of data streams, and failure tolerance. A trade-off has to be found that allows controlling the data flows in the network and at the same offers enough flexibility to change the routing of packets to avoid congested or broken links. These goals and the requirements listed in the previous section can be achieved with a layered routing approach. The basic idea of this approach is depicted in Figure 2. The *Data Routing Layer* handles the high level distribution of data streams and messages in the network. It specifies the sink nodes of data streams and provides functionality to split data streams at

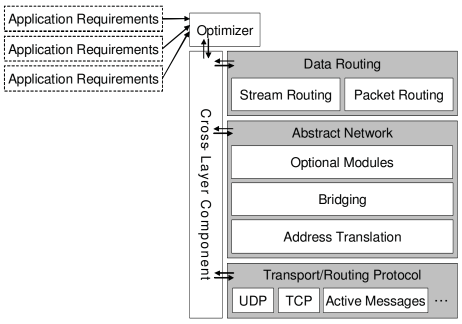


Fig. 3. Architecture of an Adaptive Network Stack

nodes. Note that the source and sink of data streams do not have to be neighbouring nodes, but can be any node in the embedded network, even nodes from different subnets. A data stream only specifies on a high level that data has to be transmitted from one node to another. The transmission of individual packets of the data stream and the selection of a suitable route in the underlying network is handled by the *Transport/Routing Protocol Layer*. This layer relies on one of the various network protocols available for embedded networks to perform the transport of data between two nodes.

The layered approach has several benefits: first the Data Routing Layer does not have to deal with the clustering of nodes or different network protocols in the subnets but provides the developer with a high level overview of the data transmissions in the network. It shows the data flows in a way that is understandable for an application domain expert and safely hides details regarding the underlying network protocols. Second the high-level routing of data can be controlled through the routing of the data streams, whereas the low-level routing of packets and the failure recovery is handled transparently by the underlying network protocols. A cross-layer-component can be used to allow optimizations in both layers. The data routing can be optimized by changing the placement of services w.r.t. the network structure in order to minimize transmission costs and delays. The routing of packets can be improved based on the characteristics of the data streams generating the packets.

V. Adaptive Network Stack

The network stack presented in this section is based on two principles: modularity and re-use. A major design goal was to create a communication layer that introduces as little overhead as possible compared to existing network protocols. At the same time, the functionality of this stack should be scalable, i.e., small nodes with little processing capabilities should not be burdened with functionality not needed by the services running on this node. This can be achieved by a modular network stack that allows tailoring its functionality by adding or removing features depending on the needs of the services running on a node. To provide a resource efficient implementation of this stack, as many features of the underlying network protocol should be re-used and not re-implemented in upper layers of the network stack.

The modular network stack shown in Figure 3 is a refinement of the layered architecture developed in the previous section. It is based on an *Abstract Network* layer which is suited on top of the protocols used in the subnets. This layer provides a unified addressing scheme across all networks and provides modules for features not present in the underlying protocols. Based on the Abstract Network, two routing components are implemented: the *Stream Routing*, which is optimized for the transmission of continuous data streams and the *Packet Routing*, which is optimized for the transmission of single packets. Additionally a *Cross-Layer Component* provides access to information gathered by the different layers, allowing the *Optimizer* to adapt the configuration of the network stack to the needs of a given application. The individual layers are described in detail in the following sections.

A. Transport/Routing Protocol Layer

The bottom layer of the stack provides access to the transport or network protocols used in the different networks, such as the UDP and TCP protocols used in IP based networks, the Active Messages used in TinyOS, the RS-232 interface for serial data transmission, etc. The minimum requirement for protocols that should be incorporated in the network stack is support for a unicast end-to-end communication between nodes. Besides this basic functionality, many protocols offer additional features, such as multicast support, reliable transport, encryption, QoS guarantees, etc. These features are stored for every protocol. During the installation of an application, an Optimizer determines which of the features required by an application can be provided by the underlying protocols, and which features have to be provided through the installation of an additional module. The Cross-Layer Component allows the transport/routing protocols to publish topology information and link characteristics and to access application level information such as the data rates of streams, etc. The former information can be used to optimize the placement of services in the network. The latter can be used by the transport/routing protocol to optimize the routing of packets in the network.

B. Abstract Network Layer

The bridging functionality of the Abstract Network Layer resembles the functionality of network stacks known from overlay networks, such as Peer-to-Peer networks. However there is an important difference: The *Bridging* component only handles the message routing across network boundaries. A message sent within a subnet will be transmitted directly via the underlying transport protocol. Therefore the Abstract Network is not a full-fledged overlay network but can be seen as a thin wrapper that allows communication across heterogeneous subnets. The rationale for this decision are performance considerations. If communication occurs within a subnet, the Abstract Network introduces no additional overhead because all messages are sent directly via the underlying network protocol. If a packet is addressed at a node in another subnet, the packet is sent to a bridge node which converts the packet and injects the new packet in the other network.

The *Address Translation* provides a unified addressing scheme across all subnets. It uses n-bit network addresses, which comprise a network identifier (the first m bits) and a node identifier (the remaining $n - m$ bits). The number of bits used for addresses, n, and the distribution of these bits between the network and node part can be chosen during the development of the sensor network. This allows to reduce the network header size for small installations and support scenarios with multiple different subnets.

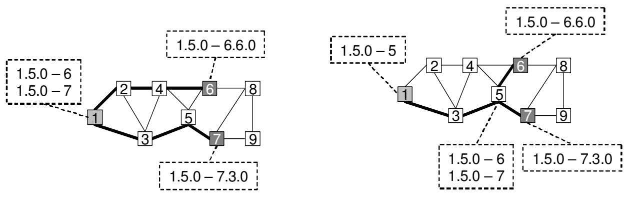
The Abstract Network currently supports two address assignment schemes: static addresses and dynamic configuration. Static addresses are assigned during development. This allows creating very compact and small images for devices, which are not added to networks dynamically, e.g., switches, static sensors or similar devices. The second address assignment scheme is based on a DHCP-like dynamic configuration mechanism. New nodes request an address by sending a broadcast message after installation. A coordinator node handles these requests and creates unique addresses for new nodes.

Optional modules can be plugged into the Abstract Network layer to provide features not present in the underlying protocols. Consider an application requiring a reliable message transmission between two services. If this application is running on top of an TCP/IP network, the TCP protocol already provides this feature out of the box. If the underlying protocol supports only unreliable transmissions, e.g., UDP/IP, an additional reliability module is installed in the network stack at the sender and at the receiver. The optional modules can be used in two ways: to provide new features for the communication inside a subnet and to provide new features for an end-to-end communication across heterogeneous subnets.

The modules are organized as a stack, i.e., on the sender side the message is passed to all modules in a top-down, on the receiver side a in bottom-up manner. This allows modules that are located higher up in the stack, to treat modules lower down in the stack as black boxes. Assume an application requires a connection between two services, which offers reliable transport and data encryption. In this case the reliability module is installed below the encryption module in the network stack. At the sender, the encryption module is called first, which can encrypt the message payload. Subsequently the reliability module is called, which can extend the message header with data needed to perform the reliable transport, e.g., a sequence number. The modules on the receiver side will be invoked in reverse order, i.e., first the reliability module and second the encryption module. If a transmission problem occurs, e.g. through a lost packet or a duplicate packet, this situation will be handled transparently by the reliability module and will not be visible to the encryption module.

C. Data Routing Layer

The previous sections covered the lower part of the network stack and how a transmission of single messages can be performed efficiently. Based on this functionality, the network stack offers two routing components, a *Stream Routing* component and a *Packet Routing* component. As mentioned during



(a) No Stream Sharing (b) Stream Sharing

Fig. 4. Data Stream Rounting

the requirements analysis, most of the data transmitted in an embedded network is stream based, i.e., the source and sink of the streams are known á-priori. A crucial observation is that often data from one source is consumed by multiple sinks. Consider a temperature sensor which provides data that is recorded for long term monitoring and simultaneously used by a heater and an air condition. Such an example scenario is depicted in Figure 4(a). The data produced by node 1 is used by two other nodes, 6 and 7. The thick lines indicate the two resulting data streams routed through the network. A possible optimization which reduces the network utilization is shown in Figure 4(b). Instead of sending two distinct data streams, a single stream is sent to node 5, where it is split into two streams targeted at nodes 6 and 7. The detection of such synergies and the calculation of such splitting points is the task of the Optimizer which is not in the focus of this paper.

The routing of data streams is performed by the Stream Routing component of the network stack. A Data Stream Router, which runs on top of the Abstract Network Layer, is installed on every node in the network. It contains a routing table which indicates the target nodes for messages stemming from a specific sender. Data streams are identified by triples: the node address, instance id and port number of the service that produces the data stream. The dashed boxes in Figure 4(a) and 4(b) show the routing tables for the example. Every row contains one routing entry, the part on the left side of the "-" is the stream identifier, i.e., the source address, the part on the right side is the target address. Like the source address, the target address consists of a node address, an instance id and a port number. Because the instance id and port number are only needed at the target node, the Stream Router stores only the node address for remote services. The routing table of node 1 in Figure 4(a) can be read as: transmit the data produced by service instance 5 port number 0 to nodes 6 and 7. At node 6 the routing table specifies that the data should be processed by service instance 6 port number 0. The shared use of the data stream produced by node 1 can be achieved with the routing tables shown in Figure 4(b). Instead of two different streams, node 1 only creates a single stream that is targeted at node 5. Because the stream identifier is contained twice in the routing table, the stream is splitted at node 5 and sent to both nodes, 6 and 7. The routing tables at nodes 6 and 7 remain unchanged.

The routing of the remaining messages is done by the Packet Routing component. It offers a simple interface that allows sending data to a designated target identified by an address triple containing the node address, instance id and port number.

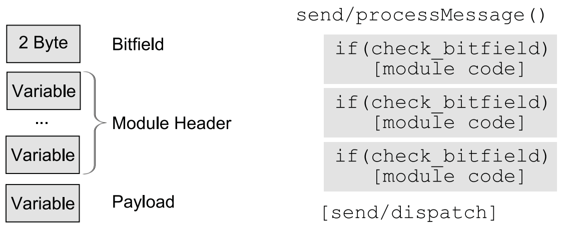
D. *Message Routing Schemes*

Individual messages can be routed using one of two paradigms: source based routing or target based routing. Using the former paradigm, messages are routed depending on the data source that created them. Each intermediate node has to be configured prior to the transmission of messages. This is done by adding a route that tells the node where to send data stemming from a specific originator to the Stream Routers on all intermediate nodes before the data transmission starts. Using the latter paradigm, the target of a message is contained in the message itself, so no additional configuration is required at intermediate nodes. For data stream routing with á-priori known routes (which we assume for control oriented networks), these two paradigms have similar efficiency for a unicast communication pattern. If a message has to be delivered to multiple recipients, the two routing schemes behave differently.

If the underlying protocol supports multicast, the Stream Router at the originating node can be configured to send data via multicast to all sink nodes. Intermediate nodes for the splitting of streams are not needed in this case. In order to achieve this communication, the nodes have to join corresponding multicast groups what requires a configuration prior to the transmission of messages. In this case both, target and source based addressing, require a pre-configuration of nodes and therefore a comparable management overhead.

If the underlying network offers no multicast support, intermediate nodes, which split the data stream, have to be used. In this case source based addressing has a big advantage: messages remain unchanged throughout the whole transmission in the network. At a splitting node, the message is simply duplicated and routed to all destinations. For target based addressing there are two possible solutions, which both incur an overhead. One possibility is to pre-configure splitting nodes to send messages to multiple destinations. But in contrast to source based addressing, the messages have to be modified at these nodes to add the new targets. The other solution is to add all targets to the message. In this case the message can remain unchanged throughout the transmission. The drawback is that the message size grows and the parsing becomes more complex. Summing up, source based routing is beneficial if a message should be transmitted to multiple destinations and the underlying network supports no multicast transport.

We assume that for control oriented applications the one-to-many communication pattern will occur often for data streams. Since our approach has to be efficient in different networks, supporting multicast or not, we chose source based routing for the implementation of the Stream Routing interface. Our reasoning is that the source based routing incurs less processing costs for individual messages but requires more management overhead before the start of a transmission, whereas the opposite is true for target based routing. Because data streams comprise a huge amount of messages (potentially infinite) and are changed seldom, the additional management overhead is negligible compared to the performance gains for the processing of messages. Another benefit of source based addressing is the possibility to seamlessly switch to multicast transport if it is supported by the underlying network. On the other hand, management and configuration messages sent through the network do not show a predictable behaviour like data streams and do not require support for stream splitting. The Packet Routing interface therefore uses a target based addressing scheme.



```
send/processMessage()
    if(check_bitfield)
       [module code]
    if(check_bitfield)
       [module code]
    if(check_bitfield)
       [module code]
  [send/dispatch]
```

(a) Message Format (b) Message Handling Code

Fig. 5. Modular Message Handling

E. *Message Handling*

In the ϵSOA project, we promote a model based development approach for embedded networks[10]. It allows generating executable code and reconfiguration scripts for the individual nodes in an embedded network based on a model of the available hardware and the applications that should run on this hardware. The communication layer proposed in this paper fits seamlessly into such a development environment, as the communication requirements imposed by individual applications can be automatically derived from the models and be used to create and configure an adapted network stack for individual nodes. Nevertheless, the benefits of the proposed communication stack can also be exploited in non model-based environments, e.g. by using configuration files to specify the communication requirements of individual nodes.

To support the optional modules in the Abstract Network Layer, some additional information has to be added to the message header. Figure 5(a) shows the corresponding message format. The first two byte of the message header contain a bitfield that denotes which modules should process the message. After these bytes, the optional modules can add module specific information to the message header, e.g., a sequence number for reliable message transfers.

The corresponding message handling code is shown in Figure 5(b). During code generation, a conditional code block, denoted with gray boxes in the figure, is added to the message handling routines. At the beginning of every code block it is checked whether the corresponding module should be executed or not. For incoming messages on the receiver side this is determined by the bitfield from the message header. If the module is executed, it has to extract its module specific part of the message header and increment a position marker that stores the current offset in the message header. This marker allows subsequent modules to correctly determine their header parts. During execution, the module has the possibility to stop the processing of the message, e.g., if an unexpected or invalid message is detected. The code on the sender side is similar, but the order of the modules is reversed. For every data connection in the system, the user can specify which modules should be used. This information is passed as bitfield to the message handling code on the sender side to trigger the conditional execution of the modules. If a module changed the message, typically by adding information to the message header, it also sets its module bit in the bitfield of the message header in order to trigger the execution of the module on the receiver side. If no action is taken by the module it may leave the message unchanged (e.g. a fragmentation module can take no action if the message size is small enough to fit in a transport package). Note that the activation of the modules on the sender side can be configured in a fine-grained per-connection basis, i.e., not every data sent by a node has to use all modules present at the node. This allows supporting different kinds of connections, e.g., reliable and non-reliable connections, on a single node.

VI. Implementation

We have implemented the network stack for ZigBee based motes and PCs. We implemented some basic modules to demonstrate the feasibility of the modularization. Some of the modules are very simple and should be seen as a technological demonstration, modules used in a productive environment will most likely be more complex. The supported modules are:

Reliability Module: the Reliability Module supports a simple per-packet reliable transport mechanism. Outgoing packets are assigned with a unique sequence number and stored at the sender. The receiver submits an acknowledgement for every received packet. The next packet of the stream is sent after the acknowledgement of the previous packet is received. If the acknowledgement is not received in a configurable period of time, the packet is treated as lost and resubmitted by the sender.

Fragmentation Module: the Fragmentation Module is required if the underlying network protocol is packet based and a payload that exceeds the packet's capacity should be transmitted. The Fragmentation Module will break the message into pieces that fit into the network packets and reassemble the original payload at the receiver.

Encryption Module: providing secure communication and authentication mechanism for embedded networks is an open research area. The resource constraints on the devices often prohibit the use of public key infrastructures and asymmetric encryption algorithms known from other distributed systems. Often it is impossible to store long cryptographic keys on the nodes or perform complex calculation required by algorithms such as RSA. The Encryption Module should be seen as a demonstration of how encryption support can be added to the network stack, once a suitable mechanisms has been developed. It supports a simple stream cipher and is based on the assumption that a symmetric key pair is installed on each node and a trusted coordinator in the network. The coordinator authenticates new nodes through a challenge response mechanism and generates session keys for the communication between nodes in the network.

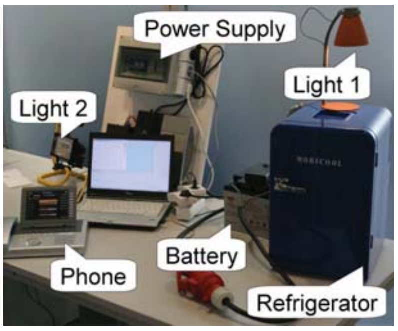


Fig. 6. Smart Home Demonstrator

Based on the ϵSOA platform and the communication layer described in the previous sections, we developed a demonstrator, which covers a future home automation scenario. The assembling of our demonstrator is shown in Figure 6. We assume that in the near future energy providers use dynamically changing energy prices in order to influence the overall energy consumption in a way that smoothes load peaks. We further assume that some kind of power storage system, such as the battery of an electric car, is present in future homes. We implemented the following scenario: a household comprising a battery and loads (a refrigerator and 2 lights) with different power consumption and energy saving options. One task of the automation logic is to minimize the energy costs throughout the day. If prices are cheap, the battery is charged and the refrigerator cools down to a lower threshold. If prices are high, the house is disconnected from the power grid and draws its energy from the battery. Additionally, the refrigerator is put to energy saving mode, i.e., it stops cooling until an upper temperature threshold is reached.

The demonstrator comprises three different networks: an IP network comprising the laptop (attached via W-LAN), the Phone (attached via Ethernet) and a modified FritzBox that acts as a bridge to the embedded network. The second network is the RS-232 connection between the FritzBox and a mote, and the third network is the ZigBee based communication between the different motes. Due to the bridging support provided by our communication layer, services running on the PC and the FritzBox (which are running a Java based version of the middleware) are able to seamlessly communicate with services running on the motes (using a nesC version of the middleware). An example of such a communication is the monitoring application running on the Laptop that shows the temperature of the fridge and the state of the lights. Another example is the re-configuration of service parameters, like the temperature thresholds of the fridge, which can be performed at the Phone shown in the bottom left part of the figure.

VII. Related Work

The goal of creating a converged communication architecture supporting different protocols is also persued by other projects. The Sensor Network Protocol (SP) [11][12] is intended to provide a "narrow waist", i.e., an anchor point around which the rest of the architecture evolves, just like the

IP protocol does for the Internet. In the case of SP, this anchor is a best-effort single-hop broadcast communication primitive.

Based on SP, the authors of [13] propose a modular network layer. Based on a decomposition of monolithic protocol implementations into small interoperating modules, this layer provides a higher re-usability of code and eases the implementation of new protocols.

The Rime[14] network stack extends the best-effort single-hop broadcast communication primitive known from SP with additional more complex primitives. The stack supports different link-layer protocols through a header transformation module and allows cross-layer information exchange through message attributes which are passed on to the different layers of the Rime stack.

The above mentioned projects and the communication layer presented in this paper, aim at overcoming the difficulties introduced by the multitude of different communication protocols encountered in embedded networks. However, this is done at different levels. The projects mentioned in this section target wireless sensor networks and are intended to unify and ease the development of communication protocols optimized for individual application fields. This approach is promising for a well defined hardware environment, such as the WSNs used in these projects. In contrast to this, the communication layer presented in this paper is targeted at heterogeneous networks comprising fundamentally different subnets, such as Ethernet links, bus-systems, wireless networks, etc. It is very unlikely that a single unified communication protocol can be found that can be implemented efficiently for every single subnet. As a consequence an abstraction layer, like the one proposed in this paper, is required that allows a seamless and efficient communication across these networks.

Despite these differences, there are a lot of synergies between the WSN oriented approaches and the higher-level abstraction layer proposed in this paper. First, our abstraction layer can benefit from the cross-layer information provided by the Rime network stack. Second, we can benefit from the diversity of network protocols provided for WSNs, allowing a developer to choose the most suitable protocol. A promising vision for future embedded network communication architecture is a combination of all approaches in order to create a communication layer that autonomously adapts to the applications running in the network. At the beginning, this network uses a simple communication paradigm like the one provided by SP to facilitate communication between nodes. If new applications and new constraints, e.g., reliability requirements, have to be supported by the system, these are at a first step provided by additional modules installed on the corresponding nodes. Depending on these constraints, an optimization component can decide whether it is beneficial to change the communication paradigm and as a consequence de-install some of the modules which provide functionality redundantly available by the new paradigm. These optimizations do not have to be restricted to single protocols, but could also span the simultaneous use of multiple protocols optimized for different application classes. As an ultimate vision, one could imagine the automatic creation of tailored communication protocols based on the application requirements. The abstract network layer would then act as a bridging component between the protocols in the individual subnets and provide support for end-to-end guarantees over heterogeneous subnets, e.g., end-to-end reliable transport from a subnet with reliable transport to a subnet without reliable transport mechanisms.

VIII. Summary and Ongoing Work

In this paper, we proposed a communication architecture based on a modular design. It allows leveraging existing communication protocols and provides a unified addressing scheme and a seamless communication over heterogeneous networks. The proposed architecture can be adapted to the requirements imposed by different applications and the capabilities of the underlying hardware leading to an efficient and scalable solution that is suitable for a broad range of devices. We showed the feasibility of the approach for a demonstrator in a home automation scenario comprising several different networks.

We are currently investigating how a model-based development paradigm and a specification of application requirements can be used to create self-tuning embedded networks that adapt their communication layer to the requirements, especially timing constraints, of the running applications.

References

[1] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *TODS*, vol. 30, no. 1, 2005.

[2] Y. Yao and J. Gehrke, "The Cougar Approach to In-Network Query Processing in Sensor Networks," *SIGMOD Rec.*, vol. 31, no. 3, 2002.

[3] A. Scholz, C. Buckl, S. Sommer, A. Kemper, A. Knoll, J. Heuer, and A. Schmitt, "ϵSOA – service oriented architectures adapted for embedded networks," in *INDIN'09*, 2009.

[4] Devices Profile for Web Services, "http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf."

[5] F. Jammes and H. Smit, "Service-oriented Paradigms in Industrial Automation," in *IEEE Transactions on Industrial Informatics*, vol. 1, 2005, pp. 62–70.

[6] L. de Souza, P. Spiess, D. Guinard, M. Khler, S. Karnouskos, and D. Savio, "SOCRADES: A Web Service Based Shop Floor Integration Infrastructure," *IOT'08*, 2008.

[7] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits, "OASiS: A Programming Framework for Service-Oriented Sensor Networks," in *COMSWARE'06*, 2007.

[8] MORE – Network-centric Middleware for Group communication and Resource Sharing across Heterogeneous Embedded Systems, "http://www.ist-more.org/."

[9] P. Costa, G. Coulson, C. Mascolo, G. P. Piccoand, and S. Zachariadis, "The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems," in *PIMRC'05*, 2005.

[10] C. Buckl, S. Sommer, A. Scholz, A. Knoll, and A. Kemper, "Generating a Tailored Middleware for Wireless Sensor Network Applications," *SUTC*, 2008.

[11] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A unifying link abstraction for wireless sensor networks," in *SenSys '05:*, 2005.

[12] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao, "Towards a sensor network architecture: lowering the waistline," in *HOTOS'05*.

[13] C. T. Ee, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica, "A modular network layer for sensornets," *USENIX OSDI*, 2006.

[14] A. Dunkels, F. Österlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *SenSys '07*, 2007.