Universität Karlsruhe (TH)
Fakultät für Informatik
System Architecture Group

Fabian Knittel

Study Thesis

Scalable Source Routing in the AmbiComp Environment

Tutors: Johannes Eickhold, Pengfei Di
and Björn Saballus
Registration date: October 30, 2008
Submission date: April 30, 2009

Contents

1 Introduction **3**

2 The AmbiComp platform **5**
2.1 Hardware 5
2.2 Software 6
2.3 Build environment 7
2.4 Integration of the SSR protocol 7
2.4.1 Interaction 7
2.4.2 Environment 9

3 Introduction to ssr-core **11**
3.1 Important components 11
3.1.1 The node 11
3.1.2 Routing 11
3.1.3 Timers 13
3.1.4 Messages 14
3.2 State machine 14

4 Changes to ssr-core **17**
4.1 Avoiding heap memory and the STL 17
4.1.1 SSR messages 18
4.1.2 SSR payload messages 18
4.1.3 SSR timer events 19
4.1.4 NIC addresses 20
4.1.5 Interface table 20
4.1.6 Routing table 21
4.1.7 Neighbour table 21
4.1.8 Source paths 21
4.1.9 Maximum node 22
4.2 Reducing and choosing table sizes 22
4.2.1 Interface table 23
4.2.2 Source paths 23
4.2.3 Routing table 23
4.3 Exceptions 25
4.3.1 Errno value 25
4.3.2 Exceptions from constructors 25
4.4 Run-Time Type Identification 26
4.5 Removing inlining 26

4.6 Code renovation 27

5 The new ssr-core library **29**
5.1 Code configuration 29
5.1.1 Abstract cNode class 29
5.1.2 Compile-time options 30
5.2 State machine 31
5.2.1 State SSRConstr 32
5.2.2 State SSRIsoMax 32
5.2.3 State SSRMax 32
5.2.4 State SSR 33
5.2.5 State SSRShutd 34
5.2.6 State SSRDestr 34
5.3 Class diagrams 34
5.3.1 cNode 34
5.3.2 cRouteCache 35
5.3.3 BitFieldArray 35
5.3.4 Events 36
5.3.5 cAddr 36
5.3.6 Interfaces 37
5.3.7 NeighborTable 38
5.3.8 MaxNodeAnnounceStore 38
5.3.9 NicAddr 39
5.3.10 Paths 39
5.3.11 Messages 40
5.3.12 cNodeEnumeration 42
5.3.13 cStaticPool 42

6 Evaluation **43**
6.1 Unit tests 43
6.2 Automated simulation tests 44
6.3 Running on the AVR platform 45

7 Conclusion and future work **47**
7.1 Conclusion 47
7.2 Safety in the wild 47
7.2.1 Errors in constructors 47
7.2.2 Length limitation on source paths 48
7.2.3 Limited number of physical neighbours 48
7.2.4 Table sizes need reality check 48
7.3 Integration with the AmbiComp environment 48
7.3.1 Link-layers 48
7.3.2 ACVM 49

A Source tree and build environment **51**
A.1 Files and directories 51
A.2 Build targets 54

List of Figures

2.1 Ambient Intelligence Control Unit (AICU) consisting of several sandwich modules. . . . 5
2.2 Overview of the AmbiComp software stack. . . . 6
2.3 SSR in the AmbiComp software stack. . . . 7
2.4 Interactions with SSR. . . . 8

3.1 A virtual ring of node addresses [Fuh05, fig. 1]. . . . 11
3.2 An example source path. . . . 12
3.3 An example routing table. . . . 13
3.4 Simplified state machine, documenting the states of an SSR node. . . . 14

5.1 Users of the *ssr-core* library need to sub-class the abstract *Node* class. 29
5.2 State machine, documenting the states of an SSR node. . . . 31
5.3 Class hierarchy of class cNode. . . . 34
5.4 Class hierarchy of class cRouteCache. . . . 35
5.5 Class hierarchy of class cBitFieldArray. . . . 36
5.6 Event class hierarchy. . . . 36
5.7 Class hierarchy of class cAddr. . . . 37
5.8 Interfaces class hierarchy. . . . 37
5.9 Class hierarchy of class cNeighborTable. . . . 38
5.10 Class hierarchy of class cMaxNodeAnnounceStore. . . . 38
5.11 Class hierarchy of class cNicAddr. . . . 39
5.12 Path class hierarchy. . . . 40
5.13 Message class hierarchy. . . . 41
5.14 Class hierarchy of class cNodeEnumeration. . . . 42
5.15 Class hierarchy of template class cStaticPool. . . . 42

6.1 Message passing within the OMNeT++ simulation framework. . . . 44
6.2 SSR test application in the partial AmbiComp software stack. . . . 45

List of Tables

4.1 Shows which timer events were embedded in which class. 19
4.2 Structure of a single entry in the routing table before space optimisation. 24
4.3 Structure of a single entry in the routing table after space optimisation. 25
4.4 Code and data sizes of ssr-core in bytes – before and after removing excessive inlining. 27

5.1 Reflexive edges on the SSR state. 30
5.2 Abbreviations used within the state machine. 32
5.3 Reflexive edges on the SSRIsoMax state. 33
5.4 Reflexive edges on the SSRMax state. 33
5.5 Reflexive edges on the SSR state. 34

Statement of authorship

I hereby certify that this study thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged.

Karlsruhe, April 30, 2009

Chapter 1

Introduction

The AmbiComp project [EFS+08] envisages thousands of small, self-organised, embedded devices, all scattered through-out a building or even a whole neighbourhood. The devices only provide very limited amounts of RAM and CPU processing power. The project's vision is to let the devices communicate transparently with each other, forming a potentially large distributed system.

The project assumes some kind of underlying networking protocol, but does not yet provide or specify one. At the time of writing, the project uses direct network- or link-layer access without the envisaged transparency, automatic routing or scalability features.

The main focus of the project is to help software engineers to cope with the above scenario, by providing a development environment that allows easy development and deployment of applications. In addition to the development environment, a matching software and hardware stack is provided. One of the main building blocks of the software stack is the Ambient Computing Virtual Machine (ACVM), a highly optimised runtime environment for transcoded Java bytecode.

The Scalable Source Routing (SSR) protocol [FDKC06] is a routing protocol designed for message and memory efficient routing in large unstructured networks. It provides routing based on virtual addresses (key based routing). The virtual addresses form a ring and allow the routing of messages with only small amounts of routing state, making SSR very scalable. In contrast to other protocols (e.g. Chord [SMK+01]), the SSR protocol provides these features without depending on a separate network layer, which greatly improves the protocols efficiency.

In summary, the SSR protocol was designed for environments which closely match the environments assumed by the AmbiComp project. An integration of the protocol implementation into the AmbiComp's software stack would therefore provide a key building block to complete the network transparency aspects of the AmbiComp project.

This study thesis aims at taking a first step towards this integration. It focuses on the following topics:

- Analysis of the restrictions imposed by AmbiComp's software stack.
- Analysis of the old SSR protocol implementation, regarding its code quality, extensibility and portability.
- Preparation of the implementation's code for future integration with the AmbiComp stack.

- Porting the protocol implementation to AmbiComp's hardware platform.
- Documentation of the new SSR protocol implementation and its design decisions.

The paper is structured as follows: Chapter 2 provides an overview of the hardware and software stack of the AmbiComp project. In addition, it briefly discusses how the SSR protocol fits into the software stack and which aspects of this integration are within the scope of this work and which are not. Chapter 3 introduces the SSR protocol implementation *ssr-core*. It concentrates on a high-level view, presenting the larger concepts which remain unaffected by the changes introduced by the thesis. Based on these introductory chapters, chapter 4 identifies problems with the existing SSR implementation and discusses solutions to these problems. In chapter 5, the new, refactored SSR protocol implementation is detailed and explained. The new implementation is evaluated in chapter 6, based on automatic testing and a test implementation for the target environment. Finally, chapter 7 describes unresolved issues within the new implementation and other future work.

Chapter 2

The AmbiComp platform

This chapter provides an overview of the AmbiComp project's hardware and software stack and the specific limitations imposed by them. In addition, it explains where and how the Scalable Source Routing should be integrated into the software stack and which aspects of that integration are within the scope of this work and which are not.

2.1 Hardware

The AmbiComp project developed the concept of an AICU (Ambient Intelligence Control Unit) [EFS+08, p. 1]. AICUs can serve as flexible building-blocks and therefore ease the development of products which intend to make use of the features promoted by the AmbiComp project.

One AICU consists of one or more sandwich modules (SMs). The SMs are connected via a backplane. There are different types of SMs, each with distinct capabilities, such as network connectivity, generic I/O interfacing or acting as the power supply for the AICU. Figure 2.1 shows an example configuration.

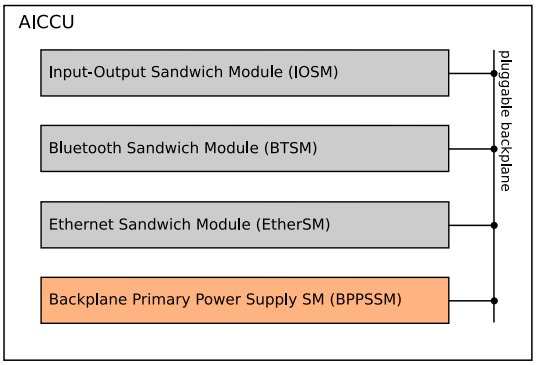

Figure 2.1: Ambient Intelligence Control Unit (AICU) consisting of several sandwich modules.

For the purpose of this thesis, we only need to take note of the "intelligent SMs", meaning all SMs with a CPU. (In the above AICU example, all SMs apart from the BPPSSM have a CPU.)

The CPU used at the time of writing is the *Atmel AVR ATmega2561 RISC processor* [EFS+08, p. 2]. It features clock speeds between 7.37 MHz and 16 MHz and *8 KiB of internal SRAM*. SMs have *up to 512 KiB External SRAM*. Persistent storage is provided in the form of *256 KiB of flash memory*.

All intelligent SMs run an instance of the AmbiComp software stack, specifically configured for the purpose of the respective SM and the purpose of the AICU.

The AICUs described here are currently the only *hardware* available for the AmbiComp project, so the capabilities and limitations present in the hardware described, directly affect the integration of the SSR protocol stack.

The only alternative run-time environment is the development environment on Linux, based on the Linux-BIOS and the Linux-ACVM. Please see the next section for more details.

For more details on the hardware, please see [EFS+08, p. 3f].

2.2 Software

The software stack is divided into three layers (see figure 2.2).

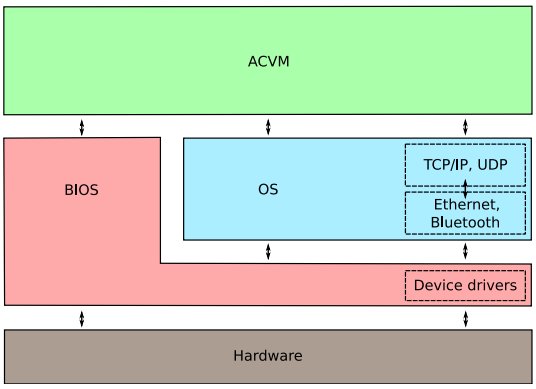

Figure 2.2: Overview of the AmbiComp software stack.

The lowest layer of the software stack, the BIOS, provides basic hardware abstraction. It interfaces with the underlying hardware and hides most system specific issues behind a common set of interfaces. It also contains the hardware specific parts of device drivers.

The so-called OS layer solely consists of a collection of libraries providing network stacks, e.g. TCP/IP or the various Bluetooth protocols. The network stacks can be enabled or disabled, so for certain SMs, the OS layer may be completely empty.

The upper layer, the Ambient Computing Virtual Machine (ACVM), runs the actual application code. It has no direct contact with the underlying hardware. Applications are developed in Java and specially transcoded to reduce the size of the resulting byte code.

For all layers apart from the ACVM, no regular heap memory is available.

The Linux-BIOS is a special BIOS variant. Instead of interfacing with real embedded hardware, it emulates the BIOS API on a regular Linux system. Together with a Linux-ACVM, this allows for convenient development and testing without AICUs.

2.3 Build environment

While the application developer solely produces Java code and interacts with Java interfaces, the underlying layers, especially the OS- and BIOS-layers are written in C code (with a few seldom occurrences of assembly). For the AVR platform, they are compiled using the GNU C cross-compiler for Atmel AVR. As C library, the AVR Libc[1] is used, which provides a subset of the standard C library. (For the Linux development platform, the regular GNU C compiler and GNU Libc is used.)

2.4 Integration of the SSR protocol

As the SSR protocol provides networking functionality, its obvious place is within the OS layer, together with the other network stacks (compare figure 2.3). In this position, it is directly accessible by the ACVM and at the same time, has access to other network stacks for use as link layers.

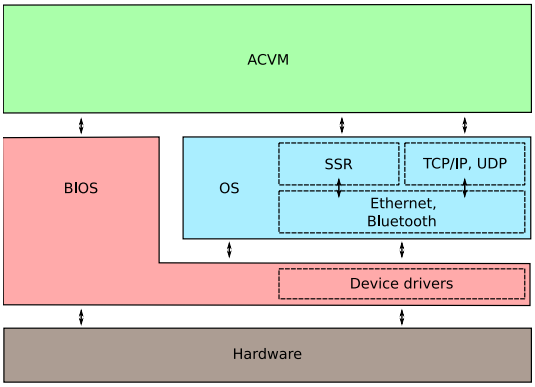

Figure 2.3: SSR in the AmbiComp software stack.

2.4.1 Interaction

The necessary interactions with the SSR implementation can be roughly grouped into four categories (see figure 2.4 for an overview).

Link layer

The SSR library expects to get passed-in packets which were received by the link-layer via `HandleFromNic()`. Each link-layer packet signifies an SSR message.

The library expects to be able to send out packets via the link-layer to physical neighbours via direct addressing (`SendToNic()`) or broadcasting (`Broadcast()`).

The network stacks present within the OS-layer present no common interface. I.e. the library providing the capability to send packets out via an Ethernet interface is quite different from the library providing the various Bluetooth protocols. Therefore, a

[1] See `http://savannah.nongnu.org/projects/avr-libc/`

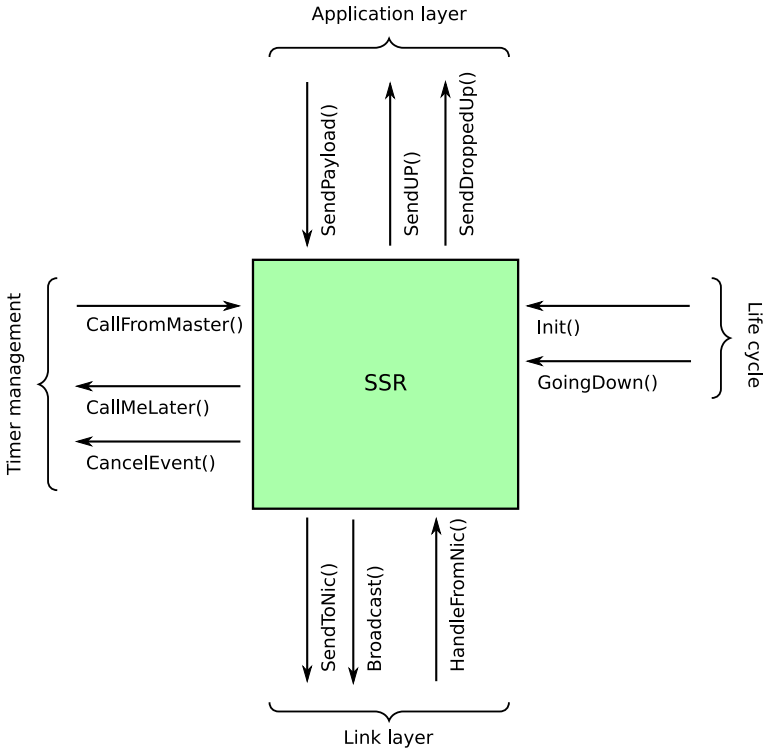

Figure 2.4: Interactions with SSR.

simple adapter needs to be written per link layer, providing a way to send and receive SSR packets via that layer.

SSR only provides a single entry point for sending and receiving link-layer packets. So to support more than one link-layer at a time, a multiplexer/demultiplexer needs to be created.

Neither the creation of a common interface, nor the creation of a multiplexer/demultiplexer for access to multiple link-layers is part of the work presented here.

Application layer

To send messages, the application layer can pass down a block of payload data and an SSR destination address (`SendPayload()`).

When the SSR library receives a message intended for the local node, it will pass up the data to the application layer (`SendUp()`). It also informs the application layer of messages that couldn't be delivered (`SendDroppedUp()`).

For the AmbiComp software stack, the ACVM is the "application layer". Either a Java native interface needs to be built, to send messages to SSR nodes from within Java applications. Or, alternatively, to support Globally Accessible Objects [SEF08] or similar mechanisms providing transparent communication to Java applications, the ACVM needs to make use of SSR directly.

Neither approach is within the scope of this paper.

Timer management

SSR expects to be able to register (`CallMeLater()`) and unregister (`CancelEvent()`) timer events, allowing call-backs into the SSR code (`CallFromMaster()`) to occur after a requested amount of time.

The BIOS provides only a single timer event. Therefore the more sophisticated timer management expected by the SSR implementation needs to be provided elsewhere. It can either be adapted from timers provided within the ACVM or by a newly developed timer library called from within the ACVM's event queue.

A simple timer library was developed in the course of this study thesis, which allows to be driven by the single BIOS timer event. The final integration with the ACVM remains as future work.

Life-cycle management

The SSR implementation needs to initially register a few timers and announce its presence to its physical neighbours via the link layer. This is done via the `Init()` call, which should be called when all other layers within the OS layer have been initialised.

The `GoingDown()` call performs the opposite operation. It announces the shutdown of the node to its physical neighbours and halts any running timers. Although the SSR protocol has means to detect nodes which disappear silently, explicitly calling `GoingDown()` should be preferred for efficiency reasons.

2.4.2 Environment

The SSR library instance needs to cope with the limitations presented by AmbiComp's hardware stack and build environment. Specifically

- the library's code and static data need to fit into the flash memory,
- the run-time memory foot-print needs to fit into the internal or external RAM,
- the library needs to be compilable by the build environment currently used within the AmbiComp project.

Analysing these limitations and preparing the SSR implementation to overcome them, is the main focus of this study thesis.

Chapter 3

Introduction to ssr-core

This chapter reveals the inner workings of the SSR black-box shown in the previous chapter and introduces the main building blocks of the library.

The library implementing the SSR protocol is called *ssr-core*. It is written in C++ and was developed in parallel to the evolving specification of the SSR protocol.

3.1 Important components

3.1.1 The node

The main component within *ssr-core* is the abstract node class. One instance of the node class represents a single SSR node. The node class provides all entry points and hooks presented in chapter 2.4.1, i.e. it serves as the main entry point into the library. The outgoing hooks are implemented as purely virtual methods. Users of the library are expected to sub-class the node class and implement the purely virtual methods.

3.1.2 Routing

Each node has two types of addresses: virtual and physical addresses. A node has exactly one virtual address, but potentially several physical addresses: one physical address per available link-layer network interface.

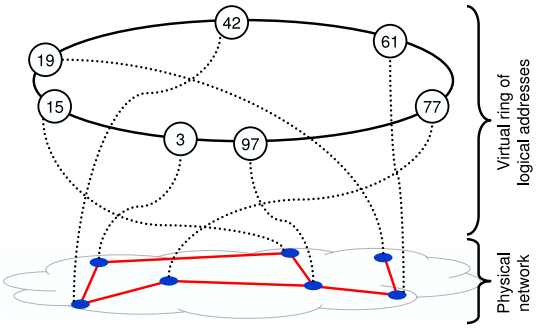

Figure 3.1: A virtual ring of node addresses [Fuh05, fig. 1].

Based on the virtual addresses, the nodes in an SSR network form a virtual ring (figure 3.1). Each node has two virtual neighbours: a predecessor and a successor in the ring.

The list of available physical neighbours (and their respective virtual and physical addresses) is stored in the *InterfaceStore* class. It allows the routing of messages from the local node to a node that is a direct physical neighbour.

Messages that need to be sent beyond the node's physical neighbours potentially need more work: If the message contains a source path in its header which has a valid next hop, it is forwarded to that hop. For messages that originate from the local node or who's source path does not describe a valid next hop, the node's local routing table is used to determine a new path.

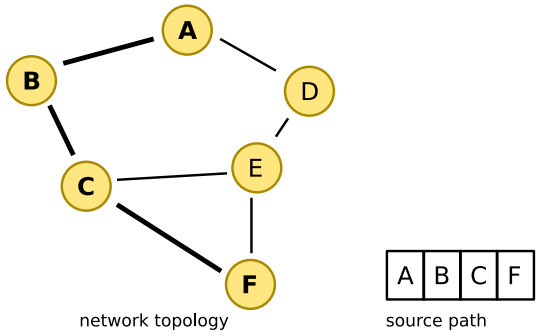

Figure 3.2: An example source path.

Source paths are a list of virtual addresses, which define the routing path of a message. The path is initially created by the sending node, but it is potentially amended or changed by nodes forwarding the message. Forwarding nodes also use source paths from received messages to update their knowledge of active nodes and routes in the network.

The forwarding nodes need no stored knowledge about valid routes to the messages' destination node. In figure 3.2, the source route contains the hops A, B, C and F. For the routing to work, only node A needs to know of the route initially. All forwarding nodes (B and C) can forward the message solely on the basis of the source route (prepared by node A) and the knowledge of their direct physical neighbours.

The *RouteCache* class provides the routing table, which consists of a fixed-size list of SSR node addresses and links between those addresses. The routing table forms a tree of nodes with additional auxiliary links, representing the known routes between nodes.

The routing table seen in figure 3.3 shows a possible routing table for node A. Routing a message from node A to node F is performed by looking up the destination node F in the table. In the example, the node F is located at table entry 5. Following the uplink indices, the source route is created step-by-step in reverse order: Node F's uplink is entry 3, which is node C. Node C's uplink is node B in entry 1. Finally, node B's uplink is the originating node. At that point the route is completed and just needs to be reversed to serve as a source route for the message. The resulting source route matches the one from figure 3.2.

The alternative parent uplinks shown in the routing table allow alternative routes to be stored.

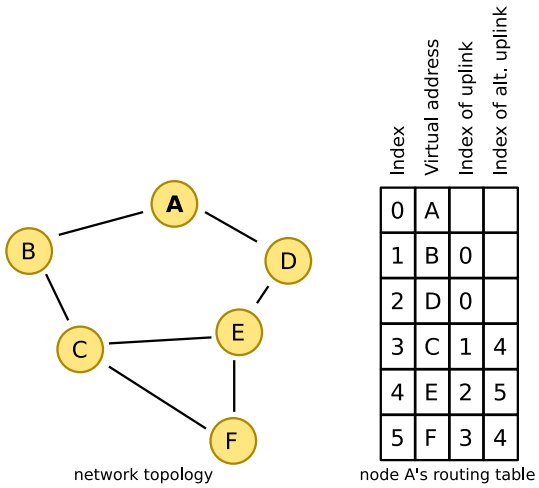

Figure 3.3: An example routing table.

The routing table contains at least the routes to the node's virtual predecessor and successor. Assuming a correctly and fully formed virtual ring, a message can be routed to its destination simply by passing it along the virtual ring. Solely using this approach would lead to very inefficient routing.

By storing paths to additional nodes in the routing table, the average routing step and path length is significantly reduced. In the optimal case, the stored paths should lead to nodes with exponentially increasing virtual distance to the local node. This allows for routing steps to be, on average, within $O(\log N)$, for a network consisting of N nodes [Fuh05, p. 244].

In case the routing table does not contain a direct route to the intended destination, the message is instead forwarded to the node with the virtually closest address and the shortest hop count. These routing steps repeat until the intended destination is reached or the closest virtual address determines, that the destination node does not exist.

3.1.3 Timers

The *ssr-core* library uses timers to cope with node churn within the network:

- The broadcast timer event (*EventBroadcast*) causes the node to regularly broadcast its existence to all physical neighbours.
- The interface[1] timeout event (*EventIfTimeout*) exists once per physical neighbour. The timer's timeout is reset in the case of activity from the physical neighbour. In case of a timeout, the physical neighbour is assumed to have vanished (e.g. because it has lost power or because it has moved out of radio range) and is removed from the *InterfaceStore*.
- The notification timer (*EventNotification*) causes the node's virtual neighbours to be contacted. This keeps the routing table up-to-date regarding the paths to

[1]The term "interface" currently stands for a single physical neighbour. Originally, there were more interface types apart from the remaining point-to-point one and the "interface" term was more justified.

the virtual neighbours. If the neighbour table feature is activated, the notification message also includes a copy of the local neighbour table. This further improves the robustness of SSR in case of node churn. (See 5.3.7 for further details on neighbour tables.)

See section 5.3.4 for details on the timer events' implementation.

3.1.4 Messages

All communication between SSR nodes takes place in the form of SSR messages. All messages have a common header, which indicates the message's type.

The *MsgPayload* message transports the actual data from the upper application layer. All other message types are responsible for maintaining the SSR network, e.g. by announcing new nodes, invalidating specific routes, etc.

See section 5.3.11 for details on the messages.

3.2 State machine

The simplified state machine in figure 3.4 shows the states in which an SSR node can reside in. Additionally, it shows the state transitions. The transitions are caused by the incoming function calls `Init()`, `GoingDown()`, `CallFromMaster()` and `HandleFromNic()`.

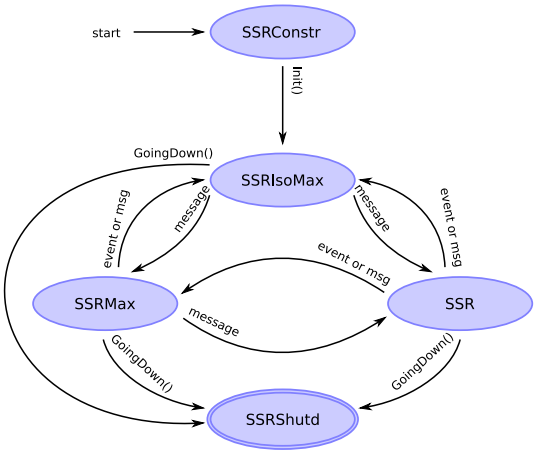

Figure 3.4: Simplified state machine, documenting the states of an SSR node.

The states implicitly result from the contents of the node's routing table and interface store. They are not explicitly modeled within the library.

On `Init()` an SSR node changes from state *SSRConstr* to state *SSRIsoMax*. At this point, all timers have been set and the node has announced its existence to the direct physical neighbours. It does not yet know of any neighbours and therefore remains isolated. Additionally, due to its isolation, the node's address is the highest known address in the (empty) virtual ring.

The the node's shutdown is caused by calling `GoingDown()`, in which case the state changes to the final state *SSRShutd*, regardless of the previous state. Any physical neighbours are now informed about the node's removal and all timers are disabled.

The events shown on the state diagram are timer events and are triggered by the `CallFromMaster()` function, which is the call-back for timers that were previously registered by *ssr-core* and have timed out. The only timer event relevant for state changes is the interface timeout event.

The messages shown are SSR messages, received via `HandleFromNic()`. Any SSR message routed through the SSR node can implicitly change the state of the SSR node. This is due to updates to the routing table from source paths stored within the SSR messages. The SSR messages *explicitly* changing the node's state are *MsgHello*, *MsgKill*, *MsgRouteUpdate*, *MsgNeighborNotification* and *MsgMaxNodeAnnounce*.

During regular operation, the SSR node is either in state *SSRMax* or in state *SSR*, depending on whether it has the largest address in the virtual ring or not[2]. Interface timeouts and *Kill* messages can lead back to the nodes isolation (*SSRIsoMax*) or it could turn the node into the node with the largest address (*SSRMax*). The introduction of new nodes might lead to a change from state *SSRMax* to state *SSR*, if one of the new nodes has a higher virtual address.

Detailed information regarding the state machine is given in section 5.2.

[2]Specifically, it checks whether the node's virtual successor has a *lower* virtual address than the local node.

Chapter 4

Changes to ssr-core

This chapter describes the changes applied to the *ssr-core* library in the course of the study thesis. The changes can be roughly categorised into two groups: One group of changes was made necessary by the fact that the library needed to be ported to a new platform. The other group of changes had to do with general code maintenance. The changes in the former group will be thoroughly documented in this chapter. The changes in the latter group will be presented in far less detail, as they were mainly implementation work without much academical value.

As described in chapter 2, the library needed to be ported to the AVR platform and its build environment. Before this porting effort, the *ssr-core* library had been integrated and tested with simulation frameworks like *OMNet++* [1] and run on regular desktop PCs. In addition, for the *Linyphi* project [DEF08], the implementation had been ported to small Linux-based routers running on *MIPS* processors [DEF08, p. 685]. Compared to the targeted AVR platform, both previous environments provided relatively large amounts of RAM and had build environments without restrictions to the available tool-set. To make the code-base usable within AmbiComp's OS layer and on the AVR platform, it needed changes regarding the used C++ feature-set, the memory usage and memory management and its code size.

The following sections will analyse which aspects of the implementation needed changes. They will then develop possible solutions for each aspect and finally present the chosen solution.

4.1 Avoiding heap memory and the STL

The SSR protocol stack will reside within the OS layer of the ACVM. The OS and BIOS layers don't provide regular `malloc()` and `free()` functions. Instead, allocations on the stack, static allocations or specific memory assignments at link-time have to be used. So the use of `malloc()` and `free()` needs to be removed and dynamic memory allocations should be replaced or avoided as much as possible.

In the following sub-sections, central structures or repetitive patterns using dynamic memory allocations will be analysed. Based on the analyses, approaches for replacing the dynamic allocations with storage on the stack or static allocations will be presented. Every sub-section will conclude with the approach chosen for the new *ssr-core* implementation, when configured for the AVR platform.

[1] `http://www.omnetpp.org/`

In addition to the memory allocation restrictions, the AVR platform does not provide any C++ libraries. Specifically, the STL is not included and including it would cause the code-size to increase dramatically. Therefore, any code using STL constructs is analysed and approaches for replacement are discussed.

4.1.1 SSR messages

Problem

The SSR message classes represent the protocol's messages sent over the network between SSR nodes (e.g. payload messages, hello messages, etc.). The classes can serialise and deserialise themselves.

All SSR messages are created dynamically. Their life-time is restricted to calls into the SSR core, i.e. they get created during an SSR core method call and get released before its return. So in theory, many dynamic allocations could be replaced by placing the objects on the stack.

Unfortunately, messages retrieved (i.e. deserialised) from the network are turned into objects by a factory method. In addition the (optional) neighbour table distribution requires SSR message cloning functionality. Both features aren't easily transferred to static or stack-based allocations.

Without an extensive design change, the only viable approach is one, that emulates dynamic memory allocation without using a regular heap. The allocation mechanism would only need to provide space for a very small number of messages and could therefore work from a statically allocated buffer of fixed size.

Solution

The new implementation follows the dynamic memory allocation approach. It does this by using a `cStaticPool<>`-based `cMessagePool`, which is a statically sized and allocated message pool, providing N blocks of equal size for the storage of N messages[2]. The syntax of message allocations is unchanged, as the pool is connected to the base message class through `operator new` and `operator delete` methods. The latter aspect kept the amount of work needed to implement the change very low.

4.1.2 SSR payload messages

Problem

The payload messages are the messages containing the actual data, i.e. they transport the data sent by the layer above the SSR protocol. Due to the large payload buffer, these messages are – by far – the largest messages among all SSR messages and would thus waste a lot of memory in the `cMessagePool`, which was proposed as solution above.

As mentioned in section 4.1.1, a message's life-time is restricted to call into the SSR core. While processing a payload message, the SSR core does not create an additional one, so no two payload messages will be handled at the same time. Therefore a global static payload buffer would be able to replace the dynamically allocated buffers.

[2]The block size is equal to the largest message size. The largest message size is determined at compile-time, using a list of all message classes in MessagePool.cc. Space for 2 messages is reserved, as *ssr-core* only processes one incoming message at a time and creates a maximum of one additional message as response, i.e. never needs space for more than 2 messages at a time.

Alternatively, instead of creating a private copy, the storage could also be avoided completely by reusing the original buffer containing the payload.

Solution

The latter approach was used in the new implementation. The payload buffer pointer was modified to point to a constant buffer, i.e. the library modifies the data and applications which get handed the payload may not modify it either. This allowed the payload message class to reuse the raw incoming buffer from the callee, simply pointing from within the message instance into the raw buffer. The approach completely avoids any buffer copying and therefore also avoids the memory allocation problem.

In case the message comes from the network, the payload pointer points into the network buffer. In case it comes from the application, the pointer refers to the buffer supplied by the application. The pointer's constness assures, that no modifications are applied.

4.1.3 SSR timer events

Problem

As mentioned previously, SSR's timer event system uses event class instances to regularly send broadcasts, to watch out for physical neighbour timeouts (called interface timeouts) and to keep its virtual neighbours updated.

The event objects are created dynamically and get passed out of SSR core as transparent pointers (via `CallMeLater()`). They therefore have a life-time that exceeds SSR core method calls, ruling out the possibility of using stack-based storage.

For all timer events in SSR, there is an object that has a matching life-time: The `cEventNotification` and `cEventBroadcast` events have a life-time matching that of the local node. The `cEventIfTimeout`'s life-time matches that of its associated physical neighbour, i.e. the `cPointToPointInterface`. A straightforward approach could therefore be to make them member objects of the object who's life-time matches theirs. Thereby the problem of allocating event objects would be reduced to the already existing problem of allocating their container objects.

Solution

The above approach was implemented, by first modifying the event classes to be embeddable within other classes. Specifically, they needed to be made assignable by receiving correct assignment operators. (Allocating the events on the heap and simply storing pointers within their new storage locations would have defeated the whole purpose of the approach.)

Next, the dynamic event object allocations were replaced by embedding the objects, as shown in table 4.1.

Timer event	Embedded in
`cEventBroadcast`	`cNode`
`cEventNotification`	`cNode`
`cEventIfTimeout`	`cPointToPointInterface`

Table 4.1: Shows which timer events were embedded in which class.

In addition, the timer event system needed a way to inform the external event loop of timers that should be disabled. Therefore the new call-back `CancelEvent` was introduced. To ease the event management, an additional `void*` handle property was added to the events, to allow the external event loop to store a local identifier.

4.1.4 NIC addresses

Problem

SSR uses a small data block to transparently represent local physical addresses within the SSR core. SSR does not interpret the NIC address internally, i.e. they are only meaningful to the caller. The data blocks are copied into dynamically sized and allocated buffers.

As the physical address size typically depends on the used interface types (e.g. 48 bit Ethernet hardware addresses or 64 bit ZigBee addresses) and we will know the used interface types at compile-time, the dynamic allocations can be replaced with buffers of fixed size.

Solution

In the new implementation, the pointers to dynamic memory were replaced with `cNicAddr` wrapper objects which can use an internal, embedded buffer of static size to store the address (if configured to use the `cStaticNicAddr` storage back-end, see section 5.1.2). The class is implemented as a value type (i.e. the instances can be copied or assigned similar to built-in C++ types), so all dynamic memory usage can be avoided.

4.1.5 Interface table

Problem

The interface table maps SSR node addresses (i.e. virtual addresses) of all known physical neighbours to a `cPointToPointInterface` instance, which contains further information about the neighbor (e.g. the link-layer address). The table uses a `std::map` without any size limitations. This needs to be replaced with an approach not depending on the STL and not requiring heap memory.

Any non-dynamic approach unavoidably restricts the number of physical neighbours stored in the table. Therefore a mechanism needs to be designed, that decides what to do with newly detected neighbours while the neighbour table has no space left.

A complete analysis of the possible approaches is not within the scope of this thesis.

Solution

The ported *ssr-core* library uses a fixed-size list of interfaces instead of the `std::map`, aiming for simplicity in implementation. The interface size is defined at compile-time. No replacement strategy is used, so interface announcements will be dropped if more interfaces are active than allowed for at compile-time.

4.1.6 Routing table

Problem

The routing table is an array of SSR nodes, containing information about parent / child relationships and when each entry was last used in a routing decision.

The routing table is well prepared for the necessary conversion to static memory allocation, as all memory is allocated via the `new[]` operator at construction time and can be easily replaced with a embedded arrays of fixed size.

Solution

All involved routing table member types are made embeddable and the allocated arrays are replaced with arrays of fixed size, which are embedded within the routing table class (`cStaticCacheCore`). The size, i.e. the number of routing entries, is defined at compile-time (see section 5.1.2).

4.1.7 Neighbour table

Problem

The neighbour table storage has no dynamic elements and does not use the STL. The implementation is incomplete though (see 5.3.7), so this code section could be made optional.

Solution

The code supporting the neighbour table feature is enclosed within the preprocessor symbol `SSR_WITH_NEIGHBOR_TABLE` and disabled for the AVR target to minimise the code-size.

4.1.8 Source paths

Problem

Source paths consist of a list of virtual node addresses that can be traversed to reach a certain target node. Within *ssr-core* such paths are stored in `cPath` objects.

The `cPath` objects are used on the stack, as return values and embedded in message objects. The life-times of the reference-counted paths is very diverse.

The old `cPath` implementation stores the path in a dynamically allocated and reference-counted memory buffer, allowing for unrestricted path lengths.

Assuming that a maximum path length can be determined at compile-time, a buffer of fixed size could be used for the path.

Ignoring the existing reference-counted approach, this buffer could be embedded directly within the `cPath` object. The benefit would be that out-of-memory (OOM) conditions could be handled easier: The allocations would already fail when allocating the `cPath` instance instead of later on, within the `cPath` constructors. An additional benefit would be, that the required code for the embedded buffer approach would be very simple.

As `cPath` objects are passed by value with-in *ssr-core*, the obvious down-side to the above approach would be the frequent copying of the path lists, possibly causing

performance problems. Another problem could be the increased memory usage due to multiple copies of the same path using up memory.

A different approach is to continue supporting the reference counted approach. Instead of dynamic allocation from the heap one could use a central fixed-size pool of fixed-size source path lists which could then be shared between all `cPath` objects. This would avoid the frequent copying, but could lead to out-of-memory conditions due to an insufficient pool size, i.e. due to more paths allocated than available in the pool.

Any solution not assuming fixed-length paths would require far more complex memory management functions or the use of dynamic memory provided by the ACVM.

Solution

For the new implementation, the reference-counted approach with fixed maximum path lengths was chosen to optimise for speed and memory usage, while assuming well-chosen path lengths and pool sizes. The fixed-size source paths are stored in a statically allocated `cStaticPool<>`.

It remains unclear whether a fixed-length path structure is a viable long-term solution.

4.1.9 Maximum node

Problem

To avoid disjoint rings, a node who's virtual address is higher than its successor's (i.e. has the highest address in the virtual ring) needs to broadcast this information using `cMsgMaxNodeAnnounce` messages [CF05, sec. 2.3]. The messages are flooded through-out the SSR network.

On receiving such a message, the virtual address of the message originator (i.e. of the node that initially sent the message and has the highest virtual address) is stored in a `std::set`. If the same node address already exists in the set, the message won't be processed.

This prevents the node from processing the same information multiple times and stops the message from being endlessly flooded back and forth through the network.

An approach to get rid of the `std::set`, would be to use a static queue of the last L maximum node announcements.

Solution

To resolve the problem, the maximum node announcement storage was split out into a class that uses a queue of fixed length to keep track of the last L maximum node announcements.

4.2 Reducing and choosing table sizes

As the hardware used by the AmbiComp project has limited memory resources, all large data structures need to be analysed, so that their size can be reduced as far as possible.

Some structures already have a default size in the old *ssr-core* implementation. It will be determined whether that size can be reduced.

Additionally, a few data structures were previously dynamically sized. Therefore, appropriate static sizes for these previously dynamically sized structures need to be found.

4.2.1 Interface table

Depending on which interface types are compiled in and the expected device density, the optimal interface table size will vary widely.

E.g. a wired ethernet interface connected to a switched network can theoretically have an almost unlimited number of physical neighbours. Estimating a useful interface table size is very difficult in this case and will have to be based on an assumed deployment environment (i.e. expected number of neighbours in the specific environment).

The number of physical neighbours of a wireless interface is easier to foresee, as there are restrictions on the radio's range and (e.g. for Bluetooth piconets) on the number of simultaneous communication partners.[3]

Depending on the interface types involved, the assumed number of neighbours will always be arbitrary. A one-size-fits-all value seems unlikely, so this parameter will need to be adjusted depending on the device's intended environment.

During past simulations performed by Pengfei Di with the SSR protocol, nodes had no more than 20 physical neighbours at a time, so 20 will be used as a default table size for AmbiComp's implementation.

More tests should be performed to determine typical average numbers of physical neighbours depending on node density, to allow educated guesses for a specific environment.

4.2.2 Source paths

The SSR protocol and the AmbiComp project both assume a random, distributed Ad-Hoc sensor network. In these types of networks, the worst case node topology, where all nodes are chained together in a long line, is assumed to be very unlikely. Even a network forming a unit disk graph induces potentially long average paths: $O(\sqrt{N})$.

According to [Fuh05, p. 248], small-world network scenarios have far shorter average path lengths. Assuming such a scenario, it is still unclear what specific values to choose, especially during earlier phases of the network setup boot-strap.

The actually necessary sizes will need to be determined by further simulations. The implementation now reserves space for path lengths of 30 hops, which should be a safely high number of hops for small-world scenarios.

4.2.3 Routing table

The routing table contains SSR nodes, organised in a tree structure. The links in the tree represent known routing paths. The tree is stored as an array, where each array element represents a single node with information about parent / child relationships and when each entry was last used in a routing decision (compare figure 4.2).

The SSR protocol assumes $O(\log N)$ routing table entries in a network with N nodes [Fuh05, p. 240]. For the old *ssr-core* implementation, each entry (i.e. routing table line) was 18 bytes large (see figure 4.2). The implementation always used 255 entries and therefore required 4,080 bytes of RAM for the routing table entries alone.

[3] Depending on whether the Bluetooth stack supports scatternets, this limitation could be circumvented. In theory, there's no real limitation, but real-world implementations are often restricted in their capabilities.

Type	Name	Bytes	Description
`addr_t`	address	4	Virtual SSR node address.
`u16_t`	uplink	2	Array index of parent entry.
`u16_t`	auxlink	2	Possible, alternative parent entry.
`u8_t`	hop	1	Number of hops from the local node.
`u16_t`	fan	2	Number of child nodes.
`bool`	protect	1	Is node protected?
`bool`	used	1	Is entry in use?
`bool`	reserved	1	Is entry reserved?
`u16_t`	time	2	Value of global time counter at last use.
Total		16	Total number of bytes used per entry.

Table 4.2: Structure of a single entry in the routing table before space optimisation.

We only intend to use about 2 to 3 KiB of RAM for the complete implementation, so the routing table needs to be significantly reduced in size.

Reducing the number of routing table entries

The SSR protocol implementation was successfully tested with a routing table consisting of 255 entries for up to $N = 128,000$ nodes [Fuh05, p. 247].

Unfortunately, none of the papers on SSR analyse the effect of routing tables with significantly less than 255 entries. Apparently, the assumption was, that 255 entries would not use up more than 4 KiB of memory and that using 4 KiB of memory would be suitable for sensor networks [Fuh05, p. 246, "Global consistency and achieved routing stretch"]. Currently, although the routing table itsself only uses 4 KiB, the rest of the library uses additional memory and therefore exceeds the 4 KiB limit and additionally, even using 4 KiB might still be too much for targetted the AVR target system.

As the big-O-notation does not provide us with a straight-forward mathematical way of calculating smaller table sizes for smaller values of N, we can only rely on informed guesses for now:

The minimum number of routing entries that need to be cached consists of the path to the node's virtual successor and predecessor nodes as well as all physical neighbours. In the worst case, all nodes could form a chain, which would result in the successor and predecessor each being up to $N - 1$ hops away. This would mean (assuming M neighbours) that we'd need $O(N + M)$ routing table entries. As that would exceed the used table size by far, the lower boundary obviously is not helpful.

Unfortunately, it remains unclear how the $O(\log N)$ entries translate to a *specific* number of entries (and what N to assume in our AmbiComp environment). We will likely have to reduce N until all data structures fit into the available memory space.

Reducing the size of each routing table entry

Looking at the routing table structure, one finds potential for space optimisations. Shrinking down the integer size of entries that serve as indexes to a byte and using a single byte for the boolean values (which previously used a byte each), leads to 11 bytes per entry (figure 4.3). This amounts to a total of 2,805 bytes for the 255 entries. An even more compact representation of the remaining two boolean flags (using the

newly written `cBitFieldArray`, a structure that uses one bit per boolean entry), saves another 159 bytes. As the code made no use of the protect flag, it can be removed completely, saving yet another 31 bytes. After all optimisations, the table is 10.25 bytes per entry, amounting to 2,613 bytes for 255 entries.

type	name	bytes
`addr_t`	address	4
`u8_t`	uplink	1
`u8_t`	auxlink	1
`u8_t`	hop	1
`u8_t`	fan	1
`bool`	used	0.5 (0.125)
`bool`	reserved	0.5 (0.125)
`u16_t`	time	2
Total		11 (10.25)

Table 4.3: Structure of a single entry in the routing table after space optimisation.

4.3 Exceptions

As the AVR target-environment currently misses support for exceptions, that C++ feature needed to be removed from the ssr-core library. Exceptions were used through-out the code for out-of-memory (OOM) conditions, various internal error states and even abused to indicate the successful delivery of a message[4].

4.3.1 Errno value

An approach similar to libc's `errno` was chosen: A thread-specific error number variable, accessable via global `SetErrno()` and `GetErrno()` methods was introduced. This is used to provide detailed information about the last error.

Functions originally without return value now return a boolean value indicating failure or success and set the Errno value appropriately. To easily identify legacy code that does not check the return value, the attribute `SSR_MUST_CHECK_RETVAL` is introduced to enable compile-time warnings.

Functions which already returned a value needed to be modified to return an obviously invalid value for the error case. To allow this, a few types needed to be enhanced. E.g. `cPath::zero` and `cNicAddr::zero` were introduced.

4.3.2 Exceptions from constructors

One class of exceptions could not be easily replaced: Exceptions thrown from constructors. Constructors cannot return an error value, so there's no easy way for them to indicate an error. A typical approach to reporting errors in constructors is to move the actual work into an init function. That function would then be able to indicate initialisation errors through regular return values.

[4]The *DestinationReachedException* was used to indicate that a message had reached its destination.

For the old and new code, this problem affects the `cPath` classes. The constructors attempt memory allocations and cannot indicate or resolve OOM conditions gracefully. Unfortunately, the `cPath` classes are used in hundreds of different places through-out the code (routing code, message handling, etc.), making the proposed init-function-approach very work-intensive.

It is assumed, that OOM conditions can be avoided for controlled network environments, so for now, the library calls `SSR_ERROR()` (typically causing std::abort to be called) on OOM conditions. This temporary approach allows the exceptions to be removed without the need for a lot of redesign work.

4.4 Run-Time Type Identification

Although the target platform supports RTTI, the feature is disabled by default. When activated, the feature would requires a few additional bytes per class and should therefore be avoided if possible. The old code uses RTTI through the use of `dynamic_cast<>` to determine the type of message classes. As every message already has an embedded type ID[5], this can be used as type information instead.

Therefore, in the new *ssr-core* implementation all `dynamic_cast<>`s were replaced with checks against `getID()` or a few helpers[6] and `static_cast<>`s.

Ideally, the functionality using explicit message type tests should be refactored into virtual methods of the `cMessage` class, so that explicit type checks are no longer necessary. The refactoring was not performed and remains as future work.

4.5 Removing inlining

The goal of removing the inlining used within the old code-base was to reduce the code size. Currently, small code-size is more important for the use-cases of the *ssr-core* library than speed, especially for the AVR platform.

The old implementation inlined large portions of the code either implicitly, by embedding the code within the class definition in the class' header file, or explicitly by defining the method in the header and using the `inline` keyword.

Both of these inlining approaches were removed from the new code-base. All method implementations now reside within the implementation files (`*.cc`), i.e. the header files (`*.hh`) only contain the class declarations and definitions.

This is true even for template classes. The template classes' implementation was moved to `*.inl` files. These template inline files are included in `*.cc` files, where the template classes are then explicitly instantiated[7].

As can be seen in table 4.4, removing inlining saved $98,152 - 84,320 = 13,832$ bytes, i.e. the approach was rather successfull.

The table's *text* column signifies the code size, the *data* column represents the size of the initialised static data and the *BSS* column represents the size of the uninitialised static data. The *total* is the total space used on the flash. As the code is not loaded to the RAM, the *text* column has no influence on the RAM usage.

[5] The ID is defined in the enum *tMessageType* and is used in serialised messages to determine the type.

[6] *IsMsgConnect()* and *IsMsgHopByHop* are helpers to support checking against the messages' class hierarchy.

[7] For explicit template instantiation in C++, see e.g. `http://msdn.microsoft.com/en-us/library/by56e477.aspx`

Text	Data	BSS	Total	Inlining	cBitFieldArray
92928	1094	4130	98152	with	without
94820	1094	3872	99786	with	with
79584	702	4034	84320	without	without
79652	702	3872	84226	without	with

Table 4.4: Code and data sizes of ssr-core in bytes – before and after removing excessive inlining.

The `cBitFieldArray` is a template class minimising the size of the routing table (see section 4.2.3). As can be seen from the table and the earlier section on routing table shrinking, the space used on the flash increases by only 68 bytes, while saving 159 bytes of RAM.

4.6 Code renovation

Previous to the porting work described in this paper, the *ssr-core* code-base was difficult to understand. Before analysing the code, a thorough code cleanup was performed.

This involved:

- Applying a consistent coding style to the code. This included the renaming of classes, functions and member variables as well as applying consistent indentation.
- The detection and removal of unused code and unused variables.
- The normalisation of class hierarchies. Specifically, the node classes were pulled together into a single abstract node class. Previously, the various classes were divided into parent and base classes, but followed no abstraction between one-another, which only caused confusion and unnecessary up-castings.
- Making various methods either non-virtual or purely virtual to clearly document the intended usage through C++ language features.
- Splitting up the header and implementation files into one header and one implementation file per class. Previously, the classes were organised in only a hand-full of header and implementation files, which made navigating the classes difficult.
- Creating a Makefile based build system, that properly handled incremental rebuilds and supported building for several target platforms.
- Improving the automated testing of the code and integrating the unit testing into the build system. (See chapter 6 for details.)
- Introducing an SSR namespace to allow intuitive naming of the implementation's classes without fearing collisions with application code. Any existing SSR-prefixes in class names were removed.

Chapter 5

The new ssr-core library

This chapter provides a detailed configuration-, state- as well as class-level overview of the reworked *ssr-core* implementation.

5.1 Code configuration

The ssr-core library is designed to be usable in different applications and environments without the need for code modifications. Therefore only one code-base needs to be maintained and all applications and platforms profit from any fixes or enhancements added to the code.

The flexibility necessary to achieve the above goal is reached through two approaches, which are detailed below.

5.1.1 Abstract cNode class

The library provides an abstract `cNode` class with purely virtual methods performing application and system specific operations. This includes sending and receiving network packets on the link layer, handling of the payload data, timer handling and more. A new application therefore only needs to sub-class and implement the abstract `cNode` class and does not need to modify the ssr-core library itsself. The child class will most likely act as adapter class, relaying actions between the application and the environment through itsself to `cNode`, as shown in figure 5.1.

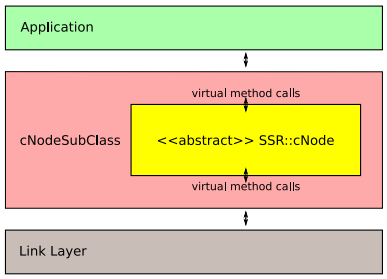

Figure 5.1: Users of the *ssr-core* library need to sub-class the abstract *Node* class.

5.1.2 Compile-time options

To enable or disable certain code-paths *within* the library, preprocessor symbols are used.

The preprocessor symbols listed in table 5.1 can be defined (or left undefined) in the target directory's *config.hh* file.

The study thesis added all compile-time options apart from `SSR_WITH_DEBUG`. The concept of one `config.hh` file per target, containing all config preprocessor symbols, was also newly introduced.

Preprocessor symbol	Description
`SSR_WITH_DEBUG`	Enable debugging.
`SSR_WITH_SIMULATION_SUPPORT`	Enable methods and settings relevant for simulating the code with OMNeT++.
`SSR_WITH_STATIC_TABLES`	Activate static tables.
`SSR_WITH_NEIGHBOR_TABLE`	Enable support for the neighbour table feature.
`SSR_HAS_STL`	Platform provides support for STL.
`SSR_HAS_HEAP`	Platform provides heap memory.
`SSR_HAS_EXCEPTIONS`	Architecture has support for exceptions.
`SSR_HAS_TLS`	Architecture has thread-local storage.
`SSR_SET_NUM_SIMULTANEOUS_MSGS`	Number of messages expected to be instantiated at once.
`SSR_SET_NIC_ADDR_SIZE`	Size of link-layer addresses for physical neighbours in bytes.
`SSR_SET_ADDR_SIZE`	Size of virtual node addresses in double words (4 bytes).
`SSR_SET_CACHE_CORE_SIZE`	Number of entries within the routing table.

Table 5.1: Reflexive edges on the SSR state.

Switches either indicate features available or unavailable on a specific platform (`SSR_HAS_*`), control optional functionality within the ssr-core library (`SSR_WITH_*`) or set certain configuration values (`SSR_SET_*`).

Note: In case the existence of a class or the relationship between classes depends on the above symbols, the following diagrams are coloured appropriately.

As described in section 4.1, dynamic classes where replaced with static variants for the AVR platform. The dynamic and static code-paths are disabled or enabled by the above mentioned preprocessor symbols. To avoid littering the code with preprocessor `#ifdefs`, the code was most often refactored into two separate classes: One class providing the dynamic approach – often using the STL – and one class providing the static one with fixed table sizes.

The dynamic code parts are compiled as long as the platform provides the necessary features. E.g. symbol `SSR_HAS_STL` and symbol `SSR_HAS_HEAP` are very often necessary for the dynamically sized structures. The static code variants are activated as long as `SSR_WITH_STATIC_TABLES` is specified.

The idea behind allowing both code paths to be activated at the same time is to allow the compile-testing of both code-paths on a single build target. This can quickly reveal obvious bugs during development.

Thread-local storage is optionally used for the storage of error numbers (see section 4.3.1) in thread-specific memory. If the platform has support for it and `SSR_WITH_STATIC_TABLES` is disabled, multiple threads can each run an instance of `cNode`.

If a switch configuration is chosen that does not allow the library to work, an error is displayed at compile-time. An example for such an errornous configuration would be disabling `SSR_WITH_STATIC_TABLES` and `SSR_HAS_HEAP` at the same time.

5.2 State machine

As already mentioned in section 3.2, the ssr-core library neither has explicit states nor an explicit state machine. The state machine in figure 5.2 none-the-less attempts to document the major implied states in which the SSR node can reside.

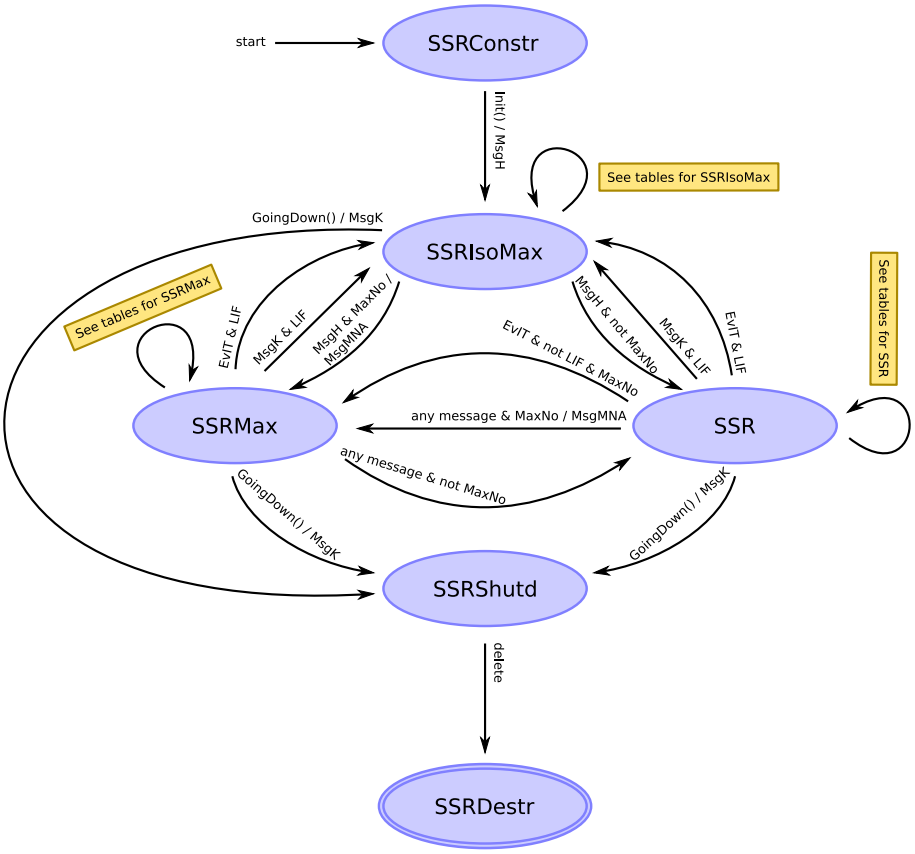

Figure 5.2: State machine, documenting the states of an SSR node.

The states are implied by the entries contained within the routing table, by the interface table (section 5.3.6 and by the object life-cycle of the node object (section 5.3.1).

Details on the names and abbreviations used within the state diagram are given in table 5.2. The states are explained in the following sub-sections.

Event name	*Event class*
EvIT	`cSsrEventIfTimeout`
Message name	*Message class*
MsgH	`cMsgHello`
MsgK	`cMsgKill`
MsgMNA	`cMsgMaxNodeAnnounce`
Condition name	*Description*
LIF	Last interface, there was only one interface left.
MaxNo	The local node now has the max virtual address in the known ring.
Entry point	*Description*
`Init()`	Initialises the node, activates various timers and announces the nodes existence.
`GoingDown()`	Notify the network, that this node is shutting down.
`delete`	Destruct the node instance.

Table 5.2: Abbreviations used within the state machine.

5.2.1 State SSRConstr

The *SSRConstr* state is the initial state directly after construction of an SSR node. Only the basic initialisations have been performed, no timers are active and the physical neighbours know nothing of this new node.

As soon as `Init()` is called, the node transitions to state *SSRIsoMax* and a `cMsgHello` message is broadcast via `cNode::Broadcast()`, to announce the node's existance.

5.2.2 State SSRIsoMax

In the *SSRIsoMax* state, the node is isolated. It has no known physical neighbours and therefore the known network is empty, apart from itsself (i.e. the routing table and the interface store are both completely empty). The virtual ring has the local node as its only member, which means that the local node is the node with the highest virtual address.

The *SSRIsoMax* state is the initial state directly after initialising the node. At that point, none of the potential neighbours had the opportunity to announce themselves yet.

The state is left as soon as one of the physical neighbours reacts to the regularly broadcasted hello messages (or if one of the neighbours hello messages is received).

The state can of course be reached again at a later point in time, if all known physical neighbours disconnect or go out of range.

The timer events and entry-points documented in table 5.3 cause no state changes and are therefore reflexive edges on the state.

5.2.3 State SSRMax

In the *SSRMax* state, the node has the largest virtual address in the known virtual ring. The node regularly broadcasts this knowledge to the whole network.

Incoming event	*Action*
`cSsrEventNotification`	*None*
`cSsrEventBroadcast`	send `cMsgHello`
Entry point	*Action*
`SendPayload()`	call `SendDroppedUp()`
`SendPayload()`	call `SendUp()`

Table 5.3: Reflexive edges on the SSRIsoMax state.

Apart from having the highest virtual address in the SSR network, the state is equal to the *SSR* state, see below.

In case a message is received, which indicates that there is another node with a higher virtual address, the state transitions to the *SSR* state.

In case all physical neighbours are lost, the state transitions back to the *SSRIsoMax* state.

The timer events, messages and entry-points documented in table 5.4 cause no state changes and are therefore reflexive edges on the state.

Incoming event	*Action*
`cSsrEventNotification`	send `cMsgMaxNodeAnnounce` and `cMsgNeighborNotification`
`cSsrEventBroadcast`	send `cMsgHello`
`cSsrEventIfTimeout` and more than one interface left	*None*
Entry point	*Action*
`SendPayload()` for remote node	send `cMsgPayload`
`SendPayload()` for unreachable node	call `SendDroppedUp()`
`SendPayload()` for local node	call `SendUp()`
Incoming message	*Action*
`cMsgPayload` for remote node	cMsgPayload
`cMsgPayload` for unreachable node if not marked as routed	`SendDroppedUp()`
`cMsgPayload` for unreachable node if marked as routed	`SendUp()`
`cMsgPayload` for local node	`SendUp()`

Table 5.4: Reflexive edges on the SSRMax state.

5.2.4 State SSR

The *SSR* state is the state in which most nodes within the SSR network will reside most of the time. The node has knowledge of other SSR nodes within the network. It is not the node with the largest virtual address (see state *SSRMax*).

Sending and receiving messages is possible and the sent messages should be able to reach their destinations.

The following timer events, messages and entry-points cause no state changes and are therefore reflexive edges on the state:

Incoming event	*Action*
`cSsrEventNotification`	send `cMsgNeighborNotification`
`cSsrEventBroadcast`	send `cMsgHello`
`cSsrEventIfTimeout` and more than one interface left and not the maximum node	*None*
Entry point	*Action*
`SendPayload()` for remote node	send `cMsgPayload`
`SendPayload()` for unreachable node	call `SendDroppedUp()`
`SendPayload()` for local node	call `SendUp()`
Incoming message	*Action*
`cMsgPayload` for remote node	cMsgPayload
`cMsgPayload` for unreachable node if not marked as routed	`SendDroppedUp()`
`cMsgPayload` for unreachable node if marked as routed	`SendUp()`
`cMsgPayload` for local node	`SendUp()`

Table 5.5: Reflexive edges on the SSR state.

5.2.5 State SSRShutd

In the *SSRShutd* state, the node was deregistered from the SSR network. All timers are inactive. No messages can be received or processed.

State transitions to this state cause `cMsgKill` to be broadcast to all physical neighbours.

5.2.6 State SSRDestr

In the *SSRDestr* state, the node instance was destructed and its memory freed. All timers are inactive. No messages can be received or processed.

5.3 Class diagrams

5.3.1 cNode

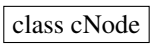

Figure 5.3: Class hierarchy of class cNode.

The `cNode` class as seen in figure 5.3 is an abstract class representing an SSR node in the network. Most library interaction is performed through this class.

The class provides a set of pure-virtual methods through which the node communicates with the upper and lower layers. Users of the library are expected to sub-class `cNode` and implement the virtual methods. This allows to adapt the *ssr-core* library to many different environments.

See section 2.4.1 for a detailed description of the expected interactions.

5.3.2 cRouteCache

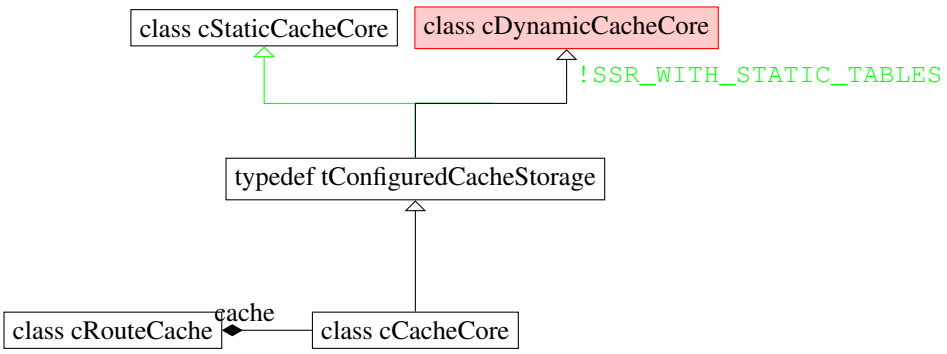

Figure 5.4: Class hierarchy of class cRouteCache.

A `cNode` instance, as seen in figure 5.4, maintains one instance of `cRouteCache`. It maintains the node's routing table. It contains `cCacheCore` as a member element, which contains all known source routes in a tree-like structure (see section 4.2.3).

The class contains all routing, path updating and merging logic. The actual storage of the source route elements, i.e. the known routing hops, is performed by the `cache` member of type `cCacheCore`.

The `cCacheCore` has two storage options: One option is to allocate memory on the heap (using `cDynamicCacheCore`). The other option is to use a fixed-size embedded member variable (using `cStaticCacheCore`). The latter option effectively causes the memory to be embedded within the `cRouteCache` instance, which itsself is embedded within the `cNode` instance.

The `cDynamicCacheCore` class is only built if `SSR_HAS_HEAP` and `SSR_HAS_EXCEPTIONS` are enabled. And depending on whether `SSR_WITH_STATIC_TABLES` is defined or not, `cStaticCacheCore` or `cDynamicCacheCore` are used as storage.

5.3.3 BitFieldArray

The `cBitFieldArray` template class, as seen in figure 5.5, provides an array of N elements, each providing storage for `NUM_BITS` number of bits. The storage is optimised for space. Single bits within the list can be read or written. Read and write complexity is within $O(1)$.

The class allows to select from two storage types: `cDynamicBitFieldArray` allows N to be specified at run-time and stores the elements in heap memory. `cStaticBitFieldArray` sets N at compile-time and uses a fixed-size buffer.

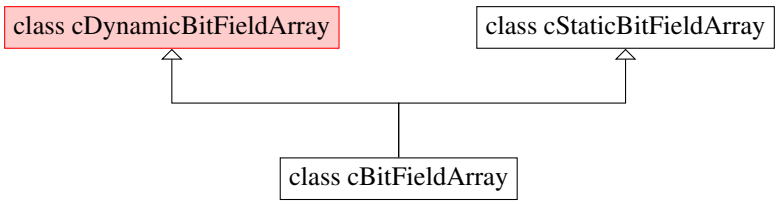

Figure 5.5: Class hierarchy of class cBitFieldArray.

The data structure is used within the `cCacheCore` storage back-ends to store a few bits per entry in a space efficient way (see section 4.2.3).

5.3.4 Events

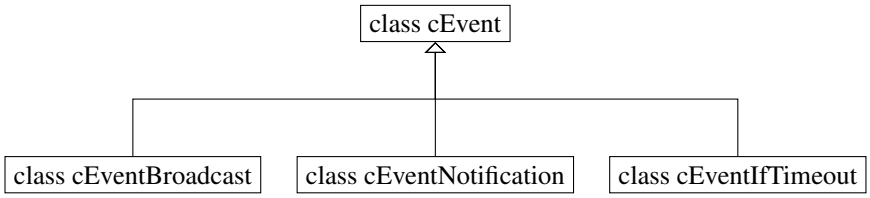

Figure 5.6: Event class hierarchy.

The classes extending the abstract base class `cEvent`, as seen in figure 5.6, represent time-based events. The event management is handled transparently through virtual methods in `cNode` (see `cNode::CallMeLater()` and `cNode::CancelEvent()`), so there's no event-loop in ssr-core.

The purpose of the event subclasses `cEventBroadcast`, `cEventNotification` and `cEventIfTimeout` was already described in 3.1.3.

As described in section 4.1.3, the events are stored within objects that match their life-time. I.e. `cEventBroadcast` and `cEventNotification`, which regularly notify physical and virtual neighbours of the local node's existence, are stored within the `cNode` object. Accordingly, `cEventIfTimeout`, which times out physical neighbours, is stored within the physical neighbour's entry in the interface table.

5.3.5 cAddr

The `cAddr` class, as seen in figure 5.7, stores a single SSR node address of fixed size. Although the class is intended to allow SSR node addresses of configurable sizes, it currently assumes an address size of 4 bytes and will need further work to function with different sizes.

`cBaseAddr` provides an address-length independent base for address storage and may be used by code external to the ssr-core, e.g. for representing a NIC address.

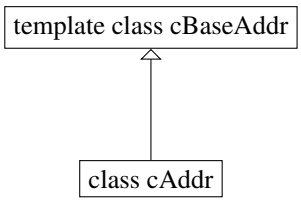

Figure 5.7: Class hierarchy of class cAddr.

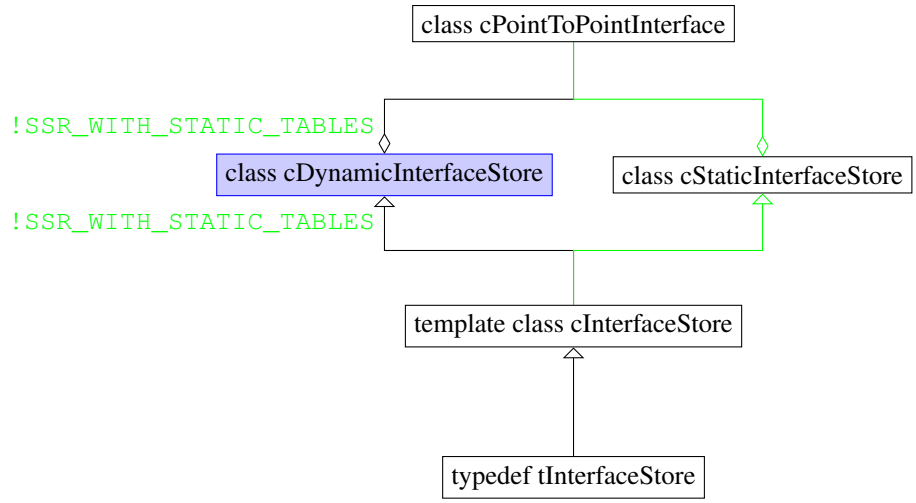

Figure 5.8: Interfaces class hierarchy.

5.3.6 Interfaces

The SSR node maintains one instance of `tInterfaceStore`, which keeps track of all known, active, physical neighbours.

The class, as seen in figure 5.8, provides a dynamically sized and a statically sized storage backend. The dynamically sized backend (`cDynamicInterfaceStore`) uses a map of unrestricted size, mapping each physical neighbour's virtual address to its `cPointToPointInterface` instance. The statically sized backend (`cStaticInterfaceStore`) uses a fixed-size array to store the same mapping. It does not yet provide a proper replacement strategy in case the array is full.

A `cPointToPointInterface` instance mainly stores the physical neighbour's physical address of type `tNicAddr` and keeps track of when the neighbour has last shown activity. This is regularly checked for timeouts by its embedded `cEventIfTimeout` timer.

The `cDynamicInterfaceStore` class is only built if `SSR_HAS_STL` is defined. And depending on whether `SSR_WITH_STATIC_TABLES` is defined or not, `cStaticInterfaceStore` or `cDynamicInterfaceStore` are used as storage backend.

5.3.7 NeighborTable

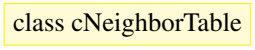

Figure 5.9: Class hierarchy of class cNeighborTable.

The SSR node maintains one instance of the `cNeighborTable` seen in figure 5.9, which stores the virtual neighbours of a node and a subset of the virtual neighbours' physical neighbours.

Copies of the table, filled with physical neighbours of the local node, are sent to the node's virtual neighbours via the `cMsgNeighborNotification` message. On reception of such a message from one of the virtual neighbours, the information is used to update the local `cNeighborTable` instance.

The table increases the likelyhood of quickly finding an alternative path to a virtual neighbour in case a path breaks. The feature is especially interesting for networks with a lot of node churn. (Please see the paper introducing this feature [Fuh06, p. 37] for further details.)

In its current state the class only works for simulation environments, as the serialisation and deserialisation functions are incomplete. This aspect of the code was not enhanced or modified by this work.

The `cNeighborTable` class is only built, if `SSR_WITH_NEIGHBOR_TABLE` is defined.

5.3.8 MaxNodeAnnounceStore

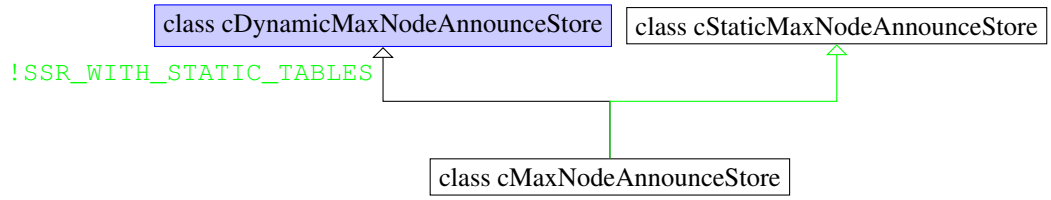

Figure 5.10: Class hierarchy of class cMaxNodeAnnounceStore.

The SSR node maintains one instance of `cMaxNodeAnnounceStore`. The class as seen in 5.10 keeps track of the last N `MsgMaxNodeAnnounce` messages, avoiding duplicate message handling (and endless flooding with the same message).

See section 5.3.11 for details on its use.

The class provides a dynamically sized and a statically sized storage backend. The dynamically sized backend (`cDynamicMaxNodeAnnounceStore`) uses an `std::set` class, storing an unlimited number of node addresses. The statically sized backend (`cStaticMaxNodeAnnounceStore`) uses a simple FIFO, storing the last N addresses.

The `cDynamicMaxNodeAnnounceStore` class is only built if `SSR_HAS_STL` is defined. And depending on whether `SSR_WITH_STATIC_TABLES` is defined or not,

cStaticMaxNodeAnnounceStore or cDynamicMaxNodeAnnounceStore are used as storage.

5.3.9 NicAddr

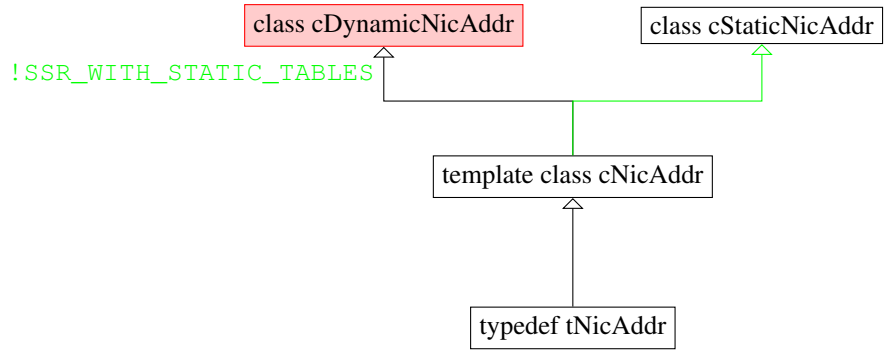

Figure 5.11: Class hierarchy of class cNicAddr.

Instances of the tNicAddr class as seen in figure 5.11 represent a physical neighbour's link-layer address. Within the SSR core, these addresses are handled completely transparently as a block of bytes and comparisons between two addresses are performed byte by byte.

This allows the library user to hand in any kind of structure that identifies a single physical address. A sensible data block might be e.g. a 48 bit Ethernet MAC-address or a 32 bit memory pointer, pointing to a more complex data structure on the heap.

The cDynamicNicAddr storage back-end uses heap memory to dynamically store a copy of any number of address bytes. The cStaticNicAddr storage back-end instead uses a fixed-size embedded buffer for the copy.

The cDynamicNicAddr class is only built if SSR_HAS_HEAP and SSR_HAS_EXCEPTIONS are defined. Depending on whether SSR_WITH_STATIC_TABLES is defined or not, cStaticNicAddr or cDynamicNicAddr are used as storage.

5.3.10 Paths

The cPath class as seen in figure 5.12 stores a source route, which consists of an array of SSR node addresses. The cWritablePath class is a sub-class that allows the manipulation of the stored source route and the cEmptyPath sub-class represents an empty route.

The cDynamicPath storage back-end uses heap memory to dynamically allocate memory for the list of SSR node addresses. The length of the path is not restricted. The cStaticPath storage back-end instead uses a fixed-size array, therefore restricting the possible path lengths to a value defined at compile-time.

Both storage back-ends use reference counting, to avoid duplicate copies of the same path. For cStaticPath, this is achieved by using cStaticPool, see section 5.3.13.

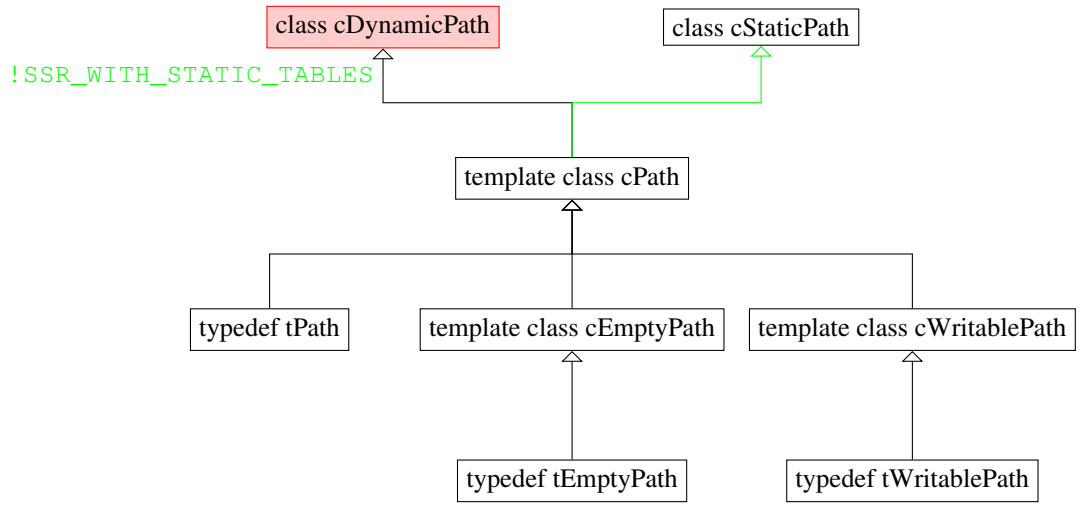

Figure 5.12: Path class hierarchy.

The `cDynamicPath` class is only built if `SSR_HAS_HEAP` and `SSR_HAS_EXCEPTIONS` are defined. The typedefs `tPath`, `tWritablePath` and `tEmptyPath` are defined to use the static or dynamic path storage depending on whether or not `SSR_WITH_STATIC_TABLES` is defined.

5.3.11 Messages

The leaf classes within the class hierarchy tree, as seen in figure 5.13, represent SSR messages sent over the wire. The messages are either unserialised from the bit-stream using the `cMessage::ReadFromBuffer()` factory method, newly constructed by the library via `new` or cloned via the `Clone()` virtual method.

The non-leaf classes `cMsgHopByHop` and `cMsgConnect` are implementation helpers and do not represent real SSR messages.

In case the environment provides no heap memory (i.e. `SSR_HAS_HEAP` is not defined), the messages are created using the `cMessagePool` (which is based on `cStaticPool`, see section 5.3.13).

The hello message, represented by the `cMsgHello` class, is used to broadcast the local node's existence to all physical neighbours. The message is regularly broadcast by the local node using the `cEventBroadcast`-based timer. On reception of such a message, it is not forwarded to other nodes, as it only concerns direct physical neighbours.

The kill message, represented by the `cMsgKill` class, is used to indicate a node's shutdown. It is broadcast to all physical neighbours. When receiving the message, the indicated node is removed from the interface table and the routing table. The message is not forwarded to other nodes.

The max node announcement message, represented by the `cMsgMaxNodeAnnounce` class, is used by the node with the highest virtual address to broadcast its existence to the whole SSR network. Every node receiving the message will first check whether it has received the same message before. If the

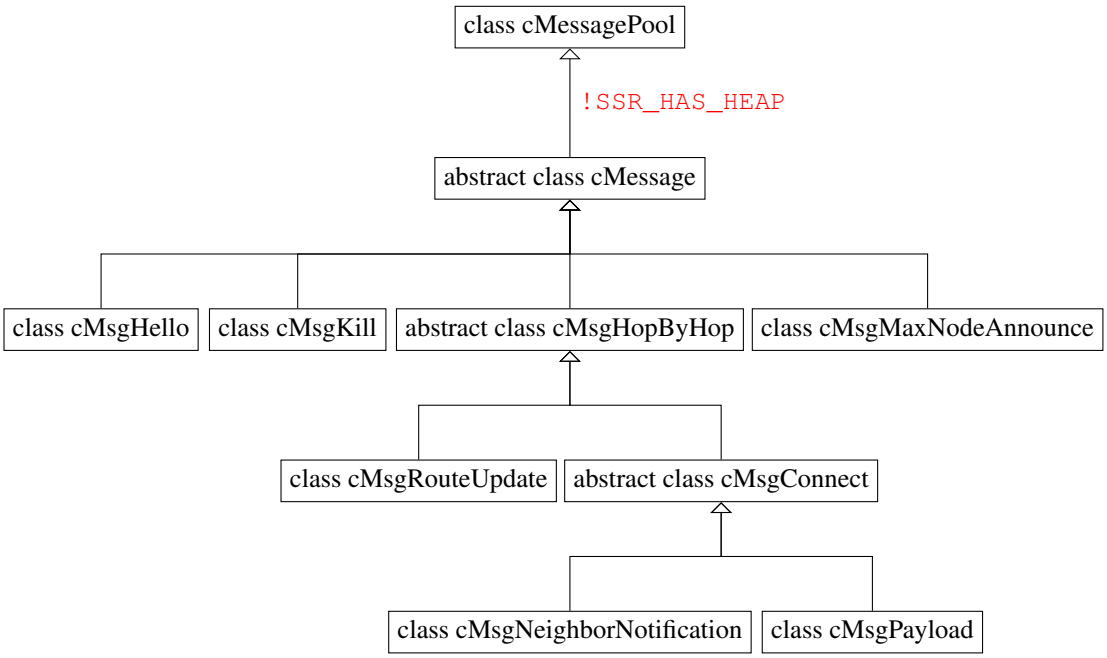

Figure 5.13: Message class hierarchy.

message is new, it will check whether the broadcasted information matches with its knowledge of the virtual ring. Based on this it might possibly send updates to its virtual neighbours. In the end, the announcement message is broadcast to all physical neighbours (apart from the one it originally received it from).

The neighbour notification message, represented by the `cMsgNeighborNotification` class, is regularly sent to both virtual neighbours in the virtual ring. In case the neighbour table feature is enabled, the message includes a copy of the local neighbour table. (See section 5.3.7 for details on neighbour tables.)

The payload message, represented by the `cMsgPayload` class, transports the actual data from the upper layer. The class instance does not hold a copy of the payload data within the instance, instead only a pointer to the original data is stored. If the message is newly created, the payload pointer points to the application's provided data buffer. It it was received from the link-layer, the pointer points to the raw message buffer. If the message has the routed flag set, it is assumed that the message's target address might not refer to an *existing* node's address, but instead be intended for the node that is *virtually nearest* to the target address.

Route update messages, represented by the `cMsgRouteUpdate` class, contain route updates and always target a specific node. It is used to inform nodes about broken links, so that they can update their routing tables. The message implicitly contains freshly updated replacement routes through its source path.

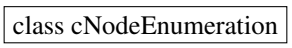

Figure 5.14: Class hierarchy of class cNodeEnumeration.

5.3.12 cNodeEnumeration

The `cNodeEnumeration`, as seen in figure 5.14, is used to allow node addresses from `cRouteCache` to be enumerated by users of the `cRouteCache` class.

5.3.13 cStaticPool

Figure 5.15: Class hierarchy of template class cStaticPool.

The `cStaticPool` template class, as seen in figure 5.15, provides a pool of memory chunks. Each chunk is of equal size. The class only provides simple acquiring or freeing of chunks. The memory pool is statically allocated at compile-time.

The class is used as implementation detail in `cMessagePool` for messages and in `cStaticPath` for a pool of reference counted source paths.

Chapter 6

Evaluation

The following sections detail how the modifications to the *ssr-core* library were verified and tested.

6.1 Unit tests

Modifications to any non-trivial piece of code need to be verified and tested thoroughly to avoid introducing bugs. The old *ssr-core* implementation used a separately maintained set of unit tests to verify some of the library components.

As the code coverage of the unit tests was very limited and the tests weren't integrated into the build process, they weren't very effective in preventing the introduction of bugs. Therefore, the test-suite is now integrated into the build tree and executed after each compile cycle, causing the build to fail in case one of the tests fails.

Errors within the unit tests are now reported in a format, that allows common graphical development environments to visibly point to the failed test. [1]

The unit tests were extended to cover more major components present within the library. Specifically, the following tests were added in this work:

- In suite `cTestPath`, the tests `TestSingleHop`, `TestWritable`, `TestReverse` and `TestZero`.
- In the newly added suite `cTestRouteCache`, the tests `TestEmptyHasSelfRW`, `TestEmptyThrowsNoPathFound`, `TestLearnsPath`, `TestRemoveLink`, `TestPhysicalNeighbors`, `TestPredecessorSuccessor` and `TestPathTowards`.
- In the newly added suite `cTestBitFieldArray`, the tests `TestSizeCalculation`, `Test3And8Single`, `Test3And8Highest`, `Test3And8All`, `Test8And3Single`, `Test8And3Highest` and `Test8And3All`.

Further tests were later added by Pengfei Di, the maintainer of the *ssr-core* library.

According to `make check-test-coverage`, which uses the *LTP GCOV extension*[2], the current code coverage remains at only about 43.5 percent.

[1] The messages are displayed in a format that is sufficiently similar to the error output of the GNU GCC compiler. This was verified with Eclipse.

[2] See `http://ltp.sourceforge.net/coverage/lcov.php`

Although the code coverage is far from perfect, the existing unit tests helped iron out a few bugs during the *ssr-core* porting effort.

6.2 Automated simulation tests

Unit tests, as described in the previous section and as used within the project, are explicitly intended for the testing of classes without the interaction of other classes (and their bugs). Therefore, unit tests cannot test whether the components in the ssr-core library properly interact and whether the library correctly implements the SSR protocol.

To achieve something close to this kind of verification, SSR network simulations are used, based on the *OMNeT++*[3] framework. *OMNeT++* provides a simulation environment that passes messages between user defined layers (figure 6.1). The environment allows the definition of hosts and network topologies, the tracking of the messages between the defined network layers and hosts. It also simulates the progress of time, allowing network tests spanning several hours to be simulated within seconds.

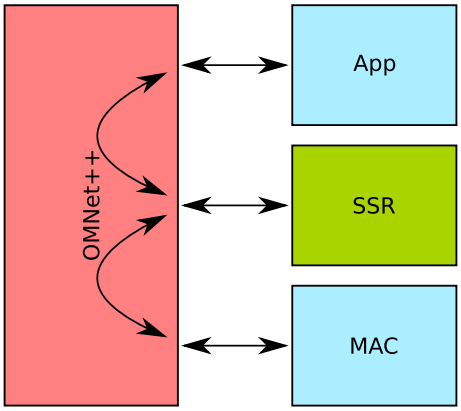

Figure 6.1: Message passing within the OMNeT++ simulation framework.

Using this framework and a simulation of a typical MAC layer allows the realistic modelling and full simulation of an SSR network. These types of full simulation scenarios were used to test *ssr-core* since the early stages of the library.

For reproducible simulations, as required for automated testing, the existing, complex simulation of the MAC-layer was too unpredictable. Therefore, the complex MAC simulation was reduced by Pengfei Di to a new, simple and deterministic message-passing layer. This layer is provided within the new `easyOmMac` framework.

One of the features of the new framework, provided by the simulated application component, is the detailed logging of packet paths, packet run-times and similar information. Logs of known good simulation runs – using a trusted *ssr-core* library version – are saved as *reference log files*. Based on these reference log files, log differences during future simulation runs can be detected and reported.

The various simulation scenarios are stored in the `scenario/` subdirectory, while the expected content of the simulation log output files are stored within the `results/`

[3] `http://www.omnetpp.org/`

subdirectory. By running the simulation for each scenario and comparing the log results with the matching reference log files, the simulator was turned into an automated simulation tester.

The simulations based on the `easyOmMac` predictable MAC-layer proved very important for the *ssr-core* porting effort, as it was the only run-time integration test available during most of the development time. Although the simulations never showed any changes in routing behaviour, they *did* uncover interaction bugs or crashes within the library.

According to the *LTP GCOV extension*, the simulation scenarios cover 47.3 percent of the code, so more scenarios should be created to increase the code coverage. Especially error and packet-loss cases are currently untested.

6.3 Running on the AVR platform

To verify, that the ported *ssr-core* library actually executes on the AVR target platform and can use the basic AmbiComp software stack, the library was integrated into a test environment (figure 6.2).

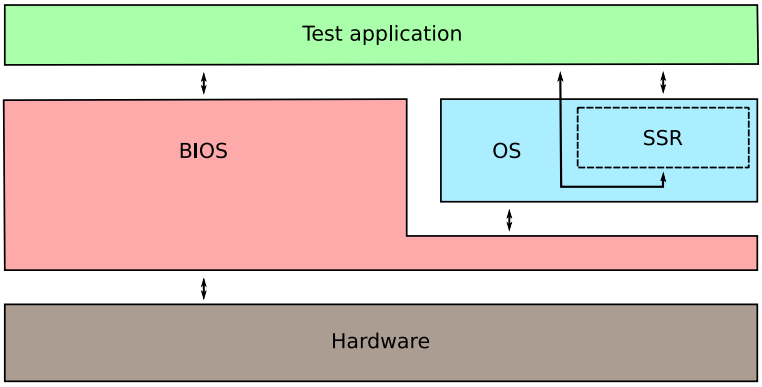

Figure 6.2: SSR test application in the partial AmbiComp software stack.

The test environment layers on top of the AmbiComp BIOS version 1.0. To avoid integration complexity and memory limits, it uses a custom test application instead of the ACVM. No interaction with any link-layer is performed. Instead, all activity of *ssr-core* is fed back into the test application and logged to the BIOS' serial console.

To allow the test application to run, the *ssr-core* library first needed to be compiled for the AVR platform. As GCC's AVR port only provides very basic C++ support, two simple libraries needed to be created: `avr-libstdc++` and `avr-libsupc++`.

The `avr-libstdc++` library serves as a very simple replacement for the missing `libstdc++`. It only provides a few missing, standard C++ headers and most of them merely wrap around standard C headers (e.g. `cstring` wraps around `string.h`).

The `avr-libsupc++` library provides method stubs and definitions, that belong to the compiler side of C++ support and are absolutely necessary to link a C++ program when using virtual methods or certain allocation and deallocation operators.

Next, based on the fully ported *ssr-core* library, a `cStubSsrNode` class was written, sub-classing `SSR::cNode`. For all virtual methods the stub class logs the activity

and returns. The only methods implementing real functionality are the timer event handlers (`CallMeLater()` and `CancelEvent()`). They interact with a timer event scheduler.

As the ACVM would normally provide the timer management, the test application needed to provide that facility itsself. Therefore, `cTimerEventScheduler` was created, to queue and schedule a limited number of events. It gets repeatedly called by the BIOS' single timer event.

As the node and timer instances are too large for the stack, they are allocated on the first external SRAM page (XRAM, page 0).

Tests have shown, that the node correctly registers its timers and regularly attempts to announce itsself to its neighbours:

```
XRAM: Detected 4 pages, each page 32 Kbytes large
XRAM: Creating cTimerEventScheduler
XRAM: Creating cStubSsrNode
cStubSsrNode constructed
Calling node init
cStrubSsrNode::CallMeLater(2.520563)
cStrubSsrNode::CallMeLater(1.183149)
Starting timer
Entering event loop
cStrubSsrNode::HandleTimerEvent for event type 3
cStrubSsrNode::CallMeLater(3.849298)
cStrubSsrNode::HandleTimerEvent for event type 1
cStrubSsrNode::Broadcast
cStrubSsrNode::CallMeLater(3.895320)
cStrubSsrNode::HandleTimerEvent for event type 3
cStrubSsrNode::CallMeLater(4.234942)
cStrubSsrNode::HandleTimerEvent for event type 1
cStrubSsrNode::Broadcast
cStrubSsrNode::CallMeLater(2.092654)
cStrubSsrNode::HandleTimerEvent for event type 1
cStrubSsrNode::Broadcast
cStrubSsrNode::CallMeLater(2.505668)
cStrubSsrNode::HandleTimerEvent for event type 3
cStrubSsrNode::CallMeLater(3.804689)
...
```

Any further testing would need to involve at least a second node instance, which would require adapting the code to at least one of the available link-layers. Due to time constraints, these additional efforts are not within the scope of this work.

Chapter 7

Conclusion and future work

7.1 Conclusion

In the course of this study thesis, the restrictions imposed by the AmbiComp environment, regarding the integration of the *ssr-core* library, were determined. Approaches were analysed to allow the library to cope with these restrictions. In the end, appropriate approaches were chosen and implemented. This involved the removal of heap allocations, exceptions, RTTI usage and STL usage. It also involved shrinking data structures and removing function inlining to shrink the memory and code foot-prints.

The changes were applied in such a way, that the *ssr-core* code-base remains fully usable for all previous and future use-cases. This was achieved by refactoring data storage classes and making the code configurable through compile-time options.

By improving the library's unit test coverage and by the introduction of reproducible network simulation scenario tests, the library was continuously verified.

The final *ssr-core* implementation was tested on the target platform AVR, interacting with AmbiComp's BIOS layer and performing basic operations.

Through-out the work, the library's code was cleaned up and refactored.

7.2 Safety in the wild

The new implementation of the ssr-core library compiles, links and runs on the targetted AVR platform and for the previously existing environments (i.e. within the OmNet++ simulation and on the MIPS platform). Before it is safe for usage in an environment outside of simulations and carefully managed nodes, at least the following issues need to be resolved.

7.2.1 Errors in constructors

As mentioned in section 4.3.2, the `cPath`-based classes do not handle out-of-memory (OOM) conditions gracefully. To fix this, the `Init()`-approach mentioned in that section could be implemented. It would involve moving the actual work out of the constructors into `Init()` functions. Errors would then be indicated by an appropriate return value and OOM conditions could be handled appropriately, e.g. by dropping packets. Those conditions would therefore no longer cause the program to abort.

7.2.2 Length limitation on source paths

The `cStaticPath` class assumes that paths do not exceed a certain hop-count (i.e. length). Apart from the unsafe allocation problems mentioned in the previous section, this static length approach might conflict with regular routing. Simulation runs might help to determine typical, maximum path lengths. It could also be analysed how many paths are used at the same time within the ssr-core implementation. Based on this information, useful values could be determined for the `cStaticPath` hop length and pool size.

7.2.3 Limited number of physical neighbours

The fixed-size interface storage table (`cStaticInterfaceStore`) needs a strategy for entry replacement in case it is full. Currently, in case a new neighbours appears and the table is full, the operation fails and the new neighbour is ignored.

Consider the following problems:

- How can network fragmentation be avoided if not all nodes within the local network are known to each other? How can a consistent virtual ring be then maintained?
- Would it help to accept new neighbours by picking them randomly, so that hopefully all nodes within the local network have connections to each other?
- Should nodes be prioritised before being thrown out? The priorisation could be based on network type (to keep links between different types of networks active), activity, age or some other scheme.
- Nodes announce themselves in regular intervals, so a full interface storage table would regularly lead to the same overflow conditions. Should interface churn be avoided within the table?

7.2.4 Table sizes need reality check

The table sizes chosen for various newly fixed-size structures should be verified by simulations. Currently, most of the sizes are arbitrarily chosen, as they can't easily be determined by theoretical observations.

This includes the interface table, the source path lengths and the size of the static source path pool.

7.3 Integration with the AmbiComp environment

7.3.1 Link-layers

To be useful for the AmbiComp environment, the `SSR::cNode` needs to communicate with other nodes. It therefore needs to have access to link-layers, so that communication with other AICUs is possible.

An adapter for link-layers would need to handle support for more than one link-layer type per node, e.g. for both ethernet and bluetooth at the same time. It would therefore have to multiplex/demultiplex the `HandleFromNic()`, `SendToNic()` and `Broadcast()` calls to and from the appropriate link-layers.

7.3.2 ACVM

Event loop

As the ACVM handles the event loop, any timer management would need to interact with the ACVM or use possibly existing timer infrastructure within it. In case the ACVM provides insufficient timer handling capabilities, `cTimerEventScheduler` (introduced in section 6.3) could be used as a starting point.

Access to SSR functionality

The ACVM needs to gain access to the functionality provided by the *ssr-core* library. This can either be *explicit* access, by providing Java Native Interfaces for message sending and receiving or *transparently*, by adapting the GAO implementation (compare [SEF08]) to make use of SSR.

Dynamic memory

As the *ssr-core* resides within the OS-layer and neither that layer nor the accompanying BIOS-layer provide dynamic memory management, all dynamic memory allocations had to be circumvented. It might be worthwhile to explore possibilities of using the ACVM's memory management facilities within the SSR.

Appendix A

Source tree and build environment

This appendix explains the layout of the library's source files and build system. It introduces naming conventions for source code and code configuration.

A.1 Files and directories

A clean checkout from the code repository will contain the following files and directories:

File or directory	*Description*
`/`	The base directory contains all source files, including all headers (except config.hh).
`/README.txt`	Contains information about the library and its structure. Briefly documents how to use it.
`/TODO`	Short list of open items.
`/Makefile`	The central makefile. It provides short-cuts for the actual makefiles within the compilation target directories.
`/common.mk`	Library of build targets used by the compilation target makefiles.
`/Doxyfile`	Configuration file for Doxygen, which is a documentation generator.
`/*.hh`	All C++ header files use the file name extension `.hh`. A single header file should describe a single C++ class. The base name is derived from the class name.
`/*.cc`	All C++ implementation files use the file name extension `.cc`.
`/*.inl`	All C++ inline files use the file name extension `.inl`. The inline files typically contain the implementation of C++ template classes and methods.

`/cross-avr/`	Each compile target has a separate directory. It contains a Makefile and a config.hh file. The "cross-avr" target is intended for use within the ACVM. The code gets cross-compiled for the AVR architecture. It is configured for static tables and no heap. No unit tests are run.
`/cross-avr/config.hh`	Defines a set of preprocessor symbols to configure the build for the "cross-avr" compile target.
`/cross-avr/Makefile`	Based on `/common.mk` this makefile builds the library for the "cross-avr" compile target.
`/cross-mipsel/`	Cross-compilation target for MIPS routers. No unit tests are run. The directory contains the same files as the `/cross-avr/` directory.
`/debug-linux/`	Full-featured dynamic linux target with debugging. The unit tests are run. The directory contains the same files as the `/cross-avr/` directory.
`/release-linux/`	Full-featured dynamic linux target *without* debugging. The unit tests are run. The directory contains the same files as the `/cross-avr/` directory.
`/static-linux/`	Linux build with static tables and no heap. The unit tests are run. This allows the static code paths used by AVR to be tested on a linux machine. The directory contains the same files as the `/cross-avr/` directory.
`/tests/`	Directory containing the source files for the unit tests.
`/tests/main.cc`	Contains the unit test runner.
`/tests/Test*.cc`	Unit test implementation file names are prefixed with "Test" by convention.
`/tests/cppunit2junit.xsl`	XSL file to translate the XML produced by CPPUnit to the XML format normally produced by JUnit.
`/tests/tests-common.mk`	Library of build targets specific to unit test building. It includes `/common.mk` and is used by the unit test compilation target makefiles.
`/tests/release-linux/`	Each native target has a matching build directory within tests. It contains a Makefile and all compile-time generated files. This specific directory is for the "release-linux" target.
`/tests/release-linux/Makefile`	Based on `/tests/tests-common.mk` this makefile builds the unit tests for the "release-linux" compile target.

/tests/static-linux/	Build directory for the static-linux target. The directory contains the same files as the /tests/release-linux/ directory.
/tests/debug-linux/	Build directory for the debug-linux target. The directory contains the same files as the /tests/release-linux/ directory.

As soon as one or more targets were built, the above directories contain the following new files and directories:

File or directory	*Description*
/<target>/	This should be read as the target's directory, i.e. replace <target> with the name of the built target (e.g. /cross-avr/).
/<target>/build_flags.mk	This makefile inclusion file contains the CPP_FLAGS used to build the library. It is automatically created by the build-system and can be included in projects that make use of the library.
/<target>/*.d	All files with the .d suffix are dependency files, documenting the matching .cc file's header dependencies, i.e. the build dependencies of the resulting object file.
/<target>/*.o	All files with the .o suffix are the object files for the matching .cc files.
/<target>/libssr-core.a	The *ssr-core* library variant intended for static linking.
/<target>/libssr-core.so	The *ssr-core* library variant intended for dynamic linking.
/<target>/include/ssr-core/	This directory is created at build-time. It contains symbolic links to the actual header files. By providing this directory structure, the headers can be referenced through fully qualified #include<ssr-core/Class.hh> statements, instead of through less specific #include<Class.hh> statements.
/tests/<target>/*.d	Dependency files. Same as /<target>/*.d, only for the unit tests.
/tests/<target>/*.o	Object files. Same as /<target>/*.o, only for the unit tests.
/tests/<target>/SSR-Test	The executable running all unit tests.
/tests/<target>/SSR-Test.cppunit.xml	After running the above unit test executable, this file contains the unit test results in CPPUnit XML format. Calling make SSR-Test.junit.xml would then create the same information in the JUnit XML format. (See below.)

A.2 Build targets

The following build commands are possible from the source's root directory:

Make command	*Description*
`make` or `make all`	Build the library for all build targets.
`make check-test-coverage`	Builds debug-linux and static-linux build targets and gathers code coverage for the unit tests. Generates a code coverage static in the `test-coverage-html/` directory.
`make clean`	Clean all build targets.

The following build commands are possible within a build target directory (e.g. within `/debug-linux/` for the "debug-linux" build target):

Make command	*Description*
`make` or `make all`	Build the library for the build target. For native builds, i.e. non-cross-compile builds, this automatically triggers `make run-tests` as well.
`make run-tests`	This make target only exists for non-cross-compile builds. It compiles and runs the unit tests. (It runs `make` within the matching unit test's build directory, see below.)
`make clean`	Cleans all build artifacts for current build-tree and the matching unit test tree.

The following build commands are possible within a unit tests' build directory (e.g. within `/tests/debug-linux/` for the "debug-linux" unit test build):

Make command	*Description*
`make` or `make all`	Builds the unit tests and runs them in case something changed (either within the tests or within the library).
`make run-tests`	Builds and runs the unit tests.
`make gdb-run-tests`	Builds the unit tests and runs them within a gdb session.
`make SSR-Test.junit.xml`	This reads the `SSR-Test.cppunit.xml` file and creates the JUnit format.
`make clean`	Cleans all build artifacts for the current unit test build tree.

Bibliography

[CF05] Curt Cramer and Thomas Fuhrmann. Isprp: A message-efficient protocol for initializing structured p2p networks. In *Proceedings of the 24th IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 365–370, Phoenix, AZ, April 7–9 2005.

[DEF08] Pengfei Di, Johannes Eickhold, and Thomas Fuhrmann. Linyphi: creating ipv6 mesh networks with ssr. *Concurr. Comput. : Pract. Exper.*, 20(6):675–691, 2008.

[EFS+08] Johannes Eickhold, Thomas Fuhrmann, Bjoern Saballus, Sven Schlender, and Thomas Suchy. Ambicomp: A platform for distributed execution of java programs on embedded systems by offering a single system image. In *AmI-Blocks'08, Workshop at the European Conference on Ambient Intelligence 2008*, Nuremberg, Germany, 2008.

[FDKC06] Thomas Fuhrmann, Pengfei Di, Kendy Kutzner, and Curt Cramer. Pushing chord into the underlay: Scalable routing for hybrid manets. Interner Bericht 2006-12, Fakultät für Informatik, Universität Karlsruhe, June 21 2006.

[Fuh05] Thomas Fuhrmann. Scalable routing for networked sensors and actuators. In *Proceedings of the Second Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, pages 240–251, September 2005.

[Fuh06] T. Fuhrmann. Scalable routing in sensor actuator networks with churn. *Sensor and Ad Hoc Communications and Networks, 2006. SECON '06. 2006 3rd Annual IEEE Communications Society on*, 1:30–39, Sept. 2006.

[SEF08] Bjoern Saballus, Johannes Eickhold, and Thomas Fuhrmann. Global accessible objects (gaos) in the ambicomp distributed java virtual machine. In *Proceedings of the Second International Conference on Sensor Technologies and Applications (SENSORCOMM 2008)*, Cap Esterel, France, August 25 – 31, 2008. IEEE Computer Society.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.