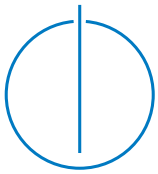
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Extension of L4 Fiasco.OC and Genode OS Framework by the Concept of a Distributed Shared Memory

Alexander Weidinger



DEPARTMENT OF INFORMATICS

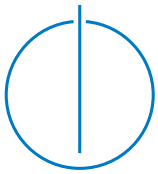
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Extension of L4 Fiasco.OC and Genode OS Framework by the Concept of a Distributed Shared Memory

Erweiterung von L4 Fiasco.OC/Genode um das Konzept eines verteilten gemeinsamen Speichers

Author:	Alexander Weidinger
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Sebastian Eckl, M.Sc.; Daniel Krefft, M.Sc.
Submission Date:	January 15, 2016



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, January 15, 2016 Alexander Weidinger

Acknowledgments

I want to thank my two advisors, Sebastian Eckl and Daniel Krefft, for always giving me new starting points when I got in trouble with some part of the thesis. Also I want to thank Christian Helmuth of Genode Labs and all the other developers on the #genode IRC-Channel, as they were very helpful when the API of Genode made problems. I want to thank as well Prof. Dr. Uwe Baumgarten, for giving me the opportunity to work on this interesting topic.

Abstract

In this thesis, the Fiasco.OC/Genode OS is extended by the concept of a Distributed Shared Memory (DSM). It starts by introducing the DSM concept and its development from the first shared memory system until software based DSM approaches. In the second part, the underlying platform Fiasco.OC/Genode is explained, by highlighting the characteristics of the system and presenting important software components of Genode. The main part explains the implementation and ideas of the DSM solution, while providing a sample showcase to show the capabilities of the DSM system. The final implementation creates a prototype, capable of providing transparent access to remote and local memory, without changing the code basis of Fiasco.OC/Genode. This establishes a good starting point for further development.

Contents

1 Introduction

In this chapter, the objective of this thesis is explained by giving an overview over the main points. The second part explains the motivation behind this approach.

1.1 Objective

In this thesis the Genode OS Framework, operating on top of the Fiasco.OC microkernel, is extended by the concept of a Distributed Shared Memory (DSM). This means, by combining some of the RAM from several physically separated nodes, one big shared memory is introduced to the nodes. This distributed shared memory is seamlessly integrated into the virtual memory management of each node. Because of that, it is possible to use familiar operators and libraries. The exchange of information is done by transmitting page-sized memory portions from the requested addresses to the origin of the request. The connection between all nodes is established over Ethernet.

The version of Genode OS, used for this thesis is Genode 15.11, which was published on 30th November, 2015. [Lab15] The suitable version of Fiasco.OC will be automatically built by Genode. To run the sample code, Qemu is used for virtualistation, while VDE (Virtual Distributed Ethernet) is utilized for providing an Ethernet connection between the Qemu instances. As the implementation is intended for use on embedded devices, the pbxa9 hardware was chosen for simulation.

The thesis starts with explaining the basic concepts and ideas of DSM, by providing an overview of the first hardware based shared memory systems, up to modern page based distributed shared memory systems. In the next section the underlying platform Fiasco.OC/Genode is introduced, by highlighting the characteristics of the system and presenting important software components. Afterwards in the main part, the ideas for

implementation and the implemented solution itself is explained. The last part shows a simple use case, where the capabilities and limits of this approach are presented.

1.2 Motivation

Fiasco.OC is a real-time capable microkernel with advanced security features. Because of this, it will be used together with the Genode OS Framework as part of a project, bringing few advanced and flexible ECUs into cars. Instead of relying on the old specialised constructed ECUs, the new system will replace the hardware redundancy through software.

ECUs produce and consume a lot of data, which will be exchanged, not by a classic vehicle bus system, but rather via an Ethernet connection. This approach allows the dynamic integration of external devices, which were not directly designed for usage in a car system, for example smartphones.

The distributed shared memory will be used in the project, to access sensor values without the need of a programmer being confronted with a message passing system. This means, that the whole message exchange is completely transparent to the programmer and done as a service. As the distributed shared memory is provided as a service, it can be easily expanded by caching strategies in the future, to reduce the delay when accessing distant memory locations, or even pre-fetch often used values.

2 Distributed Shared Memory

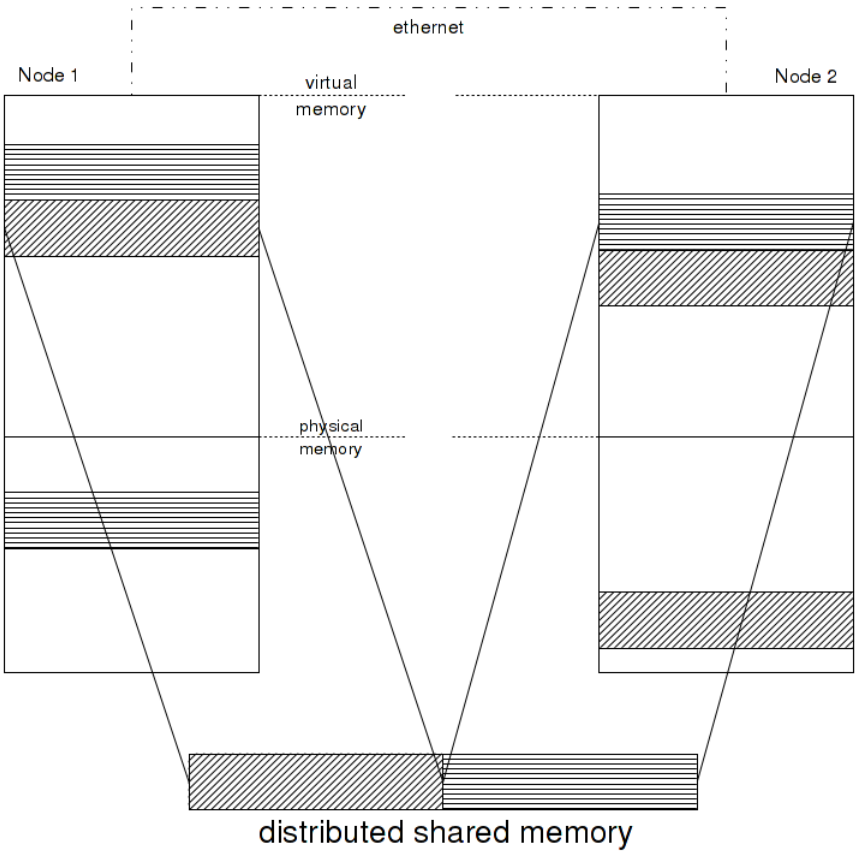


Figure 2.1: Distributed Shared Memory [Own diagram]

"Distributed shared memory (DSM) is a model for interprocess communication that provides the illusion of a shared memory on top of a message passing system." [AW04,

p.189]

By using DSM, the general idea is to expand a nodes memory by including memory pages from distant, physically separated nodes into its own virtual memory. (See figure 2.1) Applications on both nodes therefore can access the same memory, without the need to exchange data by themselves or depend on message passing libraries and their function calls. The message exchange is completely transparent to them.

In the following chapter, DSM will be compared with Message Passing and the two different DSM types, hardware and software based, will be shown. Since a part of this thesis is the implementation of a software based DSM, different approaches for a possible implementation will be shown and explained. In the last section of this chapter, the problem of Memory Consistency (MCS) will be introduced.

2.1 DSM vs. Message Passing

DSM uses Message Passing as part of the memory page exchanging and therefore is slower compared to just using Message Passing, because an additional layer of complexity is added on top. This slowness can be counteracted by adding an intelligent caching mechanism to the used DSM system. [Lu+95, p.16]

When using DSM, a big advantage is the possible complete transparency to the applications. By accessing memory locations, for the applications it seems like the memory is stored locally and not on a remote host. But this also comes with a disadvantage, because when an application is not aware of the position of the memory, the time costs for an access are also transparent to the application.

Since it is possible to alter the whole shared memory in a DSM environment, this also raises potential security issues. For example, any malicious application could change specific memory regions of the shared memory, which may contain important data. By counteracting this behavior with controlled read and write calls, this potential security issue can be reduced.

By using message passing systems, one has to wrap memory into variables, in order to exchange them via methods. Each exchangeable variable therefore needs its own

method, definition and constants. The DSM system makes exchange a lot easier, as a client only needs to know the correct address and structure of a variable to access it.

Due to the given access to memory locations, compared to the classical Message Passing approach, you are able to use libraries on these memory locations, without creating extra buffers. Taking the `char *strcpy(char *dest, chonst char *src)` [man-page of strcpy] as an example. While one can just give two pointer by using a DSM system, Message Passing would require taking additional steps. If the destination is part of the external memory, a buffer must be created for the string, to allow the usage of the `strcpy` function. Afterwards a `sendValue()` call must be made, to write the changes to the remote memory location.

2.2 Forms of DSM

Since the idea of DSM has evolved since the late 80s, different approaches have been created and established. Those will be explained in the next sections.

2.2.1 Hardware based DSM

The first approaches towards DSM were made after multiprocessor systems got more common. It started with the usage of shared memory for more than one processor in a multicore environment. Synchronizing the access calls on this shared memory were easy to synchronize, as one could use single bus and therefore control the order of accesses. [Vinb, p. 14]

Since only one processor can access the memory at a time, every other processor has to wait until it is allowed to access the shared memory. This slows the whole system down or can even makes it unresponsive.

Different shared memory solutions in one-physical-units, like modern multicore processors, are still present today but the handling of the memory changed to increase the performance.

First distributed shared memory approaches were realised, by using physically sepa-

rated systems linked via via a quick connection over short distances. These systems were cheaper than multicore ones and also allowed sharing each others memory while the drawbacks of one-bus based shared memory approaches were weakened.

Examples for this development are *Dash*, *Memnet* and *Plus*. [NL91, p. 52ff]

2.2.2 Software based DSM

These hardware solutions were later replaced by software approaches, where read and write accesses to memory locations were interpreted and transmitted to the distant computing nodes. These remote nodes themselves interpret these accesses and transfer the requested part of memory to the asking node.

Examples are *Intel Ivy*, *Lina* and *Mermaid*. Intel's Ivy thereby was the first page based DSM system but had a rather poor performance.[NL91, p. 52ff]

2.2.3 Implementation possibilities

Since this thesis implements a software based solution, some common approaches for an implementation are explained and compared in this section.

2.2.3.1 Object based

The idea of object based DSM is to encapsulate shared data into an object. By providing functions, one can access values and variables of that object.

This solution can be implemented in any environment, as it does not depend on the systems memory management.

By providing an object and interfaces, the approach is not transparent to applications using it. Instead of relying on the methods of this object, one could instead directly use Message Passing. [Vina, p. 5]

2.2.3.2 Shared space

By using a shared space, every node has its own private memory and additionally one piece, which is shared across all of them. This additional piece of memory must be placed on every node and continuously synchronized between all systems in the DSM.

Also, like the object based solution, it does not manipulate the virtual memory management and therefore can also be used in any environment.

But in comparison to the object based solution, this approach provides the possibility to use read and write access on a specific portion of the memory.

2.2.3.3 On-demand-paging

DSM can also be implemented by using the virtual memory management, provided by the OS and adding a custom pager. One would manipulate the DSM part of the virtual memory, to call the custom pager, if read or write faults occur. The pager itself would establish a connection to that remote node, where the requested page is present, fetch it and resolve the page fault. [Vina, p. 5]

Using on-demand-paging for DSM seems to be the rational approach, as it provides two main advantages, compared to the other possibilities:

- The paging process stays completely transparent to the application, when accessing a remote memory location.
- Libraries can be used, as if the application is working on local memory.

2.3 Memory Consistency

One of the biggest problems of distributed shared memory is the memory consistency or coherence. In a message passing system the remote values are received and updated on demand, or via a publish-subscribe system. But in a distributed shared memory approach, memory pages are received from a remote host, stored via a cache on the

local machine and must be updatet via an, often called, Memory Consistency System (MCS). A vast amount of proposed algorithms and ideas exist but MCS is not part of this thesis. [LH89, p. 1]

3 Introduction to Fiasco.OC and Genode

This chapter gives an introduction to the software, the implementation will be running on. First the Fiasco.OC kernel is explained. Afterwards, a general overview of Genode is given, followed by some basic concepts and libraries. These are provided by Genode and will be used for the implementation of the DSM system.

3.1 Fiasco.OC

Fiasco.OC is a L4 based microkernel and can be used for embedded programming, up to HPC.

According to [TUD07] L4 describes a wide range of microkernels, as it was adopted by many people, since the first development by Jochen Liedtke. But all implementations share some basic ideas:

- Only absolutely necessary functionality should be placed in kernel code, everything else should be written as user code
- No policy but pure mechanisms should be deployed in the kernel

The Fiasco.OC has support for multiprocessors in both - x86 and ARM architecture - and allows paravirtualization and hardware assisted virtualization (AMD SVM, Intel VT and ARM Hyp). [TUD15a]

By using an object-oriented capability system, it offers access to kernel objects. Owning a capability basically means, that one has access to the specified kernel object. Each application has its own capability table, where the capabilities are stored but is allowed and able to pass capabilities to other applications on the same host and therefore share

the access to kernel objects. [TUD15b]

It is a microkernel, which can be used for real-time applications, since it achieves low latency on interrupts, due to being fully preemptible in comparison to other L4 microkernels.

3.2 Genode OS Framework

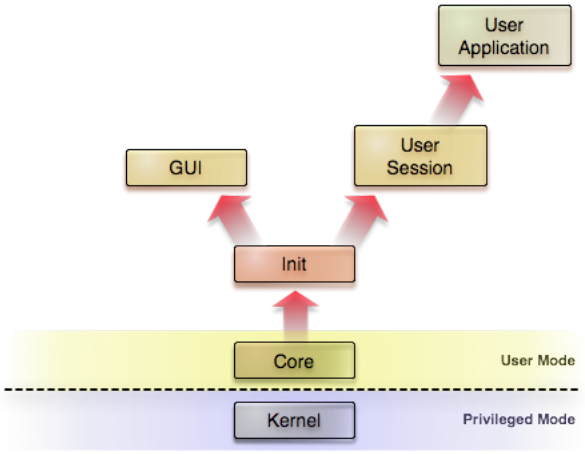


Figure 3.1: The basic hierarchical structure of Genode [Fes15]

The Genode OS Framework is a developer tool-kit, which provides a basis for special purpose operating systems. Because it is highly secure and efficient in resource usage, it can be used from small embedded systems, up to highly dynamic general-purpose tasks.

The Genode OS Framework supports different kernels, not only microkernels, but also monolithic ones like Linux. By using a microkernel like the Fiasco.OC, the system is likely to be much more secure, because of the small trusted computing base and the Genode OS Framework was especially designed for the usage with L4 based microkernels. [Gen15]

3.2.1 Limited resources

Current general purpose operating systems are designed, to use the physical resources in the best possible way. By providing the illusion of unlimited resources to every application, one must uphold this illusion in case of insufficient memory, by swapping memory pages of inactive processes on the hard drive.

Genode instead uses a different approach and limits the available resources from the beginning. When creating an application, it only gets a specific amount of memory but can increase that by asking its parent for more. This is the so called Quota System of Genode. [Fes15] It provides more transparency and control of the available resources, compared to the classic approach where an application can allocate as much memory as it wants and in some cases lead to crashes and slow behavior of the complete system.

3.2.2 Core services

Genode provides a lot of services and concepts as a simple API and therefore it is easy to make use of them. In general, they are a good starting point for every project. In the following, some concepts and services are described, which were used for the development of a basic DSM system.

3.2.2.1 RAM Service

The RAM service provides (abstracted) access to physical memory, meaning one can allocate memory in form of dataspace capabilties.

By using the `alloc(...)` function, one can obtain physical memory and by calling the `free(...)` function, give it back again.

This is only possible if the quota of the session has enough memory left. `quota()`, `used()` and `avail()` give information about the current quota status. The parent of the current session can transform some of its quota to its childs with the `transfer_quota(...)` function.

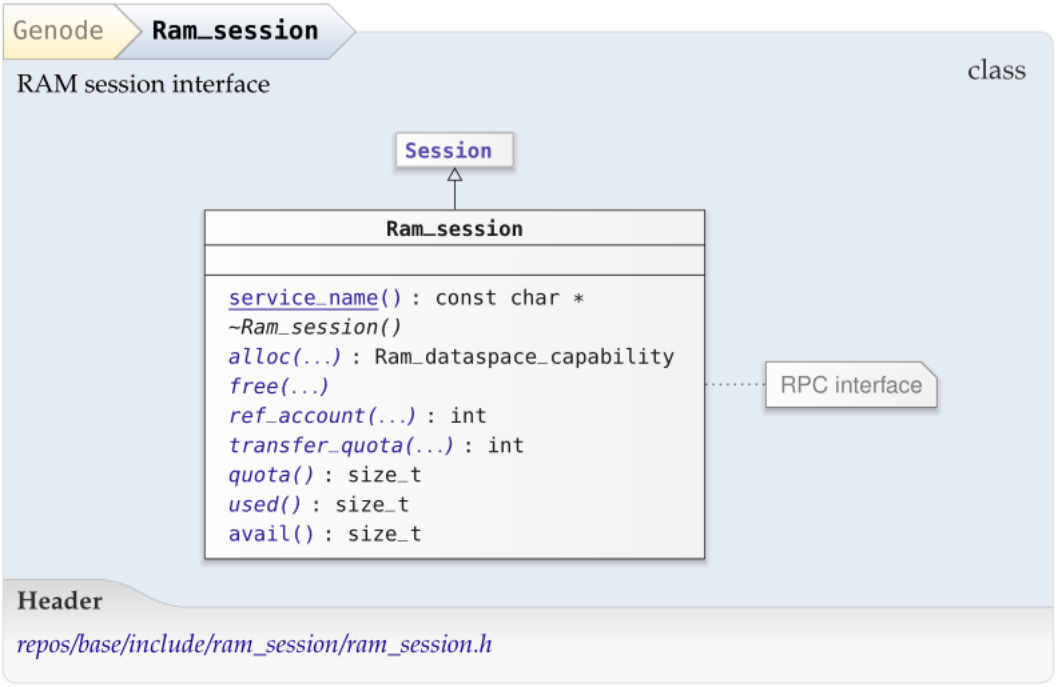


Figure 3.2: The RAM session interface [Fes15]

3.2.2.2 RM Service

The Region Manager (RM) service manages the address-space layouts, in which one can attach dataspace capabilities in the address-space of the current RM session.

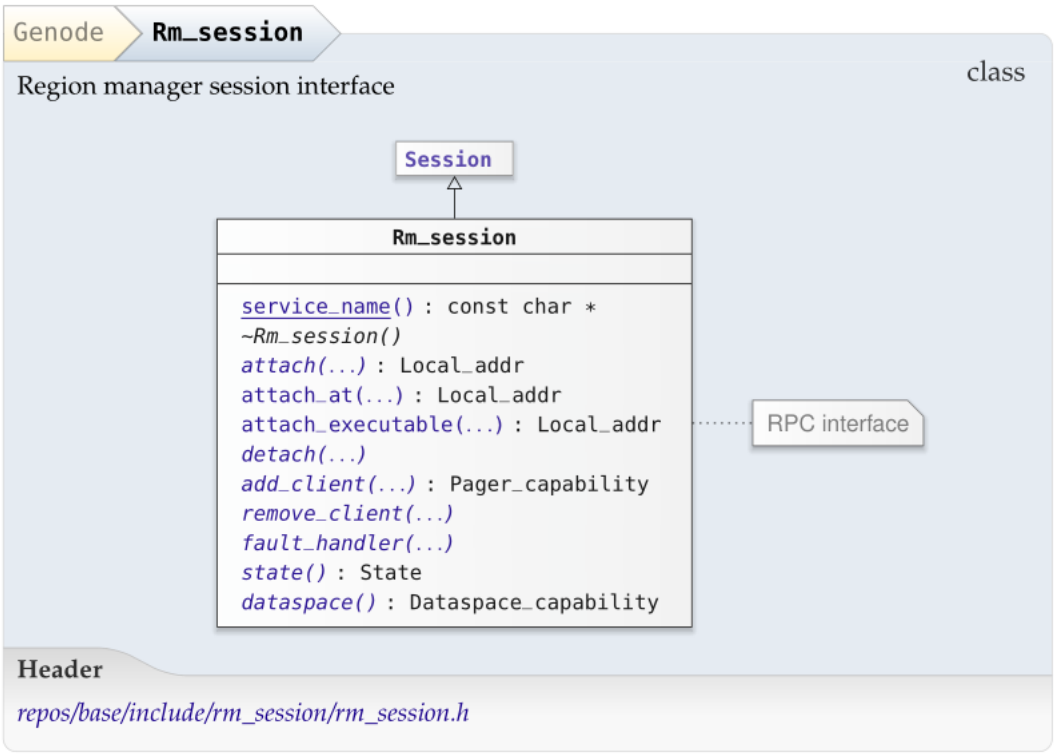


Figure 3.3: The RM session interface [Fes15]

Two methods are provided to do this. One for attaching it at the first possible position in the memory, where enough address space is free with the `attach(...)` function. But it is also possible to attach a dataspace capability at a specific location in the memory by using the `attach_at(...)` function. To remove the dataspace object from the RM session, the `detach(...)` function is provided.

With the release of Genode version 8.11, a new feature was introduced to the RM session interface. The RM session now also provides a function, to assign a custom page handler to the RM session by the `fault_handler(...)` function.

genode/repos/base/include/rm_session/rm_session.h

```
43      struct State
44      {
45              /**
46               * Type of occurred fault
47               */
48              Fault_type type;
49
50              /**
51               * Fault address
52               */
53              addr_t addr;
54
55              /**
56               * Default constructor
57               */
58              State() : type(READY), addr(0) { }
59
60              /**
61               * Constructor
62               */
63              State(Fault_type fault_type, addr_t fault_addr) :
64                      type(fault_type), addr(fault_addr) { }
65      };
```

The state of a RM session was added to describe, if a page fault occured, what type of page fault it is and at which address the fault emerged.

The four possible fault types can be seen here:

genode/repos/base/include/rm_session/rm_session.h

```
30      enum Fault_type {
31              READY = 0, READ_FAULT = 1, WRITE_FAULT = 2, EXEC_FAULT = 3 };
```

3.2.3 Concepts

Genode already comes with a wide range of implemented concepts. Since we will need a few concepts for our implementation, the most important ones are explained in the following subsections.

3.2.3.1 Shared Memory

As one obtains dataspace capabilities as representation of physical memory, these can also be used to establish a shared memory between two applications.

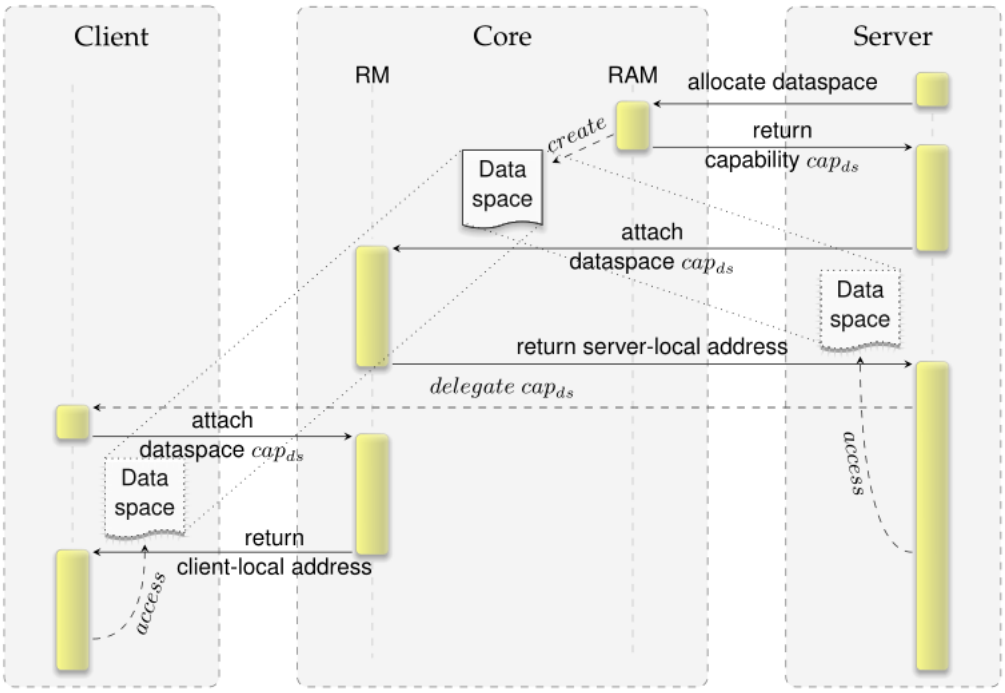


Figure 3.4: The process of establishing a shared memory between two applications. [Fes15]

Therefore one application is obtaining a dataspace capability, passes this capability to another application and both can attach the dataspace capability in their RM session

and therefore have access to the same physical memory.

3.2.3.2 Managed Dataspaces

These new functions and variables in the RM session lead to the development of the new, so called managed dataspace concept. Prior to Genode version 8.11, a dataspace object always was a representation of physical memory. As it is now possible, to handle page faults, this dataspace $\leftrightarrow$ physical memory context is soften up a bit and RM sessions now also can return themselves as dataspace capabilities, as seen in figure 3.3. These returned dataspace capabilities can of course be attached in every RM session.

Once someone accesses memory, which has no physical representation, the page fault handler of this address space is called. If no function was provided as fault handler, an error occurs and the page fault stays untreated.

3.2.3.3 Thread

Already available in Genode is an implementation of threads. One has to implement a class, which inherits the thread class and override the `entry()` method, which is the entry function called in the thread.

3.2.3.4 Client-Server

Genode makes heavy usage of the Client-Server concept. An application in Genode can offer services in the form of a Client-Server, which means another application can call functions of the server as a client. A detailed example can be found in the hello_tutorial directory in the genode repository. [Döb13]

3.2.4 lwIP library

lwIP, which stands for light weight ip stack, is the default TCP/IP stack in Genode. It was especially designed to reduce resource usage, while still providing the complete TCP/IP protocols. This makes it perfect for the usage in embedded systems. As reference on how to use the library, the exmaples from the Genode repo can be used. [Fes13]

4 General Approach & Concepts

This chapter will introduce all ideas and concepts developed, while researching possible ways to extend Genode by the concept of a software based DSM system. The ideas are ordered chronological from the early ideas until the final approach. Since Genode is not a widely spread operating system like Windows or Linux, part of the knowledge about the inner workings of Genode was obtained by contacting developers over their IRC channel (#*genode* on freenode), searching in the mail archive [08] and reading the Genode Book [Fes15].

One of the first ideas was to manipulate dataspace capabilities, by introducing a new dataspace capability type and resolve access requests, similar to page fault handling. This would extend the virtual RAM of an application and therefore be a transparent solution. It would also allow the usage of libraries, directly operating on memory pointers. But as this would require major changes in the Genode basis, other ideas were developed, while dropping this one.

The next idea was to use a *trampolin* like effect. By storing instruction codes in specific memory cells, the broker would execute them and replace the instructions with the requested value or return a pointer with the destination of the values. The third idea, establishing a DSM-Class was a similar approach, compared to the second idea. This Class would offer read and write functions with parameters to address a specific external Node and a page Offset. But both approaches also share the same disadvantage, as they wouldn't transparently integrate into the virtual memory of an application. This also means, that using functions operating on pointers are not possible.

The final and implemented solution picks up the first solution. By using the concept of managed datapsaces and custom page fault handling, there is no need to manipulate dataspaces, as one can extend the virtual memory with the managed dataspaces.

5 Establishing a coding basis

In this chapter, all necessary steps to establish a coding basis are explained. Starting with the installation of all needed dependencies, we will continue to build all applications and explain their usage in the thesis. After dependencies and applications are installed, Genode will be downloaded and prepared for further development.

5.1 Preparing the OS

It is suggested to build and run Genode on a Linux based OS, so for development and building, xubuntu 14.04 LTS was used.

Some dependencies have to be installed, to allow the compilation for every component:

```
1 sudo apt-get install make libSDL-dev tclsh expect byacc genisoimage
2     autoconf2.64 autogen bison flex g++ git gperf libxml2-utils subversion
3     xsltproc libncurses5-dev libexpat1-dev texinfo gawk vde2
```

5.2 Qemu

Qemu will be used to simulate the pbxa9 (a sample ARM) hardware. As the Qemu version available in the official repositories of Ubuntu is not compiled with VDE support and seems to be buggy, Qemu needs to be compiled from the source code. Because only ARM support is needed for the simulation, we exclude every other target platform to speed up the compilation process. `./configure --help` provided the necessary information.

```
1 sudo apt-get install libvde-dev libvdeplug-dev wget
2 sudo apt-get build-dep qemu
3 wget http://wiki.qemu-project.org/download/qemu-2.5.0.tar.bz2
4 tar xfvj qemu-2.5.0.tar.bz2
5 cd qemu-2.5.0
6 ./configure --enable-vde --target-list=arm-softmmu
7 make
8 sudo make install
```

5.3 Virtual Distributed Ethernet

In order to let two or more QEMU instances communicate with each other over ethernet, Virtual Distributed Ethernet (VDE) is used. VDE is a set of tools and programs, to simulate ethernet hardware. QEMU provides direct support for VDE when it is enabled at compile time.

As only a simple ethernet switch is needed for the implementation, one is created in the tmp directory of the host running QEMU [12]:

```
1 vde_switch -s /tmp/switch1
```

5.4 Preparing Genode

To get the source code of Genode, one can simply clone the git repository:

```
1 git clone https://github.com/genodelabs/genode.git
```

A new folder is created with the title genode. From now on, this folder will be referred as <genode-dir>!

To work with the currently latest release, one must change the branch from `master` to the tag `15.11`:

```
cd <genode-dir>
git checkout 15.11
```

The compilation and installation of the Genode toolchain is initiated by:

```
cd <genode-dir>
./tool/tool_chain arm
```

To download additional and necessary external libraries and source code run:

```
cd <genode-dir>/repos/libports
make prepare
cd <genode-dir>/repos/dde_linux
make prepare
cd <genode-dir>/repos/base-foc
make prepare
```

The next step is to create a build directory for genode. The path to this direcotry will be referenced as <genode-build>:

```
cd <genode-dir>
./tool/create_builddir foc_pbxa9 BUILD_DIR=<genode-build>
```

The following lines need to be uncommented in `<genode-build>/etc/build.conf`:

```
REPOSITORIES += $(GENODE_DIR)/repos/libports
REPOSITORIES += $(GENODE_DIR)/repos/dde_linux
```

5.5 Creating a new Genode repository

Creating applications for Genode can be done by adding a folder to the *<genode-dir>/repo/* directory. In this folder source code, makefiles for the compilation and run configurations are stored. [Fes15]

The build of Genode only searches predefined locations for repositories, therefore the new repository must be introduced to the Build by adding the following line to <genode-build>/etc/build.conf:

```
1 REPOSITORIES += $(GENODE_DIR)/repos/<repo>
```

By calling a run file via make run/<run-file> in the <genode-build> directory, Genode will be built with all necessary dependencies and run the newly created application afterwards.

6 Implementation

In this chapter a prototype is developed. This is done with the usage of the services and concepts described in chapter 3.

The whole implementation can be split into two applications, the Broker, which provides the DSM service and a Dummy, which will use the DSM service.

The primary goal of this thesis was to find out, if Genode and Fiasco.OC - with their current state, are capable of providing the tools and framework to implement such a DSM system. Therefore a MCS is not introduced in this application and the presumption is taken, that only one node writes to a remote page at any time. In order to further reduce the complexity, the current implementation only takes two nodes. This ensures a static system, which provides more scalability for testing.

6.1 Broker

The Broker application serves as a control unit for the DSM. By providing the DSM service to applications on the same node, it has to manage every distant memory and the own portion of memory, which will be added to the DSM.

For each portion of remote memory, information about the size of the memory and the location in the network (IP-address) must be saved.

The implementation is based on pages, which means, exchange of remote memory is done in pages of the size 4KB. This size was chosen accordingly to the minimum page size used by memory allocations from the RAM session.

6.1.1 Memory handling

We have to deal with two different types of storage, when implementing the DSM system. While handling the internal memory is done via the shared memory approach - explained in chapter 3, handling of external memory must be done with a more complex construction.

6.1.1.1 Internal memory

In order to provide some of the storage of one node as shared memory for other nodes, the Broker must implement a shared memory approach. This is done by allocating some its RAM

```
1 Genode::env()->ram_session()->alloc(1 * 1024 * 1024};
```

which returns a new dataspace capability, representing the internal part of the DSM.

The Broker is now able to share this capability with applications on its own node and thereby producing the first part of the DSM.

Applications can integrate that capability into their own RM session, by calling

```
1 Genode::env()->rm_session()->attach(<dataspace_capability>);
```

In a dynamic implementation, the Broker also would either need to broadcast the size of the memory it provides to external nodes, or send that information to a Command & Control unit in the DSM system. By doing so, remote nodes can also access that memory by querying read and write calls to the Broker.

6.1.1.2 External memory

Next to the internal memory, represented as shared memory, there also is external memory, which is provided by making use of the managed dataspace concept.

```
1 enum { MANAGED_DS_SIZE = 64 * 1024 * 1024 };
2 Rm_connection rm(0, MANAGED_DS_SIZE);
```

This code creates a new RM session (and therefore a managed dataspace), with an empty address space and therefore no physical counterpart. [Gen10] The Broker can now return the dataspace capability of that RM session, by calling

```
1 Dataspace_capability ds = rm.dataspace();
```

Just as with the internal memory, this dataspace capability can now be passed to every application, wanting to access the DSM.

6.1.2 Providing the DSM service

At the current state, the Broker is capable of creating dataspace capabilites, representing either internal or external memory. These capabilties now need to be passed to applications, who want to access the DSM.

This is done by using the Client-Server concept (section 3.2.3.4), to delegate dataspace capabilities to applications. Like with the RM session or RAM session, any application can implement functions, which can be called as services from other applications on the same host.

Therefore in the implementation, the Broker works as the Server and provides two functions as services.

```
1 Dataspace_capability getRemoteDataspaceCapability();
2 Dataspace_capability getLocalDataspaceCapability();
```

6.1.3 Adding a custom page fault handler

The current code would lead to unresolved page faults when trying to access external memory. In order to handle these page faults and resolve them correctly, a custom pager must be added to the external memory implementation. First, the Broker needs to run

```
1 Signal_receiver rec;
2 Signal_context client;
3 Signal_context_capability sig_cap = rec.manage(client);
4 rm.fault_handler(sig_cap);
```

The signal receiver gets notified, when a page fault occurs (via the blocking `wait_for_signal();` function) and resolves it according to its signal context.

6.1.4 Resolving page faults

As access to the local available shared memory does not need and special treatment, this section describes the way remote memory pages are exchanged between the nodes.

6.1.4.1 Memory page exchange protocol

The current protocol, implemented for exchanging memory pages, is a message constructed out of three main parts:

- 1 Byte: Read/Write Flag
- 4 Byte: Offset, in order address each pages in one shared memory block
- (4096 Byte: The page's content to be written on the remote node)

The read/write flag indicates whether a node wants to read a page, or if it wants to write a page to a remote node. As pages are always in a size of 4KB, the offset describes

the position of the page, in the portion of shared memory, provided by the remote node. The last part of the message, the page itself, only needs to be added, when a node wants to write a page to a remote node.

6.1.4.2 Reading a page

After the `READ_FAULT` page fault occurred at the Dummy and the remote page was fetched by the Broker, a new 4KB sized memory is allocated from the RAM session by the Broker. By shortly attaching the memory page to the Broker's RM session, it gets filled with the content from the remote page and detached afterwards. This cache page - filled with the remote content - now gets attached in the managed dataspace, the external memory RM session. Automatically the Dummy can now access the requested page and continue reading from that address.

6.1.4.3 Writing a page

When a `WRITE_FAULT` page fault occurs at the Dummy, the Broker allocates a new 4KB sized memory page from its RAM session. This cache page gets attached in the managed dataspace, the external memory RM session. The Dummy is now able to continue writing to that page.

After the Dummy is done writing to that page, it needs to be transported to the remote node by using the exchange protocol.

At the second node, the write request, the offset and the content of the page is received. The Broker on the second node now copies the content of the page into the according physical memory by using a memory copy function.

6.2 Dummy

The implementation was focused on providing applications a DSM service with as much transparency as possible.

Therefore only this lines

genode/repos/dsm/src/dummy/main.cc

```
20    // establish connection to broker
21    Broker::Connection bc;
22
23    /*
24     * get local shared memory capability
25     * and include into rm session
26     */
27    Dataspace_capability dsc = bc.getLocalDataspaceCapability();
28    char *ptr_node1 = env()->rm_session()->attach(dsc);
29
30    /*
31     * get remote shared memory capability
32     * and include into rm session
33     */
34    dsc = bc.getRemoteDataspaceCapability();
35    char *ptr_node2 = env()->rm_session()->attach(dsc);
```

needs to be inserted into any application using the DSM service.

These lines establish a connection to the Broker and call the two functions provided by it. `getLocalDataspaceCapability();` returns the local memory, the shared memory between the Dummy application(s) and the Broker. `getRemoteDataspaceCapability();` returns the remote memory, the managed dataspace of the second node in the DSM.

After these calls, one is able to make normal use of the pointers, received after attaching the capabilities into the RM session of the application.

7 Showcases

This chapter will show how to run the DSM implementation and explain the included showcases.

7.1 Compile & Run

First the OS and Genode needs to be prepared according to chapter 5, afterwards the download gets initiated:

```
1 git clone https://github.com/702nADOS/Genode-DSM-AW-BA.git
2     <genode-dir>/repos/dsm
```

Because Genode has problems to include all files of lwIP, an additional line with the absolute path to the include directory is needed in the *target.mk* file of the broker:

```
1 INC_DIR += <genode-dir>/repos/libports/include/lwip
```

After creating a build directory and launching a VDE switch, according to chapter 5, one can start the two example nodes by calling

```
1 make run/node1
2 make run/node2
```

in the <genode-build> directory.

Note: node1 must be running, before executing node2!

7.2 Showcase 1: Locally available memory

The dummy application on node 1 first does a read and write access test on the local shared memory.

```
1 strcpy(ptr_node1, "Hello␣Node1!");
2 printf("Read␣\"%s\"␣from␣%p!\n", ptr_node1, ptr_node1);
```

After compilation and execution

```
1 [init -> dummy] int main(int, char**): Writing to local shared memory!
2 [init -> dummy] int main(int, char**): Wrote "Hello␣Node1!" to 100000!
3 [init -> dummy] int main(int, char**): Reading from local shared memory!
4 [init -> dummy] Read "Hello␣Node1!" from 100000!
5 [init -> dummy] int main(int, char**): Writing to remote shared memory!
```

is obtained. Lines with the `int main(int, char**):` prefix are debugging messages. As expected no page fault handling is involved in this type of access, as the Dummy operates on Memory, locally available.

7.3 Showcase 2: Remotely available memory

In the second part, the dummy application on node 1 tries to access part of the memory, which is physically available on node 2.

While the access implementation is done in the same way, as with the locally available memory,

```
1 strcpy(ptr_node2, "Hello␣Node2!");
2 printf("Read␣\"%s\"␣from␣%p!\n", ptr_node2, ptr_node2);
```

the debugging output on node 1 differs:

```
1  Genode::Pager_entrypoint::entry()::<lambda(Genode::Pager_object*)>:
2      Could not resolve pf=200000 ip=100102c
3  [init -> broker] void DSM::external_memory::handle_fault():
4      DSM::external_memory:: rm session state is WRITE_FAULT, pf_addr=0x0
5
6  [...]
7
8  Genode::Pager_entrypoint::entry()::<lambda(Genode::Pager_object*)>:
9      Could not resolve pf=200000 ip=538ec
10 [init -> broker] void DSM::external_memory::handle_fault():
11     DSM::external_memory:: rm session state is READ_FAULT, pf_addr=0x0
```

We can see, that when the dummy application tried to access the memory address, when a page fault occured and the handling routine got called. By the Broker, the memory page is now copied to the second node. We can verify that, by reading from the memory address on node 2:

```
1  printf("%s", ptr_node1);
```

This call will return

```
1  [init -> dummy2] Hello Node2!
```

in the debugging output of the second node. As the memory page is locally stored on the second node, no output regarding page fault handling is observable as expected.

8 Limitations and Possibilities for Extension

The goal of a DSM system was successfully implemented with a solution, which is nearly completely transparent to the applications using it. This means, that Genode is capable providing the needed tools for Distributed Shared Memory.

Still there are some points, which could be implemented in a cleaner and much more reliable way:

- By now, the implementation only works with two static nodes, and the IP addresses are configure via a run file. A better solution would be to add a Command & Control node, which knows how much nodes are there and broadcasts information about them to every other DSM node.
- In order to not provide each distant memory via its own capability to the application, the better way would be, to create one external memory managed dataspace. One additional layer of managed dataspace is then added for each distant memory into the one global managed dataspace.
- Currently the pages received from remote nodes are not cached. For testing purposes only new memory was allocated, the page was written in there and this dataspace capability was attached to the address, the page fault occured.
- Getting remote pages is already slow with a simple message passing system. As the DSM system adds even more complexity to the data exchange, it is slower if you don't provide mechanisms to prefetch data and already have it cached. By doing so, applications read more from local available memory/cache, without the need to often request distant memory.

- To be able to determine if an application was done writing to a distant memory cell was not achieved by this implementation. One possible solution would be, to use a mutex - still this would decrease the transparency to the application, because only it knows, when it is done writing.
- The implementation currently does not handle consistency in any way, which means the Broker is not able to determine if cached pages are outdated or not.

Acronyms

DSM Distributed Shared Memory.

ECU Electronic Control Unit.

MCS Memory Consistency System.

RAM Random Access Memory.

RM Region Manager.

VDE Virtual Distributed Ethernet.

List of Figures

Bibliography

[08] *Mailing Lists*. http://sourceforge.net/p/genode/mailman/?source=navbar, Last Accessed: 2016-01-13. 2008.

[12] *VDE Basic Networking*. http://wiki.virtualsquare.org/wiki/index.php/VDE_Basic_Networking, Last Accessed: 2016-01-10. 2012.

[AG96] S. V. Adve and K. Gharachorloo. "Shared memory consistency models: A tutorial." In: *computer* 29.12 (1996), pp. 66–76.

[AW04] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons, 2004.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. *Munin: Distributed shared memory based on type-specific memory coherence*. Vol. 25. 3. ACM, 1990.

[CBZ95] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. "Techniques for reducing consistency-related communication in distributed shared-memory systems." In: *ACM Transactions on Computer Systems (TOCS)* 13.3 (1995), pp. 205–243.

[CKK95] J. B. Carter, D. Khandekar, and L. Kamb. "Distributed shared memory: Where we are and where we should be headed." In: *Hot Topics in Operating Systems, 1995.(HotOS-V), Proceedings., Fifth Workshop on*. IEEE. 1995, pp. 119–122.

[CL95] M. Cierniak and W. Li. *Unifying data and control transformations for distributed shared-memory machines*. Vol. 30. 6. ACM, 1995.

[Döb13] B. Döbel. *hello_tutorial*. https://github.com/genodelabs/genode/tree/master/repos/hello_tutorial, Last Accessed: 2016-01-02. 2013.

[Fes13] N. Feske. *genode repository - lwip examples*. https://github.com/genodelabs/genode/tree/master/repos/libports/src/test/lwip, Last Accessed: 2016-01-13. 2013.

[Fes15] N. Feske. *GENODE Operating System Framework 15.05*. 2015.

[Gen08] Genode Labs. *Release notes for the Genode OS Framework 8.11*. `http://genode.org/documentation/release-notes/8.11#Managed_dataspaces__experimental_`, Last Accessed: 2016-01-03. 2008.

[Gen10] Genode Labs. *Release notes for the Genode OS Framework 10.11*. `http://genode.org/documentation/release-notes/10.11#On-demand_paging`, Last Accessed: 2016-01-03. 2010.

[Gen15] Genode Labs. `http://genode.org/about/`, Last Accessed 2016-01-14. 2015.

[HP11] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[Jin10] L. Jin. "Software-oriented distributed shared cache management for chip multiprocessors." PhD thesis. University of Pittsburgh, 2010.

[JKW95] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. *CRL: High-performance all-software distributed shared memory*. Vol. 29. 5. ACM, 1995.

[Kar09] B. Karp. *Distributed Shared Memory: Ivy*. `http://www0.cs.ucl.ac.uk/staff/B.Karp/gz03/f2011/lectures/gz03-lecture5-Ivy.pdf`, Last Accessed: 2016-01-11. 2009.

[Lab15] G. Labs. *Genode OS Framework release 15.11 Nov 30, 2015*. `http://genode.org/news/genode-os-framework-release-15.11`, Last Accessed: 2016-01-10. 2015.

[Lee02] C. Lee. "Distributed Shared Memory." In: *Proceedings on the 15th CISL Winter Workshop Kushu, Japan February*. Citeseer. 2002.

[LH89] K. Li and P. Hudak. "Memory coherence in shared virtual memory systems." In: *ACM Transactions on Computer Systems (TOCS)* 7.4 (1989), pp. 321–359.

[Lu+95] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. "Message passing versus distributed shared memory on networks of workstations." In: *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*. IEEE. 1995, pp. 37–37.

[NL91] B. Nitzberg and V. Lo. "Distributed shared memory: A survey of issues and algorithms." In: *Distributed Shared Memory-Concepts and Systems* (1991), pp. 42–50.

[PTM98] J. Protic, M. Tomasevic, and V. Milutinović. *Distributed shared memory: Concepts and systems*. Vol. 21. John Wiley & Sons, 1998.

[TUD07] TU Dresden. *What is L4*. `https://wiki.tudos.org/What_is_L4`, Last Accessed: 2016-01-13. 2007.

[TUD15a] TU Dresden. *Fiasco Features*. `https://os.inf.tu-dresden.de/fiasco/features.html`, Last Accessed: 2016-01-13. 2015.

[TUD15b] TU Dresden. *Fiasco.OC & L4Re*. `https://os.inf.tu-dresden.de/L4Re/doc/l4re_intro.html`, Last Accessed: 2016-01-13. 2015.

[Vina] B. Vinter. *Distributed Shared Memory*. `http://www.diku.dk/hjemmesider/ansatte/vinter/cc/Lecture11DSM.pdf`, Last Accessed: 2016-01-11.

[Vinb] B. Vinter. *Shared Memory SMP Architectures and Programming*. `http://www.diku.dk/~vinter/cc/Lecture2SMP.pdf`, Last Accessed: 2016-01-11.

[Wad11] D. Waddington. *Simple custom pager*. `http://sourceforge.net/p/genode/mailman/message/27799605/`, Last Accessed: 2016-01-13. 2011.