Course of Study:	Computer Science

Examiner:	Prof. Dr. rer. nat. Klaus Lagally
Supervisor:	Dr. rer. nat. Thomas Schöbel-Theuer

Thesis No. 2251

Transparently distributed ATHOMUX

Hardy Kahl <hardy.kahl@gmx.net>

Institute of Formal Methods in Computer Science (FMI)
Operating Systems Group
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Commenced:	2004-12-15
Completed:	2005-06-14
CR-Classification:	D.4.0, D.4.7, D.1.m

The goal of this thesis is to add transparently distributed system functionality to the ATHOMUX operating system. As a proof of concept, we present prototype implementations of network related bricks, most notably `remote_*` and bricks generating network transparent views. Building upon this, we demonstrate that the goal of this thesis can be achieved.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Objective 2
1.3 Structure 2

2 Concepts **4**
2.1 Transparently distributed ATHOMUX 4
2.2 Creating views 9
2.2.1 Global views 10
2.2.2 Connected views 12
2.2.3 Network transparent views 13
2.3 Sharing nests 14
2.4 The `mirror` brick 15
2.5 Building upon Linux sockets 17
2.5.1 Traditional streams 17
2.5.2 Logical streams 17
2.5.3 Physical streams 19
2.5.4 Mapping logical streams to TCP streams 20
2.6 Issues in the existing implementation 20
2.6.1 Memory nests 21
2.6.2 The state of bricks 21

3 Implementation **23**
3.1 Linux Sockets in ATHOMUX 23
3.1.1 `device_socket_ulinux` 23
3.1.2 `device_tcp_client_ulinux` 24
3.1.3 `device_tcp_listen_ulinux` 24
3.1.4 `device_tcp_server_ulinux` 24
3.2 Streams 25
3.2.1 `adapt_strmr` and `adapt_strmw` 25
3.2.2 `adapt_strmr_packet` and `adapt_strmw_packet` 26
3.2.3 `pstrm_duplex` and `pstrm_simplex` 27
3.2.4 `strm_duplex` and `strm_simplex` 27
3.2.5 `adapt_strm` 28
3.2.6 `strm_multiplex` 29

3.3 Pipes and queues 29
3.3.1 pipe 30
3.3.2 queue 30
3.4 Multiuser capabilities 30
3.4.1 adapt_multi 31
3.4.2 adapt_strm_multi 31
3.5 remote 31
3.5.1 remote_client 32
3.5.2 remote_server 33
3.5.3 remote_client_tcp and remote_server_tcp . 33
3.5.4 remote_server_socket 34
3.6 mirror 35
3.6.1 mirror_client_1_tcp 35
3.6.2 mirror_server_1_tcp 35
3.7 Network transparency 36
3.7.1 strategy_merge 36
3.7.2 strategy_netconnect 37
3.7.3 strategy_nettransparent 37
3.7.4 strategy_dot 37
3.7.5 strategy_display 38
3.8 Miscellaneous bricks 39
3.8.1 adapt_complete2 39
3.8.2 merge_lock 39
3.8.3 map_delta 39
3.8.4 adapt_tmp 40
3.9 Foreign bricks 40
3.10 Dependencies 40
3.11 Pseudo-statelessness 45
3.12 Assumptions 46
3.13 Workarounds 46
3.14 Runtime environment 47

4 Demonstration 48
4.1 Instantiating and running demo bricks 48
4.2 Helper bricks 49
4.2.1 strategy_msconnect 49
4.2.2 strategy_stratconnect 50
4.3 Dynamic nests 51
4.4 Pipes 51
4.5 TCP connection establishment 52
4.6 Remote test 53
4.7 Network transparency 54
4.8 Mirror test 63

5 Conclusion **64**
5.1 Summary 64
5.2 Future work 64

Bibliography **66**

List of Figures

2.1 strategy_merge brick 11
2.2 Connecting server and client parts 13
2.3 Hiding a remote brick 13
2.4 Removing wires in a network transparent view 14
2.5 remote brick 15
2.6 mirror brick with two inputs and two outputs 16
2.7 Writing and reading a stream 18

3.1 device_socket_ulinux 23
3.2 device_tcp_client_ulinux 24
3.3 device_tcp_listen_ulinux 24
3.4 device_tcp_server_ulinux 25
3.5 adapt_strmr 25
3.6 adapt_strmw 26
3.7 adapt_strmr_packet 26
3.8 adapt_strmw_packet 27
3.9 pstrm_simplex 27
3.10 pstrm_duplex 27
3.11 strm_duplex 28
3.12 strm_simplex 28
3.13 adapt_strm 28
3.14 strm_multiplex 29
3.15 pipe 30
3.16 queue 30
3.17 adapt_multi 31
3.18 adapt_strm_multi 31
3.19 remote_client 32
3.20 remote_server 33
3.21 remote_client_tcp 34
3.22 remote_server_tcp 34
3.23 remote_server_socket 34
3.24 mirror_client_1_tcp 35
3.25 mirror_server_1_tcp 36
3.26 strategy_merge 37
3.27 strategy_netconnect 37

3.28 strategy_nettransparent 37
3.29 strategy_dot 38
3.30 strategy_display 38
3.31 adapt_complete2 39
3.32 merge_lock 39
3.33 map_delta 39
3.34 adapt_tmp 40
3.35 dynamic brick dependencies 43
3.36 static brick dependencies 44
3.37 static and dynamic brick dependencies 45

4.1 strategy_msconnect 49
4.2 msconnect example 50
4.3 strategy_stratconnect 50
4.4 stratconnect example 51
4.5 demo_map dependencies 51
4.6 demo_pipe dependencies 52
4.7 demo_pipe2 dependencies 52
4.8 demo_pps dependencies 53
4.9 demo_ppc dependencies 53
4.10 demo_rpcs dependencies 54
4.11 demo_rpcc dependencies 54
4.12 demo_strats dependencies 55
4.13 demo_stratc_init dependencies 56
4.14 demo_stratc dependencies 56
4.15 demo_merge dependencies 57
4.16 local view on demo_strats 58
4.17 local view on demo_stratc_init 58
4.18 local view on demo_stratc 59
4.19 local view on demo_merge 60
4.20 merged and connected view 61
4.21 network transparent view 62
4.22 demo_mirror dependencies 63

List of Tables

2.1 List of all elementary operations 5
2.2 Signatures of all elementary operations 9

3.1 List of all implemented bricks 43

Chapter 1

Introduction

ATHOMUX is a prototype implementation of a new architecture for operating system which is based on two abstractions called *bricks* and *nests*. The programming method of this architecture is called *instance orientation*. These concepts were proposed by Thomas Schöbel-Theuer. A more detailed description can be found in [1, 2, 3, 4].

The development of ATHOMUX is at its beginning, but functionality is increasing steadily. Currently, it is relying on Linux as a host operating system. However, first attempts to adapt Linux device drivers were already made by Jens-Christian Korth [5]. Florian Niebling [6] examined ways of running POSIX programs in ATHOMUX.

Within the scope of this work, distributed systems functionality is added to the ATHOMUX project.

1.1 Motivation

In traditional programming methods, software components usually have many different interfaces. That is why these components are not arbitrarily combinable. Object oriented methods alleviate this effect by introducing abstract interfaces. However, the system still resembles a puzzle where only a small subset of all parts fits in another part.

Instance orientation tries to overcome these limitations of traditional programming methods. The *nest* abstraction can be viewed as an address space with a number of operations defined on it, comparable to sparse files in UNIX. The components of functionality are called *bricks* and act as transformers between nest instances. A brick instance can have several inputs and outputs. It transforms the nest instances on its inputs to nest instances that are offered on its outputs which can be in turn the input of other brick instances. Therefore a certain relationship between brick instances is implicitly given. In ATHOMUX this relationship is represented by *wires*.

Bricks themselves have no control about their relationship to other bricks. This is called *anonymity of relation*, resulting in increased modularity. The wiring is done by special *strategy bricks*.

The operations defined on nests represent a universally generic interface. Therefore, the wiring can change dynamically and the number of valid wirings is not limited by incompatible interfaces. This enables to better code reuse.

Functionality not only grows with the number of bricks. Much functionality comes from different ways of combining bricks. This is comparable to a pipe-and-filters style of programming [7], but avoiding consuming semantics. Instead of separating consumer and producer, passive resources (e.g. memory) may live on one side, and active resources (e.g. CPU) on the other side. Concerning combinability, the system now is similar to the LEGO®[1] toy system.

With this new architecture, ATHOMUX tries to closely integrate the ideas of organic computing [8], database systems [9] and distributed systems into a new operating system.

1.2 Objective

This thesis focuses on the part of distributed systems. Its goal is to add transparently distributed system functionality to the ATHOMUX operating system.

- At first, prototypes of bricks demonstrating distributed systems functionality should be implemented. In particular, `remote` bricks make nests available over the network to other nodes[2], as well as simple variants of the `mirror` brick.
- A network transparent view of this distributed system should be generated to show that ATHOMUX can be transparently distributed.

Bricks should be stateless wherever it is feasible, so that they can be migrated easily to other nodes by means of dynamic reinstantiation and rewiring.

1.3 Structure

This thesis is organized as follows:

- In Chapter 2 the prerequisites and concepts are described.

[1] LEGO® is a trademark of the LEGO Group of companies

[2] In this context, a node is an instance of the ATHOMUX operating system. Note that multiple ATHOMUX instances can run on one machine.

- Chapter 3 introduces the implemented bricks. A short description is given for each brick. Furthermore, problems which occurred during the implementation are discussed.
- Chapter 4 validates the implementation and gives a demonstration of its functionality.
- Chapter 5 concludes this thesis and gives proposals for future work.

Chapter 2

Concepts

In this chapter, the necessary concepts are developed in a top-down approach, beginning with network transparent views, and ending with communication methods and the gateway to the Linux host operating system.

2.1 Transparently distributed ATHOMUX

First of all, the expression “transparently distributed” means that the system is distributed. It can consist of many nodes that communicate and interact with each other. The word “transparently” means that the location of these nodes is hidden. One can access the system as if there was only one node. This is called network transparency.

In ATHOMUX, any necessary functionality is offered by using the universally generic nest interface. That includes communication between brick instances and even views of the system. In order to achieve network transparency, it is sufficient to consider this interface. Table 2.1 shows a list of all elementary operations defined on nests, and table 2.2 on the next page shows their signatures. A detailed description can be found in [10].

operation	**description**
`$trans`	Transfer of data from physical to logical memory and vice versa
`$wait`	Wait until pending transfers are completed
`$get`	Allocate physical buffer for a logical memory area
`$put`	Signal that a previously allocated physical buffer is not needed any more
`$lock`	Request a lock for data or address space
`$unlock`	Release any locks for a region
`$gadr`	Reserve an address region
`$padr`	Unreserve an address region

continued on next page...

...continued from previous page

operation	**description**
`$create`	Create a defined area in logical address space
`$delete`	Create a hole in logical address space
`$move`	Move a logical memory area to a different address
`$output_init`	Initialize and deinitialize an output
`$input_init`	Initialize and deinitialize an input
`$instbrick`	Instantiates and initializes a brick
`$deinstbrick`	Deinstantiates and deinitializes a brick
`$instconn`	Instantiates an input or output
`$deinstconn`	Deinstantiates an input or output
`$connect`	Connect an input with an output
`$disconnect`	Removes a connection
`$getconn`	Inquire an input or output about its connected partner
`$findconn`	Find an input or output with a certain param value
`$retract`	Recollect resources previously given to other bricks

Table 2.1: List of all elementary operations

operation	**signature**

`$trans`	`(addr_t log_addr,`
	`len_t log_len,`
	`paddr_t phys_addr,`
	`direction_t direction,`
	`prio_t prio := prio_normal)`
	`=>`
	`(success_t success,`
	`plen_t phys_len)`

`$wait`	`(addr_t log_addr,`
	`len_t log_len,`
	`bool forwrite := FALSE)`
	`=>`
	`(success_t success,`
	`paddr_t phys_addr,`
	`plen_t phys_len,`
	`version_t version)`

continued on next page...

...continued from previous page

operation	**signature**
`$get`	`(addr_t log_addr,`
	`len_t log_len,`
	`bool forwrite := FALSE)`
	`=>`
	`(success_t success,`
	`paddr_t phys_addr,`
	`plen_t phys_len,`
	`version_t version)`
`$put`	`(addr_t log_addr,`
	`len_t log_len,`
	`prio_t prio := prio_none) =>`
	`(success_t success)`
`$lock`	`[mandate_t mandate]`
	`(addr_t log_addr,`
	`len_t log_len,`
	`lock_t data_lock := lock_write,`
	`lock_t addr_lock := lock_read,`
	`addr_t try_addr := log_addr,`
	`len_t try_len := log_len,`
	`action_t action := action_wait)`
	`=>`
	`(success_t success,`
	`addr_t try_addr,`
	`len_t try_len)`
`$unlock`	`[mandate_t mandate]`
	`(addr_t log_addr,`
	`len_t log_len,`
	`addr_t try_addr := log_addr,`
	`len_t try_len := log_len)`
	`=>`
	`(success_t success,`
	`addr_t try_addr,`
	`len_t try_len)`

continued on next page...

...continued from previous page

operation	**signature**

`$gadr`	`(len_t log_len,`
	`bool reader := FALSE,`
	`bool exclu := TRUE,`
	`action_t action := action_wait,`
	`len_t try_len := log_len)`
	`=>`
	`(success_t success,`
	`addr_t log_addr,`
	`len_t log_len)`

`$padr`	`(addr_t log_addr,`
	`len_t log_len,`
	`bool reader := FALSE)`
	`=>`
	`(success_t success)`

`$create`	`(addr_t log_addr,`
	`len_t log_len,`
	`bool clear := FALSE,`
	`bool melt := TRUE)`
	`=>`
	`(success_t success)`

`$delete`	`(addr_t log_addr,`
	`len_t log_len,`
	`bool melt := TRUE)`
	`=>`
	`(success_t success)`

`$move`	`(addr_t log_addr,`
	`len_t log_len,`
	`offs_t offset,`
	`offs_t offset_max := offset)`
	`=>`
	`(success_t success,`
	`offs_t offset)`

`$output_init`	`(bool destr,`
	`bool constr,`
	`bool clear := FALSE)`
	`=>`
	`(success_t success)`

continued on next page...

...continued from previous page

operation	**signature**
`$input_init`	`(bool destr, bool constr, bool clear := FALSE) => (success_t success)`
`$instbrick`	`(addr_t log_addr, name_t name, bool constr := FALSE, bool destr := FALSE) => (success_t success)`
`$deinstbrick`	`(addr_t log_addr, bool destr := TRUE) => (success_t success)`
`$instconn`	`(struct conn_info *conn1, bool clear := FALSE, bool constr := TRUE, bool destr := FALSE) => (success_t success)`
`$deinstconn`	`(struct conn_info *conn1, bool destr := TRUE) => (success_t success)`
`$connect`	`(struct conn_info *conn1, struct conn_info *conn2) => (success_t success)`
`$disconnect`	`(struct conn_info *conn1) => (success_t success)`
`$getconn`	`(struct conn_info *conn1, struct conn_info *res_conn, index_t conn_len) => (success_t success, index_t conn_len)`

continued on next page...

...continued from previous page

operation	**signature**

`$findconn`	`(struct conn_info *conn1,`
	`struct conn_info *res_conn := NULL,`
	`index_t conn_len := 0)`
	`=>`
	`(success_t success,`
	`index_t conn_len)`

`$retract`	`(prio_t prio,`
	`addr_t log_addr := 0,`
	`len_t log_len := (len_t)-1,`
	`addr_t try_addr := log_addr,`
	`len_t try_len := log_len)`
	`=>`
	`(success_t success)`

Table 2.2: Signatures of all elementary operations

The `remote` brick which is introduced in 2.3 on page 14 consists of a server and a client part communicating with each other in order to transparently forward operation calls to a remote nest instance. This can be compared to traditional *Remote Procedure Calls*. With the help of this brick, nest instances can be accessed from other nodes as if they were locally present.

However, strategy nests provide information about instantiated bricks and their wiring which includes information about instantiated `remote` bricks. `remote` instances represent the boundaries of a node. They have to be hidden in order to achieve network transparency. This information resides in the contents of strategy nests and is not considered by the `remote` brick which only forwards operation calls.

The following sections describe how network transparency can be achieved in both cases.

2.2 Creating views

In ATHOMUX, views are special nests called strategy nests. They are created by `control_simple` instances, for example. When `$trans` is issued in read mode at a certain address, an ASCII string is returned. It describes the brick instance at that address, including its wiring. The information about all bricks together can represent the actual physical wiring of the system, or it may be a virtual view. Examples for virtual views are redundancy transparent or network transparent views.

In contrast to network transparent views which describe a distributed system, local views describe the brick instances and their wiring on one node. Again, this can be the physical wiring or a virtual view.

The operations defined on strategy nests not only allow read access. It is also possible to modify the view (e.g. instantiating bricks or wiring instances).

Furthermore, multiple views can exist in parallel. Starting with the view of physical wiring provided by the control brick, it can be transformed by adaptor bricks to virtual views.

Generating network transparent views is one of the main tasks of this thesis. If these views behave like local views, it will actually imply a transparently distributed system. Two bricks can then be wired like in a local view, even if they exist on different nodes. New `remote_*` bricks will automatically be instantiated in order to connect the two bricks.

The task of creating network transparent views can be split into three parts:

1. Merge the local views of all nodes in the distributed system into one global view.
2. The connection between the client and server parts of `remote` instances has to be represented in the system view. Therefore, corresponding server and client parts are merged to a virtual `remote` instance.
3. Finally, the virtual `remote` instances are replaced by wires, thus hiding node boundaries.

These three steps are described in detail in the following paragraphs.

2.2.1 Global views

In order to generate a global view, the local views have to be available. `remote` bricks can be used to access the strategy nests on other nodes. They are introduced later in section 2.3 on page 14. The merging procedure itself could be done by a brick, called `strategy_merge`, that requires local views on its inputs and offers the global view on its only output (see figure 2.1 on the following page).

Considering read access, merging views means that every single brick instance in the local views is also visible in the global view. It is possible that brick instances on different local views occupy the same logical address. This conflict can be resolved by assigning virtual addresses to those brick instances on the global view. The virtual address can be incremented with each processed brick instance on the local views. That way, it can be guaranteed that every brick instance gets a conflict-free address.

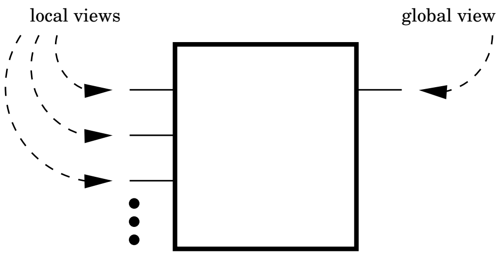

Figure 2.1: `strategy_merge` brick

Write access to the global view is more complicated. In general, all operations accessing brick instances on the global view have to be routed to the corresponding address in the corresponding local view. Therefore, for each brick instance the association to an input nest has to be memorized in addition to the address mapping. However, some operations require extra consideration:

`$gadr` and `$instbrick`: To which input should these operations be forwarded? On which node should the address range be reserved, and respectively on which node should the brick be instantiated? There is no way to know for sure. One could implement different strategies for a decision. Another way is to give the operations a parameter that specifies the desired input nest or node. However, this would partly destroy the desired network transparency.

`$connect`: A connection establishment usually involves two different brick instances. If these instances are located on the same node, the operation can be forwarded regularly to the corresponding input nest. In case of location of instances at different nodes, `remote_*` bricks have to be instantiated in order to represent the connection. For this reason, `strategy_merge` needs to know the network addresses of each node which is connected to one of its inputs. This information can be given to `strategy_merge` upon initialization of its inputs.

In every view, there is a special virtual brick instance at address 0x00 called `ATHOMUX_MAINDIR`. It has only outputs but does not implement any operations. Since it is always expected at address 0x00, it cannot be assigned any other address. Therefore, all `ATHOMUX_MAINDIR` instances have to be merged by building the union of all outputs.

Views are not static. At any time, brick instances can be instantiated, deinstantiated or rewired by parallel accessors. There are situations where consistency of a view is not that important, however, usually it is. Therefore, two methods that ensure consistency in the `strategy_merge` brick are introduced:

- One could trigger a rebuild of the global view when it is needed. This can be signaled by a `$lock` operation call on the output of `strategy_merge`.

Still, the local views could change during the building process. To ensure the validity of the local views, a read lock for the whole address space in every input nest has to be requested. These locks can be released when the global view is not needed any more which can be signaled by an `$unlock` operation call on the output. This method rebuilds the global view every time from scratch, however only when it is needed.

- Changes in the local views can be detected automatically by using locks and the retract mechanism. First, `strategy_merge` acquires read locks on the whole address space in every input nest. As in the first method, this ensures the validity of the local views. The global view is then built once from scratch. When changes are made in the local views, write locks for the concerning regions have to be acquired. Since the held read locks cover the whole address space, they are in conflict with the requested write locks. That causes retract calls. `strategy_merge` releases the read locks for the given regions. Right after that, it requests a read lock again. Once the modifications have been made it eventually gets these read locks. Then, the global view can be adapted to the modifications. There is no need for rebuilding the view from scratch. However, even if the global view is not needed, this method adapts it every time a modification in a local view is made.

These methods can ensure consistency only if all involved bricks use locking correctly.

2.2.2 Connected views

In the global view, the `remote_server_*` and `remote_client_*` instances are still separated. We have to find out which client instance is connected to which server instance. Then these two instances can be merged to a new virtual `remote` instance.

Upon instantiation, `remote_*` instances are given an identification number. This number matches on connected server and client instances.

With this identification number it is possible to find the server and client parts that belong together. They have to be removed from the connected view and replaced by a new virtual `remote` instance. The wires on all inputs of the client parts and on all outputs of server parts have to be rerouted to the virtual instance. Figure 2.2 on the next page shows this process.

When a `$deinstbrick` call occurs on one of these virtual instances, the corresponding client and server parts have to be deinstantiated.

In order to ensure validity of the provided connected view, methods as described in 2.2.1 on the preceding page can be used.

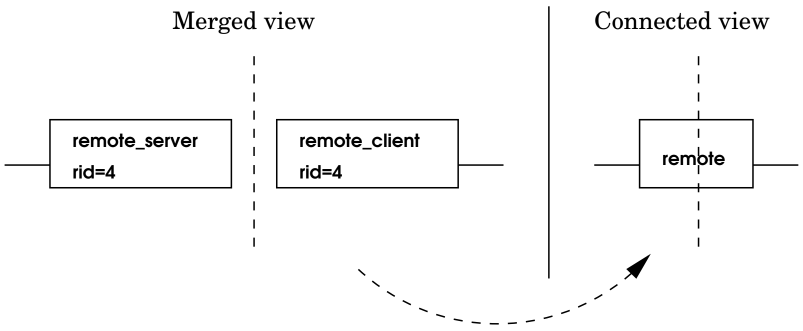

Figure 2.2: Connecting server and client parts

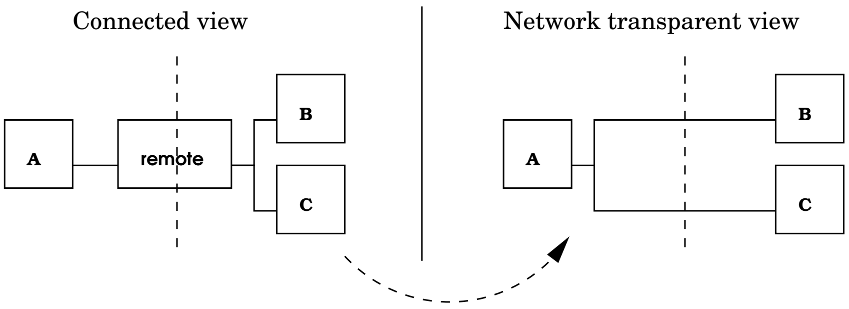

Figure 2.3: Hiding a `remote` brick

2.2.3 Network transparent views

This is the last step in creating network transparent views. We already have a connected view. Only the virtual `remote` instances still indicate a network connection. In this last step, these instances are simply reduced to wires.

When a virtual `remote` instance is removed from the network transparent view, all the connections on its input and all the connections on its output have to be rewired. It has to be done in a way so that the cartesian product of the two sets is built. This means that every brick that is connected with an input of the virtual instance has to be wired to every brick connected to an output, and vice versa. This might introduce many new wires. An example is shown in figure 2.3. The dashed line denotes node boundaries.

The `$disconnect` operation has to treat these new wires differently from regular wires. One of these wires given to `$disconnect` represents two wires in the input nest. However, these two wires can be represented by multiple wires in the network transparent view. Therefore, a wire in the input nest should only be removed if there are no related wires in the network transparent view left. Case $a)$ in figure 2.4 on the following page shows this situation. The wire from brick A to brick B is removed in the network transparent view. In the connected view,

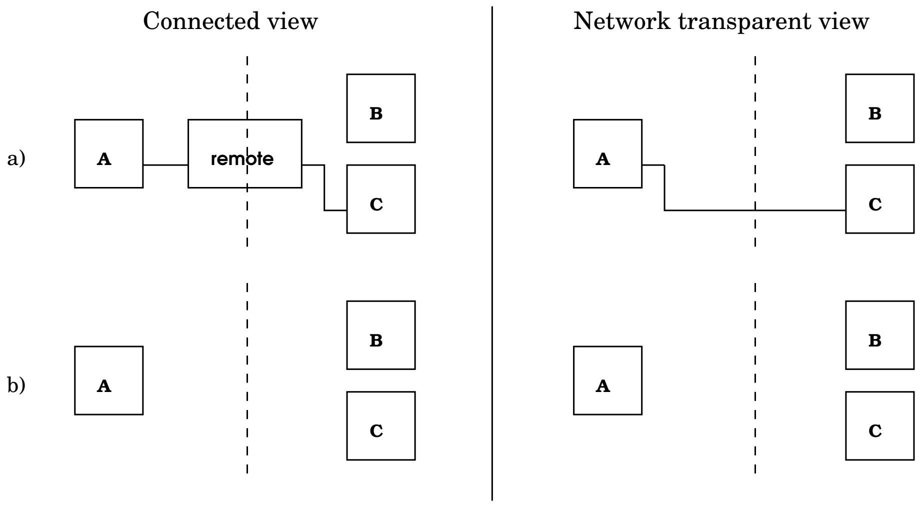

Figure 2.4: Removing wires in a network transparent view

the wire from the `remote` brick to brick B is removed accordingly, however, the wire from brick A to the remote brick is still necessary.

After all connections introduced by a certain virtual `remote` instance have been removed, this instance has to be removed as well. This situation is shown in case *b)* in figure 2.4. The last wire is removed in the network transparent view. Thus, the `remote` brick in the connected view along with its remaining wires have to be removed as well.

In order to ensure validity of the provided network transparent view, methods as described in 2.2.1 on page 11 can be used.

2.3 Sharing nests

The `remote` brick is a virtual brick which may only exist in virtual views like the connected view. In reality, it consists of two bricks, the client and the server[1] part, with a network connection between them. Figure 2.5 on the next page shows a virtual `remote` brick. The dashed line denotes node boundaries.

The nest instance on the server's output is made available on the input of the client part. One could say that a nest instance is shared by two nodes.

[1] The *client* and *server* naming comes from the number of operations offered on inputs and outputs. Operation calls may occur on inputs just as well as on outputs. However, outputs offer a lot more functionality than inputs. Therefore the preferred direction of operation calls is from outputs towards inputs. One could say that functionality is offered or served on the left side (provided that the inputs are drawn on the left). That is why the left part of the `remote` brick is named server and the right part client.

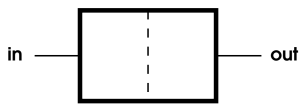

Figure 2.5: `remote` brick

Thanks to anonymity of relation and an universal generic interface, it is possible to insert `remote` instances anywhere in the system in order to make certain nest instances available on other nodes.

In general, the `remote` brick is similar to traditional remote procedure calls. Operation calls from both sides are marshalled and sent over the network. On the other side, the packets are decoded and the corresponding operation calls are executed. Finally, the result is marshalled and sent back again.

However, there are a few cases where special precaution is needed:

- The operation `$trans` gets a physical address as parameter pointing to a buffer. In read mode data is transferred to that buffer whereas in write mode data is transferred from that buffer. When marshalling, this data has to be considered as well.
- Physical addresses must not be transmitted over the network since they are only valid at a local machine. Therefore, `$get` and `$put` have to be handled separately from other operations. In the case of `$get`, a temporary buffer is created on the client side and returned with the flag `version` set to `vers_undef` indicating that the data is not the newest. The rest is done by `$trans`. The buffer can be deleted after the corresponding `$put`. However, if the parameter `prio` is not set to `prio_none`, the content of the buffer has to be transferred to the server first. This can be done by explicitly calling `$trans`.
- Currently, there is no agreement concerning the allocation of mandates in a distributed ATHOMUX system. A solution would be to assign disjunct ranges of mandates to each node.

In order to separate the transport mechanism from the marshalling procedure, the `remote_*` bricks write marshalled operation calls to a logical stream. In section 2.5 on page 17, we will show how a logical stream can be sent over the network.

2.4 The `mirror` brick

The `mirror` brick is a virtual brick as well. Unlike `remote` it may connect more than two nodes, but has similar functionality: It shares a nest instance among

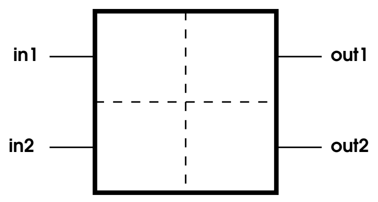

Figure 2.6: `mirror` brick with two inputs and two outputs

several nodes.

In the most simple case of having one input and one output, `mirror` is reduced to a `remote` brick. Once there are multiple inputs or outputs it gets interesting.

Having one input and multiple outputs means that there is one server and multiple clients. The client part could simply be a `remote_client`. Respectively, the server part could consist of multiple `remote_server` bricks. However, relying on point to point connections is not very efficient. The `mirror` brick should greatly benefit from the symmetric optional locking approach and group communication [11, 12, 13].

As soon as there are multiple servers, a certain replication model has to be defined. One extreme would be using no replication at all. Another possibility would be to replicate everything on every node. That means every input-nest of the `mirror` brick has the same contents. One could even think of models that lie between the two extremes like those described in some of the RAID[2] concepts.

Once replication is used, the problem of different data versions occurs. Many consistency models and synchronization methods are available. Again, the `mirror` brick is not restricted to any of them.

As we can see, many variants of the `mirror` brick implementing different strategies are possible. It depends on what kind of replication and consistency model [14] is chosen. Most of the research field of distributed systems is contained within this brick. A simple `mirror` brick with two inputs and two outputs is shown in figure 2.6. It is not apparent what kind of replication and consistency model it implements.

When generating network transparent views, `mirror_*` instances have to be treated like `remote_*` instances. Server and client parts have to be merged to a virtual `mirror` instances which finally have to be replaced by wires. Note that there can be more than one server and more than one client that belong together.

In this work, we are building upon the point to point communication methods which are also used for the `remote` bricks. No group communication is used so

[2]Redundant Array of Inexpensive Disks

far. A simple mirror variant that uses local `remote_*` instances is introduced in 3.6 on page 35.

2.5 Building upon Linux sockets

The `remote_*` and `mirror_*` bricks write the marshalled operation calls to a logical stream. So far, we didn't address the problem of transporting a logical stream over the network.

Since ATHOMUX is still relying upon Linux as a host operating system, we have to interact somehow with the Linux networking concepts. We decided to use TCP streams for communication with other nodes because they are simple, reliable, and order conserving. However, in some cases TCP streams imply too much communication overhead (e.g. mirror bricks). In these cases significant speedups are expected when using other communication methods like UDP broadcast or multicast. These optimizations might be subject for subsequent work.

By using Linux sockets, we can concentrate on the implementation of higher level bricks. However, this dependency on Linux should be confined to as few bricks as possible which map the functionality to the ATHOMUX nest interface. Additionally, these bricks should be as simple as possible to allow easy replacements.

2.5.1 Traditional streams

Streams play an important role in operating systems (pipes, TCP, the three standard streams in UNIX, ...). Usually they are streams of bytes, from the user's point of view. In the case of TCP for example, the underlying transport mechanism splits the stream in packets for sending the data over the network. At application level, the byte stream is often assembled to packets by means of protocols.

The logical streams in ATHOMUX already support packets so that some unnecessary conversions between byte and packet streams can be omitted. It is even possible to transfer holes which gives streams a functionality comparable to sparse files in UNIX.

2.5.2 Logical streams

Logical streams are quite complex compared to TCP streams. The reason is that the reading and writing procedure on logical streams is distributed over several operations, that are `$gadr`, `$padr`, `$create`, `$delete`, `$get`, `$put`, `$trans`, and `$wait`.

Writer:

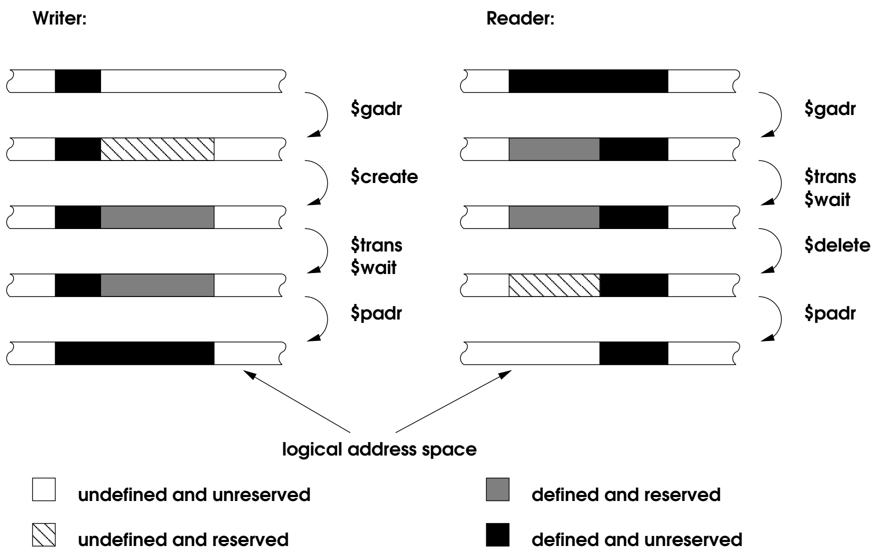

Figure 2.7: Writing and reading a stream

1. `$gadr`: Reserve an area in undefined logical address space.
2. `$create`: Create a defined area.
3. `$trans`: Transfer data from a physical buffer into the previously defined area.
4. `$wait`: Wait until transfer is completed.
5. `$padr`: Unreserve the area in logical address space. This acts like a *commit*.

Reader:

1. `$gadr`: Reserve next area in defined logical address space.
2. `$trans`: Transfer data from logical memory into a physical buffer.
3. `$wait`: Wait until transfer is completed.
4. `$delete`: Throw away the read data.
5. `$padr`: Unreserve the area in logical address space. The data is now consumed.

The operations `$get` and `$put` can also be used instead of `$trans`.

As we can see, in both cases we begin with calling the `$gadr` operation. In order to distinguish between reader and writer, `$gadr` and `$padr` require a parameter called `@reader` indicating the role. The other operations just operate on addresses and are independent of the role.

Figure 2.7 on the page before shows how the reading and writing procedure affects memory regions.

ATHOMUX distinguishes between addresses and memory. An address range can be reserved without reserving any memory behind these addresses. When writing to a stream, first an address range is reserved by calling `$gadr`. The memory at this address range is not yet defined. It has to be defined by calling `$create` before writing to it. The reservation of actual memory is done by `$create`. Holes can be written to the stream by omitting `$create` completely, or by calling `$create` only on subregions of that returned by `$gadr`. Thereby, undefined regions still may exist when calling `$padr` to complete the writing process.

Concurrent access to the stream is possible. However, if lookahead is needed, there is a tradeoff between concurrency and consistency. The `$gadr` operation has a parameter called `@exclu`. By setting it to `FALSE` the reader indicates a lookahead request. Usually, it is a concurrent lookahead, which means that the address range is not reserved for this reader exclusively. Other threads might read and consume that region at the same time. That means that after a reader had issued a lookahead, the data might not be available for consumption any more.

If other semantics are preferred, adaptor bricks can be implemented. For example, locks could be used to prevent other threads from consuming data while doing the lookahead. That implies some reduction of concurrency but high consistency as well.

Another option would be an adaptor brick that uses locks only on the address region that is read. It always consumes, regardless of the `@exclu` parameter, so that other threads can continue reading the following packets. In the case of a lookahead, it buffers the data so that it can be consumed later if desired. If it is not consumed at the time of the `$unlock` call on the corresponding address region, the packet is made available to other readers. This has the advantage of higher concurrency, but also introduces corruption of data ordering.

2.5.3 Physical streams

We decided to introduce a new stream concept, called physical streams, for intermediate mapping. Physical streams are not intended to be broadly used. They were introduced for the purpose of mapping logical streams to TCP streams and should only be used in this context.

Physical streams are based on the `$trans`, and the `$wait` operation. Reading and writing on physical streams works straight forward by calling the `$trans` operation. The logical address, which is given as a parameter, is ignored. Data in the stream has the same order as the corresponding operation calls in time. `$wait` has the usual semantics: It waits until all IO requests are completed.

2.5.4 Mapping logical streams to TCP streams

As noted before, the conversion is done in two steps. At first, the logical stream is mapped to a physical stream.

This task is divided into two parts. Read-only streams are considered separately from write-only streams. A bidirectional stream has to be split into two unidirectional streams before it can be converted, resulting in the advantage that some overhead could be avoided where only unidirectional streams are needed.

Read-only streams:
Upon a `$gadr` call, the required amount of data is transferred from the physical stream to a temporary buffer, residing in a temporary nest instance. All subsequent operation calls are forwarded to that nest instance. After the corresponding `$padr` call the buffer can be deleted. However, if `$gadr` signalled a lookahead, the buffer has to be kept for the next read request.

Write-only streams:
Upon a `$gadr` call, a buffer is created in a temporary input nest. All subsequent operation calls are forwarded to that buffer until the corresponding `$padr` call is issued. The buffer contents then have to be transferred to the physical stream by using the `$trans` operation. The buffer can be deleted afterwards.

Without further precautions, this method destroys all information about packet borders. In order to keep them, packet border information has to be encoded somehow in the physical stream. That can be accomplished by writing a small header that contains the packet length to the physical stream. This has to be done for each packet and prior to the actual data. The reader first extracts the header and with this information it can then reconstruct the original packet.

The second step is the conversion from physical streams to TCP streams. Since physical streams are similar to TCP streams, a mapping can be done almost one-to-one without effort.

2.6 Issues in the existing implementation

During this work, some issues in the existing implementation emerged. It is important to explain these issues in order to understand the implementation and problems that might come with it.

2.6.1 Memory nests

There are two different methods for memory allocation in ATHOMUX: Using a nest with heap semantic and using a nest with nest semantic.

Nests with heap semantic are used for memory allocation at unknown addresses. Therefore, some kind of memory management functionality has to be present. Memory is allocated by first calling `$gadr` to find and reserve a free address range in logical address space. Before it can be used, this address range has to be defined using the `$create` operation. Finally a physical pointer for this address range can be obtained by using the `$get` operation. `$gadrcreateget` corresponds to the classical `malloc()` function. Inputs requiring and outputs offering heap semantic are named "*mem*".

In the case of nest semantic, memory allocation is done just by using `$create` and `$get`. The caller can allocate memory at any logical address within address space. The responsibility for memory management lies on the caller side. Inputs requiring and outputs offering nest semantic are named "*tmp*".

Stateful bricks can use *tmp* or *mem* inputs for storing their state and thus become pseudo-stateless.

2.6.2 The state of bricks

Whenever it is feasible, the implementation of bricks should be stateless so that bricks can easily be migrated. If statelessness cannot be achieved, a pseudo-stateless implementation should be preferred over a stateful implementation to maintain the possibility of migration. We have to distinguish between the migration of bricks and the migration of state.

When migrating a brick, a "flush state" request is issued on the corresponding brick. Afterwards, all state is stored in its input nests. The brick can be deinstantiated, and reinstantiated elsewhere. After the brick has been connected again with the original nests (possibly by using `remote` instances), it can read its state and resume work.

Migrating the state information is another problem. It is easy to copy the contents of a nest instance to another nest instance. However, it is difficult to distinguish between state information of different brick instances which have stored their state in the same nest instance (intermixed). The migration of state of a single brick can be achieved by prepending a kind of logging brick memorizing where state is stored.

The Pointer Cache [10] is a useful tool for implementing pseudo-statelessness: One can allocate memory for state information in a *tmp* nest[3] at known logical address (e.g. at address 0) and use the Pointer Cache to get a physical pointer whenever access to the state information is necessary. All state information is then automatically flushed by calling `PC_FLUSH`. However, there still are some unresolved issues:

- The current implementation of the Pointer Cache guarantees only one valid pointer at a time. A call to `PC_GET` can invalidate any other pointer that was previously obtained.
- When a “flush state” request is issued, pointers that were obtained using the Pointer Cache are invalidated. Other threads might be executing brick code at that time and would crash when using the invalidated pointers.

A solution for both problems would be the explicit releasing of pointers. Thus, a synchronization with `PC_FLUSH` is possible. Pointers stay valid until they are explicitly released, and a `PC_FLUSH` call has to wait until all pointers are released.

[3]Note that state has to be stored at a known hardcoded address. Otherwise new state information will be introduced, which is the address where the state is stored. That is why a nest with heap semantic is not adequate for this purpose.

Chapter 3

Implementation

This chapter describes the implementation of all bricks and how they interact with each other. The implementation was written in a C-like language that was developed particularly for ATHOMUX. A preprocessor [15, 10] transforms it into plain C-code, which can be compiled using an ordinary C-compiler. The ATHOMUX project has its own build system [16] supporting automatic compilation for different runtime environments. At the end of this chapter, the runtime environment and the prerequisites for compilation are described. The ATHOMUX source code is available at the BerliOS Open-Source-Center [17].

3.1 Linux Sockets in ATHOMUX

The following bricks represent the gateway to Linux, regarding socket communication. Currently, all the socket related bricks offer a physical stream output because socket functions and stream operations can be mapped almost one-to-one without effort. Later implementations, running in a native ATHOMUX environment, may offer a logical stream output so that some unnecessary conversions between byte and packet streams can be omitted (see 2.5.1 on page 17).

3.1.1 `device_socket_ulinux`

This brick maps part of the Linux socket functionality to ATHOMUX, in particular the `recv()` and the `send()` function. The `$trans` operation uses them to send data over the network. `$brick_init` expects a valid socket number in `@param` (syntax: “`socket=`<*socket number*>”). The output implements physical stream capabilities.

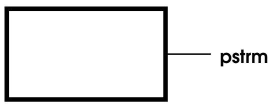

Figure 3.1: `device_socket_ulinux`

3.1.2 device_tcp_client_ulinux

Upon instantiation it creates a TCP connection to a server. `$brick_init` expects the destination address and port in `@param` (syntax: "`host=`<*host name*> `port=`<*port number*>"). It uses `device_socket_ulinux` as a local instance for offering physical stream capabilities on the output. The brick itself is only responsible for establishing the connection. All operation calls on the output are forwarded to the local `device_socket_ulinux` instance.

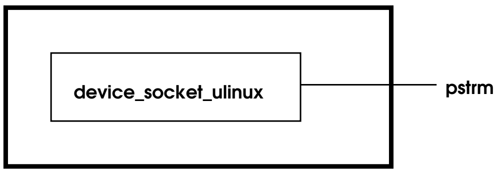

Figure 3.2: `device_tcp_client_ulinux`

3.1.3 device_tcp_listen_ulinux

Upon instantiation it listens on a port for incoming TCP connections. `$brick_init` expects this port number in `@param` (syntax: "`port=`<*port number*>"). An incoming connection can be accepted using the `$trans` operation in read mode. When the operation returns, the read buffer contains the socket number of the new connection. With this socket number a `device_socket_ulinux` brick can be instantiated to access the stream.

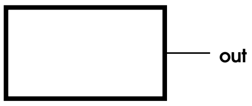

Figure 3.3: `device_tcp_listen_ulinux`

3.1.4 device_tcp_server_ulinux

This brick combines the functionality of `device_tcp_listen_ulinux` and `device_socket_ulinux`. It uses the first one to listen for incoming connections and the second one for stream access. `$brick_init` expects the port number to listen on in `@param` (syntax: "`port=`<*port number*>"). The port number is passed to the local instance of `device_tcp_listen_ulinux`. Any TCP connection on this port is accepted. The socket number for this new connection is passed to the local instance of `device_socket_ulinux` which is used for offering physical stream capabilities on the output.

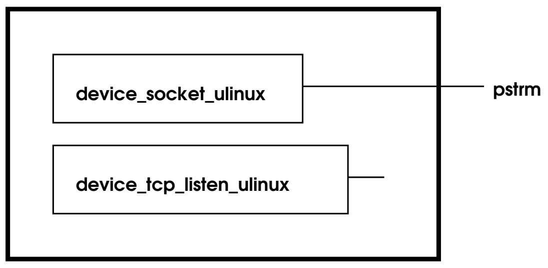

Figure 3.4: `device_tcp_server_ulinux`

3.2 Streams

The following bricks are stream manipulators converting between physical, logical, unidirectional and bidirectional streams. The task of multiplexing is covered as well. Currently, the transfer of holes and lookaheads in logical streams are not supported yet.

3.2.1 `adapt_strmr` and `adapt_strmw`

The conversion from logical to physical streams is covered `adapt_strmr` and `adapt_strmw`. No information about packet sizes is extracted. Packets are created just in the requested size.

The *mem* input is needed for buffering data until the write procedure on the logical stream is completed. Then the buffer contents are transferred to the physical stream.

The *tmp* input is needed for a similar reason. Data that is read from the physical stream is buffered there until the read procedure on the logical stream is completed.

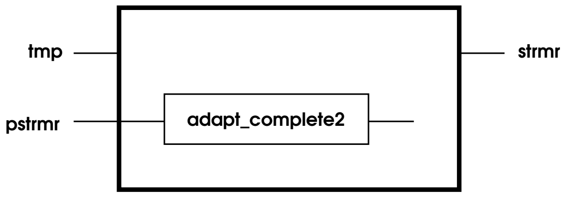

Figure 3.5: `adapt_strmr`

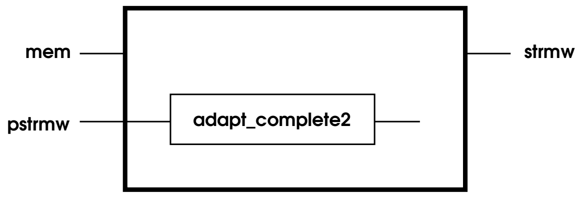

Figure 3.6: `adapt_strmw`

3.2.2 `adapt_strmr_packet` and `adapt_strmw_packet`

Unlike `adapt_strmr` and `adapt_strmw`, these bricks keep the information about packet borders when converting logical to physical streams. When writing to the stream, `adapt_strmw_packet` encodes the the packet borders in the physical stream. `adapt_strmr_packet` decodes them and creates appropriate packets that can be read.

A local instance of `adapt_complete2` is used to ensure that read and write operations on the physical stream either occur completely or fail.

The encoding and decoding protocol uses a header that stores information about the packet size.

```
struct strm_packet_header_t {
  uns4 identification;
  len_t length;
};
```

To every packet written to the stream, this header is prepended. When a packet is read, first the header information is extracted and then the packet of correct size is reconstructed. The identification code is always set to “`PSP `”.

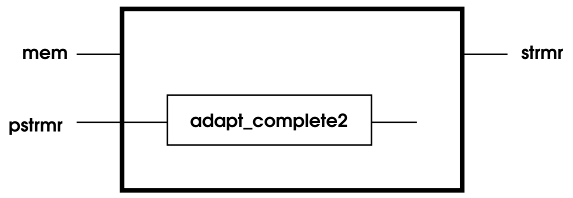

Figure 3.7: `adapt_strmr_packet`

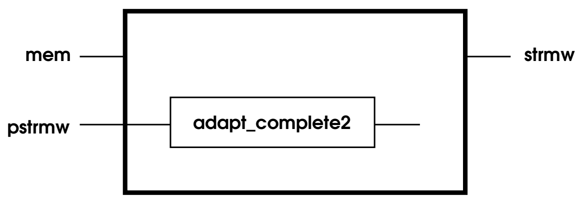

Figure 3.8: `adapt_strmw_packet`

3.2.3 `pstrm_duplex` and `pstrm_simplex`

Concerning physical streams, the task of splitting a bidirectional in two unidirectional streams or respectively merging two unidirectional into one bidirectional stream is accomplished by `pstrm_simplex` and `pstrm_duplex`.

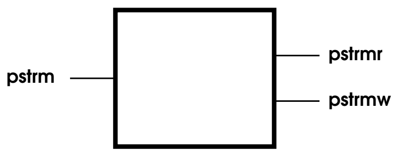

Figure 3.9: `pstrm_simplex`

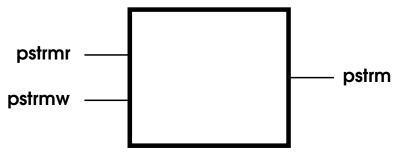

Figure 3.10: `pstrm_duplex`

3.2.4 `strm_duplex` and `strm_simplex`

Like `pstrm_duplex` and `pstrm_simplex`, these two bricks are responsible for splitting and merging streams. The only difference is that they operate on logical instead of physical streams.

Merging logical streams is more complex as in the case of physical streams where read and write functionality is contained within one operation. Here, this functionality is spread over several operations (see [10]), namely `$gadr`, `$padr`, `$create`, `$delete`, `$get`, `$put` , `$trans` and `$wait`. Operation calls from readers could interleave with the ones from writers.

The reader or writer role is determined at the time of `$gadr` and `$padr` calls. The association of the role with other operations is not straightforward.

There are two ways for solving this problem:

- The interleaving can be inhibited right at the beginning.
- The role has to be associated with something else. For example one could use the address range returned by `$gadr` to distinguish the roles. That is possible if the address ranges returned on both streams are, or have been made disjoint.

In this implementation a local instance of `adapt_strm_multi` inhibits the interleaving of operation calls. However, that implies a reduction of concurrency and probably also a reduction of performance.

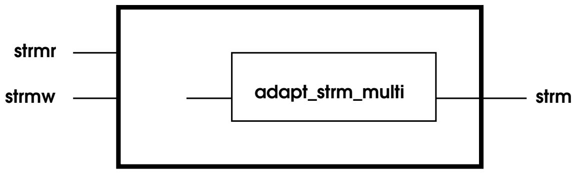

Figure 3.11: `strm_duplex`

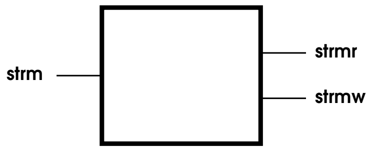

Figure 3.12: `strm_simplex`

3.2.5 `adapt_strm`

This brick does not implement own functionality. It uses local instances of previously introduced bricks for converting a bidirectional logical stream to a bidirectional physical stream.

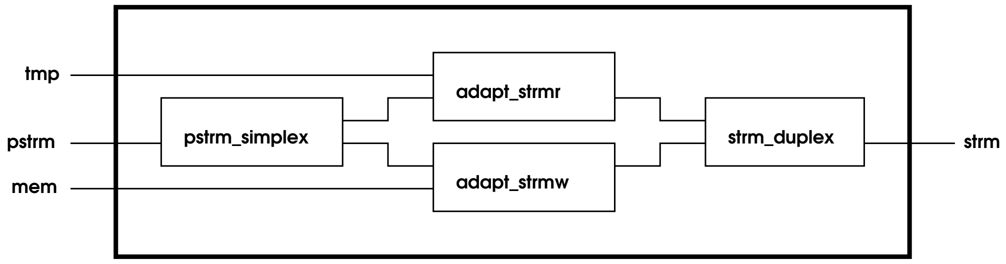

Figure 3.13: `adapt_strm`

3.2.6 `strm_multiplex`

This brick multiplexes two streams. It can be cascaded in order to multiplex more streams. A protocol is used to encode the association of packets with the streams. Every packet that is written is enlarged and gets a header. Respectively, from every packet that is read the header is removed.

```
struct strm_mplex_header_t {
  uns4 identification;
  uns1 channel;
};
```

The header consists of two fields. The first one always contains the character sequence “`SMPX`” as an identification. The field *channel* describes the association to the stream which is either the number 1 or 2.

Since operation calls on both outputs can occur simultaneously, a local instance of `lock_ulinux` is used to achieve mutual exclusion in critical sections.

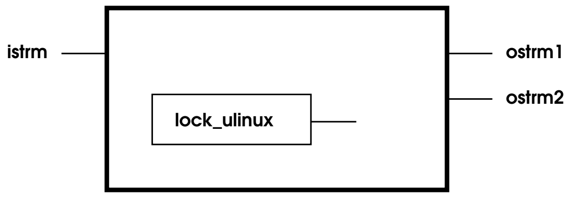

Figure 3.14: `strm_multiplex`

3.3 Pipes and queues

Pipes and queues are FIFO buffers operating on streams. Pipes operate on byte streams and queues on packet streams. The following bricks apply this functionality on logical streams. They have two outputs with unidirectional stream semantics. One for reading and the other one for writing. Threads may be suspended until a request can be completed, e.g. until data is available for reading.

The buffer’s size and the length of the queue are limited and fixed at runtime. Therefore a writing thread might also be suspended until some data is removed (read) from the buffer.

Data written to the *strmw* output is buffered in the *mem* input until it is read on the *strmr* output. The *stat* input is used to store state information when a “flush state” request occurs. The *mem* input cannot be used for this purpose, because nest semantic is needed to store state information (see 2.6.2 on page 21). `$lock` and `$unlock` operation calls are issued on the *ilock* input for synchronization between reader and writer.

3.3.1 `pipe`

The `pipe` is operating on logical streams and interprets them as byte streams. That means that packet borders are lost. The granularity of read and write operations are independent of each other.

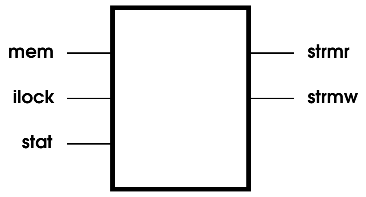

Figure 3.15: `pipe`

3.3.2 `queue`

In a queue the packet borders are kept. Therefore reading granularity must match the writing granularity. The `queue` brick implements this semantic.

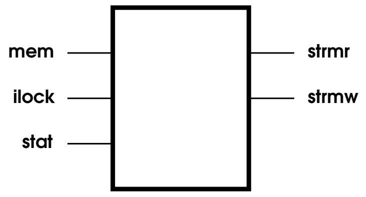

Figure 3.16: `queue`

3.4 Multiuser capabilities

Often bricks implement only singleuser capabilities on their outputs. Multiuser capabilities can be added subsequently by putting adaptor bricks on the outputs. What kind of adaptor brick can be used, depends on the nest semantics and the implementation of the affected brick.

The bricks introduced in this section are using locks for synchronization between threads. Thread mandates are used when requesting locks. A second request on the same address but with a different mandate blocks until the address is unlocked.

As we can see, this implementation functions correctly only if thread mandates are distinct for every thread. Later implementations should not rely on mandates being distinct.

3.4.1 `adapt_multi`

`adapt_multi` prevents that more than one thread enters the brick at the same time. When a thread enters an operation, a lock is set which is released at the end of the operation. Therefore other threads would block in that time.

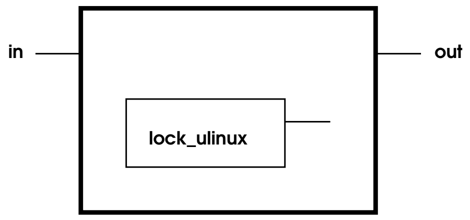

Figure 3.17: `adapt_multi`

3.4.2 `adapt_strm_multi`

In some situation even `adapt_multi` is not restrictive enough, e.g. in the case of logical streams (see `strm_duplex` in 3.2.4 on page 27). `adapt_strm_multi` sets a lock at the beginning of the `$gadr` operation and releases it at the end of `$padr` to protect the whole reading and writing procedure.

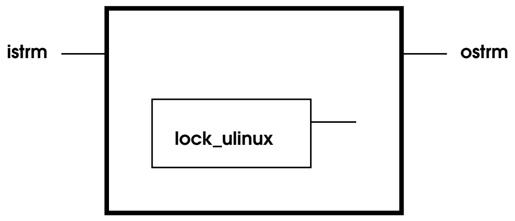

Figure 3.18: `adapt_strm_multi`

3.5 `remote`

The `remote_*` bricks are implementations of the concepts described in the previous chapter.

Output-calls on the client side, and input-calls on the server side are marshalled and transmitted. Additionally, the replies have to be sent back. Each task uses its own stream. There are four streams in total, two in each direction. `strm_multiplex` is used in order to multiplex streams of the same direction so that only two streams or one bidirectional stream is left.

The current implementation always sends the whole `args` structure, regardless of what parameters an operation actually needs. Some bandwidth could be saved by distinguishing between different operations, and by only sending necessary parameters. However, that would also imply a loss of genericity. Support for new operations would have to be added explicitly.

In order to be able to receive operation calls (input-calls on the client side and output-calls on the server side), a new thread is started by the help of `thread_ulinux`. This thread listens constantly on the corresponding readable stream, decodes the received packets and issues the corresponding operation calls. It also sends the results back after the call has returned.

The basic `remote_client` and `remote_server` bricks offer logical stream inputs and outputs which can be wired to bricks handling the transportation over the network. The `remote_*_tcp` bricks show how this can be done in the case of TCP connections.

It is possible to use asynchronous IO. However, it is not implemented in this version. All IO is done synchronously. Considering high network latency, it could be worth the effort of implementing asynchronous IO.

3.5.1 `remote_client`

All operation calls on the output *out* are marshalled and sent regularly, except `$get` and `$put` which are handled locally as described in 2.3 on page 14. The temporary buffers needed for the locally handled operations are allocated in the *tmp* nest.

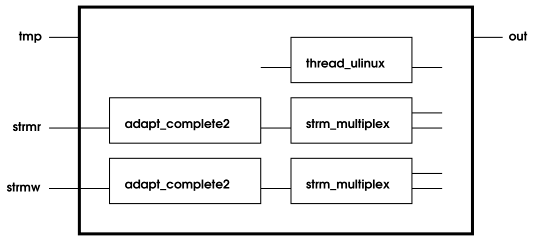

Figure 3.19: `remote_client`

3.5.2 `remote_server`

All operation calls on the input *in* are marshalled and sent with no exception.

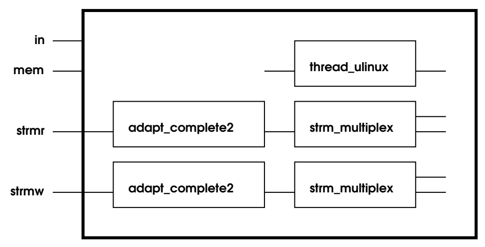

Figure 3.20: `remote_server`

3.5.3 `remote_client_tcp` and `remote_server_tcp`

For simplicity reasons, brick instances handling TCP streams were included in the `remote_*_tcp` bricks. Thus, the visible wiring is kept clear and manageable for the demonstration purpose.

The *server* and *client* parts in the brick names have nothing to do with the TCP connection establishment (see 1 on page 14). A `remote_client_*` could use a local instance of `device_tcp_server` if desired.

In order to establish a connection, the name or address of the destination and the port number have to be passed to the client. The server only needs the port number on which it should listen. This information has to be passed to the operation `$brick_init` encoded in `@param`. In the server case it has to be in the form of "`rid=`<*id*> `port=`<*port number*>", in the other case in the form of "`rid=`<*id*> `host=`<*host name*> `port=`<*port number*>". The additional *rid* parameter is used to identify client and server parts that belong together.

Local instances of `remote_client`, and `remote_server` respectively, are responsible for encoding and decoding operation calls. Adaptor instances convert the logical streams into a physical stream. Finally, the physical stream is connected with a TCP stream by using `device_tcp_*` instances. A `device_mem_ulinx` instance provides memory for buffers.

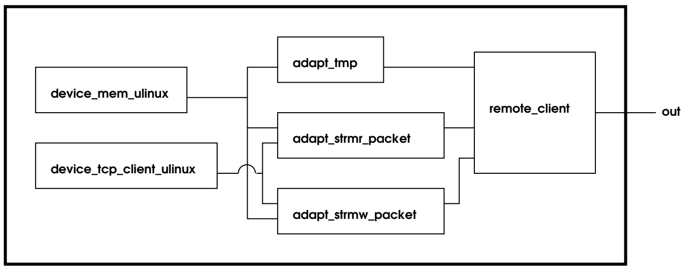

Figure 3.21: `remote_client_tcp`

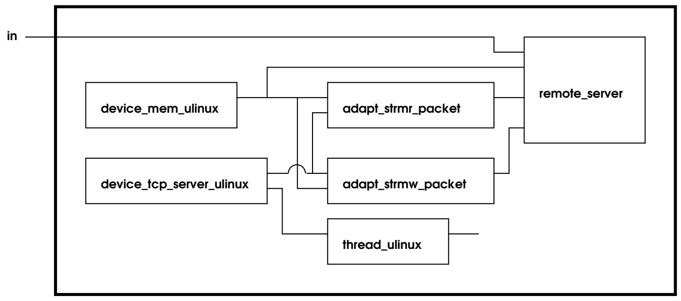

Figure 3.22: `remote_server_tcp`

3.5.4 `remote_server_socket`

If it is preferred to handle connection establishment externally, this brick can be used instead of `remote_server_tcp`. `$brick_init` expects the socket number instead of the port number in `@param` (syntax: "`rid=`<*id*> `socket=`<*socket number*>").

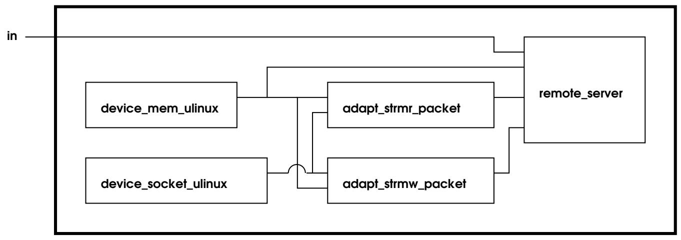

Figure 3.23: `remote_server_socket`

3.6 mirror

The `mirror` brick is a virtual brick and is split into a server part and a client part, as well.

The implemented mirror variant is very simple. It uses only one server and two clients. No replication is done in this variant.

TCP connections are used for communication between clients and the server. No communication is done between the clients. Locking is done entirely on the server side.

3.6.1 mirror_client_1_tcp

`$brick_init` expects an identification number, a host name and a port number in `@param` (syntax: "`rid=`<*id*> `host=`<*host name*> `port=`<*port number*>"). The *rid* parameter is used to identify client and server parts that belong together.

Upon initialization a connection to the server with the given host name and port number is established.

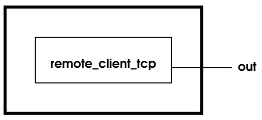

Figure 3.24: `mirror_client_1_tcp`

3.6.2 mirror_server_1_tcp

`$brick_init` expects an identification number and port numbers in `@param` (syntax: "`rid=`<*id*> `port1=`<*port number*> `port2=`<*port number*>").

It listens on the given port numbers for incoming connections from `mirror_client_1_tcp` bricks. Once the connections are established, the operations issued on the client's outputs effectively operate on the server's input nest. In order to synchronize the two clients, `$lock` and `$unlock` operation calls can be issued on the client side. These calls are also forwarded to the server's input nest.

The *rid* parameter is used to identify client and server parts that belong together.

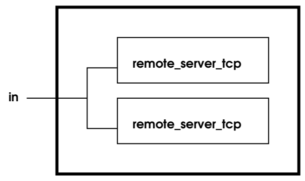

Figure 3.25: `mirror_server_1_tcp`

3.7 Network transparency

As described in 2.2 on page 9, the task of providing network transparent views is split into three parts:

At first, strategy nests from different nodes have to be merged. This is done by `strategy_merge`. `remote` bricks can be used to access strategy nests on other nodes.

`strategy_netconnect` merges the server and client parts of `remote_*` and `mirror_*` instances. The result is a connected view with virtual `remote` and `mirror` instances.

Finally, `strategy_nettransparent` replaces all `remote` and `mirror` instances by wires and thus provides the network transparent view.

Internally, all these bricks buffer the view in *mem* inputs using a doubly-linked cyclic list. A read transfer at address 0 is sufficing in order to reread the wiring and rebuild the view. Later implementations may use locks to detect changes in the wiring, and adapt the view automatically. Local instances of `adapt_strat` are used to convert `$trans` operations calls issued in write mode into strategy operations.

When building the view, it is assumed that the system wiring is a tree structure with `ATHOMUX_MAINDIR` as root node. That means that all bricks with no inputs have to be hooked to `ATHOMUX_MAINDIR` or they will be missing in the view. Currently, the ASCII representation of the wiring is parsed. Once an operation to get a list of all instantiated bricks is available, this should be used instead, since it is a lot faster.

Views can be displayed by using `strategy_display` instances.

3.7.1 `strategy_merge`

This brick merges two strategy nests. It can be cascaded in order to merge more nests. On `$connect` operations over view boundaries, `remote` bricks are inserted automatically to make the connection.

$gadr is always forwarded to the first strategy input, unless a destination is given in the form of “dsid=<*id*>”.

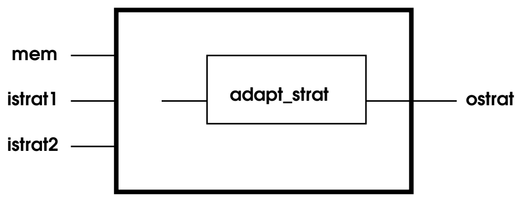

Figure 3.26: strategy_merge

3.7.2 strategy_netconnect

This brick merges server and client parts of remote_* and mirror_* instances into a virtual instance. It uses the *rid* parameter given to these instances to detect which parts belong together.

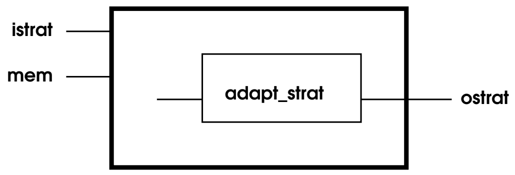

Figure 3.27: strategy_netconnect

3.7.3 strategy_nettransparent

strategy_nettransparent generates a network transparent view. It expects an already merged and connected view on its input and replaces virtual remote and mirror instances by new wires.

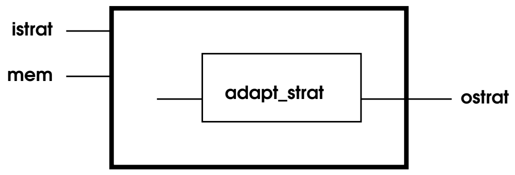

Figure 3.28: strategy_nettransparent

3.7.4 strategy_dot

As the previous strategy bricks, strategy_dot reads the wiring from its input. This is done every time $trans is called in read mode on its output. The graph description is then converted to *dot language*[1] [18] and given to the caller. If the length of the physical buffer is not sufficient, `$trans` fails.

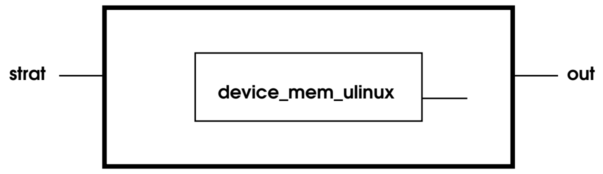

Figure 3.29: `strategy_dot`

3.7.5 `strategy_display`

This brick displays all the bricks and wirings that can be detected over its strategy input in a separate window on the screen . It uses temporary files and external utilities for the conversion and display procedure. `$brick_init` expects a name in `@param` (syntax: "`fname=`<*file name*>"). Two temporary files with then filenames "*fname*`.dot`" and "*fname*`.ps`" are created.

Upon initialization it creates a new thread which runs in a loop:

1. First it uses a local instance of `strategy_dot` to generate a graph description of the system in dot language.
2. Then it writes the graph description to a file named "*fname*`.dot`".
3. The *dot* utility is then called to convert the graph description into a postscript file named "*fname*`.ps`".
4. Finally the generated postscript file is displayed using the *display* utility.

As soon as the window displaying the graph is closed, the loop starts over.

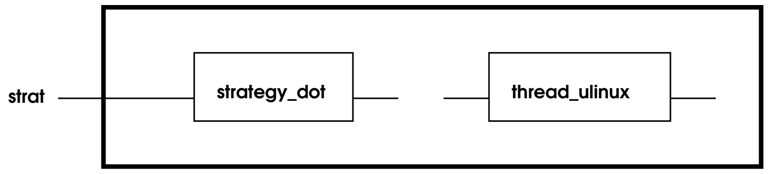

Figure 3.30: `strategy_display`

[1]The dot language is used in the *graphviz* package to describe graphs

3.8 Miscellaneous bricks

3.8.1 `adapt_complete2`

`adapt_complete2` ensures that `$trans` operations either transfer the whole buffer or fail.

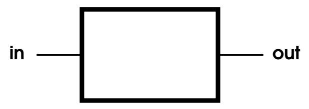

Figure 3.31: `adapt_complete2`

3.8.2 `merge_lock`

This brick directs `$lock` and `$unlock` operations to the *ilock* input. Any other operation is directed to *in*.

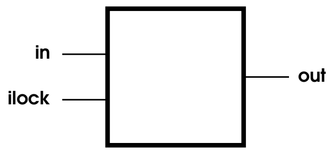

Figure 3.32: `merge_lock`

3.8.3 `map_delta`

`map_delta` adds `$move` operation functionality to the input *in*.

This implementation uses the cyclic doubly-linked lists of ATHOMUX to store information about moved regions. It is not the best data structure for this purpose. Often range queries similar to “what are the blocks between address1 and address2?” occur. Later implementations should use a b^*-tree or any other data structure supporting efficient range queries.

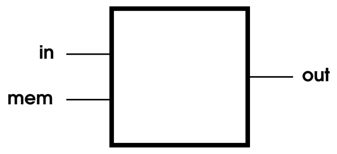

Figure 3.33: `map_delta`

3.8.4 `adapt_tmp`

`adapt_tmp` requires an input nest with heap semantics. By help of a local `map_delta` instance, it offers an output with nest semantics (see 2.6.1 on page 21). The operation `$create` may not be called when the given address range already is defined.

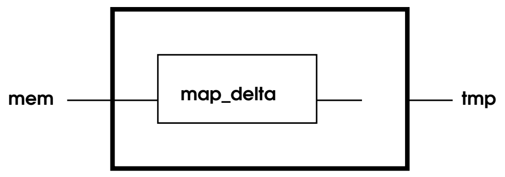

Figure 3.34: `adapt_tmp`

3.9 Foreign bricks

The following bricks have been implemented by Thomas Schöbel-Theuer and were often used in this work.

- `adapt_strat`
- `control_simple`
- `device_mem_ulinux`
- `lock_ulinux`
- `thread_ulinux`

3.10 Dependencies

Table 3.1 shows a list of all bricks implemented within the scope of this thesis. In addition to the name, it gives further information about the bricks:

SL: Stateless bricks are marked with “X” here.

PSL: Pseudo-stateless bricks are marked with “X” here.

SF: Stateful are marked with “X” here.

DL: The dependency level on the Linux runtime environment is given. A “0” means that the brick directly depends on Linux. A “1” means that the brick does not directly depend on Linux, but uses at least one other brick with dependency level “0”, and so forth. A “-” stands for no dependency at all.

brick name	**SL**	**PSL**	**SF**	**DL**
`adapt_complete2`	X			-
`adapt_multi`	X			1
`adapt_strm`	X			3
`adapt_strm_multi`	X			1
`adapt_strmr`			X	-
`adapt_strmr_packet`			X	-
`adapt_strmw`	X			-
`adapt_strmw_packet`	X			-
`adapt_tmp`	X			-
`demo_map`				0
`demo_merge`				0
`demo_mirror`				0
`demo_pipe2`				0
`demo_pipe`				0
`demo_ppc`				0
`demo_pps`				0
`demo_rpcc`				0
`demo_rpcs`				0
`demo_stratc`				0
`demo_stratc_init`				0
`demo_strats`				0
`device_socket_ulinux`				0
`device_tcp_client_ulinux`				0
`device_tcp_listen_ulinux`				0
`device_tcp_server_ulinux`				0
`map_delta`			X	-
`merge_lock`	X			-
`mirror_client_1_tcp`	X			2
`mirror_server_1_tcp`	X			2
`pipe`		X		-
`queue`		X		-
`pstrm_duplex`	X			-
`pstrm_simplex`	X			-
`remote_client`	X			1
`remote_server`	X			1
`remote_server_socket`	X			1
`remote_client_tcp`	X			1
`remote_server_tcp`	X			1

continued on next page...

...continued from previous page

brick name	**SL**	**PSL**	**SF**	**DL**
`strategy_display`			X	0
`strategy_dot`			X	1
`strategy_merge`			X	-
`strategy_msconnect`			X	1
`strategy_netconnect`			X	-
`strategy_nettransparent`			X	-
`strategy_stratconnect`	X			1
`strm_multiplex`	X			1
`strm_duplex`			X	2
`strm_simplex`	X			-

Table 3.1: List of all implemented bricks

Dependencies between the individual bricks are presented in the following figures. The arrows represent a “depends on” relation. Static dependencies (the using of local instances in bricks) are shown in figure 3.36 on the following page, whereas dynamic dependencies (the using of `$instbrick`) is shown in figure 3.35. Figure 3.37 on page 45 combines the previous graphs to a full dependency graph.

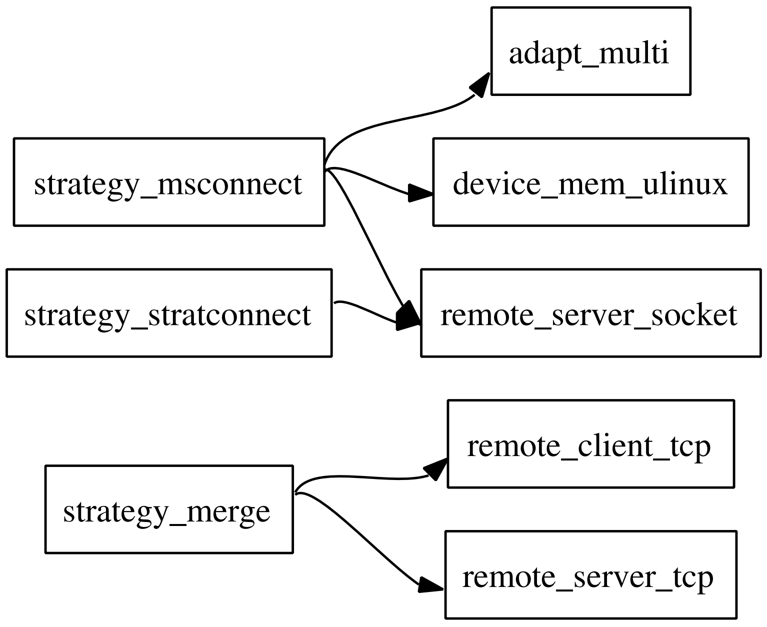

Figure 3.35: dynamic brick dependencies

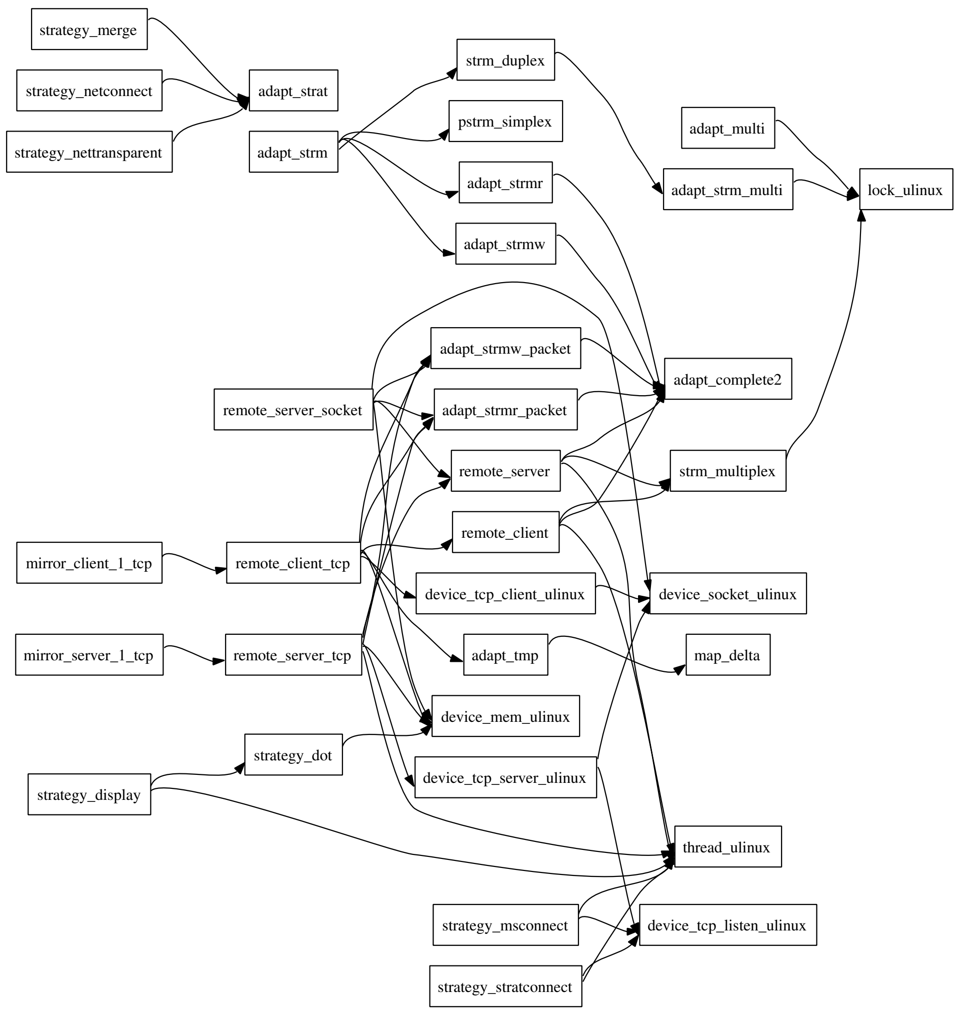

Figure 3.36: static brick dependencies

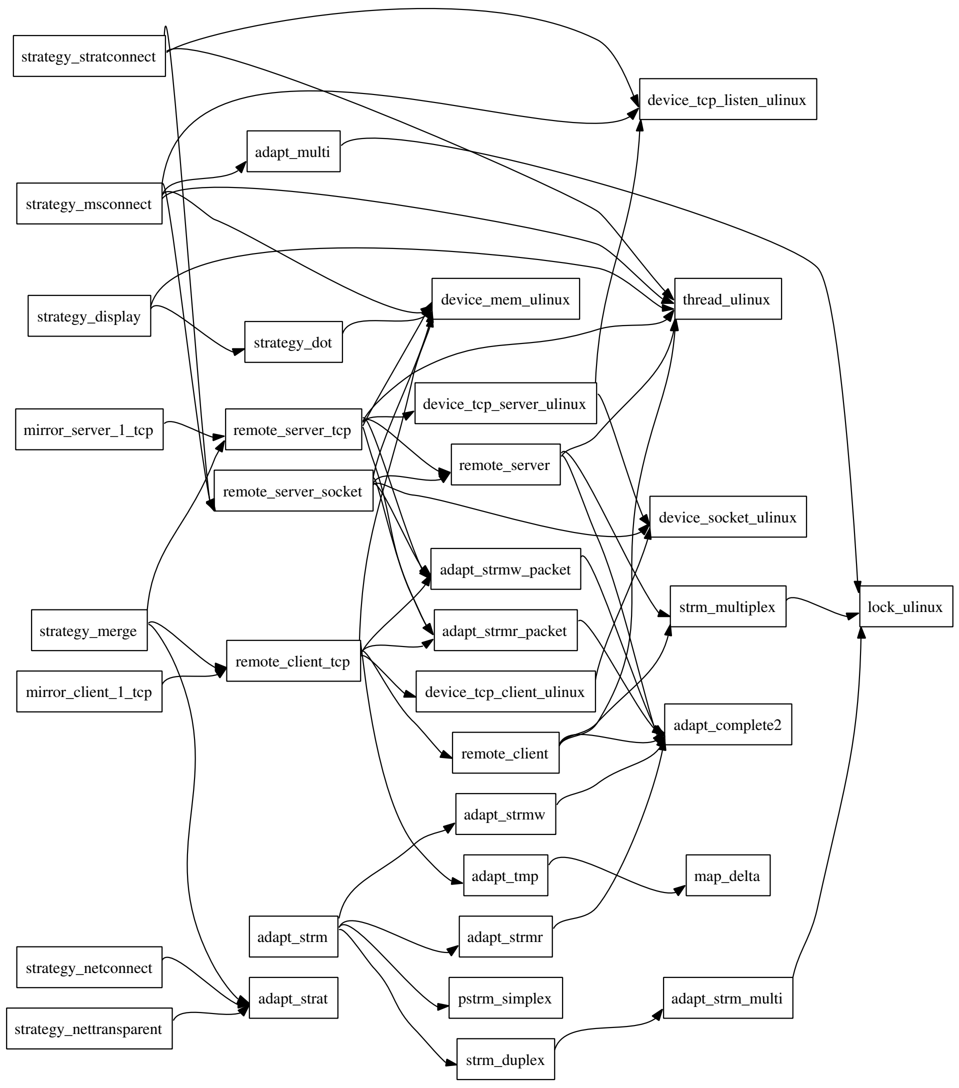

Figure 3.37: static and dynamic brick dependencies

3.11 Pseudo-statelessness

In order to achieve true pseudo-statelessness, bricks may not have local instances of bricks that offer any passive or active resources (e.g. `lock_ulinux` or `device_mem_ulinux`). One of the main tasks of this thesis is a network transparency demonstration. This involves many bricks and a complex wiring. For the purpose of a meaningful demonstration, some of the concerning bricks use local instances of resource offering bricks. Whereas the wiring is kept as simple as possible thereby, it also introduces stateful bricks. However, these stateful bricks can be made pseudo-stateless in the future by simply using accordingly wired inputs instead of the resource offering local instances.

3.12 Assumptions

Certain assumptions have been made during the implementation. While the implemented bricks are functioning correctly during demonstration, they could fail in the general case when these assumptions do not hold.

- A `control_simple` instance is expected at address 0x1000.
- `ATHOMUX_MAINDIR` is expected at address 0x00.
- `control_simple` is expected to be multiuser capable.
- There is an agreement about mandate allocation in a distributed ATHOMUX system.

3.13 Workarounds

Since the ATHOMUX environment is still missing many features, some workarounds were necessary:

- `$output_init` is not called on local outputs. This has to be done manually in `$brick_init`.
- Sometimes it is necessary to call `$brick_init` separately for each local brick instance in order to pass different `@param` for initialization. Currently, there exists only one macro (`INIT_ALL_INSTANCES()`) that deals with all local instances at once. Therefore, all bricks requiring initialization have an output called *init*. Calling `$output_init` on this output will do the initialization instead of `$brick_init`.
- Parameter names in ATHOMUX functions and parameter and variable names in ATHOMUX macros may conflict with outer names. Currently, neither the preprocessor nor the C compiler output warnings in that case. Since the generated code could lead to unintentional behavior, and the sources of these failures are hard to find, it is absolutely necessary to make these names unique. In this work, it is done by prefixing all concerning names with the enclosing function or macro name:

```
@.define foo(addr_t start)
{
  len_t bar;
  ...
}
```

This would be replaced by:

```
@.define foo(addr_t _fooprefix_start)
{
  len_t _fooprefix_bar;
  ...
}
```

3.14 Runtime environment

So far, ATHOMUX runs in userspace Linux as a guest operating system. Linux plays the role of a host operating system. This has the great advantage that we don't have to bother with hardware drivers and can concentrate on the development of higher level bricks. Nevertheless, on the long term it is the goal to run ATHOMUX independently.

Most bricks would already function in a native ATHOMUX environment. The number of bricks that still depend on the Linux environment is kept to a minimum. Table 3.1 on page 40 lists all bricks with dependency information.

The `graph_disp` brick has additional requirements on the runtime environment that might not be provided on a typical Linux system. It uses the *dot* program for translating a graph description given in dot language into a postscript file. This postscript file is then displayed using the *display* utility. Both tools have to be installed on the system. *dot* is part of the *graphviz* package [18] and *display* is part of the *imagemagick* package [19].

Chapter 4

Demonstration

In order to demonstrate and validate the implemented bricks, several test cases and demonstrations have been prepared in so called demo bricks. The names of all these demo bricks begin with "`demo_`". Demo bricks are special bricks that can be loaded and executed by using the `abl` tool. In this chapter, every demonstration procedure is explained and for each demo brick its dependency graph is shown to give a notion what bricks are part of the demonstration.

4.1 Instantiating and running demo bricks

The implementation of demo bricks has to follow two rules:

- It must have an input called *strat*. Before the `$brick_init` operation is called, this input is wired to the strategy output of the `control_simple` instance at address 0x1000.
- The only operation that is called on the demo brick is `$brick_init`. Hence, this operation has to contain the test code.

`abl` stands for "ATHOMUX Brick Loader". It implements the `main()` function so that the C-compiler can generate an executable. With the help of this tool, it is possible to instantiate and run the demo bricks.

The call syntax of this executable is:
`abl` *<brick name> [<parameter>]*

At first, `abl` creates a `control_simple` instance at address 0x1000. After that, it uses the strategy output of `control_simple` to instantiate the demo brick with the name given as first parameter to `abl`. The second parameter is passed to `$brick_init` in `@param`. As soon as `$brick_init` returns, the bricks are deinstantiated and `abl` terminates.

4.2 Helper bricks

The following bricks are used in some demo bricks in order to easily create complex test situations. They are not intended to be used elsewhere.

4.2.1 `strategy_msconnect`

This brick implements a strategy for offering shared memory to an unlimited number of clients. Any client can connect to the given port via a `remote_client_tcp` instance and thereby gets access to the nest instance offered by `device_mem_ulinux`. It is used in `demo_strats` to demonstrate the correct functioning of `remote_*` bricks.

`$brick_init` expects a port number and a remote identification number in `@param` (syntax: "`rid=`<*id*> `port=`<*port number*>"). At first, it instantiates a `device_mem_ulinux` and a `adapt_multi` brick. Then it listens on the given port for incoming connections. For each connection, it instantiates a `remote_server_socket` brick, and wires it to the `adapt_multi` instance. The remote identification number is passed to each newly instantiated `remote_server_socket`. Figure 4.2 on the following page shows how the dynamically instantiated bricks are wired.

Deinstantiation of the `device_mem_ulinux`, `adapt_multi`, and `remote_server_socket` bricks has to be handled externally.

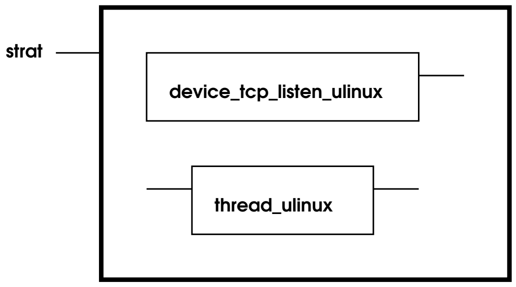

Figure 4.1: `strategy_msconnect`

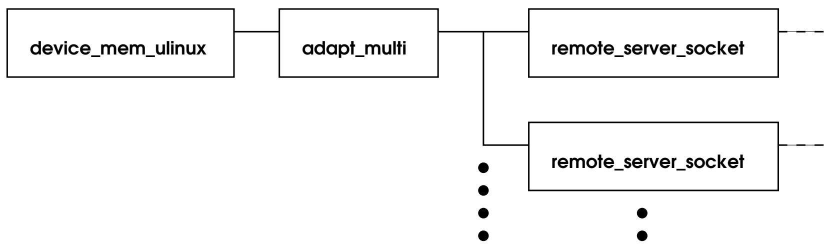

Figure 4.2: msconnect example

4.2.2 `strategy_stratconnect`

Like `strategy_msconnect`, it listens on the given port for incoming connections and dynamically instantiates `remote_server_socket` bricks. Again, `$brick_init` expects the port number and remote ID in `@param` (syntax: "`rid=`<*id*> `port=`<*port number*>"). However, this time the input of each instantiated `remote_server_socket` bricks is wired to the strategy output of the `control_simple` instance at address 0x1000. Figure 4.4 on the next page shows how the dynamically instantiated bricks are wired.

It is used in `demo_strats`, `demo_stratc_init`, and `demo_stratc` in order to give `demo_merge` easy access to the remote strategy nests.

Deinstantiation of the `adapt_multi`, and `remote_server_socket` bricks has to be handled externally.

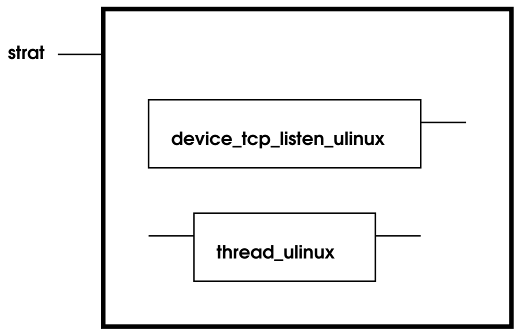

Figure 4.3: `strategy_stratconnect`

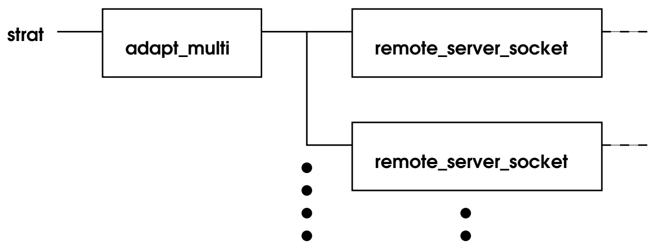

Figure 4.4: stratconnect example

4.3 Dynamic nests

`demo_map` tests the functioning of `map_delta`. For this purpose, a `device_mem_ulinux` instance is wired with a `map_delta` instance. Then, multiple `$move` operations are performed on the *tmp* output of `map_delta` and their result is checked.

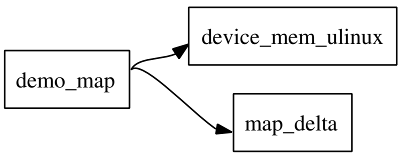

Figure 4.5: `demo_map` dependencies

4.4 Pipes

`demo_pipe` demonstrates FIFO functionality in the case of a logical pipe. It uses a `pipe` instance and writes some strings of different length to the pipe. After that the pipe is read and the strings are printed to the screen.

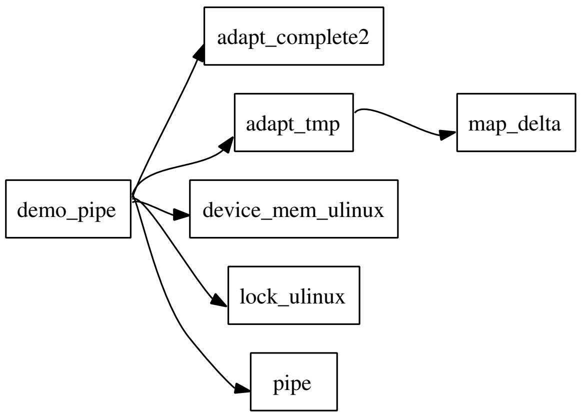

Figure 4.6: `demo_pipe` dependencies

`demo_pipe2` uses a logical pipe as well, but in this case the demonstration concentrates on the multiuser capabilities. Two threads are created, one for writing to the pipe, and one for reading. It is shown that the reading thread blocks when the pipe is empty, and that it wakes up after the writing thread has written to the pipe.

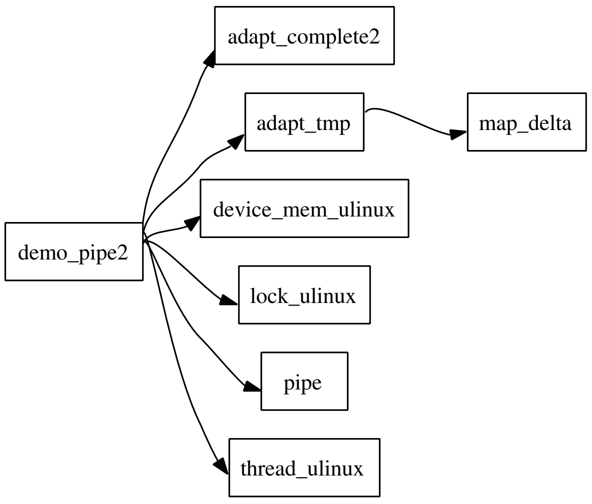

Figure 4.7: `demo_pipe2` dependencies

4.5 TCP connection establishment

This demonstration is divided into two bricks: `demo_pps` and `demo_ppc`. A TCP connection is established and data is sent in both directions over the offered physical stream.

`demo_pps` expects a port number as parameter (syntax: "`port=`<*port number*>"). It instantiates a `device_tcp_server_ulinux` brick which listens on the given port for an incoming connection.

`demo_ppc` expects a host name and a port number as parameter (syntax: "`host=`<*host name*> `port=`<*port number*>"). It uses a `device_tcp_client_ulinux` instance in order to connect to the given destination.

After the connection establishment `demo_ppc` it waits for user input on *stdin*. The string is then sent to `demo_pps` where it is slightly modified and finally sent back and printed to the screen.

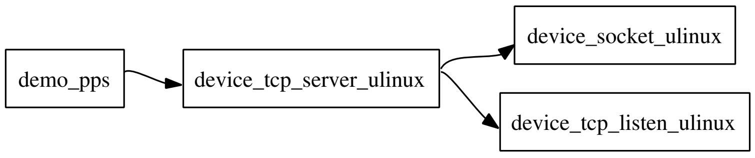

Figure 4.8: `demo_pps` dependencies

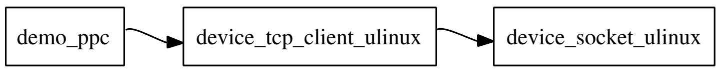

Figure 4.9: `demo_ppc` dependencies

4.6 Remote test

This demonstration is also divided into two parts: `demo_rpcs` plays the server role and `demo_rpcc` the client role. It is a simple test for the `remote_*` bricks.

`demo_rpcs` expects a port number as parameter (syntax: "`port=`<*port number*>"), whereas `demo_rpcc` expects a host name and a port number (syntax: "`host=`<*host name*> `port=`<*port number*>"). The connection is established as in 4.5 on the preceding page.

The remote server is connected to a `device_mem_ulinux` instance. Any operation call issued by the client should arrive at this brick.

In order to validate this scenario, the client performs three tasks on its input nest:

1. It allocates some memory in the input nest. This is actually done on the server side due to the remote brick.
2. It writes a string to the address it got from step 1.

3. It reads again from this address and compares the result with the original string.

Since no buffering is performed and the input nest is connected to the `device_mem_ulinux` instance over a `remote` instance, the string has to be transferred over the network in step 2 and in step 3. From the client's point of view the input nest is just a nest instance it writes to and reads from. Due to anonymity of relation, it does not matter whether it is a local `device_mem_ulinux` instance or a remote instance.

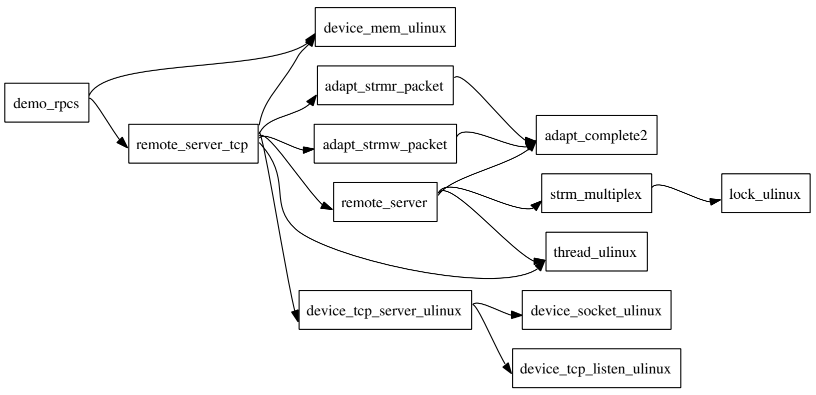

Figure 4.10: `demo_rpcs` dependencies

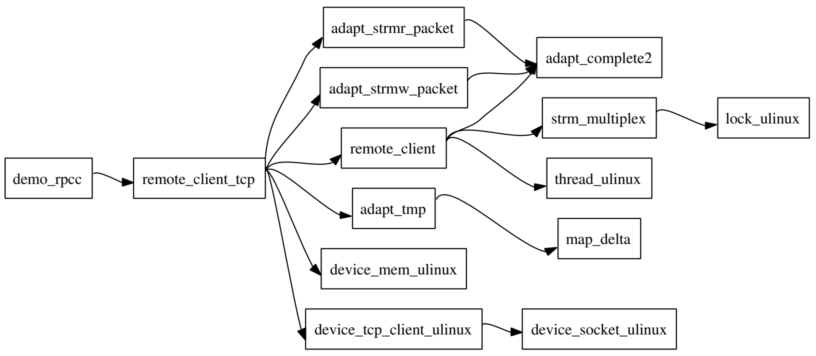

Figure 4.11: `demo_rpcc` dependencies

4.7 Network transparency

The network transparency demonstration is split into four demo bricks: `demo_strats`, `demo_stratc_init`, `demo_stratc` and `demo_merge`.

The first three bricks build a system with distributed shared memory functionality. Each brick is instantiating a `strategy_stratconnect` and thus offering a connection to the local `control_simple` instance to any other node (see 4.2.2 on page 50).

Every `remote_*` instance in the system is given an identification in the form of "`rid=`<*id*>" that associates the server part with the client part. `demo_merge` uses this identification to generate global and network transparent views.

`demo_strats` plays the role of a server and only instantiates a `strategy_stratconnect`. Thereby, the strategy nest offered by the `control_simple` instance is made available to others. Other bricks are instantiated later using a connection to this strategy nest.

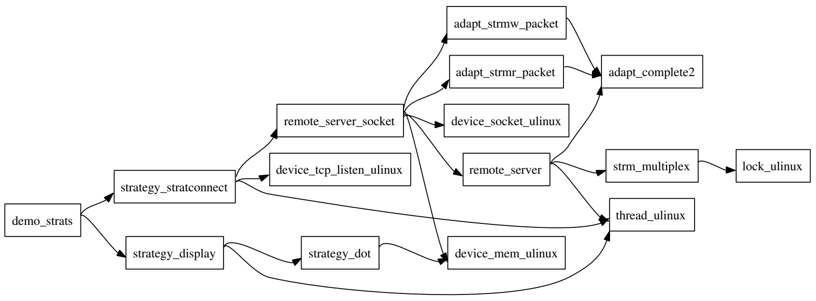

Figure 4.12: `demo_strats` dependencies

`demo_stratc_init` establishes a connection to the `control_simple` instance on the server. It uses this strategy nest to instantiate a `strategy_msconnect` on the server. `strategy_msconnect` instantiates a `device_mem_ulinux` brick and makes it available to others (see 4.2.1 on page 49).

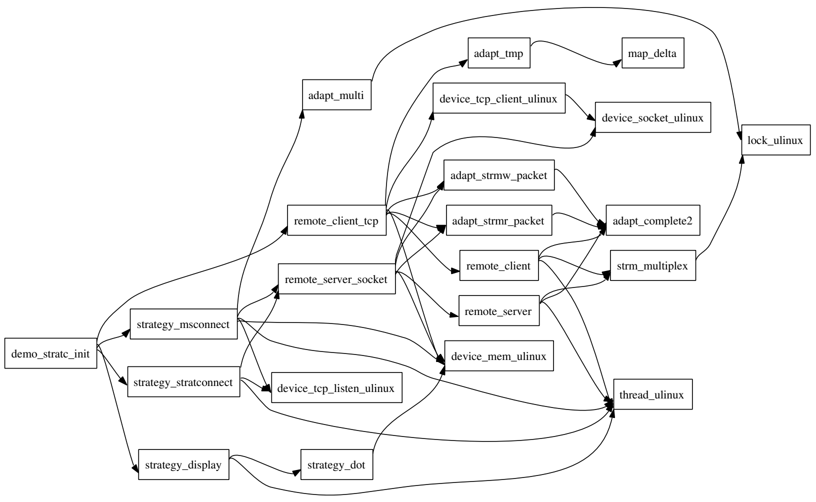

Figure 4.13: `demo_stratc_init` dependencies

`demo_stratc` plays the role of a client. It connects to the `device_mem_ulinux` instance on the server. The user can issue commands on the console in order to allocate, write and read memory regions.

- `help`: Prints a short list of available commands.
- `alloc` <*len*>: Allocates a memory block of given length and returns its address.
- `write` <*addr*> <*string*>: Writes a string to the given address. The number of bytes actually written is returned.
- `read` <*addr*> <*len*>: Reads from given address and prints the content.

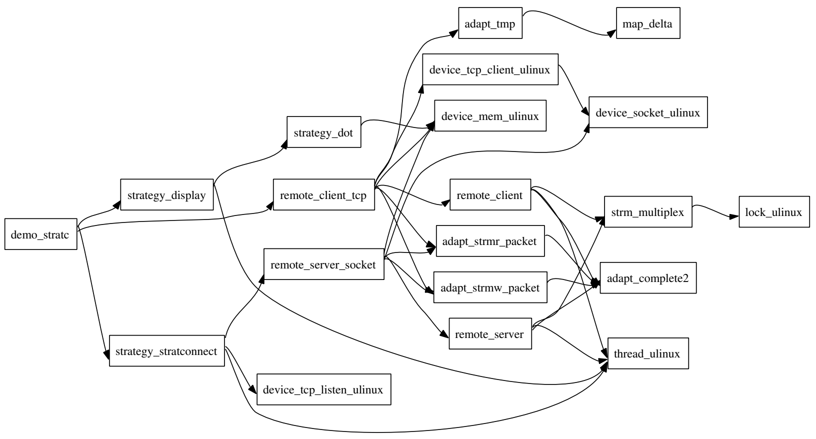

Figure 4.14: `demo_stratc` dependencies

`demo_merge` finally creates a global view of the system. It connects to the `control_simple` instance on all other nodes and uses `strategy_merge` to merge the views. The server and client parts of the `remote_*` bricks are merged by `strategy_netconnect`. At the end, `strategy_nettransparent` removes all virtual `remote` instances from the view and thus generates the network transparent view.

In order to demonstrate that the network transparency is functioning correctly even when using modifying operations, the following tasks are performed:

1. Instantiating a `device_mem_ulinux` brick on a remote node.
2. Connecting this instance to `ATHOMUX_MAINDIR`.
3. Disconnecting and deinstantiating it.
4. Disconnecting and deinstantiating the virtual remote brick responsible for the connection between the server and the client node using the connected view.
5. Reconnecting them by issuing a single `$connect` operation call in the network transparent view. A new virtual remote brick is inserted automatically to make the connection possible. The new connection can be tested by issuing commands (alloc, read, write) on the client side.

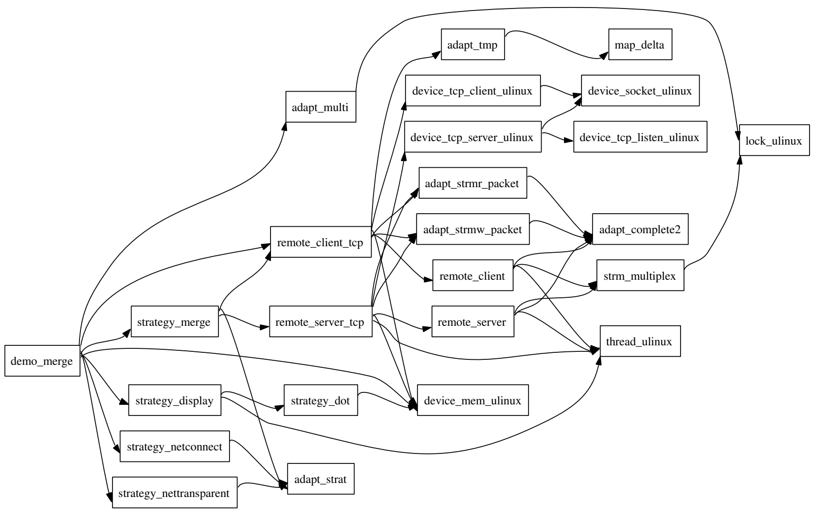

Figure 4.15: `demo_merge` dependencies

To run this demonstration, the individual parts have to be started in the following order:

1. `demo_strats`

2. `demo_stratc_init`
3. `demo_stratc`
4. `demo_merge`

Several `strategy_display` instances in all parts are used in order to display the views at the different stages. Figures 4.16 - 4.19 show the local view for each system.

Figure 4.20 on page 61 shows the merged and connected view which is provided by the `strategy_display` brick at address 0x023000 in this figure.

Figure 4.21 on page 62 shows the network transparent view which is provided by the `strategy_display` brick at address 0x025000 in this figure.

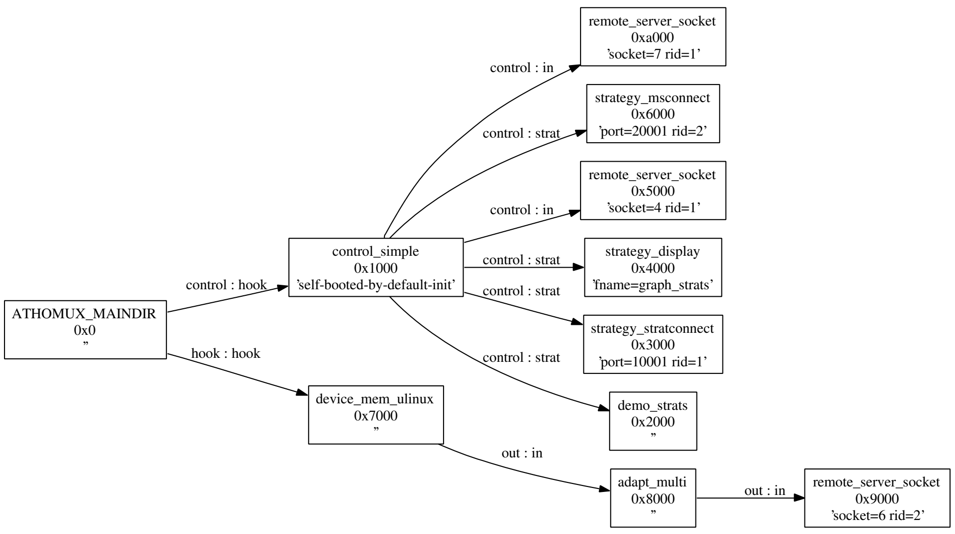
remote_server_socket 0xa000 'socket=7 rid=1'
control : in
strategy_msconnect 0x6000 'port=20001 rid=2'
control : strat
remote_server_socket 0x5000 'socket=4 rid=1'
control : in
control_simple 0x1000 'self-booted-by-default-init'
control : strat
strategy_display 0x4000 'fname=graph_strats'
control : strat
strategy_stratconnect 0x3000 'port=10001 rid=1'
control : hook
ATHOMUX_MAINDIR 0x0 ''
hook : hook
control : strat
demo_strats 0x2000 ''
device_mem_ulinux 0x7000 ''
out : in
adapt_multi 0x8000 ''
out : in
remote_server_socket 0x9000 'socket=6 rid=2'

Figure 4.16: local view on `demo_strats`

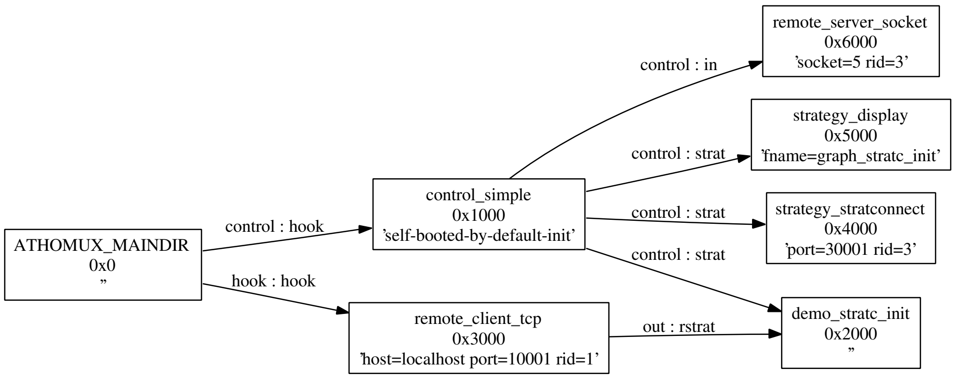
remote_server_socket 0x6000 'socket=5 rid=3'
control : in
strategy_display 0x5000 'fname=graph_stratc_init'
control : strat
control_simple 0x1000 'self-booted-by-default-init'
control : hook
control : strat
strategy_stratconnect 0x4000 'port=30001 rid=3'
ATHOMUX_MAINDIR 0x0 ''
control : strat
hook : hook
remote_client_tcp 0x3000 'host=localhost port=10001 rid=1'
out : rstrat
demo_stratc_init 0x2000 ''

Figure 4.17: local view on `demo_stratc_init`

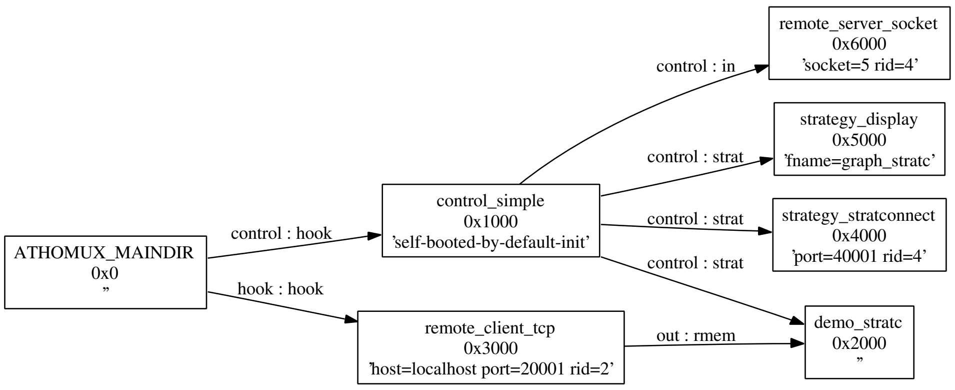

Figure 4.18: local view on `demo_stratc`

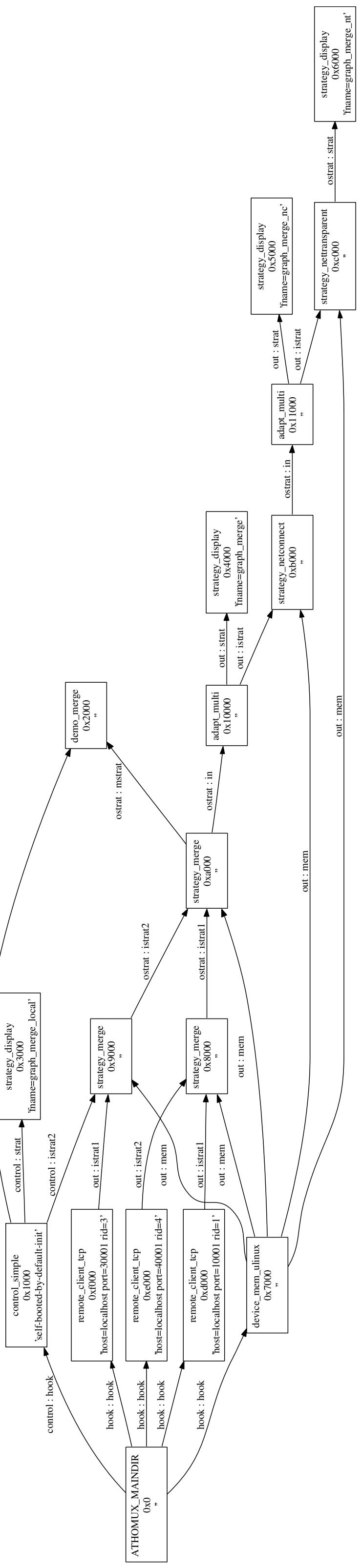
ATHOMUX_MAINDIR 0x0 ''
control_simple 0x1000 'self-booted-by-default-init'
remote_client_tcp 0xf000 'host=localhost port=30001 rid=3'
remote_client_tcp 0xe000 'host=localhost port=40001 rid=4'
remote_client_tcp 0xd000 'host=localhost port=10001 rid=1'
device_mem_ulinux 0x7000 ''
strategy_display 0x3000 'fname=graph_merge_local'
strategy_merge 0x9000 ''
strategy_merge 0x8000 ''
strategy_merge 0xa000 ''
demo_merge 0x2000 ''
adapt_multi 0x10000 ''
strategy_display 0x4000 'fname=graph_merge'
strategy_netconnect 0xb000 ''
adapt_multi 0x11000 ''
strategy_display 0x5000 'fname=graph_merge_nc'
strategy_nettransparent 0xc000 ''
strategy_display 0x6000 'fname=graph_merge_nt'
control : hook
hook : hook
control : strat
control : istrat2
out : istrat1
out : istrat2
out : mem
ostrat : istrat2
ostrat : istrat1
ostrat : mstrat
ostrat : in
out : strat
out : istrat
ostrat : strat

Figure 4.19: local view on `demo_merge`

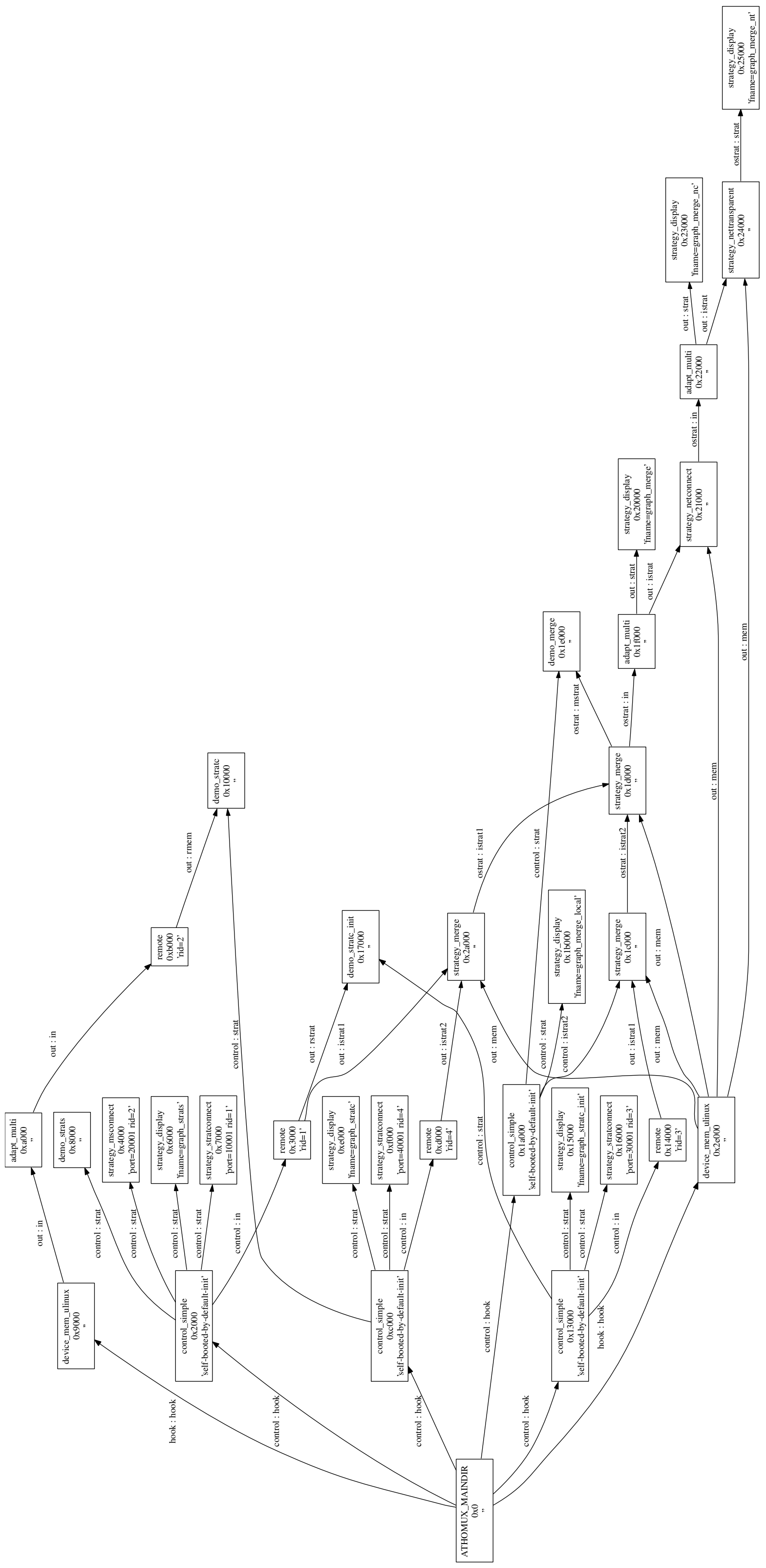
ATHOMUX_MAINDIR 0x0 ''
device_mem_ulinux 0x9000 ''
control_simple 0x2000 'self-booted-by-default-init'
control_simple 0xc000 'self-booted-by-default-init'
control_simple 0x13000 'self-booted-by-default-init'
control_simple 0x1a000 'self-booted-by-default-init'
device_mem_ulinux 0x2e000 ''
adapt_multi 0xa000 ''
demo_strats 0x8000 ''
strategy_rmsconnect 0x4000 'port=20001 rid=2'
strategy_display 0x6000 'fname=graph_strats'
strategy_stratconnect 0x7000 'port=10001 rid=1'
remote 0x3000 'rid=1'
strategy_display 0xe000 'fname=graph_strate'
strategy_stratconnect 0xf000 'port=40001 rid=4'
remote 0xd000 'rid=4'
strategy_display 0x15000 'fname=graph_strate_init'
strategy_stratconnect 0x16000 'port=30001 rid=3'
remote 0x14000 'rid=3'
remote 0xb000 'rid=2'
demo_strate 0x10000 ''
demo_strate_init 0x17000 ''
strategy_merge 0x2a000 ''
strategy_display 0x18000 'fname=graph_merge_local'
strategy_merge 0x1c000 ''
strategy_merge 0x1d000 ''
demo_merge 0x1e000 ''
adapt_multi 0x1f000 ''
strategy_display 0x20000 'fname=graph_merge'
strategy_netconnect 0x21000 ''
adapt_multi 0x22000 ''
strategy_display 0x23000 'fname=graph_merge_nc'
strategy_nettransparent 0x24000 ''
strategy_display 0x25000 'fname=graph_merge_nt'
hook : hook
control : hook
control : strat
control : in
out : in
out : rmem
out : rstrat
out : istrat1
out : istrat2
out : mem
ostrat : istrat1
ostrat : istrat2
ostrat : mstrat
ostrat : in
out : strat
out : istrat
ostrat : strat

Figure 4.20: merged and connected view

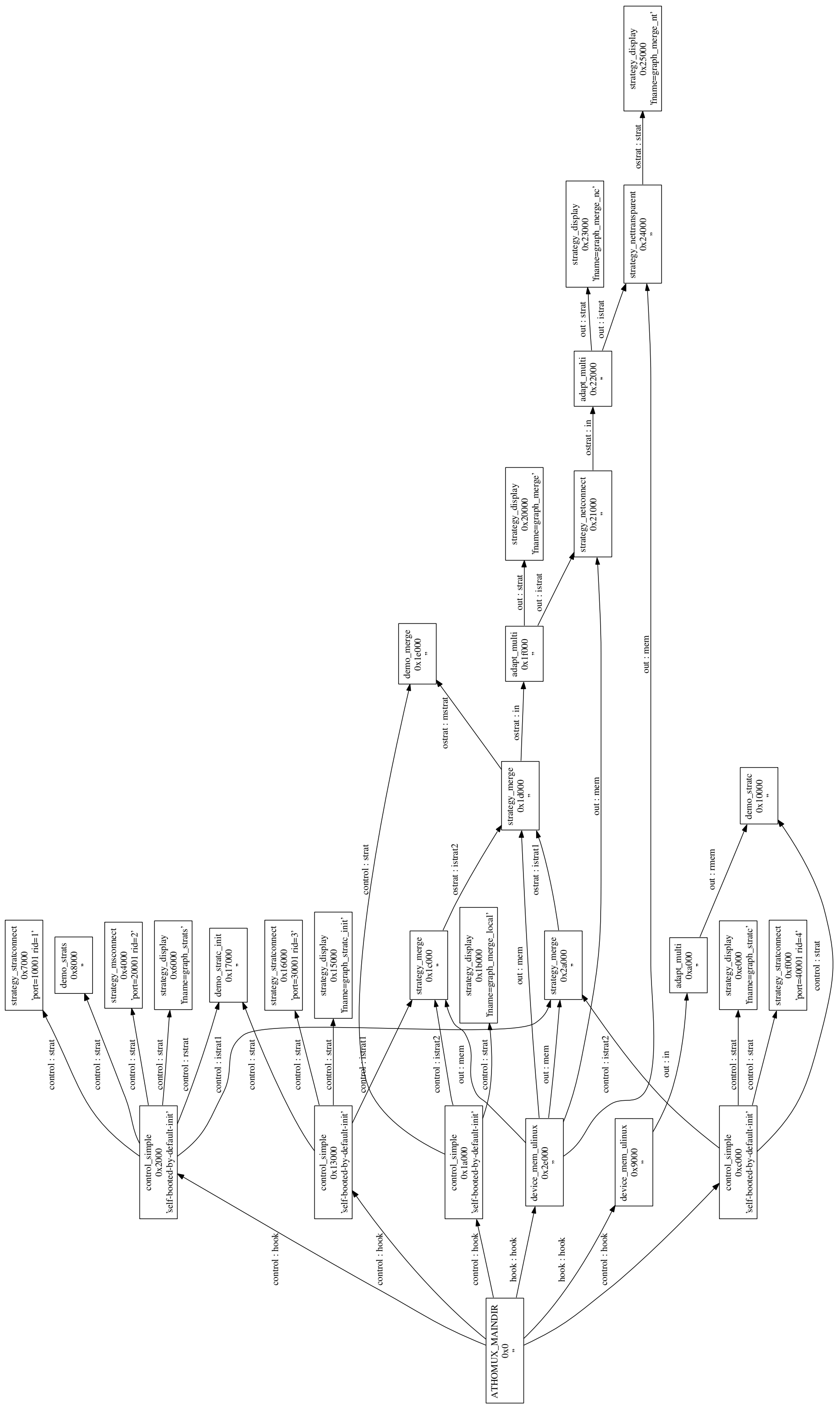

Figure 4.21: network transparent view

4.8 Mirror test

`demo_mirror` is a demonstration for a simple mirror brick that uses one server and two clients. A string is written to the server through one client, and is read again through the other client. Furthermore, a connected view with virtual `mirror` instances, and a network transparent view are displayed on the screen.

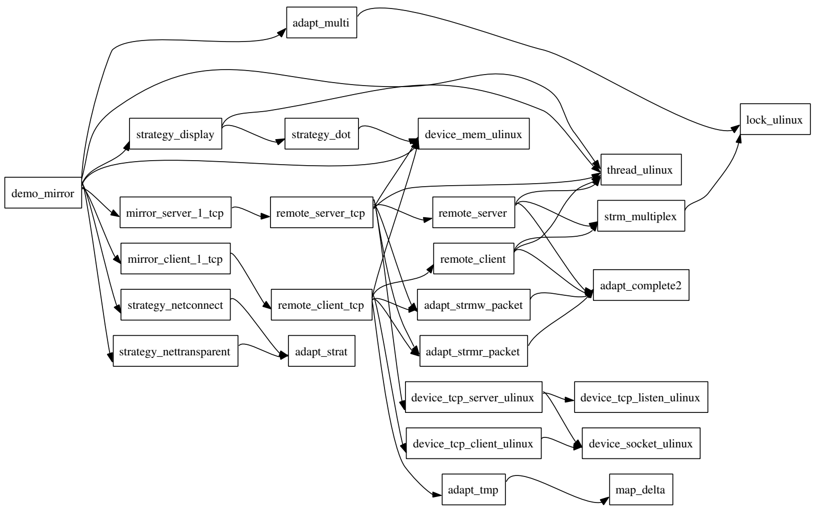

Figure 4.22: `demo_mirror` dependencies

Chapter 5

Conclusion

5.1 Summary

The demonstration programs demo_strats, demo_stratc_init, demo_stratc and demo_merge as well as demo_rpcs and demo_rpcc have shown that the implemented remote bricks are functioning as expected and that they can be inserted anywhere in the system.

The functioning of a simple mirror variant was demonstrated by demo_mirror.

Multiple views were merged by strategy_merge and network transparency could be achieved by using strategy_netconnect and strategy_nettransparent. This was demonstrated by demo_merge and demo_mirror. The network transparent view is fully functional which was also demonstrated by demo_merge.

Many features in ATHOMUX are still missing. Workarounds had to be implemented to bypass some necessary features that already exist in concepts, but are still missing in the current ATHOMUX environment. In near future, these workarounds might not be needed any more.

5.2 Future work

In this work, the implementation focussed rather on functionality than on performance. Many optimizations are still possible:

- More efficient data structures can be used instead of doubly-linked cyclic lists. E.g. b^*-trees would be more appropriate for range queries, which are often issued.
- Asynchronous IO in remote_* bricks could alleviate the effect of high network latency.

- Using group communication should improve performance of network related bricks, especially `mirror_*`.
- Instead of parsing the ASCII output of strategy nests, operation calls should be used once they are available.

The bricks implemented in this work represent only the basis of transparently distributed ATHOMUX. To cover the whole domain of distributed systems, would clearly go beyond the scope of this thesis. However, there are two issues closely related to this work that need further efforts.

- Only a simple `mirror` variant could be implemented. Further research is needed here, especially in the direction of replication and fault tolerance.
- Full write support in network transparent views should bring a truly transparently distributed ATHOMUX. The concepts for achieving this have been described in chapter 2 whereas the implementation still needs some work.

During the implementation and testing procedure, the limitation or rather the inappropriateness of available debugging tools emerged. Bricks are written in ATHOMUX code which is transformed to C code by a preprocessor tool. During the debugging of bricks, only the C code is available. One has to do the mapping to corresponding parts in the ATHOMUX code by hand, which is very tedious. Another issue is that breakpoints can only be set at a certain point in a brick. Several instances of the same brick are not distinguishable. For that reason and since the wirings are getting more and more complex, there is a need for a debugger tool especially designed for ATHOMUX. It should provide the user with views of the system as `strategy_display` does, but with the possibility of interaction. This should significantly reduce development time. A visual brick editor would be of great help as well.

Bibliography

[1] SCHÖBEL-THEUER, THOMAS: *On Variants of Genericity*.
In *Proceedings of the Fifteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2003)*, pages 359–365. Knowlegde Systems Institute, 2003.

[2] SCHÖBEL-THEUER, THOMAS: *Skizze einer auf nur zwei Abstraktionen beruhenden Betriebsystem-Architektur: Nester und Bausteine*.
Arbeitspapier und Vortrag auf dem Herbsttreffen der GI, Fachgruppe Betriebssysteme, Berlin, 7. – 8.11.2002.

[3] SCHÖBEL-THEUER, THOMAS: *Instance Orientation: A Programming Methodology*.
In *SEA 2004*, pages 173–179. IASTED conference proceedings, Cambridge, MA, November 2004.

[4] SCHÖBEL-THEUER, THOMAS: *Eine neue Architektur für Betriebsysteme*.
Unveröffentlichtes Manuskript einer Monographie, erhältlich auf Anfrage bei ts@athomux.net, 2003.

[5] KORTH, JENS-CHRISTIAN: *Strategies for Automated Porting of Linux Device Drivers to Athomux*, 2004.

[6] NIEBLING, FLORIAN: *Development of a Process Model for Athomux*, 2004.

[7] SHAW, MARY and DAVID GARLAN: *Software Architecture - Perspectives on an Emerging Discipline*.
Prentice Hall, 1996.

[8] SCHÖBEL-THEUER, THOMAS: *A LEGO-like Lightweight Software Component Architecture for Organic Computing*.
In PETER DADAM, MANFRED REICHERT (editor): *Informatik 2004 - Informatik verbindet, GI workshop on organic computing*, volume 2 of *Lecture Notes in Informatics, Proceedings P-51*, pages 621–625. GI Edition, 2004.

[9] SCHÖBEL-THEUER, THOMAS: *A Way for Seamless Integration of Databases and Operating Systems*.
In *Proceedings of the International Conference on Computer Science and its Applications (ICCSA-2003)*, pages 82–89. National University, 2003.

[10] SCHÖBEL-THEUER, THOMAS: *Programming in the ATHOMUX Environment*.
http://www.athomux.net/, 2004.

[11] SCHÖBEL-THEUER, THOMAS: *Symmetric Optional Locking*.

Unpublished paper, available on request: ts@athomux.net, 2004.

[12] SCHÖBEL-THEUER, THOMAS: *Generalized Optional Locking in Distributed Systems*.
In *PDCS 2004*, pages 216–225. IASTED conference proceedings, Cambridge, MA, November 2004.

[13] SCHÖBEL-THEUER, THOMAS: *Speculative Prefetching of Optional Locks in Distributed Systems*.
In *PDCN 2004, Innsbruck*. IASTED conference proceedings, 2004.

[14] GEORGE COULOURIS, JEAN DOLLIMORE, TIM KINDBERG: *Distributed Systems*.
Addison-Wesley, 2001.

[15] SCHÖBEL-THEUER, THOMAS: *The ATHOMUX Preprocessor User's Guide*.
http://www.athomux.net/, 2004.

[16] SCHÖBEL-THEUER, THOMAS: *The Build System of Athomux*.
http://www.athomux.net/, 2004.

[17] *The ATHOMUX Project*.
http://developer.berlios.de/projects/athomux/.

[18] *Graphviz - Graph Visualization Software*.
http://www.graphviz.org/.

[19] *ImageMagick Toolkit*.
http://www.imagemagick.org/.

Declaration

All the work contained within this thesis,
except where otherwise acknowledged,
was solely the effort of the author.
At no stage was any collaboration
entered into with any other party.

(Hardy Kahl)