Resource Management and Concurrency in Athomux

Roland Niese

23rd July 2006

Chapter 1

Introduction

1.1 What is Athomux?

Athomux is an architecture for operating systems that aims at maximal flexibility, scalibility and extensibility through strict modularisation and uniformness of the interfaces (called *nests*) between the modules (called *bricks*). An often used metaphor for this structure is the LEGO® toy brick system, that too allows extremely flexible compositions out of its bricks, due to its uniform interface, the knobs. Just like newly developed LEGO bricks in arbitrary forms and colours can be seemlessy included into any existing model without having to modify any bricks in it, Athomux allows inclusion of newly developed technologies at almost any place, in form of new *bricks*.

1.2 Motivation and Objects

New concepts like instance orientation and optional locking were implemented during this work to test these concepts in practice. This thesis deals with concurrency in Athomux and tests increase in performance when using optional locking. Also, the basic structure of the whole architecture is still in development and was tested during this work as well. Many times, fundamental elements like the uniform interface (*nests*) had to be modified.

1.3 Environment

There are currently two runtime environments available for Athomux, both hosted by Linux: user and kernel space Athomux. To ease debugging, the lock mechanism of this thesis was developed in Linux user space ("athomux-ulinux"), but it was taken care of easy extensibility to other runtime environments through encapsulation of the runtime-specific parts. The Athomux-loader program already used for transparent distributed work [4] was used as runtime system and

was later adapted to new developments in the basic system.

Chapter 2

Prerequisites

2.1 Fundamentals

Athomux is instance oriented. What does that mean? An instance oriented [2] system is composed of isolated modules (*bricks*), each having a number of well-defined interfaces (*nests*) for other modules. The *nests* of *bricks* can be connected (*wired*) to allow them to communicate with other bricks. Multiple *instances* of one brick type can be created and wired individually. It is the composition of bricks through wires that defines the behaviour of the "whole thing".

Note that brick instances have no influence on the wiring of their nests. In Athomux, this means that bricks' view of the world is limited to their respective nests, nothing more. In contrast to, e.g., object orientation, there are neither public global ("static") variables that would allow direct (not through wires) access to other bricks, nor is there a way for a brick to manipulate its own wiring to other bricks. This is the domain of one local "control" brick instance that is the only one capable of brick creation, destruction and (dis-)connection in its scope. It can give control over sub-domains to other bricks by defining a nest and wiring them to it. Such "privileged" bricks are called *strategy bricks* and can be used, e.g., for automatic or user-requested instantiation of certain bricks or for filtering of the strategy nest, thus providing a new level of strategic view.

2.2 Resources

Nests provide resources. The term "resource" in Athomux means nothing more (and nothing more specific) than the term "information". A nest defines a uniform interface to access information. "Access" means reading and/ or modification of information. For this purpose, a nest defines a number of *operations*. This set of operations is fixed and uniform across all nests in the whole Athomux system. There is no extension mechanism whatsoever. To a programmer,

this may seem a restriction at first, but is rather a gain of freedom, as it allows arbitrary composition of resource *providers* and resource *users*.

Speaking of which, bricks can be resource providers as well as resource users, or both at the same time. Their role is defined by the definition of *connectors*. Each brick can, at compile time, define *input* and *output* connectors. These terms do *not* relate to the term pairs "read/ write" or "produce/ consume" as they do in UNIX systems when chaining programs through *pipes*. The terms *input* and *output* rather specify the role of the connector as *provider* or *user* of information. The *flow* of information can be bidirectional. A brick provides a nest of information at an output connector. An arbitrary number of bricks can have their input connectors wired to this output connector to make use of the information provided there. The relation of outputs to inputs is 1:n. While an output connector can provide its information nest to one, many or no *inputs*, inputs must be connected to exactly one output to be operable.

2.3 Bricks

Bricks implement the functionality in Athomux. They process information taken from the inputs and provide information on their outputs. Bricks without inputs are original resource providers ("device bricks") while bricks without outputs are pure "user" bricks that just act upon information provided to them. A brick *without* connectors does not really make sense, as it is totally separated from the rest of the system. Bricks that have both inputs and outputs are in most cases some kind of transformer whose information provided on the outputs directly depends on the data at the inputs.

2.3.1 Lifecycle

A brick's life cycle always starts with its *instantiation*. A control brick loads the code and other static properties of the brick class, assigns enough memory to it and creates the runtime data structure. If the brick has vital inputs without which it cannot function properly, they must be connected to outputs before the brick can be initialized (activated). Once this condition is fulfilled, the brick can be activated. From then, all active outputs can be used by wiring bricks via their inputs to the output. This shows a certain dependency relation:

- The activation of an input depends on the connection to an active output. Note that an input can be *wired* alias *connected* to an inactive output. An inactive input will never emit operations to the output.
- The activation of an output depends on the activation of the brick it is provided by. Furthermore, it may depend on the activation of certain inputs of the same brick that are not vital for *brick* initialization. Note that it never depends on the connection to any inputs.
- Brick activation may depend on the initialization of certain inputs.

These preconditions imply certain dependencies at brick *deinitialization*, or better, *shutdown*.

- The deactivation of an input may depend on a brick shutdown, but can be performed *during* a brick shutdown as soon as the resources used via the input are not needed anymore. In any case, all resources allocated via the input must be returned during shutdown.
- Shutdown of the brick depends on the shutdown of all outputs of the brick.
- Shutdown of an output depends on the deactivation (but not disconnection) of *all* connected inputs.

At the end of its (local) life, a brick is destroyed by the supervising control brick. The only precondition is disconnection of all its inputs and outputs and, of course, deactivation of the brick.

2.3.2 State

Bricks should be *stateless*. This means they should not contain internal data that cannot be viewed from the outside. So public attributes like the connections or explicit attributes are not considered 'state' because they don't change during the lifetime of a brick. But most bricks need to maintain certain configuration data, typically dynamic data structures describing the contents of an output nests, or io buffers. Nonetheless, Athomux allows local variables. But bricks should not use them to save state but only for volatile data or caches.

Whatever is stored in local variables should be flushed into input nests on brick shutdown. The reason is reconstruction. A brick that has been shutdown may be locally destroyed but reinitialized elsewhere. It should be possible to reconstruct the brick from the pure information contained in the input nests. Athomux aims to be (also) an architecture for distributed network operating systems. Components of such an operating system should be able to be *migrated* to other cluster nodes anytime.

2.4 The Nest Interface

A nest is an interface to an abstract set of information. It defines a logical address space for access to the information provided by it. At the time of this work, this is a 64bit address space, independent of the runtime environment. It should not be confused with a machine's physical address space. The data behind a nest's logical address space can only be accessed via a well-defined and fixed set of operations.

2.4.1 Transfers

A nest defines an operation to read information from or to write it to a nest, we call it `$trans`. These *transfers* are always between the nest and physical data

buffers, the only place where algorithms running on a physical machine can directly access data (alias information). The smallest unit of information equals the one of the physical machine the brick is running on, on today's machines this is usually one *byte*. So the *transfer* operation transfers blocks of bytes between the logical address space of a nest and the physical one of the local machine.

2.4.2 Data Mapping

It may be inefficient to always *copy* information from the logical address space to the physical one. Therefore, a nest defines an operation for users to obtain a (temporary) physical buffer that directly refers to a certain part of the information space of the nest. We call this operation $get. The buffer should be released after the user brick has finished work on it. The operation to do this is called $put. It depends on the nature of the nest whether the buffer obtained is a direct physical reference "into" the nest, is a mere copy of the nest's input or refers to some kind of cache the nest maintains for performance reasons. The user *does not know*.

2.4.3 Address Mapping

A nest does not have to provide one byte of data for every logical address. There is always a map between address and data space. This map may sometimes be static, but most bricks use a dynamic map. The nest interface defines operations to modify the the address map. A dynamic nest will (in most cases) start with an empty address space where any data access operation will fail upon. An operation called `$create` maps some of the background data space to a specified subset of the address space, logically this is equivalent to the "creation" of data, hence the name. To "destroy" data (undefine that mapping), Athomux nests define an operation called `$delete`.

An Athomux special is an operation to "move" a mapping to a different address range. Logically, this looks like a movement of the data to a different location, but without the efforts of copying it. The operation performing this is called $move. This mechanism allows users to maintain arbitrary dynamic data structures as content of the nest without worrying about such things like linked lists or trees. For example, lineary sorted information can be kept in an array. Information can be efficiently found via binary search, while insert and delete operations are cheap in terms of time (one simple move operation).

2.4.4 Address Management

A nest can implement management of the resources it provides through this operation group. A user can allocate exclusive address space and later return it to the nest, analogous to the C-style functions "malloc" and "free". The main difference is that *addresses* are allocated. What can be found behind these addresses depends on the nest implementation. For example, a "heap" nest could decide to return an empty address space or one filled with zeroed data. A

pipe reader nest could fill it with information read from some stream. The basic definition of the operation pair only specifies that each "allocate" operation will never return an address range that has been allocated before but has not been returned via the "free" operation. In other words, these operations are used to (temporarily) claim some address space as exclusive domain.

2.4.5 Lock management

This (last) group of operations is what this thesis is about. The operations of this group handle synchronization of concurrent access to nest resources. With these operations, users can lock part of a nest and later release it again. When a user issues a lock request that conflicts with an existing one then it will either fail or wait until the conflict is solved, depending on what behaviour the user requested. The operations that perform locking and unlocking are, not surprisingly, called `$lock` and `$unlock`.

2.4.6 Resource Retracts

Until now, nests have not shown any solution for an old problem in programming, depletion of resources due to resource "leaks", meaning allocated but unreturned resources. Athomux provides an interface for this: resource retracts. On resource shortage (or for some other reason), an output may issue a special operation to one or multiple inputs connected to it, asking for return of resources. On serious resource shortage, it may simply notify the input of the forced retract of resources, what will most probably make the brick that suffered from the retract fail in the near future.

2.5 Strategy Bricks

Bricks cannot influence the wiring of their inputs and outputs. This is the domain of control bricks. Every part of the whole Athomux brick network is managed by such a control brick. It can create, connect and destroy bricks. If itself is part of its own domain, this brick can (dis-)connect its own inputs and outputs or even self-destruct. Such a brick can give control over parts of its domain to sub-control bricks by providing a "strategy" output nest. This is one of possible definitions of the *contents* of a nest. A strategy nest provides *information* about the structure of the brick network and may provide ways to *modify* it. So any brick that gets connected to this output gains power over the brick structure. This is probably the area most sensitive to security issues in Athomux. Any brick connected to a strategy nest can in turn provide an own perspective on the brick structure to other strategy bricks. For example, this could be a filtered view of the strategical information at its strategy input, a new strategical subdomain, or two strategical inputs merged into one output.

The concrete format of the strategical information has not been fixated yet. It is not that important, anyway, because on introduction of a new format, back-

wards compatibility can always be provided through adapter bricks translating between the new and the old format. First was a textual format that aimed at easy debugging and maximal generality. Its main benefit is human-readability and easy extensibility. The main drawback is low performance due to parsing and reconversion to strings. During this work, an operational strategy nest using a new operation group was introduced, together with an adapter brick between them. Hardy Kahl tested it in his work about network transparency [4]. Its main benefits are surely easy use of the interface and performance. The main drawback is the ballast non-generic operations every nest must carry but have no function in non-strategy nests. Moreover, once we recognized that the disadvantages outweigh the advantages, we saw that a revision (removal of those additional operations) cannot be compensated simply by inserting an adapter brick because the nest interface is uniform and operations can removed only everywhere or nowhere. Anything else (to no doubt possible) would destroy the major advantage of Athomux, universal brick composition.

Chapter 3

Concepts

3.1 Concurrency

The term "concurrency" describes the problem of simultaneous access to shared data by multiple processes. The terms "process" and "thread" have already been given very specific meanings by modern operating systems (i.e., a "process" to be the unity of an address space and a number of "threads", each an active flow of operation). We will avoid these terms, since they could lead to confusion. Instead, the term "control flow" will be used as abstract description for an algorithm being executed by a processor. Modern systems allow for simulteneous execution of multiple control flows, called *multitasking*. Semantically, there is no difference between those control flows being executed on separate processors, on separate network nodes, or by one processor simulating simultaneous execution by rotatory assignment of short time slices to each active flow (*preemptive multitasking*, in contrast to *cooperative multitasking* requiring explicit voluntary handover of the processor resource to other control flows, what would pose a semantic difference due to predictable scheduling).

Multiple control flows each working on exclusive resources (e.g. a local stack frame) cannot influence each other and can run totally indedendently. A problem arises as soon as resources are *shared* between multiple control flows. Algorithms are the stepwise implementation of mathematical functions with an input and output domain. To achive a correct result after execution of an algorithm, they require

- the input domain to stay constant during its evaluation
- the output domain to be exclusive during the storage of the result

Furthermore, there can exist a dependency relation between multiple executions of different algorithms, forcing the availability of synchronization mechanisms.

3.2 Synchronization

Synchronization denotes mechanisms to delay execution of a certain part of code by control flows until a predefined condition arrives. Such control flows either repeatedly keep checking for that condition (*busy wait*) or are sent to sleep (removed from the active schedule) by the supervising process management until the condition arrives. Examples of synchronization mechanisms are

- *Mutex*: permittance of only one execution of a certain algorithm at a time.
- *Semaphores*: execution delay due to exceeding of a predefined maximum number of control flows allowed to execute a certain algorithm.
- *Monitors*: protection of a certain data object (encapsulated data with well-defined access operators) from simultaneous access by multiple control flows.

All these basic mechanisms are equivalent and can usually be implemented efficiently using very few native machine instructions. Higher- level synchronization mechanisms like Athomux locks can be implemented based on such low-level operations.

3.3 Synchronization in Athomux

Data operations in Athomux occur exclusively in (output) nests, and so does synchronization. Synchronization in Athomux means locking of an address range in a a nest against concurrent access. Athomux distinguishes two lock grades, "read" and "write", agreeing with common understanding:

- *Read locks* are applied to a nest to signal temporal immutability of the data in a certain nest region. Multiple concurrent read locks do not conflict with each other.
- *Write locks* could also be called exclusive locks, since only a single write lock of a certain nest region is allowed at a time. Any other concurrent lock request will conflict and so either fail or block the requester.
- A write lock always implies a read lock, analogously does the release of a read lock imply the release of a (possible) write lock.
- Locks can be up- and downgraded after obtaining, to signal a change in the demands of the operation to the locked nest region.

Viewing an Athomux nest operation as an algorithm, it would lock its input domain against concurrent modification through a read lock and its output domain against any concurrent access through a write lock. While the input domain can be read by other operations concurrently, the output domain remains exclusive to the algorithm until release. Transition between lock grades is performed by

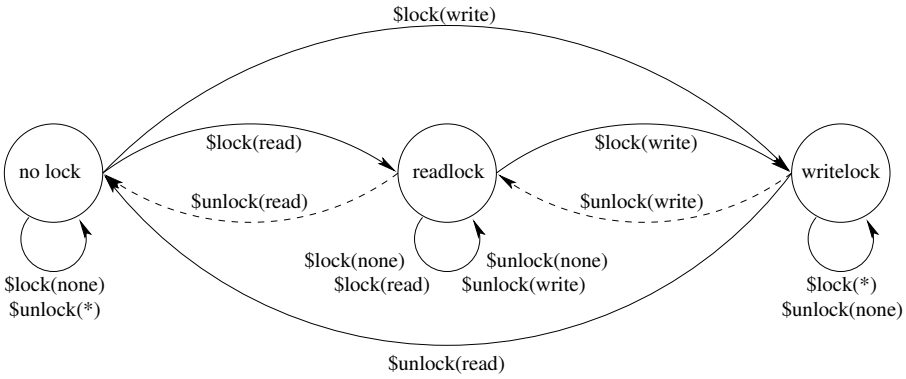

Figure 3.1: Nest recources can be locked in different grades to increase concurrency. Lock grade increases from left to right. The `$lock` and `$unlock` operations can be used for transitions between lock grades. Only operations increasing the grade of a lock can cause conflicts and thus either fail or block the caller.
Note: At the time of this writing, the transitions illustrated in dashed lines were not implemented due to a ongoing discussion about their usability.

the nest operations $lock and $unlock, as shown in figure 3.1. Note that these transitions can only be performed by the lock owner. Lock operations issued by others will either coexist or conflict (see above) with existing locks.

Most conventional lock mechanisms apply to predefined data objects and either lock them entirely or not at all. Locks in Athomux instead apply to *logical address ranges*, held locks can be released in different granularity (i.e., different partitioning) than they were originally obtained in [3].

Apart from data io operations, Athomux nests also define an address mapping, so it is natural to also define locks on addresses. They have the following meanings:

- *Address read locks* signal an immobilization of a nest region, to protect it from being destroyed, moved away, or overwritten by another address region moved "onto" it.
- *Address write locks* signal the requestor's will to modify the nest structure and to keep off other control flows depending on its stability.

Normally, Athomux processes are expected to cooperate, (i.e., to make explicit use of lock operations when accessing a nest). The `$lock` and `$unlock` operations are implemented independently from the other operations (*orthogonality*), allowing a separate implementation of these operations "on top" of another nest anywhere where necessary. This avoids unnecessary overhead wherever synchronization is *not* necessary (e.g., when only single-user operation is demanded). In cases where a nest really needs "protection" (e.g., from concurrency- unaware

bricks not using lock operations, or from external, untrusted bricks), an appropriate protection brick can be inserted that implements this functionality.

3.4 Resources

In Athomux' philosophy, nests are considered resource providers. In contrast to consuming semantics, like in UNIX pipes, resources are temporarily allocated ("borrowed") from a nest to read or modify and to finally return them. These resources are address space, data space, and locks. Data and address locks are separate resources (instead of a functional dimension of the resources "data_space" and "address_space") because they are implemented independently from the other functions working on these resources.

Allocation from and return to a nest is specified by the *paired* operations (i.e., `$gadr`/ `$padr` for address space, $get/ `$put` for data space, and `$lock`/ `$unlock` for locks). Resource allocation can only be temporary, they all *must* be returned to the nest sooner or later. On demand, they may be even *reclaimed* by the output nest.

Singular operations, on the other hand (e.g., `$trans`, $move), just make one-time use (reading or modification) of resources' *contents* without "retrieving" them from the nest. Note that `$create` and `$delete` are *singular* operations, no *paired* ones. They do not *allocate* part of the resource "address space" from a nest, but modify its *contents* (i.e., (re-)assignment of address to data space). This is analogous to the `$trans` operation reading or modifying the *contents* of the resource "data space" (i.e., the data) but not allocating any. You can also think of `$create` and $delete as special `$move` operations to or from some background pool of spare data space.

3.5 Mandates

When you lend out posessions of yours to someone then you want to know to *whom*. For an effective resource management, nests also need to identify the *owner* of a resource. This is important for nests to

- know what resources are being returned (address ranges of read locks, as well as those of allocated data buffers, may overlap, so the logical address alone is no unique identification for a resource).
- associate "stale" resources with their owner who is unable (or unwilling) to return them (e.g., after having crashed and been removed) and need to be retracted ("garbage collection").

In a distributed system like Athomux, physical resources cannot be used as identification of *individuals* because they have only local (i.e., on one host machine) scope and could be ambiguous across network nodes. Athomux defines *mandates* to identify resource owners. Every operation invocation is labeled with a mandate to identify the *individual* that authorized it.

Note that mandates not only sign *passive* resources (allocated from nests) but also *active* resources (processor time) that need association with *individuals*, too. In Athomux, there are no "process" or "thread" ids because they, as physical resource identifiers, too would be meaningless across host boundaries, just like physical identifiers of passive resources (e.g., pointers). Also note that multiple control flows are allowed to carry the same mandate, what also is unlike process- and thread-oriented systems where a thread id is uniquely assigned one thread.

A mandate identifies an abstract *task* (do not confuse this term with the notation for a "process" in certain operating systems) to accomplish, and all resources allocated by it. For example, a mandate could be associated with an initial operation invocation and be passed on to further operations caused by it. The mandate would then end with the return of the original operation (and resources left behind could be collected). Another association target of a mandate could be a nest or a brick instance (or a group of brick instances) that uses it to sign all operations originating from it. The mandate then would become free for garbage collection on deinstantiation of the mandate-holding instance.

Of course, mandates must always be unique. One way to achieve this could be, for instance, a central mandate management nest where all mandates are allocated from. But this could make later connection of two brick networks difficult, if they had been totally independent before, because mandate ambiguities could arise. But there are ways to solve such a situation, for example through (transparent) insertion of mandate mappers between the brick networks.

3.6 Bottlenecks

Bricks inmidst of the global brick network seldomly act as a provider of new resources, they mostly act as a resource *transformer* from their inputs to their outputs. As a consequence, operations usually cause a chain of operation invocations up to original resource provider bricks. On the way, they can pass through potentially many mappers, buffers, caches, heaps, pipes, connectors, etc. In this chain, there may exist "bottlenecks", bricks that cause significant latencies (e.g., bricks implementing network connections or encryption). To reduce negative impact on performance, we implement *lock caching*.

3.7 Lock caches

Lock caches are a key technology for efficient concurrency management in Athomux. A lock cache is an "outpost" of a lock manager that appears to client bricks just like an original lock manager. Unlike an original lock manager, however, a lock cache does not have the whole nest as its domain but only the parts it has "borrowed" from the original lock manager (or from a previous lock cache). When a request for a lock arrives that it does not have "in storage", it redirects the request to the lock manager. Otherwise, it can simply grant the lock from

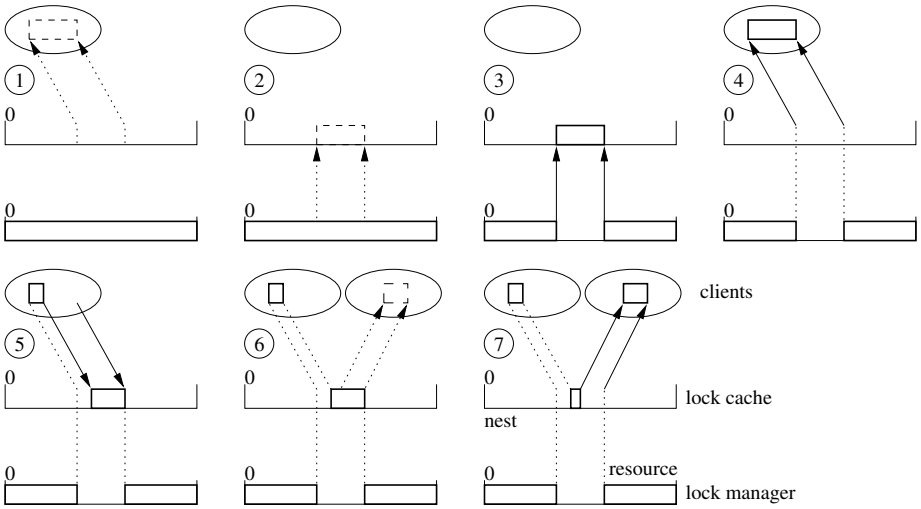

Figure 3.2: Function of a lock cache, illustrated in 7 steps. A client (the oval above) is connected to a lock manager's output nest (the scale below) via a lock cache (whose output nest is symbolized by the scale in between). (1) Initially, no resources of the lock manager (i.e., the locks) have been allocated yet. A client requests to lock a part of the nest it is connected to. (2) The lock cache does not own the requested resource (it does not hold a lock on this address range), so it requests it itself from the lock manager. (3) The lock manager grants the lock to the lock cache. (4) The lock cache in turn grants the lock (it now owns and so can redistribute) to the client. (5) Later, the client returns part of the obtained lock, but the lock cache keeps it, instead of returning it to the lock manager. (6) Another client allocates a lock from the nest. (7) This time, the lock cache already owns the requested lock and so can immediately grant it to the client, sparing (potentially costly) communication with the lock manager.

its own "stock" (see figure 3.2). The advantage is obvious: many lock requests with high locality can be reduced to few by a lock cache preserving locks released by a client, instead of returning them to the superordinate lock manager. This can drastically reduce communication traffic, and so increase performace at slow connections. Widely-shared resources, like distributed shared memory (DSM [5]), can be significantly be increased in performance by lock caching.

Since the lock cache works as kind of a "lock reseller", it issues all resource transactions to the resource manager under its own *brick mandate*. This way, the lock cache always keeps control over the resources allocated by itself. To the superordinate lock manager, all locks held by clients of the lock cache are marked as owned by the lock cache. When redistributing the lock resources to clients that the lock cache stores the client mandates in its own local registry.

It is hard for a lock cache to decide when to return resources currently not

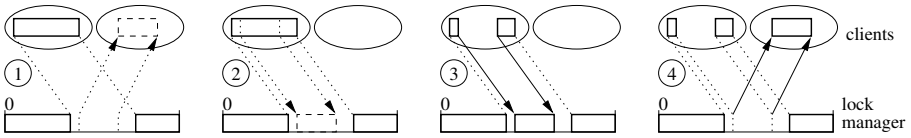

Figure 3.3: Resource retract. An example in four steps. Two clients are connected to a lock manager's (could as well be lock cache's) output nest. (1) The right-hand client requests a lock on a nest range that is (at least partly) held by the client to the left. (2) The conflicting nest range is requested for retract from the owner by the lock manager. (3) The retract is granted by the left-hand client (otherwise, the client on the right would have to wait, or the lock manager could force retract). (4) The returned resources are granted to the client to the right.

being used by clients. Actually, it is designed to keep them as long as possible. So another client directly connected to the lock manager might get blocked when requesting a lock held by a lock cache but not used by any client (i.e., a *cached* lock). To prevent a deadlock, the lock manager needs a way to *reclaim* resources.

3.7.1 Resource Retracts

Athomux input nests define an operation for such a *resource retract*. Currently, it is the only operation that is directed "the other way around", from outputs to inputs. When an incoming lock request conflicts with an existing one, the lock manager can decide to ask the current owner of the lock resource to return it. The requester has to wait for the retract to succeed, or to abort and try again later (when the lock has been returned). If retract is repeatedly denied by the owner, the lock manager can later decide to urgently demand return of the locks or even forcibly retract them without further notice (comparable to a SIGKILL in UNIX after an unsuccessful SIGTERM).

3.7.2 Lock registry of a lock cache

A lock manager maintains an internal dynamic data structure where it registeres what address ranges have been locked by whom (by what mandate). This way, locks can be effectively managed among lock clients of the lock cache. But unlike an original lock manager, a lock cache also has a superordinate lock manager that can retract locks from the cache. But instead of maintaining a separate data structure to store which locks the cache itself holds and which still belong to the lock manager, the lock cache can use a little trick to maintain both sub- and superordinates in one registry: it considers the lock manager another lock client. When having an abstract look at the operations clients and the lock manager send to the lock cache, it turns out that they are equivalent. The following table illustrates this:

Logical operation	Client-side operation	Manager-side operation
request	$lock	$retract
grant	return $lock=ok	$unlock[1]
deny	return $lock=failure	return $retract=failure

The operations `$lock`, $unlock and `$retract` are considered different versions of the abstract operations *request*, *grant* and *deny*. *Request* has the direction "from client to cache", the other two operations "from cache to client". The cache has just to maintain a unique mandate to represent ownage by the lock manager. The example implementation chose the lock cache's own brick mandate since it is unused in the lock registry (lock ranges owned by the cache itself but not redistributed to clients are simply marked as "not locked" in the registry).

One special operation has net been mentioned in the table, because it can only be sent by the superordinate lock manager: the urgent retract. Logically this could be called a "forced return" of locks. The lock cache can't do anything else than check its registry for clients concerned by this operation, delete their locks and forward the retract notification to them.

3.7.3 Order of Operation Delivery

A problem of distributed systems arises in Athomux, too: consistent order of delivery. In Athomux, all operation calls should be regarded messages. One brick that concurrently sends two (or more) operations out of one of its inputs can *not* assume that both operations will "arrive", i.e., will be executed, in the same order at the destination brick. This is true even if the connection is just local: when the scheduler transfers the cpu resource just after the first operation has been called, but before the first instruction of the operation in the destination brick has been executed, the later operation may be executed before the first one. In most cases, this is not an issue. But on lock requests by different clients (one of which could be the superordinate lock manager, makes no difference) overlapping both in space (address range) and time (concurrency), a race condition can arise with serious results if the wrong candidate wins:

- Client A and client B both request a lock on an address range X (the overlapping range of both requests).
- The request of client A is granted first. The operation starts returning success.
- Client B must wait for the lock that has been granted to A. Nonetheless, a `$retract` is sent out to notify the holder of the lock: Client A, who indeed does not hold the lock yet, since the grant has not yet reached him.
- If the lock grant reaches Client A first, everything is alright and consistent.
- If the retract reaches Client A before the lock grant, it will "succeed" (because A does not yet know that it owns the lock).

- When the lock grant finally reaches Client A, it now knows that it has become the owner of the lock, but the `$retract` operation already has returned.
- If Client A does not return the lock without request (e.g., if it is a lock cache itself), Client B will starve waiting forever for the lock to be returned.

(Solution follows)

3.8 Retract Filters

In Athomux, multiple inputs can connect to one output, but not vice versa. A retract operation invoked at an output with multiple inputs connected to it (in contrast to a "selective" invokation at a certain input) results in multiple operation invocations, one for each connection. For performance reasons, they will be invoked in parallel. Thus, a retract, while propagated further down the network, can spread into a potentially huge number of operations, increasing exponentially. This can seriously impact overall system performance by using up a lot of processor time unnecessarily, because retracts will most probably in the end address only one or few more bricks that hold the locks to be retracted. So it would be best to find out the paths to the bricks concerned in advance, but a major property of *instance orientation* is obscurity of neighboring brick instances, so the lock manager cannot "know" anything about the paths to the lock owners. There has to be a way (i.e., a brick) to show retract operations the right way through the brick network, *after* they have been "unleashed".

To design such a *filter* brick, we must first have a look on operation paths and possible furcations and unions of this path in Athomux brick networks. An input operation can fork either explicitly, by a brick forwarding it to multiple outputs, or implicitly, through multiple inputs connected to one output. Two paths can also (re-)join inside a brick, what could cause a duplication of a previously forked `$retract` operation but is no problem, because a resource can only be retracted once. An example path a retract operation could take through a brick network is illustrated in 3.4, together with the correct placement of filter bricks to reduce the number of unnecessary operation invocations.

A retract filter brick must determine whether the `$retract` operation at its input should be forwarded to its output, and if it should, to which input(s) connected to it. It can do so by logging successful $lock and $unlock operations going through it. When a `$retract` operation arrives at its input, the brick can decide if the lock addressed by the operation was granted via the path through it or not, by looking at the log. If not, the `$retract` operation will simply return *true*, meaning "retract of all specified resources that were allocated via this path will be released". That statement *is true* if the set described by that statement is empty. At a furcation of the path, the results of all returning `$retract` operations will be combined by a logical AND and then returned to the retract requester. So only if *all* parts of the request for retract have been granted, the

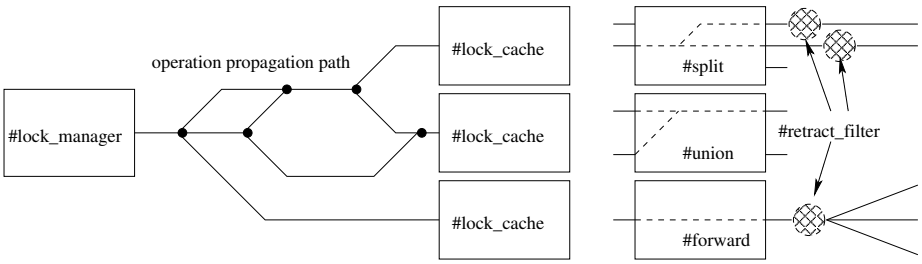

Figure 3.4: Potential furcation and joins of an input operation's path starting at `#lock_manager`. On the right-hand side, the possible ways to split and join paths, together with senseful placement of filters, are illustrated (path furcation and joining through input operation forwarding, and path furcation through multiple input connection).

result finally returned to the lock manager will be *true*. This is also the reason why the return of the `$retract` operation to the lock manager cannot cause an immediate release of the locks. If the retract of only one little part of the whole request has been denied then the final result is *false*. In that case, the lock manager would not know *which* locks have been successfully retracted and which have not. So the actual release of the retracted resources has to take part via an explicit `$unlock` operation. Except for forced retracts, of course, because they do not expect an answer in form of an explicit lock release. However, in both cases, requested and forced retract, the result of the `$retract` operation is irrelevant to the lock manager, so it does not even need to return, what reduces traffic even more.

3.9 Optional Locking

Optional locking [3] is the Athomux way of *speculative locking*. When requesting a lock on a certain address range in a nest, the requester can add an *optional* part that can be granted partially by the lock manager, depending on conflicts with existing locks. The optional part of the request *never* causes a lock operation to fail or, on synchronous invocation, to wait. Of course, the *obligatory* part of the lock request still shows standard behavior (i.e., it is either granted or denied entirely). This way, communication traffic an be reduced even more, optional locking works as a kind of "forward caching" 3.5.

It is cumbersome, however, for a user brick to implement speculative locking itself, although it may be the instance that can best predict what will be locked next. But besides the expense of additional programming overhead, to concentrate features in bricks is not the Athomux way. Outsourcing of optional locking has many advantages:

- No additional (and redundant) programming efforts necessary in user bricks.

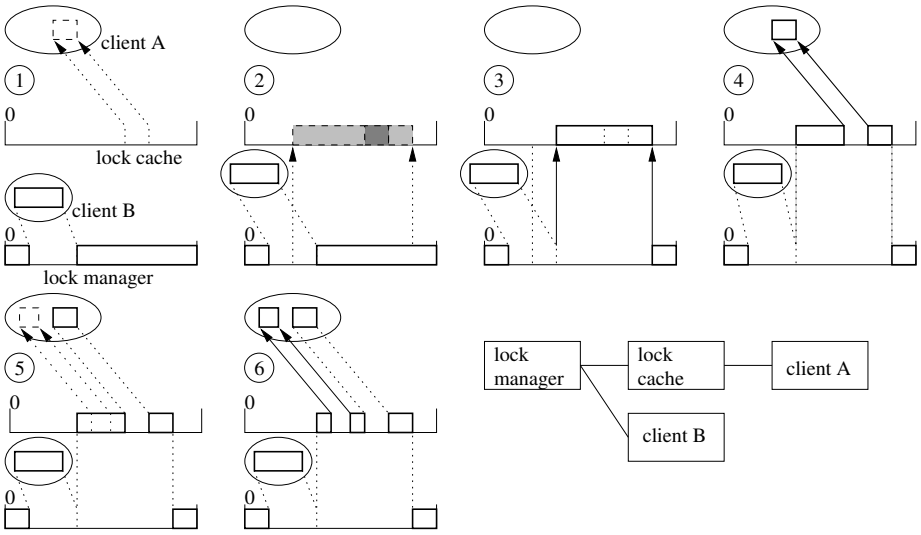

Figure 3.5: Optional locking. A demonstration in 6 steps. We have a client "A" connected to a lock manager indirectly via a lock cache, and a second client "B" connected directly to the lock manager. (1) Client "B" already owns a lock on part of the lock manager's output nest. Now client "A" reqests a lock from the lock cache. (2) The lock cache adds an optional lock request, speculating for future lock requests from client A, and sends the request to the lock manager. (3) The lock manager recognizes a conflict between the lock request from the lock cache and the existing lock held by client "B". It decides not to send a retract request to client "B" but instead only partly grants the optionally requested lock to the lock cache. *Note*: a lock manager could as well decide to retract from client "B" and wait for the answer before granting the lock cache's request. (4) The lock cache can now grant the lock requested from client "A". (5) Later, client "A" requests another lock from the lock cache. (6) Due to speculative locking, the lock cache already owns the requested lock and can serve client "A" immediately, without any requests to the lock manager.

- Increased modularity (an important feature of Athomux).
- Application of optional locks only where necessary (e.g., for a user brick "not far from" the lock brick, optional locking would yield no increase in performance but only cause additional overhead).

3.10 Problem of Operation Atomicity in Multiuser nests

Until now, only *inter-brick* synchronization has been addressed. Until now, all nests have been presumed to be *multiuser nests*, what means nothing more than the nest supporting multiple simultaneous operation calls with a determinable result. In the context of concurrent programming this is called *atomicity* or *linearizability* of operation calls. This includes *reentrance* of all operations the nest implements.

The convention that Athomux bricks should be implemented *statelessly* (no brick-internal variables) could lead to the assumption that the property "multiuser" of a brick's output nests is determined solely by that of its inputs, but this is only true for stateless bricks where every incoming operation results in maximally one outgoing operation (e.g., in simple, static mappers). As soon as the execution of an operation requires invokation of more than one operation, further (internal) synchronization is necessary.

In most Athomux bricks, operations access a common input nest region, often a dynamic data structure (e.g., a list, tree, etc.). This data structure configures the transformation implemented by the brick, it is often referred to as "brick state", although data stored in external nests is not part of a brick. Anyway, such a data structure shared by potentially several operations running concurrently is a good example for data needing brick-internal synchronization. Remember that *reentrace*, and so *atomicity*, of a group of functions needs explicit synchronization as soon as modification of shared variables is involved. Now, mandate-based locks pose a problem: two simultaneously executed operations with the same mandate could cause conflicting modifications in the "state" nest. For the sake of atomicity, they would need synchronization but locks carrying the same mandate do never conflict.

To implement true *critical sections*, a new lock brick is required that ignores mandates and where locks never "melt" instead of conflicting. This is a much simpler approach to synchronization, and one could think of always using this brick for synchronization, even *between* bricks. But such a simple lock brick lacks resource owner identification, a feature that is absolutely necessary for lock caching, retract, optional locking and garbage collection. *Inside* a brick, this is no severe issue, since the "individuals" there are not mandate holders but operation executions. An operation that allocated local locks is to release them before returning to the caller. And even if it does not, due to faulty code or a sudden (i.e., externally forced) abortion of the control flow, the resulting error condition can simply be resolved by reinitialization of the surrounding

brick (what involves reinitialization of all internal bricks, too, including the lock brick). So, *locality* of errors is preserved.

An additional convention implied by the use of an internal lock brick is the use of an exclusive *brick mandate* for operations synchronized by it. Semantically, they express resource ownage by the brick that uses internal locks. The reason for this is the danger of deadlocks that can occur when subsequent nests block an operation that holds an internal lock that in turn blocks another operation that is supposed to release the subsequent lock. This is equivalent to the problem of *nested monitors* [7]. Usage of the same *brick mandate* for all operations working on the internally synchronized nest will make subsequent mandate-based blockings less likely (locks with the same mandate do not conflict) and, even if they do occur, they will make the source of the deadlock identifiable and thereby the deadlock itself resolvable.

3.11 Adapters

The problem of a singleuser nest (presuming only one operation call at a time) to be connected to inputs requiring a multiuser nest could occur[2]. This may never happen, because this could take the brick providing the nest into an undetermined state (in other words, make it crash). The best solution is to upgrade the brick to make the respective output a multiuser nest. But this is only possible if the brick source code is available (and analyzable with reasonable efforts). An alternative could be reimplementation of the brick. Another solution is to set up a "monitor" (in classic terminology) "around" the nest, just like you would, for example, prepare an obscure *Java* [6] object for concurrent access by deriving a "synchronized" (Java keyword) version. In Athomux, this can be done by connecting an appropriate *adapter brick* to the nest, a brick that forces operation execution one-by-one. This is to no doubt effective but highly inefficient because it destroys all benefits of a multiuser nest over a monitor. Multiuser adapters pose tight "bottlenecks". When placed "to the left" (near original resource providers), they can cause a great loss of performance due to a "traffic jam" of operations waiting for the operation currently being executed "behind" the adapter. When placed on the far right (near user bricks), the operation currently being executed can delay subsequent operations for a long time (see figure 3.6).

The conclusion is to avoid multiuser adapters whenever possible. Bricks should always implement multiuser output and input nests to maximize concurrency.

Furthermore, the problem of deadlocks mentioned in 3.10 becomes even worse because the adapter brick cannot simply change the mandates of incoming operations to its own one just to prevent deadlocks. Such behaviour could

[2]This is equivalent to the use of libraries that are not *thread-safe* in a conventional, multithreaded environment. For example, some old versions of the standard C library (printf, errno etc.) are not thread-safe and need external synchronization.

Figure 3.6: `#adapt_multi` bricks pose an extreme bottleneck wherever they are placed. At the left border of the brick network (near original resource providers), there is a potentially high load of incoming operation invokations that can only pass one-by-one. At the right border (near "application" bricks), an operation may take a relatively long time and make other operations wait for it.

cause new deadlocks due to later interference with operations *not* having gone through the adapter.

3.12 Strategy

Now we have gathered all technologies necessary for an efficient concurrency model in Athomux. We have seen the benefits from optional locking and that of lock caches mitigating "bottlenecks". Now, an instance is required to control automatic placement of lock caches. A *strategy brick* has to be designed to analyze the brick network and to insert lock caches and lock filters wherever necessary.

On instantiation, the lock strategist must analyze the brick network it is responsible for by reading the strategy nest it has been connected to. It first searches it for lock managers whose existance is necessary to make a sense out of lock caching (otherwise, there is nothing to cache). Next, all possible paths lock operations (`$lock`, `$unlock`, `$retract`) can take from or to each resource manager through the brick network must be traced. To be able to do this reliably, every brick on the way needs to specify *attributes* that tell about abstract internal "connections" between its inputs and outputs[3] (i.e., what lock operations on what brick connectors can cause corresponding operations on what connectors on the opposite side). Only paths between lock managers and existing lock generators (they also must identify via an attribute) must be considered because on other paths, no lock traffic is expected. Whereever the path splits (through a brick "connecting" one of its inputs to multiple outputs, or by several inputs connected to an output), the strategy brick should insert retract filters and caches appropriately (see 3.7). Application of caches and filters not only depends on the way the path splits but also on the subpaths' containing of lock generators. Paths that do not lead to any lock generators are excluded from retract by a filter and not designated for further lock cache placement.

[3]Such attributes are hard, if not impossible, to generate automatically. This would need systematic brick code analyzis, what is impossible in the common case (see the Hoare calculus [9]).

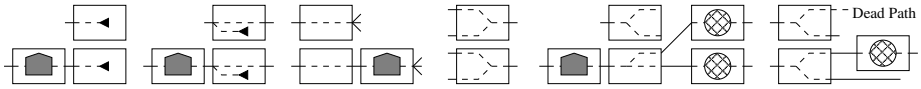

Figure 3.7: Brick configurations the lock strategy brick encounters along a lock path, and correct application of filters and caches (above the respective situation, below the solution). The "warehouse" symbolizes a lock cache, the "sieve" a retract filter. From left to right: (1) lock generator at the end of a path (2) A "hybrid" lock generator and lock forwarder (3) Lock forwarder with multiple paths connected (4) a path unifier (5) A path splitter (6) A path splitter with only one connection leading to lock generators.

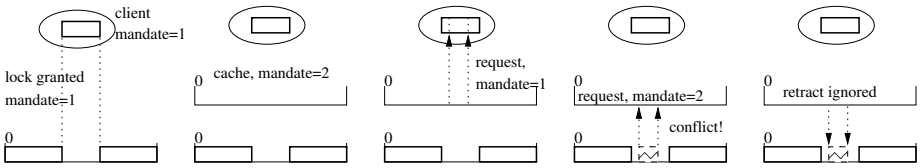

Figure 3.8: Insertion of a lock cache at runtime without reset of the clients. One client holds a lock of the lock manager with its own mandate which is 1. Now a lock cache gets inserted that has no knowledge about locks held by clients. The client now requests its lock again, what normally is harmless. But the lock cache tries to obtain the lock under its own mandate (2), what causes a conflict with the existing lock held by mandate 1.

Removal or insertion of lock caches implies deactivation of all bricks that use resources the brick is supposed to cache. This means all bricks that are connected to the nest the lock cache gets connected to. This is necessary since the lock cache must track *all* lock requests. It *must* own all locks "handed on" to client bricks, otherwise conflicts could occur due to the lock cache issuing its lock requests with its "own" mandate: imagine a client holding a lock obtained before the insertion of the lock cache (or see figure . For some reason, the lock client may try to request the lock it already holds again. By operation definition, the lock request with the same mandate as the existing lock should be granted immediately (and not change anything). It would be if there was not the lock cache in between. It cannot know that the lock requested has already been granted by the lock manager in the past. It forwards a lock request *with its own brick mandate*, what results in a conflict at the lock manager with the lock originally granted to the client. The resulting retract operation would not reach the client because the lock cache acts as a retract filter too. The range to be retracted is not owned by the lock cache, so it ignores the retract operation.

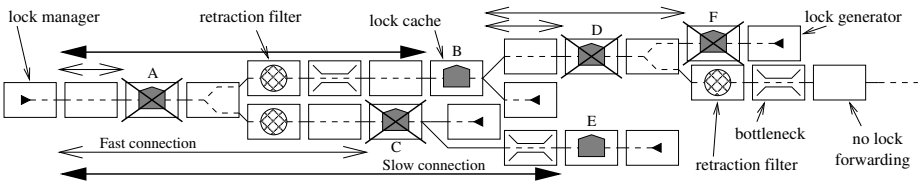

Figure 3.9: Optimization after lock cache placement. Lock caches A and C signal a fast connection to the lock manager to the left, and so get removed by the strategy brick. B and E now measure the connection "distance" to the lock manager directly, instead of the removed caches. They each measure long operation durations and so are kept. Now, lock cache D measures connection quality up to lock cache B which appears as a lock manager for D. It is a fast connection, so lock cache D gets removed. The same applies to lock cache F.

3.13 Optimization

The rules for placement of lock caches, initial or on runtime changes in the brick network, can result in too many lock caches that may yield an overall performance gain, but far from the optimum: a lock cache always means overhead that may outweigh the benefits of it. Initially, lock caches are placed at any point where they could make sense. At runtime, it is up to the strategy brick to determine superfluous lock caches (i.e., lock caches that get their benefits outweighed by the overhead they cause).

A brick cannot be "tested" for performance directly by calling an operation and measuring the time it takes "through" the brick because an external brick observer could not tell whether an operation emitted from a brick was caused by one of its operations previously called. Besides, in distributed systems, there is no global time, anyway. Time measurement can only take place at a single spot. In Athomux, this means measurement of the time a single operations takes for execution (until return), but operation execution will probably involve several operations in different bricks. Fortunately, in case of lock caching, we do not need to test bricks for performance one by one, it is sufficient to measure the time of travel for operations between a lock cache and its superordinate lock manager (that could be a lock cache, too).

One way to test connection quality is to "ping" (call a special operation that returns immediately from) the lock manager it caches locks of. The lock cache measures the time until the operation returns. The result of this measurement is then published at an output nest of the lock cache for evaluation by the strategy brick. This way, the strategy brick gains an overview of the durations for operation travel from lock cache to lock cache, and can then decide to remove certain lock caches. For decision, the strategy brick can, for instance, use a predefined threshold, a minimum time this "ping" operation must take to legitimate the existance of the lock cache.

The order in which lock caches are to be removed is important. The order

"from left to right" guarantees that the lock caches directly to the right of "bottlenecks" are more likely to be kept by the strategy brick than those nearest to lock generators (see figure 3.9). If the path to the right of the "bottleneck" splits further, this technique keeps the number of lock caches low, and also the number of retracts, in the case of lock generators at the ends of those subpaths have interleaving lock requests.

The strategy brick must, of course, repeat this optimization whenever the brick network is changed. Additionally, periodical measurements of connection quality at all places where lock caches were removed are necessary, since bricks could become bottlenecks at a later time. It would be an unnecessary waste of resources to keep "sniffer" bricks at all such positions all the time. It is sufficient to temporarily connect such a brick to a nest where a lock cache was originally connected to. It could test the connection without slowing down other traffic significantly. Only an increase of the "ping" time over the threshold would make the strategy brick insert a new lock cache.

It can be temporarily very costly to insert a lock cache into a brick network, because for removal or insertion of a lock cache it is necessary to have all locks returned to the superordinate lock manager.

Chapter 4

Appendix

4.1 Appendix A: Nests formally

This section is only provided for deeper understanding of Athomux nests and is not vital for the understanding of this thesis. Skip it if you are not interested.

Formally, a nest is a composition of three sets and two functions between them. The sets are *address space*, *data space* and *values*, the functions are $addr : address\,space \rightarrow data\,space$ and $data : data\,space \rightarrow values$. A nest specifies operations to (de-)allocate subsets of both functions' domains, to lock the functions partially, and to redefine the functions (i.e., "modify their contents"). Locking of one of the resource can be viewed as a separate resource, a second dimension of allocation. Both address and data space locking has been merged into one operation pair, due to their close relation (often, there is the need to atomically lock both resources).

address space is a total order of integers with a (current) cardinality of 2^{64}, *values* also is a total order with cardinality of 2^8 (that of bytes, but a higher cardinality can be achieved by combining multiple elements). The set *data space* is not defined any closer, its character depends on the way the brick implements it (its origin). Thus, its elements cannot be expressed directly but only indirectly through the function *addr*. The definition of *addr* can only be modified in the way the operations `$create`, `$delete`, and `$move` are defined, the *domain* of *data* automatically changes with any modification of *addr*. The "standard" behavior[1] of these operations keeps *addr* a partial, bijective function.

The lack of a definition for *data space* has a few side effects:

1. The operations allocating *data space* and modifying *contents* (`$get`, `$put`, `$trans`, `$wait`) do not take a subset of *data space* as argument but a subset of *address space* which is then mapped to a subset of *data space*.

[1] The "standard" specification for a dynamic nest is (1) initially totally undefined, (2) `$create` assigning new (not assigned until then) data space to logical addresses.

2. Locks of *address space* subsets analogously have to be expressed through *address space* subsets.

3. For the whole duration of a *data space* allocation or locking, the *address space* subset used for the respective operation must be fixated at least (i.e., read-locked), otherwise a later release of the resource or lock may become impossible for the user.

Remark: there has been a discussion about whether to make `$put` take a physical pointer as argument (that formerly has been returned by the corresponding `$get`) instead of a logical address, to work around 3. Besides the fact that this method cannot be used for `$lock` and `$trans`, the host's physical address space cannot be used synonymously for *data space* because the nest the data originated from may reside on a different host (e.g., a different cluster node) on which the physical address obtained is meaningless and therefore cannot identify a subset of *data space* on that host. As an example, think of a `#remote` brick [4] receiving a `$put` operation. The (local) physical buffer has to be flushed to a network stream with a *logical* address as destination because a *physical* pointer would have no meaning on the remote machine. So the addressed data space still needs to be reachable through its logical address.

Bibliography

[1] Thomas Schöbel-Theuer: Athomux

[2] Thomas Schöbel-Theuer: Instance Orientation

[3] Thomas Schöbel-Theuer: Optional Locking

[4] Hardy Kahl: distributed Athomux

[5] ??: DSM (Literaturempfehlung?)

[6] java.sun.com

[7] nested monitors

[8] Polymorphism

[9] Hoare calculus