Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

MoCha, a model for distributed mobile channels

J.V. Guillen Scholten

REPORT SEN-E0418 OCTOBER 2004

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

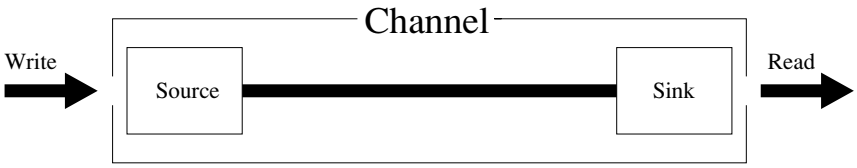
ISSN 1386-369X

MoCha, a model for distributed mobile channels

ABSTRACT
A mobile channel is a link that provides an asynchronous and anonymous means of communication between components in a distributed system. A channel is called mobile if either of its (channel-)ends can be moved from one component to another without the knowledge of the component at its other end. Such mobility allows dynamic reconfiguration of channel connections among the components in a system, a property that is very useful and even crucial in systems where the components themselves are mobile. In this thesis we present MoCha, a model for distributed Mobile Channels. MoCha describes an efficient and non-trivial implementation of an asynchronous mobile channel in a distributed environment. After presenting an intuitive explanation of MoCha using text and figures, and a simplified version first, we present the full version. Finally, we discuss MoCha, show that it is indeed an efficient and non-trivial implementation, and discuss future work.

1998 ACM Computing Classification System: C.1.4, C.2.4, D.1.3, D.1.5, D.3.2, D.3.3, E.1.0
Keywords and Phrases: Channels; Components; Coordination; Distributed communication;Dynamic distributed systems;Mobility
Note: This work was carried out at CWI and is written as a master thesis for the computer science department (LIACS) of Leiden University.

MoCha,
A model for distributed *Mo*bile *Cha*nnels



Juan Guillen Scholten
jguillen@liacs.nl
juan@cwi.nl

Master Thesis
May, 2001

Leiden Institute of Advanced Computer Science (LIACS), Leiden University,
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands.

Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413,
1098 SJ Amsterdam, The Netherlands.

Abstract

A mobile channel is a link that provides an asynchronous and anonymous means of communication between components in a distributed system. A channel is called mobile if either of its (channel-)ends can be moved from one component to another without the knowledge of the component at its other end. Such mobility allows dynamic reconfiguration of channel connections among the components in a system, a property that is very useful and even crucial in systems where the components themselves are mobile.

In this thesis we present MoCha, a model for distributed *Mo*bile *Cha*nnels. MoCha describes an **efficient** and **non-trivial** implementation of an asynchronous mobile channel in a distributed environment.

After presenting an intuitive explanation of MoCha using text and figures, and a simplified version first, we present the full version. Finally, we discuss MoCha, show that it is indeed an **efficient** and **non-trivial** implementation, and discuss future work.

Acknowledgements

Many thanks to dr. Farhad Arbab, dr. Marcello Bonsangue, and dr. Frank de Boer, for their support, comments, and ideas. I thank my thesis advisor prof.dr. Joost Kok for his support and his interest in the progress I made.

I thank my roommates at CWI, drs. Kees Everaars and Freek Burger, for their comments on MoCha, and I thank the rest of the coordination group at CWI.

A special thanks for my family and friends. And last, but not least, I want to thank the following persons, at LIACS, for their interest and help with many things during my study: Riet Derogee, dr. Jeannette de Graaf, dr. Walter Kosters, and dr. Michael Lew.

Contents

Chapter 1

Introduction

A mobile channel is a link that provides an asynchronous and anonymous means of communication between components in a distributed system. The mobility of a channel allows dynamic reconfiguration of channel connections among the components in a system, a property that is very useful and even crucial in systems where the components themselves are mobile. Examples include software for mobile telephones and wearable computers, as well as for smart-card-based electronic commerce.

To our knowledge, no efficient implementation for a general mobile channel has been presented in the literature. In this thesis, we present and discuss MoCha: a model for distributed *Mo*bile *Cha*nnels.

1.1 Outline of this Thesis

In this chapter we give an introduction to *component-based software*, *channels*, *mobile channels* and their importance. After that we present MoCha and give the motivation for its development. At the end, we discuss related work.

Chapter 2 gives an (intuitive) explanation of MoCha using figures and text, for a better understanding.

Chapter 3 presents a simplified version of MoCha. In this simplified version we do not take the movement of the channel-ends into account.

Chapter 4 gives the full version of MoCha.

In chapter 5 we discuss MoCha, give conclusions, and discuss future work.

1.2 Component-Based Software

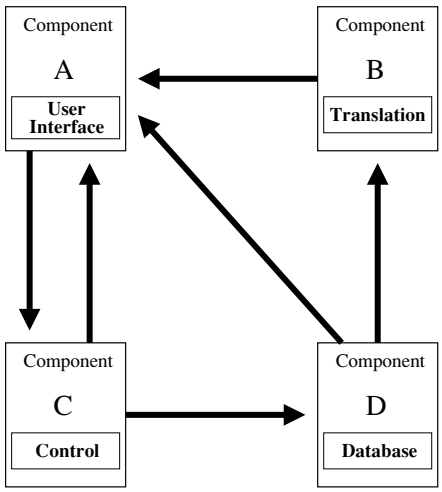


Figure 1.1: A Component-Based System.

In **Component-Based Software** systems are built out of *components* and *connections* among them. In figure 1.1 an example of such a system is given, which consists of four components and six (one-way) connections among them (arrows). This simple system allows users to obtain data from a database.

Component *A* collects user-requests, which is then sent to the **control**-component, which, in turn, checks whether the user can have access to it. If access is denied, the component sends an error message to the **user interface**-component, otherwise it sends the request to the **database**-component. This last component sends its data to the **user interface**-component. If necessary, the data is sent first through the **translation**-component. Component *D* must then somehow know which connection to use. A nice solution here is to use a mobile channel (see section 1.3.2). Another solution is letting component *C* tell the **database**-component which connection to use.

Components

A *component* is an abstraction of an entity or group of entities. It is a **black box**, no details of the inside of the component are visible to the external world. For interaction with the external world, components have an *interface*. This interface describes the input, output, and the observable behavior of the component [1].

For example, an *interface* of a component may tell us that, given a specific input, a window with a message will appear on the screen. However, how this is implemented in the component is hidden for the outside world.

Usually people define a component as an abstraction of a software entity. However, it is better not to restrict the implementation of components to just software. For example, a component can be a mixture of software and hardware. It can also consist of another component-based system itself. Therefore leaving the definition of a component as abstract as possible is a good thing to do.

For simplicity, in this thesis we assume that components consist of just one entity. This allows us to say that a component *writes*, *reads*, *waits*, etc, instead of saying that one of the entities in the component performs these actions. For example, when we say that a component *waits* we mean that the entity performing the operation waits, if the component consist of more entities, the others need not wait.

Connections

Besides components, a system needs also *connections* among them. Examples of connections are **message passing**, **shared data spaces**, and **channels**[4]. The last one is the type of connection used in this thesis.

Message passing. Message passing is used in the object oriented world. With this type of connection, components send messages to each other. These messages need not be explicitly targeted; a component can send a message meant for any component having some kind of specific service, instead of sending it to a particular component. Nevertheless, this method has the disadvantage of a component having to know something about the structure of the system, even if it is implicit, in the sense that it has to know which services are provided by the other components.

Shared data spaces. In a shared data space, all components read and write values, usually tuples, from and to a shared space. The tuples contain data, together with some conditions. Any component satisfying these conditions can read a tuple; tuples are not explicitly targeted. This type of connection allows truly anonymous communication between components. However, the shared data space has some disadvantages. For example, it is not a primitive connection. A shared data space must be build using another primitives like message passing, and/or channels. Another disadvantage, is that in a distributed environment it may not be really efficient.

Channel. The last type of connection, and the one used in this thesis, is a channel. A channel is less restrictive than message passing, and it is really a primitive connection (unlike the shared data space). A channel consists of two ends, available to the components, the *source* and the *sink*. A component can write by inserting values to the *source*-end, and read by taking values from the *sink*-end of a channel. The communication is anonymous: the components do not know each other, just the channel-ends they have access to. Channels can be synchronous, asynchronous, mobile, with conditions. More about channels is explained in the next section (1.3).

Advantages

Why use component-based software? There are several advantages, all due to the fact that the implementation of a component is not relevant for the functionality of the system; only its interface is. The most important advantages are:

- **Easy building**. To build a system, it is enough to specify the interfaces of the components and the connections among them. Then, the components can be implemented, and/or reused from other systems, and/or bought from vendors.
  - **Reuse**. Components created for other systems can be reused, without trouble, for a new system when they have the same required interface. There is no need to implement them again.
  - **Buying components**. Components can be bought from vendors to save time. Many standard components can be bought these days. Also buying tailor made components can be a good idea when creating them is more costly.

- **Adaptability**. Components with new features can easily replace existing ones, making the system easy to adapt. For example, in the system of figure 1.1, component A could be replaced by a new component E that provides a better user interface with new features, without having to change major parts of the system.

- **Fast time-to-market**. Because of the advantages above, systems (and improved versions of the system) can be developed more rapidly. This is perhaps, the most important reason for companies to use components.

1.3 Mobile Channels

A channel, as described in section 1.2, is an important type of connection between components. In this section we explain more about channels. First, we give a general explanation, and after that we introduce mobile channels.

1.3.1 Channels

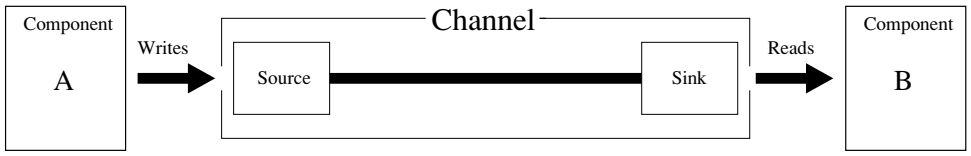


Figure 1.2: A Channel.

A channel consists of two ends, available to the components, the *source* and the *sink* (see figure 1.2). A component can write by inserting values to the *source*-end, and read by taking values from the *sink*-end. The communication is anonymous, the components do not know each other, just the channel-ends they have access to.

The typical actions that components can do on a channel are *write* and *read* values. If desired, a distinction between source and sink can be made like in [1], where if $c \in Channel$ then $\bar{c}$ denotes the source-end and c the sink-end. The operation for writing becomes $\bar{c}!v$, and for reading $c?v$, where $v \in Value$. $Values$ can be of any type, any data-structure can be send with a mobile channel.

In this thesis the channels are asynchronous and mobile. The first property allows a component to write without waiting for an acknowledgement. Because values are not read immediately, a unbounded buffer is needed to store data until read. The mobile aspect of the channel is explained in section 1.3.2.

1.3.2 Mobile Channels

Assuming that the connections of figure 1.1 are all channels, it would be nice to have a channel with the source-end at component D (database) and the sink-end moving between component A (user interface) and component B (translation). At this moment, component D has to know which channel-end it needs to use. It is better for the component to just have one channel(-end) and that a control-layer moves the sink-end of the channel to the right component each time. Such a channel is called *mobile*.

Channels are called *mobile*, when the channel-ends can be moved on to other components in the system. This allows dynamic reconfiguration of channel connections among the components of a system. Because the communication via a channel between two components is *anonymous*, when a channel-end moves, the component at the other channel-end does not notice anything.

Mobile Components

When components are in a distributed environment, they can be mobile; they can move from one location to another. This means that the structure of such a system changes dynamically during its lifetime. Laptops and mobile phones are examples of mobile components.

Importance of Mobile Channels

The mobility aspect of a channel is a property that is very useful, as seen in the example of figure 1.1. The ability to dynamically change the connections among the components yields great benefits.

In systems where components themselves are mobile, mobile channels become crucial. A system that changes dynamically needs channels that allow dynamic reconfiguration of connections among the components. Examples of such systems include software for mobile telephones and wearable computers, as well as for smart-card-based electronic commerce.

1.4 MoCha

Despite the importance of mobile channels (see section 1.3.2), up to now no (efficient) implementation for a general mobile channel has been presented in the literature (see section 1.5). Such an implementation is far from trivial if efficiency is required; it is difficult to develope an efficient implementation that minimizes the amount of non-local data-transfer.

In this thesis we present MoCha, a model for distributed *Mo*bile *Cha*nnels. MoCha describes an **efficient** and **non-trivial** implementation of an asynchronous mobile channel in a distributed environment by giving a set of functions. These functions are abstract algorithms covering all major operations on a channel: *Create_Channel*, *Write*, *Read*, *Move*, and *Destroy_Channel*. MoCha can easily be implemented in any modern language like Java and C++, that support data-structures, instances of data-structures, pointers/references, distributed networking, and threads.

I developed the MoCha model at CWI (centrum voor Wiskunde en Informatica) in a joint-work with: Farhad Arbab, Marcello Bonsangue, and Frank de Boer.

1.5 Literature

In this section we present previous work related to MoCha. Up to now no implementation of a mobile channel is given in the literature. However, there is some related work on channels and other types of connections among components (see section 1.2).
We discuss:

- *Java Message Queue*, a message passing type of connection for Java.
- *JavaSpaces*, a shared data space type of connection for Java.
- *Communicating Threads for Java*, a CSP-based model for Java. There is also a C and C++ version.
- *CSP for Java*, another CSP-based model for Java.
- *Pict*, a concurrent programming language based on the π-calculus.

For other interesting literature concerning languages for components using mobile channels, see the papers of some of the MoCha-team members, e.g., [1] [2].

1.5.1 JMQ

Java Message Queue (*JMQ*) enables the transmission of messages between application processes in a distributed environment. With JMQ processes running in different architectures and operating systems simply connect to the same *virtual network* (explained below) to send and receive information [6]. JMQ also handles all data translation between application processes. JMQ implements the Java Message Service (*JMS*) [5] open standard.

JMQ implements a *virtual fully connected network*, as oppose to a *fully connected network*. In the latter type of network, all processes are directly connected to all other processes in a system. Therefore, each process must have a list of all other processes and keep track of its connections with them. In a *virtual fully connected network* a separate process, called the **router**, is responsible for maintaining the connections and delivering the messages. All other processes have only a single connection each - each with its local Java Message Queue router.

JMQ supports two communication models: The *point-to-point model*, and the *publish-and-subscribe* model. In the first model, a sending process addresses the message to the queue that holds the messages for the intended receiving process. In the second model, a sending process addresses (publishes) the message to a **topic** to which multiple processes can be subscribed.

The communication in JMQ can be either asynchronous or synchronous. This is determined by the operation performed by the sending process.

Although JMQ provides a mean of communication among application processes instead of components, it can be said that it implements the **message passing** type of connection as explained in section 1.2. This type of connection is different than the **channels** of MoCha. The most important differences are:

- The communication of JMQ is not anonymous when using the *point to point* model.

- When using the *publish-and-subscribe* communication, a process still has to know something about the structure of the system in a indirect way; it needs to know what kind of **topics** are available.
- Whether the communication among processes is synchronous or asynchronous, is determined by the operation performed by a process - this operation is synchronous or asynchronous. With channels this is different; channels themselves are synchronous or asynchronous, not the operations performed on them - components do not even know what type of channel they are using.

More information about JMQ can be found in [6], and on its home page [7].

1.5.2 JavaSpaces

JavaSpaces technology is a powerful Jini service from Sun Microsystems. The JavaSpaces model involves persistent object exchange **spaces** in which remote processes can coordinate their actions and exchange data. *JavaSpaces* provides a simple unified mechanism for dynamic communication, coordination, and sharing of objects among Java technology-based network resources like clients and servers.

A space is a shared, network-accessible repository for objects. Processes use the repository as a persistent object storage and exchange mechanism; instead of communicating directly, they coordinate by exchanging objects through spaces. Processes perform simple operations to *write* new objects into a space, *take* objects from a space, or *read* (make a copy of) objects in a space. The persistent property of the space means that a collection of data remains intact even if its source is no longer attached to (or temporarily disconnected from) the network.

The JavaSpaces technology is a good step in the direction of component based-software. JavaSpaces provides the **Shared data space** type of connection as explained in section 1.2. This type of connection is useful in some component-based systems, for example; a blackboard system. However, in other systems the point-to-point connection provided by **channels** is more efficient. Therefore, a technology as JavaSpaces does not make the necessity of channels obsolete.

More information about JavaSpaces can be found in [8], and through its home page [9].

1.5.3 CTJ, CTC, CTC++

Communicating Threads for Java (*CTJ*) is based on the CSP paradigm [10]. In CSP, *Communicating Sequential Processes*, a system is described as a number of processes and channel connections among them. These processes operate independently (parallel) and communicate with each other using channels.

The CTJ package is an implementation of processes and channels in the Java language. There is also a C (CTC) and a C++ (CTC++) version by the same author[1]. The packages/libraries are developed to make parallel programming easier in these programming languages.

The channels of CTJ, (CTC, and CTC++), provide anonymous communication (between processes) and can be used in a distributed environment. These are the similarities with the channels described in this thesis. However, there are quite some differences:

- Channels are synchronous[2].
- Channels are not mobile.
- Channel use in distributed environments is indirectly supported. Channels are not primarily developed to work in a distributed environment.

More information about CTJ can be found in [11], and on its home page [12].

1.5.4 JCSP

CSP for Java (*JCSP*) is also based on the CSP paradigm [10], like CTJ (see section 1.5.3). The authors of JCSP argue that the *monitor-threads* model provided by Java, while easy to understand, proves very difficult to apply *safely* in any system above a modest level of complexity [14]. This makes parallel programming in Java very hard. However, parallel composition of CSP processes is easier to apply, is mathematically clean, yields no engineering surprises and scales well with system complexity.

JCSP is a Java class library providing a base range of CSP primitives plus a rich set of extensions; It provides processes and channels in Java. The

[1]The CTC and CTC++ versions run 100 times faster than the CTJ version.

[2]According to the paper [11], CTJ also provides buffered channels. However, we found no trace of asynchronous channels in the package. Therefore, we regard all channels in CTJ as synchronous.

channels provide anonymous communication (among processes) and can be used in a distributed environment. The differences with the channels in this thesis are actually the same as with the channels of CTJ (see section 1.5.3):

- Channels are synchronous[3].
- Channels are not mobile.
- Channel use in distributed environments is indirectly supported. Channels are not primarily developed to work in a distributed environment.

More information about CTJ can be found in [13], and through its home page [14].

1.5.5 Pict

Pict is a concurrent programming language based on the π-calculus. The π-calculus is a model for describing concurrent computation as systems of communicating agents (processes). In the computational world modeled by the π-calculus there are two entities: processes and mobile channels. Processes, sometimes called agents (as above), are the active components of a system; they interact by synchronous rendezvous on mobile channels, also called names or ports [15]. Pict includes a Pict-to-C compiler, reference manual, language tutorial, numerous libraries, and example programs.

The channels of Pict are quite different than the ones defined in this thesis. Both allow anonymous communication, but there are many differences:

- Channels are synchronous. Some form of asynchronous output is allowed, but the receiver must always send an explicit acknowledgement back to the sender.
- Channels are not mobile as described in this thesis. Conceptually, the mobility of channels in the π-calculus and MoCha is the same. However, the implementation in Pict is different. In Pict channels are mobile, because channels identities can be send through a channel to other processes. A process that receives a channel identity can connect to it. This is different from the mobile channel of MoCha, where a channel has channel-ends that really move from one location to another.

[3]JCSP also provides buffered channels. However, we found no trace of asynchronous channels in the package. Therefore, we regard all channels in JCSP as synchronous.

- Pict does not support distributed environments.

More information about Pict can be found in [15] [16], and through its home page [17].

Chapter 2

MoCha, An Explanation

Before presenting MoCha (see chapter 4), in this chapter we first present an (intuitive) explanation using text and figures instead of algorithms. This explanation gives a better understanding of the MoCha model. The explanation uses several figures, figure 2.1 shows the legend.

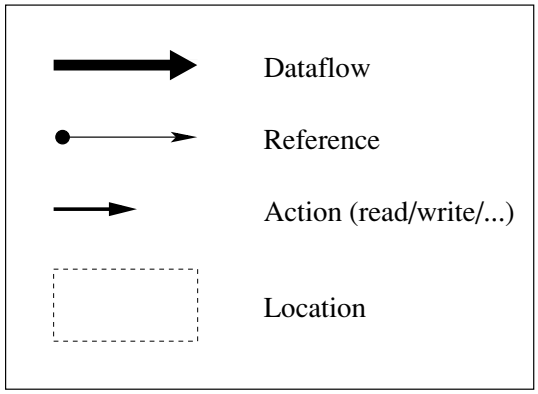


Figure 2.1: Legend.

In the next section the general properties and some assumptions about the mobile channels are given. After that the explanation of MoCha operations begins, starting with the creation of a channel, then writing and reading without mobility, mobility, and ending with more explanation about reading.

2.1 Properties and Assumptions

In this section the general properties and assumptions concerning the mobile channels are given.

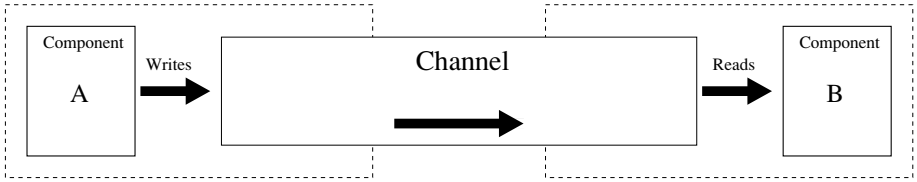


Figure 2.2: A channel.

As mentioned in the last chapter, the components are in a distributed environment and they communicate through mobile channels. In figure 2.2 a general view of a channel is given. The components write and read *values* to/from the channel. *Values* can be of any type. Any data-structure can be send with a mobile channel.

The figure (2.2) shows that the data-flow of the channels is just *one way*, one end is always for reading and the other end for writing.

Another property of a channel in MoCha is that it has a loosely FIFO-structure. We assume that data can get lost or deleted due to working with values with expiring dates. However, for simplicity, in this chapter we assume that a channel has a normal *FIFO-structure*.

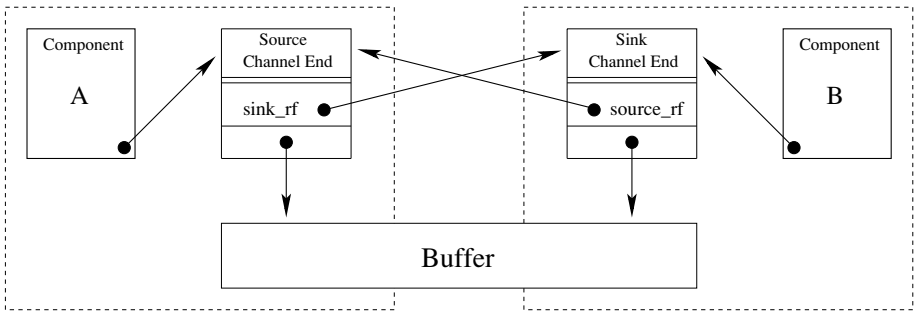


Figure 2.3: A channel (refined).

Figure 2.3 gives a refined view of a mobile channel. For the components the channel consists of two ends (which are available to them), the *source* and the *sink* (see figure 2.3). A component writes values into the channel by

accessing its *source*-end and reads from it by accessing its *sink*-end.

The communication by a mobile channel is **anonymous**. The components do not have to know each other, just the channel-ends. Therefore, a channel-end can be moved from one component to another without the knowledge of the component at the other side of the channel. However, the channel-ends (of a channel) do have to know each other for keeping the identity of the channel, and control communication. For this purpose, the channel-ends have a reference to each other (the *sink_rf*- and *source_rf*-fields in figure 2.3).

The communication, besides being anonymous, is also **asynchronous**. This allows the writer to write without waiting for an acknowledgement of the reader, and the reader to read when values are available and it desires. Because reading is on demand a (unbounded) buffer is needed to store data until read. That is why in figure 2.3 the channel-ends refer to a buffer. The implementation of this buffer consists of local buffers that are distributed over the system, this is made clear and explained later on.

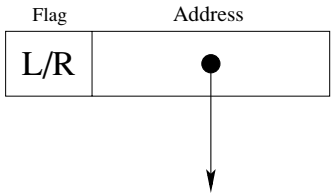


Figure 2.4: A reference.

The references used by MoCha either point to a local or to a remote entity. In figure 2.3, examples of both cases are given. A reference has a structure given in figure 2.4. It is a pointer with a flag that indicates whether it is local (L) or remote (R), so that the algorithms can test on this flag. The address-part of a reference not only specifies the memory-address of the entity but also its location. For the definition of location in this model, it is sufficient to say that it is a *logical address space* where components and other entities run. The mapping of locations to physical implementations is irrelevant at this level of abstraction[1].

We restrict components having a reference to a channel-end, by not being able to access the fields of that particular channel-end, only perform pre-

[1]In JAVA [18], for example, a location is a virtual machine.

defined operations on it. Therefore, components can not access any other data-structures of the channel than channel-ends.

For simplicity, we assume that components read and write only single values. It is easy to expand this to more values if needed.

Also to simplify our explanation in this chapter, we assume that only one component can have access to a particular channel-end at a given time (as in figure 2.3), and that this channel-end must be local to that component. MoCha itself is more general, leaving this *one-to-one restriction* to an upper level to enforce, if necessary (see chapter 3).

2.2 Creation of Channels

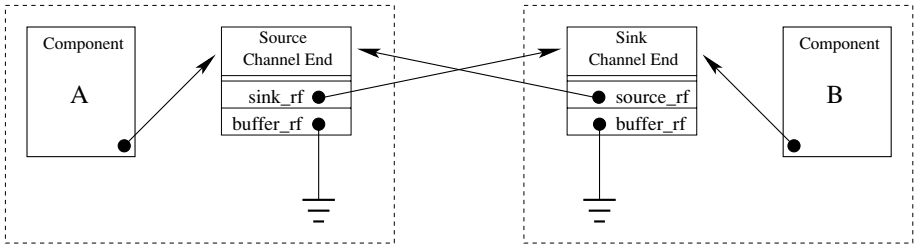


Figure 2.5: Channel Creation

Upon creation of a new channel, its channel-ends (its *sink* and its *source*) are created at two given locations, which need not to be distinct. A reference to each channel-end is then given to the component that, at creation, must have access to it[2]. From now on having access to a channel-end is, sometimes, also referred to as *holding* it (a component holds the channel-end).

An example of creation is given in figure 2.5. In this figure, components A and B need not be distinct, because the same component can hold both channel-ends. At creation, the channel-end fields *sink_rf* and *source_rf* are set in the proper way (pointing to the other channel-end), and the *buffer_rf*-fields, a field that points to a buffer, are set to **NULL**. The reason for doing this is that there is no guarantee that the component, at which the channel-end is placed, really starts to write or to read. The channel-end can be

[2]MoCha just gives the references to the creator of the channel, leaving it with the responsibility of distributing the references to the channel-ends.

moved to another component, without the component holding it now having actually read or written any data. Therefore creating a buffer is only done when really needed, which is more efficient (examples will follow later on).

2.3 Writing and Reading (Without Mobility)

After creating a channel, the components holding the channel-ends can use them. In this section, we explain writing and reading under the assumption that the channel-ends **do not move**. Writing and reading with mobility is explained in the next section, where at the end a general summary of these actions is given (see 2.4.3).

There is an ***invariant*** on the *buffer_rf*-fields of the channel-ends: They must be both **NULL** or both **non-NULL**. In figure 2.5 the channel is just created, the *buffer_rf*-fields are, initially, **NULL**. Therefore, after the creation of the first buffer of the channel, they must be both **non-NULL** (example follows).

Writing

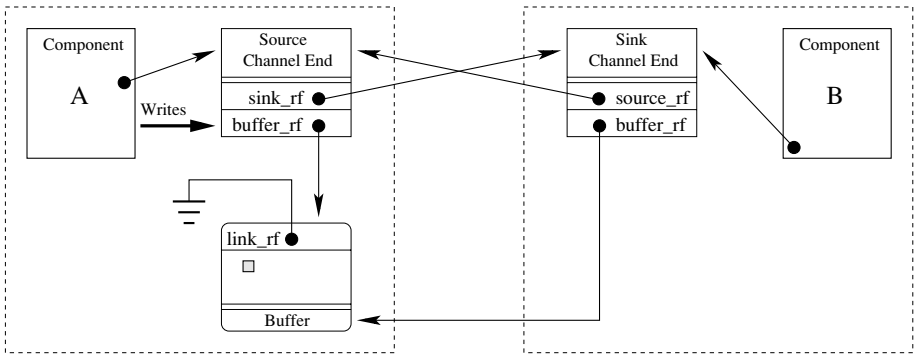


Figure 2.6: First Write Action.

In figure 2.6, component A starts to write for the first time. Because its *buffer_rf*-field is **NULL** the source creates a local buffer. This is an *unbounded* FIFO buffer that has more functionality than a normal FIFO buffer (this is explained later on, see section 2.5). The buffer has a field called *link_rf*, this is a reference to another buffer. There is just one buffer now, therefore the field is set to **NULL**. Because the *buffer_rf*-field of the source was **NULL**, the source notifies the sink about the new buffer. The sink updates its *buffer_rf*-field to point to this new buffer in order to satisfy the

invariant.

The first buffer of a channel is always created by its source-end. After the first write, there is always a local buffer. Future writes will insert their elements into the existing buffer[3].

Reading

In figure 2.5 both *buffer_rf*-fields are **NULL**. If component B wants to read, the sink-end will notice its *buffer_rf*-field is **NULL** and conclude that there are no elements in the channel. It then waits for the first write.

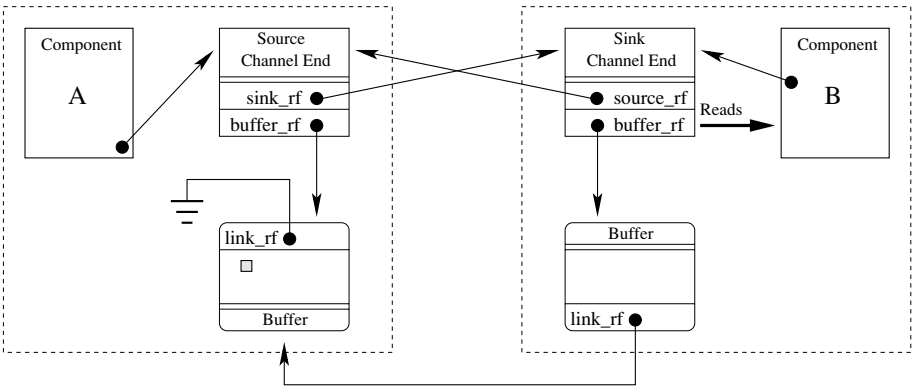


Figure 2.7: First Read Action.

In figure 2.6 the *buffer_rf*-field of the sink is not **NULL** anymore, but it is pointing to a non-local buffer. If component B attempts to read now the sink creates a new local buffer. The *link_rf*-field of this buffer is set to point to the old buffer (where the *buffer_rf*-field of the sink is pointing to), and the *buffer_rf*-field is updated to point to the new local buffer. The result is figure 2.7. If components A and B are equal, then no new local buffer is made, because the *buffer_rf*-field is already pointing to a local one.

The actual reading of the elements can now begin (this is explained in section 2.5). After the first read there is always a local buffer. Future reads will read elements from this buffer[4].

[3]Assuming that the source does **not** move.

[4]Assuming that the sink does **not** move.

2.4 Mobility

Channel-ends are mobile, they can be passed on to other components in the system[5]. This allows dynamic reconfiguration of channel connections among the components of a system. Because the communication via a channel between components is *anonymous*, when a channel-end moves, the component at the other channel-end does not notice anything.

In this section mobility and mobility in combination with writing and reading are explained. Next, we give a summary, and finally we explain the chain of buffers, that can result from mobility in combination with reading and writing.

2.4.1 Source and Writing

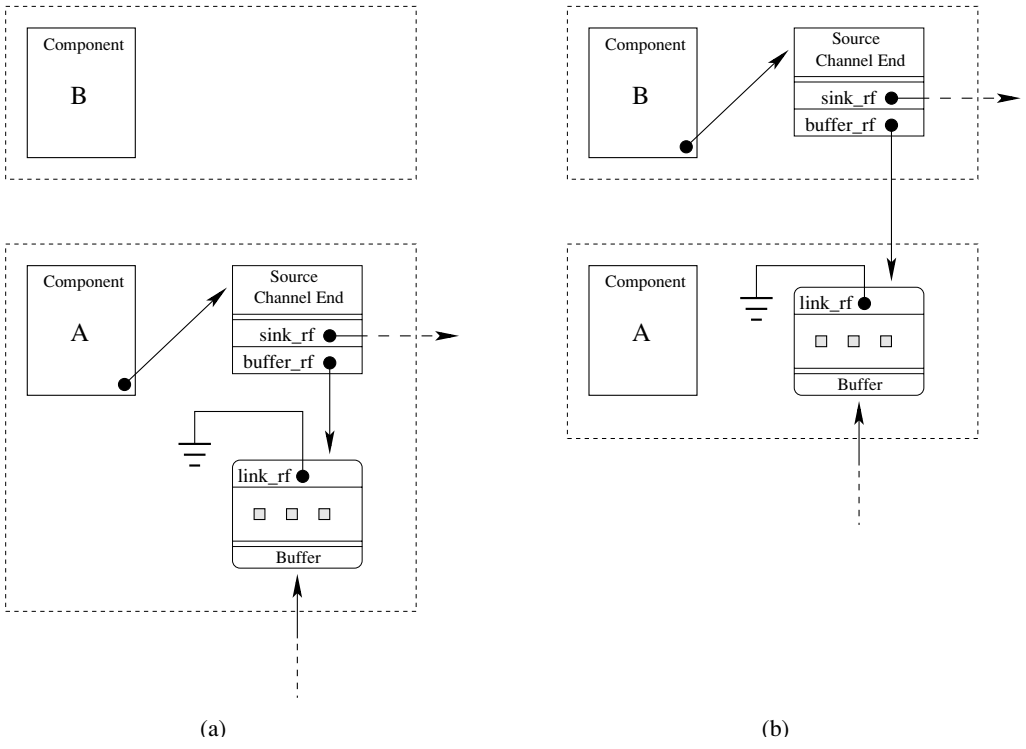


Figure 2.8: Mobility of Source Channel-End.

[5]MoCha allows any component that has a correct reference to the channel-end, to move this channel-end.

We explain the mobility of a source channel-end and writing using two figures (figures 2.8, and 2.9). In figure 2.8, component A holds the source (a) [6]. At some point in time the channel-end is given to component B (b) (component B does not write yet). The buffer with elements stays at component A, and does not move with the channel-end. The only thing that changes is the *buffer_rf*-reference of the source, from local to non-local. Also, no new buffer is created at the location of component B (**until** it actually starts to write), because the channel-end can move again without component B having written anything. After the source moves, it notifies the sink about its new location.

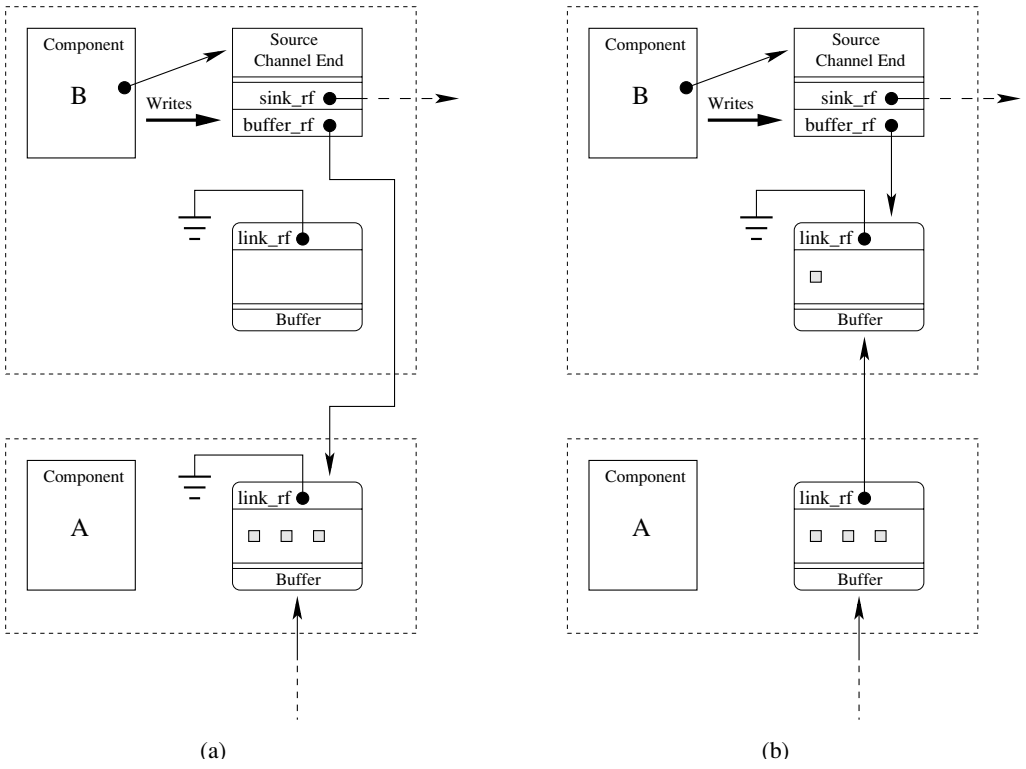


Figure 2.9: Adding a new Buffer to the Chain at the Source-end.

Instead of the source moving to another component, component B starts to write in figure 2.9(a). The *buffer_rf*-reference of the source is non-local, therefore a new local buffer is created. Both the *buffer_rf*-field, and the

[6] One dashed reference arrow expresses that something is pointing to the buffer; this is either another buffer or the sink channel-end. The other one expresses that the *sink_rf*-field is pointing to the sink channel-end.

link_rf-field of the old buffer (the buffer where the *buffer_rf*-field is pointing to) are changed by setting them to point to the new buffer. The result is figure 2.9(b). The elements can now be inserted into this new local buffer.

2.4.2 Sink and Reading

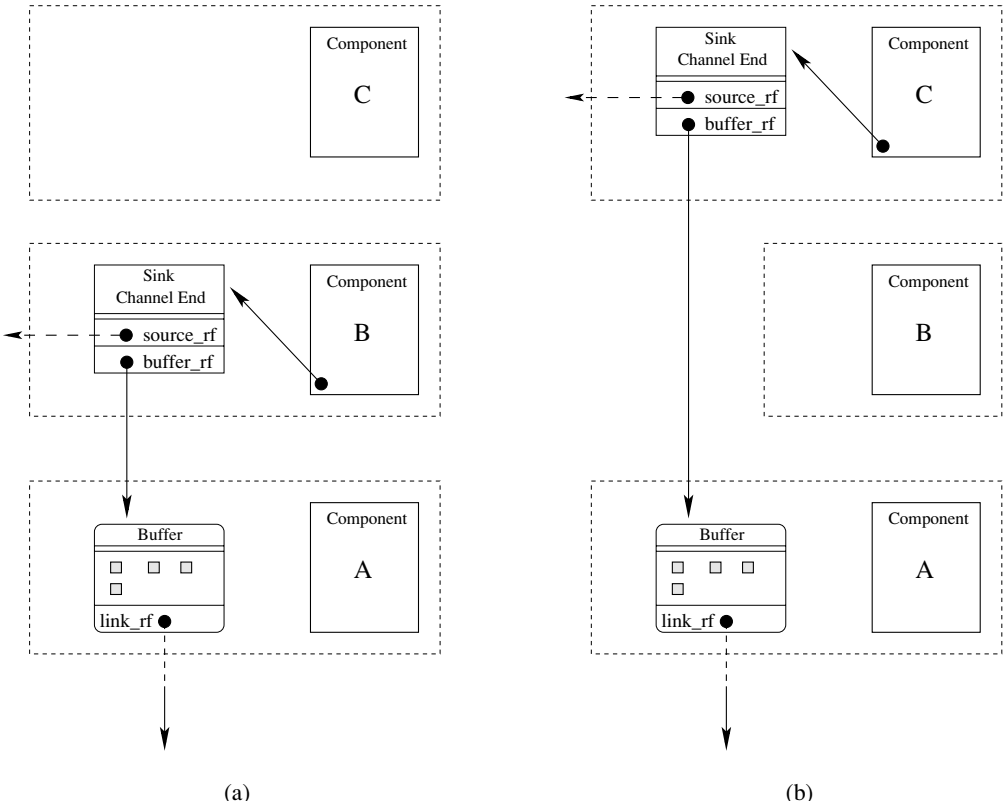


Figure 2.10: Mobility of Sink Channel-end

We explain the mobility of the sink channel-end and reading by using figures 2.10 and 2.11. In figure 2.10(a) component B holds the sink and it has never read anything (there is no local buffer). At some point in time, this sink channel-end is given to component C (b). Nothing changes, the *buffer_rf*-field still points to a non-local buffer and no local buffer is created **until** component C actually starts to read. After the sink moves, it notifies the source about its new location.

In figure 2.11(a) component C starts to read. The *buffer_rf*-reference of the sink is non-local, therefore a new local buffer is created. The *link_rf*-field of

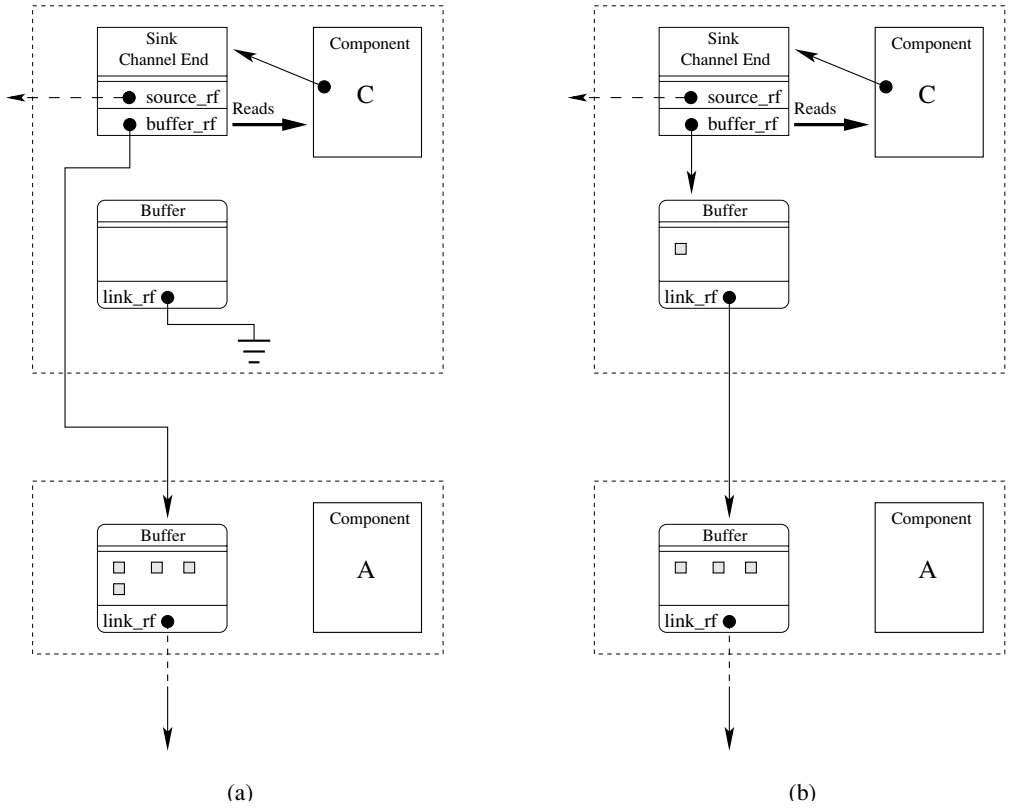


Figure 2.11: Adding a new Buffer to the Chain at the Sink-end.

the new buffer is changed by setting it to point to the old buffer (the buffer where the *buffer_rf*-field is pointing to), and the *buffer_rf*-field is changed by setting it to point to the new buffer. The result is figure 2.11(b)[7]. The reading can now begin (more details on reading is given in section 2.5).

2.4.3 Summary

Before continuing with our explanation, we give a summary of the movement of the channel-ends and the actions write and read, in general.

About the movement of the channel-ends:

- The *buffer_rf*-field reference of a channel-end can change from local to non-local. This is the case when the channel-end moves to another loca-

[7]This figure is just an example, if the old buffer is empty, then the end result is different. The destructive aspect of reading is explained in section 2.5.3.

tion. When moving to a component in the same location the reference remains local.

- After moving, the channel-end notifies the other end about its new location.
- Only the channel-ends move, the buffers do not.
- Buffers are only created when really needed (for reading or writing).

About writing and reading:

- When writing, a local buffer is created at the location of the source if its *buffer_rf*-field is either **NULL** or pointing to a non-local buffer. If the *buffer_rf*-field is **NULL**, the source notifies the sink about the newly created local buffer.
- When reading, a local buffer is created at the location of the sink if its *buffer_rf*-field is pointing to a non-local buffer. If the *buffer_rf*-field is pointing to a local buffer, then reading can proceed. If it is **NULL**, then no elements exist in the channel and the sink waits.

2.4.4 Chain of Buffers

The mobility of channel-ends in combination with the actions write and read can lead to a **chain** of buffers that is distributed over many locations. Figure 2.12 shows a general case. The components need not be distinct (although they have different labels), and the *buffer_rf*-field references need not be local. The *first* buffer of the chain is the one where the *buffer_rf*-field of the sink points to, it contains the first elements inserted in the channel. The *last* buffer is the one where the *buffer_rf*-field of the source points to, it contains the last elements inserted in the channel. In this way the FIFO structure of the channel is preserved (the buffers have a FIFO structure).

Figure 2.13 shows an instance of the general case (figure 2.12). There are two buffers at the location of component B, the source moved twice between the locations of components B and A. Component B has no access or knowledge of the buffers (only channel-ends can access the buffers). At the location of component F, there are also two buffers. This situation can exist only if a buffer, between the two now present, has been deleted (see section 2.5.3 for an explanation about destruction of buffers).

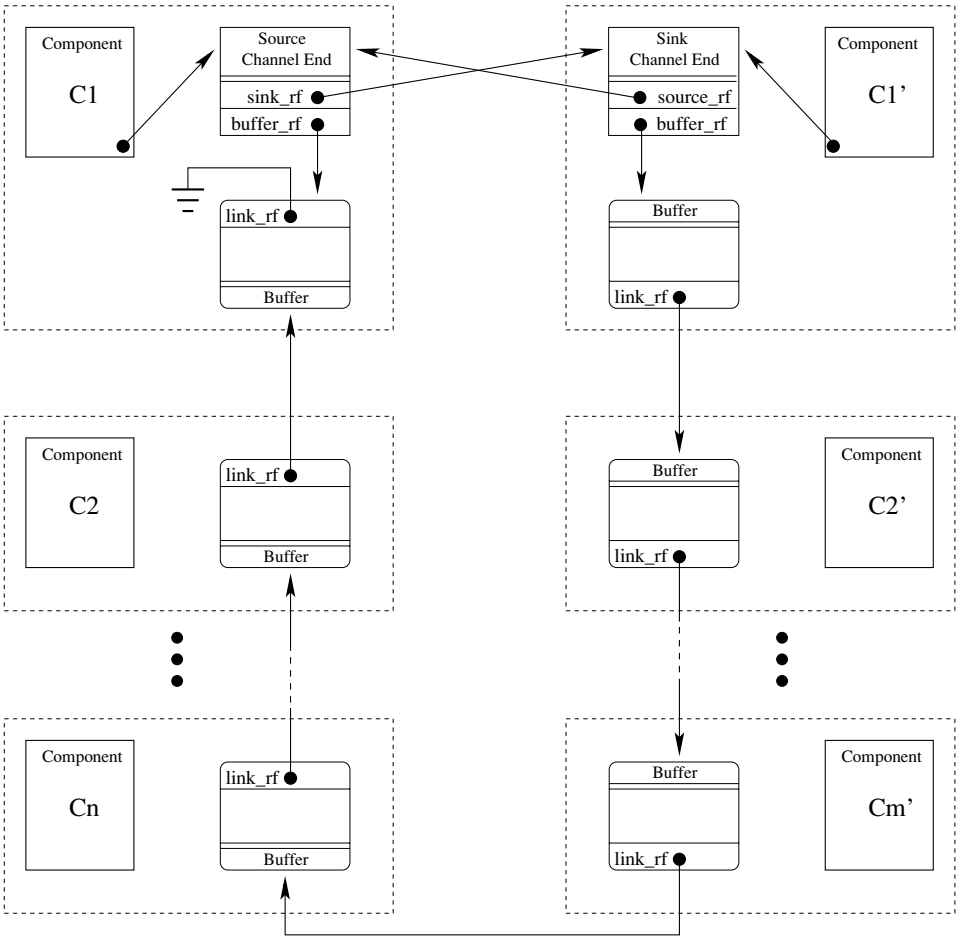


Figure 2.12: Chain of Buffers.

2.5 Reading

Up to now it is clear how buffers are created both by reading and writing in combination with (and without) the mobility of the channel-ends. For writing this is sufficient because this action just creates local buffers, sets ups the references, and fills them with elements. Reading, however, needs more explanation.

When reading, elements move through the chain of buffers. Reading is also destructive; meaning that the read values/elements are eliminated from the channel. Therefore, buffers get empty and get destroyed. In this section we explain: the extra functionality of the FIFO buffers, reading heuristics used

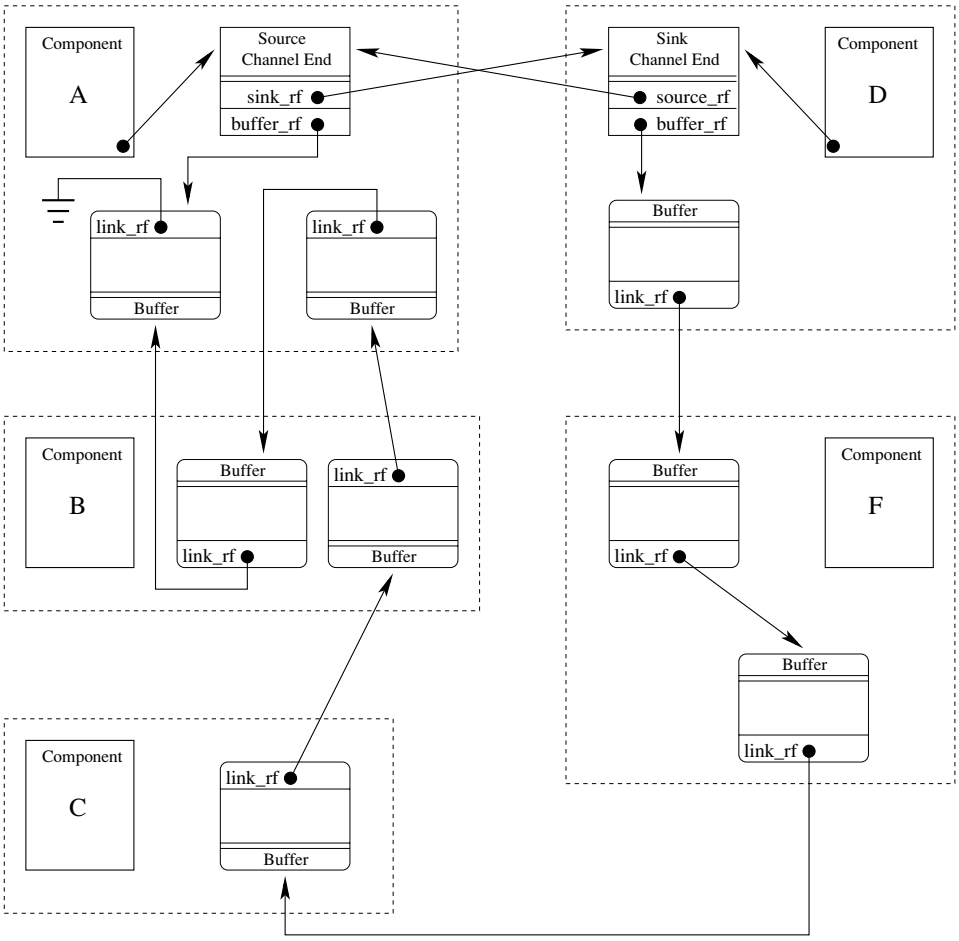


Figure 2.13: An instance of the General Case.

for moving the elements, and the destruction of empty buffers.

2.5.1 Smart Buffers

The buffers are *smart* FIFO buffers because they have more functionality than a normal one. The functionality of the buffers can be divided into two groups: a buffer is either the *sink-buffer*, or it is not. The *sink-buffer* is the **local** buffer where the *buffer_rf*-field of the sink is pointing to. For each channel, only one buffer can be the *sink-buffer* at a given time, but it can change; Any buffer can become the *sink-buffer* (depending on the behavior of the system). There does not have to be a *sink-buffer* all time. This is important in combination with reading.

When reading, the sink asks the *sink-buffer* for an element. If the *sink-buffer* contains at least one element, it gives an element to the sink (normal FIFO behavior). If it is empty, it asks the next buffer in the chain (if any) for a certain amount of elements. This amount is determined by a reading heuristic explained in the next subsection. The *sink-buffer* receives from this next buffer either:

- The requested number, less, or zero elements together with a reference to the next buffer in the chain. This indicates that the buffer is now (or was) empty. The *sink-buffer* changes its *link_rf*-field to this new buffer.
- The requested number, less, or zero elements together with a reference that is **NULL**. This indicates that the buffer is not empty after giving the elements, or it is the last buffer of the chain. The *sink-buffer* **does not** change its *link_rf*-field.

Examples of the behavior of the buffers appear in the next subsections.

2.5.2 Reading Heuristics

For efficiency, the sink-buffer uses heuristics for the number of elements it requests from the next buffer in the chain. For one of the heuristic, the sink channel-end has an extra field called *consumed*, it is a variable that keeps track of how many elements the current component holding the sink has already consumed. Initially, and every time the channel-end moves, the variable is set to zero.

Knowing nothing of the behavior of the system in general, it seems reasonable to base a reading heuristic on the amount of elements, the component holding the sink, already has consumed. A good heuristic, that is also used in memory management, is to assume that the component is going to consume (in total) **twice** the amount it already has consumed. Therefore the amount of elements to be requested should be *consumed*, but to make MoCha simpler the sink-buffer requests ***consumed* + 1** elements.

Figure 2.14 shows an example. Component C is holding the sink channel-end and has already consumed 3 elements. It then asks for another element. The sink-buffer is empty, therefore it requests elements from the next buffer in the chain. We assume that the component is going to consume a total of 7 elements (twice + 1), therefore the sink-buffer requests 4 elements (consumed + 1). In figure 2.14(b) the 4 elements of the non-local buffer are transfered

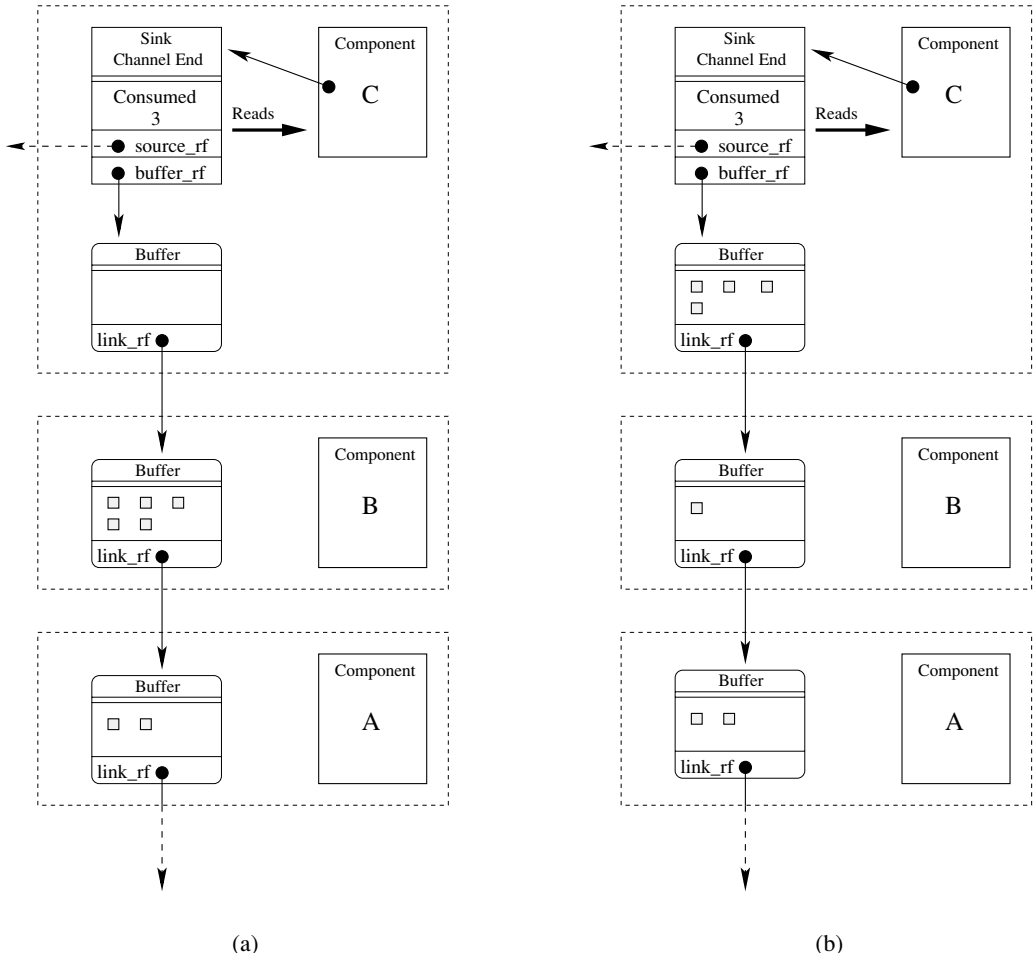


Figure 2.14: Heuristic of Sink Buffer.

to the sink-buffer. An element is then given to the component (by the sink), this is not shown in the figure.

Why use this heuristic instead of letting the sink-buffer request all the elements from the next buffer? The answer is that because the sink channel-end can move anytime, there is no point in moving extra unnecessary elements; this can be very costly due to the fact that nothing is known about the future behaviour of the system.

For the same reason, *consumed + 1* is an **upper bound** for the elements to be supplied. The sink-buffer settles for less (but more than zero). In figure 2.15, component C already consumed the 4 elements from the last figure

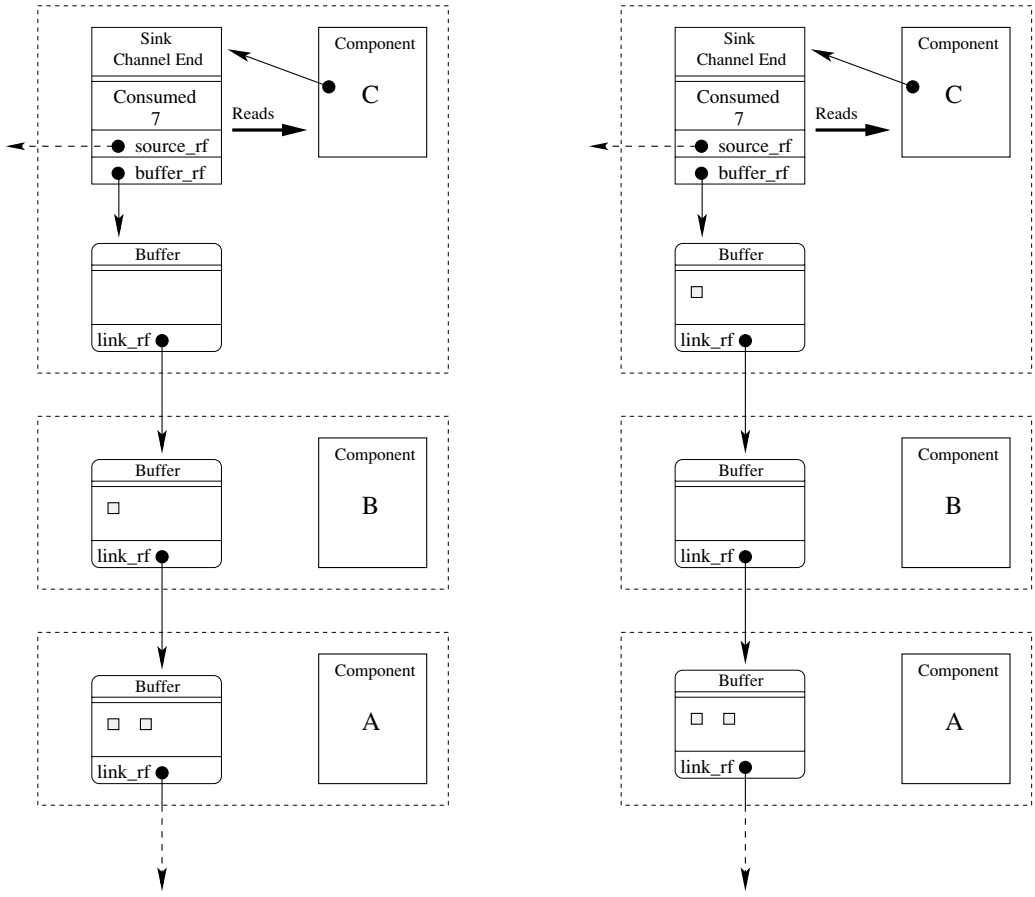


Figure 2.15: Heuristic of Sink Buffer, Upper Bound.

(2.14) making it a total of 7 elements, and leaving the sink-buffer empty again. It then asks for a new element and the sink-buffer requests 8 elements (consumed + 1). Unfortunately the next buffer has just one element, but the sink-buffer settles for this one, it does not try to get the remaining elements from another buffer.

There is another upper bound concerning the amount of elements that is given to the sink-buffer. When transporting one element during a remote call, there is usually space left on the message/frame for transporting more elements for the same cost. Naturally, there is also a limit on the number of elements that can be transported during a remote call for the same cost. This number is also an upper bound, making sure that the cost of requesting

X elements is the same as requesting one.

2.5.3 Destruction of Buffers

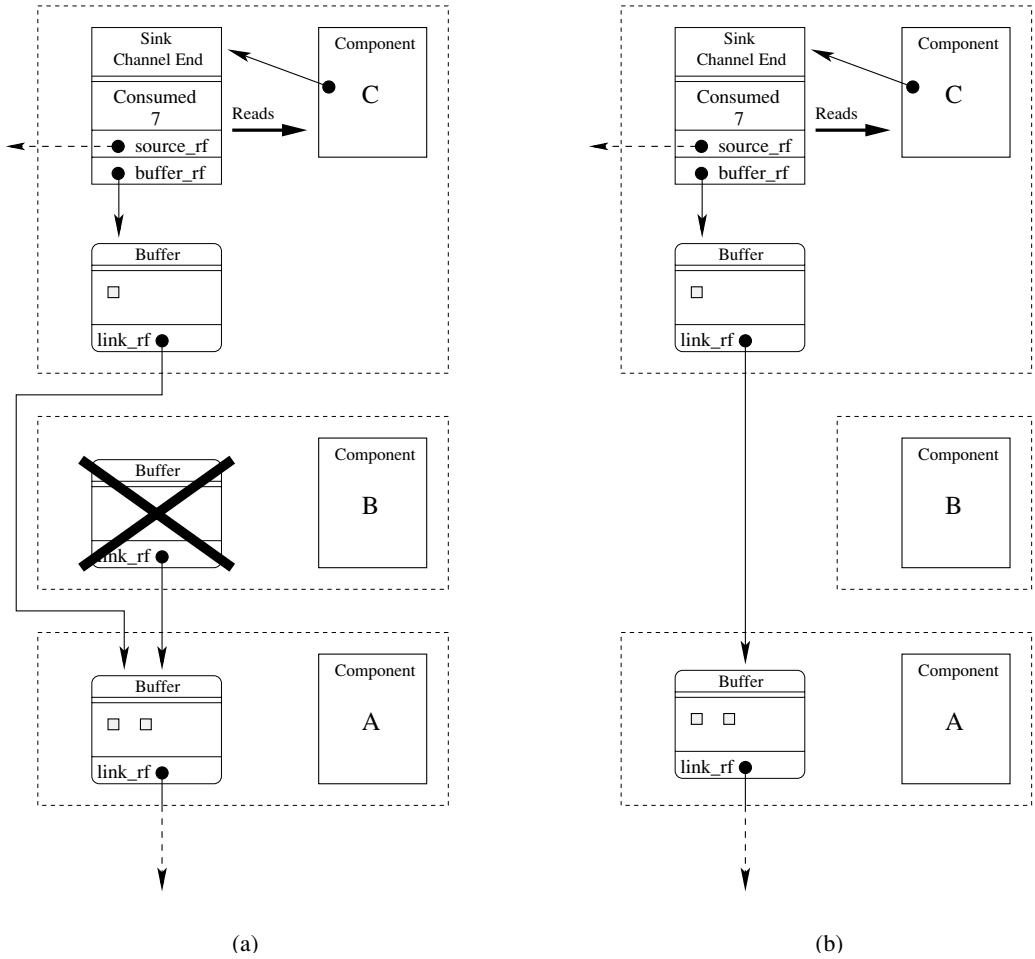


Figure 2.16: Destruction of Buffers.

A buffer can get empty when elements are read out of it[8]. An empty buffer is no longer of use, unless it is the first or last buffer of the chain. Therefore an empty buffer destroys itself when it gives its *link_rf*-reference (the reference to the next buffer in the chain) to the sink-buffer, unless it is the last buffer

[8]Moreover, a buffer can also get empty when working with data with expiration dates, when data gets lost, or a combination of the three. We give more details about this in chapter 4.

of the channel (in which case its *link_rf*-reference is **NULL**). The behavior of the buffers is described in section 2.5.1.

In figure 2.15 a buffer got empty due to reading. Its destruction is described in figure 2.16. The buffer, being empty (due to the request), gives the sink-buffer not only the element but also a reference to the next buffer in the chain. The sink-buffer updates its *link_rf*-field to this next buffer. The empty buffer, after giving a valid reference to the sink-buffer, then concludes that it can safely destroy itself because it is no longer part of the chain.

Chapter 3

MoCha, Simplified Version

After the intuitive explanation of how MoCha works in the previous chapter, it is time to present MoCha. However, due to its complexity, and for better understanding, in this chapter we first present a simplified version of MoCha.

In the simplified version, the movement of the channel-ends is not taken into account. Although a chain of buffers can exist due the movement of the channel-ends, the functions do not take into account that the channel-ends actually can move. MoCha is simplified by not having extra semaphores for the movement of the channel-ends, error-handling, and other operations where movement must be considered. The full version is presented in the next chapter. It is worthwhile to read this chapter first as the next chapter refers to this one many times.

3.1 Data-structures

MoCha uses three data-structures: *Source_Channel_End*, *Sink_Channel_End*, and *Buffer*. In the complete version, (see next chapter), the source and sink channel-ends have more fields than described here.

Source_Channel_End = <Buffer_rf, Sink_rf, CE_Lock>

- Buffer_rf = A reference to a Buffer data-structure.
- Sink_rf = A reference to the Sink_Channel_End of the same channel.
- CE_Lock = A binary semaphore. It is used for mutual exclusion of the Channel-End.

Sink_Channel_End = <Buffer_rf, Source_rf, Consumed, CE_Lock>

- Buffer_rf = A reference to a Buffer data-structure.
- Source_rf = A reference to the Source_Channel_End of the same channel.
- Consumed = An integer that keeps track of the number of values consumed by the component(s) since the last move.
- CE_Lock = A binary semaphore. It is used for mutual exclusion of the Channel-End.

Buffer = <Vals, Link_rf, Buffer_Lock>

- Vals = A sequence of Value, initially empty.
- Link_rf = A reference to a Buffer data-structure, initially **NULL**.
- Buffer_Lock = A binary semaphore. It is used for mutual exclusion of the buffer.

The binary semaphores, (CE_Lock and Buffer_Lock), have two operations: *Lock* and *Unlock*. Initially all semaphores are unlocked. We assume that the waiting-queue of a semaphore, where the blocked processes wait, has a FIFO-structure[1].

3.2 Mutual Exclusion

In the explanation of chapter 2, only one component was assumed to have access to a particular channel-end at a time. MoCha is more general, because every component having a **valid** reference to the channel-end can use it[2], leaving the *one-to-one restriction* to an upper level to impose if necessary. For this reason, a general *lock* is needed on each channel-end, to ensure that only one action (read, write, or move) is being carried out at a time on the same channel-end. This general lock is the **CE_Lock** semaphore.

[1]This is not necessary for the protocol, but it is assumed for fairness of the system.

[2]The case where a component has an invalid reference to a channel-end and tries to use it, is explained in the next chapter. See section 4.3.

3.3 Data-structures and References

We use of a $\rightarrow$ notation, in the algorithms of MoCha, to express that something is a reference to a particular data-structure type. Thus, $\rightarrow$`data-structure` is the set of all references to this data-structure type. For example; `Next_rf` $\in$ $\rightarrow$`Buffer`, `Next_rf` is a reference to a `Buffer` data-structure type[3].

Another notation is used for expressing the operation that takes a reference and gives the instance of the data-structure it refers to. For this operation we use the symbol $\uparrow$. For example; `Next_rf`$\uparrow$`.Vals := < >`, assigns the empty sequence to the *Vals*-field of the `Buffer` data-structure, that `Next_rf` refers to[4].

3.4 Functions, an Overview

A channel can be accessed using five functions that operate on the Channel-end data-structures: Create_Channel, Write, Read, Move, and Destroy_Channel. In this simplified version of MoCha, we discuss only the three first, and leave the other two for chapter 4.

- **Create_Channel** (*Loc1* $\in$ Location, *Loc2* $\in$ Location)
  returns <*Source_rf* $\in$ $\rightarrow$Source_Channel_End, *Sink_rf* $\in$ $\rightarrow$Sink_Channel_End>
- **Write** (*Source_rf* $\in$ $\rightarrow$Source_Channel_End, $X \in$ Value)
- **Read** (*Sink_rf* $\in$ $\rightarrow$Sink_Channel_End)
  returns <$X \in$ Value>

These three functions use the following functions to operate on the buffer data-structure:

- **Write** (*Buffer_rf* $\in$ $\rightarrow$Buffer, $X \in$ Value)
- **Sink_Read** (*Buffer_rf* $\in$ $\rightarrow$Buffer, *ToBeAsked* $\in$ Integer)
  returns <$X \in$ Value, *Success* $\in \{\text{☺}, \text{☹}\}$ >
- **Buffer_Read** (*Buffer_rf* $\in$ $\rightarrow$Buffer, *Amount* $\in$ Integer)
  returns <$V \in Value^*$, *Next_Buffer_rf* $\in$ $\rightarrow$Buffer>

[3]The reference needs not be valid, i.e., refer to an existing instance of the data-structure.

[4]The Algorithms of MoCha must ensure that the reference is valid before executing this operation.

3.5 Creation of a Channel

To create a channel the following function is used:

```
Create_Channel (Loc1 , Loc2)
//
// Loc1 = The location where to create the Source_Channel_End.
// Loc2 = The location where to create the Sink_Channel_End.
//
// returns <Source_rf ∈ →Source_Channel_End,
// Sink_rf ∈ →Sink_Channel_End>
//
begin
   Source_rf := new Source_Channel_End @ Loc1;
   Sink_rf := new Sink_Channel_End @ Loc2;

   Source_rf↑.Buffer_rf := NULL;
   Source_rf↑.Wait_Read := FALSE;
   Sink_rf↑.Buffer_rf := NULL;
   Sink_rf↑.Consumed := 0;

   // Knowing each other.
   //
   Source_rf↑.Sink_rf := Sink_rf;
   Sink_rf↑.Source_rf := Source_rf;

   return <Source_rf, Sink_rf>;
end
```

This function creates the channel-ends at the given locations, initializes their fields, and returns to the creator of the channel a reference to each channel-end.

3.6 Writing

A component that has a valid reference to a source channel-end can write to it using the following function:

```
Write (Source_rf, X)
//
// Source_rf ∈ →Source_Channel_End, X ∈ Value.
//
begin
   Lock(Source_rf↑.CE_Lock);

   if ( Source_rf↑.Buffer_rf = NULL ) then
      // Create first buffer of channel.
      //
      Source_rf↑.Buffer_rf := new Buffer;
      Source_rf↑.Buffer_rf↑.Link_rf := NULL;

      // Tell the other side.
      //
      Source_rf↑.Sink_rf↑.Buffer_rf := Source_rf↑.Buffer_rf;
   else
      if ( ¬Local(Source_rf↑.Buffer_rf) ) then
         // Create new local buffer
         //
         TempBuffer_rf := new Buffer;
         TempBuffer_rf↑.Link_rf := NULL;
         Source_rf↑.Buffer_rf↑.Link_rf := TempBuffer_rf;
         Source_rf↑.Buffer_rf := TempBuffer_rf;
      fi
   fi

   // At this point there is always a local buffer.
   //
   Write(Source_rf↑.Buffer_rf, X);

   Unlock(Source_rf↑.CE_Lock);
end
```

This function creates a local buffer (at the location of the source) if the *buffer_rf*-field of the source is either **NULL** or pointing to a non-local buffer.

If the *buffer_rf*-field is **NULL**, by the ***invariant*** that the *buffer_rf* fields of the channel-ends of a channel must either both be **NULL** or both be **non-NULL**, it concludes that the *buffer_rf*-field of the sink is also **NULL**, and sets it to point to this new buffer. Because the simplified version does not take into account the moving of the channel-ends, no locking is needed when setting this field of the sink.

To write the value in the local buffer the following function is used:

```
Write (Buffer_rf, X)
//
// Buffer_rf ∈ →Buffer, X ∈ Value.
//
// This function operates on the Buffer data-structure.
//
begin
   Lock(Buffer_rf↑.Buffer_Lock);

   Buffer_rf↑.Vals := Buffer_rf↑.Vals ∘ X;

   Unlock(Buffer_rf↑.Buffer_Lock);
end
```

More explanation about writing can be found in chapter 2 (sections 2.3, and 2.4.3).

3.7 Reading

A component that has a valid reference to a sink channel-end can read using the function *Read*. This function, which operates on the Sink_Channel_End data-structure, then calls the function *Sink_Read*, which operates on the Buffer data-structure. This last function may call the function *Buffer_Read* (if necessary), which also operates on the Buffer data-structure, if the *sink-buffer* is empty; See chapter 2 (section 2.5). In figure 3.1, a scheme of the functions is given[5]; to avoid confusion, no references are given.

This section is divided in subsections to cover each function. At the end, there is a subsection about the creation of empty buffers by writing and reading in combination with mobility. More explanation about reading can

[5]The second buffer can be local in some situations; see figure 2.13.

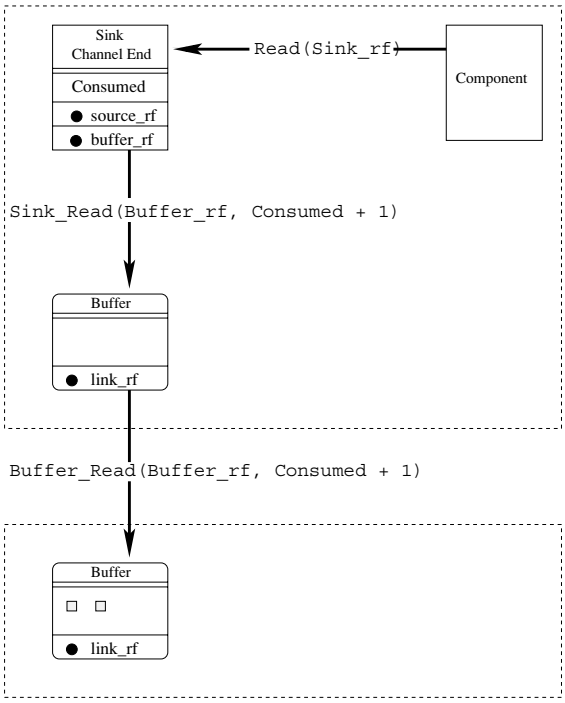


Figure 3.1: Read Functions Scheme.

be found in chapter 2 (sections 2.3, 2.4.3, and 2.5).

3.7.1 Read

A component that has a valid reference to a sink channel-end can read using the following function:

```
Read (Sink_rf)
//
// Sink_rf ∈ →Sink_Channel_End.
//
// returns <X ∈ Value>
//
begin
   Lock(Sink_rf↑.CE_Lock);

   Success ∈ {☹,☺} := ☹;
```

```
   while ( Success ≠ ☺ ) do
      if ( Sink_rf↑.Buffer_rf ≠ NULL ) then
         if ( ¬Local(Sink_rf↑.Buffer_rf) ) then
            // New local Buffer.
            //
            TempBuffer_rf := new Buffer;
            TempBuffer_rf↑.Link_rf := Sink_rf↑.Buffer_rf;
            Sink_rf↑.Buffer_rf := TempBuffer_rf;
         fi
         // At this point there is always a local buffer.
         //
         // Success indicates whether there is an element
         // to read in the channel.
         //
         <X, Success> :=
         Sink_Read(Sink_rf↑.Buffer_rf, Sink_rf↑.Consumed + 1);
      fi

      if ( Success ≠ ☺ ) then
         // Wait for the source-end to insert elements
         // in the channel.
         // Is worked out in the full version.
         fi
      fi
   done

   Sink_rf↑.Consumed := Sink_rf↑.Consumed + 1;
   Unlock(Sink_rf↑.CE_Lock);
   return <X>;
end
```

This function creates a local buffer at the location of the sink, if the *buffer_rf*-field of the sink is pointing to a non-local buffer. If the *buffer_rf*-field is pointing to a local buffer then the reading can proceed. If it is **NULL** then there are no elements in the channel and the sink waits. This waiting is worked out in the full version of this algorithm, see chapter 4 (section 4.7).

Under normal circumstances, the `while`-loop executes no more than **two** times, and it can be replaced by a couple of `if` statements. However, in order to make MoCha as general as possible, we take into account the possibility that data can get lost and/or that the elements can have expiration dates. In

such cases, if the source inserts elements into the channel and the sink tries to read them (after waiting), there is no guarantee that the sink actually reads them (and may have to wait again). Therefore, a `while`-loop is needed in this function.

3.7.2 Sink_Read

The function *Read* calls the function *Sink_Read* which operates on the Buffer data-structure, see figure 3.1.

```
Sink_Read (Buffer_rf, ToBeAsked)
//
// Buffer_rf ∈ →Buffer
//
// ToBeAsked ∈ Integer, number of values to be asked
// to the next buffer in case Buffer_rf↑ is empty.
//
// returns <X ∈ Value, Success ∈ {☺, ☹} >
//
// This function operates on the Buffer data-structure.
//
begin
    Lock(Buffer_rf↑.Buffer_Lock);

    V := < >; // a sequence of Value.
    Reference := Buffer_rf↑.Link_rf;

    // Execute loop while buffer is empty and
    // there is (still) a reference to another buffer.
    //
    while ( (|Buffer_rf↑.Vals| = 0) ∧ (Reference ≠ NULL) ) do
        // Buffer_Read returns, besides V, a reference to a
        // next buffer in the chain (if any exists) or NULL (if not).
        //
        <V, Reference> := Buffer_Read(Buffer_rf↑.Link_rf, ToBeAsked);
        if ( Reference ≠ NULL ) then
            Buffer_rf↑.Link_rf := Reference;
        fi
        Buffer_rf↑.Vals := V;
    done
```

```
    if (|Buffer_rf↑.Vals| > 0 ) then
        X := Head(Buffer_rf↑.Vals);
        Buffer_rf↑.Vals := Tail(Buffer_rf↑.Vals);
        Success := ☺;
    else
        // No elements found in channel.
        //
        X := UNDEFINED;
        Success := ☹;
    fi

    Unlock(Buffer_rf↑.Buffer_Lock);

    return <X, Success>;
end
```

The `Buffer_rf↑` buffer is the *sink-buffer*, the **local** buffer where the sink's *buffer_rf*-field points to. As explained in chapter 2 (section 2.5), when the *sink-buffer* is empty, it requests a certain amount of elements (according to the heuristic **consumed + 1**, explained in section 2.5.2) from the next buffer in the chain. This is the call to the function *Buffer_Read*. This function returns:

- The requested number, less, or zero elements together with a reference to a next buffer in the chain. This indicates that the asked buffer is now (or was) empty. The *sink-buffer* changes its *link_rf*-field to this new buffer.
- The requested number, less, or zero elements together with a reference that is **NULL**. This indicates that the buffer is not empty after giving the elements, or it is the last buffer of the chain. The *sink-buffer* **does not** change its *link_rf*-field.

Unfortunately, due to the behavior of writing and reading in combination with the mobility of the channel-ends a (sub)chain of empty buffers can exist in the chain of buffers. This is explained in section 3.7.4. Therefore, a `while`-loop is needed in the function *Sink_Read* to handle the case where the *sink-buffer* requests elements from an empty buffer and gets a reference to the next buffer in the chain and has to try again.

Under normal circumstances, this `while`-loop executes no more than **three** times, because there cannot be more than two empty buffers in a chain,

excluding the *sink-buffer* (see section 3.7.4). Again (as in the previous subsection), in order to make the algorithm as general as possible, we take into account that data can get lost and/or that the elements can have expiration dates. Therefore, more than two empty buffers may exist in the chain of buffers.

This function returns a sad face if there are no elements in the channel.

3.7.3 Buffer_Read

The function *Sink_Read* calls the function *Buffer_Read* (see figure 3.1), which operates on the Buffer data-structure, if the *sink-buffer* is empty (see the previous subsection).

```
Buffer_Read (Buffer_rf, Amount)
//
// Buffer_rf ∈ →Buffer
//
// Amount ∈ Integer, number of values requested.
//
// returns <V ∈ Value*, Next_Buffer_rf ∈ →Buffer>
//
// This function operates on the Buffer data-structure.
//
begin
   Lock(Buffer_rf↑.Buffer_Lock);

   V := < >;
   Destroy := FALSE;

   if ( |Buffer_rf↑.Vals| ≤ Amount ) then
       V := Buffer_rf↑.Vals;
       Buffer_rf↑.Vals := < >;
       Next_Buffer_rf := Buffer_rf↑.Link_rf;
       if ( Buffer_rf↑.Link_rf ≠ NULL ) then Destroy := TRUE; fi
   else
       for 1 to Amount do
           V := V ∘ Head(Buffer_rf↑.Vals);
           Buffer_rf↑.Vals := Tail(Buffer_rf↑.Vals);
       od
       Next_Buffer_rf := NULL;
```

```
   fi

   // Unlocking before destroying would make
   // the algorithm unstable.
   //
   if ( Destroy ) then
      Delete(Buffer_rf);
   else
      Unlock(Buffer_rf↑.Buffer_Lock);
   fi

   return <V, Next_Buffer_rf>;
end
```

For an explanation of what this function returns, please see the previous subsection.

The variable *Amount* is not the only upper bound for the amount of elements to be returned. There is another upper bound, set by the number of elements that can be transported during a remote call for the same cost. The amount of data that fits in a message/frame determines this upper bound. Unfortunately, this amount varies for each system and situation, therefore this upper bound is not expressed in the abstract function, but it is given as a comment here.

According to the explanation in chapter 2 (section 2.5.3) a buffer that becomes (or was) empty while reading gets destroyed, unless it is the last buffer of the chain (in which case, its link_rf is **NULL**). If the current buffer is (or was) empty after the read and it returns to the *sink-buffer* a valid reference pointing to its next buffer, then it concludes that it is not longer part of the chain and destroys itself.

At the end of the function, the buffer gets deleted or the *Buffer_Lock*-field is Unlocked. This is done this way because unlocking before destroying the buffer makes the algorithm unstable, somebody else may access the buffer between the times of unlocking and deleting.

3.7.4 Subchain of Empty Buffers

The behavior of writing and reading in combination with the movement of the channel-ends, can lead to a (sub)chain of empty buffers in the chain of

buffers of a channel. Although in this simplified version, the movement of the channel-ends is not taken into account by the functions, it is assumed that a chain of buffers can exist due to the movement of the channel-ends. The issue of empty buffers is important for the function *Sink_Read*.

When a channel-end moves from one location to another, its *buffer_rf*-field changes from local to non-local. No new local buffer is created until the component(s) having access to the channel-end actually read from or write to the channel. More about the movement of the channels-ends is explained in chapter 2 (section 2.4). Buffers get destroyed only when reading elements out of them (or trying to read if the buffer was already empty), as explained in section 3.7.3. This behavior of the channel allows a subchain of empty buffers to exist in the chain of buffers, but there is a limit of at most **two** empty buffers for the whole channel.

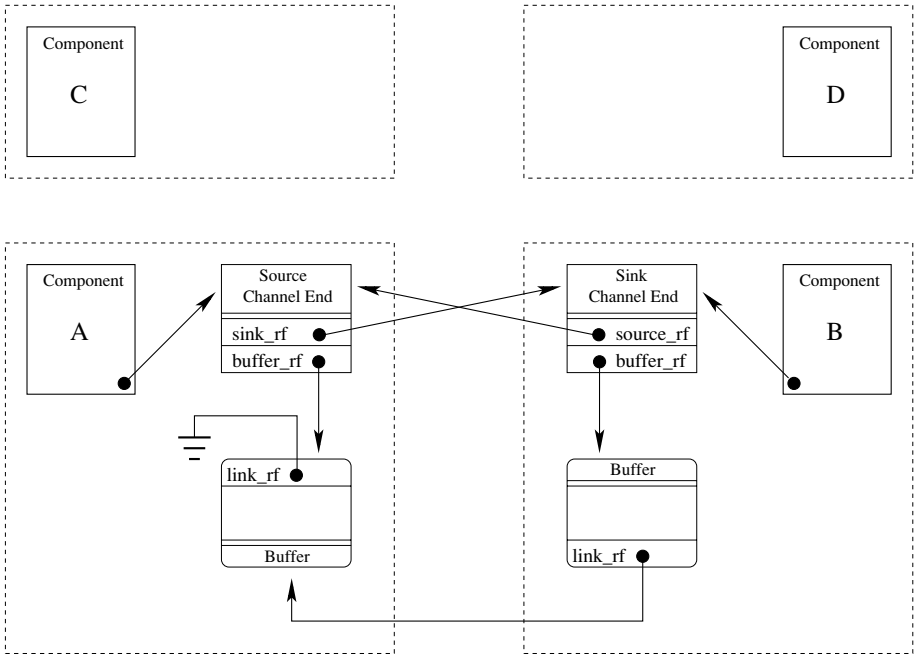


Figure 3.2: Subchain of Empty Buffers, 1.

The worst case, where there is a (sub)chain of two empty buffers, is shown in figure 3.2. Component A has access to the source-end and component B to the sink-end. Both channel-ends have a reference to a local buffer. These buffers are empty. The buffer at the location of the source contained elements

in the past but they are all read now. It does not get destroyed because it is the last buffer of the chain.

Two buffers is the limit. If at some point in time the sink channel-end is moved to another location and a component having access to it starts to read, (in figure 3.2, for example, component D), then the buffer at the location of component B gets destroyed and there are still two empty buffers. If, on the other hand, the source channel-end moves to another location and a component having access to it starts to write, (in figure 3.2, for example, component C), then a new local buffer is created but it will not be empty. Again, there are only two empty buffers in the channel.

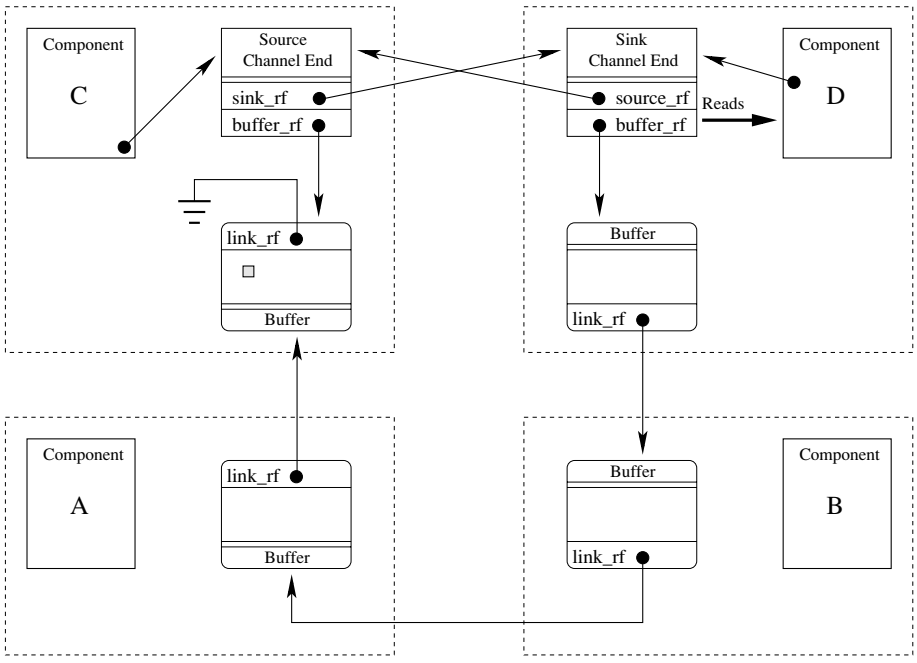


Figure 3.3: Subchain of Empty Buffers, 2.

This limit of two buffers does not apply **while** executing the actions read and/or write, it applies when observing the channel before and after these actions. Otherwise the limit is **four** empty buffers, because when a new local buffer is created, it is initially empty. In figure 3.3, both channel-ends have moved to another location. Component C has written into the channel, and component D is still in the process of reading an element. A local (empty) *sink-buffer* is created, and in this situation, there are **three** empty buffers.

If component C is also still in the process of writing the first element, then there are **four** empty buffers. After the action read, the situation is the same as in figure 3.2.

The limit of two buffers is important for the function *Sink_Read*, as explained in section 3.7.2. Due to empty buffers, a `while`-loop is needed in the function to handle the case that the *sink-buffer* requests elements from an empty buffer but gets a reference to the next buffer in the chain and has to try again. Because of the limit on the number of empty buffers this `while`-loop executes at most **three** times. In figure 3.3, component D is reading and a local *sink-buffer* is created. Because this buffer is empty, it requests elements from the next buffer. Due to the empty subchain of buffers it must request **three** times for elements before getting any.

The algorithm can be changed in such a way that the *sink-buffer* has to request elements only from one buffer in order to get any, or none if this buffer is the last one of the channel:

- The sink channel-end can check before it moves whether the local buffer, (where the *buffer_rf*-field points to), is empty and not the last one in the chain. In that case, it can destroy it, set up its *buffer_rf* to the next buffer in the chain, and then move.
- The source channel-end can check, after it moves and does the first write, but before it creates a new local buffer, whether the buffer that it's *buffer_rf*-field points to, is empty. In that case it can destroy it, and fix the references in the chain of buffers. To do this, the source channel-end needs a reference to the last-but-one buffer of the chain[6].

Changing the algorithm in the way just described, alleviates the need for a `while`-loop in the function: two `if`'s are enough. However, in order to make the algorithm as general as possible, we take into account that data can get lost and/or that the elements can have expiration dates. Therefore, empty buffers can still exist and the `while`-loop is needed anyway. It is then not worthwhile to apply this change to the algorithm, the gain of less empty buffers does not compensate for the extra complexity of the changed algorithm.

[6]The algorithm gets more complicated here because it must make sure this reference remains valid. The buffer can get destroyed by a read action.

Chapter 4

MoCha

Due to the complexity of MoCha, a simplified version was presented in the previous chapter. This version was simplified by excluding extra semaphores for the movement of the channel-ends, error-handling, and other operations where movement has to be considered. In this chapter, we present the full version of MoCha.

First, we present the updated data-structures. In section 4.2, we describe semaphores and their use. In section 4.3, we describe errors and error-handling. Finally, we give an overview of the functions that implement MoCha.

4.1 Data-structures

The algorithm uses three data-structures: *Source_Channel_End*, *Sink_Channel_End*, and *Buffer*. These are the same as in the simplified version, described in chapter 3 (section 3.1), with some new added semaphores and a boolean: *Move_Lock*, *Read_Lock*, and *Wait_Read*.

Source_Channel_End = <Buffer_rf, Sink_rf, CE_Lock, Move_Lock, Wait_Read>

- Buffer_rf = A reference to a Buffer data-structure.
- Sink_rf = A reference to the Sink_Channel_End of the same channel.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Wait_Read = A boolean used to express that the Sink_Channel_End is waiting for values.

Sink_Channel_End = <Buffer_rf, Source_rf, Consumed, CE_Lock, Move_Lock, Read_Lock>

- Buffer_rf = A reference to a Buffer data-structure.
- Source_rf = A reference to the Source_Channel_End of the same channel.
- Consumed = An integer that keeps track of the number of values consumed by the component(s) since the last move.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Read_Lock = A binary semaphore used to block the sink when waiting for values to be written into the channel (in case it was empty), initially locked.

Buffer = <Vals, Link_rf, Buffer_Lock>

- Vals is a sequence of Value.
- Link_rf = A reference to a Buffer data-structure.
- Buffer_Lock = A binary semaphore. It is used for mutual exclusion of the buffer.

The binary semaphores have two operations: *Lock* and *Unlock*. Initially all semaphores are unlocked, except for the Read_Lock semaphore, which is initialized to be locked. We assume that the waiting-queue of a semaphore, where the blocked processes are waiting, has a FIFO-structure[1].

4.2 Semaphores

In this section we explain the use of the semaphores in MoCha. This explanation makes the reading of the code, later on, easier to understand. The semaphores we discusse are: *CE_Lock*, *Move_Lock*, and *Read_Lock*. The

[1]This is not necessary for the protocol, but it is assumed for fairness of the system.

semaphore *Buffer_Lock* is not discussed because its use for the mutual exclusion of its buffer data-structure is trivial.

From an abstract point of view the semaphores we discusse non-exclusively serve two different purposes. They are used either for interaction between components and channel-ends, or for interactions between the channel-ends. The semaphore *CE_Lock* is used for the first purpose. The semaphores *CE_Lock*, *Move_Lock*, and *Read_Lock* are used for the second purpose.

4.2.1 CE_Lock

The *CE_Lock* semaphore is used for both interactions between components and channel-ends, as well for interactions between the source- and the sink-ends of a channel.

Component $\longleftrightarrow$ Channel-end Interaction

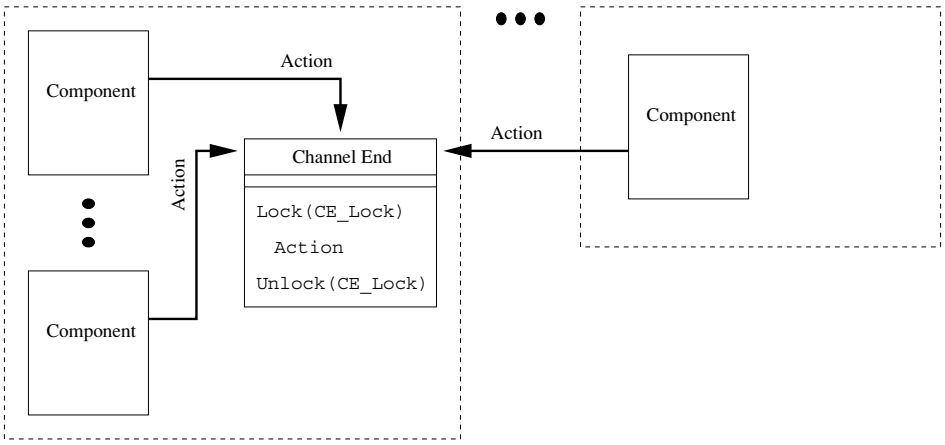


Figure 4.1: CE_Lock Semaphore.

For the interaction between components and a channel-end, a general lock is needed for mutual exclusion. In the explanation of chapter 2, only one component can have access to a particular channel-end at a time (see chapter 2, section 2.1). However, MoCha is more general because every component that has a **valid** reference to a channel-end can use it, leaving the enforcement of a *one-to one restriction* to an upper level , if necessary. For this reason, there is a semaphore on each channel-end used for mutual exclusion. The *CE_Lock* semaphore makes sure that only one action (read, write, move, or

destroy) is being carried out at the same time on the same channel-end (see figure 4.1).

Source $\longleftrightarrow$ Sink Interaction

The *CE_Lock* is also used for some interactions between the two ends of the same channel. It is used for this purpose in the functions *Read* and *Destroy_Channel*. In both cases, the sink channel-end needs to lock the source. More explanation about the first case is given in sections 4.2.3, and 4.7. More explanation of the second case is given in section 4.9.

4.2.2 Move_Lock

The *Move_Lock* semaphore is used for movement synchronization between the channel-ends. Channel-ends can move from one location to another. However, the source and the sink cannot move at the same time, because this causes problems with the references they have for each other (*source_rf*, *sink_rf*), leading to the possibility that they would "lose" each other. Therefore, when one channel-end is moving the other one must wait until the move is complete, and its reference to this channel-end is updated.

One implementation of this restriction is to let the channel-end that wants to move lock the *CE_Lock* of the other channel-end first, then move, then update the reference of the other channel-end, and then unlock it. While this is a valid solution, it has some disadvantages. For example, locking the *CE_Lock* of the other channel-end not only blocks it to prevent **move** actions, but also blocks the actions **read** and **write** as well. However, it is okay to read from or write to one end of a channel while its other end is moving. Another disadvantage is that this solution leads to a considerably more complicated implementation than a version using an extra semaphore for moving. This extra complication arises, for example, in the *Read* function in the situation where the sink channel-end has to wait for elements, when the channel is empty (see section 4.7).

For the reasons just described, an extra semaphore is added to the channel-ends for movement synchronization, the *Move_Lock*. A channel-end has to lock both its own *Move_Lock* and that of the other channel-end before it can move itself. To avoid deadlocks, the locking of the semaphores is always done in the same order: first lock the *Move_Lock* of the sink, and then the one of the source.

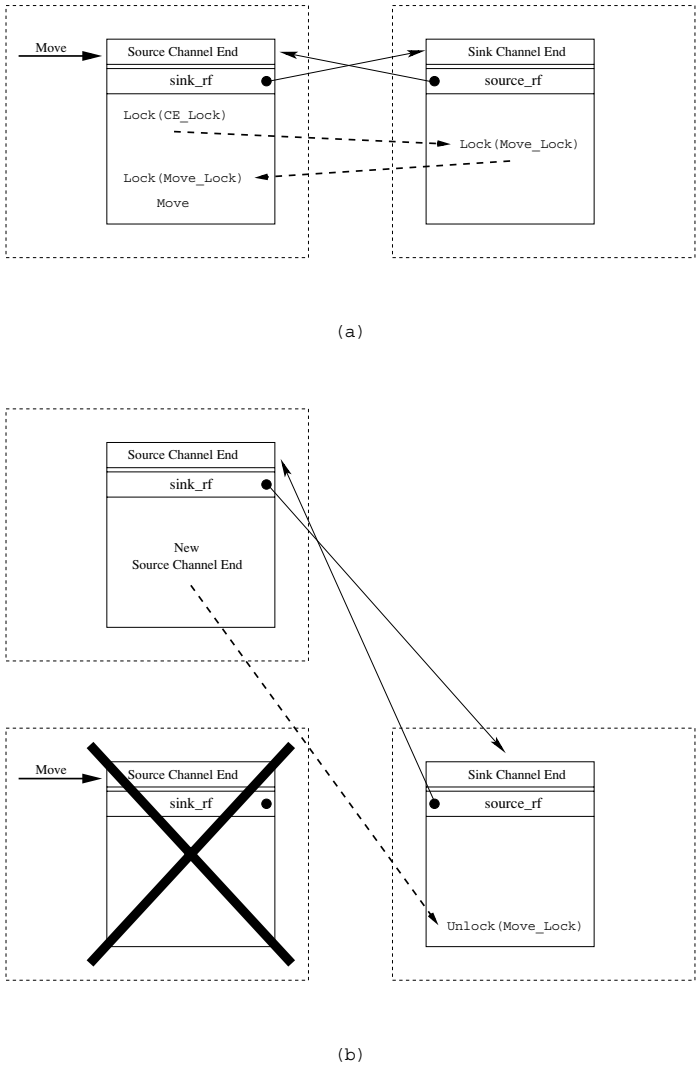


Figure 4.2: Move_Lock Semaphore, Moving of Source.

Figure 4.2 shows an example of the movement of the source channel-end. After acquiring the general lock (*CE_Lock*) a move action on the source then locks the *Move_Lock* of the sink, then the *Move_Lock* of the source. Now the source can move to another location, and after updating the *source_rf*-field of the sink, the move action is complete and it unlocks the sink. The moved source, which is actually a new source channel end (the old one gets destroyed), does not need to unlock itself; it is already initialized unlocked. More explanation about moving is given in section 4.8.

4.2.3 Read_Lock

The *Read_Lock* is used for blocking the sink channel-end when it must wait for elements because the channel is empty. While executing the action **read**, the channel can be empty, in which case the sink must wait for new elements to be inserted by the source. The sink can either wait for a period of time and try reading again (polling), or ask the source to notify it when it writes new elements. The first solution is not desirable because looking periodically at the channel to see if there are elements can be quite costly (for the system) if none is written to it for a long time; costs include, for example, unnecessary traffic in the network, and unnecessary use of resources. The second solution is much nicer and costs less, because the sink looks again only when it is sure[2] that there are elements in the channel[3].

To implement the solution where the source notifies the sink when it writes elements into the channel, the semaphore *Read_Lock* is used in combination with a boolean called *Wait_Read*. The semaphore is part of the *Sink_Channel_End* data-structure and the boolean is in the *Source_Channel_End* data-structure. Figure 4.3 shows the use of both fields. In 4.3(a) the sink attempts to read but there are no elements in the channel. It, therefore, locks the source and sets its *Wait_read* boolean to **TRUE**. After unlocking the source, the sink blocks itself on the *Read_Lock* (initially this semaphore is locked) and waits. In 4.3(b) the source writes an element into the channel, notices that its *Wait_read* is **TRUE**, and therefore it unlocks the *Read_Lock* of the sink, after which it resets its *Wait_Read* boolean to **FALSE**. Both sink and source continue with their execution; the source unlocks its *CE_Lock*, the sink attempts to read again, succeeds, and then unlocks its *CE_Lock*.

Figure 4.3 shows an example of the case where the elements do not have an expiration date and/or data does not get lost in the chain of buffers. Also the situation when the sink has access to the source while it wrote elements when the sink tried to lock it, is not discussed here. More explanation about this is given, together with the function *Read*, in section 4.7.

[2]When working with data elements with expiration-dates and/or assuming that data can get lost, the sink cannot be sure, but chances are considerably higher that elements are read than in the first solution

[3]The polling solution is cheaper than the second one only in systems where there is a lot of writing all the time. However, if this is the case, then the chances of not finding elements in the channel in the first place are low, making the costs of both solutions almost equal in this case.

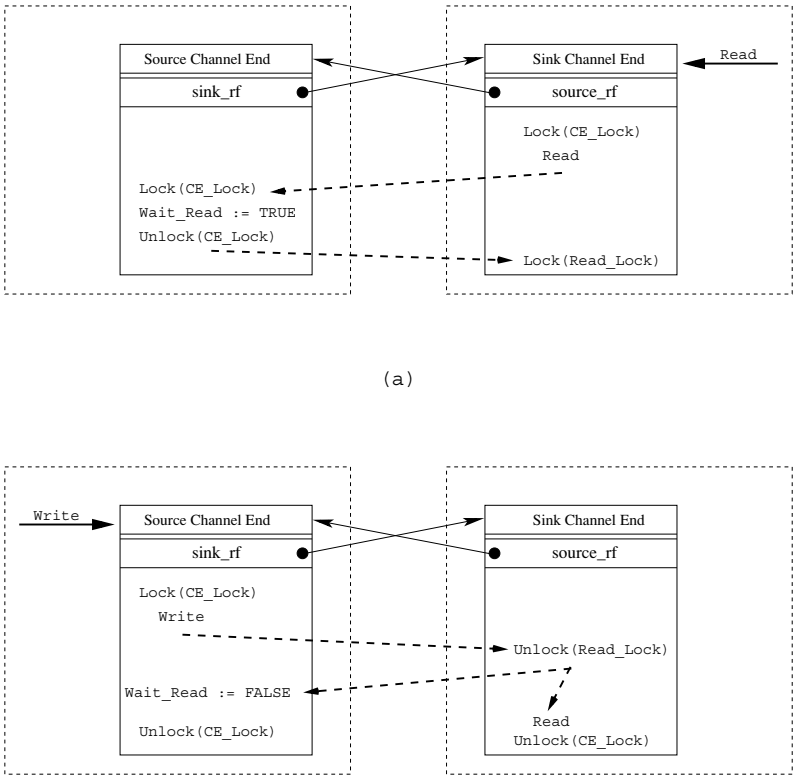


Figure 4.3: Read_Lock Semaphore.

4.3 Error-handling

In our abstract algorithms we do not take into account the typical errors that can occur during the execution of an implementation. However, there are three kinds of errors our algorithms must take into account, due to their structure: the errors caused by invalid references (dangling references), the errors caused by destruction of channel-ends, and the errors caused by non-existing or non-reachable locations. To cope with these errors, the operations *Lock* and *new* have a special property; they can detect these cases and return an error, which the algorithms can detect. Due to this errors, all functions available to the components, return whether the action/function succeeded or failed[4].

[4]In programming languages that support exceptions this will be the mechanism of choice to deal with this type of the errors.

4.3.1 Dangling References and the Operation *Lock*

The movement and destruction of channel-ends lead to dangling references. This can happen at two levels: at the level of components and channel-ends interactions, and at the level of sink and source interactions. The operation *Lock* returns an error if it is passed an invalid reference.

Components $\longleftrightarrow$ Channel-ends Interaction

Several components can have access to the same channel-end at the same time when each has a **valid** reference to it. However, if the channel-end moves or gets destroyed, these references are not valid anymore unless they are updated. Our algorithms do not keep track of the components that reference a particular channel-end, and thus cannot update such references[5]. Therefore, dangling references may exist in the system, and the algorithms must take this into account.

To cope with this situation the operation *Lock* returns an error if it is passed a dangling reference. This is expressed in the code in the following way:

```
ERROR := Lock(Channel_End_rf↑.CE_Lock);
if ( ERROR ) then return FAILURE;
```

When the channel-end reference is invalid, the function returns FAILURE. The component that receives it then realizes, after looking at the type of the error, that its reference to this particular channel-end is invalid and discards it.

Another solution for the dangling references problem, is to use *virtual channel-ends*. If a channel-end moves, a virtual channel-end, which refers to the new location, is left at the old location. A component trying to access the channel-end is then redirected to the new location. If a channel-end gets destroyed, a virtual channel-end is created to let components know that the channel-end does not exist anymore. With this solution, components can not have dangling references to channel-ends caused by channel-end movement or destruction. However, this solution has two disadvantages: the extra administration of updating the virtual channel-ends when a channel-end moves, and the extra space needed for the virtual channel-ends. Therefore, MoCha does not implement this solution.

[5]This can be left to a third party, e.g. , the instance that moves a channel-end.

Sink $\longleftrightarrow$ Source Interaction

When either the sink or the source is moving, the other end has a dangling reference to it for a short period of time. This is the period between the destruction of the old channel-end and the updating of the reference to the new one. During this period the non-moving channel-end can try to access the moving channel-end and receive an error (given by *Lock*). It then tries accessing again because it knows that the reference will be updated shortly. In the code it is expressed in the following way:

```
do
    ERROR := Lock(Channel_End_rf↑.Any_Lock);
until ( ¬ERROR )
```

4.3.2 Destruction of Channel-ends and the Operation *Lock*

A channel-end gets destroyed either when it moves, (the old channel-end gets destroyed and a new one is created), or when the whole channel is destroyed (by the function *Destroy_Channel*). If a component/channel-end is blocked at the semaphore of a channel-end, and this channel-end gets destroyed (therefore destroying the semaphore as well), then the operation *Lock* returns an error. This error is related to the error produced by the dangling references explained in the previous section (4.3.1), the code to handle with this error is the same as in that section.

4.3.3 New

When trying to create a new instance of a channel-end, the given location(s) may be non-existing or unreachable. To cope with this case, the operation *new* returns an error when it fails to create a channel-end at a non-existing or unreachable location. Otherwise it creates the channel-end and returns a valid reference to it. This is expressed in the code in the following way:

```
Channel_End_rf := new Channel_End @ Location;
if ( Channel_End_rf = ERROR ) then return FAILURE;
```

Because the location where a new buffer is created, is always local to the channel-end that creates it, a possible error is not checked in our algorithms; this location always exists and is reachable. Our algorithms check a possible error given by the operation *new* only when creating a new channel-end.

4.4 Functions, an Overview

After the intuitive explanation of chapter 2, the presentation of a simplified version of MoCha in chapter 3, an overview of the semaphores in section 4.2, and an overview of error-handling in section 4.3, we now present the functions of MoCha.

A channel can be accessed through five functions, which operate on the Channel-end data-structures: Create_Channel, Write, Read, Move, and Destroy_Channel. For an explanation of the $\rightarrow$ and $\uparrow$ notation, see chapter 3 (section 3.3).

- **Create_Channel** (*Loc1* $\in$ Location, *Loc2* $\in$ Location)
  returns $<$*Source_rf* $\in \rightarrow$Source_Channel_End, *Sink_rf* $\in \rightarrow$Sink_Channel_End, *Action_Status* $\in$ {SUCCESS, FAILURE}$>$
- **Write** (*Source_rf* $\in \rightarrow$Source_Channel_End, $X \in$ Value)
  returns $<$*Action_Status* $\in$ {SUCCESS, FAILURE}$>$
- **Read** (*Sink_rf* $\in \rightarrow$Sink_Channel_End)
  returns $<X \in$ Value, *Action_Status* $\in$ {SUCCESS, FAILURE}$>$
- **Move** (*Channel_End_rf* $\in$ {$\rightarrow$Source_Channel_End, $\rightarrow$Sink_Channel_End}, *Target* $\in$ Location)
  returns $<$*New_Channel_End_rf* $\in$ {$\rightarrow$Source_Channel_End, $\rightarrow$Sink_Channel_End}, *Action_Status* $\in$ {SUCCESS, FAILURE}$>$
- **Destroy_Channel** (*Channel_End_rf* $\in$ {$\rightarrow$Source_Channel_End, $\rightarrow$Sink_Channel_End})
  returns $<$*Action_Status* $\in$ {SUCCESS, FAILURE}$>$

These five functions use three functions that operate on the buffer data-structure. They are listed in chapter 3 (sections 3.6, 3.7.2, and 3.7.3). Because they are the same in this full version of MoCha, they are only mentioned here:

- **Write** (*Buffer_rf* $\in \rightarrow$Buffer, $X \in$ Value)
- **Sink_Read** (*Buffer_rf* $\in \rightarrow$Buffer, *ToBeAsked* $\in$ Integer)
  returns $<X \in$ Value, *Success* $\in$ {☺, ☹} $>$
- **Buffer_Read** (*Buffer_rf* $\in \rightarrow$Buffer, *Amount* $\in$ Integer)
  returns $<V \in Value^*$, *Next_Buffer_rf* $\in \rightarrow$Buffer$>$

4.5 Creation of a Channel

To create a channel the following function is used:

```
Create_Channel (Loc1 , Loc2)
//
// Loc1 = The location where to create the Source_Channel_End.
// Loc2 = The location where to create the Sink_Channel_End.
//
// returns <Source_rf ∈ →Source_Channel_End,
// Sink_rf ∈ →Sink_Channel_End,
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   Source_rf := new Source_Channel_End @ Loc1;
   if ( Source_rf = ERROR ) then
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Sink_rf := new Sink_Channel_End @ Loc2;
   if ( Sink_rf = ERROR ) then
      Delete(Source_rf);
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Source_rf↑.Buffer_rf := NULL;
   Source_rf↑.Wait_Read := FALSE;
   Sink_rf↑.Buffer_rf := NULL;
   Sink_rf↑.Consumed := 0;

   // Knowing each other.
   //
   Source_rf↑.Sink_rf := Sink_rf;
   Sink_rf↑.Source_rf := Source_rf;

   return <Source_rf, Sink_rf, SUCCESS>;
end
```

This function is an updated version of the one presented in chapter 3 (section 3.5), which returns FAILURE when one or both of the given locations are non-existent or unreachable (see section 4.3.3). If the locations are valid

then the function returns references to the created *channel-ends*, plus the *Action_Status* SUCCESS.

4.6 Writing

A component that has a valid reference to a source channel-end can write using the following function:

```
Write (Source_rf, X)
//
// Source_rf ∈ →Source_Channel_End, X ∈ Value.
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
begin
   ERROR := Lock(Source_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;

   if ( Source_rf↑.Buffer_rf = NULL ) then
      // Create first buffer of channel.
      //
      Source_rf↑.Buffer_rf := new Buffer;
      Source_rf↑.Buffer_rf↑.Link_rf := NULL;

      // Sink can not change its buffer field while it is NULL
      // and by locking the move of the source
      // the sink can not move either.
      //
      Lock(Source_rf↑.Move_Lock);
      // Tell the other side.
      //
      Source_rf↑.Sink_rf↑.Buffer_rf := Source_rf↑.Buffer_rf;
      Unlock(Source_rf↑.Move_Lock);
   else
      if ( ¬Local(Source_rf↑.Buffer_rf) ) then
         // Create new local buffer
         //
         TempBuffer_rf := new Buffer;
         TempBuffer_rf↑.Link_rf := NULL;
         Source_rf↑.Buffer_rf↑.Link_rf := TempBuffer_rf;
```

```
            Source_rf↑.Buffer_rf := TempBuffer_rf;
        fi
    fi

    // At this point there is always a local buffer.
    //
    Write(Source_rf↑.Buffer_rf, X);
    if ( Source_rf↑.Wait_Read ) then
        // Sink is waiting for values
        //
        Unlock(Source_rf↑.Sink_rf↑.Read_Lock);
        Source_rf↑.Wait_Read := FALSE;
    fi

    Unlock(Source_rf↑.CE_Lock);
    return SUCCESS;
end
```

The main functionality of this function is explained in chapter 3 (section 3.6). The new added features are: the implementation of error-handling features of the *Lock* operation, locking of the *Move_Lock* semaphore when setting up the *Buffer_rf*-field of the sink, and notification to the waiting sink when writing an element.

The error-handling features of the *Lock* operation are explained in section 4.3. It is applied to all *Locks* except when trying to lock the *Move_Lock*. When trying to lock this semaphore, no error is possible; the source_rf is a valid reference, and the source cannot move or get destroyed. Therefore, no error-checking is needed.

When setting up the *Buffer_rf*-field of the sink, the sink must not move. Therefore, the function should lock the *Move_Lock* of the sink. However, by locking the *Move_Lock* of the source, the sink cannot move either. This operation is much cheaper due to the fact that it is always a local lock, instead of a non-local one.

After writing an element, the source checks if the sink is waiting for an element and notifies it when needed. The sink is waiting if the boolean *Wait_Read* is TRUE. In that case, the source notifies the sink by unlocking its *Read_Lock*. More explanation is given in section 4.2.3, and in the description of the function *Read* (section 4.7).

4.7 Reading

A component that has a valid reference to a sink channel-end can read from it using the following function:

```
Read (Sink_rf)
//
// Sink_rf ∈ →Sink_Channel_End.
//
// returns <X ∈ Value, Action_Status ∈ {SUCCESS, FAILURE}>
//
begin
   ERROR := Lock(Sink_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   Success ∈ {☺,☹} := ☹;

   while ( Success ≠ ☺ ) do
       if ( Sink_rf↑.Buffer_rf ≠ NULL ) then
           if ( ¬Local(Sink_rf↑.Buffer_rf) ) then
               // New local Buffer.
               //
               TempBuffer_rf := new Buffer;
               TempBuffer_rf↑.Link_rf := Sink_rf↑.Buffer_rf;
               Sink_rf↑.Buffer_rf := TempBuffer_rf;
           fi
           // At this point there is always a local buffer.
           //
           // Success indicates whether there is an element
           // to read in the channel.
           //
           <X, Success> :=
           Sink_Read(Sink_rf↑.Buffer_rf, Sink_rf↑.Consumed + 1);
       fi

       // In case that there is no element in
       // the channel "consult" the source
       //
       if ( Success ≠ ☺ ) then
           do
               ERROR := Lock(Sink_rf↑.Source_rf↑.CE_Lock);
```

```
            until (¬ERROR)
            if ( Sink_rf↑.Source_rf↑.Buffer_rf ≠ NULL ) ∧
                |Sink_rf↑.Source_rf↑.Buffer_rf↑.Vals| ≠ 0 ) then
                Unlock(Sink_rf↑.Source_rf↑.CE_Lock);
            else
                Sink_rf↑.Source_rf↑.Wait_Read := TRUE;
                Unlock(Sink_rf↑.Source_rf↑.CE_Lock);
                // Read_Lock is initially locked!
                //
                Lock(Sink_rf↑.Read_Lock);
            fi
        fi
    done

    Sink_rf↑.Consumed := Sink_rf↑.Consumed + 1;
    Unlock(Sink_rf↑.CE_Lock);
    return <X, SUCCESS>;
end
```

The main functionality of this function is explained in chapter 3 (section 3.7). The most important new added feature is the implementation of the waiting for elements by the sink.

The function *Sink_Read*, (see chapter 3, section 3.7.2), can return a sad `Success`, in which case the channel is empty and the sink must wait for an element to be written by the source. It locks the source, (the `until-do` loop is explained in section 4.3), sets the *Wait_Read* boolean to TRUE, unlocks the source, and blocks itself by trying to lock the *Read_Lock* (this lock is initially locked). This procedure is explained in section 4.2.3, and illustrated in figure 4.3.

However, it is possible for the source to write elements before the sink successfully locks it. Therefore after locking the source, the sink looks at the buffer that the *Buffer_rf* of the source refers to. If this buffer is non-empty, then the source wrote elements during this period; it does not matter if it moved and (even) perhaps created new buffers, if this buffer is non-empty it wrote elements. If the buffer is empty or non-existent[6], then the source did not write any elements; the source creates new buffers only when it must write elements to it. In the first case, the sink just unlocks the source and

[6]When no elements are written to the channel since its creation, there exist no buffers in the channel.

reads again. In the second case, the sink sets up everything in order to wait for an element.

4.8 Move

A component that has a valid reference to a channel-end can move it using the following function:

```
Move (Channel_End_rf, Target)
//
// Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End}
//
// Target = The location target.
//
// returns <New_Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End},
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   ERROR := Lock(Channel_End_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   if ( Channel_End_rf↑ ∈ Sink_Channel_End ) then
      // Lock Sink first, then Source.
      // There is no Unlock for the following Lock.
      //
      Lock(Channel_End_rf↑.Move_Lock);
      Lock(Channel_End_rf↑.Source_rf↑.Move_Lock);

      New_Channel_End_rf := new Sink_Channel_End @ Target;
      if ( New_Channel_End_rf = ERROR ) then
         Unlock(Channel_End_rf↑.Source_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.CE_Lock);
         return <UNDEFINED, FAILURE>;
      fi

      Copy_Information(Channel_End_rf, New_Channel_End_rf);
      New_Channel_End_rf↑.Consumed := 0;
      Channel_End_rf↑.Source_rf↑.Sink_rf := New_Channel_End_rf;
      Delete(Channel_End_rf);
```

```
        Unlock(New_Channel_End_rf↑.Source_rf↑.Move_Lock);

    else

        // Lock Sink first, then Source.
        //
        do
            ERROR := Lock(Channel_End_rf↑.Sink_rf↑.Move_Lock);
        until ( ¬ERROR )

        // There is no Unlock for the following Lock.
        //
        Lock(Channel_End_rf↑.Move_Lock);
        New_Channel_End_rf := new Source_Channel_End @ Target;
        if ( New_Channel_End_rf = ERROR ) then
            Unlock(Channel_End_rf↑.Move_Lock);
            Unlock(Channel_End_rf↑.Sink_rf↑.Move_Lock);
            Unlock(Channel_End_rf↑.CE_Lock);
            return <UNDEFINED, FAILURE>;
        fi
        Copy_Information(Channel_End_rf, New_Channel_End_rf);
        Channel_End_rf↑.Sink_rf↑.Source := New_Channel_End_rf;
        Delete(Channel_End_rf);

        Unlock(New_Channel_End_rf↑.Sink_rf↑.Move_Lock);
    fi
    return <New_Channel_End_rf, SUCCESS>;
end
```

This function locks the *CE_Lock* of the given channel-end, then checks whether it is a sink- or a source channel-end. If it is a sink channel-end it first tries to lock its *Move_Lock* and then the one of the source. After that, it creates a new sink channel-end in the given location (if the location is unreachable or non-existent, then the function returns FAILURE and unlocks all locks). Then it copies the information of the old channel-end to the new one and destroys the former. Finally, it unlocks the source[7]. If the channel-end is a source, the procedure is symmetrical, except that it tries to lock the *Move_Lock* of

[7]There is no need to unlock the sink. The old sink is destroyed and the semaphores of the new sink are initialized properly.

the sink first and then its own. More information about moving is given in chapter 2 (section 2.4), and in section 4.2.2 more information is given about the *Move_Lock*.

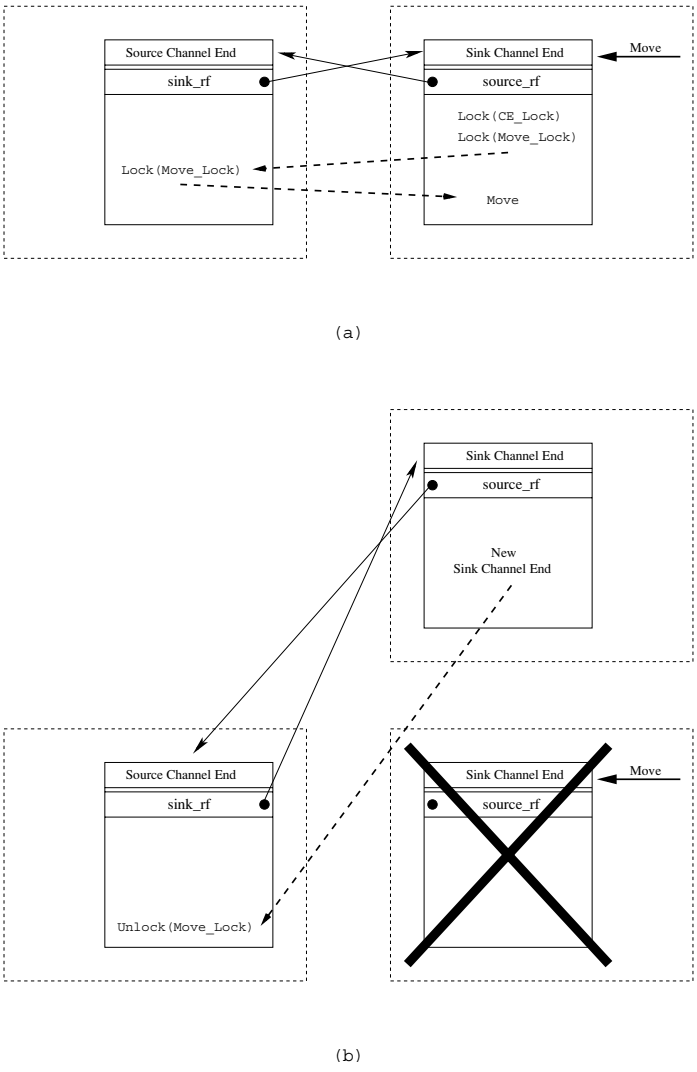


Figure 4.4: Moving of Sink Channel-end.

Figure 4.2 illustrates the moving of a source channel-end. For completeness, figure 4.4, shows the movement of the sink channel-end.

4.9 Destruction of a Channel

A component that has a valid reference to any of the two ends of a channel, can destroy the entire channel using the following function:

```
Destroy_Channel (Channel_End_rf)
//
// Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End}
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
// There is no Unlock in this function
//
begin

   if ( Channel_End_rf↑ ∈ Sink_Channel_End ) then
      Sink_rf := Channel_End_rf;
   else
      // Check for dangling reference
      //
      ERROR := Lock(Channel_End_rf↑.CE_Lock);
      if ( ERROR ) then return FAILURE;
      Sink_rf := Channel_End_rf↑.Sink_rf;
      Unlock(Channel_End_rf↑.CE_Lock);
   fi

   // At this point Sink_rf ∈ Sink_Channel_End,
   // but needs not be valid anymore due to possible
   // movement of sink channel-end.

   // Lock Sink first, then Source.
   //
   ERROR := Lock(Sink_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;

   do
      ERROR := Lock(Sink_rf↑.Source_rf↑.CE_Lock);
   until ( ¬ERROR )

   Source_rf := Sink_rf↑.Source_rf;
```

```
   // Delete chain of buffers.
   //
   i := Sink_rf↑.Buffer_rf;
   While ( i ≠ NULL ) do
       j := i↑.Link_ref;
       Delete(i);
       i := j;
   done

   Delete(Source_rf);
   Delete(Sink_rf);
   return SUCCESS;
end
```

This function receives a reference to a channel-end and instead of trying to lock it immediately, it searches for a reference to the sink channel-end of the channel. If the channel-end itself is the sink-end then it already has the reference, otherwise it has to lock the source, get the reference to the sink, and then unlock the source. After acquiring a reference to the sink channel-end, the function locks the sink and then the source. It then deletes the buffers of the channel (if any) and at the end the two channel-ends.

After acquiring a reference to the sink channel-end, it can be (or become) invalid due to movement of the sink. Therefore, when trying to lock the sink this can give an error. That is why the function checks for a possible error.

The order of locking the channel-ends is sink first and then the source. This order is chosen to avoid deadlock. Reversing this order gives a deadlock in combination with the function *Read* (section 4.7). This is the case when the sink just read but got no element, and therefore tries to lock the source to setup the boolean (if needed). At the same time, the channel is being destroyed and the source is already blocked by the function *Destroy_Channel* (with the reversed order). Then, this last function wants to lock the sink but has to wait, and the *Read* function is also waiting to access the source channel-end: the result is deadlock.

The function just needs one of the two channel-ends to be able to destroy the entire channel, and it can be either the source or the sink. The function could also have demanded both channel-ends before destroying the entire channel, but this is left to an upper level to enforce (if needed). In practice, it is really hard, for a component instance that wants to destroy a channel, to have a

reference to both channel-ends. That is the reason why the function settles for one.

Chapter 5

Discussion, Conclusions, and Future Work

In this thesis we presented MoCha, a model for distributed *Mo*bile *Cha*nnels. MoCha describes an implementation of an asynchronous mobile channel in a distributed environment by giving a set of functions. These functions are abstract algorithms covering all major operations of a channel: *Create_Channel*, *Write*, *Read*, *Move*, and *Destroy_Channel*.

Due to the assumptions we made for the environment, and given our data-structures, and the structure of our algorithms, we can conclude that MoCha can easily be implemented in any modern language like Java or C++, that support: data-structures, instances of data-structures, pointers/references, distributed networking, and threads.

In the introduction of this thesis (see chapter 1, section 1.4), we claimed that MoCha describes an **efficient** and **non-trivial** implementation of a mobile channel. In this chapter we show and conclude that this is the case. At the end of the chapter we discuss future work.

5.1 An Efficient Implementation

We claim that MoCha describes an **efficient** implementation of a mobile channel. The efficiency of MoCha lies in the fact that it minimizes the amount of non-local data-transfers.

In order to minimize the amount of non-local data-transfer, MoCha uses **local** buffers for the operations *read* and *write*. Due to the movement of the

ends of a channel, these buffers are distributed all over the system, and are linked up to each other forming a chain of buffers, (see chapter 2, section 2.4.4).
A local[1] buffer is created, and added to the chain, only when a component really starts to read or to write. Therefore, there are no buffers that are never used in MoCha. This is important for the amount of empty buffers in the chain, and therefore important for the amount of non-local calls while reading. More on this is explained in chapter 3, section 3.7.

When a channel-end moves, its local buffer does not move with it. The buffers do not move in MoCha. A channel-end can be passed on to many components before one of them actually starts reading from or writing to it. Therefore, by not moving the buffers, there is no unnecessary movement of elements.

However, while reading, non-local transfer of elements may be necessary. This is the case when the local buffer of the sink channel-end is empty, but there are elements in other buffers (see chapter 2, section 2.5). MoCha reduces the amount of non-local transfers by using two heuristics for the amount of elements to be transfered. One is based on the number of elements that, the component holding the sink, already has consumed. The other one is based on the number of elements that can be transported during a remote call for the same cost. Both heuristics are explained in chapter 2, section 2.5.2.

Because of the features given above, In MoCha:

- Every write is always a local operation.
- A read is either a local or non-local operation. However, MoCha reduces the amount of non-local read operations.
- The moving of channel-ends does not involve any data-transfer of elements at all.

MoCha versus Centralized Buffer

MoCha is more efficient than, for example, a centralized buffer implementation. Figure 5.1 shows such an implementation, with the thick arrow expressing data-flow. In a centralized buffer, every read or write is a non-local

[1] Creation of buffers in MoCha is always local.

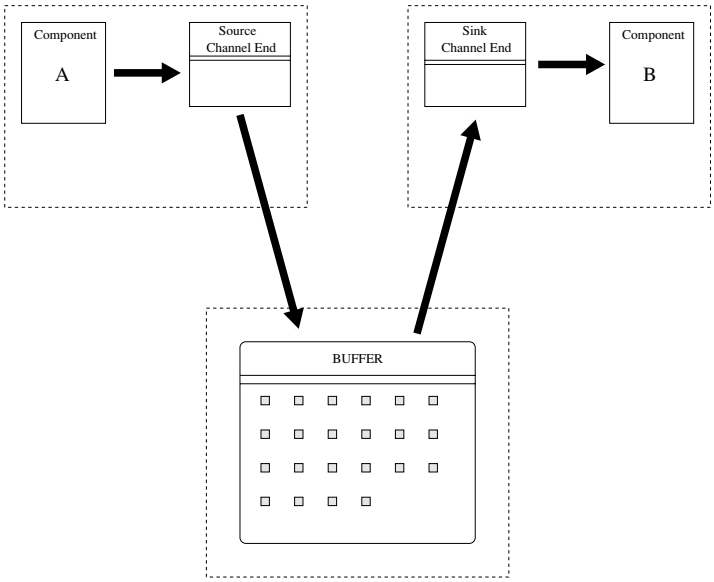


Figure 5.1: A Centralized Buffer Implementation.

operation, making it a costly implementation with respect to the amount of non-local data-transfer. The moving of channel-ends does not involve any data-transfer of elements.

MoCha versus Mailbox Implementation

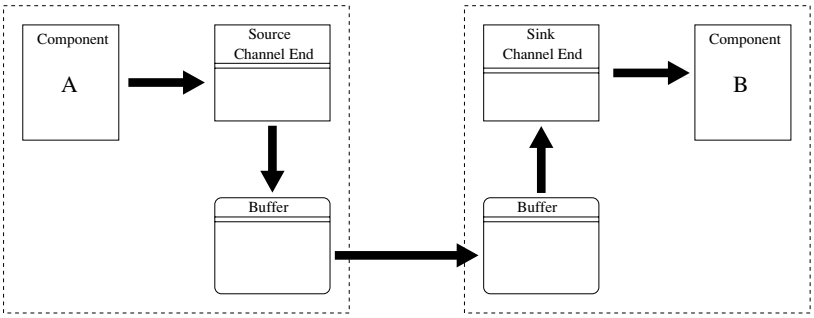


Figure 5.2: A Mailbox Buffer Implementation.

MoCha is also more efficient than a mailbox implementation. Figure 5.2 shows such an implementation. In this implementation, there are always

only two local buffers in a channel; one at the side of the source, and one at the side of the sink. When a channel-end moves, its local buffer moves with it.

In this implementation, a write is a local operation, a read is a local or non-local operation, and moving is quite costly because the elements of the buffer are transfered to a new location; they are all non-local operations.

MoCha versus improved Mailbox Implementation

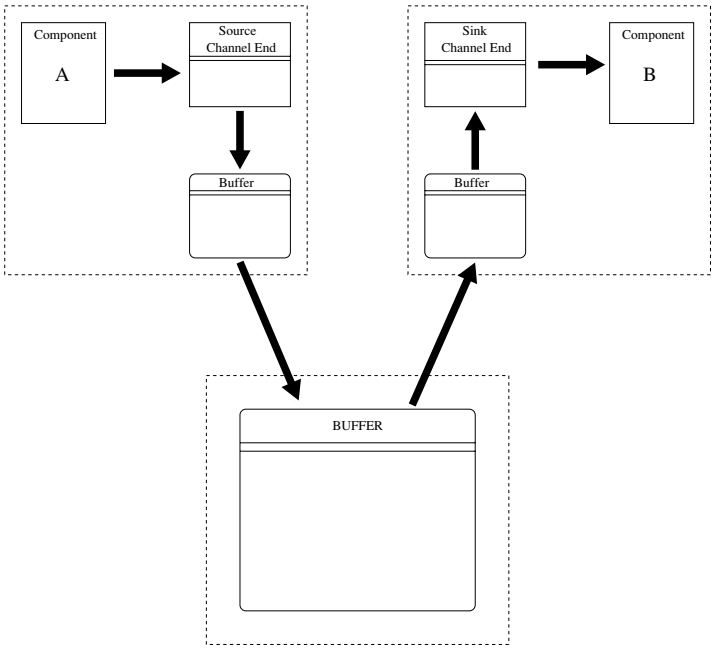


Figure 5.3: An improved Mailbox Buffer Implementation.

An improvement of the mailbox implementation is to combine the two local buffers with a centralized buffer, as shown in figure 5.3. When a channel-end moves, the contents of a local buffer (if any) are transfered to the centralized buffer. This way, not all movements are costly. However, in MoCha all movements are free of data-transfer of elements.

Conclusion

Table 5.1 gives a summary of the comparisons between MoCha and the other models. MoCha is the most efficient model. Without knowing anything of the system structure, it is not possible to do better than MoCha. Therefore,

Model	**write**	**read**	**move**
MoCha	local	local or non-local	free
Cent. Buffer	non-local	non-local	free
Mailbox	local	local or non-local	costly
Impr. Mailbox	local	local or non-local	less costly

Table 5.1: Comparing MoCha with other Implementations

we can conclude that MoCha is an efficient general implementation of an asynchronous mobile FIFO channel[2].

5.2 An Non-trivial Implementation

After reading the previous section, the reader probably agrees that MoCha describes a non-trivial implementation. Nevertheless, we point out two things that make MoCha complex and really non-trivial: the synchronization of the channel-ends of a channel, and the problem of dangling references. Both issues arise due to the movement of the ends of a channel. They are explained in chapter 4, sections 4.2 and 4.3. Therefore, we can safely conclude that MoCha, besides describing an efficient implementation, is also non-trivial.

5.3 Non-distributed systems

Without changes MoCha can also be used for non-distributed systems. In that case, each channel has only one buffer, which the source and the sink use for both reading and writing. No chain of buffers exist, and none is needed.

5.4 Persistence of Channels

Channels in MoCha are not persistent. By *persistence* we mean, that a collection of data remains intact even if its source is no longer attached to the network. For example, the objects in JavaSpaces [8] (see chapter 1, section 1.5.2) remain available to other users even if their sources are disconnected from the network.

[2]Non-FIFO channels may have more efficient implementations. For instance, if we want to implement a set, then a centralized model has some clear advantages.

In distributed systems locations can become (temporarily) unreachable or non-existent, due to, for example, machine crashes or network problems. MoCha does not have a recovery mechanism in case this happens. Therefore, when a location, containing buffers of a channel, gets disconnected from the network, data gets lost in MoCha.

The solution to this problem, is to implement some efficient backup mechanism in MoCha. However, this backup mechanism **does not** affect the efficiency of MoCha in comparison with other models, due to the fact that any other implementation of a distributed mobile channel must also implement a backup mechanism, in order to provide persistent mobile channels.

5.5 Future Work

Implementation in Java

Currently, we are working on an implementation of MoCha in Java. MoCha can easily be implemented in any modern language supporting data-structures, instances of data-structures, pointers/references, distributed networking, and threads.

The Java implementation uses the **R**emote **M**ethod **I**nvocation mechanism for working in a distributed environment. For synchronization, we use the multi-threading mechanism. Although Java does not support semaphores, it does supports monitors. A monitor can be converted into a semaphore if desired.

Up to now, we have encountered no problems with the implementation.

Extensions of MoCha and Rho

Farhad Arbab is developing a programming model and language for coordination of components at CWI (Centrum for Wiskunde en Informatica), called *Rho*[19]. The primitive operations of *Rho* can extend a programming language analogous to Linda [20], with the difference that it is based on mobile channels and not on tuple spaces.

MoCha will be extended in order to support all types of channels described by Rho. A few examples of other types of channels:

- **Synchronized channels**. This is easy to implement in MoCha, we can use the same algorithms but without creating any buffers.
- **Bag channels**.
- **Set channels**.

We are also going to put **conditions** on channels, and **filters** on each channel-end. A filter, in Rho, makes sure that only a specified type of data enters or leaves the channel. For example, a filter can be placed at the source that only allows *integers* to enter the channel and filters all other types of data.

Appendix A

Listing of MoCha Functions

```
Create_Channel (Loc1 , Loc2)
//
// Loc1 = The location where to create the Source_Channel_End.
// Loc2 = The location where to create the Sink_Channel_End.
//
// returns <Source_rf ∈ →Source_Channel_End,
// Sink_rf ∈ →Sink_Channel_End,
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   Source_rf := new Source_Channel_End @ Loc1;
   if ( Source_rf = ERROR ) then
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Sink_rf := new Sink_Channel_End @ Loc2;
   if ( Sink_rf = ERROR ) then
      Delete(Source_rf);
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Source_rf↑.Buffer_rf := NULL;
   Source_rf↑.Wait_Read := FALSE;
   Sink_rf↑.Buffer_rf := NULL;
   Sink_rf↑.Consumed := 0;

   // Knowing each other.
   //
   Source_rf↑.Sink_rf := Sink_rf;
   Sink_rf↑.Source_rf := Source_rf;

   return <Source_rf, Sink_rf, SUCCESS>;
end
```

```
Write (Source_rf, X)
//
// Source_rf ∈ →Source_Channel_End, X ∈ Value.
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
begin
   ERROR := Lock(Source_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;

   if ( Source_rf↑.Buffer_rf = NULL ) then
      // Create first buffer of channel.
      //
      Source_rf↑.Buffer_rf := new Buffer;
      Source_rf↑.Buffer_rf↑.Link_rf := NULL;

      // Sink can not change its buffer field while it is NULL and
      // by locking the move of the source the sink can not move either.
      //
      Lock(Source_rf↑.Move_Lock);
      // Tell the other side.
      //
      Source_rf↑.Sink_rf↑.Buffer_rf := Source_rf↑.Buffer_rf;
      Unlock(Source_rf↑.Move_Lock);
   else
      if ( ¬Local(Source_rf↑.Buffer_rf) ) then
         // Create new local buffer
         //
         TempBuffer_rf := new Buffer;
         TempBuffer_rf↑.Link_rf := NULL;
         Source_rf↑.Buffer_rf↑.Link_rf := TempBuffer_rf;
         Source_rf↑.Buffer_rf := TempBuffer_rf;
      fi
   fi

   // At this point there is always a local buffer.
   //
   Write(Source_rf↑.Buffer_rf, X);
   if ( Source_rf↑.Wait_Read ) then
      // Sink is waiting for values
      //
      Unlock(Source_rf↑.Sink_rf↑.Read_Lock);
      Source_rf↑.Wait_Read := FALSE;
   fi

   Unlock(Source_rf↑.CE_Lock);
   return SUCCESS;
end
```

```
Write (Buffer_rf, X)
//
// Buffer_rf ∈ →Buffer, X ∈ Value.
//
// This function operates on the Buffer data-structure.
//
begin
   Lock(Buffer_rf↑.Buffer_Lock);

   Buffer_rf↑.Vals := Buffer_rf↑.Vals ∘ X;

   Unlock(Buffer_rf↑.Buffer_Lock);
end
```

```
Read (Sink_rf)
//
// Sink_rf ∈ →Sink_Channel_End.
//
// returns <X ∈ Value, Action_Status ∈ {SUCCESS, FAILURE}>
//
begin
   ERROR := Lock(Sink_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;
   Success ∈ {☺,☹} := ☹;
   while ( Success ≠ ☺ ) do
      if ( Sink_rf↑.Buffer_rf ≠ NULL ) then
         if ( ¬Local(Sink_rf↑.Buffer_rf) ) then
            // New local Buffer.
            //
            TempBuffer_rf := new Buffer;
            TempBuffer_rf↑.Link_rf := Sink_rf↑.Buffer_rf;
            Sink_rf↑.Buffer_rf := TempBuffer_rf;
         fi
         // At this point there is always a local buffer.
         //
         // Success indicates whether there is an element to read in the channel.
         //
         <X, Success> := Sink_Read(Sink_rf↑.Buffer_rf, Sink_rf↑.Consumed + 1);
      fi

      // In case that there is no element in the channel "consult" the source
      //
      if ( Success ≠ ☺ ) then
         do
            ERROR := Lock(Sink_rf↑.Source_rf↑.CE_Lock);
         until (¬ERROR)
         if ( Sink_rf↑.Source_rf↑.Buffer_rf ≠ NULL ) ∧
            |Sink_rf↑.Source_rf↑.Buffer_rf↑.Vals| ≠ 0 ) then
            Unlock(Sink_rf↑.Source_rf↑.CE_Lock);
         else
            Sink_rf↑.Source_rf↑.Wait_Read := TRUE;
            Unlock(Sink_rf↑.Source_rf↑.CE_Lock);
            // Read_Lock is initially locked!
            //
            Lock(Sink_rf↑.Read_Lock);
         fi
      fi
   done

   Sink_rf↑.Consumed := Sink_rf↑.Consumed + 1;
   Unlock(Sink_rf↑.CE_Lock);
   return <X, SUCCESS>;
end
```

```
Sink_Read (Buffer_rf, ToBeAsked)
//
// Buffer_rf ∈ →Buffer
//
// ToBeAsked ∈ Integer, number of values to be asked
// to the next buffer in case Buffer_rf↑ is empty.
//
// returns <X ∈ Value, Success ∈ {☺, ☹} >
//
// This function operates on the Buffer data-structure.
//
begin
   Lock(Buffer_rf↑.Buffer_Lock);

   V := < >; // a sequence of Value.
   Reference := Buffer_rf↑.Link_rf;

   // Execute loop while buffer is empty and
   // there is (still) a reference to another buffer.
   //
   while ( (|Buffer_rf↑.Vals| = 0) ∧ (Reference ≠ NULL) ) do
      // Buffer_Read returns, besides V, a reference to a
      // next buffer in the chain (if any exists) or NULL (if not).
      //
      <V, Reference> := Buffer_Read(Buffer_rf↑.Link_rf, ToBeAsked);
      if ( Reference ≠ NULL ) then
         Buffer_rf↑.Link_rf := Reference;
      fi
      Buffer_rf↑.Vals := V;
   done

   if (|Buffer_rf↑.Vals| > 0 ) then
      X := Head(Buffer_rf↑.Vals);
      Buffer_rf↑.Vals := Tail(Buffer_rf↑.Vals);
      Success := ☺;
   else
      // No elements found in channel.
      //
      X := UNDEFINED;
      Success := ☹;
   fi

   Unlock(Buffer_rf↑.Buffer_Lock);

   return <X, Success>;
end
```

```
Buffer_Read (Buffer_rf, Amount)
//
// Buffer_rf ∈ →Buffer
//
// Amount ∈ Integer, number of values requested.
//
// returns <V ∈ Value*, Next_Buffer_rf ∈ →Buffer>
//
// This function operates on the Buffer data-structure.
//
begin
   Lock(Buffer_rf↑.Buffer_Lock);

   V := < >;
   Destroy := FALSE;

   if ( |Buffer_rf↑.Vals| ≤ Amount ) then
      V := Buffer_rf↑.Vals;
      Buffer_rf↑.Vals := < >;
      Next_Buffer_rf := Buffer_rf↑.Link_rf;
      if ( Buffer_rf↑.Link_rf ≠ NULL ) then Destroy := TRUE; fi
   else
      for 1 to Amount do
         V := V ∘ Head(Buffer_rf↑.Vals);
         Buffer_rf↑.Vals := Tail(Buffer_rf↑.Vals);
      od
      Next_Buffer_rf := NULL;
   fi

   // Unlocking before destroying would make
   // the algorithm unstable.
   //
   if ( Destroy ) then
      Delete(Buffer_rf);
   else
      Unlock(Buffer_rf↑.Buffer_Lock);
   fi

   return <V, Next_Buffer_rf>;
end
```

```
Move (Channel_End_rf, Target)
//
// Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End}
//
// Target = The location target.
//
// returns <New_Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End},
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   ERROR := Lock(Channel_End_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   if ( Channel_End_rf↑ ∈ Sink_Channel_End ) then
      // Lock Sink first, then Source.
      // There is no Unlock for the following Lock.
      //
      Lock(Channel_End_rf↑.Move_Lock);
      Lock(Channel_End_rf↑.Source_rf↑.Move_Lock);

      New_Channel_End_rf := new Sink_Channel_End @ Target;
      if ( New_Channel_End_rf = ERROR ) then
         Unlock(Channel_End_rf↑.Source_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.CE_Lock);
         return <UNDEFINED, FAILURE>;
      fi

      Copy_Information(Channel_End_rf, New_Channel_End_rf);
      New_Channel_End_rf↑.Consumed := 0;
      Channel_End_rf↑.Source_rf↑.Sink_rf := New_Channel_End_rf;
      Delete(Channel_End_rf);

      Unlock(New_Channel_End_rf↑.Source_rf↑.Move_Lock);

   else
```

```
      // Lock Sink first, then Source.
      //
      do
          ERROR := Lock(Channel_End_rf↑.Sink_rf↑.Move_Lock);
      until ( ¬ERROR )

      // There is no Unlock for the following Lock.
      //
      Lock(Channel_End_rf↑.Move_Lock);
      New_Channel_End_rf := new Source_Channel_End @ Target;
      if ( New_Channel_End_rf = ERROR ) then
          Unlock(Channel_End_rf↑.Move_Lock);
          Unlock(Channel_End_rf↑.Sink_rf↑.Move_Lock);
          Unlock(Channel_End_rf↑.CE_Lock);
          return <UNDEFINED, FAILURE>;
      fi
      Copy_Information(Channel_End_rf, New_Channel_End_rf);
      Channel_End_rf↑.Sink_rf↑.Source := New_Channel_End_rf;
      Delete(Channel_End_rf);

      Unlock(New_Channel_End_rf↑.Sink_rf↑.Move_Lock);
   fi
   return <New_Channel_End_rf, SUCCESS>;
end
```

```
Destroy_Channel (Channel_End_rf)
//
// Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End}
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
// There is no Unlock in this function
//
begin

   if ( Channel_End_rf↑ ∈ Sink_Channel_End ) then
      Sink_rf := Channel_End_rf;
   else
      // Check for dangling reference
      //
      ERROR := Lock(Channel_End_rf↑.CE_Lock);
      if ( ERROR ) then return FAILURE;
      Sink_rf := Channel_End_rf↑.Sink_rf;
      Unlock(Channel_End_rf↑.CE_Lock);
   fi

   // At this point Sink_rf ∈ Sink_Channel_End,
   // but needs not be valid anymore due to possible
   // movement of sink channel-end.

   // Lock Sink first, then Source.
   //
   ERROR := Lock(Sink_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;

   do
      ERROR := Lock(Sink_rf↑.Source_rf↑.CE_Lock);
   until ( ¬ERROR )

   Source_rf := Sink_rf↑.Source_rf;

   // Delete chain of buffers.
   //
   i := Sink_rf↑.Buffer_rf;
   While ( i ≠ NULL ) do
      j := i↑.Link_ref;
      Delete(i);
      i := j;
   done

   Delete(Source_rf);
   Delete(Sink_rf);
   return SUCCESS;
end
```

Bibliography

[1] F. Arbab, F. S. de Boer, and M. M. Bonsangue. *A Logical Interface Description Language for Components.* Proceedings of Coordination 2000, Lecture Notes in Computer Science, Springer, 2000.

[2] F. Arbab, M. M. Bonsangue, and F. S. de Boer. *A Coordination Language for Mobile Components.* Proceedings of the 2000 ACM Symposium on Applied Computing (SAC 2000), pp 166-173, ACM, 2000.

[3] G. A. Papadopoulos and F. Arbab. *Dynamic Reconfiguration in Coordination Languages.* Proc. of the 8th International Conf. on High Performance Computing and Networking, HPCN Europe 2000, Lecture Notes in Computer Science, vol. 1823, pp.197-206, Springer, 2000.

[4] G. Andrews, *Paradigms for process interaction in distributed programs*, ACM Computing Surveys, 23(1):49-90, 1991.

[5] Sun, *Java Message Service, Specification Document version 1.0.2* , Sun Microsystems Inc., Palo Alto (USA), November 1999.

[6] Sun, *Java Message Queue, Quickstart Guide v1.1*, Sun Microsystems Inc., Palo Alto (USA), May 2000.

[7] Home Page of JMQ (documentation), *http://docs.iplanet.com/docs/manuals/javamq.html.*

[8] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces TM Principles, Patterns, and Practice*, Chapter 1 of book, Addison-Wesley, September 1999.

[9] Home Page of JavaSpaces, *http://java.sun.com/products/javaspaces/.*

[10] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, London, UK, 1985.

[11] G. Hilderink, J. Broenink, and A. Bakkers. *Communicating Threads for Java*, Draft version available at www.rt.el.utwente.nl/javapp/information/CTJ/main.html, The Netherlands, 2000.

[12] Home Page of CTJ, *http://www.rt.el.utwente.nl/javapp/information/CTJ/main.html.*

[13] P. Welch, *CSP for Java (What, Why, and How Much?)*, Slides of Seminar, University of Kent at Canterbury, 2001.

[14] Home Page of JCSP, *http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.*

[15] B. C. Pierce, D. N. Turner, *Pict: A Programming Language Based on the Pi-calculus*, Technical report, Computer Science Department, Indiana University, 1997.

[16] B. C. Pierce, *Programming in the Pi-calculus, A Tutorial Introduction to Pict*, Tutorial, Computer Science Department, Indiana University, 1998.

[17] Home Page of Pict, *http://www.cis.upenn.edu/ bcpierce/papers/pict/Html/Pict.html.*

[18] Home Page of Java, *http://java.sun.com.*

[19] Farhad Arbab, *Private Correspondence*, farhad@cwi.nl .

[20] N. Carriero, D. Gelernter. *How to Write Parallel Programs: a First Course*, MIT press, 1990.