Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

MoCha: a framework for coordination using mobile channels

F. Arbab, F.S. de Boer, M.M. Bonsangue,
J.V. Guillen Scholten

REPORT SEN-R0128 DECEMBER 31, 2001

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 Framework for Coordination Using Mobile Channels

Farhad Arbab[1]
Frank S. de Boer[1]
Marcello M. Bonsangue[2]
Juan V. Guillen Scholten[1]

[1] *CWI*
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
E-mails: farhad@cwi.nl, frb@cwi.nl, and juan@cwi.nl

[2] *LIACS, Leiden University*
P.O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

ABSTRACT

In this paper we describe MoCha, an infrastructure for distributed communication and collaboration using mobile channels as its medium. Channels allow directed, anonymous, and peer-to-peer communication among entities, while mobility ensures that the structure of their connections can change over time in arbitrary ways. MoCha provides a distributed middle-ware designed for communication and collaboration without requiring central servers or fixed network infrastructures.

2000 ACM Computing Classification: 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, middle-ware, distributed communication, dynamic distributed systems, implementation of mobile channels, network architectures, mobility
Note: Work carried out under the project SEN3.2 "Component-based models and software architectures"

Contents

1 Introduction

In many network architectures, each process on the network is either a client or a server: servers are processes dedicated to specific tasks like managing of disk drives, printers, or network traffic, whereas clients are processes that rely on servers for resources.

In contrast to this hierarchical architecture, in a peer-to-peer architecture each node has equivalent responsibilities, enabling applications that focus on collaboration and communication. Features of a peer-to-peer architecture include a better distributed network control, high availability through the existence of multiple peers in a group, and the possibility of dynamic exchange of information about the network topology, which enables easier connections and reconfiguration.

The flexibility of peer-to-peer network architectures is increased by infrastructures that allow for the distribution of processes across several heterogeneous platforms and operating systems, and that enable applications to exchange messages with other applications without having to know *where* in the network those other applications reside, *who* produces and consumes the exchanged messages, and *when* a particular message was produced or will be consumed. In this paper, we discuss MoCha, an infrastructure for distributed communicating and collaborating processes. In MoCha communication among the active entities that comprise an application is **asynchronous**, **anonymous**, and **dynamic**, allowing the communicating entities to have completely separate lifetimes and encouraging loose coupling among message consumers and message producers. MoCha provides an application program interface to handle channels that connect inter-operating processes. A channel is a temporary storage with a unique identity of its own and two distinct directed ends: source and sink (see figure 1). A process inserts data at the source of the channel without having to know the process connected at its sink. A channel delivers its messages to the process connected to its sink, holding the messages if the process is busy or not connected. The messages can contain formatted data, requests for action, or channel sink/source identities. The latter type of message is used to reconfigure the network topology dynamically, a crucial property in systems where applications themselves can be mobile.

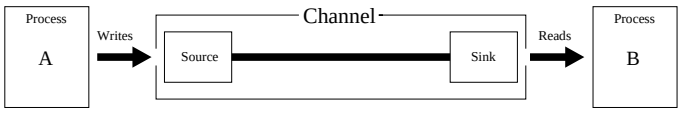

Figure 1: A Channel.

The significance of peer-to-peer architectures has motivated other research groups to propose tools to support them. For instance, the JXTA project [14] provides a set of six protocols that have been designed for *ad hoc*, pervasive, and multi-hop peer-to-peer network computing. The JXTA protocol most closely related to MoCha is the *Pipe Binding Protocol*. In contrast to MoCha channels, pipes provide the illusion of a virtual in and out mailbox that is independent of any single peer location, and network topology (multi-hops route).

MoCha is most appropriate for component-based applications because it furnishes a conceptual mechanism for anonymous communication: when a message is sent, the component hands off the responsibility of delivery to the MoCha middle-ware. Other communication models used for coordination in network architectures and component-based applications include:

- *Synchronous communication* via remote procedure calls, remote method invocations, or object request brokers. The use of synchronous communication requires that the producer and the consumer are always available and functioning. Applications are tightly coupled both in space and time. This has the advantage of preventing overloading a network, but it increases the possibility of deadlocks. Mechanisms like exception messages, timed request, or retransmissions are

used for deadlock avoidance and recovery. Examples of middle-ware based on synchronous communication include remote procedure calls within the Distributed Computing Environment [6] and Remote Method Invocations in Java [4].

- *Message-passing*, via event broadcast, queues, and publish/subscribe storage. A message is an asynchronous request, report or event that is either addressed to a destination or is associated to a topic to which application may subscribe in order to receive it. The receivers of addressed messages need not know anything about their senders, and likewise, the senders of messages labeled with a topic need not know anything about their eventual receivers. Thus applications are loosely coupled in space and time, but not completely. Examples of message-passing based middle-ware include Java Message Queue [17] and Microsoft Message Queuing Services [9] for COM+ [12].

- *Data-oriented communication*, via shared data spaces, blackboards, and shared variables. Applications communicate by inserting data into and retrieving data from a common persistent storage. Data is typically not targeted, so that applications are fully decoupled in space and time, thus leading to simpler, more flexible, and reliable systems. JavaSpaces [7], based on the data-oriented coordination model of Linda [5], is a Jini [3] service for creating collaborative and distributed applications in Java.

MoCha is in many respects based on the coordination model of Manifold [1] and can be thought of as an alternative to JavaSpaces that is based on the primitive notion of channels rather than shared data spaces. MoCha shares many of the same architectural strengths of JavaSpaces while offering some additional advantages. Three advantages of MoCha in comparison with JavaSpaces are: *efficiency*, *security*, and *architectural expressiveness*. Although JavaSpaces are useful in network architectures like blackboard systems, for most networks the peer-to-peer aspect of MoCha is more *efficient*. With JavaSpaces, any process is able to read any data on the centralized space, making it necessary to use security tools, for example, encryption. Using channels, the data can be read only by the processes that have access to their *sinks* and no third party can intercept. Finally, channels make MoCha architectural more expressive than shared data spaces. With a shared data any process can, in principle, exchange data with any other process. This makes it impossible for shared data spaces to reflect data dependencies and communication patterns within an application. Using channels, it is easy to see which processes exchange data with each other, making it easier to apply tools for analysis of the dependencies and data-flow.

There are several implementations of channels, e.g., *Communicating Threads for Java* [10], and *CSP for Java* [19], both based on the *CSP* model [11], and *Pict* [15], a concurrent programming language based on the π-calculus. However, these implementations either do not provide mobile channels, or do not support distributed environments. Up to now no (efficient) distributed implementation of mobile channels has been presented in the literature. The major contribution of this paper is to present the implementation of the *Mo*bile *Cha*nnels of the MoCha infrastructure, which are asynchronous FIFO channels. Such an implementation is far from trivial if non-local data-transfers are to be minimized. The MoCha infrastructure can easily be implemented in any programming language, like Java [13] and C++ [16], that supports data-structures, instances of data-structures, pointers/references, distributed networking, and threads.

In the next section we present the general concept of MoCha, while in section 3 we discuss more deeply the main operations on mobile channels. To simplify our presentation, we proceed first by considering channels as static entities that will not be reconnected, and later explain the consequences of channel mobility in the semantics of MoCha operations. We conclude in section 4 by comparing MoCha with other possible "natural" implementations of mobile channels.

2 The MoCha framework

The MoCha framework describes an **efficient** and **non-trivial** implementation of an asynchronous FIFO mobile channel in a distributed environment. It consists of a collection of functions that can be executed by any process in a network. These functions are the interfaces of abstract algorithms that cover all major operations on a channel: *Create Channel*, *Write*, *Read*, *Move*, and *Destroy Channel*.

Initially, a network consists of a finite number of active entities connected by some channels. These entities communicate with each other only by writing data into and reading data from channels. Thus, the two communicating parties do not need to know each other. The configuration of a network is highly dynamic: new entities can be created, new channels can connect existing entities, and existing channels can be moved and reconnected to other entities.

In the next section we explain the implementation concepts of the operations in MoCha. Naturally, concurrency implies some *locking* and *unlocking* of semaphores for internal synchronization of MoCha. We abstract from this technical detail in this paper because it is not important for the functionality of MoCha operations at the level described here. This technical detail can be found in [8].

In a distributed environment, data-transfer between different locations is more costly with respect to time than data-transfer within a location. Therefore, an efficient implementation must reduce the amount of non-local data-transfer. Trivial implementations of mobile channels in a distributed environment, such as a centralized buffer for which (virtually) every *read* and *write* operation is a non-local one, clearly do not meet this demand. MoCha reduces non-local data-transfers by allowing the distribution of data all over the system, and avoiding any kind of centralized control. The movement of data is minimized by making sure that (1) every *write* operation is local, (2) most *read* operations are local, and (3) movement of channel-ends does not involve movement of data. For comparisons of MoCha with this centralized implementation and others see section 4.

3 Mobile Channels

We start by presenting the realization of a mobile channel in MoCha, and continue with explaining the major operations on channels. For simplicity, after *Create Channel* we first explain *write* and *read* ignoring mobility. Then we introduce *Move*, and refine the semantics of the previous operations in the presence of mobility.

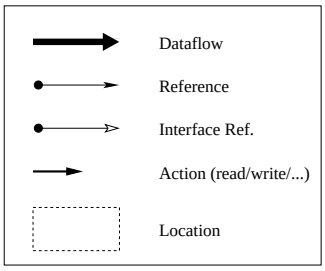

Figure 2: Legend.

Figure 2 shows the meaning of the symbols used in the rest of the figures in this paper. The references used by MoCha either point to local or to remote entities, denoted by thin arrows within a location or crossing its border, respectively. By location we mean a *logical address space* where processes, threads, objects, or components execute. In the sequel, we use the term *component* to indicate some such entity.

An *interface reference* is a reference from a component to a channel-end, restricting the access of the component to only the pre-defined operations on the channel. For simplicity, we assume that

only one component can have access to a particular channel-end at a given time (as in figure 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.

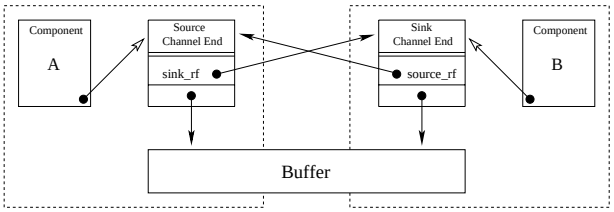

Figure 3: A mobile channel in MoCha.

In section 1, figure 1 gives a general view of a channel. In figure 3 we show how a mobile channel is realized in MoCha. For the components, a channel consists of two ends, its *source* and its *sink*, which they refer to through interface references. A component writes values into a channel by performing the write operation of its *source*-end and reads from it by performing the read operation of its *sink*-end. Communication though mobile channels is **anonymous** because communicating components need to know only their own respective channel-ends, not each other. However, the ends of a channel must internally know each other to keep the identity of the channel and control communication. For this purpose, the ends of a channel have references to each other (the *sink_rf*- and *source_rf*-fields in the figure). The communication, besides being anonymous, is also **asynchronous**. That is why the channel-ends refer to buffers. The implementation of a MoCha buffer consists of a chain of local FIFO buffers that are distributed over the system. The local buffers, and other fields, are gradually introduced in our explanation.

3.1 Create channel

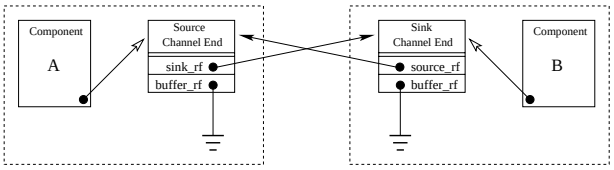

Figure 4: Channel Creation.

Upon creation of a new channel, its channel-ends (its *sink* and its *source*) are created at two given locations, which need not be distinct. A reference to each channel-end is then returned. We sometimes say "a component holds a channel-end" when it has access to this channel-end.

An example of creation is shown in figure 4. 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 each other), and the *buffer_rf*-fields are set to `NULL`. The reason for doing this is that there is no guarantee that the component, that initially holds a channel-end, actually reads or writes any data before the channel-end is moved to another component. Therefore, creating a buffer is done only when really needed, which is more efficient (examples follow).

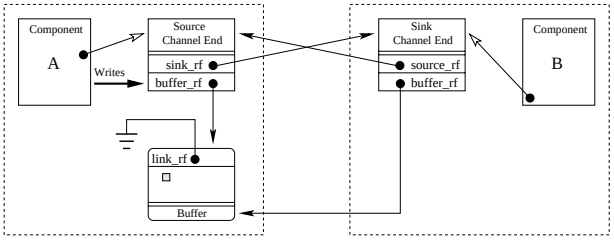

Figure 5: First Write Action.

3.2 Write (Without Mobility)

In figure 5, component A starts to write for the first time. Because its *buffer_rf*-field is `NULL` the source creates a local unbounded FIFO buffer. The buffer has a field called *link_rf*, which is a reference to another buffer. Because there is just one buffer now, this field is set to `NULL`.

For its internal consistency, MoCha requires an ***invariant*** to hold on the *buffer_rf*-fields of the two ends of a channel: they must be either both `NULL` or both `non-NULL`. In figure 4 the channel is just created and the *buffer_rf*-fields are, initially, `NULL`. Therefore, after the creation of the first buffer of the channel, they must be both `non-NULL`. The source notifies the sink about the new buffer, and 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 action, there is always a local buffer at the same location as the source.

3.3 Read (Without Mobility)

In figure 4 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 a first write.

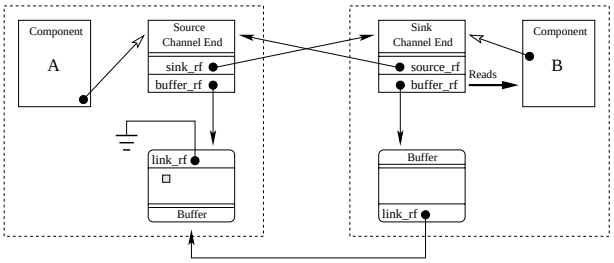

Figure 6: First Read Action.

In figure 5 the *buffer_rf*-field of the sink is not `NULL` anymore, but it points to a non-local buffer. If 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 points to), and the *buffer_rf*-field is updated to point to the new local buffer. The result is shown in figure 6. If A and B are at the same location, then no new local buffer is created, because the *buffer_rf*-field of the sink already points to a local one.

The actual reading of the elements can now begin, we explain this in section 3.8. After the first read action, there is always a local buffer at the same location as the sink.

3.4 Mobility

Channel-ends are mobile. References to them can be passed on to other components in the network via channels. This allows dynamic reconfiguration of channel connections among the components of a system.

Channel-ends can also physically move to other locations by performing the operation *move*. When a channel-end moves, the component at the other end of the channel does not notice anything.

When a channel-end moves to a component in a different location, its *buffer_rf*-field reference changes from local to non-local. Because a channel-end can move among several components before one of them actually uses it, no buffers are moved or created in MoCha. A buffer is created in MoCha only when necessary: when a component actually starts to read or write. After the movement, a channel-end notifies the other end of the channel of its new location. Examples of movement follow.

3.5 Write (with mobility)

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 its newly created local buffer, see section 3.2.

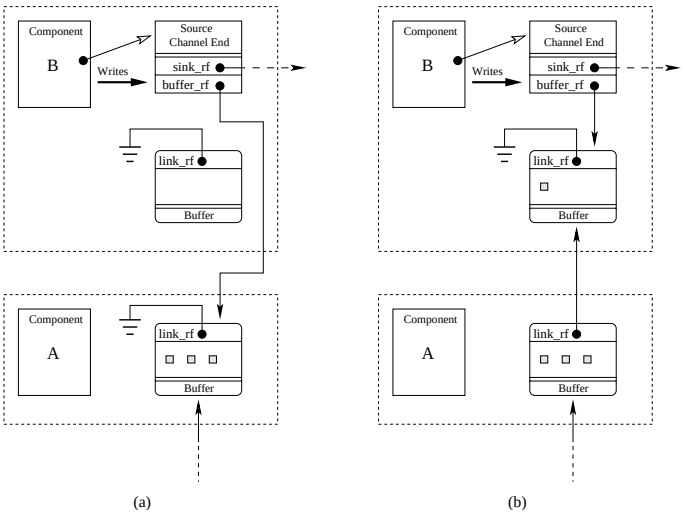

Figure 7: Writing after Movement of Source.

Figure 7 is an example of writing after the source end of a channel is moved to a new location. At some point in time component A has the source channel-end and writes some elements into the channel. Then the channel-end moves to component B in another location, at which time no buffer is created or moved. Then component B starts to write (figure 7a). The *buffer_rf*-reference of the source is non-local, therefore a new local buffer is created. Both the *buffer_rf*-field, and the *link_rf*-field of the old buffer (the buffer where the *buffer_rf*-field is points to) are changed to point to the new buffer. The result is figure 7b. The data written by component B can now be inserted into this new local buffer.

3.6 Read (with mobility)

When reading, a local buffer is created at the location of the sink if its *buffer_rf*-field points to a non-local buffer. If the *buffer_rf*-field points to a local buffer, then reading can proceed. If it is `NULL`, thin no elements exist in the channel and the sink waits (see section 3.3).

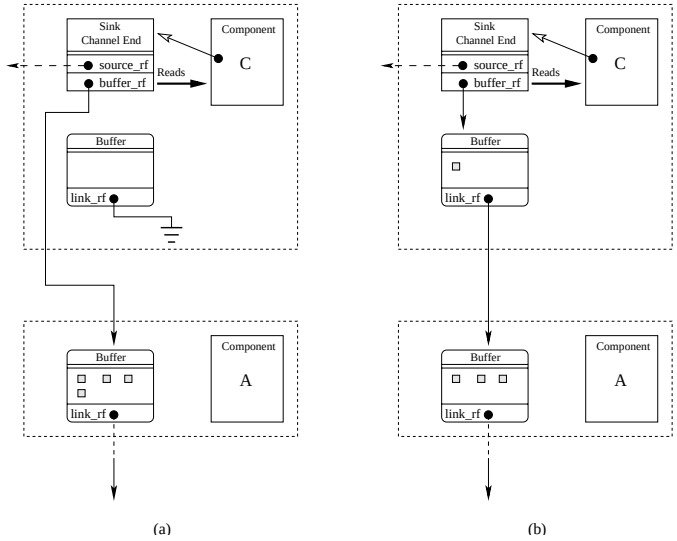

Figure 8: Reading after Movement of Sink.

Figure 8 shows an example of reading after the sink end of a channel is moved to a new location. Component C holds the sink channel-end right after the move, the sink moved. In figure 8a 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 the new buffer is changed to point to the old buffer (the buffer where the *buffer_rf*-field points to), and the *buffer_rf*-field is changed to point to the new buffer. The result is shown figure 8b. The reading can now begin. More details on reading is given in section 3.8.

3.7 Chain of local 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 9 shows a general case. The components need not be distinct 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 oldest elements inserted in the channel. The *last* buffer is the one where the *buffer_rf*-field of the source points to, it contains the most recent elements inserted in the channel. In this way the FIFO structure of the channel is preserved.

3.8 More about reading

We showed how local buffers are created as a consequence of reading and writing. For writing this is sufficient because this action just creates local buffers, sets ups the references, and fills them with elements. For reading, however, we still need to explain how elements are moved through the chain of buffers, how this can be improved by means of heuristics, and how local buffers become empty and are destroyed.

3.8.1 How data elements move

The buffers in MoCha support additional functionality beyond the typical FIFO buffers. A buffer is either a *sink-buffer*, or a normal MoCha buffer. A *sink-buffer* is one that is **local** to the sink-end of

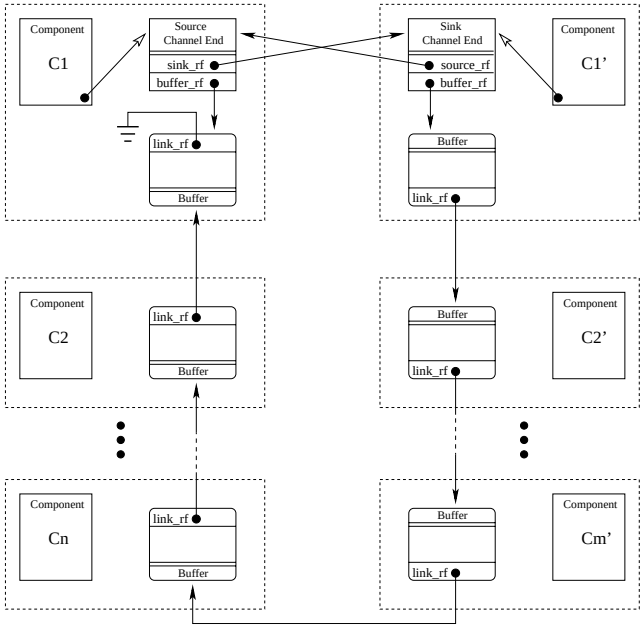

Figure 9: Chain of Buffers.

a channel. A channel can have at most one *sink-buffer* at any given time, but any buffer in its chain can become its *sink-buffer* at some point in time.

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. If it is empty, it asks the next buffer in the chain (if any) for a certain number of elements. This number is determined by the reading heuristics explained in the next subsection. The *sink-buffer* receives from this next buffer either (1) the requested number, (2) fewer or, (3) no elements. The *sink-buffer* also receives a reference that is either:

- `NULL`, which indicates that the buffer is either not empty after it provided these elements, or it is the last buffer of the chain. Receiving a `NULL` reference, the *sink-buffer* does not modify its *link_rf*-field.
- `none-NULL`, which indicates that the buffer is now (or was) empty. The reference points to the next buffer in the chain, the *sink-buffer* receiving a `non-NULL` reference changes its *link_rf*-field to point to this new value.

3.8.2 Reading heuristics

For efficiency, sink-buffers use certain heuristics to determine the number of elements they request from their next buffers in their chains. For this purpose, the sink channel-end has an extra field called `consumed`, that keeps track of how many data elements the current component holding the sink has already consumed. Initially, and every time the channel-end moves, this variable is set to zero.

Knowing nothing of the behavior of the system it is reasonable to assume that the component performing a read is going to consume (in total) **twice** the amount it has already consumed. This is similar to a good heuristic rule used in dynamic memory management. By this heuristic, the number of elements to be requested is `consumed + 1`.

Figure 10 shows an example. Component C holds the sink channel-end and has already consumed 3 elements. It now asks for another element. The sink-buffer is empty, therefore it requests elements

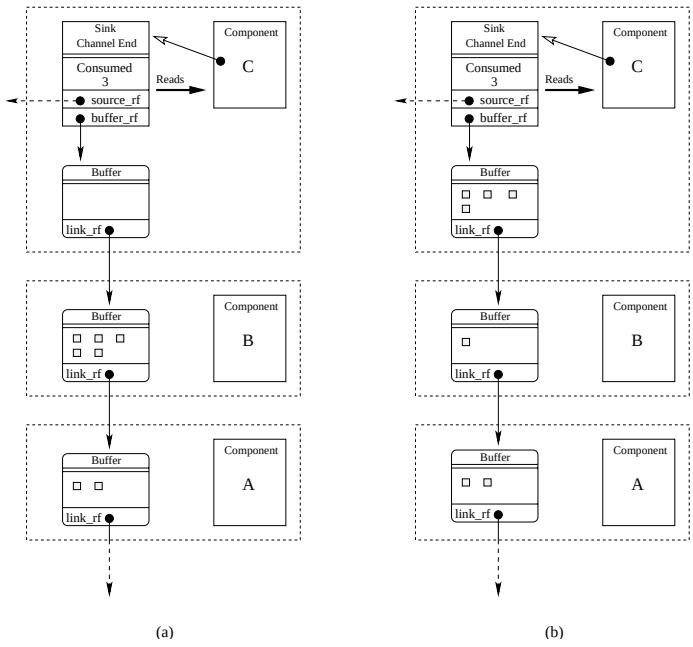

Figure 10: Heuristic of Sink Buffer.

from the next buffer in the chain. We assume that the component is going to consume a total of 7 elements (2∗3 + 1), therefore the sink-buffer requests 4 elements (`consumed + 1`). In figure 10b the 4 elements of the non-local buffer are transfered to the sink-buffer. An element is then given to the component by the sink, which is not shown in the figure.

The rationale for asking for more data elements than needed (just 1) is the fact that the cost of communication in a distributed system is not really dependent on the amount of transfered data, but rather, on the number of exchanged messages. Consequently, within reasonable bounds, it cost the same to exchange a short message as to exchange a long one. Furthermore, the majority of the exchanged messages in a system are well below those "reasonable bounds" and leave some considerable amount of free bandwidth that can be used to transfer additional data at no extra cost.

Asking for more data items than it needs, the *sink-buffer* informs the remote buffer of the likelihood that it will consume those additional elements in the near future. The remote buffer, then, determines the maximum number of data items, up to the request amount, that fit in the free bandwidth of one message and packs and sends at most that many, if it indeed has that many.

Observe that it makes no sense for the remote buffer to try to send more data elements that it actually contains, by obtaining them from its next buffer in the chain. Also, it is not necessarily a good idea for the remote buffer to ignore the (heuristically determined) number of requested data items and always "flush" itself sending its entire contents to the *sink-buffer*, because the sink may move to the same location as the remote buffer before all those elements are consumed.

3.9 Destruction of buffers

A buffer can become empty when elements are read out of it. An empty buffer is no longer of use, unless it is the first or the 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 of the chain (in which case its *link_rf*-reference is `NULL`). The behavior of the buffers is described in section 3.8.1.

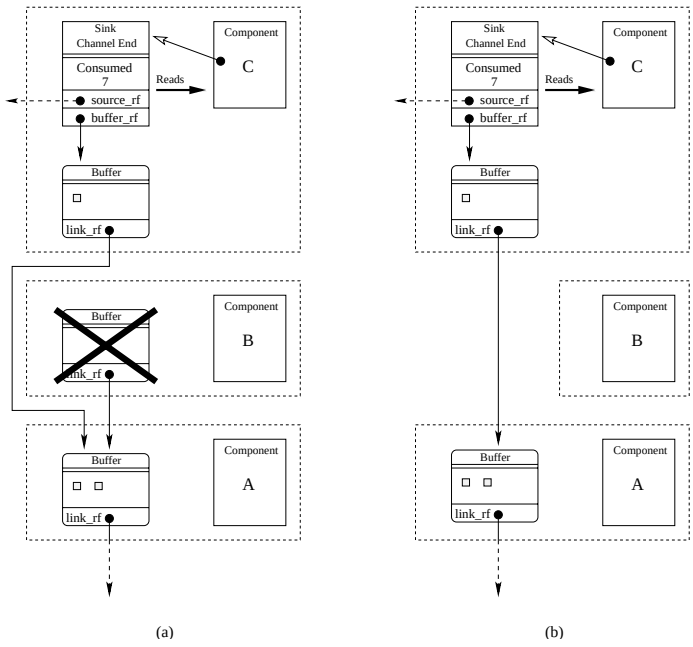

Figure 11: Destruction of Buffers.

In figure 11, a buffer has become in a read operation. The buffer detects that and gives the sink-buffer not only the requested data elements but also a reference to its next buffer in the chain. The sink-buffer updates its *link_rf*-field to this next buffer. The empty buffer, having passed its next-buffer reference to the *sink-buffer*, concludes that it can safely destroy itself because it is no longer part of the chain.

4 Conclusion and future work

In this paper we presented MoCha, an infrastructure for network systems based on mobile channels. MoCha describes an efficient distributed implementation of an asynchronous FIFO mobile channel because it minimizes the amount of non-local data-transfers, which are far more costly than the local ones. MoCha does this by making sure that:

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

We compare MoCha to three alternative models for implementation of mobile channels: the *centralized buffer*, the *mailbox*, and the *improved mailbox*. In the first implementation model the source and the sink channel-ends write and read data to/from a centralized remote buffer, making virtually every read and write non-local. In the mailbox implementation (see figure 12) 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 buffer moves with it, making this, on the average, a costly operation because some of the data movements are in vain. An improvement to the mailbox implementation is to move

the buffers only when a component, holding the channel, actually starts to read or to write. This way, only the first use of a channel-end after it has been moved is costly. Table 1 gives a summary of the comparisons between MoCha and the other models. MoCha is the most efficient model. Moreover, without specific knowledge about the behavior of a system, it is unclear if it is possible to do better than MoCha. Observe that any "improved" heuristics for the prediction of the behavior of the system can easily replace our current heuristics.

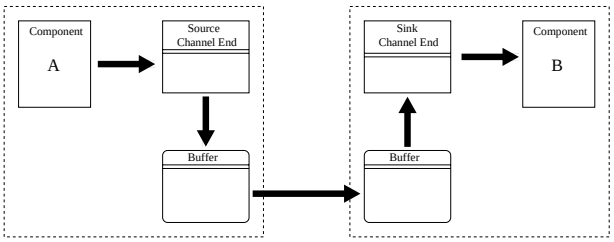

Figure 12: A Mailbox Buffer Implementation.

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 first use after move is costly	local or non-local first use after move is costly	free

Table 1: Comparing MoCha with other Implementations

Currently, we are finishing an implementation of MoCha in Java. Test results will be presented in future work. Besides Java, due to the assumptions we made for the algorithms, MoCha can easily be implemented in any modern language that supports data-structures, instances of data-structures, pointers/references, distributed networking, and threads. For example, C++.

MoCha is the basis for the implementation of a paradigm for composition of software components based on the notion of mobile channels [2]. MoCha is going to be extended in order to support all types of mobile channels described by this paradigm, including the addition of **filters** on these channels. A filter makes sure that only a specified type of data leaves the channel, all other kind of data inserted into the channel gets lost. For example, a filter can be placed on the channel that only allows *integers* to leave the channel and filters all other types of data.

References

[1] F. Arbab, *Manifold Version 2: Language Reference Manual*, Technical Report, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1996. Available online at the URL: `http://www.cwi.nl/ftp/manifold/refman.ps.Z`.

[2] F. Arbab, *Coordination of Mobile Components*, Electronic Notes in Theoretical Computer Science Vol 54, Elsevier Science B.V., 2001.

[3] K. Arnold, B. O' sullivan, R.W. Scheifler, J. Waldo, A. Wollrath, *The Jini^{TM} Specification*, Addison-Wesley, 1999.

[4] M. Campione, K. Walrath, A. Huml, et al. *The Java^{TM} Tutorial Continued: The Rest of the JDK^{TM}*, Addison-Wesley, 1998.

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

[6] D. Fauth and C. Shue, *Remote Procedure Call: Technology, Standardization and OSF'S Distributed Computing Environment*, Report OSF-O-WP10-1090-2, Open Software Foundation Inc., 1991.

[7] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces^{TM} Principles, Patterns, and Practice*, Addison-Wesley, 1999.

[8] J.V. Guillen Scholten, *MoCha, a Model for Distributed Mobile Channels*, Internal Report 01-07, Master's Thesis, LIACS, Leiden University, May 2001.

[9] Dan Hay, *COM+ Technical Series: Queued Components*, Microsoft Corporation, 1999.

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

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

[12] Microsoft Corporation. Home page of COM+, `http://www.microsoft.com/com/tech/complus.asp`.

[13] Sun Microsystem Inc., Home Page of Java, `http://java.sun.com`.

[14] Sun Microsystem Inc., Home Page of the JXTA project, `http://www.jxta.org`.

[15] B. C. Pierce, D. N. Turner, *Pict: A Programming Language Based on the Pi-calculus*, Technical report, Computer Science Department, Indiana University, 1997. Home Page of Pict, `http://www.cis.upenn.edu/ bcpierce/papers/pict/Html/Pict.html`.

[16] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.

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

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

[19] P. Welch, *CSP for Java (What, Why, and How Much?)*, Slides of Seminar, University of Kent at Canterbury, 2001. Home Page of JCSP, `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.