Coordinated Anonymous Peer-to-Peer Connections with MoCha

Juan Guillen-Scholten and Farhad Arbab

Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands
{juan,farhad}@cwi.nl

Abstract. MoCha is an exogenous coordination middleware for distributed communication based on mobile channels. Channels allow anonymous, and point-to-point communication among nodes, while mobility ensures that the structure of their connections can change over time in arbitrary ways. MoCha is implemented in the *Java* language using the *Remote Method Invocation package* (RMI) [15]. In this paper we promote the use of mobile channels for P2P applications and show the benefits of the MoCha middleware.

1 Introduction

Today, a big percentage of the Internet traffic is generated by file-sharing applications. Most applications of this kind are based on a so called peer-to-peer (P2P) network. P2P networking refers to a class of systems, applications and architectures that employ distributed resources to perform any kind of task in a decentralized and self organizing way[11]. The popularity of P2P networks originates from the introduction of the Napster[5] application in the year 2000 and it is continued by many other P2P file-sharing applications like Kazaa[12], BitTorrent[4], and many clients of the Gnutella network[9].

P2P networks are often put in contrast with *client/server networks*. That is because 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. The clients themselves do not share any resources. In a *peer-to-peer* architecture each node is both a client and a server at the same time. Therefore, the nodes are said to be *equal*. They have equivalent responsibilities, enabling applications that focus on collaboration and communication in a decentralized and self organizing way. 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.

The flexibility of P2P network architectures is increased by infrastructures that (1) allow making connections between distributed nodes across several heterogeneous platforms and operating systems, that (2) enable nodes to establish anonymous connections between them, that (3) provide some kind of mechanism

N. Guelfi et al. (Eds.): FIDJI 2004, LNCS 3409, pp. 68–77, 2005.
© Springer-Verlag Berlin Heidelberg 2005

for easy dynamic reconfiguration of the network topology, that (4) provide exogenous coordination by letting the creator of the connection choose between a synchronous or an asynchronous type, and that (5) offer a clear and easy high-level API for P2P applications.

Such an infrastructure is the MoCha middleware[6]. In this paper we show and discuss the advantages of using MoCha for P2P applications. In the next section we give a short description of MoCha and its general concepts. Next, in section 3, we present an example of the two major types of P2P networks and how to implement them with MoCha. Finally, in section 4, we end with conclusions and related work.

2 MoCha

MoCha is an exogenous coordination middleware for distributed communication and collaboration using mobile channels as its medium. In this section we first introduce the general notion of a mobile channel and discuss its major features. Then, we give a set of mobile channel types supported by MoCha. Finally, we present MoCha's application programming interface (API).

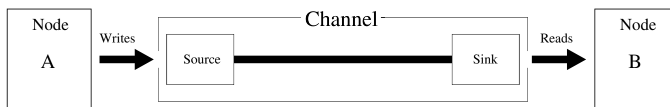

Fig. 1. General View of a Channel

2.1 Mobile Channels and Their Features

A channel, see figure 1, consists of two distinct ends: usually (*source*, *sink*) for most common channel-types, but also (*source*, *source*) and (*sink*, *sink*) for special types. These channel-ends are available to the processes that constitute a *node*. Processes can *write* by inserting values to the source-end, and *read* by removing values from the sink-end of a channel; the data-flow is locally *one way*: from a process into a channel or from a channel into a process.

Channels are *point-to-point*, they provide a directed virtual path between the nodes involved in the connection. Therefore, using channels to express the communication carried out within a system is *architecturally very expressive*, because it is easy to see which nodes (potentially) exchange data with each other. This makes it easier to apply tools for analysis of the dependencies and data-flow.

Channels provide *anonymous connections*. This enables P2P client 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. Since the applications do not know each other, it is easy to update or

exchange any one of the nodes without the knowledge of the node at the other side of the channel.

The ends of a channel are *mobile*. We introduce here two definitions of mobility: logical and physical. The first is defined as the property of passing on channel-end identities through channels themselves to other nodes in the system; spreading the knowledge of channel-ends references by means of channels. The second is defined as physically moving a channel-end from one location to another location in a distributed system, where location is a *logical address space* where node processes execute. Both kinds of mobility are supported by MoCha.

Because the communication via channels is also *anonymous*, when a channel-end moves, the node at the other side of the channel is not aware nor affected by this movement. Mobility allows dynamic reconfiguration of channel connections among the component nodes in a system, a property that is very useful and even crucial in systems where the components themselves are mobile. A component is called mobile when, in a distributed system, it can move from one location (where its code is executing) to another. Laptops, mobile phones, and mobile Internet agents are examples of mobile components. The structure of a system with mobile components changes dynamically during its lifetime. Mobile channels give the crucial advantage of moving a channel-end together with its component, instead of deleting a channel and creating a new one.

Channels provide transparent *exogenous coordination*. Channels allow several different types of connections among nodes without them knowing which channel types they are dealing with. Only the creator of the connection knows the type of the channel, which is either synchronous or asynchronous. This makes it possible to coordinate nodes from the 'outside' (exogenous), and, thus, change the systems behavior without changing the nodes.

2.2 Channel Types Supported by MoCha

MoCha supports eleven types of channels. Here we give a short description of five representative channel types. For more details and the remaining channel types we refer to the MoCha manual [6].

- *Synchronous channel.* The I/O operations of the two ends are synchronized. A *write* on the source-end can succeed only when simultaneously a *take* operation is performed on the sink-end, and vice-versa. A *take* operation is the destructive version of the *read* operation.
- *Lossy synchronous channel.* If there is no I/O operation performed on the Sink channel-end while writing a value to the Source-end, the *write* operation always succeeds but the value gets lost. In all other cases, the channel behaves like a normal synchronous type.
- *Filter (synchronous) channel.* The Filter channel behaves like a *synchronous* type. However, values that do not match the channel's pattern are filtered out (lost). *Write* operations where the value is filtered out of the channel have no influence on, nor are they influenced by, *take* operations that are performed on the same channel.

- *Asynchronous unbounded FIFO channel.* The I/O operations performed on both channel-ends are done in an asynchronous way. Values written into the Source channel-end are stored in the channel in a FIFO distributed buffer until taken from the Sink-end.
- *Asynchronous bounded FIFO (FIFO n) channel.* This channel behaves in the same way as the unbounded FIFO one, except that is has a capacity of n elements. If the channel is full a *write* operation has to wait until an element is taken out of the channel first.

2.3 MoCha's Implementation and API

MoCha is implemented in the *Java* language using the *Remote Method Invocation package* (RMI) [15] and comes in three different flavors: *MoCha*, *easyMoCha* and *chocoMoCha*.

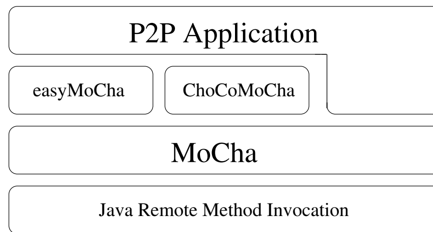

Fig. 2. The MoCha Middleware

As indicated in figure 2, *MoCha* is the basic package build upon the *RMI* layer. *MoCha* contains all the features needed to properly work with channels. However, non-experts find it difficult to work with this basic package due to two reasons. One is that the user interface is rather short and meant for expert programmers. The other reason is that in this basic MoCha package dangling references may occur due to channel-end movement. This means that the user has to write its own protocol for dealing with these invalid channel-end references. This is our intention since the choice for a particular protocol depends on the kind of system one wants to build. For non-experts or people who simply do not want to be concerned with such things we have developed *easyMoCha*. This layer is build on top of MoCha and has all the features of plain MoCha plus: a more richer and easier to use interface, and a build-in protocol for taking care of invalid channel-end references. We have also developed a package that has more or less the same features as *easyMoCha* but with the addition that nodes have to first successfully connect to a channel-end before being able to use it. We call this package *chocoMoCha* (channel connection MoCha).

The middleware has a clear and easy high-level application programming interface (API). In figure 3 we list the main classes of *MoCha* with their most

method	parameters	return	description
MoChaLocation: MoCha needs a location that points to a particular IP and Virtual Machine.			
`constructor`	`()`	`void`	creates a location.
`equals`	`(MoChaLocation loc)`	`boolean`	compares the given location with this one.
SourceEnd: The source-end of a channel.			
`write`	`(Object element)`	`void`	writes data or a channel-end reference.
SinkEnd: The sink-end of a channel.			
`read`	`(void)`	`Object`	reads an element and leaves it in the channel.
`take`	`(void)`	`Object`	destructive version of read.
ChannelEnd: An abstract class implemented by Source- and Sink-end.			
`move`	`(MoChaLocation loc)`	`void`	moves channel-end to loc.
`equals`	`(ChannelEnd ce)`	`boolean`	compares the ce with this one.
`equalsChannel`	`(ChannelEnd ce)`	`boolean`	does ce belong to the same channel?
`empty`	`(void)`	`boolean`	is the channel empty?
`full`	`(void)`	`boolean`	is the channel full?
MobileChannel: An instance holds two ChannelEnd references.			
`constructor`	`(MoChaLocation loc, String type)`	`void`	creates a new channel.

Fig. 3. MoCha API

important operations: *create channel*, *write*, *read*, *take*, and *move*. Full API details can be found in [6] and at the MoCha web page (`http://homepages.cwi.nl/~juan/MoCha/`).

3 P2P Applications on Top of MoCha

In this section we show the advantages of using MoCha for P2P applications. We discuss the two current architecture types for P2P networks. These are the *hybrid* and the *pure* P2P network architectures as defined in [11]. We explain how to implement both P2P architectures using mobile channels by giving an example of each them.

In a *hybrid* P2P network there is always a *central entity* necessary that provides parts of the offered network services. Such a central entity is often regarded as a server in the traditional way. However, the definition of a hybrid P2P architecture is not equal to the one of the client/server architecture; All the nodes of the first potentially share resources, while the clients of the second do not.

Figure 4 shows a *hybrid* P2P network that uses mobile channels for connections between its nodes. This network example is similar to the one of the Napster application[5]. Each *application node* has a set of resources to share among the other nodes of the network. However, an application node does not know any other nodes, nor the resources these other nodes are sharing. Instead, an application node connects to a *central index server* that contains a list of all the resources available from all the nodes connected to it. Once a node receives a list of resources from the *server* and requests a particular resource, the *server* arranges a connection between the requesting- and the providing-node.

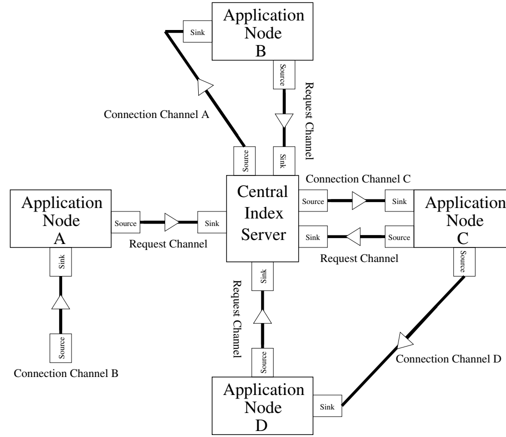

Fig. 4. A Hybrid P2P Network

```
class P2PApplicationNode
   P2PApplicationNode (SourceEnd ce)
       requestSource = ce;
       location = new MoChaLocation();
       connection = new MobileChannel(location,"Synchronous");
   connect()
       // connection.ce1 = source-end
       // sharelist = list of resources we share.
       if (!connection.ce1.full()) {
           Message msg = new Message("Joining network", connection.ce1, shareList);
           requestSource.write(msg); }
       else { // find another server or try later. }
   getResource(String resource)
       // connection.ce2 = sink-end
       Message msg = new Message("Request", resource, connection.ce1);
       requestSource.write(msg);
       while(!finished) {
           msg = connection.ce2.take();
           result.add(msg); } //done
class centralizedIndexServer
   centralizedIndexServer()
       location = new MoChaLocation();
       request = new MobileChannel(location, "FIFO 100");
   void performConnect()
       msg = request.ce2.take(); // request.ce2 = sink-end
       shareList.add(msg.shareList, msg.source);
       msg.source.move(location); // move conn. chan. source-end to us.
       msg.source.write(new Message("Connected to server"));
   void performGetResource()
       msg = request.ce2.read(); // request.ce2 = sink-end
       Node tmp = shareList.getRandomNodeWith(Resource);
       msg.source.move(node.location); // move conn. chan. source-end to resource node.
       tmp.source.write(msg); // msg already contains target SourceEnd.
```

Fig. 5. Partial Abstract Java Code of a Hybrid P2P Network

Implementing this example in MoCha is fairly easy. In figure 5 we show the most important methods of a possible implementation. Figure 4 shows a snapshot of our example network. The server has several *request channels*. The sink-

ends of these channels are kept private by the server and are meant for reading requests. However, the source-end references are known to all the application nodes in order for them to write requests to these channels. Since, in our example, the server is always on-line the nodes get a source-end reference at their creation, see their `constructor` method. Each application node has a *connection channel* meant for receiving data from the outside world, it does so by reading from the sink-end of this channel. The nodes spread the reference of the source-end to the server when connecting to it, as specified in the method `connect`. Suppose that node A is in the process of connecting to the server, then node B represents the resulting state. The server moved the source-end to its location and wrote an acknowledgment message back.

At some point in time node B requests a resource, see the `getResource` method. The server, in response, reads the request but it does not take it out of the channel, see the `performGetResource` method. Instead, the server looks randomly for a node that has the requested resource and moves the *connection source channel-end* to the found application node. This is the state represented by the nodes D and C. Node C receives a resource request from the server that was written by node D. The request remained in the channel unaffected by the channel-end move and without node D begin aware of it. Since the request also contains the target source-end, node C writes the data to it. However, it does not know that node D is receiving the data, nor does node D know that it is getting the requested data from node C. Therefore, the connection is *completely anonymous*.

To illustrate the advantage of exogenous coordination: we chose the types of the *request channels* and the *connection channels* to be respectively *asynchronous FIFO* and *synchronous*. However, we could choose other channel types as well, if desired. For example, we can make the *request channels* to be of type *synchronous*. This way, we get a different system behavior with the big advantage of not having to change, nor re-compile, the code of the application nodes. Moreover, the nodes don't even know with what kind of channel type they are dealing with.

A *pure* P2P network has no *central entity*, it is completely decentralized. Figure 6 shows a *pure* P2P network that uses mobile channels for connections between its nodes. This network example is similar, but not entirely the same, to the Kazaa network[12]. Actually, the implementation of this example is also similar to the one of the *hybrid* network example. That is why, due to space limitations, we do not present any code for this example. Most of the functionality is already given in figure 5.

Instead of having a fixed central server, we now have *supernodes*. A *supernode* is a normal application node, but at the same time it performs some of the tasks of the *server* in the *hybrid* example; it keeps a resource-list of the connected clients, and it arranges connections between the different connected nodes in the same manner as the server did. For legal reasons, the nodes cannot share any resources of the supernode they are connected too, and vice-versa.

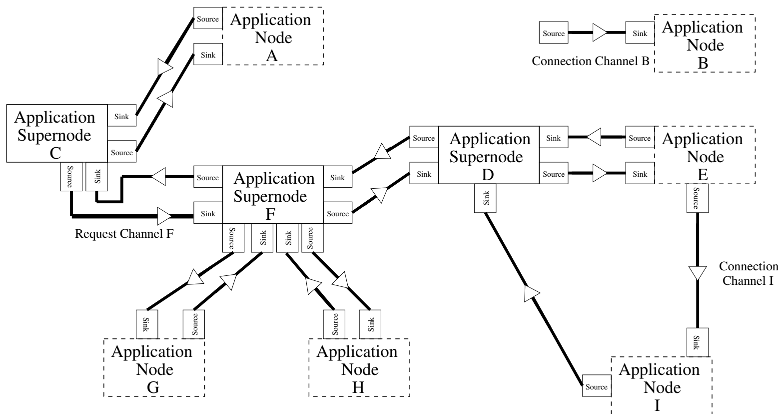

Fig. 6. A Pure P2P Network

A supernode itself is a normal node connected to another supernode. Any node can become a supernode and back to normal depending on the network state and heuristics. This means that nodes need a dynamic list of supernode source-ends, for if the supernode they are connected to becomes unavailable. To keep its list updated a node can request source-end references of neighbor nodes from its supernode. However, to keep network traffic down, a node can connect to only one supernode at the same time.

In figure 6 nodes E and I are connected to supernode D, nodes C, D, G and H are connected to supernode F, node A is connected to supernode C, and node B is not connected to any supernode. In this snap-shot nodes E and I are the only ones involved in resource transfer; node E is writing data to the connection channel of node I. The anonymous connection between the two nodes is made by supernode D in the same way as the index server in the hybrid network example.

In this *pure* P2P network the topology changes more than the one in the previous example. This more dynamically changing network example clearly shows the benefits of the mobility feature of MoCha's channels. Instead of creating and deleting channels every time a topological change occurs, we just simply move its ends to other nodes. When moving one end, the nodes using the other end of the channel are not even aware of the channel-end movement.

Just like in the previous example, we can change the network's behavior by choosing different types for the channels between the nodes. All of this is done in an exogenous way.

4 Related Work and Conclusion

In this paper we introduced and promoted the use of the MoCha middleware for P2P networks. The examples in section 3 showed the advantages of using MoCha for both a *centralized* and a *decentralized* P2P network. The advantages include, (1) the mobility of channel-ends to cope with dynamic changes in the

network. (2) The anonymous connections between nodes, that makes it possible to share resources without the involved nodes knowing each other. (3) The exogenous coordination feature, that provides different system behavior by choosing different channel types, without changing the entities using the channel. And, finally, (4) the high-level API, that makes it more easy to implement, update, and dynamically changing P2P networks.

The MoCha middleware is primary designed to provide a separation of concerns between the computational and the coordination aspects of distributed systems in general. In this paper we want to show how P2P systems benefit from MoCha. Especially P2P systems where coordinated anonymous exogenous connections are desired. However, our middleware provides only a coordination mechanism (mobile channels) and does not provide certain P2P services like *searching for particular data*, *load balance*, and *indexing*. The second generation of P2P middleware offers a complete package for such systems. Well-known middlewares are *Chord* [13], *Pastry* [10], *Tapestry* [16], and *CAN* [8]. They all provide means for locating nodes and data in the network, as well as efficient and scalable routing protocols of messages. However, they do not provide explicit (exogenous) coordination between the nodes. Therefore, designers using these middlewares can still profit from MoCha by making prototypes of their systems using mobile channels to explicitly show the coordination aspects of these systems. Later they can implement MoCha's mobile channels in these second generation P2P middlewares (if desired).

Shared data spaces are another kind of coordination mechanism. With this mechanism nodes read and write values, usually tuples like in *Linda* [3], from and to a shared space. The tuples contain data, together with some conditions. Any nodes satisfying these conditions can read a tuple; tuples are not explicitly targeted. In [2] an infrastructure for P2P networks is suggested using the Linda middleware Lime [7]. However, we think that for most P2P networks it is more efficient to use MoCha's point-to-point channels than the centralized shared data spaces.

MoCha relates to the JXTA project [14]. The JXTA middleware provides a set of 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 strongly relates to Reo[1], an exogenous coordination language where complex channel connections are compositionally build out of simpler ones. Reo provides high-level connection specifications whose semantics are independent of the entities using the connection.

References

1. F. Arbab, *A channel-Based Coordination Model for Component Composition*, Tech. Report, Centrum voor Wiskunde en Informatica, Amsterdam, 2002. Available online at `http://www.cwi.nl/`

2. N. Busi, C. Manfredini, A. Montresor, G. Zavattaro, *Towards a Data-driven Coordination Infrastructure for Peer-to-Peer Systems*, Proc. of Workshop on Peer-to-Peer Computing Co-located with NETWORKING'02, 2002.
3. N. Carriero, D. Gelernter. *How to Write Parallel Programs: a First Course*, MIT press, 1990.
4. B. Cohen, *Incentives Build Robustness in BitTorrent*, Technical Report, May 22, 2003. Available at `http://bitconjurer.org/BitTorrent/documentation.html`
5. drscholl@users.sourceforge.net, *Napster Messages*, on-line document, April 7, 2000. Available on-line at `http://opennap.sourceforge.net/napster.txt`
6. J.V. Guillen-Scholten, F. Arbab, *MoCha and easyMoCha Manual v1.0*, CWI Technical Report, Amsterdam, 2004.
7. A.L. Murphy, G.P. Picco, and G.-C. Romjan. *Lime: A coordination middleware supporting mobility of hosts and agents.* Technical Report WUCSE-03-21, Washington University, Department of Computer Science, St. Louis, MO (USA), 2003.
8. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, *A Scalable Content-Addressable Network*, ACM SIGCOMM '01, San Diego, 2001.
9. M. Ripeanu,*Peer-to-Peer Architecture Case Study: Gnutella Network*, Technical Report, University of Chicago, 2001. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker: A Scalable Content-Addressable Network; ACM SIGCOMM '01, San Diego, 2001.
10. A. Rowstron, P. Druschel, *Pastry: Scalable, Decentralized Object Location and Routing for LargeScale Peer-to-Peer Systems*, 18 Conference on Distributed Systems Platforms, Heidelberg (D), 2001.
11. R. Schollmeier, *A Definition of Peer-to-Peer Networking towards a Delimitation Against Classical Client-Server Concepts*, Proceedings of EUNICE-WATM, pp. 131-138, Paris, France, September 3-5, 2001.
12. Sharman Networks, *Kazaa, Detailed On-line Guide*, On-line Manual, 2003. Available at `http://www.kazaa.com/us/help/guide.htm`
13. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001, San Diego, CA, August 2001, pp. 149-160.
14. Sun Microsystem Inc., Home Page of the JXTA project, `http://www.jxta.org`
15. Sun Microsystems Inc., *Java Remote Method Invocation – Distributed Computing for Java*, white paper available at `java.sun.com/rmi`, 2004.
16. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.