ActorSpaces: An Open Distributed Programming Paradigm

Gul Agha and Christian J. Callsen

Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: { agha | callsen }@cs.uiuc.edu

Abstract

We present a new programming paradigm based on Actors called ActorSpace, where the communication style is replaced with a one to one-out-of-many communication style. This style was inspired by Linda, and extended with broadcast primitives to allow group-based communication. In order to add scoping to broadcast messages, we introduce the notion of ActorSpaces, which are collections of actors. We define several operations on ActorSpaces, and their names can be communicated as with actor-names. The resulting paradigm provides powerful support for distributed computing in scalable, open systems. We give an example to demonstrate the ideas and use of the new concepts developed and briefly describe work in progress to implement a prototype for ActorSpaces.

1 Introduction

In this paper, we develop a model to unify two scalable models of concurrent computing, namely *Actors* [Agh86] and *Linda* [CG89]. The Actor model provides point-to-point communication and object-style encapsulation. By contrast, Linda provides pattern directed invocation. The difference in the parallel problem-solving approach used by Actors and Linda is quite instructive: point-to-point communication provides efficient communication in a distributed system by allowing locality to be directly expressed and optimized. The locality property of Actors states that there is no way for an actor to contact other actors whose name it has not received in a previous communication. This simplifies the task of reasoning about actors (cf. [AMST92]). Furthermore, communication in Actors is secure: for example, it is not possible to "steal" messages by creating an actor with the same name as an existing actor. Finally, actors do not have to commit to a single type of message when waiting for a message to arrive.

Linda is decoupled in both space and time: information may be available so that anyone can potentially access it. Furthermore, only an abstract specification of the class of potential

receivers of a message must be given when sending a message, thus decoupling processes in space and offering better flexibility. For example, Linda provides an appropriate model for supporting class libraries of processes in a concurrent object oriented programming environment: the interface specifications of classes could be represented by patterns and used to dynamically access them. Another example of the flexibility offered by pattern directed invocation is the construction of services in a large-scale system. A service can be replicated transparently without notifying any of the clients, resulting in better availability and performance. Concurrent Aggregates (cf. [CD90]) offers a communication style similar to Linda; clients name a group of servers, and one of the members of this group will actually receive the message.

Linda, however, has some inefficiencies. It does not provide any locality, i.e. communication cannot be declared local to two or more processes that want to exchange information without allowing other processes to potentially interfere. Furthermore, it is not possible in Linda to select one out of multiple tuples for input. This makes Linda less flexible than the Actor model.

Our goal is to capture the open access of Linda, where processes are decoupled from each other, but at the same time offer locality to provide more efficient, secure communication, and allow processes to receive more than one "kind" of message from other processes. We also intend to support the communication style of Concurrent Aggregates, as a flexible scheme for performing message passing while still retaining locality. We introduce a new concurrent programming paradigm called *ActorSpace*, which we characterize as providing support for Actor locality and multiple input selection integrated with Linda decoupling and pattern based invocation. The ActorSpace model is based on message passing, but allows the user to specify a more abstract destination of messages. ActorSpace is well suited for expressing open distributed systems which may run on different architectures.

Outline of the Paper

Section 2 describes our view of open systems and why we feel that openness in systems is useful. The following section informally defines the new concurrent programming paradigm – ActorSpace. By adding a few primitives to the Actor model, we obtain ActorSpace while still retaining Actors as a special case of ActorSpace. Pattern-based communication in ActorSpace will reflect the perspective offered in Linda, where processes communicate with multicast-like primitives. Section 4 gives an example of an application, which demonstrates how the programming paradigm elegantly solves particular situations. Section 5 briefly describes the work being done on implementing a first prototype of ActorSpace. The final section concludes the paper and discusses directions of future research in ActorSpace.

2 Open Systems

We want to support an open perspective on computing where a system continues to execute after a particular application ends. New processes of computation or coordination arrive at any time and leave the system when they terminate. We want to develop systems that offer

resources to processes that perform continuous computation and reclaim these resources after their use has finished. Processes which do not know each other in advance should be able to exchange information. A particular example of these systems is distributed operating systems which support the arrival, creation and later termination of distributed applications which are using the available services for their computation. We think of distributed systems or applications as being a set of active agents that compute and coordinate. Each of these agents play the following roles during computation of a given problem:

Client: A client requests service from a server in order to perform a computation.

Server: A server provides a service to a (set of) client(s).

Manager: A manager surveys the system and adjusts it to suit needs as they arise.

Clients and servers are the "usual" clients and servers in the client–server model which can be implemented directly in Actors or Linda. Managers are not necessary for computation per se: they are needed in an administration role for keeping resources available in an open system and for maintaining security and safety. These are roles rather than sorts or types; an active object may be a client requesting service from servers, while at the same time offering service to other clients, or a manager controlling other active objects. In an open system clients cannot be trusted, so security must be enforced in order to prevent clients from misuse of the system. Some objects must have authorization to perform powerful operations such as manipulating servers. The applications do not emcompass this maintenance task which is why we appoint managers to keep the system running.

An open interface allows communication between processes which have no explicit reference to each other. Client applications may use the interface to access the services provided and perform communication with other applications, and terminate when their work has finished by exiting the system in a coherent state. An open system persists after a certain application has finished. One view is to think of it as an abstract operating system.

3 ActorSpace

In this section we present the ActorSpace paradigm and describe the new operations that extend an Actor language. The primitives of the Actor model that we refer to are the fundamental primitives of Actor-based languages. A formal semantics describing these primitives has been given in [AMST92]. ActorSpace adds three new concepts to Actors, namely: *patterns* as an abstract specification of a group of receivers, *actorspaces* as a scoping mechanism for pattern resolution, and *capabilities* as a security mechanism. The new paradigm will support programs that use both the perspective of Actors and Linda simultaneously in different stages of their computation, as we feel that neither of the two perspectives exclude the other. In the following we give an overview of ActorSpace.

3.1 Names and Patterns

In ActorSpace, an actor-name is two identifiers, namely a unique ID and a string. An actor-name uniquely identifies a single actor in an actor-context and actor-names are immutable. Actor-names are first-class values and they are created using a call to the underlying system. Actors can refer to their own name with self.

In order to name a group of actors, we introduce *patterns*. Patterns are regular expressions over actor-names and they are represented as strings. Patterns can be constructed using strings, actor-names and regular expression operators, and can be stored, copied and communicated. The normal regular expression operators are kleene-star, alternate choice, etc. Patterns are matched against actor-names at run-time and all actor-names that match the pattern belong to the same group of potential receivers, as shown below.

Names and Patterns:

Patterns match names as regular expressions match strings. Assume, that we have declared two actor-names, first-actor and second-actor:

```
first-actor  =  "Groucho Marx";  second-actor  =  "Chico Marx";
```

The following three patterns will match both of these actor-names (note that "." means any character and "*" means Kleene-star):

```
first-pattern   =  .* " Marx";
second-pattern  =  ( "Groucho" | "Chico" ) " Marx";
third-pattern   =  ( first-actor | second-actor );
```

3.2 Creation of Actors

A unique actor-name which matches a given pattern may be created with the primitive new-actor; the argument pattern given serves as a template for the name of the new actor. new-actor returns an actor-name that identifies a single actor and the new name is unique in its context. The reason for enforcing unique actor-names is that actor-names are localized to specify the actor precisely for efficient message passing. In the example above, we should not have specified the actor-names directly as we did. Instead, we must create two new actors giving a pattern using the new-actor primitive.

Actor creation:

Assume that we want to create two file servers. In order to make it easy for the clients to send messages to the servers, file servers have a name that starts with "FileServer". The two file servers could be created like this:

```
fileserver-pattern  =  "FileServer" .* ;
first-file-server   =  new-actor(fileserver-pattern );
second-file-server  =  new-actor(fileserver-pattern );
```

new-actor returns a unique name each time it is called. Therefore, we may use the same pattern for creating multiple file servers.

3.3 Communication in ActorSpace

Communication in ActorSpace is done using one of two primitives which send messages asynchronously to their destinations. When sending a message, the sender specifies a pattern which defines the group of potential receivers of a message. A message can be sent to *one* actor in a group by using the primitive send, or it can be sent to *all* actors in a group by using the primitive broadcast.

send takes two arguments: a pattern which defines the group and a message to be delivered. In this case, the target is non-deterministically chosen out of the group of potential receivers. This is useful when several actors are replicating a service offered to clients. As the messages to the servers are distributed non-deterministically, the load may be balanced automatically, and none of the clients need to know the exact number of receivers.

broadcast takes the same arguments as send. The reason for introducing broadcasting is that we want to allow the sender to specify how many of the potential receivers should receive a message without having to worry about the number of receivers. Broadcasting could be simulated by explicitly sending a message to all actors in the group, but this requires that the sender knows all of these actors. By simply specifying a pattern, the sender leaves it to the system to determine exactly which actors should receive the message. The following illustrates communication in ActorSpace.

Actor communication:

Assume that the two file servers declared in the previous example are offering a file service to clients. If a client desires to send a message to one of the servers, but does not really care which of them receives the message, it may be done in the following manner:

```
send(( first-file-server | second-file-server ),aMessage );
send(fileserver-pattern,aMessage );
```

If the client instead wanted to send a message to both of the file servers, it could be done like this:

```
broadcast(( first-file-server | second-file-server ),aMessage );
broadcast(fileserver-pattern,aMessage );
```

3.4 Creation of Actorspaces

In order to give some control over pattern matching, we define a new entity called an *actorspace*. An actorspace is a computational passive container which may contain actors and actorspaces. Actorspaces are referred to by their name, which is exactly like an actor-name. Actorspace names are first class values. Actorspace names are matched by the same patterns as actor-names, and therefore cannot be distinguished from actor-names. The only way to create an actorspace name is by creating an actorspace instance with the primitive new-space, giving a pattern which serves as a template for the name of the actorspace just as with new-actor. new-space returns an actorspace name and the new name is unique in its context.

Actorspace Creation:

Assume we want to create create an actorspace for the fileservers mentioned previously with the pattern `"FileService"`:

```
space-pattern      =  "FileService" .* ;
file-server-space  =  new-space(space-pattern );
```

Pattern matching is restricted by actorspaces so that only the aquaintances of the sender and the names of actors or actorspaces residing in the same actorspace as the sender will be matched against the pattern. Acquaintances of an actor are all the names of actors and actorspaces which the actor has stored as a part of its state. Pattern based message passing communicates to actors inside the same actorspace as the sender, unless the pattern explicitly refers to another actorspace. This provides a structural communication hierarchy. The names of actorspaces and actors may be combined to form a structural name (with a special combination operator '`/`'), much like the file names in a conventional filesystem such as the UNIX filesystem.

Pattern Construction and Matching in ActorSpace:

Assume that the two file servers we created previously are residing in the actorspace "`FileService`" we created above. If a client wanted to send a message to one of the servers, it could be done like this:

```
file-server-pattern1 = file-server-space / file-server-pattern ;
send(file-server-pattern1,aMessage);

file-server-pattern2 = space-pattern / .* ;
send(file-server-pattern2,aMessage);
```

The first pattern matches the actors inside `file-server-space` whose names match the file-server pattern whereas the second pattern matches all the actors inside all actorspaces whose names are matched by `space-pattern`.

3.5 Visibility in ActorSpaces

When actors or actorspaces are created, they are not immediately visible in the surrounding actorspace, i.e. subject to pattern matching. Actors and actorspaces must be made visible explicitly to be subject to pattern matching; this preserves the locality of the Actor model as the default. Actors are autonomous entities, so they are able to make themselves visible or invisible given an actorspace. Since actorspaces are computationally passive, however, they cannot make themselves visible or invisible in another actorspace.

We introduce two new primitives: make-visible, which explicitly subjects actors or actorspaces to pattern matching inside a specified actorspace, and make-invisible, which removes actors or actorspaces entirely from the specified actorspace. Visibility leads to the issue of security and the role of managers in the system. Managers are supposed to control the system and we clearly do not want every actor to have the ability to change the visibility of another actor or actorspace. We provide this by introducing capabilities: only the holder of *the capability* for an actor or an actorspace can change its visibility. Capabilities are unique keys that can only be created by calling the underlying system. Capabilities can be

stored, copied and, in some systems, communicated in messages. When creating an actor or an actorspace, a capability may be bound to it.

Making Actors and ActorSpaces Visible:

Assume that we want to create an actorspace which provides a file-service to clients and which is visible to the clients in the surrounding actorspace. We install two actors in the actorspace that will provide the file-service. Assume that we defined `space-pattern` and `file-server-pattern` as in the previous examples.

```
// Create a new capability and bind it to a new actorspace
space-cap = new-capability();
file-server-space = new-space(space-pattern,space-cap);

// Create a server-capability and bind it to two new file servers
server-cap = new-capability();
first-file-server = new-actor(file-server-pattern,server-cap);
second-file-server = new-actor(file-server-pattern,server-cap);

// Make the actorspace and the file servers visible
make-visible(self-space,file-server-space,space-cap);
make-visible(file-server-space,first-file-server,server-cap);
make-visible(file-server-space,second-file-server,server-cap);
```

Notice how we can bind the same capability to more than one actor. This allows the manager of the fileservers to operate on all the fileservers in the same manner with the same key. Also, note that an actor can specify the actorspace in which it is residing by referring to `self-space`. If one of the file servers are not needed, at can be made invisible by calling make-invisible with the file server's name and the server-capability as arguments.

3.6 Destruction of ActorSpaces

In addition to being able to create actorspaces and to change their visibility, we provide the possibility of destroying actorspaces explicitly when they are no longer needed, with the primitive delete-space. One motivation for this is to free up the actors for garbage collection. This also supports the concept of having libraries written in the ActorSpace programming paradigm: libraries or actor-modules may be instantiated in an actorspace, perform a task, and, upon completion, the actorspace may then be destroyed.

ActorSpace Deletion:

Assume that we have created the actorspace offering fileservice as in the previous example. At some point, the manager (which in this case is the creator), determines that the clients no longer need the file service, and deletes the actorspace.

```
// Create an actorspace and two fileservers and do some work
..
// Delete the actorspace.
delete-space(file-server-space,space-cap);
```

3.7 The ActorSpace Mail System

If there is no actor whose name is matched by a pattern at the time when a message is sent using the send primitive, one of two things happen. If the given pattern is an immutable actor-name, the message is discarded (or sent to an error handler), preserving the normal semantics of an Actor system. On the other hand, if the pattern was not an actor-name, the message and the pattern matching is suspended until at least one actor appears whose name is matched by the pattern. This definition preserves the semantics of Actors, but at the same time decouples Actors in time: when using the localized actor-names, the message is either delivered to the actor (if it exists) or consumed by the run-time system. When using less specific patterns, the message is suspended until an actor appears whose name is matched by the pattern. If a message was send using the broadcast primitive and there is no actor whose name is matched by the pattern, there are several possibilities, including: the broadcast could be discarded, or the broadcast could be suspended until there is at least one actor whose name matches the pattern. We will not go into this discussion here, and currently we choose the discard the broadcast.

If an actorspace receives a message, the message is forwarded to the actors that is contained in it. If the original message was sent using the broadcast primitive, all the actors that are visible in the actorspace receive the message; if the original message was sent using the send primitive, only one of the actors in the actorspace will actually receive the message. As an actorspace is not an active entity in itself (it is a computational passive container of active entities), the actorspace cannot perform any computation or sending new messages as the result of receiving a message, which is the normal behavior of an actor.

4 Example: A Simple Fileservice

This section presents an example of an application that can be efficiently expressed in the ActorSpace paradigm. We have chosen a simple distributed file system to demonstrate how ActorSpace elegantly solves some particular problems and how ActorSpace provides an open system for computing.

Consider a simple replicated file-service. The service is offered by several servers working together to offer higher availability and better performance. At any moment, new servers who wish to participate in offering the service could arrive – this may happen if the system decides that a higher degree of fault-tolerance is needed than currently available. The service is offered to several clients who can manipulate the replicated files as usual files by sending messages to the file-servers. Figure 1 shows an ActorSpace with a file-service, and two clients.

Assume a situation where a client wants to open a file – say "foo" – for reading. First, the client sends a message to the ActorSpace in which the servers reside. The message contains a request as well as a copy of its own name, so that the server can reply to the client. The client sends the message: send("FileService","open","foo",self). The message will be sent to the actorspace which delegates the message to one of the actors (servers) inside it – for instance, server number "i". The server opens the file and returns a file-descriptor

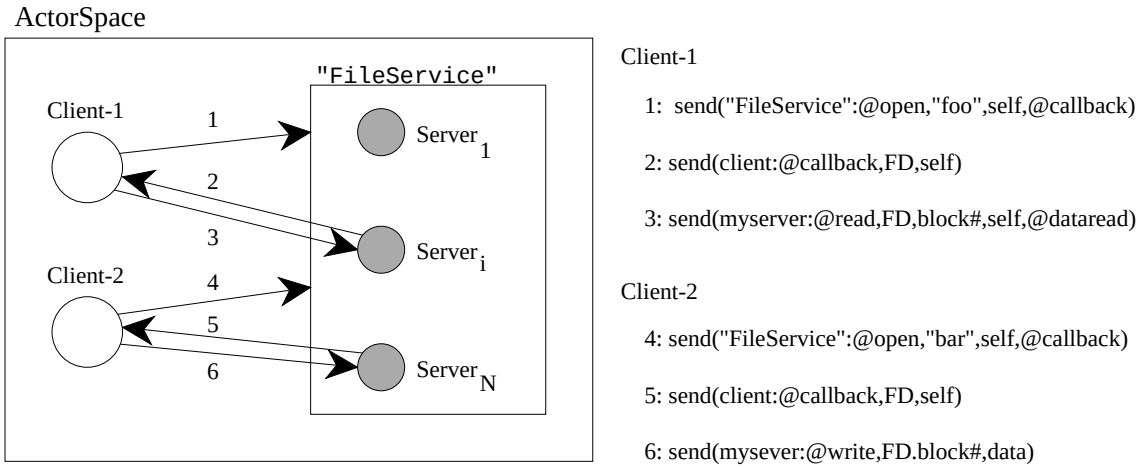

Figure 1: A replicated file-service, offered to two clients. The clients access the service by sending messages to the servers.

together with the address of itself by sending the client a message: send(client,FD,self). The client can now read blocks from the file by sending messages to the server directly: send(server,"read",FD,block#,self).

Meanwhile, another client wants to write to a file, say "bar". As with the previous client, this client sends the message send("FileService","open","bar",self) to the file-service with a request to open the desired file. Assume that another file-server, number "n", receives the request. The server opens the file, and returns the file-descriptor in the message send(client,FD,self). The client can now access the file by sending messages to the server with the requested action: send(server,"write",FD,block#,data).

Neither of the clients need to know the number of servers. By sending a message to an ActorSpace, the system selects one of the servers as a receiver, which then processes the request made by the client. After establishing contact, the clients communicate directly with the servers for better performance. By using the ActorSpace model, the run-time system will keep track of the available servers. Clients that do not communicate with a particular server, will not observe a failure of the server. By distributing the server processes, better performance may be achieved.

This example could not have been implemented in Linda or Actors as easily. Linda offers no way of "selecting" one of the two tuples; because different requests a client can make look different, a fileserver would have to dedicate itself to requests from a particular client or to a particular kind of request. Otherwise it would have to re-open the file for every request, instead of being able to serve the requests from multiple clients. In Actors, it would have been more difficult to specify "one of the fileservers" and the number of fileservers currently available would have to be known.

5 A Prototype Implementation

Work has begun to implement a first prototype for the ActorSpace programming paradigm. The system is designed in an object-oriented manner in a way that will allow maximal flexibility. To provide for an easy extension to accomodate heterogenous computing certain system-classes may be subclassed to provide transportation and data representation suitable for other architectures. The design associates the executing actors with a single coordinator on each node. This coordinator connects to other coordinator processes on other nodes, providing a transparent interface to actors that are located on other nodes in a network. The coordinator and the executing actors communcate through abstract transport objects which are subclasses to a specific message passing mechanism. This is also shown in figure 2.

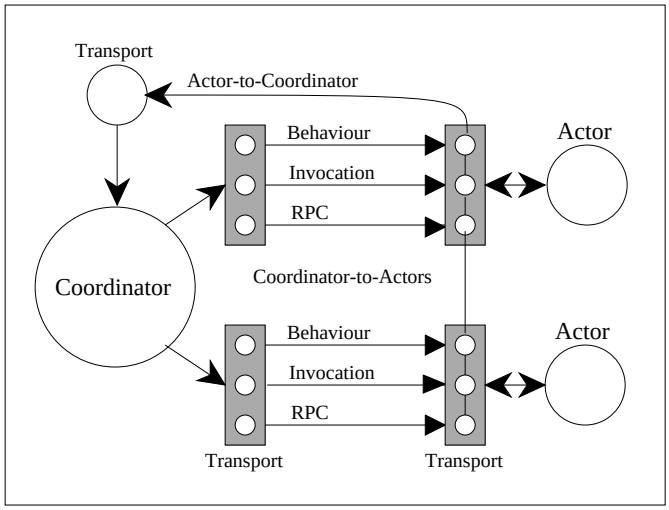

Figure 2: An overview of the design of a single node in ActorSpace.

The coordinator process has a single network connection (not shown on the figure) – which is used to broadcast information to other executing coordinators – to maintain coherency of the state of ActorSpace. This state includes "live" actors and actorspaces as well as visibility of actors. Also, the coordinators automatically determine the location of an actor given its name and forwards any outgoing messages to the appropriate node using the network connection. For local incoming messages from the executing actors, the coordinator has a single communication port which is shared among all the executing actors.

The executing actors are supplied with three different message ports, each of which have a different purpose. The first port, the Behaviour-port, is used for sending the actor its next behaviour. The second port, the Invocation-port, is used for sending the actor any messages sent to it using **send** or **broadcast**. The last port, the RPC-port, is used when an actor performs a system call that expects a return value, such as the creation of a new actor. In this case, the name of the new actor must be returned. All messages from an actor are sent to the coordinator's single port where the message, in turn, will be processed by the local coordinator.

6 Conclusions and Research Directions

In this paper, we have motivated our goals for a new distributed programming paradigm based on the Actor model and Linda paradigm. One of our main motivations was to provide an open system where applications could arrive, use the existing services, and then leave whenever their task was finished. We proposed a new programming paradigm called ActorSpace which uses message passing for coordination, but uses patterns rather than names for specification of the potential receivers of a message. The pattern based message passing decouples the individual communicating actors from each other. In order to control resolution of patterns, we introduced the idea of ActorSpaces as a scoping mechanism. This mechanism limits pattern matching to encompass only actors that reside in a single actorspace. Finally, we briefly described the ongoing work on a first prototype of ActorSpace and some of the basic design ideas.

Our ongoing work and further research includes a formal definition of ActorSpace based on a semantic definition of Actors. We also intend to study ideas for user defined arbitration instead of the current nondeterministic choice which happens when a single actor gets selected out of a group of actors. An idea that also looks promising is the idea of persistent messages that would be sent and automatically received by a new participant whenver it enters an existing group.

Acknowledgments

This research has been supported in part by the Office of Naval Research (ONR contract number N00014-90-J-1899), by the Digital Equipment Corporation, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195). Christian J. Callsen is sponsored in part by a research fellowship from Århus University and Aalborg University, and a generous grant from the Danish Research Academy (V910219).

The authors would like to thank Svend Frølund, Shingo Fukui, Wooyoung Kim, Rajendra Panwar and Daniel Sturman for helpful comments and stimulating discussions on the structure and problems of ActorSpaces, as well as critical reading of the manuscript.

References

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[AMST92] Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott. Towards a Theory of Actor Computation. In R. Cleaveland, editor, *The Third International Conference on Concurrency Theory (CONCUR '92).* Springer–Verlag, 1992. LNCS (forthcoming).

[CD90] Andrew A. Chien and William J. Dally. Concurrent Aggregates. *ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–196, 1990.

[CG89] Nicholas J. Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[Geh84] Narain H. Gehani. Broadcasting Sequential Processes (BSP). *IEEE Transactions on Software Engineering*, SE-10(4):343–351, July 1984.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Gel89] David Gelernter. Multiple tuple spaces in Linda. In E. Odjik, M. Rem, and J.-C. Syre, editors, *PARLE '89, volume 2, LNCS 366*, pages 20–27. Springer–Verlag, June 1989.

[Laz92] Edward D. Lazowska. System Support for High Performance Multiprocessing. In *Distributed & Multiprocessor Systems (SEDMS III)*, pages 1–11. USENIX Association, 1992.

[Yon90] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. MIT Press, 1990.