ActorSpace: An Open Distributed Programming Paradigm

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

Christian J. Callsen
Department of Math. and Comp. Science
Frederik Bajers Vej 7E
Aalborg University
9220 Aalborg Øst, DENMARK
Email: chris@iesd.auc.dk

Abstract

We present a new programming paradigm called ActorSpace. ActorSpace provides a new communication model based on *destination patterns*. An *actorSpace* is a computationally passive container of actors which acts as a context for matching patterns. Patterns are matched against listed attributes of actors and actorSpaces that are *visible* in the actorSpace. Both visibility and attributes are dynamic. Messages may be sent to one or all members of a group defined by a pattern. The paradigm provides powerful support for component-based construction of massively parallel and distributed applications. In particular, it supports open interfaces to servers and pattern-directed access to software repositories.

1 Introduction

Our goal is to develop a programming paradigm to support scalable component-based software construction. We believe that large software systems can be modelled as massively parallel, distributed, and open systems. The task of mega-programming [32] requires support for coordinating autonomous software systems which may, for example, consist of active processes, distributed databases, and intelligent problem-solving experts. Some of the key issues concerning composition in such systems are related to reference and access scope rules. We have developed the ActorSpace model to provide a potential method for addressing these problem and for experimenting with different alternatives.

4th ACM PPOPP,5/93/CA,USA

ActorSpace extends actor-style point-to-point asynchronous communication with *pattern-directed* invocation and broadcasting. Point-to-point communication provides efficiency in a distributed system by allowing locality to be directly expressed and optimized. On the other hand, pattern-directed communication allows abstract specification of a group of recipients. Actor groups are defined using attribute patterns, scoped within a specified actorSpace: i.e., the potential targets of a message may be defined within a specific actorSpace using destination patterns. Note that actorSpaces may overlap, and in particular, may be nested. The intuition behind ActorSpace can be roughly given in terms of two metaphors as follows.

In mathematics, a set may be described in one of two ways: by enumerating its elements, or by specifying a characteristic function which defines a subset of a domain. Explicitly identifying objects corresponds to enumerating the elements whereas providing patterns of object attributes corresponds to specifying a characteristic function. Of course, in conventional mathematics the two characterizations are equivalent as mathematical objects are static. The mathematical metaphor breaks down since computational objects, like real-world objects, may dynamically change their behavior while retaining their identity.

A second analogy is with mailing lists and telephone directories in the real world. Individuals may appear in many lists. Each list may contain a set of attributes associated with the individual – as viewed by that list. More complex databases may allow retrieval of individuals using search based on patterns of attributes.

Broadcasting can be (and typically is) implemented in terms of message passing and does not extend the expressibility of the paradigm, but using the notion of a group of receivers provides greater abstraction – a system can hide the actual number of members and their location from an application process. The application may then leave it to the system to deliver a message to all appropriate receivers. This provides an abstraction

that may be easily applied to replicating services, for instance to enhance reliability or increase performance. Moreover, in ActorSpace, services may be structured using nested actorSpaces; computations may then be successively localized. Alternately, diffusion scheduling may be obtained by successively transferring work using actorSpaces representing local neighborhoods of processors.

The ActorSpace model allows open flexible interfaces for pattern-directed retrieval from software repositories. For example, pattern-directed communication provides an appropriate model for supporting access to class libraries in a concurrent object-oriented programming environment. Consider each class as a 'factory' actor which may return its instances. The interface specifications of classes may be represented as attributes which are then used to dynamically access classes from the library.

Outline of the Paper

Section 2 describes our view of open systems and why we feel that openness in systems is useful. Section 3 briefly mentions related work with respect to open systems, process groups and broadcasting. Section 4 briefly describes the primitives of the Actor model. The following section informally defines ActorSpace. By adding a few primitives to the Actor model we obtain ActorSpace, while still retaining Actors as a special case of ActorSpace. Section 6 gives an example of an application, which demonstrates how the programming paradigm elegantly solves a specific problem. Section 7 briefly describes ongoing work on a prototype implementation of ActorSpace. The final section concludes the paper and discusses directions for future research.

2 Open Systems

We want to develop systems which offer resources to applications and reclaim resources after some application has finished using them. A particular class of such a system is distributed operating systems. Distributed operating systems provide resources to support the arrival, creation and later termination of distributed applications. We think of distributed systems or applications as being a set of active objects that compute and coordinate. Moreover, we would like the system to allow objects to exchange information even when the objects do not know each other in advance. In our terminology, each object plays 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 language models such as Actors or Linda. Managers are not necessary for computation per se: they are needed in an administrative role to keep resources available in an open system and to maintain security and safety. Note that the above 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.

The role of managers is what is new in open systems; in an open system clients cannot be trusted (for example, there is no way to know if they are buggy), so security must be enforced in order to prevent clients from contaminating a shared resource. Managers have authorization to perform powerful operations such as manipulating actorSpaces.

3 Related Work

Naming is a key issue in achieving openness in object-based systems that use sending of messages for coordination. In order to coordinate with an object, the "initiator" of the coordination must be able to name or identify the objects with which the coordination should take place.

Concurrent programming systems such as Actors [1], Emerald [21], Orca [6] and Concurrent Aggregates [11] support an object-based programming model, where objects may invoke methods in other objects by giving a reference to the object and parameters for the invocation. Open systems which use explicit references to objects and message passing as coordination primitives, usually offer a global naming service to which all objects have a reference. This naming service can then be queried for other references by accessing the accumulated data. Objects may register themselves if they want other objects to send messages to them.

On the other hand, ActorSpace supports *open interfaces* that allow pattern-based 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.

In newer operating systems, object groups or process groups are offered to support group-oriented applications, which are an important aspect of distributed systems. Generally, object groups can be viewed as an association of one name with a set of names (corresponding to members of the group), which when bundled with primitives for manipulation of groups and extension of communication primitives to groups of receivers support group oriented communication. Amoeba [22], the V Distributed System [10], and the ISIS toolkit [7] offer process or object groups, which for instance may be used for replication (to increase reliability or availability), or for faster access to services through selection of a local server instead of a non-local server.

Our work builds on the Actor model [1]. The *locality* property of Actors states that an actor may only send a message to an actor whose mail address has been explicitly given [18]. Furthermore, because mail addresses of new actors are unique, it is possible to reason it terms of subcomponents without interference caused by potential name collisions in an open system.

In other words, locality simplifies the task of reasoning about components in terms of their interfaces [4]. On the other hand, the locality property implies that two actors may communicate only if they have a common ancestor in the partial order defined by the causal chain of events. This has some drawbacks: for example, because the recipients must be explicitly known in the Actor model, changes in a group of potential receivers must be explicitly communicated. We address these difficulties with the ActorSpace model.

Linda [8, 16] provides process interaction through a globally shared memory with associative operations on the contents. Thus information is available so that anyone can potentially access it. Our goal is to provide an open access similar to Linda. At the same time, we offer locality for more efficient, secure communication.

Variations of the Linda model include first-class tuple spaces embedded in Scheme [20]. In this model, tuple spaces may be created dynamically, passed as arguments or returned as results of functions, and used in tuples or data structures. The behavior of tuple spaces may be customized, as tuple spaces define policies which allow customization of matching rules, conditions for automatic forwarding to other tuple spaces, blocking of other processes and exception handling for failures in tuple operations. Our notion of customizable actorSpace managers incorporates the power of the first-class tuple space model.

Note that in Linda and its variants (for example, [8, 20, 27]), processes must actively poll a tuple space and specify the type of tuple they want to retrieve. This model results in a number of significant differences with the ActorSpace paradigm. First, race conditions may occur as a result of concurrent access by different processes to a tuple space. Second, communication cannot be made secure against arbitrary readers – for example, there is no way of abstractly specifying that a process with certain attributes may not consume a tuple. In ActorSpace, by contrast, the visible attributes of a message's recipient are specified by the sender. Finally, in Linda, one cannot give an abstract specification which guarantees that communication is localized once initiated using patterns.

An earlier proposal using pattern-based data storage and retrieval was the *Scientific Community Metaphor* [25]. The Scientific Community Metaphor proposed problem-solving by pattern based access to a shared knowledge base by a community of computational agents, called *Sprites*. In fact, the Sprites model and Linda are remarkably close: the main difference between them is that Linda allows communication objects (tuples) to be removed from the tuple space whereas Sprites support only a monotonically increasing knowledge base [24].

Concurrent Aggregates [11] offers a communication style similar to Linda; clients name a group of actors when sending a message, and one of these actors will actually receive the message. Furthermore, Concurrent Aggregates supports nesting of aggregates, so that an entire group of aggregates may be targeted for a message. Note that membership and containment relationships in this model correspond to a strict hierarchy. On the other hand, actorSpaces may overlap arbitrarily.

The ActorSpace coordination primitives we develop include broadcasting messages to groups of receivers, also known as multicasting. Initial work was done on extending RPC to support replication [12], on broadcasting as a programming paradigm [15], and on protocols for reliable broadcasting [9]. Later work has focused on protocol design [23, 28] and on improving and supporting reliability of broadcasting protocols [7, 14, 22].

4 Actors

Actors are self-contained, interactive components of a computing system that communicate by asynchronous message passing. The basic actor primitives are:

create: creating an actor from a behavior description and a set of parameters, possibly including existing actors;

send to: sending a message to an actor; and,

become: an actor replacing its own behavior by a new behavior.

These primitives form a simple but powerful set upon which to build a wide range of higher-level abstractions and concurrent programming paradigms. The *create* primitive is to concurrent programming what definition of a lambda abstraction is to sequential programming: it extends the dynamic resource creation capability provided by function abstractions to concurrent computation. Each actor has a unique mail address determined at the time of its creation.

The *become* primitive gives actors a history-sensitive behavior necessary for shared mutable data objects. This is in contrast to a purely functional programming model and generalizes the Lisp/Scheme/ML sequential style sharing to concurrent computation.

The *send to* primitive is the asynchronous analog of function application. It is the basic communication primitive, causing a message to be put in an actor's mailbox (message queue). To send a message, the target of a communication needs to be specified. Finally, note that the arrival order of messages is nondeterministic but every message sent to an actor is guaranteed to be eventually delivered.

Practical actor languages incorporate a number of high-level abstractions such as higher order functions, inheritance, local synchronization constraints, RPC-style message-passing, and implicit functional parallelism (see, for example, [3, 19]).

5 ActorSpace

ActorSpace provides a set of coordination primitives that are independent of any specific architecture. Moreover, in the tradition of Linda and Actors, our intention is not to provide a programming language; rather we want to provide a set of coordination primitives that can be used for communication between concurrent computations. The computations themselves may be expressed in different programming notations. The runtime system for ActorSpace will support heterogeneity by selecting transport protocols and data representation formats at run-time, a task we feel is possible to perform automatically with good efficiency. By letting the communication primitives be identical on all architectures, ActorSpace provides powerful abstraction over the communication architecture.

ActorSpace adds several new concepts to Actors:

- *Attributes* are patterns which provide an abstract external description or view of an actor. Attributes may be generalized and specialized through conjunction and disjunction, respectively. Thus attributes may be embedded in a description lattice (e.g., see [5]). We do not further specify the representation of attributes, but as a simple realization, they may be based on the familiar property lists in Lisp. Pattern matching may be used to pick actors whose attributes satisfy a given pattern.
- *actorSpaces* are a scoping mechanism for pattern matching. Actors and actorSpaces may be made visible or invisible in an actorSpace. Visibility allows association of the mail addresses of actors with their attributes (as viewed by some registrar).
- *Capabilities* provide keys for secure access control, for example, in validating requests for visibility or attribute change requests.

Note that corresponding to each actorSpace is a *manager* who validates capabilities and enforces visibility changes. Although we describe default policies for actorSpaces, further customization may be obtained by manipulating managers (as we discuss later). We give an overview of the ActorSpace constructs and discuss some of the issues involved below.

5.1 Attributes and Pattern-Matching

ActorSpace provides two kinds of handles to access an actor: the usual actor mail address, which corresponds to the identity of an actor, and the attributes for an actor that are visible in some actorSpace. Patterns may be used to define groups of actors using their visible attributes. Abstractly, each actorSpace maps a pattern to a set of actor mail addresses by matching on its list of registered attributes of visible actors.

5.2 ActorSpaces

An *actorSpace* is a computationally passive container of actors and acts as a context for matching patterns. Patterns will only be matched against listed attributes of actors and actorSpaces that are *visible* in a specified actorSpace. An actorSpace is created by the expression create-actorSpace(capability) which returns a unique actorSpace mail address. The specified capability may be used to authenticate visibility operations in the actorSpace created. As with actors, actorSpaces may be visible in other actorSpaces. Thus, actorSpaces can be referred to by their actorSpace mail address or by a pattern.

5.3 Communication

ActorSpace communication is done using one of two primitives: send or broadcast. Both the primitives send

a message asynchronously to the specified receiver(s). The sender specifies a set of receivers by giving a pattern and an actorSpace. The pattern is matched against all listed attributes of actors visible in the specified actorSpace.

When send(pattern@actorSpace,message) is used to send a message, a *single* target actor is nondeterministically chosen out of the group of potential receivers. This is useful when several actors are replicating a service offered to clients. For example, as the messages to the servers are distributed non-deterministically, the load *may* be balanced automatically by an implementation, and none of the clients need to know the exact number of potential receivers. Note that the actorSpace specification, here and below, may itself be pattern based.

When broadcast(pattern@actorSpace,message) is used to send a message, *all* of the actors whose attributes match the pattern receive the message. A reason for introducing broadcasts is that we want to allow the sender to specify the kind of potential receivers which should receive a message, without having to also 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 know each 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 broadcast primitive greatly simplifies expressing many applications. For instance, in search problems such as the Traveling Salesman, a new lower bound can be broadcast to all nodes participating in the search for the shortest route.

In ActorSpace, broadcasting is done by specifying a group of receivers that should receive a message. The run-time system then carries out the message delivery. We assume that message delivery is only finitely delayed, but that the message order is not necessarily preserved; thus, unlike other broadcast implementations [7, 22], we do not guarantee a global or partial order on broadcast messages. Broadcasts may be received by two actors in a different order and point to point messages may be interleaved between two broadcasts. If a global order on broadcasts to a specific group is desired, it can be obtained by sending all messages that are to be broadcast to a special actor whose sole purpose is to receive messages from group members, and then broadcast these serially to the group using some agreed upon protocol (cf. sequenced send in the actor language HAL [19]). However, better performance may be obtained by not guaranteeing any order on broadcast messages, when such an ordering is not necessary or desirable [7, 29], which is why we do not enforce any ordering of broadcasts.

5.4 Visibility in ActorSpace

When an actor or an actorSpace is created, it is *not* automatically placed in an actorSpace; thus it may not be subject to pattern matching on its attributes. Actors and actorSpaces must be made *explicitly* visible to be subject to pattern matching; thus the default preserves the locality properties of the Actor model. 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 a given 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 security by the standard technique of introducing *capabilities*: only the holder of *the capability* for an actor or an actorSpace can change its visibility. Capabilities are unforgeable unique keys that can only be created by calling the underlying system with the primitive new-capability(). Capabilities can be stored, compared, copied and, in some systems, communicated in messages. When creating an actor or an actorSpace, a capability may be bound to it, and only if this capability is presented, may an actor's visibility be changed. A capability may also be bound to more than one actor or actorSpace.

We introduce two new primitives for controlling visibility: make-visible(α,attributes@space,capability), which explicitly subjects the actor or actorSpace α to pattern matching inside a specified space, and make-invisible(actor,space,capability), which removes actors from the specified actorSpaces – and thus any other enclosing actorSpace. Finally, note that attributes of visible actors or actorSpaces may change. Such changes may be made with the change-attributes operation, using the same parameters as make-visible.

5.5 Garbage Collection

The presence of actorSpaces affects the garbage collection of both actors and actorSpaces. As long as an actor (or actorSpace) is visible in an actorSpace, it may be potentially reachable and thus cannot be garbage collected until the container actorSpace has been garbage collected. An actorSpace may be deleted if no actor has a way of accessing it (and, as with actors, no messages containing its mail address are pending).

However, when an actorSpace is garbage collected, the actors contained in that actorSpace themselves are *not* deleted, rather, they are no longer visible using the actorSpaces (which is moot since the actorSpace itself

is no longer accessible). Conversely, when an actor is no longer reachable, and furthermore cannot potentially reach a reachable actor, a garbage collection algorithm may be able to delete it. Note that since actorSpaces are viewed as passive containers, garbage collecting them is simpler than actors: inverse reachability need not be considered. We will not discuss garbage collection further, but we expect that a garbage collection algorithm for the Actor model [30] may be adapted in designing a garbage collector for ActorSpace.

5.6 Fairness and Asynchrony

Generally, ActorSpace messages have the same properties as Actor messages: delivery is asynchronous, but is guaranteed to eventually happen. However, there are, a few exceptions which we describe briefly. Consider a message sent using a pattern which is not satisfied by any visible actor's attributes. In this case, the pattern matching may be suspended until at least one actor appears whose attribute is matched by the pattern. This allows asynchrony in attribute updates and pattern-based message passing. On the other hand, such a message could be considered an error – forcing additional synchronization.

Again, if a message was sent using the broadcast primitive and there is no actor whose name is matched by the pattern, there are several possibilities, including: the broadcast is discarded, the broadcast is suspended until there is at least one actor whose attributes match the pattern, or broadcasting could be persistent, so that any actor (existing or created in the future) whose attributes match the pattern, will receive the broadcast message *exactly once*. The last case may be useful in enforcing a protocol or assuming some other common knowledge in a group.

In our current implementation, send and broadcast messages are suspended until at least one actor arrives whose attribute matches the pattern for the broadcast. This is the cheapest option that avoids repeated synchronization that would otherwise be needed to address the asynchrony in the system. However, a particular choice of semantics cannot satisfy all requirements. By allowing actorSpace managers to be customized, we can vary the temporal constraints on the matching rules.

5.7 The Problem of Cycles

The consequence of an actorSpace being visible in itself can be quite catastrophic: if its attributes are matched by some broadcast message, an infinite number of messages may be generated by the message. In case of a pattern directed send, the damage is less significant: the message could be simply lost in an infinite forwarding loop. As part of the semantics of make-visible we do not allow an actorSpace to be made visible in itself, or recursively in any contained actorSpace. This avoids cycles in the directed acyclic graph defined by the visibility relation between actorSpaces.

In implementation terms, avoiding such cycles means that a visibility relation graph must be constructed before an actorSpace is allowed to be visible. Furthermore, since actors are encapsulated and should not be sent arbitrary bookkeeping messages, type information must be maintained to determine whether a mail address refers to an actor or an actorSpace.

An alternate strategy is to tag message and compare tags with those of previously sent messages. This may offer a way of trapping cycles of messages simply forwarded by actors as well. Again, we believe no single strategy will provide a universally desirable solution. The problem is probably best addressed by customizing actorSpace managers.

6 Example: A Dynamic Process Pool

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

Consider a parallel system with a number of processors in a pool that can be allocated to solve problems. We consider algorithms for parallel problem-solving based on divide-and-conquer. Specifically, we envision a number of actors (which for efficiency reasons could be located on different nodes), which participate in solving the problem. All these actors reside in an actorSpace, and new actors may come along while the system is running to help to solve the problem. This is shown in figure 1.

The figure shows a processor pool located in a local actorSpace, which is not visible except to the client. Assume that an actorSpace called `ProcPool` contains a pool of available processors. The client could contain code as follows:

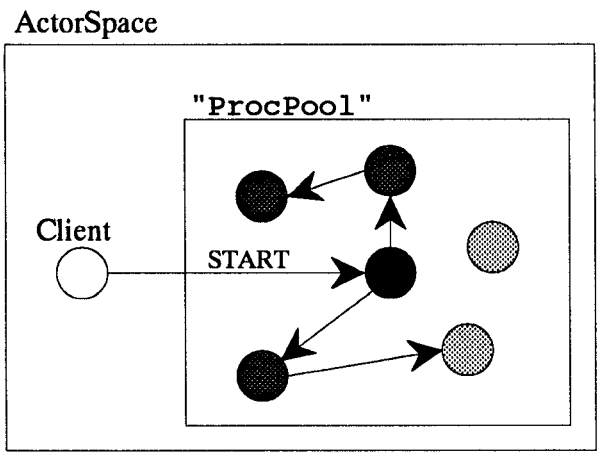

Figure 1: A processor pool where the actors participate in solving a problem. The Client starts by sending a problem into the pool, which is divided among the processors. The lighter circles denote newly arrived processes.

```
// Create a processor pool
..
// Create the job to be done
..
send(*@ProcPool,job,self);
// Wait for the answer to return
..
```

The client starts the data-processing by sending a message to an arbitrary processor inside the ActorSpace `ProcPool` and a return address for answers. Note that * in this example is a pattern that matches any attribute. Now, each of the processing actors might contain code as follows:

```
// Receive a new job
..
if job-too-big(job) then
  subjobs = divide-into-subjobs(job);
  // distribute the jobs among neighbors
  for i in "all subjobs" do
    send(*@MyNghbrProcs,subjobs[i],self);
  // Collect answers and merge
  ..
else
  result = process(job);
endif
// Return the answer
send(answer-destination,result);
```

The first processor which receives the job may decide that the job is too large to handle; it then divides the job into smaller subjobs, sends them to one of the other actors in its neighborhood processor pool and waits for the partial answers. Note that `MyNghbrProcs` is an identifier bound to some actorSpace. The binding will depend on the code for the particular processor – different processors may refer to different actorSpaces.

In particular, depending on the organization of the actorSpaces, the above divide and conquer can be done within a nested subspace. Thus, the broadcast can happen to representatives of a WAN whereas the subsequent distribution can be localized to be within a LAN.

Note that the processors in the pool do not have to know how many processors participate in solving the problem, and indeed, the number may vary during the computation. By only specifying that a message should be sent to one of the actors in the "current" actorSpace, one of the visible processors will receive the message, thereby receiving part of the problem. By letting the processors divide the job as the problem is analyzed, we remove a bottleneck around a master process in the more traditional "one master, multiple slaves" approach to parallelize problems. And by using patterns, the number of processors allocated to the task can be adjusted during execution – without having to stop the system.

7 A Prototype Implementation

The implementation of the first prototype for ActorSpace focuses on making a small system for initial experimentation with the ActorSpace coordination paradigm. We want to keep the prototype implementation small, thus we have limited its extent and kept the system simple. Instead of building a compiler which will compile the programming language, we have chosen to build a small sequential interpreter for interpreting the code associated with each method definition. An interpreter gives us the additional flexibility of easily loading behaviors at run-time. A future extension will include a byte-compiler which will compile the code into an intermediary form, similar to early implementations of other object–oriented programming languages (such as SmallTalk).

The system is designed in an object-oriented manner to allow maximal flexibility. To provide for an easy extension to accommodate heterogeneous computing, certain system-classes may be subclassed to provide transportation and data representation suitable for other architectures. The overall design of the prototype can be separated into two parts: the design of a single node, and the design of inter-node coordination.

7.1 Representation of Attributes

In our current prototype, attributes are concatenations of atoms, and patterns are regular expressions over atoms – rather analogous to the structure of files and directories in UNIX. Actors are actually created inside an actorSpace (their host space), although they are not visible in this actorSpace unless explicitly made so. Furthermore, they may be made visible in other actorSpaces, regardless of whether or not they are visible in their "host" actorSpace.

Patterns are resolved inside the sender's host actorSpace, unless the pattern explicitly refers to another actorSpace. This provides a structural communication hierarchy. The attributes of actorSpaces and actors may be combined to form a structured attribute (with a special combination operator '/'), much as is the case with file names in a conventional file-system such as in the UNIX file-system. In the same manner, a globally visible actorSpace which is the "root" of the tree of actorSpaces, is created automatically by the run-time system. This ensures that the first actorSpaces created by the users may be made visible to other users. On the other hand, such global visibility makes automatic garbage collection of actorSpaces generally infeasible, and therefore the current prototype provides explicit means of destroying actorSpaces.

7.2 Single-node Design

The single-node design associates all the executing actors on a node with a single local coordinator. Each node in our prototype implementation contains the following:

- A small sequential interpreter for the computations.
- An ActorInterface which allows methods defined in the actor behaviors to be invoked.
- A Coordinator which provides the main run-time support and carries out the ActorSpace coordination primitives.

A major goal of the implementation was to make the design of the interpreter, the ActorInterface and the Coordinator as modular as possible. We have achieved this by defining a communications format between the interpreter, the ActorInterface and the Coordinator. As far as the ActorInterface is concerned, behaviors allow the initialization of variables and parameterized invocation of methods. The Coordinator is only concerned with receiving requests, carrying them out, and possibly returning messages to the ActorInterface or the interpreter. The interpreter uses a parsed representation of the behavior specification for interpretation and occasionally accesses the ActorInterface for sending and receiving messages from the Coordinator.

The Coordinator and the executing actors communicate through abstract transport objects which are subclassed to use a specific message passing mechanism; the mechanism may be selected at run-time. By using transport objects for communication between actors and the coordinator, we enhance portability and obtain a good abstraction: actors communicate explicitly with the local coordinator which carries out the ActorSpace primitives. This is illustrated in figure 2.

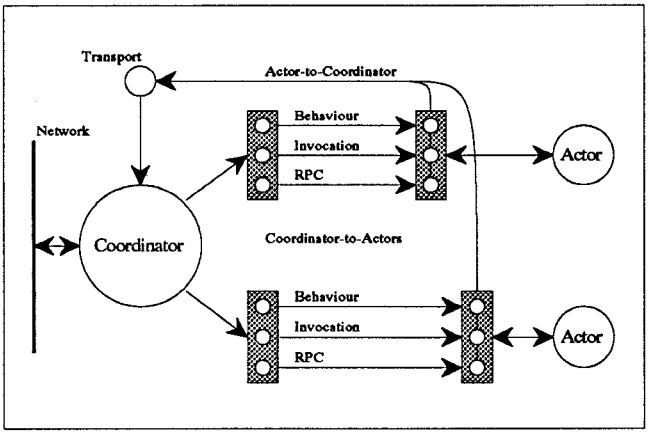

Figure 2: An overview of the design of a single node in ActorSpace.

The executing actors are supplied with three different message ports, each of which has a different purpose. The Behavior-port is used for sending the actor its next behavior. The Invocation-port is used for sending the actor any messages sent to it using send or broadcast. 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.

7.3 Inter-node Design

The local coordinator connects to coordinators on other nodes using a (virtual) coordinator bus. This provides a transparent interface to actors that are located on other nodes in a network. The network is illustrated in figure 3.

A coordinator process uses the network connection to broadcast information to other coordinators in order

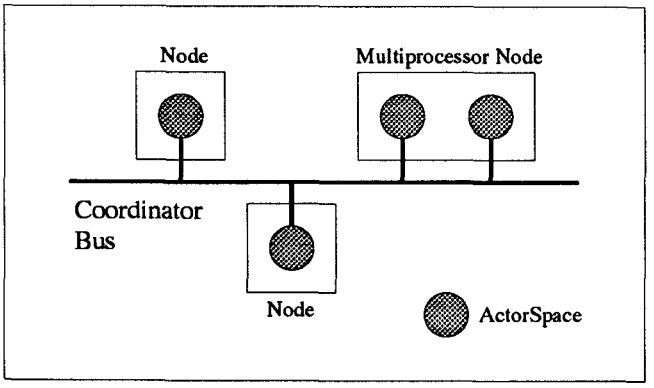

Figure 3: The connection of several coordinators on a (virtual) coordinator bus.

to maintain coherence of the state of ActorSpace. This state includes "live" actors and actorSpaces as well as visibility of actors. 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. The broadcast network does not rely on having a network with a hardware broadcast facility at its disposal. However, the current design needs a global ordering on individual broadcasts between coordinators to order visibility changes globally, so that all nodes have the same view of visibility in ActorSpace (although *not* necessarily the same order on broadcasts to actors). The broadcasting between the coordinators could, for instance, be done using either the Amoeba broadcast protocol [23] or a centralized broadcaster and sequencer [9]; both have orderings of some sort on broadcast messages.

8 Conclusions and Research Directions

Our ongoing work and further research includes a formal definition of ActorSpace based on a semantic definition of Actors and their MetaArchitectures [4, 31]. We intend to develop customizable managers to allow experimentation with different coordination and scheduling mechanisms. For example, this would allow experimentation with arbitration mechanisms which may be used instead of the current indeterminate choice that happens when a single actor is selected out of a group of actors. More powerful managers could use dæmons to monitor actors in an actorSpace and update attributes in order to maintain specified coordination constraints [2, 13]. Finally, it may be useful to allow persistent messages that would be automatically received by a new participant whenever 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 work was done in part while the second author was a visiting scholar at the Unviersity of Illinois.

The authors would like to thank Svend Frølund, Wooyoung Kim, Rajendra Panwar, Anna Patterson, Daniel Sturman, and Carolyn Talcott for helpful comments and stimulating discussions on the structure and problems of ActorSpaces, as well as critical reading of the manuscript. The first author would especially like to acknowledge a number of fruitful interactions with, as well as inspiration from the work of David Gelernter, Carl Hewitt, John Holland, Suresh Jagannathan, Ken Kahn, Henry Lieberman, Nicholas Carriero, Carolyn Talcott, Peter Wegner, and the late Alan Perlis.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*, pages 197–207, Palermo (Sicily), Italy, September 1992. IFIP.

[3] K. Wooyoung and G. Agha. Compilation of a Highly Parallel Actor-Based Language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Workshop on Languages and Compilers for Parallel Computing.* Yale University TR DCS RR-915, 1992. to appear in LNCS, Springer-Verlag.

[4] Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott. Towards a Theory of Actor Computation. In *Third International Conference on Concurrency Theory (Concur '92)*, pages 565–579. Springer–Verlag, August 1992. LNCS 630.

[5] G. Attardi and M. Simi. Semantics of Inheritance and Attributions in the Description System Omega. In *Proceedings of IJCAI 81*, Vancouver, B. C., Canada, August 1981. IJCAI.

[6] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[7] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[8] 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.

[9] Jo-Mei Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[10] David R. Cheriton and Willy Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[11] Andrew A. Chien and William J. Dally. Concurrent Aggregates. *Second ACM Symposium on Principles and Practice of Parallel Programming, SIGPLAN Notices*, 25(3):187–196, March 1990.

[12] Eric C. Cooper. Replicated Procedure Call. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing Conference*, pages 220–232. ACM, August 1984.

[13] S. Frølund and G. Agha. A Language Framework for Muti-Object Coordination. In *Proceedings of European Conference on Object Oriented Programming*, 1993. To appear in LNCS, Springer-Verlag.

[14] Hector Garcia-Molina and Annemarie Spauster. Ordered and Reliable Multicast Communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

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

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

[17] 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.

[18] C. Hewitt and H. Baker. Laws for Communicating Parallel Processes. In *1977 IFIP Congress Proceedings*, pages 987–992. IFIP, August 1977.

[19] C. Houck and G. Agha. HAL: A High-level Actor Language and Its Distributed Implementation. In *Proceedings of th 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, August 1992.

[20] Suresh Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In E. H. L. Arts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, volume 2*, pages 254–276. Springer–Verlag, June 1991. LNCS 506.

[21] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[22] M. Frans Kaashoek and Andrew S. Tanenbaum. Fault Tolerance using Group Communication. *ACM Operating Systems Review*, 25(2):71–74, April 1991.

[23] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An Efficient Reliable Broadcast Protocol. *ACM Operating Systems Review*, 23(4):5–19, October 1989.

[24] Kenneth M. Kahn and Mark S. Miller. Response to: "Linda in Context", Carriero and Gelernter, Commun. ACM 32 (4):444–458, Apr. 1989. *Communications of the ACM*, 32(10):1253–1255, October 1989.

[25] William A. Kornfeld and Carl Hewitt. The Scientific Community Metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(1):24–33, January 1981.

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

[27] Satoshi Matsuoka and Satoru Kawai. Using Tuple Space Communication in Distributed Object-Oriented Langu ages. In *OOPSLA '88 Conference Proceedings*, pages 276–284, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.

[28] P. M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[29] Pat Stephenson and Kenneth Birman. Fast Causal Multicast. *ACM Operating Systems Review*, 25(2):75–79, April 1991.

[30] N. Venkatasubramaniam, G. Agha, and C. Talcott. Scalable Distributed Garbage Collection for Systems of Active Objects. In *Proceedings International Workshop on Memory Management*, pages 441–451, St. Malo, France, September 1992. ACM SIGPLAN and INRIA, Springer-Verlag. LNCS.

[31] Nalini Venkatasubramanian and Carolyn Talcott. A MetaArchitecture for Distributed Resource Management. In *Proceedings of the Hawaii International Conference on System Sciences*. IEEE Computer Society Press, January 1993.

[32] G. Wiederhold, P. Wegner, and S. Ceri. Towards Megaprogramming. *Communications of the ACM*, 35(11), November 1992.

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