Open Heterogeneous Computing in ActorSpace

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

Gul Agha
Department of Comp. Sci.
1304 W. Springfield Avenue
University of Illinois at U.-C.
Urbana, IL 61801, USA
Email: agha@cs.uiuc.edu

Abstract

A number of efforts in heterogeneous computing involve the development of basic architecture independent communication primitives. We present a new programming paradigm, called ActorSpace, which 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 heterogeneous scalable distributed applications. In particular, it supports open interfaces to servers and pattern-directed access to software repositories.

1 Introduction

Heterogeneous systems are an integral part of computing today. Our approach to heterogeneity is to provide an abstraction layer on top of different architectures; the abstraction

allows distributed software components to be specified transparently. We want to describe systems that may communicate through the barriers of architectural differences and allow the system, as a manager of resources, to persist when applications terminate. In particular, this requires support for coordinating autonomous heterogeneous software systems which may, for example, consist of active processes, distributed database objects, and intelligent problem-solving experts.

Some of the key issues involved in addressing problems of coordination are related to reference and access scope rules. We have developed the ActorSpace model to provide a potential method for addressing these problem and experimenting with different alternatives. ActorSpace extends actor-style point-to-point asynchronous [1] communication with *pattern-directed* invocation. 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.

In ActorSpace, actor groups are defined by using specific attribute patterns, scoped within a specified actorspace: 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. Specifying an explicit collection of mail addresses of actors corresponds to enumerating the elements whereas pattern directed communication 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 actors may dynamically change their behavior.

A second analogy is with mailing lists and telephone directories in the real world. Individuals may be listed 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.

Note that broadcasting can be (and typically is) implemented in terms of message passing and does not extend the expressibility of the paradigm, but if the notion of a group of receivers is introduced, the system can hide the actual number of members and their location from the application process. The application may then leave it to the system to deliver the 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 once initiated. Alternately, diffusion scheduling may be obtained by successively transferring work to local neighborhoods of processors.

Pattern-directed communication allows open flexible interfaces for object access. For example, pattern-directed communication provides an appropriate model for supporting access to class libraries of processes in a concurrent object-oriented programming environment. Consider each class as an 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 related work. Section 3 describes the Actor model, on which our new coordination paradigm has been based. Section 4 informally defines the ActorSpace programming paradigm. ActorSpace is obtained by adding a few primitives to the Actor model and

thus retains the Actor model as a special case. Pattern-based communication in ActorSpace reflects the perspective offered in Linda, where processes communicate with multicast-like primitives. Section 5 gives an example of an application that demonstrates how the programming paradigm elegantly solves particular situations. Section 6 describes the design and implementation of the prototype that we have developed. The final section concludes the paper by discussing directions for future research in ActorSpace.

2 Related Work

Several mechanisms have been proposed for interconnecting different architectures for heterogeneous computing. The general philosophy of heterogeneous computing, and the ActorSpace model, is to have a network of nodes appear as a *coherent* system at a certain level of abstraction. Specifically, by trying to enforce a high-level uniformity in software while allowing differences in hardware, architectural differences between individual nodes are masked.

MIT's Project Athena [11] supports heterogeneous systems at the application level, by trying to standardize the appearance of applications at the interface level. Carnegie–Mellon University's Andrew System and its successor Coda File System [35, 37] offer heterogeneity at the resource level by supporting a large shared file system on a network of different computers. The same goes for SUN–NFS, a Networked File System developed by SUN Microsystems. The protocol used in SUN–NFS is stateless and makes SUN–NFS partially fault-tolerant: if a server fails, clients can simply wait until the server comes online, at which point the transfer is retried. The University of Washington's "The HCS Environment for Remote Execution" (THERE) system [36], uses *Heterogeneous Remote Procedure Calls*, or HRPC (see below), as means of interaction. THERE supports multiple standards for data

representation, transport- and control prototols, instead of legislating a single standard.

Other proposals use a more low-level approach. HRPC [12] being developed at the University of Washington provides remote procedure calls (RPC) to machines with different architectures. The Agora System [14] offers shared memory for coordination between and writing of parallel applications in multiple programming languages. ARCADE [21] offers shared memory with operations to transmit memory blocks to other machines and sharing of blocks with other processes. The Parallel Virtual Machine (PVM) [39] has been used to build a heterogeneous system called Heterogeneous Network Computing Environment, or HeNCE [23]. HeNCE is a tool for developing parallel applications with PVM, which is the main task of PVM.

It is important to notice that there are several fundamental styles of heterogeneity: loose integration through networking (HRPC and ARCADE), integration of different programming languages (Agora), common interfaces to different systems (Athena, and to some extent Andrew), and transparency at the operating system level. The important issue is that all these systems offer *coherence* at a certain level of abstraction.

The ActorSpace coordination primitives we will develop include broadcasting messages to groups of receivers, also known as multicasting. Broadcasting has been studied for some time now. Initial work was done on extending RPC to support replication [22], on broadcasting as a programming paradigm [25], and on protocols for reliable broadcasting [18]. Later work has focused on protocol design [30, 34] and on improving and supporting reliability of broadcasting protocols [24, 13]. In operating systems research, the notion of process groups has been introduced in several systems, among them the V Distributed System [19] and the ISIS Toolkit [38]. Broadcasting has also been discussed as a way of achieving fault-tolerance in operating systems [9]. Broadcasting provides a flexible way of replicating a service, where a client broadcasts a request to a group of servers, and collects one or more responses using

some voting criteria [22].

ActorSpace builds on a concurrent object-oriented programming paradigm (COOP). COOP systems, such as Actors [1], Emerald [29], Orca [10] and Concurrent Aggregates [20] 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. As in ActorSpace, the location of the object and its representation is transparent. On the other hand, ActorSpace also supports *open interfaces* that allow pattern-based access between objects which have no reference to each other. An overview of recent Actor research by the authors' group may be found in [5]. A survey of COOP systems appears in [2].

Linda [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 capture the open access similar to Linda, where processes are decoupled from each other, but at the same time offer locality to provide more efficient, secure communication.

Variations of the Linda model include Jagannathan's first-class tuple spaces embedded in Scheme [28]. In Jagannathan's model, tuple spaces are first class objects, i.e., tuple spaces may be created dynamically, passed as arguments or returned as results from 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.

Note that in Linda and its variants [16, 28, 33], 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. By contrast, in ActorSpace, the 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). Some industrial experience using actors as a high-level distributed shell language suggests that such dynamic linking capability is extremely useful (e.g. [40]).

An earlier proposal using pattern based data storage and retrieval was the Scientific Community Metaphor [32]. 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 [31].

Concurrent Aggregates [20] offers another communication model based on groups; 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.

3 Actors

The Actor model was first proposed by Hewitt [26], and later developed by Agha [1, 3]. The model can be thought of as providing an abstract representation of multi-computer

architectures. An actor[1] is an active object which interacts with other actors using asynchronous point-to-point message passing. 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 *behavior* of an actor is the actions performed in response to a message. Actors can change their behavior dynamically to any other behavior desired.

The *send to* primitive is the asynchronous analog of function application. To send a message, the target of a communication needs to be specified. It is the basic communication

[1]We will capitalize Actor when referring to the model and use lower case to refer the individual objects (except where grammatical correctness requires capitalization!) The same convention will be used for the ActorSpace model and individual actorspaces.

primitive, and messages sent to an actor are buffered in a *mail queue* until the actor is ready to process the message. Each actor has a system-wide unique identifier called its *mail address*. Mail addresses are bound to identifiers and are not otherwise visible. They may be copied, sent to other actors, or compared using primitive operators (cf. EQ in Lisp). The mail address allows an actor to be referenced in a location transparent way.

An actor's *acquaintances* are the mail addresses of actors it knows. An actor can only send messages to its acquaintances, an important property which allows local reasoning about the safety property of actor systems [7]. 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. Providing an actor with the ability to send messages to a mail address does not give that actor the ability to receive messages sent to that address, which is the case for some formal models, and introduces security leaks.

In order to abstract over processor speeds and allow adaptive routing, preservation of message order is not guaranteed. Actors can be created dynamically and have unique names, and acquaintances can be communicated to other actors. Actors do not have to commit to a single type of message when waiting for a message to arrive.

Actors can be characterized as a *safe* programming paradigm; it is possible to do local reasoning about the behavior of actors because one can limit who may send a message. Although this locality property simplifies reasoning about actor programs, it makes it impossible for an arbitrary actor (client) to contact an already existing actor, such as an actor (server) which provides a particular service, unless the mail address of the server actor is given to the client. In particular, when a new client actor is created it needs to be informed about the potential servers and, conversely, the existence of a newly added server actor needs to be communicated to potential clients – something that may not be realistic in an open system.

4 ActorSpace

The ActorSpace model incorporates the primitives of actors and extends them to capture the open access familiar in Linda. Thus actors 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. The ActorSpace model is based on message passing, but allows the user to specify the destinations of messages more abstractly. Specifically, ActorSpace adds three 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. 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).

ActorSpace coordination may be accomplished in terms of broadcasting messages to groups of receivers. Broadcasting can be (and typically is) implemented in terms of message passing, but if the notion of a group of receivers is introduced, the system can hide the actual

number of members and their identities from the application process. The application may then leave it to the system to deliver the message to all appropriate receivers. This provides an abstraction that may be easily applied to replicating services, either for reliability or for increased performance. 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.

4.1 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.

4.2 Actor and Actorspace creation

Actors are created using the primitive new-actor(capability), which creates a new actor, and returns its unique actor mail address to the caller; new-actor is identical to the *create* primitive in Actors. The specified capability, which must have been previously created by the special primitive described in the following, is associated with the new actor, and must be presented to authenticate visibility operations on the actor, i.e., making the actor visible or invisible in some actorspace. If no visibility is desired, the capability may be omitted, in which case the actor can never be made visible except if done by itself.

Recall that an *actorspace* is a computationally passive container of actors which 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 new-space(capability) which returns a unique actorspace mail address. The specified capability may be used to authenticate visibility operations *on* 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.

4.3 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). If a mail address is used as the destination, the two communication primitives become identical to the *send to* primitive of Actors, and delivers the message to the specified receiver without further interpretation of the destination. If the sender specifies a set of receivers by giving a pattern (which may include a specification of actorspaces to evaluate the pattern in), the pattern is matched against all listed attributes of actors visible in the specified actorspaces.

When send(pattern,message) is used to send a message, a *single* target actor is non-deterministically 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. The pattern may optionally include a specification of the actorspaces in which the pattern matching will be performed, and the actorspace specification may itself be pattern based.

When broadcast(pattern,message) is used to send a message, *all* of the actors whose attributes match the pattern receive the message. Broadcasting could be simulated by explic-

itly sending a message to all actors in the group, but this requires that the sender know the identity of all the members of the group. By simply specifying a pattern, the sender specifies the kind of potential receivers but leaves it to the system to determine exactly *which* actors should receive the message. Thus the broadcast primitive greatly simplifies expressing many applications.

We assume that delivery of normal and broadcast messages is only finitely delayed, but that the message order is not necessarily preserved; thus, unlike other broadcast implementations such as the ISIS Toolkit [13], 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 [27]). However, better performance may be obtained by not guaranteeing any order on broadcast messages, when such an ordering is not necessary or desirable [13, 38], which is why we do not enforce any ordering of broadcasts.

4.4 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, they cannot make themselves visible or invisible in a given actorspace, which implies that authority should be given to some entity, i.e., a manager, to control visibility.

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 altering the visibility of actors and actorspaces, namely make-visible(α,attribute@space,capability), which explicitly subjects the actor or actorspace α to pattern matching inside a specified space under the given attribute, and make-invisible(α,space,capability), which removes actors from the specified actorspace.

4.5 Destruction of Actorspaces

In addition to being able to create actorspaces and to change their visibility, we provide the possibility of destroying an actorspace explicitly when it is no longer needed. The primitive delete-space(space) achieves this effect. One motivation for this is to free up the actors for possible garbage collection: as long as an actor is visible in an actorspace, it may be potentially reachable and thus cannot be garbage collected. This also supports the concept of having libraries written in the ActorSpace coordination paradigm: libraries or actor-modules may be instantiated in an actorspace, perform a task, and upon completion, the actorspace may be destroyed.

Note that when deleting an actorspace, deletion is performed recursively through any

sub-actorspaces. However, the actors contained in an actorspace themselves are *not* deleted, rather the deletion amounts to explicitly removing all visibilities contained in the actorspace and any contained actorspaces. The actors that were visible in the actorspace still exist, although they may now be subject to garbage collection.

4.6 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).

Garbage collecting an actorspace corresponds to deleting it: any contained actors and actorspaces will be removed from it. 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 [41] may be adapted in designing a garbage collector for ActorSpace.

4.7 Fairness and Asynchrony

Generally, ActorSpace messages have the same properties as Actor messages: delivery is asynchronous, but is guaranteed to eventually happen. There are however 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 message. 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.

4.8 A Summary of ActorSpace

The ActorSpace coordination paradigm has extended the Actor model in two ways; first by decoupling actors in space and time, and second by introducing three new concepts: patterns, actorspaces and capabilities. Patterns specify groups of receivers for messages, actorspaces provide a scoping mechanism for pattern matching, and capabilities give control over certain operations performed on actors and actorspaces.

ActorSpace supports heterogeneous computing by using the message passing facilities as

an encapsulation of the individual actor's state. ActorSpace supports an open system by using a pattern as a specification of the receiving group of actors. By using the ActorSpace communication primitives, executing actors need not be aware of the fact that other actors may be executing on different architectures (actor names are location-independent). The result of this is that actors can join and later leave an ongoing computation or make use of an available service without a specific connection to a set of computing actors or to a server.

Actor names (and actorspace names) are immutable identifications that are localized to offer a precise reference. Patterns allows the holder to send a message to the actor or actorspace which matches the particular pattern. Actor names (and actorspace names) can be converted into patterns, thereby losing the locality associated with the original actor name. The capability for an actorspace allows the holder of the capability to delete an actorspace, and to make it visible or invisible in another actorspace. A capability for an actor allows the holder to make that actor visible or invisible.

5 An Example

This section presents an example, which is a simple model of a car manufacturer. The example will show how applications may be expressed in the ActorSpace paradigm, resulting in a lot of flexibility in the implementation. The syntax used in the example uses ActorSpace communication primitives and a C-like notation for computation. The communication primitives in the example accept several parameters as the message body, and use strings as patterns, with a UNIX-like directory structure, as this is what the current prototype supports. Furthermore, note that although an actor may be registered in multiple actorspaces, the current implementation allows it to also belong to a *host* actorspace. The host is used to to resolve patterns unless a different actorspace is specified.

In the example, we assume that a car consists of a chassis, four wheels, two seats, and a steering wheel; some of these components may consist of subcomponents. We assume that each of these components may be made separately and supplied to the manufacturer by other manufacturers (sub-contractors). There may be several sub-contractors that are able to supply the same product. A schematic overview is shown in figure 1.

A potential customer would like to send a request to the car factory, and later receive the car (we assume the car was already paid for). The car manufacturer uses the aforementioned four components to build a car, and would like to be able to deliver cars, in spite of the fact that some sub-contractors are out of products, as long as other sub-contractors are able to supply with the necessary products.

We implement each set of sub-contractors in a separate actorspace. This way, messages sent to the actorspace or patterns evaluated in the actorspace will be received by manufacturers of that particular category only. If we take advantage of the fact that a message sent to an actorspace is forwarded to one of the actors inside it, the code for sending out requests for making wheels, seats, the chassis and steering wheel could look like the following:

```
method @make_car() {
    int i;
    // Make 4 wheels, 2 seats, 1 chassis and 1 steering wheel
    for (i=0;i<4;i++) {
        // Each invocation makes the wheel 'i' and replies
        send(wheel_space:@make_wheel,self,i);
    }
    for (i=0;i<2;i++) {
        // Each invocation makes the seat 'i' and replies
        send(seat_space:@make_seat,self,i);
    }
    // Make the chassis
    send(chassis_space:@make_chassis,self);
    // Make the steering wheel
    send(steering_wheel_space:@make_steering_wheel,self);
}
```

Note that since ActorSpace does not include a call-return communication primitive, the invoked sub-contractors must reply explicitly by sending a reply message to the car manufacturer. Once the factory has been set up, the `make_car` method does not need to know the identities of the current sub-contractors. By sending a message to an actorspace, the system selects one of the sub-contractors as a receiver, which then processes the request made by the assembly. By using the ActorSpace model, the run-time system will keep track of the available sub-contractors, and the sub-contractors may coordinate among themselves to satisfy the incoming requests.

Each of the sub-contractors should be able to forward requests among each other, if a particular sub-contractor cannot satisfy the request. This is done by relying on the fact that all other visible actors in the particular contractor's host actorspace are able to supply with the same product as itself. By using the pattern ".*" below, we specify that a message should be sent to someone in the actor's host actorspace. The code for making a wheel, for instance, could look like this:

```
method @make_wheel(name requester,int wheel_no) {
    struct { int rim; int tire; } w;
    //  Any rims and tires left?
    if (no_of_rims < 1 || no_of_tires < 1) {
        //  No, forward request to another wheel maker
        send(".*":@make_wheel,requester,wheel_no);
    } else {
        //  Create a wheel and send it back to the requester
        ...
        //  Store the numbers in the wheel
        w.rim = no_of_rims--;
        w.tire = no_of_tires--;
        //  Reply to the requester with the wheel and its number
        send(requester:@wheel_made,w,wheel_no);
    }
}
```

Finally, we consider the situation of multiple car factories. Each of these factories would like to use some set of sub-contractors, and these sub-contractors may be able to supply more than one factory with their products. The "Car Manufacturers and Sub-contractors Union" has declared that all car factories are visible in the actorspace with the name `"/cars"`, and sub-contractors are visible in sub-actorspaces under their category, such as `"/cars/wheels"`, `"/cars/seats"`, etc. The new code for `make_car` could be the following:

```
method @make_car() {
    int i;
    //  Make 4 wheels
    for (i=0;i<4;i++)
       { send("/cars/wheels/.*":@make_wheel,self,i); }
   //  Make 2 seats
       .....
   //  Make the chassis
       .....
}
```

Implementing this example in Linda or Actors would not have the same elegance and encapsulation. The Actor model does not have the flexible naming facility found in ActorSpace, and in Linda, the multiple car factories would interfere with each other, or sub-contractors would have to be exclusive to a single factory only.

6 Design and Implementation of a Prototype

The goal of our ActorSpace prototype is to provide a proof of concept; in particular it allows us to study how the ActorSpace coordination paradigm may be implemented and provides us with some programming experience in using ActorSpace primitives for coordination. We thus focus on functionality rather than efficiency. This is not to say that our prototype

ignores efficiency; it is however, a concern secondary to ease of implementation. Our overall intentions with the prototype design are the following:

Architectural independence: The design should be as specific architecture independent as is possible. Architecture independence makes the support for heterogeneous systems easier and possibly configurable at run-time. By clearly identifying parts that are architecture dependent and factoring these parts out, we obtain a system that is easier to move to other architectures, thus allowing multiple versions of the same system to coexist on the same network with minimal porting problems.

Separation: The design should conceptually separate actors from the run-time support system. By separating the functional aspect of the actors which are executing from the run-time support (i.e., the ActorSpace kernel), and by having the actors access the kernel through a well-defined interface, we allow for extension of the system to include multiple base languages without modifications to the ActorSpace kernel.

Extensibility: The design should be minimal, but allow for easy extension. We are developing an *initial* prototype, thus we want to make a simple implementation which is nevertheless extensible.

Experience: The development of the prototype gives us initial experience with implementing ActorSpaces. The design of the prototype gives us experience with ActorSpace implementations which may be used to re-implement parts of the system for efficiency purposes.

With the above goals in mind, the implementation focuses on making a small system for initial experimentation with the ActorSpace coordination paradigm. Instead of building a compiler that compiles the programming language to native assembler, we chose to let the

compiler generate byte-codes, and build a small sequential interpreter for interpreting the byte-codes associated with each method definition. An interpreter gives us the additional flexibility of easily loading behaviors at run-time. This is similar to early implementations of other object-oriented programming languages (such as SmallTalk). Our implementation consists of the following:

- a compiler that compiles source files into byte-code assembler,
- a sequential byte-code interpreter for the computations, with a scheduler for multi-tasking the individual actors, and
- coordinators which provide the main run-time support and carry out the ActorSpace coordination primitives.

There are several reasons for using a byte-code interpreter and a scheduler. Besides simplicity, the prototype will be easier to debug for functionality as well as performance, if the computational part is interpreted instead of running as native assembler. The compiler needs only a single back-end which makes its implementation faster and easier. Finally we may control the multitasking ourselves because the byte-code interpreter is part of the system.

We chose the C++ programming language as our implementation language, because we think that the benefits from using an object-oriented programming language greatly outweighs the performance penalties for a prototype. An implementation whose main concern was efficiency would use less generic code to speed up computation.

In the following sections we present the overall system design with an overview of the system and the design of the run-time support for actors in form of the ActorSpace kernel and the support for heterogeneity as designed in the prototype.

6.1 An Overview of the System

The overall design of the prototype can be separated into two parts: the design of the kernel for a single node, and the design of a inter-node coordination module. The design of the kernel associates all the executing actors on a node with a single local Coordinator, which takes care of executing the ActorSpace primitives. All local executing actors access the coordinator through a single interface using the same message format as the coordinators located on different nodes, the main difference is that the local executing actors need not necessarily use a specific transportation mechanism and may perform a function call instead.

The coordinator also receives requests from the network, such as loading a module, or delivering a message, and carries out the task, possibly invoking the scheduler to change the state of actors from waiting for an invocation to ready to execute. The local actors are interpreted by the byte-code interpreter, calling the system-call interface when necessary. This interface builds a message for the coordinator in the proper format, and invokes the desired service, any return values are propagated back to the executing actor. This is also shown in figure 2.

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. A coordinator uses the network connection to broadcast information to other coordinators in order 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 coordinators do not rely on having a network with a hardware broadcast facility at its disposal, but would of course benefit from it. In order to maintain a coherent view of ActorSpace, an ordering on these coordinator broadcast messages is necessary (however *not* for performing the broadcast primitive). The

broadcasting between the coordinators could, for instance, be done using either the Amoeba broadcast protocol [30] or a centralized broadcaster and sequencer [18].

6.2 The Coordinator

A coordinator is the main component of the run-time support for an actorspace. A coordinator has two main tasks: storing information about the current "state" of an actorspace, and handling request messages from local actors and other coordinators. Requests can be divided into two groups: those which require a broadcast message to other coordinators – namely the operations new-actor, new-space, make-visible, make-invisible and delete-space in order to maintain a coherent view of ActorSpace, and those that do not – namely, new-capability, send, broadcast, become and init-behavior. Note that broadcast may require a multicast to those nodes where actors that match a pattern are residing; this can be determined given their names.

To increase the efficiency of the nondeterministic message send and the broadcast, the current implementation requires all coordinators to keep a coherent view of ActorSpace. This allows communication and behavior transformations to be performed in parallel, and in optimal cases where the only actors involved are residing on the same node, we avoid a network message. This is more efficient than a centralized ActorSpace coordinator, which would be a potential bottleneck for local operations. The global broadcasts between coordinators reduce the speed of creation and deletion of actorspaces, and of actors which have visible names (and thus may be registered in multiple actorspaces). The design decision is inspired by the fact that these operations are performed less frequently. On the other hand, the design decreases the efficiency of executing visibility manipulations.

A broadcast demanding request is divided into two phases: a global phase, which ini-

tializes and broadcasts the request that is to be distributed to all relevant coordinators, and a local phase, where the actual request is performed on the local node. The non-broadcast request are done in a single phase when the coordinator receives the request. When sending invocation messages (i.e., send and broadcast), the coordinator matches the pattern against all visible actors, and determines the destination. If any of the destination actors happen to be on a remote node, a message is sent to that node to forward the message to the appropriate actors.

The implementation of the coordinator is very close to the described design. To provide single node capability, a coordinator sets up the network connection only if necessary, and arranges to be interrupted whenever a message from another coordinator occurs. All the implemented operations are still split up into a "global" and a "local" phase, so the transition to networking consists of adding an implementation of a network transportation class that provides broadcast facilities.

Of greater importance is how the actor and actorspace tables are represented. These tables should not penalize searching for an actor or an actorspace if its exact name (as opposed to having a pattern which the name matches) is known. We have done this by using two hash tables, one for actors and one for actorspaces. The hash-table is accessed via hash-values that are equal to an actor's or actorspace's unique ID. We use unique 64-bit location dependent values for actor and actorspace IDs, but the current implementation does not rely on the location dependence when locating actors. As actors and actorspaces may be inserted into other actorspaces, each visible actor and actorspace have "local" names under which it appears in the actorspaces. The representation of actorspaces stores a list of actors and actorspaces that is contained in an actorspace, and this list may be browsed when matching patterns against actor-names. Therefore, an actor or an actorspace may be accessed in two ways: directly by unique ID or indirectly by using its "local" name and a set of actorspaces

that define where to look.

When the coordinator receives requests that change the visibility in ActorSpace, the coordinator changes the contents of some of the existing actorspaces by adding or removing actors or actorspaces. Removal of an actorspace means removing all actors and actorspaces from the actorspace (recursively into sub-actorspaces), which prepares the actors for garbage collection. In the prototype implementation we have not implemented garbage collection. A distributed garbage collection process may be added later – a garbage collection algorithm similar to the one that Actors use may be applied [41].

6.3 Support for Heterogeneity

Of great importance to the design is the support for run-time heterogeneity. The design builds on the work done on Heterogeneous Remote Procedure Calls (HRPC) [12]. HRPC uses three components at call-time: a transport protocol, a control protocol, and a data representation format. The transport protocol defines how the data is moved across the network, the control protocol takes care of the RPC-protocol, and the data representation format describes how data is transformed into a linear representation. These protocols can be selected at run-time, appropriate to the partners in the actual communication.

We have taken a slightly simpler approach. We need to support asynchronous message passing, which implies that we do not need a control protocol; instead it is sufficient to have a data representation format and a transport protocol for moving the linearized data from the source node to the destination node. These properties are captured in two abstract classes: an abstract class that defines a data representation format, and an abstract class that defines transportation. At run-time, an instance of an appropriate subclass is used for the actual communication and data representation. This provides a simple extension to accom-

modate heterogeneous computing configurable at run-time, and at the same time simplifies the porting of the prototype to other platforms. The choice of data representation formats and transportation protocols is hidden from the clients, i.e., the ActorSpace applications. A programmer uses the ActorSpace coordination primitives, and the run-time support decides which specific transport primitives and data representation to use based on the destination of the message.

Transportation of data from actors to its coordinator or from one coordinator to another is done via a subclass of the abstract transportation class, which defines three methods: send, which sends a message of a given length in bytes to a destination, receive, which receives a message from a source, and RPC which performs RPC, by doing a send followed by a receive. Note that RPC-style message protocol is not primitive in Actors, however, such message-passing may be transformed to asynchronous message-passing [6]. These three methods are architecture independent, and do not assume any structure of the message that is to be sent or received. This class is used for the message passing between the coordinators, and could in principle also be used between the actors and the coordinator.

The actual implementation of the data representation format and the transportation is not relevant to their use. It may turn out that a given architecture results in better efficiency if transportation is done via shared memory through message queues with a synchronization primitive, and in this case, a transportation subclass might perform message passing through message queues. The only limitation of run-time selection of data representation format and transportation is that all the available subclasses must have been defined at compile-time in the current implementation. Dynamic loading of these protocols, which are specified in classes, is a topic for future research.

7 Research Directions

We discuss a number of open issues in our definition of ActorSpaces which need further study. When an actor uses a communication primitive, a pattern is specified which determines the destination of the message to be sent. We have not put any temporal constraints on *when* the pattern is matched. In particular, the ActorSpaces have complete freedom in determining when the pattern matching is done, e.g., if it is done for actors visible at an 'instance', those visible over a finite duration, or persistently. Linda does not allow the sender to determine persistence – instead, any potential recipient may remove any tuple. In contrast, Sprites [32] do not allow any message to be removed from the database. A more flexible solution would allow messages with different behaviors: persistence in some potentially bounded time, use once only (linearity), persistent until explicitly removed by a potential recipient, etc.

We introduced capabilities to empower managers of actorspaces and actors, so that certain operations can only be performed by an authorized actor, i.e., by a manager holding a capability for the actor or actorspace in question. We believe capabilities are important: a programming paradigm for open distributed systems should offer security as well as expressibility and coordination. Many distributed operating systems have capabilities to protect resources from misuse, and in ActorSpace one could think of managers and servers creating customized capabilities, which may be given out to clients. However, we have not provided specific structure or mechanisms for capabilities.

Current research includes moving the ActorSpace prototype implementation to several architectures, to evaluate the prototype design and the programming paradigm on a larger scale. A formal semantic definition of ActorSpace has been developed and will appear in [15]. However, a theory of ActorSpace which allows abstract equational reasoning about actorspaces remains to be developed.

Acknowledgments

This research has been supported in part by the Office of Naval Research (ONR contracts N00014-90-J-1899 and N00014-93-1-0273), 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 first author was a visiting scholar at the University 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 ActorSpace, as well as critical reading of the manuscript. This paper extends and subsumes [4].

References

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

[2] G. Agha, P. Wegner, and A. Yonezawa. *Research Directions in Object-Oriented Programming.* The MIT Press, Cambridge, Massachussetts, April 1993.

[3] Gul Agha. Supporting Multiparadigm Programming on Actor Architectures. In E. Odjik, M. Rem, and J.-C. Syre, editors, *PARLE '89, volume 2*, LNCS 366, pages 1–19. Springer–Verlag, June 1989.

[4] Gul Agha and Christian J. Callsen. ActorSpaces: An Open Distributed Programming Paradigm. In *Proceedings of the 4th ACM Symposium on Principles & Practice of Parallel Programming*, pages 23–32, May 1993. Appears as ACM SIGPLAN Notices 28(7), July 1993.

[5] Gul Agha, Svend Frølund, WooYoung Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel and Distributed Technology*, 1(2):3–14, May 1993.

[6] Gul Agha and WooYoung Kim. Compilation of a Highly Parallel Actor-Based Language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, Yale University Technical Report YALEU/DCS/RR-915*, pages 1–12, New Haven, CT, September 1992.

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

[8] Gul Agha and Rajendra Panwar. An Actor-Based Framework for Heterogenous Computing. In *Workshop on Heterogeneous Processing*, pages 35–42. IEEE, IEEE Computer Society Press, March 1992.

[9] Özalp Babaoğlu. Report on the 4th ACM SIGOPS European Workshop on Fault Tolerance Support in Distributed Systems. *ACM Operating Systems Review*, 25(1):19–43, January 1991.

[10] 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.

[11] Edward Balkovich, Steven Lerman, and Richard P. Parmelee. Computing in Higher Education: The Athena Experience. *Communications of the ACM*, 28(11):1214–1224, November 1985.

[12] Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogenous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

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

[14] Roberto Bisiani and Alessandro Forin. Architectural Support for Multilanguage Parallel Programming on Heterogenous Systems. *ACM Operating Systems Review*, 21(10):21–30, October 1987.

[15] Christian J. Callsen. *Open Heterogeneous Distributed Computing*. PhD thesis, Aalborg University, 1994. To appear.

[16] 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.

[17] Nicholas J. Carriero, David Gelernter, and T. G. Mattson. Linda in Heterogenous Computing Environments. In *Workshop on Heterogeneous Processing*, pages 43–46. IEEE, IEEE Computer Society Press, March 1992.

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

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

[20] Andrew A. Chien and William J. Dally. Concurrent Aggregates. In *Proceedings of the Second ACM Symposium on Principles & Practice of Parallel Programming*, pages 187–196, March 1990. Appears as ACM SIGPLAN Notices 25(3), March 1990.

[21] David L. Cohn, William P. Delaney, and Karen M. Tracey. ARCADE: A Platform for Heterogeneous Distributed Operating Systems. In USENIX *Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 373–390, Fort Lauderdale, FL, 1989. USENIX Association.

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

[23] Jack Dongarra, G. A. Geist, Robert Manchek, and V. S. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–175, 1993.

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

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

[26] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[27] Chris Houck and Gul 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.

[28] 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*, LNCS 506, pages 254–276. Springer–Verlag, June 1991.

[29] 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.

[30] 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.

[31] 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.

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

[33] Satoshi Matsuoka and Satoru Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *OOPSLA '88 Conference Proceedings*, pages 276–284, November 1988. Appears as ACM SIGPLAN Notices 23(11), November 1988.

[34] 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.

[35] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[36] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting Heterogenous Computer Systems. *Communications of the ACM*, 31(3):258–273, March 1988.

[37] Mahadev Satyanarayanan. Scalable, Secure and Highly Available Distributed File Access. *IEEE Computer*, 23(5):9–21, May 1990.

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

[39] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[40] Christine Tomlinson, Phil Cannata, Greg Meredith, and Darrell Woelk. The Extensible Services Switch in Carnot. *IEEE Parallel and Distributed Technology*, 1(2):16–20, May 1993.

[41] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable Distributed Garbage Collection for Systems of Active Objects. In *Proceedings of the International Workshop on Memory Management*, LNCS 637, pages 134–148, St. Malo, France, September 1992. ACM SIGPLAN and INRIA, Springer-Verlag.

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

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

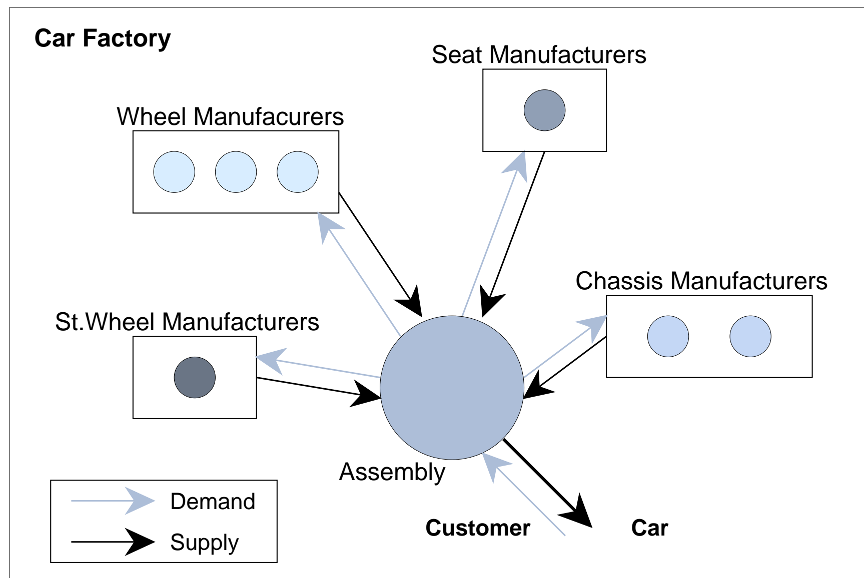

Figure 1: An automobile manufactory, consisting of one assembly and four subcontractor categories, which may contain several actual subcontractors. Each filled circle corresponds to a manufacturer.

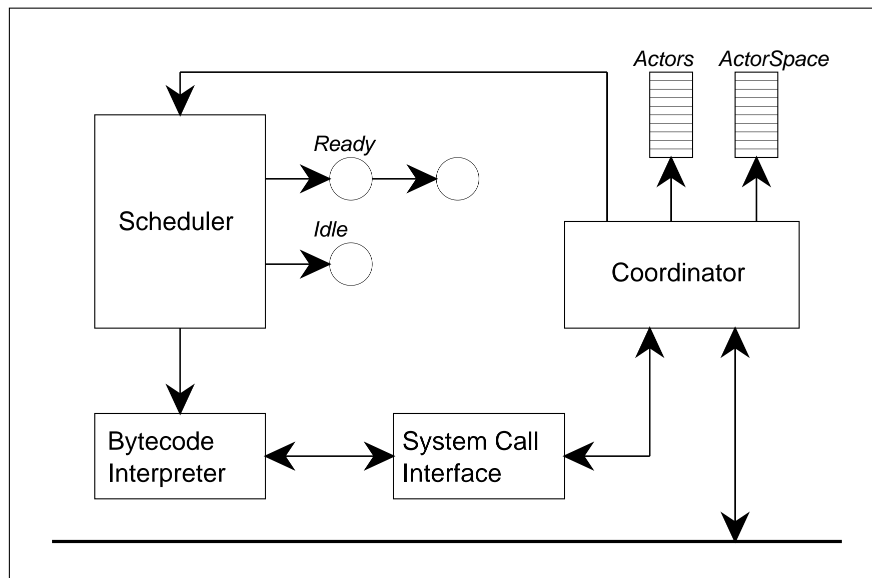

Figure 2: An overview of the design of a single node in ActorSpace.