Mobility, Security, and Autonomous Reconfigurability in Worldwide Computing

Carlos Varela, Robin Toll, Travis Desell, and Abe Stephens

Department of Computer Science,
Rensselaer Polytechnic Institute, Troy, NY 12180, USA,
{cvarela,tollr,deselt,stepha}@cs.rpi.edu,
http://www.cs.rpi.edu/wwc/

Abstract. Autonomous reconfiguration of distributed systems is a critical component of reliable and efficient execution of programs over highly dynamic open large-scale computer networks. Autonomous reconfiguration requires program mobility and a comprehensive security framework. In this paper, we present an actor language with extensions for mobility and security, as well as the operational semantics to reason about these topics in distributed systems. Furthermore, we introduce a system reconfiguration decision function that enables autonomous actors to dynamically find their optimal network location based on availability of resources, such as network bandwidth, latencies and processing power. Finally, we describe our open-source World-Wide Computer framework and preliminary implementation results.

1 Introduction

Internet based distributed computing systems benefit from open dynamically reconfigurable designs as these designs allow the systems to be used in constantly changing heterogeneous environments. Consider a data mining application running on an open distributed system using hundreds of thousands of computing devices to discover useful patterns in scientific data sets (e.g., protein folding, SETI, weather forecasting, etc.). If a system can make use of new computing resources as they become available and is both secure and resilient to failures in a subset of its computing nodes, it has the potential to leverage the power of idle computing resources around the world.

Dynamic reconfiguration of distributed systems is necessary but not sufficient to accomplish worldwide execution of programs. The continuous change in resource availability requires autonomous reconfiguration—dynamic system reconfiguration triggered by the system in response to changes in its underlying execution environment. The ultimate goal of autonomous distributed system reconfiguration is to make optimal use of available resources in a fault-tolerant and semantically consistent manner across the entire system.

In this paper, we study program component mobility as a fundamental prerequisite to autonomous systems reconfiguration. Mobility itself introduces new requirements on software, e.g., it is critical to devise strategies for secure and
controlled distributed resource management. We specify an actor-based model and formalism to reason about program component mobility and secure resource access in worldwide computing systems.

Current distributed systems are becoming increasingly complex. Not only are there increasing amounts of nodes and connections, in current wide area networks, nodes can join and leave at any time. Also, node processors may become flooded with tasks or their connections may become overloaded with packets. Connections and latency between nodes can also vary by large factors either because of network topology or because of the work load of a node at a given time. In a large scale worldwide computing system these variables change frequently often in unexpected ways as the system reconfigures; autonomous or self-configurable systems are needed as the human ability to statically manipulate such networks does not scale.

Each node in the system uses a local dynamic reconfiguration decision function to enable autonomous reconfigurability and achieve global stability. To demonstrate the feasibility of our approach, we provide preliminary empirical results using our own worldwide computing software framework. The Worldwide Computer framework consists of an actor-oriented programming language, a distributed program execution environment, and a network operating system for autonomous distributed systems reconfiguration.

Paper Outline In Section 2, we further motivate and informally introduce abstractions for programming worldwide computing applications. Section 3 specifies our model by providing an operational semantics for a simple actor language and extensions for mobility, autonomous actors, and security. Section 4 describes our software prototypes implementing autonomous system reconfiguration and presents our preliminary performance results. We conclude the paper with a discussion of research issues and related work.

2 Programming Abstractions for Worldwide Computing

2.1 Actors

The Actor model of computation is based around the concept of encapsulating state and process into a single entity. Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [1] and facilitates mobility [2]. Each actor is a unit of computation encapsulating data and behavior. The behavior defines how the actor reacts on receipt of a message. Each actor has a unique name, which can be used as a reference by other actors.

Actors only process information in reaction to messages. While processing a message, an actor can carry out any of three basic operations: alter its state, create new actors, or send messages to peer actors (see Figure 1).

Communication between actors is purely asynchronous and guaranteed. That is, when a message is sent, the model guarantees that the destination actor will
receive the message; however, it does not guarantee the order of message arrival or, therefore, the order of processing. A side effect of this is that since actors can change their own behavior based on incoming messages, unless the actor’s name is known only to the sender, its behavior could change significantly before a message arrives and is processed.

The actor model and languages provide a very useful framework for understanding and developing open distributed systems. For example, among other applications, actor systems have been used for enterprise integration [3], real-time programming [4], fault-tolerance [5], and distributed artificial intelligence [6].

![Diagram of Actors and Messages](image)

**Fig. 1.** Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, and/or (3) send messages to peer actors.

### 2.2 Universal Actors

In considering mobile computation, it becomes useful to not only model the interactions of actors with each other, but also to model the interactions of actors with their environments. In the actor model, locations are not represented, therefore, it does not matter if two actors are in the same memory space, or on two computers on opposite ends of the earth. However, when considering the problems associated with worldwide computing, it becomes important to represent the actor’s environment. Otherwise it is not possible to model the behavior of an actor, e.g., when its computation environment is unreliable, or when different resources are available in different locations.
Universal actors extend actors with locations, mobility, and the concept of universal names and universal locators. Names represent actor references that do not change with actor migration. Locators represent references that enable communication with universal actors at a specific location.

An actor’s location abstracts over its position relative to other actors. Each location represents an actor’s run-time environment and serves as an encapsulation unit for local resources. Ubiquitous resources have a generic representation—actors keep references which get updated upon migration to resources at new locations. Actors can also keep references to non-ubiquitous resources—scarce or not generally available—by using resource attachment and detachment operations. For example, a standard output stream is ubiquitous and can always refer to the current actor’s execution environment. Conversely, an actor needs to attach to a robot resource in Mars, so that the reference remains the same upon migration.

2.3 Autonomous Actors

When a system is composed of mobile actors, it can be reconfigured arbitrarily, as long as all its used resources are ubiquitous. Autonomous actors extend universal actors by:

- profiling performance to guarantee quality of service,
- including a satisfaction threshold to trigger migration, and
- introducing message priorities—to enable, e.g., a migration message to take precedence over other messages.

If an actor is processing messages at a speed faster or equal to the speed that it is receiving them, it can maintain a constant mailqueue. This means that the actor’s current location has enough system resources to handle its tasks. However, when this condition is not met, the messages in an actor’s mailbox begin to accumulate. In this case, the unsatisfied actor attempts to migrate to a more appropriate location. Likewise, if new resources become available in the network, an autonomous actor can choose to migrate to improve overall system performance.

2.4 Secure Actors

Mobile code can pose a serious danger to any environment it executes in and, conversely, any environment can prove dangerous to actors executing inside. Therefore, it is important to consider the security of host resources and actors. Secure actors restrict communication and migration behaviors to actors within specific access control lists.

The access control list for an actor or resource contains every actor allowed to send messages to it. The access control list for a location contains every actor allowed to migrate into the location. These lists can only be altered by the resource or actor in consideration, or by a resident actor in case of passive locations. Using this method, no unprivileged actor can gain access to a resource, since any unauthorized communication or migration request is rejected.
3 Programming Languages and Semantics

3.1 A Simple Actor Language and Its Operational Semantics

Agha, Mason, Smith, and Talcott introduce a simple actor language as an extension to the call-by-value lambda calculus, with primitives for actor communication [7].

The actor language, named here \( AL \), formally defines three primitives:

- \( \text{new}(b) \), which creates an actor which has behavior \( b \) and returns the new actor’s name.
- \( \text{send}(v_0, v_1) \), which sends a message with contents \( v_1 \) to actor \( v_0 \).
- \( \text{ready}(b') \), which signals the end of the current execution and makes the actor ready to receive a new message using behavior \( b' \).

**Actor Configurations** They assume as given two sets \( \text{Atoms} \) and \( \text{Variables} \), and then define the set of values, expressions, and messages, \( M \), as:

\[
\begin{align*}
\text{V} &= \text{Atoms} \cup \text{Variables} \\
\text{E} &= \text{V} \cup \text{app}(\text{E}, \text{E}) \cup \text{F}_n(\text{E}^n) \\
\text{M} &= \langle \text{V} \leftarrow \text{V} \rangle
\end{align*}
\]

where \( \text{F}_n(\text{E}^n) \) is all arity-\( n \) primitives.

Variables are used for actor names. At any given point, an actor can either be ready to receive a message (denoted \( \text{ready}(e) \), where \( e \) is a lambda abstraction); or currently executing some expression \( e \). A message sent to actor \( v_0 \) with contents \( v_1 \) is written as \( \langle v_0 \leftarrow v_1 \rangle \).

An actor configuration is a global snapshot of a group of actors. It includes the concept of an actor mapping, where each actor name is mapped to a behavior; a message set of messages in transit; a set of receptionists (internal actors known to the outside world); and a set of external actors (known actors not in the configuration).

An actor configuration with actor map, \( \alpha \), multi-set of messages, \( \mu \), receptionists, \( \rho \), and external actors, \( \chi \), is written\(^1\)

\[
\langle \alpha \mid \mu \rangle^\rho_{\chi}
\]

where \( \rho, \chi \in \mathcal{P}_\omega(\text{X}) \), \( \alpha \in \text{X} \xrightarrow{f} \text{E} \), \( \mu \in \mathcal{M}_\omega(\text{M}) \), and let \( A = \text{Dom}(\alpha) \), then:

\[
\begin{align*}
(0) & \quad \rho \subseteq A \quad \text{and} \quad A \cap \chi = \emptyset, \\
(1) & \quad \text{if } a \in A, \text{ then } \text{FV}(\alpha(a)) \subseteq A \cup \chi, \text{ and if } \langle v_0 \leftarrow v_i \rangle \in \mu \\
& \quad \text{then } \text{FV}(v_i) \subseteq A \cup \chi \text{ for } i < 2.
\end{align*}
\]

\(^1\) Let \( \mathcal{P}_\omega(\text{X}) \) be the set of finite subsets (Power Set) of \( \text{X} \), \( \mathcal{M}_\omega(\text{M}) \) be the set of (finite) multi-sets with elements in \( \text{M} \), \( X_0 \xrightarrow{f} X_1 \) be the set of finite maps from \( X_0 \xrightarrow{f} X_1 \), \( \text{Dom}(f) \) be the domain of \( f \) and \( \text{FV}(e) \) be the set of free variables in \( e \).
Operational Semantics We define a transition relation between actor configurations as the least relation satisfying the rules in Figure 2. To describe the internal transitions between configurations other than message receipt, an expression is decomposed into a reduction context filled with a redex. The notation $R[e]$ represents a redex $e$ in a reduction context $R$, as described by Honsell et al. and used by Agha et al. For a formal definition of reduction contexts, expressions with a unique hole; and for the definition of functional progress within an actor ($\overset{\Lambda_A}{\quad}$), we refer the reader to [7]. The actor redexes are: newactor($e$), send($v_0$, $v_1$), and ready($v$).

3.2 Mobile Actor Language and Its Operational Semantics

Resources When we introduce the concept of locations, we also introduce the need to model resources in those locations. One example is the standard output stream in a run-time environment. While the concept remains the same, the actual implementation may change when the actor migrates across different locations.

---

We define $\alpha[[e]_a]$ as an extended mapping which maps $a$ into $e$, and all other actor names $a'$ into $\alpha(a')$, and $\alpha[[e]_a, [e']_{a'}]$ as $\alpha\{[[e]_a], [[e']_{a'}]\}$ for $a \neq a'$. 
Each resource is referred to by a universally understood resource name. This resource name is used to contact the implementing resource actor without necessarily ever knowing the specific name of the service providing that resource, or the implementation of that resource. These resources may be ubiquitous, such as a standard output stream. The name 'standard output' may apply to output on a console, or a text field on a graphical user interface, a log file, or even a printer interface, but almost any executing program has access to a primary output stream. If an actor migrates, the primary output stream changes but the transition is transparent to the actor. Referring to some standard resource name allows the environment to handle requests properly.

Resource Maps Resource-to-actor translations are stored in resource maps. These functions are maps between global names and the names of local actors who fill a resource’s roll. When an actor at a location sends a message, resource maps are searched for the target’s name; first the actor’s resource map followed by the location’s resource map. If the target is found, the message is diverted to the resolved actor- otherwise conventional message sending proceeds.

Remote access to resources is permitted because both actors and locations have independent resource maps. Consider the case when an actor migrates but needs to retain a reference to a resource at the original location, such as an output device; the actor’s map will direct messages to the original location, where the resource’s name is resolved to the implementing actor.

Mobile Actor Language To reflect mobility in the universal actor model, we add four primitives to AL, forming the Mobile Actor Language (MAL). MAL thus formally defines seven primitives:

- `new(b)`, which creates an actor which has behavior `b` and returns the new actor’s name.
- `send(v_0, v_1)`, which sends a message with contents `v_1` to actor `v_0`.
- `ready(b')`, which signals the end of the current execution and makes the actor ready to receive a new message using behavior `b'`.
- `newloc(Y)`, which indicates the appearance or creation of a new location, with an initial resource map denoted by `Y`.
- `migrate(l)`, which moves an actor from its current location to one denoted by `l`.
- `attach(v)`, which saves a resource denoted by `v` into the actor’s resource map.
- `detach(v)`, which removes a resource denoted by `v` from the actor’s resource map.

We assume as given two sets At(Atoms) and X(Variables), and we then define the set of values, `V`, expressions, `E`, and messages, `M`, as:

\[ V = \text{At} \cup \text{X} \cup \lambda X.E \cup \text{pr}(V, V) \]
\[ E = V \cup \text{app}(E, E) \cup F_n(E^n) \]
\[ M = \langle V \leftrightarrow V \rangle \]

where \( F_n(E^n) \) is all arity-n primitives.
The set $X$ of variables represents both actors—which includes all resources—and locations. We introduce a set $L$, of locations, with $L \subseteq X$. We also introduce a set $R$, of resource identifiers, with $R \subseteq X$. We modify the definition of $\alpha$ so that $\alpha \in X^L \times (E \times L \times (R^L \times X))$. We extend the definition of an actor configuration to include a mapping, $\pi$, from locations to resource maps, with $\pi \in L \mapsto (R^L \times X)$. A universal actor configuration is written

$$\langle \alpha \mid \mu \mid \pi \rangle^\rho$$

We add another rule to the definition of actor configurations to denote that actor names and locations are disjoint:

\begin{align*}
(0) & \quad \rho \subseteq A \text{ and } A \cap \chi = \emptyset, \\
(1) & \quad \text{if } a \in A, \text{ then } \text{FV}(\alpha(a)) \subseteq A \cup \chi, \text{ and if } <v_0 \leftarrow v_1> \in \mu \text{ then } \text{FV}(v_i) \subseteq A \cup \chi \text{ for } i < 2. \\
(2) & \quad \text{Range}(\alpha) \downarrow_2 \cap A = \emptyset
\end{align*}

**Operational Semantics** We define a transition relation between universal actor configurations as the least relation satisfying the rules in Figures 3 and 4.

### 3.3 Secure Mobile Actor Language

We extend the mobile actor language to form the Secure Mobile Actor Language (SMAL), which defines nine primitives:

- $\text{new}(b)$, which creates an actor which has behavior $b$ and returns the new actor’s name.
- $\text{send}(v_0, v_1)$, which sends a message with contents $v_1$ to actor $v_0$.
- $\text{ready}(b')$, which signals the end of the current execution and makes the actor ready to receive a new message using behavior $b'$.
- $\text{newloc}(Y)$, which indicates the appearance or creation of a new location, with an initial resource map denoted by $Y$.
- $\text{migrate}(l)$, which moves an actor from its current location to one denoted by $l$.
- $\text{attach}(v)$, which saves a resource denoted by $v$ into the actor’s resource map.
- $\text{detach}(v)$, which removes a resource denoted by $v$ from the actor’s resource map.
- $\text{allow}(v)$ changes the actor’s access list to include actor $v$.
- $\text{allowloc}(v)$ changes the actor location’s access list to include actor $v$.

The $\text{allow}$ and $\text{allowloc}$ primitives can receive as an argument a null access list, represented by $\bot$. This indicates the absence of restrictions on messaging and migration.

---

3 $\alpha(a) = e \times l \times r$ implies that actor $a$ has behavior $e$ and is currently executing at location $l$ with resource map $r$.

4 $\downarrow_i$ indicates the projection of a cross product onto its $i^{th}$ coordinate.
We assume as given two sets At(Atoms) and X(Variables), and we then define the set of values, V, expressions, E, and messages, M, as:

\[ V = At \cup X \cup \lambda X, E \cup pr(V, V) \]
\[ E = V \cup app(E, E) \cup F_n(E^n) \]
\[ M = \langle V \leftarrow V \rangle_X \]

where \( F_n(E^n) \) is all arity-n primitives.

The set of variables, X, including resource names, R, and locations, L, remains the same. The structure of M allows selection of valid senders via access control lists. We define a set of access control lists as \( ACL \in P_\omega[Dom(\alpha) \cup \{\bot\}] \). We alter the actor configuration definition to change the actor and location maps, \( \alpha \) and \( \pi \) respectively. The actor configuration is written

\[ \langle \alpha \mid \mu \mid \pi \rangle^\rho_\chi \]

where \( \rho, \mu \) and \( \chi \) are as in MAL, \( \alpha \in X^L(E \times L \times (R^L X) \times ACL) \), and \( \pi \in L^X((R^L X) \times ACL) \). Changes in \( \alpha \) and \( \pi \) reflect the inclusion of access control lists for actors and locations.\(^5\)

\(^5\) \( \alpha(a) = e \times l \times r \times c \) implies that actor \( a \) has behavior \( e \) and is currently executing at location \( l \) with resource map \( r \) and access control list \( c \).
Operational Semantics We define a transition relation between secure universal actor configurations as the least relation satisfying the rules in Figures 5, 6 and 7.

3.4 Autonomous Actors and Reconfiguration Decision Function

Autonomous Actors Autonomous actors are implemented as universal actors that monitor the amount of resources they are using on a peer theater. The values stored by autonomous actors are used for two things; deciding autonomous actor satisfaction and in the reconfiguration decision function, and determining the gain of moving an actor from one peer theater to another. Each peer theater may reset the statistics in an autonomous actor, when the state of the peer theater changes (an actor migrates to or from that peer theater). An autonomous actor keeps track of the number of messages in its mail queue, the number of messages it has sent, processed and received since its last reset,
and the time spent processing and sending messages since its last reset. The actual time an actor spends processing messages is equal to the total time spent processing minus the time spend sending messages, as message sending occurs during message processing. This actual message processing time is what is used by the reconfiguration decision function.

**Autonomous Actor Satisfaction** Actors can be in one of three states: sending, processing, or waiting to receive a message. An actor is considered satisfied if it is processing and sending messages faster than it receives them. An autonomous actor is considered satisfied by a peer theater when the number of messages it receives is less than the number of messages it has processed, or it
<migrate : l>
\[ \langle \alpha \{ \{ R \text{[migrate}(l')], l, r, c \}_a \} \mid \mu \mid \pi \uplus [c', r'] \rangle \_X \Rightarrow \]
\[ \langle \alpha \{ \{ R[l'], l, r, c \}_a \} \mid \mu \mid \pi \uplus [c', r'] \rangle \_X \]
\( \text{if } a \in c' \text{ or } c' = \bot \)

<newloc : Y>
\[ \langle \alpha \{ \{ R \text{[newloc}(Y)]}, l, r, c \}_a \} \mid \mu \mid \pi \rangle \_X \Rightarrow \]
\[ \langle \alpha \{ \{ R[l'], l, r, c \}_a \} \mid \mu \mid \pi \uplus [(a), Y] \rangle \_X \]
\( \text{if } \ell \text{ fresh} \)

<attach : a'>
\[ \langle \alpha \{ \{ R \text{[attach}(a')]}, l, r, c \}_a \} \mid \mu \mid \pi \rangle \_X \Rightarrow \]
\[ \langle \alpha \{ \{ R[\text{nil}], \ell \cup (a' \rightarrow a''), c \}_a \} \mid \mu \mid \pi \rangle \_X \]
\( \text{if } (a' \rightarrow a'') \in \pi(l) \)

<detach : a'>
\[ \langle \alpha \{ \{ R \text{[detach}(a')]}, l, r \cup (a' \rightarrow a''), c \}_a \} \mid \mu \mid \pi \rangle \_X \Rightarrow \]
\[ \langle \alpha \{ \{ R[\text{nil}], l, r, c \}_a \} \mid \mu \mid \pi \rangle \_X \]
\( \text{if } (a' \rightarrow a'') \in r \)

Fig. 6. Secure Mobile Actor Language (SMAL) Semantics, Part II: Mobility

has blocked waiting for a new message. Often if the mail queue is not empty, the number of messages will increase or remain the same. In this case the actor is considered overloaded and may perform more efficiently at a different theater. When this occurs, the peer theater will use the reconfiguration decision function to determine where the autonomous actor should migrate. Since each autonomous actor attempts to increase its sending and processing rate individually, the reconfiguration decision function does not take into account how fast other actors are sending messages to the autonomous actor being evaluated; only the rate the current actor is processing messages as well as the sending rate and the recipient location of messages from the actor.

Reconfiguration Decision Function The objective of the reconfiguration decision function is to determine when it is most optimal to migrate an autonomous actor from one peer theater to another. The best configuration across the entire system of autonomous actors will optimize the flow of messages through the system. The more messages processed and sent during a period of time, the greater the system’s overall performance. Peer theaters monitor how much time they spend not processing or sending messages to determine how much available time they can offer an incoming autonomous actor.

To increase the speed at which an autonomous actor processes messages, the CPU at the Peer Theater where the actor is running and the amount of CPU time used by other autonomous actors running at that peer theater are considered. Peer theaters keep track of their CPU usage and the amount of CPU
used by each autonomous actor, this information is used to determine how fast autonomous actors are running, as well as what available CPU time exists.

Latency and bandwidth are considered to increase the speed at which an autonomous actor sends messages. While it is not reasonable to determine the latency between the peer theater an autonomous actor is running on and each possible new location, it is possible to keep track of the number of messages an autonomous actor sends and the recipient peer theaters of those messages. Local message sending takes a negligible amount of time, so if an autonomous actor migrates to a peer theater where it uses many local references, a large message sending gain is obtained. This helps solve some of the latency/locality problem.

The peer theaters keep track of their maximum bandwidth and the bandwidth in use by each autonomous actor, this information is used to determine how fast each autonomous actor is sending messages.

To determine the gain of processing and message sending comparable values, we take the statistics an autonomous actor has collected since the last time it was reset. From this we know the amount of time an actor spent processing messages and the time spent sending messages. We also know how many messages were processed and how many were sent during that time. We use these values along with the maximum available processing and message sending values to find the average cycles used per processed message, and the average bits sent per sent message. These values allow us to make an estimate of how many messages the actor would have processed on the foreign theater. The difference between these two values is the message throughput gain. If the autonomous actor would have greater throughput on the foreign theater, the gain is positive.
The reconfiguration decision function is as follows, with local peer theater L, foreign peer theater F, T_cpu the cpu on theater T, and T_bandwidth the bandwidth on theater T:

Where, $A_T^p$ is the number of messages processed by actor A on theater T since its last reset, $A_T^s$ is the number of messages sent by actor A on theater T since its last reset, $A_L^{pt}$ is the time spent processing by autonomous actor A on L, and $A_L^{st}$ is the time spent sending messages on L:

\[
CyclesPerMessageProcessed = \frac{(A_L^{pt} * L_{cpu})}{A_T^p}
\]

\[
BitsPerMessageSent = \frac{(A_L^{st} * L_{bandwidth})}{A_T^s}
\]

Therefore, on a foreign theater:

\[
foreignProcessTime = \frac{CyclesPerMessageProcessed}{F_{cpu}}
\]

\[
foreignSendTime = \frac{BitsPerMessageSent}{F_{bandwidth}}
\]

As each message processed sends a certain number of messages (on average we assume $A_L^p/A_T^p$), we add this cost to the cost to process a message:

\[
foreignTotalProcessTime = foreignProcessTime + (foreignSendTime*(A_L^p/A_T^p))
\]

From this, the estimated amount of messages processed on the foreign theater in the same amount of given time, $A_F^p$, with the foreign available time given, is:

\[
A_F^p = foreignAvailableTime/foreignTotalProcessTime
\]

and the total message throughput gain is:

\[
Gain_{MessageThroughput} = A_F^p - A_T^p
\]

4 World-Wide Computer Implementation

The implementation of the World Wide Computer architecture (WWC) consists of three main software components. SALSA provides a programming language for the implementation of actor systems, a WWC run-time component provides a distributed execution environment for SALSA programs, and the Internet Operating (IO) system implementation supports autonomous reconguration through a peer-to-peer infrastructure to keep track of network usage and monitoring.

Figure 8 displays a high-level architecture for the WWC. SALSA programs run at the application layer. IO coordinates run-time environments at the virtual network layer to support autonomous reconguration.
Fig. 8. A three-layered architecture enables dynamic autonomous application reconfigurability. The application layer consists of autonomous actors. The virtual network layer consists of peer theaters. The physical layer is the actual network of computing devices.
4.1 SALSA Programming Language

The programming language component of SALSA provides primitives for message passing. Messages are passed asynchronously between actors each of which is the instantiation of an actor behavior. SALSA programmers define behaviors much in the same way that Java programmers define object classes. Each behavior contains control logic for manipulating the state of an actor within a thread of execution. The language support for coordination includes a variety of continuation primitives.

SALSA programs may be executed in heterogeneous distributed environments. During the compilation process, SALSA code is compiled into Java. This precompilation allows SALSA programs to employ components of the Java class library. It also provides for a homogenous environment for distributed execution. The language includes migration primitives for actor mobility. We refer the reader to [9] for details on the language.

4.2 Theaters and Run-Time Components

The World Wide Computer architecture provides a framework that enables distributed computing over the Internet. Components of the framework communicate using the Universal Actor Naming Protocol (UANP) and the Remote Message Sending Protocol (RMSP). RMSP enables message sending and actor migration between execution environments. UANP provides for communication with naming authorities that map human readable names to actor locations.

Nodes of the WWC that provide execution environments to actors are called theaters. Each theater consists of an RMSP server, a mapping between relative actor locations and executing actor references, and a runtime environment. The theater also contains stationary environmental actors that provide access to stationary resources such as standard output.

The WWC provides a naming service which enables messages to be sent to an actor as its location changes from one theater to another. The naming authority allows UANP clients to subscribe to a specific universal actor name in such a way that changes to the actor's location are pushed to the client. The universal actor naming protocol consists of a small number of message types and is very extensible. We refer the reader to [10] for details on the WWC distributed run-time environment.

4.3 IO Internet Operating System

In order to handle the placement of actors in a large distributed environment, the theaters of the worldwide computer are extended to become peer theaters. This extension allows for the peer theaters to connect to IO’s peer-to-peer network, determine if more system resources are required by their autonomous actors than they can supply, migrate actors to more appropriate locations, and migrate all autonomous actors to new locations in the case of a soft failure. The network allows the individual peer theaters to broadcast to the network asking for
available node resources, and to allow for discovery of new node resources entering the network. Migration of autonomous actors is accomplished by sending a priority migration message to the autonomous actors selected for migration. Autonomous actors are selected for migration through IO’s decision function, in conjunction with the known available node resources. However, in the case of a soft failure, autonomous actors are simply migrated to known neighbors.

As the network is a peer-to-peer system, it allows for the dynamic addition and removal of new nodes, as well as support for node failures. There is no central point of failure which would cause the entire network to crash. IO only attempts to migrate autonomous actors when a peer theater is full, and in this case will only initiate one broadcast to the network. If no peer theater responds it waits for new node resources to become available. This prevents networks with high loads from being bogged down with node resource requests. In fact, in a full network, the broadcast from IO will drop to zero, until new nodes join the network. Thus, IO is stable[11].

IO builds upon the WWC run-time architecture by adding methods for monitoring its available network resources and the states of the actors running on the individual nodes. IO takes the theaters of the WWC and links them as peer theaters in its Peer-to-Peer network, using a simple protocol. This protocol allows a peer theater to poll its neighbors for their available node resources, and to be acknowledged of new node resources when a new peer theater joins the network. The peer theaters use IO’s decision function to decide the placement of the actors in the system for load balancing.

**IO’s Peer-to-Peer Protocol** The simple protocol used by IO’s Peer-to-Peer Network consists of the following packets:

- **New Resource** is broadcast from a peer theater, announcing its connection to the network, and what node resources it can offer. It contains a time-to-live (TTL) value, a message id (to prevent re-broadcasting), a theater location which is the location of the peer theater announcing its entry to the network, an available processing value, and an available message sending value.
- **Resource Request** is broadcast from a peer theater when it becomes full. It requests an amount of node resources to satisfy its most unsatisfied autonomous actor. The **Resource Request** packet contains the theater location of the peer theater requesting node resources, a TTL, a message id, a requested processing value, and a requested message sending value.
- **Resource Response** is sent to the originator of a **Resource Request** when a theater has received a **Resource Request** and can fulfill the node resources requested. It contains available processing value, available message sending value, available time value, and the theater location of those available node resources. After sending one **Resource Response** packet, a peer theater will wait until it receives an **Accept**, a **Reject** packet, or a timeout has been passed before sending another. This prevents a theater from becoming flooded with new autonomous actors.
Accept is sent after a peer theater receives a certain amount number of Resource Response packets (or a timeout has passed), the theater will run its decision function on all the unsatisfied autonomous actors and the information of the available node resources on foreign theaters from the received Resource Response packets. We take the autonomous actor and theater location that yielded the highest positive gain, send an accept message to that theater location, and send a priority message to the autonomous actor to migrate to that location. An Accept packet contains no values.

Reject is sent when an actor, or no actors have been chosen to migrate after the decision function has been run on the unsatisfied actors and all the Resource Response packets. Given the theater locations that the decision function was run on, a Reject packet is sent to all those where an Accept packet was not sent to. If no Accept packets were sent, then a Reject packet is sent to all theater locations. The Reject packet contains no values.

Ping and Pong are used to keep track of latency between theaters. Peer theaters use these to keep its list of neighbors sorted by latency. They both contain a timestamp value. Ping also includes a theater location which is where the return Pong is sent.

Peer Theater Implementation All peer theaters know their own location, stored as a theater location. The theater location is similar to an UAL, with the actor name stripped off. For example, the theater location of an actor at the UAL rmosp://europa.wcl.cs.rpi.edu:5050/SampleActor would be the theater location rmosp://europa.wcl.cs.rpi.edu:5050/.

Peer theaters add the ability to monitor their node resources, by keeping values of their available message sending and message processing capabilities. When these values are needed, the peer theaters poll their autonomous actors and sum up the used node resources. This is subtracted from the theaters maximum available message sending and message processing capabilities to find the available sending and processing values. These values are used in Resource Response packets. When an actor is received by a peer theater, or an actor is migrated from a peer theater, the amount of node resources in use changes. In these instances the peer theater resets the statistics gathered by the autonomous actors running on it, so that the actors can gather information on the new state of the peer theater.

To connect to the IO network, the peer theater first goes through a loading phase, in which it monitors its used node resources, to find available processing and message sending values. It connects to a known peer theater, or peer theaters (which it uses as its initial queue of neighbors). It then uses its available sending and processing values to create a New Resource and broadcasts it to the network. Peer theaters accomplish broadcasts by first creating a message ID. This ID begins as 0 when the peer theater is started, and incremented each time a message is broadcast. The peer theater then broadcasts the message with this message ID and a TTL to all of its neighbors. Each peer theater keeps a hashtable of message IDs with theater locations as their key. When a message is received,
if its message ID is less than the message ID in its hashtable, the message has already been received and broadcast, and will not be broadcast again. If this message ID is not in this hashtable, and the TTL is not zero, the peer theater will decrement the messages TTL, and sent it to all of its neighbors.

Peer theaters trigger migration of autonomous actors for load balancing and fault tolerance in the following three cases:

- **Theater Fullness** When a peer theater is full, it broadcasts a *Resource Request* packet to the IO network, then waits for *Resource Response* packets. With the information gained from the *Resource Response* packets, the peer theater uses its decision function to decide which actor to migrate. A peer theater will only broadcast one *Resource Request*, until one of its actors migrates away. If it receives no satisfactory *Resource Response* packets, it will wait until new node resources become available. This prevents the IO network from becoming flooded with peer-to-peer packets when the network load is high. A peer theater is considered full if it contains unsatisfied actors.

- **New Resources Available** When a Peer Theater joins the network, it will broadcast a *New Resource* packet, letting the network know that new node resources are available. A full peer theater will run its decision function on its unsatisfied autonomous actors with the available data from these, then respond with a *New Resource* packet. The new peer theater then accepts as many autonomous actors as it can without becoming full.

- **Node Failure** When a peer theater is shut down, it will send a priority message to each of its actors to migrate to the peer theater’s closest neighbors. Peer theaters keep a queue of neighbors, sorted by their latency. The autonomous actor’s running on the failing peer theater will each be distributed to those neighbors in a round robin fashion, starting at the peer theater with the lowest latency, to prevent any particular neighbor being overloaded.

**Connecting to IO - Peer Servers** Peer theaters join the IO network by sending their theater location to another theater. The receiving peer theater then adds that peer theater to its list of neighbors, if the latency between them is small enough (decided by *Ping* and *Pong* packets). If any theater locations are previously known by the connector, they may be connected to directly. Also, peer servers have been created as a method of peer discovery. Peer servers are not centralized, as many peer servers may contain information about peers in any one IO network. A peer theater may connect to one or multiple peer servers, which will accept information about the connecting peer theater and then send back a number of theater locations specified by the connecting peer theater.

**4.4 Preliminary Performance Study**

In the preliminary performance study, we look at IO’s ability to load balance many processes over a distributed network. We used a seeded SALSA program to randomly distribute actors over a given set of theaters or peer theaters. Each
actor has a randomly specified message sending value, processing value and message size. These actors are linked together in chains, to simulate many programs running on a network with and without IO support. A chain of actors is a circularly linked list of actors, each of which processes according to their random process load, and send a number of messages according to their random communication load (of size determined by the random size value) to the next actor in the chain. This creates a message sending and processing loop. This is so the chains could run indefinitely while being moved by IO. The WWC with IO was tested with 5 theaters and 5 chains of maximum size 5, and again with 10 theaters and 10 chains of maximum size 5. The same tests with the same seeds were run on the WWC without IO. In both tests, IO showed a considerable improvement in message throughput over time (see Figures 9 and 10. The WWC without IO was used to simulate a normal heterogenous network, while the WWC with IO simulates a heterogeneous network with autonomous reconfiguration.

![Graph](image)

**Fig. 9.** Average Number of Messages Sent over Time, with 5 theaters and 5 chains

5 Discussion and Related Work

The actor model was first created by Hewitt and his group at MIT [12] in the late 1970s. The model has been further developed by Agha [13] and his group at UIUC. Agha, Mason, Smith and Talcott [7] have developed a simple actor language as an extension to the lambda calculus, its operational semantics and
they have studied a family of equivalence relations on actor expressions. Talcott has developed an interaction semantics for actor systems [14,15]. These forms of actor semantics have been the basis of many studies on extensions to the actor model (e.g., for coordination [16–19], real-time [4], software architectures [20], fault-tolerance [21], adaptive and meta-level architectures [22], and artificial intelligence [2]).

Several research groups have been trying to achieve distributed computing on a large scale. Berkeley’s NOW project has been effectively distributing computation in a “building-wide” scale [23], and Berkeley’s Millennium project is exploiting a hierarchical cluster structure to provide distributed computing on a “campus-wide” scale [24]. The Globus project seeks to enable the construction of larger computational grids [25]. Caltech’s Infospheres project has a vision of a worldwide pool of millions of objects (or agents) much like the pool of documents on the World-Wide Web today [26]. WebOS seeks to provide operating system services, such as client authentication, naming, and persistent storage, to wide area applications [27]. UIUC’s 2K is an integrated operating system architecture addressing the problems of resource management in heterogeneous networks, dynamic adaptability, and configuration of component-based distributed applications [28].

Security for distributed systems has been looked at for a number of other agent systems. Safe Mobile Ambients [29] restrict mobile ambients [30] so that sensitive operations such as entering, exiting, and opening an ambient are performed with common agreement. While safe ambients preserve the expressibility
of mobile ambients, they prevent programming mistakes by controlling undesirable grave interferences. The Seal calculus [31], resembles ambients, with two important exceptions. First, seals can only move with the environment’s control, and the $\pi$-calculus is used as a basis for computation, rather than mobility itself. Other methods have been used for securing code on different levels. Ideas such as proof-carrying code [32] and stack inspection [33] are methods of protecting hosts, also discussed in [34].

While there are excellent algorithms for load balancing in clusters and other more static environments, e.g., random stealing and cluster-aware random stealing [35], the dynamic and heterogeneous nature of the nodes on the WWC make such algorithms much less efficient, especially when IO’s peer-to-peer nature is taken into account. Currently, IO’s peer-to-peer network is a variant of Gnutella [36]; however, in the future, implementing IO on top of an already existing peer-to-peer network such as JXTA [37] may prove to be a more interesting option.

References