WORLDWIDE COMPUTING WITH UNIVERSAL ACTORS: LINGUISTIC ABSTRACTIONS FOR NAMING, MIGRATION, AND COORDINATION

BY

CARLOS A. VARELA

B.S., University of Illinois at Urbana-Champaign, 1992
M.S., University of Illinois at Urbana-Champaign, 2000

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

Abstract

The enormous growth of the World-Wide Web has created the opportunity to use the combined computing and communication resources of millions of computers and devices connected to the Internet. The goal of the *World-Wide Computer* (WWC) project is to effectively turn the Web into a unified, dependable, distributed computing infrastructure. The WWC harnesses under-utilized computing resources by providing application programmers with the potential to globally distribute computations. Furthermore, the WWC provides mobile users and remote collaborators with a unified interface to their data and programs. Several applications in multiple domains – as diverse as massively parallel computing, remote collaboration, coordinated computing, and Internet agents – motivate the WWC.

To realize this vision, we develop mechanisms for naming, migration, and coordination of software components and applications running on top of the Web. We represent software components as collections of actors. Actors provide autonomy, simplicity of communication and computation, and a well-developed formal semantics. Therefore, the WWC project uses the actor model of concurrent computation as a basis for studying and implementing different strategies for distributed software component naming, migration, and coordination.

Universal naming is a critical aspect of accomplishing worldwide computing. The WWC's naming strategy, based on Uniform Resource Identifiers (URI), enables transparent migration and interconnection of actors. While a Universal Actor Name (UAN) persists over the life-time of an actor, Universal Actor Locators (UAL) change according to the actor's current location, and prescribe a protocol for communication with the actor.

Data and code migration enable scalability, more efficient network usage, improved graphical user interfaces, and mobile users and resources. Much work has been done on providing transparent access to remote objects, hiding their location information from application programmers, albeit with the unfortunate consequence of hindering efficient network programming. This thesis describes a model for both fine-grained and coarse-grained migration, which enables the development of applications with transparent actor mobility, yet allows programming control on the locality of the participating actors.

Coordination of concurrent computations in the `WWC` is difficult. Traditional coordination mechanisms rely on a shared space, and therefore do not scale to the World-Wide Web. This thesis develops a scalable hierarchical model for coordination. Actors are grouped into *casts* which contain a special actor, designated as the cast *director*. Directors may themselves belong to other casts, creating a coordination forest. Message passing is constrained by the directors of a recipient. All directors of a target actor, up to the first common ancestor with the message sender, need to approve a message before it is delivered: directors above the first common ancestor of two communicating actors need not be interrupted for internal communications. This localization of control enables scalability. One assumption is that actors within the same cast communicate much more frequently than actors across multiple casts. We develop an operational semantics for the hierarchical model of actor coordination to help guide future language and system implementations.

To demonstrate the viability of the proposed naming, migration and coordination schemes, the `WWC` project has developed a programming language (`SALSA`) and its run-time architecture. `SALSA` is an actor-oriented programming language with abstractions for remote messaging, universal naming, migration, and actor coordination through token-passing, join, and first-class continuations. The run-time architecture consists of naming servers and virtual machines. The virtual machines, called *theaters*, provide an environment for execution of universal actors and enable their migration and remote communication using the *Remote Message Sending Protocol*.

The thesis provides experimental results for a few sample programs: three multicast protocols, a messenger carrying remote exception-handling code, a worldwide migrating agent, and a buffering messenger. The `WWC` run-time system, `SALSA` parsing and code generation, and universal actor libraries have been implemented in Java. `SALSA` programs are preprocessed into Java, thus leveraging the existence of compiler technology and safe cross-platform virtual machine implementations.

To Adriana, for her love, patience, and encouragement.

Acknowledgments

First of all, I would like to thank my advisor, Professor Gul Agha, for being an excellent mentor and role model, and above all, for always treating me as a peer and a friend. He has inspired me with his intellectual abilities, and he has motivated me to think about problems in a much more abstract and profound way. I owe him my deepest gratitude for the indescribable impact he has had on my research and on my professional life.

My committee members, Professors Roy Campbell, Ralph Johnson, and Klara Nahrstedt have been a source of inspiration for me and my work. I thank them for their time, comments, and suggestions throughout my thesis research. I would also like to thank professors whose research and teaching have been very motivating during my stay in Urbana-Champaign: Professors Nachum Dershowitz, Herbert Edelsbrunner, Caroline Hayes, and Marianne Winslett. A note of thanks to Dr. Jean-Pierre Briot – at the University of Paris 6 – for his motivation, comments and support during my four-month exchange program at LIP6. Thanks also to Grégory Haïk, one of his Ph.D. students, who implemented part of the naming and messaging infrastructure in the World-Wide Computer. I would also like to thank Dr. Carolyn Talcott – at Stanford University – who provided invaluable comments and timely feedback on the thesis. Finally, John Field, Brent Hailpern, Ganesan Ramalingam, and V. C. Sreedhar – at IBM T.J. Watson Research Center – provided very helpful comments on this research.

The Open Systems Lab research group members, past and present, have provided a lively, highly intellectual atmosphere with countless discussions, ideas, and comments about this research. I would like to especially thank Mark Astley, Nadeem Jamali, Myeong-Wuk Jang, Shangping Ren, Yusuke Tada, Prasannaa Thati, Nalini Venkatasubramanian, James Waldby, Joonkyoo Yoo, Reza Ziaei, and Bill Zwicky. I would also like to ask for forgiveness for those moments when I could not be there, or I was not able to look further enough to try to help. Fabio Kon, even though not at

OSL, provided useful and timely comments on this research. Additional thanks to Bonnie Howard, Julie Legg, and Barbara Cicone for all their voluntary assistance on administrative matters.

My parents, Carlos Arturo and Flor Alba, my grandmother Ana Tulia, and my brothers Javier and Leonardo, have always given me unconditional love and emotional support, critical throughout my graduate studies. I am extremely lucky to share my most wonderful times with Adriana, my wife, who has been a constant source of love, togetherness, happiness, understanding, and encouragement. I dedicate this thesis to her. A special note of thanks also to my in-laws, Alvaro and Lucia, Ricardo, Patty, Scott, William, and my nephews Ethan and Steven. Special thanks to William for proofreading the thesis. Uncles, aunts, and cousins are too numerous to list here, but you know who you are :-)

Last but not least, I am very thankful to the many people who made my graduate life in Urbana-Champaign, a very enjoyable and fruitful experience, one to remember gladly for long times to come, from the salsa dancers to the sky watchers to the chess players to the flying and diving mates. In particular, I would like to thank Roberto, Julie, Oscar, Mary, Betty, Orlando, Chris, Nohra, Kerry, Miguel, Dario, David, Choonkyong, Esmeralda, Rodrigo, Sarah, Bianca, Victor, Pedro, Carlos, Arturo, Annette, Kim, Debra, Giovanna, Mark, Pascale, Shotaro, Frederic, Josh, Ricardo, Cesare, Roseann, Avary, Charlene, Tanya, Mosheh, Phil, Mauricio, Angie, and Cameron.

The National Center for Supercomputing Applications, the Department of Computer Science, the Beckman Institute, and their sponsors, provided an excellent research environment and generous financial support during the years at the University of Illinois at Urbana-Champaign. IBM's T.J. Watson Research Center also provided an excellent environment during the last months of my thesis writing. This research was also made possible in part by support from the National Science Foundation (NSF CCR 96-19522) and the Air Force Office of Scientific Research (AFOSR contract number F49620-97-1-03821).

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CFP Coordination Forest Path

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

LAN Local Area Network

RMI Remote Method Invocation

RMSP Remote Message Sending Protocol

SALSA Simple Actor Language System and Applications

URI Uniform Resource Identifier

URL Uniform Resource Locator

UAL Universal Actor Locator

UAN Universal Actor Name

UANP Universal Actor Naming Protocol

WAN Wide Area Network

WWC World-Wide Computer

WWW World-Wide Web

Chapter 1

Introduction

The World-Wide Web has propelled a revolution of information provision, availability and interconnection, unlike any other before, except perhaps for the printing press. The Web gives us the ability to effortlessly browse documents stored anywhere in the world, easily refer to them and instantaneously publish our own views and documents for everyone to read. The easy and ubiquitous access to information has made it even more valuable. People can now concentrate on information analysis and processing, rather than on the difficulties of getting the information in the first place. The scientific cycle has been streamlined.

The explosive growth of the Internet and the World-Wide Web creates the opportunity to reuse a massive, distributed communication infrastructure to globally perform computations on behalf of potentially mobile users.

1.1 Worldwide Computing

We define *worldwide computing* as an area in computer science and engineering that studies all aspects related to using a wide area network as a computing and collaboration platform. In this thesis, we study the Web and Internet technologies to propose mechanisms enabling the utilization of the international network infrastructure for worldwide computing.

Worldwide computing enables radically different kinds of applications. Traditional assumptions of resource locality and availability become less important. Geographically distributed, possibly specialized devices – such as cameras, credit cards, watches, rings and eyeglasses – can be uniformly accessed and thought of as part of a single, unified *World-Wide Computer*. The World-Wide

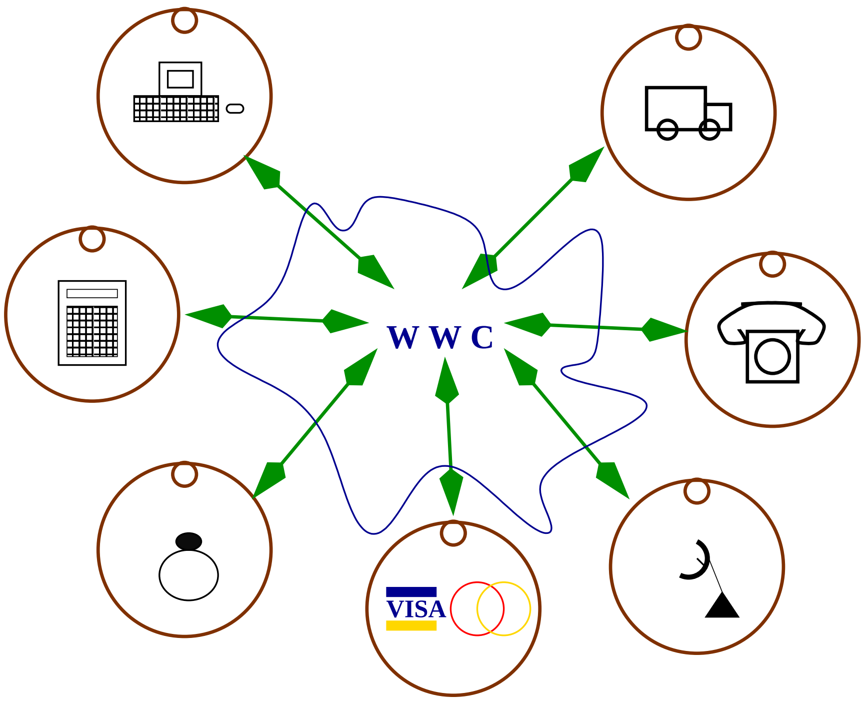


Figure 1.1: The *World-Wide Computer* (`WWC`) provides a uniform view of Internet-connected devices ranging from supercomputers and desktops, to cars, phones, smart credit cards and rings.

Computer (`WWC`) is illustrated in Figure 1.1.[1]

1.2 Problem Description

There are several technical challenges that we need to properly tackle to make the World-Wide Computer a reality. This thesis deals with an important subset of those challenges, namely: universal naming, remote communication, migration and coordination. Our strategy is to provide high-level programming language abstractions to ease the development of scalable worldwide distributed applications. Such abstractions, in conjunction with well-defined architectural run-time components, free application developers from programming low-level communication, naming, location and coordination protocols.

[1]Berners-Lee has previously used the *World-Wide Computer* term to refer to the *semantic Web*, an extension to the Web with documents prone to analysis by both humans and computers [13]. Our use of the term refers to a specific Internet-based infrastructure for worldwide computing.

1.2.1 Linguistic Abstractions for Worldwide Computing

One key ingredient in realizing the `WWC` is a platform-independent programming language and architecture which enables safe remote code execution and dynamic coordinated interconnection of open, mobile, asynchronous and distributed software components.

The Java platform has been an important step in this direction [43, 60]. Specifically, it defines a platform-independent virtual machine with bytecode compatibility, and a sandbox security model for executing classes dynamically downloaded from the network. Java also provides important high-level APIs for remote method invocation, object serialization and reflection [76]. However, although these facilities make Java a very promising technology to realize the `WWC`, its main drawback is at its very core, namely, Java's object model. Specifically, Java's object model has several deficiencies for worldwide computing:

- **synchronous interaction**: unnecessary sequencing of operations and potential deadlocks due to remote object failures.
- **passiveness**: multiple threads within an object create a potential for state corruption.
- **shared memory**: creates the need for complex coherence protocols, especially under object migration and serialization.
- **non-universal naming**: lack of a suitable mechanism to refer to mobile objects worldwide, and to dynamically obtain object references from their names, independently of their current location.

1.2.2 Universal Naming

Worldwide computing systems require a scalable and global naming mechanism. Moreover, the naming mechanism must facilitate object mobility; this implies that the object name should completely abstract over the location of an object, so that migration does not break existing references. Contrast this to the Web infrastructure, which uses location-dependent references (URLs) thereby inhibiting transparent document relocation.

Because of the heterogeneous nature of devices connected to the Internet, an object naming mechanism should also be platform independent. Furthermore, because objects are different in nature and use different protocols for communication, a name should provide openness by including the protocol (or a set of protocols) that are required to communicate with the object.

Three additional important characteristics of naming in Internet-based systems include efficient resolution, safety and human readability. Safety implies that it is not possible to "steal" messages by creating an object with an existing name. Human readability of object names implies that it is possible to "make" and "guess" object names, similarly to Web document URLs today.

1.2.3 Mobility

There are three types of mobility in distributed systems: resource migration to improve locality of access, code migration for dynamic applications behavior, and user mobility – multiple points of application access.

When applications use mobile resources, it is convenient if the programming language provides transparent access to those resources – i.e., programmers need not be aware of the current location of those resources. However, such transparency should not imply that programmers cannot possibly know or change the location of those resources – for example, to improve locality of access.

A programming language and run-time infrastructure for worldwide computing is better off enabling both fine-grained and coarse-grained migration. Migrating groups of correlated entities – i.e. objects with high frequencies of communication, or code for classes in the same hierarchy – hides the latency of pair-wise interactions which in turn, reflects on better overall application performance.

User mobility implies that applications may be accessed from different locations and using heterogeneous devices. Universal naming provides transparent access to resources in a location and device independent way. Coarse-grained migration enables applications to adapt or reconfigure themselves for more efficient worldwide access.

1.2.4 Coordination of Concurrent Activities

In addition to naming, communication and migration issues, worldwide computing application developers have to deal with the increasing complexity of coordination of concurrent activities in those applications. Coordinating concurrent, distributed and mobile activities in inherently asynchronous systems is a difficult task. Traditional coordination mechanisms rely on a shared space, and therefore do not scale to worldwide execution. Many coordination models rely on reflection, and most often coordination and communication code are intermixed in applications.

Reflective approaches to coordination enable the customization of basic operations; for example, one can constrain operations on a particular object according to the object's state or the state of the system as a whole. Additionally one can constrain the creation of other objects, or temporarily constrain message sending to particular objects. While a fully reflective approach can provide a lot of flexibility, its implementation often requires a meta-level infrastructure that obscures the semantics of the base-level coordinated applications.

Another limitation of coordination mechanisms is their lack of modularity. Current software development methodologies and tools do not provide with the necessary abstraction mechanisms for separating multiple interrelated design concerns at the implementation level. Instead, it is very common to see code for functional properties intermixed with code for non-functional properties, including distributed activity coordination.

1.3 Approach

Our approach to worldwide computing has been to use a well-founded model of concurrent and distributed computation – namely the *actor model* [1, 44] – to study naming, remote communication, migration and coordination.

The thesis contributions are three-fold. First, we develop a programming language with high-level constructs for building actor behaviors with Internet-wide access. Second, a run-time infrastructure enabling the worldwide execution of programs developed using the language. And finally, a scalable model for coordination of concurrent activities in a distributed environment, based on a hierarchical component organization.

1.3.1 Actor Model

Actors extend sequential objects by encapsulating a thread of control along with procedures and data in the same entity; thus actors provide a unit of abstraction and distribution in concurrency. Actors communicate by asynchronous message passing (see Figure 1.2). Moreover, message delivery is weakly fair – message delivery is guaranteed for actors infinitely often ready to receive messages. Unless specific coordination constraints are enforced, messages are received in some arbitrary order which may differ from the sending order. An implementation normally provides for messages to be buffered in a local mailbox and there is no guarantee that the messages will be processed in the same order as the order in which they are received.

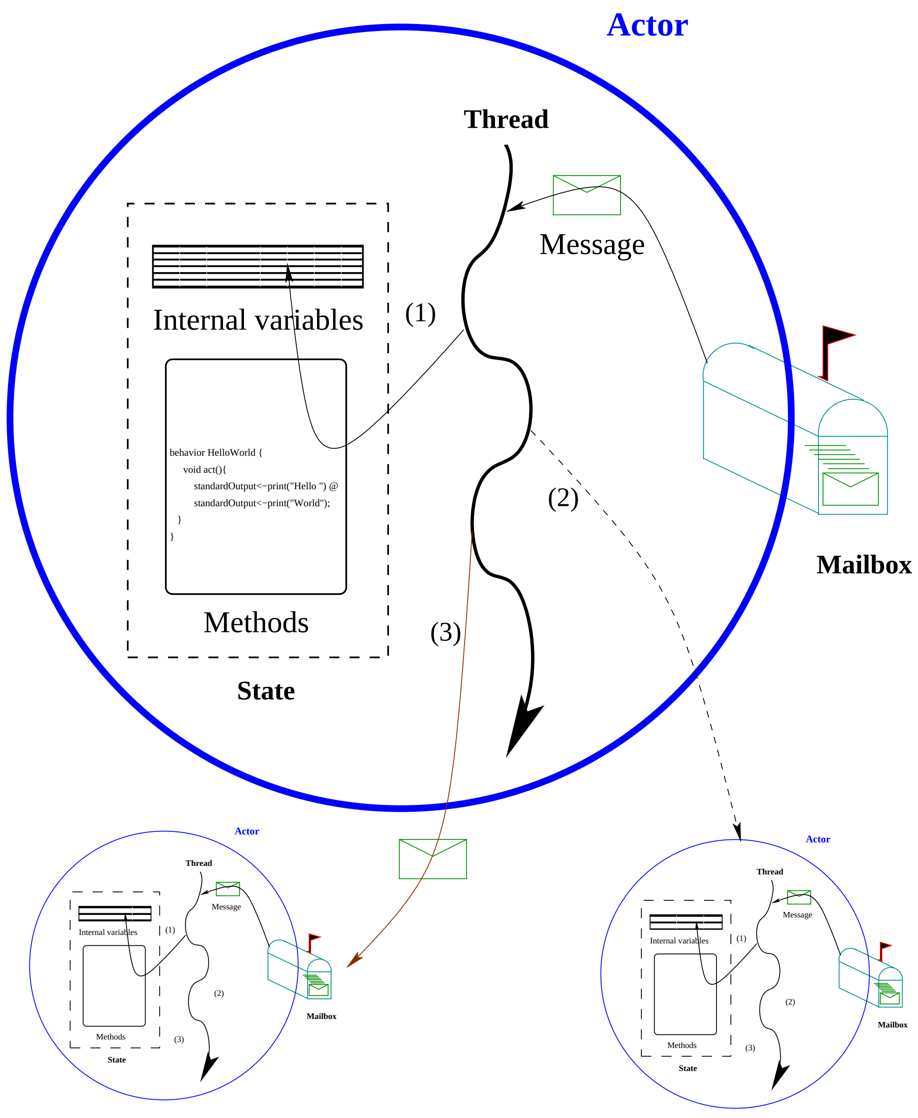
Actors are inherently independent, concurrent and autonomous which enables efficiency in parallel execution [54] and facilitates mobility [5, 6]. The actor model and languages provide a very useful framework for understanding and implementing open distributed systems. For example, among other applications, actor systems have been used for enterprise integration [81], real-time programming [70], fault-tolerance [4], and distributed artificial intelligence [36].

We develop an extension to actors consisting of providing unique uniform names to actors executing in the WWC. Our naming strategy enables remote communication with transparent actor migration and dynamic, unanticipated creation of actor references from their names. We call such actors, *universal actors*, and their names, *Universal Actor Names* (UAN).

1.3.2 SALSA: A Universal Actor Programming Language

One potential programming language for implementing universal actor systems is Java [43], which provides platform compatibility through the use of a virtual machine and supports multithreading. However, Java uses a passive object model in which threads and objects are separate entities. As a result, Java objects serve as surrogates for thread coordination and do not abstract over a unit of concurrency. This relationship between Java objects and threads is a serious limiting factor in the utility of Java for building concurrent systems [85]. Specifically, multiple threads may be active in a Java object and Java provides only the low-level *synchronized* keyword for protecting against the threads that are manipulating an object's state simultaneously. It would be better to provide higher-level linguistic mechanisms for characterizing the conditions under which object methods may be invoked. Java programmers often overuse *synchronized* and resulting deadlocks are a common bug in multi-threaded Java programs.

Java's passive object model also limits mechanisms for thread interaction. In particular, threads exchange data through objects using either polling or wait/notify pairs to coordinate the exchange.



```
behavior HelloWorld {
    void act(){
        standardOutput<-print("Hello ") @
        standardOutput<-print("World");
    }
}
```

Figure 1.2: In response to a message, an actor can: (1) modify its local state, (2) create new actors, and/or (3) send messages to acquaintances.

In decoupled environments, where asynchronous or event-based communication yields better performance, Java programmers must build their own libraries which implement asynchronous message passing in terms of those primitive thread interaction mechanisms.

Although actors can greatly simplify coordination and are a natural atomic unit for system building, they're not directly supported in Java. A number of libraries and frameworks have been created (e.g. [66, 20]) for the development of actor systems. However, there are several advantages to using an actor programming language instead of an actor library or framework.

- First, certain semantic properties can be guaranteed at the language level. For example, an important property of actors is that they provide complete encapsulation of data and processing within an actor. Ensuring that multiple threads are not actively mutating shared state is useful in guaranteeing safe and efficient actor migration.

- Second, by generating code from an actor language, it is possible to ensure that proper interfaces are always used to create and communicate with actors. In other words, programmers cannot incorrectly use the host language which has been used to build an actor library or framework.

- Finally, using an actor language improves the readability of programs developed. Writing actor programs using a framework often involves using constructs of the defining language to simulate actor operations. For example, one may use method invocation to perform actor creation, or message sending. It is unnatural for programmers to continually relate the actor semantics with the syntax of the defining language.

We develop a programming language called `SALSA` (Simple Actor Language, System and Applications) [67]. `SALSA` is a dialect of Java designed to implement `WWC` applications – it provides abstractions for actor programming, universal naming, remote asynchronous communication, migration and coordination.

`SALSA` specifically provides two actor coordination mechanisms. First, application programmers can specify a customer for a given computation's return value – which we call a *token*. Customers can be chained to create a workflow-like style of concurrent programming. Second, join continuations allow concurrent computations to rendezvous into a single customer continuation, enabling the easy development of more complex interaction patterns such as different multicast and multiple phase commit protocols.

To ease the technology transition path, `SALSA` programs are currently preprocessed into Java source code with at least two obvious advantages. First, we leverage the current investment in Java compilation and execution technologies. And second, we view a heterogeneous, insecure, network of physical machines as a relatively homogeneous and secure network of Java virtual machines.

1.3.3 World-Wide Computer

The World-Wide Computer (`WWC`) is a global distributed infrastructure enabling naming, communication and migration for universal actors. `SALSA` programs can execute in a single virtual machine or on the `WWC` infrastructure. The `WWC` consists of a set of virtual machines hosting universal actors – called *theaters* – and a set of naming servers.

Universal actors are reachable Internet-wide through their globally unique *Universal Actor Names* (UAN) – identifiers which persist over the life time of an actor, and can be used to obtain an authoritative answer regarding the actor's current Internet location, or theater. *Universal Actor Locators* (UAL) uniformly represent a reference to an actor within a specific theater as well as a protocol for communication with that actor.

The *Remote Message Sending Protocol* (RMSP) defines how an actor sends a message to any other actor in the World-Wide Computer. RMSP allows an actor to send to another actor, specified by a UAN or a UAL: (1) a `SALSA` message containing a method, its arguments, and optionally a token-passing continuation, (2) a messenger – an actor that receives a `deliver()` message upon entrance to the theater, or (3) a *cast* – a group of interrelated actors. RMSP uses a specialized version of Java object serialization.

A messaging component (`rmspd` – for RMSP dæmon) in a theater performs the actual delivery of a message from a remote actor. The same component is used for actor migration across theaters. Naming servers (`uand` – for Universal Actor Name dæmons) provide the mapping between UANs and UALs, using a simple text-based naming protocol, the *Universal Actor Naming Protocol* (UANP).

1.3.4 A Hierarchical Model for Coordination

We introduce a scalable model for actor coordination, as a basis for reasoning about and building worldwide distributed systems.

We model coordination hierarchically by grouping actors into *casts*. Casts are meant to be abstraction units for coordination, naming, migration, synchronization and load sharing. Each cast is coordinated by a single actor designated as *director*. Coordination in the hierarchical model is accomplished by constraining the receipt of messages that are destined for particular actors. An actor can only receive a message when the coordination constraints for the message reception are satisfied. The coordination constraints are checked for conformance by the cast *directors*.

The hierarchical model [86] is motivated by human organizations, where groups and hierarchical structures allow for effective information flow and coordination of activities. The near-decomposability property of hierarchic systems also allows for their natural evolution. Simon [72] illustrated the argument with examples drawn from systems such as multi-cellular organisms, social organizations, and astronomical systems. Regardless of the philosophical merits of Simon's proposition, we believe that a hierarchical organization of actors for coordination is not only fairly intuitive to use but quite natural in many contexts. By using a hierarchical model, we simplify the description of complex systems by repeatedly subdividing them into components and their interactions.

We define an operational semantics for a simple language providing hierarchical actor coordination.[2] Following Agha, Mason, Smith and Talcott [7], we use as a language an extension to the call-by-value lambda calculus and define a transition system between actor configurations. There are two motivations for providing an operational definition. First, it provides clarity in the model specification. And second, it can guide future coordination language and system implementations.

1.4 Contributions of this Research

There are three main contributions in this thesis:

1. The `SALSA` Actor Programming Language.

[2]A full theory would need to include methodologies for testing equivalences of actor expressions and configurations.

We design and implement `SALSA` – a high-level programming language for building software components to be executed on the Internet. Software components are modeled as groups of *universal actors*. Application programmers can easily provide globally unique names to universal actors; migrate them efficiently and transparently to other computers on the Internet; obtain references to software components via asynchronous messages; and coordinate concurrent activities via token-passing and join continuations.

2. The *World-Wide Computer*: An Architecture for Worldwide Computing.

   We define the *World-Wide Computer* – an Internet-based worldwide computing infrastructure. We specify universal actor naming and messaging formats and protocols, and we demonstrate their feasibility with reference implementations. We measure the performance of the `WWC` with a test-bed including hosts in the U.S., France and Japan.

3. The Hierarchical Model for Coordination of Concurrent Activities.

   We introduce *casts* and *directors* as abstractions for coordinating activities in an actor-based model of computation. The hierarchical model provides scalability by localizing coordination control. We define an operational semantics for the model based on transitions on actor configurations.

1.5 Related Work

We provide a global overview of related work. For detailed comparisons to specific research, we refer the reader to the related work sections in the naming, remote communication and migration, programming language, worldwide computing infrastructure, and coordination chapters.

1.5.1 Universal Naming

ActorSpaces [22] is a model which provides the equivalent of a Yellow Pages service – communication uses an abstract pattern-based description of groups of message recipients. *Smart Names* or *Active Names* (WebOS) provide scalability of read-only resources in the World-Wide Web by enabling application-dependent name resolution in Web clients [83]. The 2K distributed operating system [56] builds upon and enhances CORBA's naming system [46].

1.5.2 Remote Communication and Migration

The Common Object Request Broker Architecture (CORBA) [65] has been designed with the purpose of handling heterogeneity in object-based distributed systems. Sun's JINI [88] architecture has a goal similar to that of CORBA; the main difference between the two is that the former is Java-centric. One of the main components of JINI is the Remote Method Invocation (RMI) [77]. JavaSpaces [78] is another important component of JINI that uses a Linda-like [26] model to share, coordinate, and communicate tasks in Jini-based distributed systems.

Emerald [49] was one of the first systems including fine-grained migration. Obliq [24] is a lexically-scoped, untyped, interpreted language, with an implementation relying on Modula 3's network objects. Java [43] was the first programming language allowing Web-enabled secure execution of remote mobile code. Such mobile code – called *applets* – is downloaded, verified and interpreted in a virtual sandbox protecting the executing host from potentially insecure operations: e.g., reading and writing from secondary memory and opening arbitrary network connections. IBM Aglets [58] is a more recent framework for the development of Internet agents which can migrate, preserving their state.

Cardelli's Mobile Ambients [25] model object group migration through different administrative domains (e.g. through firewalls). Sekiguchi and Yonezawa [71] describe a flexible calculus for comparing mechanisms of code, data and execution state migration.

1.5.3 Actor Programming

There are several libraries and frameworks developed in an object-oriented language to implement the Actor model of computation. Three examples are the Actor Foundry [66], Actalk [20] and Broadway [73]. Several actor languages have also been proposed and implemented to date, including Rosette [82], ABCL [91], Concurrent Aggregates [29], and Thal [53].

1.5.4 Worldwide Computing

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 [9], and Berkeley's Millennium project is exploiting a hierarchical cluster structure to provide distributed computing on a "campus-wide" scale [21]. The Globus project seeks to enable the construction of larger *computational grids* [37]. 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 [28]. WebOS seeks to provide operating system services, such as client authentication, naming, and persistent storage, to wide area applications [83]. 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 [56].

1.5.5 Coordination

A number of coordination models and languages [31, 42] use a globally shared tuple space as a means of coordination and communication [26]. Reflective models approach coordination by intercepting and controlling base-level operations (e.g., see [63, 74, 89, 91]). ActorSpaces [22] are computationally passive containers of actors. Synchronizers [38, 39] are linguistic constructs that allow *declarative* control of coordination activity between multiple actors.

Agha, Mason, Smith, and Talcott define an operational semantics and an equivalence theory for open, composable configurations of actors [7]. Interaction semantics uses diagrams and paths for reasoning about sets of sequences of interactions of components (actor configurations) with their environment [79, 80]. We extend Agha, Mason, Smith, and Talcott's operational semantics to reason about coordinated actor configurations.

1.5.6 Software Engineering of Distributed Systems

Current software development methodologies and tools do not provide the necessary abstraction and modularity mechanisms for reliable and fast distributed systems development. A number of researchers, including the author's group members, have argued for a separation of code implementing the functionality of software components (the *how*) from code for "non-functional" properties. Non-functional properties include the relation between otherwise independent events at different actors (the *when*) [3, 39, 74], and the mapping of actors on a particular concurrent architecture (the *where*) [68]. This separation of concerns results in modular code which allows software developers to reason compositionally and to reuse code implementing functionality, coordination, placement policies, and so forth independently.

The software engineering community has worked on architecture description languages (ADLs) to simplify the development of complex systems. Examples include Wright [8], Rapide [61] and DCL [10]. In DCL, components and connectors to describe a software architecture can be specified. Components represent the functional level, while connectors are lower-level abstractions which define how an architecture is deployed in a particular execution environment.

Much research and development in industry has focused on open distributed systems; some examples include CORBA [65] and Java-related [43] efforts (e.g., see RMI [77], JavaSpaces [78], JINI [88], and concurrency patterns [59, 41]). Java transforms a network of heterogeneous machines into a network of homogeneous virtual machines, while CORBA works on facilitating interactions between heterogeneous environments [2].

1.6 Thesis Outline

Chapter 2 presents motivating scenarios where worldwide computing applications could be useful, highlighting the needs for universal naming, migration and coordination. Chapter 3 introduces the naming requirements imposed by worldwide computing systems. Chapter 4 presents a general model for actor and cast migration. Chapter 5 introduces `SALSA`, its coordination abstractions, examples of simple applications developed in the language, and algorithms used for Java code generation. Chapter 6 describes the World-Wide Computer architecture, including the run-time system for universal actors, the Remote Message Sending Protocol, the naming service and actor migration. Chapter 7 describes examples that use `SALSA` and the `WWC`, and gives a preliminary performance study of the presented worldwide computing architecture. Chapter 8 presents the hierarchical model for actor coordination and its operational semantics. The last chapter concludes with a discussion and potential future research directions.

Chapter 2

Application Scenarios

We introduce a sample set of motivating applications for worldwide computing platforms like the *World-Wide Computer*. The motivating applications are divided into four broad categories: massively parallel divide-and-conquer problems, collaborative applications, coordinated computations, and network agents for personalized services and electronic commerce. Our categories, and indeed our applications, are not meant to be exhaustive but rather representative. We will discuss each one of these categories in the following sections.

2.1 Massively Parallel Divide-and-Conquer Problems

Several projects use the Internet or the Web as a supercomputing infrastructure in order to solve problems which can be easily or naturally parallelized – problems for which the main algorithm is a divide-distribute-and-conquer strategy. The main problem is decomposed into several sub-problems, which are given to multiple sub-networks or machines to solve. The solution to the larger problem is integrated from the partial solutions computed in a distributed fashion.

One potential way to attack massively parallel divide-and-conquer problems in a worldwide computing infrastructure, is what we call *vampire computing*. *Vampires* are mobile actors which carry a computationally expensive goal and use the power of idle workstations worldwide by "hiding from the sun." Assuming that most workstations in the world are idle in the local times of 11pm to 5am, there is always a large region in the world which has such local time (6 hour time span) and therefore, there is always a chance for vampires to move there and consume local computational power.

Notice how both the divide-distribute-and-conquer strategy and the vampire computing strategy require data and code migration as a basic service provided by the programming language and run-time architecture used to solve the problem.

We describe three particular applications in the massively parallel category: finding cryptographic keys, searching for extraterrestrial intelligence, and rendering complex images.

Crypto-Analysis: Finding Decryption Keys

Many modern encryption algorithms use pairs of keys to encode and decode data that wants to be protected. A pair of keys consists of a *public* key and a *private* key. The key pair has the property that data encrypted with one key can only be decrypted with the other one. This mathematical property has the advantage that the encryption algorithm itself can be public, making it widely known and studied, and therefore, presumably less vulnerable to attacks.

The confidentiality of the encrypted data depends on the privacy level of the private key, as well as the intractability of guessing the private key. The larger the space of keys, the longer it takes to guess a particular key by trying out all possible key values without the use of any heuristic function. Typical encryption keys are 128 bits or larger, creating a space of at least 2^{128} potential keys.

The `distributed.net` project has been using Internet computers' idle-time since 1997 to break encryption keys by trying out all possible keys. A special client written in C++ is distributed in binary form. As of 2000, there are more than 60,000 individuals who have participated in the project. RSA Inc. presents open challenges for different key sizes and algorithms. The owner of the workstation that finds the right key shares an economic incentive associated with the challenges.

SETI: Search for Extraterrestrial Intelligence

Human kind has been looking at space for centuries in search of knowledge, clues as to the origin of life, stars and planetary systems, as well as the possibility of extraterrestrial intelligence. Satellites, antennas, and reception devices gather vast amounts of data coming from the infrared to the ultraviolet spectrums of light.

Large computing capabilities are required to analyze space data in search of patterns and potential "signals" from intelligent life in outer space. The larger our data gathering and analyzing capabilities, the more wavelengths and points in space we can "look at", in search for potentially very valuable information.

With this goal in mind and to promote a wide participation, Berkeley's Search for Extra Terrestrial Intelligence (SETI) project has used idle computers from participants around the world with two different programs: "SETI at work" and "SETI at home" [75]. As of 2001, there are more than two million users who have participated in the project.

Rendering Complex Images and Animations

Ray tracing is a technique to create realistic digital images, by modeling objects, light sources and cameras (or "eyes") in a three-dimensional space. The rays of light originating in light sources are traced across three-dimensional space until they collide with scene objects, and bounce back illuminating other objects and so on. Ray tracing produces excellent images but its computational requirements are very high. Computational requirements in straightforward implementations explode when the goal is to produce animations with moving objects.

Ray tracing can be naturally decomposed into subproblems and implemented in parallel by computing different regions of a given image in a distributed manner. Similarly, different animation sequences can be computed independently and combined into a single large animation.

Javelin [30] has used Java applets and a work balancing strategy – similar to a work stealing algorithm introduced by MIT's Cilk project [16] – to render complex images using anonymous Internet computers.

Contributions in this Domain

This thesis presents three advantages over previous mechanisms to distribute computations over the Internet. First, a scalable model for coordinating results in a distributed fashion – using directors – thereby avoiding the problem of centralized control (see Chapter 8). Second, a high level language for migrating executing code – `SALSA` – which nonetheless, produces Java to leverage the existence of virtual machine implementations in multiple heterogeneous platforms (see Chapter 5). Third, a tight integration between the programming language and the run-time infrastructure, enabling the deployment of `WWC` applications on multiple *theater* implementations – e.g., an applet theater, a servlet theater, or an application theater (see Chapter 6). Section 5.4 presents sample pseudo-code that can be used as a starting point for implementing massively parallel divide-distribute-and-conquer applications.

2.2 Collaborative Applications

Another broad category of worldwide computing applications are those employed for human cooperation. In this particular context, the goal is to enable remote collaboration among distributed users, providing a single interface to their data. The "What You See Is What I See" (WYSIWIS) strategy consists of providing the illusion of a single computer running a single program, even though the users may be remotely located, and the data and code distributed as well. There are variations of WYSIWIS, in which some participants see different information or have control on different actions at different times; for example, in a checkers game, only one user has the ability to make a move at a given point in time.

We describe some collaborative applications motivating the worldwide computing infrastructure presented in the thesis: virtual meeting rooms, shared electronic blackboards, voting tools, calendar sharing tools and multi-user domain games.

Virtual Meeting Rooms

`talk` and Internet Relay Chat (IRC) were among the first collaborative applications on the Internet. Users sharing a common theme gather together in a virtual "room", where they can discuss and exchange information about a particular topic in real time.

To support IRC rooms, a server listens to messages from IRC clients and multicasts the messages to all other parties connected to the given server and talk session. Each user is identified with a "handle" – which along with messages from that user – gets displayed in the consoles of all other participants in that room. Different clients and servers provide more advanced features, such as the ability to send a message to a single user – as opposed to everybody in the room – and to restrict access to a room to a particular set of users.

More recent virtual meeting applications like ICQ[1] (for "I seek you") enable users to easily keep track of a list of collaborators and to send them messages while they are both online and off-line. Furthermore, users can specify their preferences in order to provide different degrees of privacy and reachability.

Shared Blackboards

More advanced collaborative tools which enable sharing not only text but also different multimedia formats such as sound, images and video, have been developed to operate over the Internet. In particular, shared electronic blackboards give the illusion of a single blackboard that is shared among users who are potentially distributed around the world.

NCSA Habanero[2] is a framework enabling the development of collaborative Java applications, or *hablets*. Sample hablets include a shared blackboard, shared Web browsing, virtual meeting rooms, and checkers. The framework provides a generic graphical user interface with automatic session management including: ground control, user authentication, session encryption, recording and replaying.

Microsoft Netmeeting[3] is a Windows-based application which enables users to share a blackboard and to collaborate using sound and video over the Internet. Netmeeting uses particular servers as centralized points of contact for collaborative sessions and provides directory services to locate and contact users.

Electronic Voting

While several electronic voting applications can be found today in the context of newspapers and magazines in the World-Wide Web, there are some important problems that must be solved before electronic voting can be used in more serious domains – e.g., presidential elections. Specifically, it is critical to prove that every vote corresponds to a valid person, and that at most one vote per person is recorded. One possible solution is to promote the wide availability of electronic authentication and authorization protocols.

[1]http://www.icq.com/
[2]http://havefun.ncsa.uiuc.edu/habanero/
[3]http://www.microsoft.com/windows/netmeeting/

Calendar Sharing and Meeting Scheduling

Many office applications – like Lotus Notes[4] – include the possibility of sharing a calendar and comparing the calendars of several collaborators to find common empty slots in order to schedule meetings. Often, calendar applications need to work together with room scheduling applications, to ensure that there is an appropriate physical facility for a meeting to take place. The main problem is that office applications often cannot communicate with other applications, because of heterogeneity, security, or even reachability constraints.

Multi-User Domain (MUD/MOO) Games

In multi-user games, different players assume different roles and the possible actions which a user may take depend on the user's current role, as well as the rules of a particular game. Current game servers such as Yahoo! Games[5] are centralized. If a server fails, all games hosted at that server must be restarted. Even though there are checkpointing mechanisms which enable resuming a particular game at a given point in time, a distributed game server could be much more resilient to failures. One initial possibility for distribution is to set up different areas of a game into different computers with some degree of replication to provide for recoverability.

Contributions in this Domain

Collaborative applications can be written to make use of a universal naming scheme to enable remote users, as well as programs and services to easily and dynamically find each other and communicate. In Chapter 3, we propose such a universal naming scheme.

Furthermore, layered infrastructures for remote communication can enable efficient local interactions along with LAN and WAN remote communications. In Chapter 4, we propose the Remote Message Sending Protocol, which updates references of migrating actors so as to speed up communication with co-located universal actors.

As a canonical example, Sections 3.2.2, 4.2.2 and 4.2.3 show how to share a calendar, get access to a remote calendar, and migrate a calendar. The applications – written in the `SALSA` programming

[4]http://www.lotus.com/notes/
[5]http://games.yahoo.com/

language – use the universal naming, remote communication, and migration schemes, presented in the thesis.

2.3 Coordinated Computing

Some distributed computing problems differ in fundamental ways from the embarrassingly parallel problems presented in Section 2.1. While the computation-to-communication ratio between sub-problems in those massively parallel applications is very high, in other applications the ratio may be much lower – requiring better network connectivity and bandwidth and creating the need for scalable coordination of independent autonomous components.

We describe four example applications in this domain: search in game playing, distributed web search engines, distributed transactions, and peer-to-peer computing.

Search in Game Playing

A typical search example is found in computer chess playing. An evaluation function computes the value of a given board position statically. A move generation algorithm produces a set of valid moves from a given position. The goal is to compute the move that optimizes (maximizes) the value of the evaluation function, assuming the opponent will play its best move as well (the one that minimizes the evaluation function.) Different depths of search in the tree of moves produce different "best" moves. Time is usually an important constraint and the more computational power that is available, the deeper that the search can be performed.

The problem in this particular domain is that the space of search grows exponentially: there are in average 35 valid moves in a given chess board position (the branching factor). Therefore, it becomes computationally intractable to evaluate all possible moves until a check mate is found. In practice, algorithms such as alpha-beta pruning, use an (α,β) interval of "plausible" values, which avoids computing the search trees for branches which are known to produce worse results than the paths that have been already explored.

With alpha-beta pruning, it is possible to double the depth of the trees being searched [55]. The success of branch pruning depends upon the order of evaluation of valid moves. If "better" moves are evaluated first, more branches can be pruned. However, the best moves are not known in advance; otherwise, there would be no need to search at all. Iterative deepening algorithms use alpha-beta pruning up to a fixed depth in the same tree, to produce an ordered move set, which is used as an input to the next iteration of search, drastically reducing the branches to analyze and therefore the overall computing time.

IBM's Deep Blue won a short match against the world chess champion Garry Kasparov in May 1997. There were two components to Deep Blue's success. First, it used standard computer chess techniques including an opening and final game database, transposition tables, and different heuristic functions for different parts of the game. Second and most importantly, the strategy was the exclusive, dedicated use of domain-specific hardware to compute the next moves and to evaluate board positions very rapidly: Deep Blue can analyze 200 million chess positions per second and search to depths of 14 levels when making a move.

Other games such as Go, with branching factors in the hundreds, currently make it difficult for computers to beat human world champions, using the same strategies as Deep Blue. Leveraging the Internet as a computing platform may enable search applications to become feasible as individual computers become faster and network latencies and bandwidths improve.

Distributed Web Search Engines

The World-Wide Web is growing exponentially, which makes information indexing and searching an increasingly challenging problem. The traditional approach to Web indexing is to use a specialized Web client – called software robot or "softbot" – to fetch a remote document, index it, and find internal references to other documents which can be fed to the softbot to start the process again. Data is often stored in a central location, which provides access via keyword searches. For example, Google[6] – as of April 2001 – indexes over 1.3 billion pages in the Web.

As the number of documents in the Web grows from billions and trillions, the indexing process becomes more and more expensive. Search results based on keywords, often result in thousands of documents, with difficulty filtering the desired information from irrelevant links. Similarly, when documents change or disappear, searching with indices produces incorrect results, because of the low frequency of the updates and the difficulty of distinguishing stale references from temporary

[6]http://www.google.com/

server failures.

There are alternative approaches to indexing the Web. One approach evaluates the relevance of documents by counting the number of external references – i.e., those coming from a different Web server. Another approach gives higher values to results that are selected frequently by users who are searching for similar information. More relevant documents are returned first in subsequent queries.

The World-Wide Web Consortium has been proposing markup languages for metadata, which among other things, would enable a smarter way to index information, based on information provided by document authors, as opposed to pure hypertext analysis. While the advantage of the semantic markup approach is clear, the requirements on Web document authors make it more difficult to accomplish on a worldwide scale. Nonetheless, a large number of proposals for data and metadata interchange with both human and computer readability have appeared. Most notably, XML, an extensible markup language that is easy to create and parse, has been specified, and many vendors have produced authoring tools, parsers and utilities which enable the creation and analysis of XML documents [18].

Eventually, it may become necessary to partition the web space, concurrently produce partial indices, and re-compose them dynamically to provide a single interface to users searching information on the web.

Distributed Transactions

A typical example for a distributed transaction is a transference of money from a bank account A to a remotely located bank account B. In this scenario, the main requirement is that money be withdrawn from account A if and only if the same amount of money is deposited in account B. If both distributed actions do not happen logically as a single unit, either the bank or the customer would lose the money.

Distributed transactions are often accomplished using a two-phase commit protocol. In the first phase, a transaction coordinator gathers the individual "willingness to commit" from the distributed parties. In the second phase, either everybody is willing to commit or somebody wants to abort the transaction. In the former case, the coordinator sends everybody a "commit" message.

In the latter case, the coordinator sends everybody an "abort" message. Temporary failures in the middle of the process are handled by persistent storage of the state at appropriate times, and the ability to "rollback" to a consistent global state.

Peer-to-Peer Computing

Truly decentralized communication infrastructures, by definition do not have a central point of contact, like the Internet or the World Wide Web. The clear advantage of these infrastructures is that they tolerate partial failures anywhere in the network without disturbing the functioning of the network as a whole.

Many services over the Internet have, nonetheless, a server-centric architecture: clients access a server to get information, effectively using the server as a central point of contact. If the server fails, the service becomes unusable.

Napster[7], for example, is a service that centralizes the querying and searching for information – files often containing music in MP3 format. However, once a search result is given to a client, the client goes directly to another client to fetch the file. Clients effectively become servers in this model.

Gnutella[8] and Freenet[9] are other examples of services on the Internet for distributed file sharing. However, unlike Napster, they provide decentralized querying and searching. There is no centralized point of contact, therefore providing a truly decentralized peer-to-peer infrastructure for file sharing.

Contributions in this Domain

Coordinated computing applications, as opposed to massively parallel applications, have a low ratio between computation and communication. Communicating peers can be co-located, or they may be participating over a WAN. This discrepancy in communication latencies and bandwidths requires careful strategies for ensuring scalability. Chapter 8 presents a model for scalable coordination using hierarchies.

To support coordinated computing, the thesis provides a language mechanism that enables

[7] http://www.napster.com/
[8] http://gnutella.wego.com/
[9] http://freenet.sourceforge.net/

arbitrary communication topologies based on message passing and token-passing continuations. For example, Section 5.2.1 illustrates how a `SALSA` program could start a transaction between two, potentially remote, bank accounts. We also introduce the use of knowledge-based protocols to provide certain guarantees to an application. For example, Section 7.1.3 illustrates a group-knowledge multicast protocol that can be used as a starting point for implementing distributed transactions.

Finally, to target truly decentralized control and coordination, Section 6.3 proposes the World Wide Computer Protocol – a protocol for using the `WWC` as a peer-to-peer infrastructure for distributed computing.

2.4 Internet Software Agents

The final category of applications for worldwide computing is inspired by electronic commerce and by personalizing the Web experience. In this broad category of applications, software agents play a very important role. Business to business (B2B) applications as well as business to customer (B2C) applications can be developed using software agents roaming on the Internet.

We describe four example applications in this category: a travel agent, an auction agent, a personalized information assistant, and an electronic news synthesizer.

Travel Agent

Consider an Internet software agent that makes travel reservations on behalf of a customer. In the simplest case, reservation requests specify a starting point, a destination, and departure and return dates. A certain amount of "money" is allocated for searching for good rates as well as for making the actual purchases. To perform its search, the *travel* agent creates additional agents to search for best airline ticket prices, hotel accommodations and car rental possibilities. These specialized agents themselves create other agents to perform additional searches in parallel. Airline ticketing agents look for airline fares, car rental agents look for rental deals, hotel reservation agents search for hotel rates, and so on. All agents are bound by the shared goals and available resources.

Travel plans can be specified in the form of constraints. Constraints lay out specific requirements, but allow significant flexibility beyond those requirements. For example, a client interested in traveling from Paris to Champaign, specifies desired departure and arrival dates for all or parts of the journey, preferences for means of travel, financial constraints, and so forth. These constraints can be absolute, relative, or a combination of the two.

Although different agents search independently, the constraints that guide them need not be static, which requires the agents to coordinate dynamically. For example, if hotel, car rental, and airline reservations need to be synchronized, they would need to be committed together. Not only does this require enabling synchronization protocols between agents and service providers, the agents must also coordinate their actions. For example, if the flight is to arrive in Chicago from Paris later than when the last flight leaves Chicago for Champaign, alternate plans would need to be considered: a train, bus, or a rental car may be used, or alternatively, a hotel room may be reserved for the night and further travel would be postponed to the following day. Alternative flights from Paris to Chicago should also be considered as an option that results in earlier arrival in Chicago. The entire activity may be in interaction with the customer.

From the customer's perspective, it is important to ensure that certain properties hold. For example, it is imperative to guarantee that the customer's credit card will only be charged if a ticket is obtained, and that it will only be charged once. More complex constraints or rules would enable the customer to establish the probabilities of modifying the original travel plan, so as to minimize the total traveling cost, considering potential penalties incurred in changing departure or arrival dates.

Auction Agent

Auctions are a very common means of off-line negotiation in everyday commerce. Online auctions are quickly catching up as their electronic counterparts. In such auctions, when someone wants to sell an item, they place the item in an auction server by specifying the item's characteristics. The seller also specifies the secret minimum price they're willing to sell the item for, and the maximum time they will wait before concluding the sale. Buyers place bids from a minimum setting – not necessarily the same as the minimum selling price. When the auction is over, the person offering the higher price can buy the item – if that price is larger than the minimum accepted by the seller.

Because auctions occur in real time, sometimes an item can be bought up to the last minute, which can be an inconvenient time for the buyer – for example, if she/he is located on the other side of the world. In this scenario, an auction agent could automatically place bids for a prospective buyer – up to a secret maximum price – up to the last minute, to guarantee that the auction agent's owner can, if feasible, make the purchase. Obviously, the auction agent should not bid more than necessary to guarantee getting the item.

Personalized Information Agent

As more and more information becomes available on the Web, it becomes imperative to provide personalized services which continuously report on specific topics to users online. As opposed to *passive* indexing and searching, it is necessary to be able to continuously monitor specific documents and live forums for information of relevance to a Web user. Eventually, distributed, mobile, personalized information agents, will roam in the network on behalf of a user, with the ability to determine what information may be of interest to the user. Such Internet software agents will need to capture, record and synthesize relevant data.

Personalized active search engines can perform real-time customized searches, where parameters such as "relevance" can be deduced from previous Web traversals by a given user, returning more personally appropriate documents. Similarly, online forums, electronic mailing lists and other nonpersistent information such as newsgroups postings, can be monitored 24 hours a day by a personalized information agent to notify its owner of critical information as soon as possible.

For example, in the case of stock fluctuations which drastically affect the portfolio of a given user, a financial information agent should immediately notify the owner whether by an online program, an electronic mail message or even a cellular phone call. Furthermore, an intelligent software agent may be authorized by its investor owner to buy or sell stock opportunistically.

Electronic News Synthesizer

One final sample application for agents is automatic news synthesis from different sources. While it is currently possible to get information on particular subjects and from particular sources in a customizable fashion (e.g. `http://my.yahoo.com/`), it is not possible to perform a semantic integration to produce a single coherent story from a group of related newspaper articles. It is often the case that newspaper articles include a substantial amount of old and contextual or background information, before saying something really "new". An agent that can collapse multiple related stories into a single story could save considerable amounts of time for a reader.

Similar strategies could be used for analyzing and synthesizing information coming from different mailing lists, newsgroups, or Web documents on a particular subject.

Contributions in this Domain

Internet software agents have a combined need for migration, remote communication, universal naming and coordination, in order to satisfy their information processing goals. Section 5.3 illustrates how `SALSA` provides universal naming for Internet agents, enabling transparent remote communication and migration in the `WWC`.

This thesis does not directly deal with artificial intelligence algorithms for data mining, relevant text extraction, multi-agent cooperation and reconciliation, and so forth. Nonetheless, we believe that the infrastructure we introduce could be used as a lower-level framework for building next-generation intelligent software agents for the Internet.

2.5 Chapter Summary

We have described four application categories which serve as motivating scenarios for our worldwide computing architecture and active object oriented programming language. The four categories are: massively parallel divide-and-conquer problems, collaborative applications, coordinated computations and Internet agents for electronic commerce and personalized Web services. The application categories introduced, illustrate specific needs for universal naming, migration, remote communication and coordination in web-based computing.

Chapter 3

Universal Naming in Worldwide Computing

Worldwide computing systems require a naming infrastructure that provides scalability, transparent object location and migration, heterogeneity of connected devices, efficient resolution, safety and human readability – as explained in Section 1.2.2.

We present our proposed naming scheme for the World-Wide Computer. A *universal actor* is referred to by a location-independent name, which is dynamically mapped to the actor's current locator – a uniform, unique handle containing location and protocol information for communication with the given actor. We define universal actor names and locators. Moreover, we propose architectural and programming language abstractions to facilitate naming. First, a naming service and protocol to implement the mapping from names to locators. And second, high-level abstractions enabling the association of actor references in our programming language to programming language independent names and locators. We conclude the chapter by relating our work to other naming mechanisms.

3.1 Naming Scheme

A major motivation behind our naming scheme is the scalability afforded by the World-Wide Web's addressing approach [14]. Much of the Web's growth and scalability is due to Berners-Lee's strategy for uniformly identifying multiple resources using Uniform Resource Identifiers (URI) [15]. URIs include location-independent Uniform Resource Names (URN), Uniform Resource Locators (URL), and Uniform Resource Citations (URC) containing metadata. Unfortunately, only URLs are currently widely deployed, hindering the transparent mobility of Web resources.

From a practical perspective, an important characteristic of the Web's addressing scheme is the ability to "write the URL of a Web page in a business card." Human readability not only enables, but also encourages the unanticipated connection of network resources worldwide. Moreover, people have the possibility to "guess" names.

In traditional actor semantics, actor names cannot be guessed. This provides certain desirable properties: i.e., an actor can only communicate with actors for which it knows the addresses – such actors are called *acquaintances*. In open systems, an advantage of this approach is the encapsulation it affords – only *exported* actor names can be used from the outside of a given actor configuration. The actors with names that have been communicated to the outside of a configuration are called *receptionists*. Conversely, the outside acquaintances of a configuration, are called *external actors*.

The actor acquaintance laws have implications in safety and garbage collection. Regarding safety, it is possible to enforce at the language level that an actor name is valid. – i.e., that it corresponds to a valid address or location. Regarding garbage collection, it is possible to recollect the memory for a given actor, when the actor is idle and there are no live actors with references to such actor – these actors are called *inverse acquaintances*.

In our extension to the Actor model, we have two types of actors: (1) *anonymous actors*, which are actors with no universal name, and (2) *universal actors* – which are actors with human-readable, guess-able, universal names. Anonymous actors are very similar to object references in object-oriented programming languages – they can only be accessed within the scope of their definition, or by passing a reference to them as an actual argument in a message. On the other hand, universal actors can be accessed beyond the scope of their definition – any actor can create a valid reference to a universal actor from its universal name.

Even though guessing names violates the treasured actor acquaintance laws, an actor starts being anonymous and it must be explicitly made universal, and thus it is like sending its name to another actor (the naming service) who in turn is known by many other actors (and humans).

Following the Web's convention for addressing documents, we use Uniform Resource Identifiers (URIs) for both universal actor names and locators. *Universal Actor Names* (UAN) are globally unique identifiers, which persist over the life-time of an actor and provide an authoritative answer

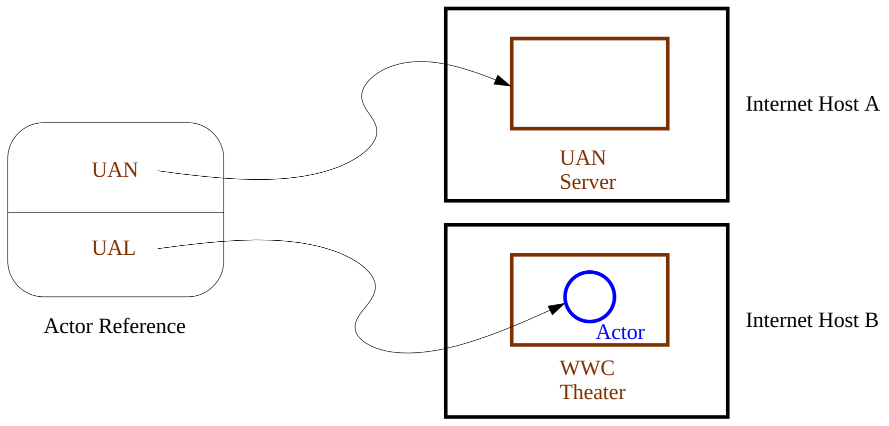


Figure 3.1: By providing a persistent name to an actor (UAN), the actor can migrate from one *theater* to another without breaking existing references.

regarding the actor's current location (see Figure 3.1). *Universal Actor Locators* (UAL) uniformly represent the current Internet location of an actor – which we call *theater*, as well as the protocol to use for communication with the actor.

When an actor migrates from one theater to another, its UAN remains the same, but its UAL is updated in the corresponding naming server to reflect the new location. Notice that migration is transparent to client actors, which always hold a valid UAN reference.

A sample UAN for an actor handling Professor Agha's calendar is:

```
uan://osl.cs.uiuc.edu/~agha/calendar/
```

A sample UAL for the actor is:

```
rmsp://agha.cs.uiuc.edu/myCalendar
```

Our naming strategy enables transparent actor migration, and allows unanticipated creation of universal actor references from persistent, unique, worldwide actor names.

3.1.1 Universal Actor Names

Universal Actor Names (UAN) are URI-compliant unique identifiers, which allow client programs to locate a given actor at any time. Although an actor may migrate multiple times in its life span, its name remains always the same. A sample UAN follows:

```
uan://yangtze.cs.uiuc.edu:3030/~cvarela/emailManager
```

The authoritative answer on the current actor location is given by a UAN server listening at the Internet host `yangtze.cs.uiuc.edu` on port number 3030 – which is the default port for UAN dæmons and can be omitted. The UANP protocol, discussed in Section 3.2.1, provides simple methods for creating entries in the name server, getting the current locator for an actor given its relative UAN, updating an actor's current locator, and removing an entry from the name server.

3.1.2 Universal Actor Locators

Universal Actor Locators (UAL) are URI-compliant unique identifiers, which describe the current location of an actor and allow client programs to send messages to the actor at any time. A sample UAL for the actor above is:

```
rmsp://oceanide.lip6.fr:4040/~cvarela/email
```

In order to send messages to the actor, the run-time system connects to a Remote Message Sending Protocol (RMSP) server listening at the Internet host `oceanide.lip6.fr` on port number 4040 – the default port for RMSP requests, which can be omitted. The RMSP server keeps a hashtable mapping relative UALs – in the example, `/~cvarela/email` – to valid actor references inside the theater. Once a message has been deserialized, as described in Section 6.2.2, it is placed in the corresponding actor's mailbox for local processing.

Notice that universal actors supporting different communication protocols can be specified by changing the protocol portion of the UAL. The scheme not only guarantees openness, but also extensibility. For example, an "http://" prefix for a UAL may mean that the actor is accessible through an HTTP server properly configured – for instance, with a CGI script decoding the URL string and sending the actor a message in its own protocol. Another example is a "mailto://" prefix, which may imply that the actor can receive and interpret electronic mail messages for communication.

The Web started being backward compatible with pre-existing Internet protocols, using URLs starting by "ftp://" or "gopher://". In a similar way, the `WWC` may enable reuse of existing distributed objects by using UALs which start by "iiop://" for CORBA objects or by "rmi://" for Java RMI objects.

We now proceed to describe the run-time architectural system designed and implemented in the World-Wide Computer for naming.

3.2 Naming in the WWC

The WWC enables universal actor names and locators in two ways. First, by providing a Universal Actor Naming service, which maintains the mapping from UANs to UALs. And second, by implementing high-level programming constructs for naming to be used in SALSA programs. We will describe the naming service and the SALSA programming language abstractions for naming in the following two sections.

3.2.1 Universal Actor Naming Service

The universal actor naming service is in charge of the mapping from UANs to UALs. It is implemented as a set of servers which understand the Universal Actor Naming Protocol (UANP).

UANP provides simple methods for creating and removing entries in a name server, as well as for getting and updating the current locator of an actor given its relative UAN. The following table shows the different UANP methods:

Method	Parameters	Action
PUT	relative UAN, UAL	Creates a new entry in the database
GET	relative UAN	Returns the UAL entry in the database
DELETE	relative UAN	Deletes the entry in the database
UPDATE	relative UAN, UAL	Updates the UAL entry in the database

The UANP protocol is text-based and a sample session is given in Figure 3.2. Future versions of UANP could include header information – like HTTP – to authenticate requests for changes and updates to the naming server database.

We now proceed to describe the second part of our naming system – namely, high-level programming language abstractions for binding universal actors to names and locators, and for creating references to actors from UANs and UALs.

```
--> telnet yangtze.cs.uiuc.edu 3030
Trying 128.174.241.200...
Connected to yangtze.cs.uiuc.edu (128.174.241.200).
Escape character is '^]'.
PUT /~cvarela/emailManager rmsp://oceanide.lip6.fr:4040/~cvarela/email UANP/0.1
UANP/0.1 201 Database Modified

Connection closed by foreign host.

--> telnet yangtze.cs.uiuc.edu 3030
Trying 128.174.241.200...
Connected to yangtze.cs.uiuc.edu (128.174.241.200).
Escape character is '^]'.
GET /~cvarela/emailManager UANP/0.1
UANP/0.1 200 Location Found
rmsp://oceanide.lip6.fr:4040/~cvarela/email

Connection closed by foreign host.
```

Figure 3.2: Transcript of two Universal Actor Naming Protocol (UANP) sessions

3.2.2 UANs and UALs in `SALSA`

Actors in SALSA programs are anonymous by default. In other words, they are reachable only from within the program they were created. In order to make an actor universal – and therefore, accessible to other `WWC` actors – it is necessary to bind it to a valid pair containing a Universal Actor Name and an initial Universal Actor Locator. The binding process, in effect, updates the naming service to reflect the new mapping and migrates the actor to the theater represented by the initial UAL.

`SALSA` provides a primitive operation for binding actors to <UAN,UAL> pairs. The `SALSA` pseudo-code for a sample calendar management program (see Section 2.2) is shown in Figure 3.3. The program creates a calendar manager and binds it to the UAN `uan://osl.cs.uiuc.edu/ agha/calendar`. The Universal Actor Naming Server at `osl.cs.uiuc.edu` gets updated with the new <relative UAN, UAL> pair: <`/ agha/calendar/, rmsp://agha.cs.uiuc.edu/myCalendar`>. Moreover, the program moves the actor to the theater specified by the UAL: the theater running at host `agha.cs.uiuc.edu`. "rmsp" stands for the *Remote Message Sending Protocol*, the default remote communication protocol for universal actors. After the program successfully terminates, the actor can be remotely accessed either by its name or by its locator.

```
behavior CalendarManager {

      MeetingTime availableMeetingTime(){...}

      void act(String[] args){
            CalendarManager calendar = new CalendarManager();
            try {
                  calendar<-bind("uan://osl.cs.uiuc.edu/~agha/calendar",
                          "rmsp://agha.cs.uiuc.edu/myCalendar");
            } catch (Exception e){
                  standardOutput<-println(e);
            }
      }
}
```

Figure 3.3: Universal Actor Name and Locator binding in SALSA

SALSA provides some specific exception types, which enable partial recovery from potential failures in the binding process. In particular, failures occurring from unreachable Internet nodes or timeouts in the communication. Additionally, exceptions are thrown for trying to use an already existing UAN or UAL. Exceptions must be handled asynchronously in an application-specific way.

3.3 Related Work

ActorSpaces [22] is a communication model that compromises the efficiency of point-to-point communication in favor of an abstract pattern-based description of groups of message recipients. ActorSpaces are computationally passive containers of actors. Messages may be sent to one or all members of a group defined by a destination pattern. The model decouples actors in space and time, and introduces three new concepts:

- **patterns** – which allow the specification of groups of message receivers according to their attributes
- **actorspaces** –which provide a scoping mechanism for pattern matching
- **capabilities** – which give control over certain operations of an actor or actorspace

ActorSpaces provide the opportunity for actors to communicate with other actors by using their attributes. The model provides the equivalent of a Yellow Pages service, where actors may publish (in ActorSpace terminology, "make visible") their attributes to become accessible. Berners-Lee, in his original conception of Uniform Resource Citations, intended to use this type of metadata to facilitate semi-automated access to resources. Actorspaces bridge the gap between actors searching for a particular service and actors providing it.

Smart Names or *Active Names* (WebOS) provide scalability of read-only resources in the World-Wide Web by enabling application-dependent name resolution in Web clients [83]. Using smart names, a client may find the most highly available resource, which depending on the application may be the closest in distance or the one that provides the best quality of service. While in the present implementation of naming in the `WWC`, the name-to-location resolution algorithm is fixed; an extension to UANP, similar to extensions to HTTP for context-dependent document delivery, could be used for smarter resolution algorithms. When actors represent read-only resources, it is conceivable to scale a service by replicating actors as much as possible and by using the naming server to resolve a name to the closest replica. However, for read-write, mobile resources; depending on the frequency of write accesses, it may become very expensive to keep replicas consistent – given the latencies in wide area networks. Therefore, if a strongly consistent memory model is desired, it is preferable to have a one-to-one mapping from names to locators, such as is the current case in the `WWC`.

The 2K distributed operating system [56] builds upon and enhances CORBA's naming system [46]. CORBA objects have Interoperable Object References (IORs), which 2K objects can find through locally available *clerks*. *Junctions* enable the use of other resource naming spaces, such as DNS or a local file system. A special junction could be used to refer to `WWC` actors from 2K objects.

Section 9.1 discusses some limitations of our universal naming scheme along with some potential solutions.

3.4 Chapter Summary

Universal naming is the property of distributed systems to uniformly name its entities in a way that worldwide access becomes possible. We have introduced a naming strategy based on Uniform Resource Identifiers which separates the name of a universal actor from its location, enabling both location and migration transparency. Universal Actor Names persist over the life time of a universal actor, while Universal Actor Locators enable remote communication with the actor by specifying its location and the protocol to use for communication. The default communication protocol for universal actors in the `WWC` is the Remote Message Sending Protocol. The `SALSA` compiler and run-time system enable universal actor reference creation from UANs and UALs. `SALSA`'s generated code uses the Universal Actor Naming Protocol to update the WWC's naming service.

Chapter 4

Remote Communication and Migration

Worldwide computing applications need to provide access to remote resources, code migration for dynamic application behavior, and location independence for application users – as explained in Section 1.2.3.

This chapter firstly introduces our model for remote communication and migration. Secondly, it describes the architectural and programming language abstractions we propose in the World-Wide Computer to partially implement our model. Finally, it points to related work on remote communication and migration models and infrastructures.

4.1 Remote Communication and Migration Model

The goal of our model for remote communication and migration is two-fold. First, we want to provide transparency for two communicating actors, i.e., sending a message to a remote actor should look like sending a message to a local actor, and likewise receiving messages should be transparent to the location of the sender. Second, transparency needs not be compromised by mobility – i.e., if the source or target actors of a message migrate, the message sending and receiving semantics should not be affected.

The advantage of achieving these goals is that it makes possible to reconfigure an application at run-time arbitrarily. The actors involved in performing a computation could be dynamically migrated to new locations without affecting the overall application semantics.

However, achieving these goals is not as simple for a reason: there are resources which are not necessarily or easily mobile – e.g., a network connection, a screen output, a big database, or a robot camera in another planet. Therefore, there are resources which are naturally *immobile* – we call them *environment actors*. We assume that when an actor moves, it will dynamically get references to local resources in the new environment. Sekiguchi and Yonezawa model environment actors as references of an *ubiquitous* type [71].

Furthermore, dealing with failures cannot be done completely transparently. An application in which all actors are local may not need to worry about reachability or partial failures – as long as the application hosting environment does not completely fail, it is reasonable to expect guaranteed message delivery. However, worldwide computing applications, because of their distributed nature, need to provide application-dependent exception handling in the case of network partitioning or remote host failures.

In our model, universal actors are hosted by *theaters*. Theaters provide the run-time system for universal actors – enabling remote communication and migration – and access to the local environment.

Remote communication is performed through asynchronous message sending between actors. Messages between actors in remote theaters may contain references to other universal actors and data as Java serializable objects.

Migration in our model consists of three elements: universal actor migration, cast migration and messenger migration.

4.1.1 Universal Actors

Universal actors are free to move in the World-Wide Computer by migrating from theater to theater. A universal actor migrates in response to a message requesting migration. All the actor acquaintances remain the same, except for environment actors – references to environment actors are given to universal actors upon entrance to a theater.

Migration involves three sub-processes. First, contacting the new theater and serializing the actor's state from the original to the new theater. Second, updating the naming service to reflect the new universal actor location. And third, leaving a temporary forwarder actor in the original theater to re-route messages to the new theater.

If the migration or remote communication process fails, application actors receive an exception – as an asynchronous message from the run-time system. Applications need to provide specific exception handling mechanisms for remote communication or migration errors.

4.1.2 Casts

Casts are meant to be units for coordination (see Chapter 8), therefore it makes sense to migrate all actors in a given cast at once. While it may not always be possible to migrate a whole cast, the more closely co-located correlated actors are, the better the expected performance of an application.

Cast definition and migration are not currently directly supported by `SALSA` and the `WWC`. However, an available, not necessarily optimal way to implement cast migration is to multicast a `migrate(UAL)` message to all members of a cast.

4.1.3 Messengers

A *messenger* is a special actor which migrates with the purpose of carrying a message from a local theater to a remote theater. A messenger may also possibly have other behaviors. For example, messengers may:

- provide more robust message delivery by persisting over temporary failures in the target actor
- attempt message return upon failure in message delivery
- follow mobile actors
- acknowledge message receipt or processing
- handle common exceptions at the target actor's theater
- be in charge of networking and naming issues.

4.2 Remote Communication and Migration in the `WWC`

We describe *theaters* – virtual machines for hosting universal actors. Then, we introduce the *Remote Message Sending Protocol* (RMSP) for remote communication and actor and messenger migration.

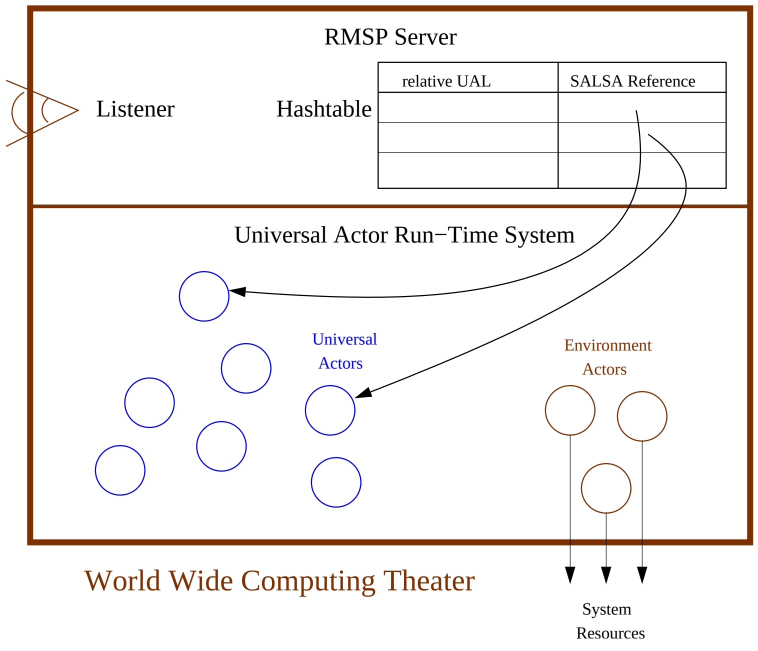


Figure 4.1: A theater contains: (1) an RMSP server with a hashtable mapping relative UALs to actual **SALSA** actor references, and (2) a run-time system for universal and environment actors.

Finally, we illustrate the high-level programming language operations provided by **SALSA** to enable both remote communication and migration of universal actors.

4.2.1 World-Wide Computer Theaters

A theater enables the execution of universal actors (see Figure 4.1.) A theater contains: (1) an RMSP server with a hashtable mapping relative UALs to actual **SALSA** actor references, and (2) a run-time system for universal and environment actors.

Universal actors in theaters never terminate, since references to those actors can be created without anticipation from their names (UAN) or even directly from their current locator (UAL).

When an actor is initially bound to or migrates into a theater, (a) its actor references get updated – previously remote actor references may become local, and previously local references may become remote –, (b) the theater's RMSP server's hashtable receives a new entry with the incoming actor's local reference, and (c) the actor gets started. A theater provides access to its immobile resources – for example, standard output, input, and error streams – through environment actors. All incoming actors get references to environment actors upon arrival.

When a theater receives an incoming communication through the RMSP server, it finds out the target for the message in its internal hashtable, and after properly translating the references in the message to the local environment, the run-time system proceeds to put the message in the target actor's mailbox. From the perspective of the target actor, the message is received in the same way as a local communication.

Theaters may run on applets. In this case, the RMSP server in charge of communication with actors in the applet theater must reside in the same server as the HTTP server which hosts the applet. This is due to security restrictions prohibiting arbitrary Internet communications by untrusted Java applets.

4.2.2 Remote Communication

Remote communication in the World-Wide Computer is accomplished using the Remote Message Sending Protocol. RMSP defines how an actor sends a message to any other actor in the `WWC`. RMSP allows an actor to send to another actor (specified by a UAN or a UAL): (1) a `SALSA` message, (2) a messenger, or (3) a cast of actors.

Universal actor references contain three parts: first, an internal actor reference pointing directly to the actor if it is co-located; second, the last known UAL; and third, the actor's UAN. A `local` bit is maintained to flag local actors.

Communication between universal actors is thus three-layered for efficiency: first, if the actor is local, an internal communication is used; second, if the UAL is available, it is used to try sending the message directly to the remote actor's theater; finally, if the previous two methods fail, the UAN is used to get the current UAL for the universal actor and communication is performed using that locator.

Communication with a remote actor, given its UAL, is performed by connecting to the RMSP server specified in the UAL. Once connection is established, the `SALSA` run-time system – acting as RMSP client – sends the message using a specialized version of Java object serialization. A message is a Java object containing the source and target actor references, the method to invoke at the target actor, the argument values, and an optional token-passing continuation[1] – represented

[1]explained in detail in Chapter 5

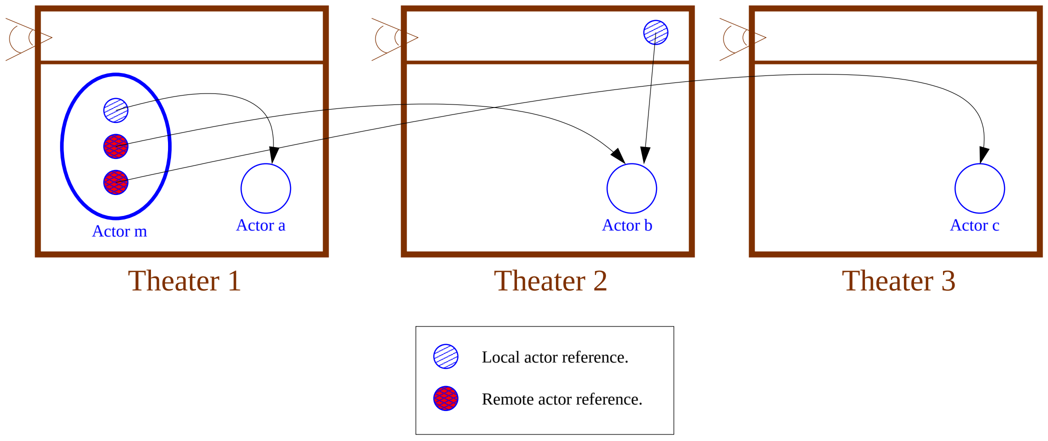


Figure 4.2: Before migration of an actor `m` from Theater 1 to Theater 2, its references to actors `b` and `c` are remote, while its reference to actor `a` is local.

as another message object.

When an incoming message is received at a theater, all its actor references are updated in the same manner as in actor migration (see Figures 4.2 and 4.3). If the UAL for the actor reference points to the current theater, its internal reference is updated with the actual `SALSA` reference for that actor found in the RMSP server hashtable and its `local` bit gets set. If the UAL for the actor reference points to another theater, the reference remains unchanged (remote actor) and its `local` bit is cleared.

After updating all actor references, the target actor reference in the message object has a valid internal reference pointing to the target actor in the current theater. The message object is subsequently placed in the local actor's mailbox. If the target actor has moved in the mean time, it leaves behind a *forwarder* actor – the same reference with its `local` bit cleared. In this case, the RMSP server at the new location – the *forwarder*'s UAL – is contacted and the message sending process gets started again.

However, *forwarder* actors are not guaranteed to remain in a theater forever. Thus, if the RMSP server hashtable does not contain an entry for the target actor's relative UAL, the UAN service for the target actor needs to be contacted to get the actor's new locator. Once a locator has been received, the message sending process gets started again and a `hops` counter gets incremented in the message object. If the counter reaches a predetermined maximum – by default set at 20 hops – the message is returned to the sender actor as undeliverable.

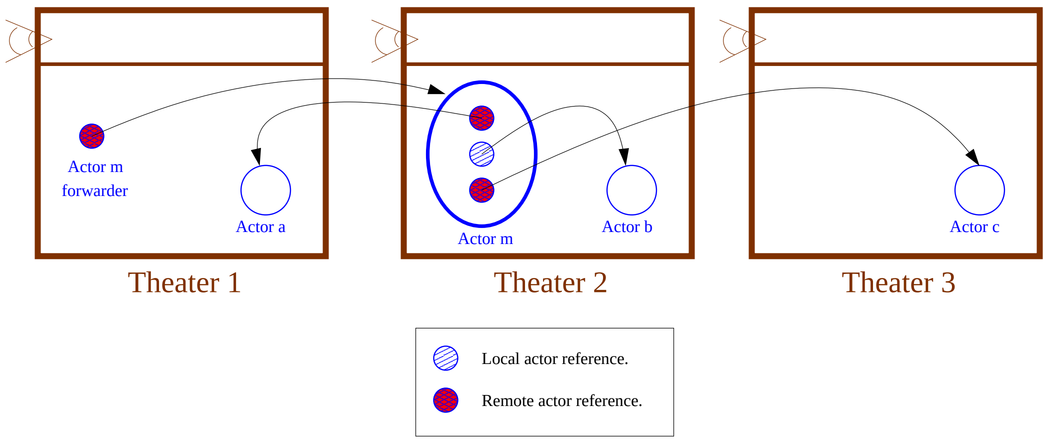


Figure 4.3: After migration of actor `m`, its reference to actor `a` becomes *remote* and its reference to actor `b` becomes *local*. Its reference to actor `c` remains unchanged.

RMSP in `SALSA`

`SALSA` enables sending remote messages to actors using RMSP. The code for sending an `availableMeetingTime()` message to the calendar manager created in Section 3.2.2, is shown in Figure 4.4.[2] The first code segment gets a reference to the calendar manager using its UAN. The second code segment uses the calendar manager's UAL. While the first segment is less efficient than the second segment, it is transparent to migration. The UAL and internal actor reference for `calendar` are recorded for faster future three-layered communication.

In the current implementation, the code for the calendar manager (or any other universal actor) needs to be locally available due to security restrictions. With a security manager in place, it would be possible to download remote code safely for the computer hosting the theater, as it is done, for example, in Java applets. Protecting an actor or a message from malicious routes could be accomplished with encryption. Preventing attacks on actors from malicious theaters could be accomplished with *security directors* filtering out messages according to specific security policies. Notice that filtering messages to an actor can protect its state, because the actor's state can only be modified by sending messages to the actor.

[2]The `@` sign represents a *token-passing continuation*, explained in Section 5.2.1.

```
//
//  Getting a remote actor reference by name and sending a message:
//

CalendarManager calendar = new CalendarManager();
calendar<-getReferenceByName("uan://osl.cs.uiuc.edu/~agha/calendar") @
calendar<-availableMeetingTime() @
standardOutput<-print;

//
//  Getting the reference by location:
//

CalendarManager calendar = new CalendarManager();
calendar<-getReferenceByLocation("rmsp://agha.cs.uiuc.edu/myCalendar") @
calendar<-availableMeetingTime() @
standardOutput<-print;
```

Figure 4.4: Remote Message Sending Protocol (RMSP) in `SALSA`

4.2.3 Migration

Universal actors migrate in response to an asynchronous message requesting migration to a specific UAL. This ensures that the actor is not busy processing any other messages, since when migration takes place, the actor must be processing the `migrate(ual)` message. Following, the algorithm for actor migration is described in more detail, from the perspective of the arrival and departure theaters.

Arrival Theater

The RMSP server described in Section 4.2.2 is also used for incoming actor migration. The server provides a generic input gate for `SALSA`-generated Java objects and the actions associated to receiving an incoming object depend upon the received object's run-time type.

When the incoming object is an instance of the `salsa.language.Message` class, the algorithm for message sending described in the previous section is used. If the object is an instance of the `salsa.language.UniversalActor` class, the internal RMSP dæmon hashtable gets updated with an entry mapping the new actor's relative UAL to its recently created internal Java reference. Then, all actor references in the incoming actor get validated in the same way as in messages. At this point, the actor is restarted locally. Lastly, if the object is an instance of the `salsa.language.Messenger` class, after initialization, the theater's run-time system automatically sends a `deliver()` message

```
//
//  Migrating a calendar manager to a remote theater:
//

CalendarManager calendar = new CalendarManager();
calendar<-getReferenceByName("uan://osl.cs.uiuc.edu/~agha/calendar") @
  a<-migrate("rmsp://agha.palmpilot.com/calendar");
```

Figure 4.5: Actor Migration in SALSA

to the actor. The default implementation of a messenger contains a single message as an instance variable and upon receiving the `deliver()` message, the message instance is delivered with the same algorithm as a passive message.

Departure Theater

When an actor is migrating away from a theater, the actor state is serialized and moved to the new theater. The current actor's internal reference is updated to reflect the new UAL and its `local` bit is set to false. Thus, the internal reference becomes a *forwarder* actor. The forwarder actor ensures that messages en-route will be delivered using the Remote Message Sending Protocol. Lastly, the UAN server containing the migrating actor's universal name gets updated to reflect the new actor location.

Actor Migration in SALSA

SALSA enables migrating an actor to a given theater. For example, the code for migrating the calendar manager above is shown in Figure 4.5. In this case, a reference to the calendar manager is obtained from its UAN (`uan://osl.cs.uiuc.edu/ agha/calendar`) and the agent is migrated to a new location given by a UAL (`rmsp://agha.palmpilot.com/calendar`). If the UAL is already in use by another actor, migration fails – the actor remains in its original place – and a remote exception is created and sent back to the client actor – the actor requesting migration.

4.3 Related Work

The Common Object Request Broker Architecture (CORBA) has been designed with the purpose of handling heterogeneity in object-based distributed systems. In CORBA, object interfaces are defined using a programming language-independent Interface Definition Language (IDL). Different tools automatically create code from IDL definitions that transparently implement the network communication, preserving the object invocation semantics for a given programming language like C++ or Java. Furthermore, CORBA facilitates service location based on IDL descriptions. CORBA provides many common object services including: transaction, security, naming, concurrency control, debugging and network monitoring, connection groups, synchronous and asynchronous operations, events, life cycle, licensing, object trader and queries [65].

Sun's JINI [88] architecture has a goal similar to that of CORBA; the main difference between the two is that the former is Java-centric. However, the Java-centric nature of JINI is due to its use of the Java virtual machine rather than its use of the Java programming language. Any programming language can be supported by a Jini system if it has a compiler that produces Java source code or Java virtual machine compliant bytecodes. The homogeneity of the virtual machine is exploited by Jini in its ability to move objects and code around the network in a safe way.

One of the main components of JINI is the Remote Method Invocation (RMI) [77] specification, which has been included in the Java Developers Kit (JDK) from release 1.1. RMI is a remote method invocation protocol preserving local method invocation semantics – RPC or blocking the sender. Objects and primitive values are marshaled in their Java representation, which is unique across platforms. RMI also allows arguments to contain references to other remotely accessible objects. Some interesting aspects about RMI include: a local registry mapping remotely accessible object names to Java object identifiers. The names are in the form of Uniform Resource Identifiers [15], e.g., `rmi://osl.cs.uiuc.edu:1972/BankAccount`. Another important component of RMI is object serialization; object serialization allows an object's representation to be recreated in a remote machine when the object is passed as an argument in a remote method invocation. There are two types of objects in Java: remote and non-remote objects – remote objects are distinguished because they implement the `java.rmi.Remote` interface. In serialization, remote objects are passed by reference while non-remote objects are passed by value. Distributed garbage collection is provided by RMI with live and weak remote object references: clients send `referenced` and `unreferenced` messages to the servers owning the remote objects.

JavaSpaces [78] is another important component of JINI that uses a Linda-like [26] model to share, coordinate and communicate tasks in Jini-based distributed systems. Its main idea is to decouple providers and requesters of network services by providing shared multiple spaces and an API to access them. Operations on a Java Space include: `read`, `write`, `take`, and `notify`. The last operation allows a specified object to be notified when an entry matching a particular template is written into the space. ActorSpaces is a Linda-like extension to the actor model that provides a way to group and coordinate actors as well as a yellow page service for finding them [22].

Emerald [49] was one of the first systems including fine-grained migration. The Emerald programming language provides different parameter passing styles, namely call by reference, call by move, and call by visit. Instance variables can be declared "*attached*", allowing arbitrary depth traversals in object serialization. Obliq [24] is a lexically-scoped, untyped, interpreted language, with an implementation relying on Modula 3's network objects. Obliq has higher-order functions, and static scope: closures transmitted over the network retain network links to sites that contain their free (global) variables. Java [43] was the first programming language allowing Web-enabled secure remote execution of mobile code; such mobile code is called *applets*. However, once an applet is instantiated, it cannot migrate to a different computer. Moreover, an applet always starts from scratch after being downloaded – i.e., it cannot maintain its own state. IBM Aglets [58] is a more recent framework for the development of Internet agents which can migrate, preserving their state.

Cardelli's Mobile Ambients [25] model object group migration through different administrative domains (e.g. through firewalls). Sekiguchi and Yonezawa [71] describe a flexible calculus for describing movement mechanisms of code, data and execution states. They prove the calculus' expressiveness by emulating Obliq and the Java class loader. Our remote communication and migration protocol use the *copy*, *reference*, and *ubiquitous* mechanisms of code migration.

Communication-passing style (CmPS) [48] refers to a semantics in which data communication is always undertaken by migrating the continuation of the task requiring the data to the processor where the data resides. *Messengers* are similar to CmPS continuations in that they improve locality by eliminating coordination communication across machines. However, messengers are migrating first-class entities, or actors. Therefore, messengers can receive messages, change their state, persist over time, and enable dynamic message delivery policies. On the other hand, messengers require the underlying run-time system to support actor migration.

Fukuda et al. [40] describe a paradigm for building distributed systems, named `Messengers`. The paradigm has similar motivation to our use of the messenger construct. However unlike Fakuda et al.'s work, we distinguish between actors and messengers. Note that in early development of the actor model, Hewitt and his colleagues considered messages to be *unserialized* actors – i.e., messages (unlike serialized actors) could not change their local state [44].

The distributed artificial intelligence research community often uses actors to model intelligent agents. Agha and Jamali [5] are studying issues on agent mobility and resource control. While our model explicitly represents locality and allows for actor migration, we do not take into consideration resource management issues. In application theaters, incoming actors must have locally preexisting behaviors, which is very limiting in that it does not allow arbitrary behaviors to be downloaded and executed. However, because code is trusted, it is possible to use secondary memory and arbitrary network connections. On the other hand, in applet theaters, code is downloaded from the network, so we use the Java sandboxing model approach. Sandboxing is a starting point even though insufficient; in particular, sandboxing does not prevent denial of service attacks. More sophisticated resource control mechanisms are needed not only to control local malicious behavior but also to prevent system-wide chaos.

Section 9.2 further discusses tradeoffs in the design of remote communication and migration models for worldwide computing.

4.4 Chapter Summary

Data and software migration enable scalability, more efficient network usage, improved software user interfaces, and mobile users and resources. We have described a model for remote communication and migration, and our initial reference implementation.

Theaters were introduced as virtual machines for hosting universal actors, a Remote Message Sending Protocol (RMSP) for remote asynchronous message sending, and migration as a basic primitive provided by the `SALSA` programming language and the World-Wide Computer. Both remote communication and migration have the same `SALSA` syntax as local message passing, facilitating distributed systems development.

A three-layered dereferencing algorithm is used for actor communication: a local reference, if existing, is used for most efficient communication; otherwise, the UAL is used to contact the target actor's theater directly; as a last fallback mechanism, the naming server given by the target actor's UAN is contacted to find the actor's current theater. Both remote communication and migration use a specialized version of Java's object serialization, which updates universal actor references in such a way that future communication is more efficient, assuming that actors do not move too frequently.

Finally, we have introduced the notion of *messengers*, which are migrating actors with the specific purpose of sending a remote message with particular customizable behaviors.

Chapter 5

SALSA Programming Language

SALSA stands for Simple Actor Language, System and Applications [67]. SALSA is a dialect of the Java programming language [43] directly enabling actor oriented programming and worldwide computing. SALSA simplifies programming on the Internet by providing universal names, asynchronous communication, active objects, and synchronization using join continuations. We provide some examples which illustrate how SALSA programs are much more concise and easier to follow than comparable Java code. Finally, we discuss the implementation of a preprocessor which translates SALSA code into Java.[1]

We first describe SALSA's primitives for actor programming. Second, we introduce continuations as language abstractions to accomplish actor coordination. Third, we introduce SALSA's abstractions for worldwide computing including universal actors' naming, remote communication and migration. Fourth, we describe essential aspects of SALSA's implementation including changes to Java's grammar, SALSA actor library components and the join continuation code generation algorithm. Finally, we relate our work to other Actor language and library implementations.

5.1 Actor Oriented Programming

The general architecture for compiling and executing SALSA programs is depicted in Figure 5.1. A program in SALSA is preprocessed into Java source code, using a SALSA compiler. The generated Java code uses a library for actors developed especially for SALSA programs. After preprocessing SALSA code, any Java compiler can be used to produce the bytecodes for the program, which can

[1]Available at http://osl.cs.uiuc.edu/salsa/

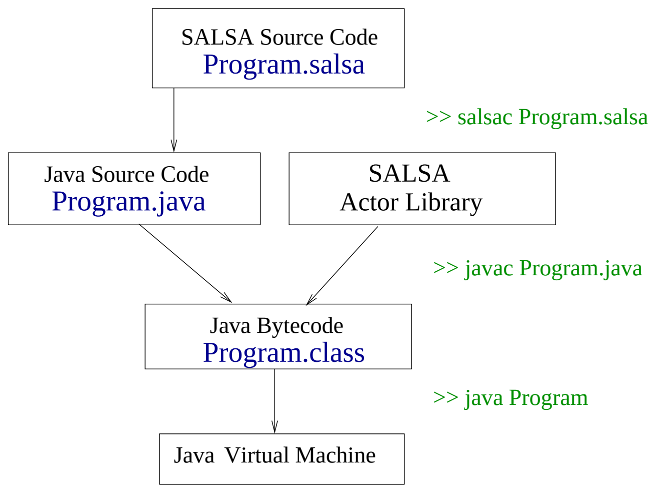


Figure 5.1: `SALSA` programs get preprocessed into Java programs, which use an actor library and get eventually compiled into Java bytecode, for execution atop any Java virtual machine.

```
module helloworld;

behavior HelloWorld {
    void act(String arguments[]){
        standardOutput <- print("Hello ") @
        standardOutput <- println("World!");
    }
}
```

Figure 5.2: "Hello World!" Program in `SALSA`

then be executed on top of any Java virtual machine implementation.

A sample program printing "Hello World!" is shown in Figure 5.2. When a SALSA program is executed, an actor with the given program behavior is created, and an `act` message is sent to the actor by the run-time system, with any given command line arguments. `standardOutput` is a standard actor provided by the environment. An arrow ($\leftarrow$) indicates message sending to an actor. The at-sign (`@`) indicates a token-passing continuation. In this example, it restricts the second message to the `standardOutput` actor to be sent only after the first message has been processed. Appendix B shows `SALSA`'s generated code for this particular example.

`SALSA` programs are grouped into related actor behaviors, which are called `modules`. A `module` can contain several actor `interfaces` and `behaviors`. A behavior may extend another behavior (single inheritance). The root of the inheritance hierarchy is the `UniversalActor` behavior, similarly to Java, where every class extends a top-level `Object` class. A `SALSA` behavior can implement zero or more interfaces (multiple interface inheritance).

We illustrate how the basic primitives of the Actor model are supported by `SALSA`.

5.1.1 Actor Creation

To create a new actor, a `SALSA` behavior is instantiated and a reference to the new actor is returned. For example, a `HelloWorld` actor with the behavior shown above, is created as follows:

```
HelloWorld helloWorld = new HelloWorld();
```

Behavior definitions can contain `constructors` which are used in actor initializations. Constructors can have formal parameters, just like message handlers.

5.1.2 Message Sending

A `SALSA` message is implemented as a potential Java method invocation. Actors contain a mailbox which acts as a buffer for messages. Actors process messages sequentially from their local mailbox.

To send a message to an actor, `SALSA` programs use the actor's reference, an arrow ($\leftarrow$), the message name and actual parameters. For example, to send a `print` message with the String argument `"Hello "` to the `standardOutput` actor, the following code is used:

```
standardOutput <- print("Hello ");
```

If an actor does not specify the target for a message, the message is put in its own mailbox. Sending a message to an actor, returns immediately – i.e., it is asynchronous. The value of a message sending expression is `void`.

5.1.3 Internal State Changes

An actor changes its state by updating its internal variables through assignments. Notice that only an actor can change its internal state since all variables are private. The only way to change another actor's state is by message passing.

```
checkingAccount<-getBalance() @
savingsAccount<-transfer(checkingAccount, token);
```

Figure 5.3: Token-Passing Continuation in SALSA

Shared memory is not allowed in SALSA, as opposed to for example, Java `static` variables which are shared among all the instances of a given class.[2] The complete encapsulation of state and processing, afforded by the actor model, enables relatively straightforward actor migration.

5.2 Coordination through Continuations

In order to coordinate interaction among multiple, autonomous, asynchronous actors, we introduce three kinds of continuations: *token-passing continuations*, *join continuations*, and *first-class continuations*.

5.2.1 Token-Passing Continuations

Since all actor communication is via asynchronous message passing, we need a strategy to "pass" the return values of methods invoked in a given actor – we call these values, `tokens`. To pass tokens, a message to an actor may contain a *customer* actor to which the token should be sent after message processing. SALSA allows the programmer to specify the customer, the message to send to the customer, and the position of the token in the arguments of the customer message. Finally, the continuation itself may have another continuation, thus enabling chains of continuations for tokens.

An example of a token-passing continuation is shown in Figure 5.3. The `checkingAccount` actor receives a `getBalance()` message, and when the `checkingAccount` actor is done processing the message, it sends a `transfer` message to the `savingsAccount` actor with parameters: `checkingAccount`, which is a reference to itself, and `token`, which is the return value of the original `getBalance()` message.

`token` is a special keyword with the scope provided by the last token-passing continuation. For example, in the expression $a_1 \leftarrow m_1(args)@a_2 \leftarrow m_2(token)@a_3 \leftarrow m_3(token);$, the first *token* refers to the result of evaluating $a_1 \leftarrow m_1$ while the second *token* refers to the result of evaluating

[2]The current implementation of SALSA (version 0.3.2) does not strictly enforce not sharing memory.

```
join(author1<-writeChapter(1), author2<-writeChapter(2)) @
editor<-review @ publisher<-print;
```

Figure 5.4: Join Continuation in SALSA

$a_2 \leftarrow m_2$.

`a <- m` is syntactic sugar for `a <- m(token)`. For example, the following two lines of code are equivalent:

```
fractal <- computePixel() @ screen <- draw(token);
fractal <- computePixel() @ screen <- draw;
```

When a continuation message is overloaded, SALSA dynamically chooses the most specific message according to the run-time type of the `token` [32, 27].

5.2.2 Join Continuations

To perform a barrier synchronization, SALSA allows join continuations. In a join continuation expression, a customer actor receives an array with the tokens returned by multiple actors, once they have all finished processing their messages. A sample join continuation is shown in Figure 5.4. The `editor` actor receives a `review` message with an array of `chapter` actors as a parameter, and then it will pass it along to the `publisher` for printing. The `review` message is sent only after all `author` actors finish processing their respective `writeChapter` messages. The join statement can also receive an array of actors which are all to receive the same message.

5.2.3 First-Class Continuations

First-class continuations enable actors to *delegate* computation to a third party, independently of the context in which message processing is taking place – i.e., independently of the current continuation for a given message's token.

Currently, first-class continuations have been designed and included in the language grammar. However, the current version of SALSA does not generate code for first-class continuations.[3]

[3]The SALSA v0.4 release is planned to include first-class continuations and applet theaters.

```
module fibonacci;

behavior Fibonacci {

    int n;

    Fibonacci(int n){
        this.n = n;
    }

    int compute(){
        if (n == 0){
            return 0;
        } else if (n <= 2){
            return 1;
        } else {
            Fibonacci fib1 = new Fibonacci(n-1);
            Fibonacci fib2 = new Fibonacci(n-2);
            join (fib1<-compute(), fib2<-compute())
                 @ add @ currentContinuation;
        }
    }

    int add(int numbers[]){
        return numbers[0]+numbers[1];
    }

    void act(String args[]){
        n = Integer.parseInt(args[0]);
        compute() @ standardOutput<-println;
    }
}
```

Figure 5.5: First-class continuation in `SALSA`

To illustrate the design, let us look at an example using recursion. The infamous `fibonacci` computation in `SALSA` is shown in Figure 5.5. Notice how the first time that the `compute()` message is sent to the `Fibonacci` actor, the `currentContinuation` is to print the value to standard output. Subsequent recursive `compute()` messages will have as `currentContinuation`, the result of combining the join continuation with the addition message and the previous `currentContinuation` taken from the recursive step.

5.3 Worldwide Computing

`SALSA` enables writing programs to be executed on the World-Wide Computer. The World-Wide Computer consists of a set of virtual machines hosting universal actors, called *theaters*. Universal actors are reachable Internet-wide through their unique Universal Actor Names – see Chapter 3.

The top-level `SALSA UniversalActor` behavior provides methods which support the following

```
behavior TravelAgent {

      void printItinerary(){...}

      void act(String[] args){
            TravelAgent a = new TravelAgent();
            try {
                  a<-bind("uan://wwc.travel.com/agent",
                          "rmsp://wwc.aa.com/agent");
            } catch (Exception e){
                  standardError<-println(e);
            }
      }
}
```

Figure 5.6: Universal Naming and Locator Binding in `SALSA`

worldwide computing primitives:

- binding an actor to a name and an initial theater,
- getting references to and communicating with a universal actor, and
- migrating an actor from one theater to another.

SALSA programs can be executed on the WWC infrastructure because all behavior definitions extend the `UniversalActor` behavior.

5.3.1 Universal Naming

The `UniversalActor` behavior provides support for binding actors to UANs and setting them up in a given theater represented by a UAL. The code for a sample travel agent program is shown in Figure 5.6. The program creates a travel agent and binds it to a particular UAN (`uan://wwc.travel.com/agent`) and UAL (`rmsp://wwc.aa.com/agent`) pair. The binding process has two effects: (1) the naming server (`wwc.travel.com`) gets updated with the new <relative UAN, UAL> pair (in this case, <`''/agent''`, `''rmsp://wwc.aa.com/agent''`>) and (2) the actor becomes *universal* by migrating to the Theater (`wwc.aa.com`) specified by the UAL. After this program terminates, the actor becomes remotely accessible either by its name or by its locator.

```
//
//  Getting a remote actor reference by name and sending a message:
//

TravelAgent a = new TravelAgent();
a<-getReferenceByName("uan://wwc.travel.com/agent") @
  a<-printItinerary();

//
//  Getting the reference by location:
//

TravelAgent a = new TravelAgent();
a<-getReferenceByLocation("rmsp://wwc.aa.com/agent") @
  a<-printItinerary();
```

Figure 5.7: Actor Location and Remote Communication in SALSA

5.3.2 Remote Communication

SALSA enables sending messages to remote actors using the *Remote Message Sending Protocol* (RMSP). SALSA-generated code automatically uses RMSP when actors are remote. SALSA programmers use the same syntax for local and remote message sending. The code for sending a `printItinerary()` message to the travel agent created above, is shown in Figure 5.7.

First, a local reference with the remote actor type (or `UniversalActor`) is created. Then, the reference gets "upgraded" from a local actor reference to a universal actor reference by using the `UniversalActor` behavior method `getReferenceByName(UAN)` or `getReferenceByLocation(UAL)`. Upon completion of this method, the local reference becomes an alias to the remote universal actor. Once we have such an alias, a message can be sent to the actor in the same way as in local message sending. Notice that a token-passing continuation is used to ensure that the reference to the remote actor is obtained before the message is sent to the actor.

5.3.3 Migration

SALSA enables migrating an actor to a given theater. For example, the code for migrating the travel agent above is shown in Figure 5.8. Notice that we first use the `getReferenceByName(uan)` method to obtain a reference to the agent and then we use the `migrate(ual)` method to trigger the migration. These two methods are provided by the top-level `UniversalActor` class.

When an actor migrates to a theater: universal actor acquaintances are passed by reference; Java

```
//
//  Migrating a travel agent to a remote theater:
//

TravelAgent a = new TravelAgent();
a<-getReferenceByName("uan://wwc.travel.com/agent") @
  a<-migrate("rmsp://wwc.nwa.com/agent");
```

Figure 5.8: Universal Actor Migration in SALSA

serializable objects and primitive types are passed by value; anonymous actors and non-serializable Java objects generate a run-time exception; and environment actors (like `standardOutput`) are bound after migration to the new theater's environment actors.

In the current implementation, the code for the travel agent (or any other universal actor) needs to be locally available at the receiving theater due to security considerations. A security manager in place, would make it possible to download remote code and safely execute it in a sandbox, such as it is done for example in Java applets.

5.4 An Example: Parallel Search

We present an example based on parallel search which illustrates the high-level abstractions and run-time system provided by SALSA and the WWC. Parallel search algorithms take advantage of available concurrency by dividing a search space into several subspaces which are explored in parallel – see Section 2.1. A network of high-performance Java platforms provides a powerful environment for developing such algorithms.

As an example, suppose there is a binary evaluation function $p(a, b)$ such that $p(a, b) = a$ if a is "more desirable" than b, and $p(a, b) = b$ otherwise. Then, we may find the "most desirable" element in a list L using a simple recursive Java function as shown in Figure 5.9.

The function becomes parallel if the recursive steps in the last "else" clause are performed in parallel. The SALSA code, shown in Figure 5.10, performs the same functionality as the Java example above, except that it randomly assigns different search actors to a group of given machines. The search actors run on the World-Wide Computer infrastructure, which requires *theaters* to be installed and running in the different machines involved. Notice that token-passing continuations,

```
Object mostDesirable(List L, EvalFunction P) {
  if (L.length() == 1)
    return L.at(0);
  else if (L.length() == 2)
    return P.eval(L.at(0), L.at(1));
  else {
    Object B1 = mostDesirable(L.range(0, L.length()/2), P);
    Object B2 = mostDesirable(L.range(L.length()/2 + 1, L.length()), P);
    return P.eval(B1, B2);
  }
}
```

Figure 5.9: Search in Java

join continuations, and high-level abstractions for remote messaging, universal naming and actor migration afford a simple specification of what would otherwise have to be explicitly programmed in Java.

5.5 Language Implementation

We describe essential aspects of the implementation of the `SALSA` programming language.[4] We first discuss the changes made to the Java grammar to accept the presented actors abstractions. Secondly, we present the actor library used by Java code generated from `SALSA` programs. Thirdly, we describe the run-time system for executing `SALSA` programs. Lastly, we provide some insights into the code generation algorithm for join continuations.

5.5.1 Grammar Changes to Java

We have based our actor language on Java's syntax and therefore we have used the grammar for Java as a starting point. However, since we wanted to make it very clear for programmers that the actor model is different in fundamental ways with the object oriented model; we have changed the name for the most general language constructs: `package` to `module` and `class` to `behavior`. In any case, `SALSA` modules get preprocessed into Java packages and `SALSA` behaviors into Java classes which extend a base `UniversalActor` class provided by the `SALSA` Actor Library. Java `interfaces` remain the same in `SALSA`.

To enable asynchronous message passing as the most basic communication primitive, we have

[4]`SALSA` version 0.3.2. contains about 20,000 lines of Java code.

```
module search;

import wwc.naming.UAL;

behavior SearchActor {

  Actor mostDesirable(List L, EvalFunction P) {

    int length = L.length();

    if (length == 1)
      return L.at(0);
    else if (length == 2)
      return P.eval(L.at(0), L.at(1));
    else {
      Actor S1 = new SearchActor();
      Actor S2 = new SearchActor();

      join (
        getNewTheater() @ S1 <- migrate(token),
        getNewTheater() @ S2 <- migrate(token)
      ) @
      join (
        S1 <- mostDesirable(L.range(0, length/2), P),
        S2 <- mostDesirable(L.range((length/2) + 1, length), P)
      ) @
      P <- evalArray @
      currentContinuation;
    }
  }

  String[] machineName = {``yangtze'', ``mekong'', ``cauca'', ``danube''};
  int noMachines = machineName.length;
  int serialActorNo = 0;

  UAL getNewTheater() {
    serialActorNo++;
    return new UAL(``rmsp://''+machineName[Random.nextInt(noMachines)]+
                   ``.cs.uiuc.edu:4040/SearchActorInstance''+
                   serialActorNo);
  }
}
```

Figure 5.10: Parallel Search in SALSA

extended Java's `PrimaryExpression` non-terminal from:

```
PrimaryExpression ::=  PrimaryPrefix ( PrimarySuffix )*
```

to:

```
PrimaryExpression ::=  PrimaryPrefix ( PrimarySuffix )*
                       ( MessageSendSuffix )?


MessageSendSuffix ::=  "<-" <IDENTIFIER> ( Arguments )?
```

Note that if the `Arguments` are not given, we know that it must be syntactic sugar for `(token)` within the context of a token-passing continuation.

Token-passing continuations and join continuations are accepted by `SALSA`'s grammar with the following rules:

```
ContinuationExpression ::=  "currentContinuation"
                        |    PrimaryExpression ( "@" ContinuationExpression )?
                        |    JoinStatement



JoinStatement ::=  "join" "(" ContinuationExpressionList ")"
                   "@" ContinuationExpression


ContinuationExpressionList  ::=  ContinuationExpression
                                 ( "," ContinuationExpression )*
```

Note that it is possible to have token-passing continuation expressions within the list of expressions in a `join` statement.

Finally, two extra possibilities for valid `Statements` were added to the language grammar:

```
Statement ::=  ContinuationExpression ";"
          |    JoinStatement
```

The `token` keyword was also added to the lexical tokens accepted by the parser, and a semantical check at code generation is performed to ensure it is correctly located within an argument list.

We have used the JavaCC parser generator to create the `SALSA` parser. We have also used the JJTree tool which works in conjunction with JavaCC to produce an abstract syntax tree after the parsing phase.[5] Then, we directly manipulate the created abstract syntax tree to generate Java code. Annex A shows the full grammar for `SALSA` 0.3.2.

[5]JavaCC and JJTree are available at http://www.metamata.com/.

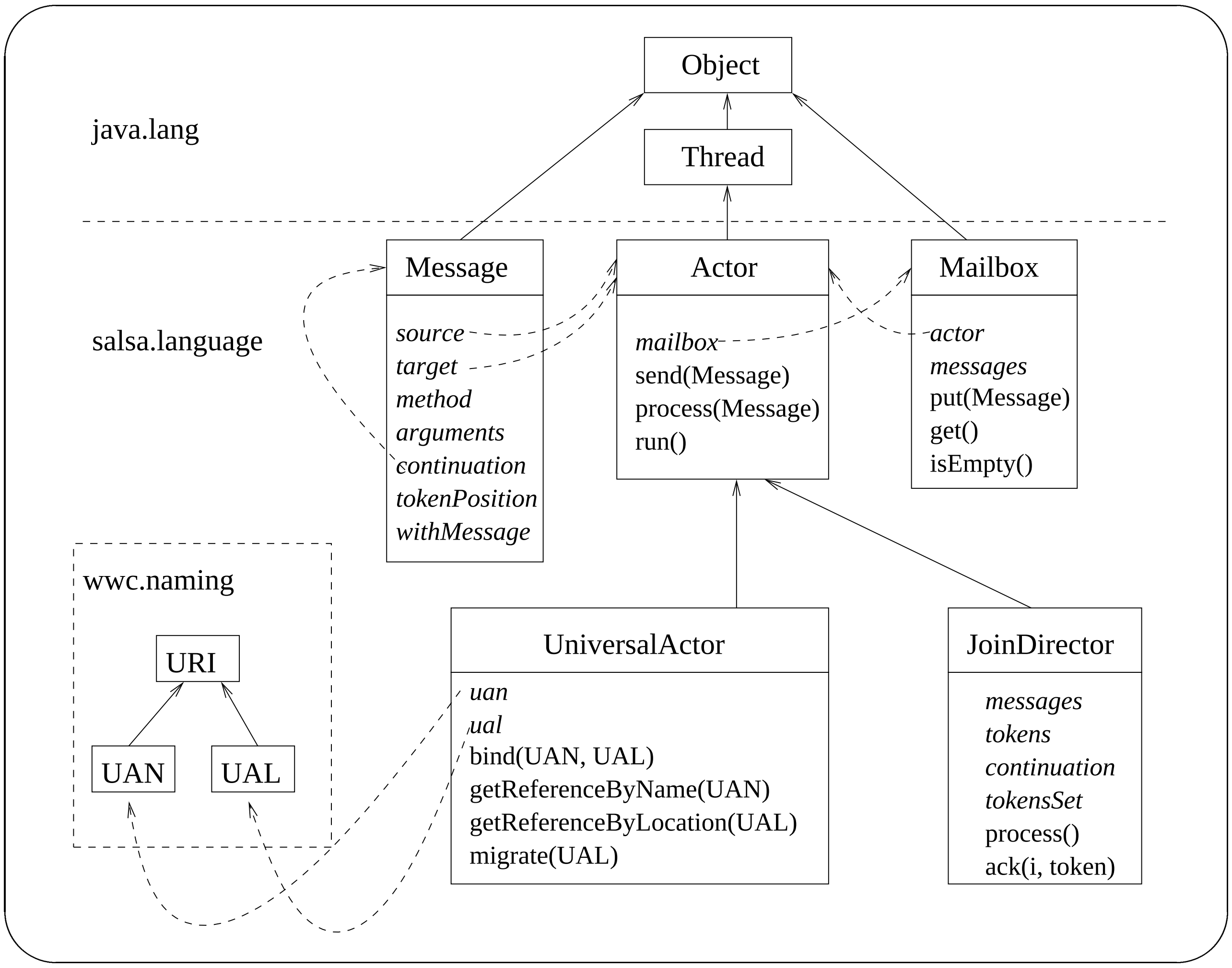


Figure 5.11: Class hierarchy diagram for the `SALSA` Actor library. `SALSA` behaviors get translated into Java classes which extend the `salsa.language.UniversalActor` class.

5.5.2 `SALSA` Actor Library

Figure 5.11 shows the structure of the main classes which compose the `SALSA` actor library. The library contains three packages: `salsa.language`, `wwc.naming` and `wwc.messaging`. [6] In the figure, rectangles represent classes, with the class name in the top center, instance variables below in italics, and method names and argument types in plain style. Solid arrows point to a superclass and curved dotted arrows point to the class of a particular instance variable. Following, we describe the `salsa.language` package.

The `salsa.language.Actor` class extends the `java.lang.Thread` class and therefore encap-

[6]The `wwc` packages were designed and implemented in collaboration with Grégory Haïk at the U. of Paris 6.

sulates a thread of execution directly. An actor contains a `mailbox`, and methods to send it a `Message` object and to process a given message. A main `run` method contains a loop that gets messages from its mailbox and processes them one by one. When a `java.lang.Thread` object is started (i.e. the thread gets actually created), the `run()` method, with the main actor loop gets executed.

The `salsa.language.Mailbox` class contains a `java.util.Vector` of messages, as well as methods to put a message into the mailbox and to get a message from the mailbox (used by the `run` method of the `Actor` class). Access to the mailbox is synchronized using the `lock` associated to the mailbox object.

The `salsa.language.Message` class abstracts over a message being sent from a `source` actor to a `target` actor. It includes a `java.lang.reflect.Method` to be eventually invoked in the target actor with a given array of `arguments`. Additionally, there is an optional token-passing `continuation` which is represented as another `Message` object. If a token is to be passed to such continuation as an argument, a `tokenPosition` value indicates the argument number in which the token should be passed along. Finally, the internal `withMessage` instance variable determines whether or not to pass `this` message as an argument to the original method – a feature is useful for implementing first-class continuations.

The `salsa.language.UniversalActor` class extends the `Actor` class to make an actor *universal* – i.e., reachable via a Universal Actor Name (`uan`) or Locator (`ual`) by other actors in the World-Wide Computer. The class provides methods to bind a universal actor to a given name and initial theater, to get a reference to a remote universal actor either by its name or by its locator, and to migrate a universal actor to a remote theater.

The `salsa.language.JoinDirector` class (named `MessageGroup` in previous `SALSA` releases) contains an array of `Message` objects and an array of tokens to be filled with the returned value of the messages. It also contains a `tokensSet` integer which keeps track of how many actors have already returned their tokens and a `continuation` message to be used to notify a customer actor of completion of the join protocol. Section 5.5.4 describes how the class is used by the `SALSA` code generator to implement a join continuation statement.

5.5.3 Run-time System

The runtime system for `SALSA` programs is in charge of starting program execution by creating an instance of the main behavior and sending an `act(args)` message to such instance with given command-line arguments. The run-time system also needs to check for the conditions that enable program termination:

- All actors are anonymous – i.e., there are no universal actors.
- All actor mailboxes are empty.
- No actor is currently processing a message.

The runtime system accomplishes this by setting up a lower priority thread that checks for these conditions in order to terminate the Java virtual machine.

In the current run-time system implementation, instead of keeping track of all non-universal actors and their mailboxes, every time a message gets sent to an actor, a synchronized global counter gets incremented. Likewise, every time a message has been processed, the counter gets decremented. When a `SALSA` program starts, the counter is set to 1 – corresponding to the `act(args)` message sent to the main actor. When such counter reaches 0, all messages ever sent in the system have been processed, and there are no more messages pending, therefore the `SALSA` run-time system proceeds to terminate the execution of the Java virtual machine.[7]

5.5.4 Join Continuation Code Generation

The general strategy for implementing the join statement in `SALSA` is to create a join director actor which takes care of implementing the join protocol: sending the messages within the join statement to the respective actors, collecting back the tokens, and contacting the customer actor with the compiled array of tokens after completion.

Figure 5.12 displays a generic join protocol, as implemented by the generated `SALSA` code for a particular join statement. In the example, the `src` actor executing the join statement creates a join director (an instance of the `salsa.language.JoinDirector` behavior) and sends a `process()` message to the director.

[7]James Waldby proposed this idea for enabling JVM termination.

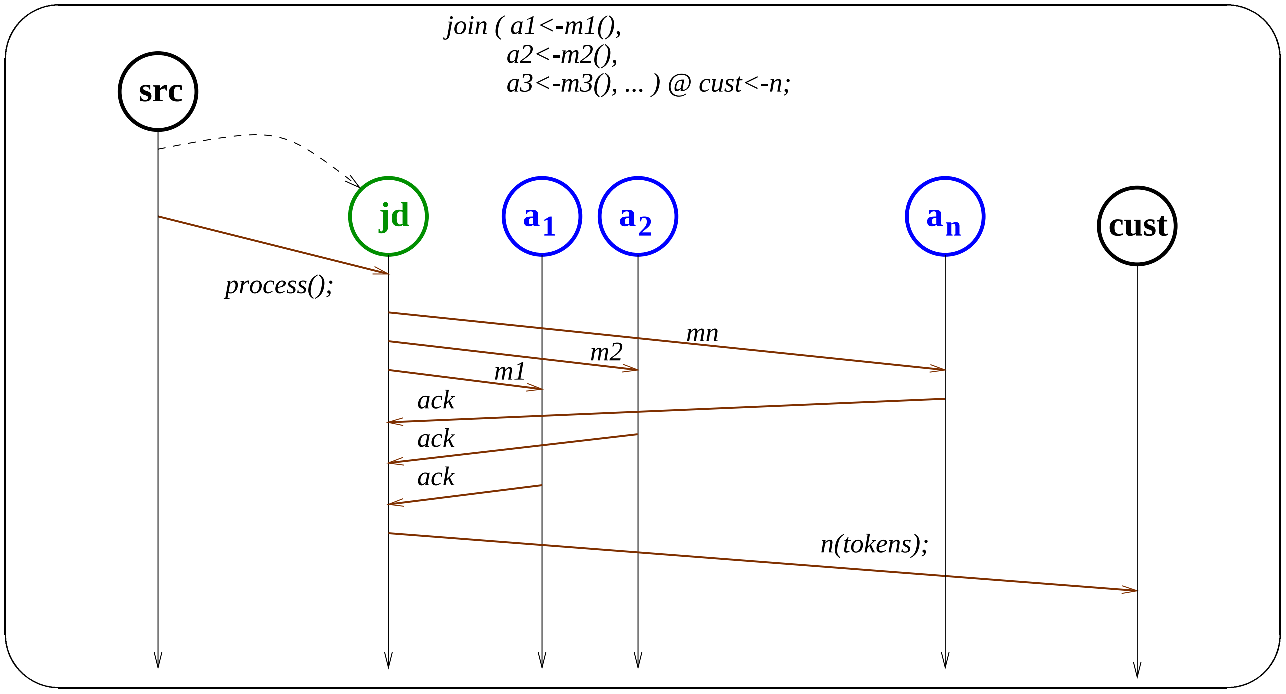


Figure 5.12: A join statement in `SALSA` generates code that creates a director actor in charge of executing the specific join protocol with the given participating actors.

In response to the `process()` message, the director sends the original messages to the actors participating in the join protocol, with the token-passing continuation `ack(i, token)`. Every actor contacts back the director, with two pieces of information: first, the position of the actor in the group of participating actors, and second, the token returned after the computation. When the join director received all the respective tokens, it sends the customer actor the continuation message along with the array of tokens returned as a parameter, if the array is desired by the customer actor.

We have shown only a simple example of a join statement and its corresponding code generation. Note that the code generation algorithm has to deal with the most general case of any potential location for the `join` statement within a `SALSA` expression. Figure 5.13 shows an acknowledged multicast protocol implemented in `SALSA`. A protocol trace is illustrated in Figure 7.1. Appendix C shows the `SALSA`-generated code including the code generated for the `join` statement.

`SALSA`-generated code creates the join director in the same machine as the actor executing the join statement. Therefore, it may be convenient if possible, to co-locate the currently executing

```
module multicast;

behavior AcknowledgedMulticast{
    SimpleActor[] actors;
    long initialTime;
    long ack(){
        return System.currentTimeMillis() - initialTime;
    }
    void act(String[] args){
        int howMany = Integer.parseInt(args[0]);
        SimpleActor[] actors = new SimpleActor[howMany];
        for (int i = 0; i< howMany; i++){
            actors[i] = new SimpleActor();
        }
        initialTime = System.currentTimeMillis();
        standardOutput<-print("Time for acknowledged multicast: ") @
        join(actors<-m())@ack()@
        standardOutput<-print @
        standardOutput<-println(" ms.");
    }
}
```

Figure 5.13: Acknowledged Multicast Protocol in SALSA

actor with the join recipients and the join customer before performing a join statement. The message-passing semantics of the `join` statement is the same independently of the location or migration behavior of the participating actors.

5.6 Related Work

There are several attempts to develop libraries and frameworks in an object-oriented language to implement the Actor model of computation. Three examples are the Actor Foundry [66], Actalk [20] and Broadway [73].

Several actor languages have also been proposed and implemented to date. We will mention a few of the most important ones along with their relationship to our work. These languages are ABCL [91], Concurrent Aggregates [29], Rosette [82], and Thal [53].

5.6.1 Actor Foundry

The Actor Foundry [66] is a framework developed in Java to provide concurrent object oriented programmers with a discipline for actor programming and a set of core services to ease their task.

Operationally, there is an actor manager per Actor Foundry node that serves as an encapsulation boundary and interaction point for actors within that node – i.e., all communication between actors has to go through the Actor Manager. Actor implementations embody the interactions of an actor with its manager. A particular actor implementation may provide for internally multi-threaded actors, for example.

A request handler is in charge of handling the interactions between actor managers in multiple nodes, and provides access (through the actor manager) to a name service. The name service is in charge of providing globally unique names to actors. "Unique" in this context means "unique with respect to all Foundry nodes".

A transport layer provides the low-level communication infrastructure for inter-node actor communication and a scheduler is in charge of managing threads within an actor node and ensuring fairness.

Our approach to actor programming (`SALSA`: Simple Actor Language, System and Applications) is similar to the Foundry in its use of portable bytecode for execution in heterogeneous platforms. However, `SALSA` programs are meant to execute in an applet/servlet context, and therefore cannot afford the operational requirements of a Foundry node.

`SALSA` also differs from the Foundry in that as a programming language it is able to enforce desirable constraints in actor programs (for example, it prohibits sharing memory between actors) at compilation time – unfortunately beyond the power of an actor library.

The World-Wide Computer approach for actor naming does not require a service to know about all the network nodes at a given point in time, which would limit scalability. In contrast, Universal Actor Names are meant to be persistent, globally unique names for actors; independent of the library or language in which they are implemented.

5.6.2 Actalk

Actalk [20] is a framework for implementing actor systems built on top of Smalltalk. The kernel of the architecture is decomposed into a set of classes and associated parameter methods, to be redefined in various subclasses. Various redefinitions emulate different models of activity, communication and synchronization schemes, such as enabled sets and guards. Similarly to the Actor Foundry, as a framework, Actalk cannot enforce desirable semantic properties in Actor programs

– such as distributed memory or proper actor access through the given API.

5.6.3 Broadway

Broadway [73] is a set of classes developed in C++ to enable actor-based programming. Its functionality is provided by a central Daemon as well as a set of system actors described by a library. The dæmon contains a scheduler, a membership monitor for node failure detection, a local and a foreign actor table, a behavior table, a factory of primitive values, and a platform object to isolate platform-dependent inter-node communication. Screed is an actor language that can be compiled into C++ programs with calls to the Broadway Daemon.

Broadway's local and foreign actor tables limit scalability. Java-based approaches can leverage the Java virtual machine's work on heterogeneity. We use Screed's approach of compiling an actor language into an object oriented language – in our case, Java – with the proper usage of a particular actor framework developed in the language.

5.6.4 ABCL

The ABCL family of languages has been developed by Yonezawa's research group [91] to explore an object-oriented concurrent model of computation, based on Actors. ABCL has been developed in Common Lisp. One important difference is that the order of messages from one object to another is preserved in their model. There are also three types of message passing mechanisms: past, now, and future. The *past* type of message passing is non-blocking as in actors. The *now* type is a blocking (RPC) message with the sender waiting for a reply. And the *future* type is a non-blocking message with a reply expected in the future. SALSA allows *past* (asynchronous) and *now* (synchronous) message passing. *Future* message passing can be simulated with token-passing continuations – a more powerful mechanism provided which allows a customer to be specified for notification after processing of a message by an actor. Section 7.2 shows an example with token-passing continuations.

5.6.5 Concurrent Aggregates

Concurrent Aggregates [29] is a dynamically-typed concurrent object-oriented language for programming large-scale parallel machines. It provides a programming model similar to Actors, but augments the model with inheritance and "concurrent aggregates", concurrent data abstractions, that can be used to build modular parallel programs. It was originally designed to program the J-Machine [33], and has been used to explore the programming of irregular applications and the efficient implementations of object-oriented programs.

An aggregate is a group of objects that provides a single interface to other objects; yet allowing for different internal representatives to concurrently process messages sent to the aggregate. Interesting abstractions of the Concurrent Aggregates programming language include first-class continuations and first-class messages. First-class continuations are similar to the reply objects in ABCL. First-class messages are similar to our *messengers*, except the former are immutable.

Aggregates differ from casts (see Chapter 8) in that actors within an aggregate are homogeneous and cannot be seen by outsiders. Instead, casts can be composed of different types of actors. Even though communication is coordinated by a hierarchy of directors, outsiders can send messages to arbitrary actors within a cast and for potentially different purposes.

5.6.6 Rosette

The Rosette actor-based programming language was used in the Carnot project at Microelectronics and Computer Technology Consortium (MCC) for the interpreter of the extensible services switch [82]. Rosette continues to be used to provide heterogeneous interoperability for middleware in intranet and enterprise integration software. Rosette allows inherent concurrency, inheritance and reflection. Synchronization is specified by *enabled sets* which define what methods can be executed under an actor's current state. In contrast, the hierarchical model (see Chapter 8) uses explicit actors – i.e., *directors* – to perform synchronization.

5.6.7 Thal

THAL, an extension to HAL (High-level Actor Language) [45], was developed by Kim [53] to explore compiler optimizations and high performance actor systems. As a high performance implementation, THAL has taken away features from HAL like reflection, and inheritance. THAL provides several communication abstractions including concurrent call/return communication, delegation, broadcast and local synchronization constraints. THAL has proven that with proper compilation techniques, parallel actor programs can run as efficiently as their non-actor counterparts. THAL has been written in C and it does not study coordination, heterogeneity or distributed computing issues – such as universal naming and migration.

5.7 Chapter Summary

We have presented `SALSA`, an actor programming language, dialect of Java, with token-passing continuations, join continuations, and first-class continuations. We described `SALSA`'s additional abstractions for programming World-Wide Computer applications. In particular, its high-level `UniversalActor` behavior methods for universal actor naming, remote communication, and migration. Finally, we discussed essential aspects of the language implementation, including grammar changes to Java, the components of the actor library used by `SALSA`-generated code, and the algorithm for join continuation code generation.

Chapter 6

World-Wide Computer

The goal of the World-Wide Computer project is to study the technologies that enabled the Web to succeed, and to propose proper mechanisms to enable the utilization of the current network infrastructure as a worldwide computing and collaborating platform [84].

We have developed naming and messaging protocols, data formats and programming abstractions for *universal actors*, the main entities performing computations in the `WWC`. The `SALSA` programming language enables the development of universal actor behaviors; the *Remote Message Sending Protocol* (RMSP) enables sending messages and migrating actors (in the form of serializable Java objects); *Universal Actor Names* (UAN) and *Universal Actor Locators* (UAL) enable referencing and contacting remote actors; and *theaters* are the run-time system required by universal actors to execute.

In Section 6.1, we describe and relate our extensions to the underlying World-Wide Web technology to enable worldwide computing. Section 6.2 describes in more detail universal actors, the Remote Message Sending Protocol, Universal Actor Names and Locators, and theaters. In Section 6.3, we describe our design for the World-Wide Computer Protocol, a resource discovery and allocation strategy that can be used by theaters in order to collaboratively perform computations on behalf of `WWC` users.

6.1 World-Wide Computer Architecture

Berners-Lee created the World-Wide Web [14] by defining:

- HyperText Markup Language (HTML): a data format,

Table 6.1: Comparison of WWW and WWC concepts

	World-Wide Web	World-Wide Computer
Entities	Hypertext Documents	Universal Actors
Transport Protocol	HTTP	RMSP/UANP
Language	HTML/MIME Types	Java ByteCode
Resource Naming	URL	UAN/UAL
Linking	Hypertext Anchors	Actor References
Run-time Support	Web Browsers/Servers	Theaters/Naming Servers

- HyperText Transfer Protocol (HTTP): a data transport protocol, and
- Uniform Resource Locators (URL): a resource locating scheme.

Table 6.1 relates the different concepts in the World-Wide Web to the analogous concepts developed for the World-Wide Computer.

6.1.1 Data Format

There are three types of structures being transferred in the World-Wide Computer.

- *Passive messages*, which do not include any special behavior or code to be sent to a recipient, just data.
- *Universal actors*, which may have references to other actors and special behaviors. A particular case is *messengers*.
- *Casts*, which are groups of highly interrelated actors.

For migrating messengers and actors, we leverage a good percentage of the work on Java serialization and dynamic class loading. It is important to notice, however, that actors are self-contained, which improves the performance bottleneck usually associated with Java object serialization. In Java serialization, all references in an object's graph must be traversed and serialized. Because actors are self contained and acquaintance references are expressed in a universal actor naming space, actor serialization can perform relatively better than object serialization.

Migrating data also leverages most of the present Java serialization API and infrastructure. XML [17] is a current attempt by the World-Wide Web Consortium to provide an extensible programming language-independent and application-independent markup language for data. It is a future direction worthwhile exploring for exchange of passive messages between WWC actors trying to collaborate in a particular discipline over an open environment.

6.1.2 Data Transport Protocol

The main entities in the World-Wide Computer are distributed actors performing autonomous, concurrent computations. They communicate via asynchronous message passing. The Remote Message Sending Protocol allows an actor to send a message to any actor in the World-Wide Computer. Furthermore, RMSP is also used for actor migration.

A main difference between RMSP and Java's Remote Method Invocation is the non-blocking semantics of actors. This intentional feature, which is directly provided by RMSP, improves on distributed system implementations with interactions based on RPC or blocking semantics. It is not necessary to explicitly manage a separate thread for network communication.

Because actors do not share memory and references are given in terms of universal actor names, actor migration is much simpler to implement than Java object migration. The serialized nature of actor execution and our universal naming scheme make actor migration as simple as migrating the state of the actor along with the mailbox of pending messages to process. However, migration in our model is delayed until the currently executing message has been completely processed.

6.1.3 Resource Naming and Location

In similar spirit to the World-Wide Web, providing scalability and ease of reference, requires WWC actors to have unique universal names. Universal Actor Names persist over time and allow the unanticipated dynamic construction of arbitrary actor interconnection topologies.

Berners-Lee envisioned for the World-Wide Web, a Uniform Resource Identifier (URI) syntax that includes:

- **Uniform Resource Names** (URN): location-independent references to resources,
- **Uniform Resource Locators** (URL): location-dependent references to resources, and
- **Uniform Resource Citations** (URC): metadata information about resources.

Unfortunately, only Uniform Resource Locators (URLs) have been widely deployed in the World-Wide Web. The World-Wide Web Consortium's work on a Resource Description Framework (RDF), and more recently on XML Schema, appears to cover and expand on the functionality of the originally conceived URCs.

`WWC` actors need to potentially migrate frequently in the accomplishment of their tasks. Therefore a location-dependent naming approach would present a very limiting, faulty design. Instead, we use location-independent names (UANs) to refer to `WWC` actors. Location-dependent Universal Actor Locators help communicate with an actor present in a given location. Both UAN and UALs are URI-compliant specifications of actor names and locators [15].

A naming service maps UANs to UALs. The Universal Actor Naming Protocol (UANP) prescribes the format for communication with naming servers in order to register a name, get an actor's location, update the location after migration, and delete an entry from a server's registry.

6.1.4 Run-time System

The `WWC` infrastructure consists of naming servers and virtual machines hosting universal actors. Naming servers provide the mapping from UANs to UALs, and enable clients to register and lookup names, update mappings and delete entry pairs. Virtual machines hosting universal actors enable actors to migrate from host to host in the `WWC`, to communicate remotely with other universal actors, and to use locally available resources by binding them dynamically as actors move from one hosting environment to another.

The developed `WWC` infrastructure includes two packages: `wwc.naming` and `wwc.messaging`. The first package contains reference implementations for UANs, UALs and the UANP. The second package incorporates the RMSP, and *theaters* as the run-time system for universal actors.

6.2 Operational Infrastructure

We describe in more detail universal actors, the Remote Message Sending Protocol, Universal Actor Names and Locators, and theaters.

6.2.1 Universal Actors

A SALSA behavior is preprocessed by the SALSA compiler into a Java class extending the UniversalActor class. Therefore, every SALSA actor instance is an instance of a class extending the UniversalActor class (see Figure 5.11).

A universal actor extends a base actor (whose only relevant state is a mailbox of pending messages) with a UAN and a UAL. The UAN and UAL classes are defined in the wwc.naming package. Application actors extend universal actors incorporating new state in the form of primitive values, actor references, and new message handlers.

The salsa.language.UniversalActor and the salsa.language.Message classes are serializable. Even though UniversalActor extends java.lang.Thread which is not serializable, actor semantics enables us to create a new thread in the remote virtual machine, with the actor's serialized mailbox and state, and restart it, preserving correct application behavior. The Message class also contains a non-serializable java.language.Method instance variable. We accomplish method serialization by marshaling the method name and argument types, and recreating it in the reading side using the Java reflection API.

If non-serializable Java objects are used inside SALSA behaviors, it is the application programmer's responsibility to write appropriate writeObject and readObject methods to guarantee serializability and therefore correct actor migration.

6.2.2 Remote Message Sending Protocol

The Remote Message Sending Protocol allows WWC actors to send messages to each other by using their Universal Actor Names. RMSP uses a Universal Actor Name dæmon (uand) to map UANs to UALs and performs the actual delivery of a message from one actor to another.

RMSP is an object-based protocol: a customized version of Java object serialization is used for sending data across the wire. When the UAL for a remote actor is obtained, the theater hosting the target actor is contacted by opening a socket to the RMSP server in the host and listening port indicated in the UAL. Once the socket connection is established, the salsa.language.Message object is sent through the wire, using Java object serialization modified as described in 6.2.1.

In the receiving side, the RMSP server gets the Message object, updates its actor references

– previously remote actor references may become local and conversely previously local references become remote – for more efficient communication, finds the local reference to the target object in the hashtable using the relative UAL and proceeds to add the incoming `Message` object to the recipient's mailbox.

A theater's RMSP server not only enables `Message` objects to enter, but also `UniversalActor` objects. When a universal actor enters a theater, its actor references get updated just like in messages, the actor dynamically gets references to the theater's environment actors, and the actor gets restarted locally. If the `UniversalActor` is a `salsa.language.Messenger` – a subclass of universal actors – the process is identical, except that the theater's run-time environment sends a `deliver()` message to the incoming actor after getting it started.

The RMSP protocol is currently very much dependent on (1) the Java programming language and its serialization API, and (2) the `salsa.language` package definition and API. The first limitation may be overcome in the future by exploring an XML based interaction protocol, in a similar spirit to SOAP, a protocol that formats complete RPC interactions through XML. The second limitation is not so bad in the sense that `SALSA` applications do not directly deal with the `salsa.language` package; it is the code generator that moves higher-level remote message sending into Java code using the given API. Evolution in the package needs to be tightly integrated with the code generation process, so that existing high-level `SALSA` code does not break with newer versions of RMSP.

6.2.3 Universal Actor Names and Locators

As described in Section 6.1.3, Universal Actor Names persist over time and allow the unanticipated dynamic construction of arbitrary actor interconnection topologies. UANs also enable actors to migrate without requiring to update all inverse acquaintances – i.e., actors containing a reference to the migrating actor.

A sample Universal Actor Name for a `WWC` actor handling Professor Agha's schedule follows:

```
uan://wwc.osl.cs.uiuc.edu/Agha/Calendar
```

A `WWC` actor may migrate from node to node keeping the same Universal Actor Name but changing its Universal Actor Locator.

Two sample Universal Actor Locators for the actor above are:

```
rmsp://agha.cs.uiuc.edu/PersonalAgents/Calendar
rmsp://howard.cs.uiuc.edu/Professors/AghaCalendar
```

Notice that both Universal Actor Names and Locators are specified with the Uniform Resource Identifier (URI) syntax. The URI syntax specification provides precise details regarding valid URI constructions [15]. The Remote Message Sending Protocol contacts a universal actor name dæmon (`uand`) at the Internet domain name `wwc.osl.cs.uiuc.edu` to get the current location of the actor. Once it has the UAL, it proceeds to deliver the specified message to the Internet domain name specified by the UAL. There is a remote message sending protocol dæmon (`rmspd`) listening on the port specified by the UAL. The default TCP/IP port numbers for UANP and RMSP are 3030 and 4040 respectively.

An actor may migrate after its UAL has been received by potential customers. If communication with a particular location fails, the authoritative answer regarding the current residence of the actor is its Universal Actor Name host.

6.2.4 Theaters and Naming Servers

Theaters enable the use of locally available resources via *environment actors* and contain a communication module for remote message sending and receiving, as well as actor migration – see Figure 4.1. Universal actors in theaters never terminate, since references to universal actors can be created without anticipation from their names (UAN) or even directly from their current locator (UAL).

If the target actor for an incoming message has a name but no location information, the naming server associated with the UAN is contacted using UANP. As a result, a UAL is received and the appropriate theater can be contacted to deliver the message. It the target actor has moved in the mean time, a *forwarder* actor is temporarily left behind in the theater with the goal of forwarding messages to the new location. If no forwarder actor is present, the naming server must be contacted again to get the new location. After a fixed number of unsuccessful message deliveries (in the current implementation, 20), a message is returned to the source actor as undeliverable.

In the current `WWC` implementation, theaters run as Java applications. The `wwc.messaging` package contains an application called `RMSPDaemon`. Starting a theater in an Internet host is accomplished by issuing the following instruction:

```
  --> java wwc.messaging.RMSPDaemon [-p <portNumber>]
```

The optional port number command line argument can be used to listen for incoming RMSP connections in a port other than the default: 3030.

In future versions of the `WWC`, theaters may run on applets. In this case, the RMSP server in charge of communication with actors in the applet theater must reside in the same server as the HTTP server which hosts the applet. This is due to security restrictions prohibiting arbitrary Internet communications by untrusted Java applets.

The naming service is currently implemented as a Java application in the `wwc.naming.locationServer` package. To start a naming server in the WWC, the following line must be issued in a given Internet host:

```
  --> java wwc.naming.locationServer.UANDaemon [-p <portNumber>]
```

The optional port number command line argument can be used to listen for incoming UANP connections in a port other than the default: 4040.

6.3 World-Wide Computer Protocol

The World-Wide Computer Protocol has not been currently implemented. We describe it here as a potential design for bootstrapping and maintaining a worldwide computing infrastructure with the concepts already presented for the `WWC`.

The `WWC` is a set of nodes hosting actors that request and provide services to peers dynamically. A node can start its participation in the `WWC` by contacting any of the currently participating nodes. Once two nodes know each other, they can do load sharing by moving computations from one node to the other. A node may also request to finalize its participation in the `WWC`, effectively migrating all currently hosted `WWC` actors to participating neighbors.

WWC nodes should also detect that a given neighboring node has failed and different policies for partial failure and recovery have to be created. The need for worldwide scalability prevents any centralized approach to reliability and dependability to be used.

We present three types of requests for interaction among WWC nodes: Participation Request, Computation Request and Finalization Request.

6.3.1 Participation Request

In order for a universal actor to start its computational life, it must be bound to a Universal Actor Name and an initial Universal Actor Locator. The UAL must refer to a valid participating theater. When a theater is just started, whether inside a downloaded applet or in a server machine, it uses the code for a *theater director* actor with a set of initial acquaintances (UANs) to broadcast a *participation request.* If participation is approved, the set of initial acquaintances adds the new theater to their list of neighbors as a new WWC neighbor theater. The participation request can be propagated according to neighbor's policies to expand the reachability of the initial theater.

The new theater director keeps a list of valid and active neighbor theaters – those responding to its original participation request. Future participation and finalization requests from those neighbor theaters dynamically change the local set of neighbors. Future requests for computation can be sent, received, or propagated to the set of active neighbor theaters.

6.3.2 Computation Request

The goal of the *Computation Request* method of WWCP is to re-allocate local casts of actors to other theaters. The request must be performed from a participating theater to any known potential theater.

If the endpoint of the computation request – a theater director – accepts the computation, unprocessed tasks in the form of casts of processing actors get migrated to the theater.

The *computation request* also allows a WWC theater to:

- make its computational services known (potentially attracting actors), and
- steal computing work from neighbor theaters.

6.3.3 Finalization Request

The goal of the *Finalization Request* method of WWCP is similar to the goal of the *Computation Request* method. There is one major difference however: all actors currently residing on the finalizing node must be reallocated, and the request is considered to have higher priority than the *Computation Request* method. Additionally, neighbor theaters inactivate or delete the finalizing theater from their set of active neighbors.

6.4 Related Work

The Globus project [37] seeks to enable the construction of *computational grids* providing pervasive, dependable, and consistent access to high-performance computational resources, despite geographical distribution of both resources and users. The Globus Metacomputing Toolkit is a set of core services including resource management, communication, security, information, health and status, remote data access and executable management. A Metacomputing Directory Service (MDS) provides pervasive and uniform access to information about the current state of the grid and its underlying components. The toolkit uses standards (CORBA, DCE, DCOM, Web technologies) whenever possible for both interfaces and implementations. Globus' communication library, Nexus, defines a relatively low-level communication API used to build a wide range of higher-level communication paradigms. Examples of these include message passing, remote procedure calls, stripped transfer and distributed database updates. In Globus' view, the Internet Protocol does not meet the needs of grid applications: its overhead is high, particularly on specialized platforms such as parallel computers. TCP's streaming model is also considered inappropriate for many interactions.

In the World-Wide Computer approach to worldwide computing, we leverage Java's approach to heterogeneity: A heterogeneous network of physical machines is converted into a homogeneous network of virtual machines. Even though the approach simplifies operational expenses – for example, there is no need for an executable management service – it may not allow to explicitly use heterogeneity when desired – for example, appropriately using a supercomputer's inter-processor communication capabilities. If performance is a critical characteristic, however, Java allows to include native code which makes appropriate use of the heterogeneous components of a particular platform, though sacrificing portability. There is no notion in Globus of universal resource names, nor is there any provision for code or resource migration nor coordination abstractions.

To summarize, Globus is being designed to leverage current supercomputing technologies for high performance distributed computing. The World-Wide Computer makes less emphasis on high performance and heterogeneity management. Its target platforms are not limited-access supercomputers but instead any platform running the Java virtual machine, from rings, cards, and phones, to supercomputers. The `WWC` is to run on top of the Internet, using the Web as a starting platform for easy access and massive participation.

The Caltech Infospheres Project [28] deals with the theory and implementation of compositional systems that enable peer-to-peer communication among persistent multithreaded distributed objects. Besides compositionality, Infospheres is studying scalability, dynamic reconfigurability, and high confidence in Internet-based distributed systems. The Infospheres research group uses temporal logic for reasoning about the correctness of distributed object systems and stochastic processes for reasoning about performance and reliability. Their current Infospheres Infrastructure (II) implementation uses Java and Web technologies, but their ideas are applicable to distributed object systems based on other tools such as CORBA and DCOM. Research issues they are investigating, include: object discovery, specifications, verification, dynamic checking, selective assurance, reliability, composable middleware communication mechanisms, aggregatable parts and on-the-fly mediation.

Javelin [30] is an infrastructure for running coarse-grained parallel applications in numerous, anonymous machines. The project also makes emphasis on the importance of ease of participation in order to harness the combined resources of millions of computers connected to the Internet. For this reason, a Java-based approach is taken so that potential users need only have a Java-enabled Web browser. In Javelin's model, there are three kinds of participating entities: brokers, clients and hosts. Clients are processes seeking computing resources, hosts are processes offering them, and brokers are processes that coordinate the supply and demand for computing resources. The model works best with computation-intensive tasks that do not require a lot of communication and coordination.

Berkeley's NOW project [9] has been effectively distributing computation in a "building-wide" scale, and Berkeley's Millennium project [21] is exploiting a hierarchical cluster structure to provide distributed computing on a "campus-wide" scale. WebOS [83] seeks to provide operating system services to wide area applications. 2K [56] is an integrated operating system architecture addressing the problems of resource management in heterogeneous networks, dynamic adaptability, and configuration of component-based distributed applications. Many other projects try to seamlessly integrate networked computers into a worldwide computing framework. `distributed.net` is a project using the Internet to solve tasks requiring large amounts of computing power. Their approach is to distribute the task at hand among multiple clients. As opposed to the World-Wide Computer, Javelin and Infospheres, `distributed.net` uses clients written in C for faster execution. Other Java-based Internet computing projects include ATLAS [11], Charlotte [12], ParaWeb [19] and Popcorn [23].

6.5 Chapter Summary

The World-Wide Computer infrastructure enables the global execution of `SALSA` applications by providing *theaters* as the run-time system for *universal actors*. Universal actors have persistent names (UAN), and non-persistent locators (UAL) which denote the theater where the universal actor is currently executing. The UAN to UAL mapping is maintained by a naming service. Theaters enable transparent remote communication of universal actors using the Remote Message Sending Protocol. The `WWC` additionally enables migration of actors. While the general model for migration includes casts, or groups of highly correlated actors, the current `WWC` architecture implementation does not directly support cast migration. Furthermore, we have introduced an initial design of a World-Wide Computer Protocol (WWCP) for resource discovery and load sharing, which could be implemented in an industrial-strength version of the World-Wide Computer.

Chapter 7

Experimental Results

We illustrate our programming language and run-time architecture using the following examples: three multicast protocols, a messenger carrying remote exception-handling code, a worldwide migrating agent, and a buffering messenger.

7.1 Multicast Protocols

We have implemented a simple multicast protocol, as well as two variations of an acknowledged multicast protocol. By an acknowledged multicast protocol, we mean that the receipt of a message by different members of a cast is acknowledged to an outside actor, signaling that all actors have received their respective messages.

7.1.1 Basic Multicast

Figure 7.1 shows the trace of one possible execution of multicasting. Vertical lines represent local time (increasing in the downward direction) and diagonal lines represent message traversals between actors. Even though the messages are originally directed to actors `a1`, `a2`, and `a3`, `SALSA`-generated code creates a join director, `d`, in charge of sending them their respective messages.

7.1.2 Acknowledged Multicast

Figure 7.2 shows a possible trace of acknowledged multicasting. The join director waits for acknowledgment of message receipt by the coordinated actors, before notifying an outside actor of "finalization" with a `done()` message.

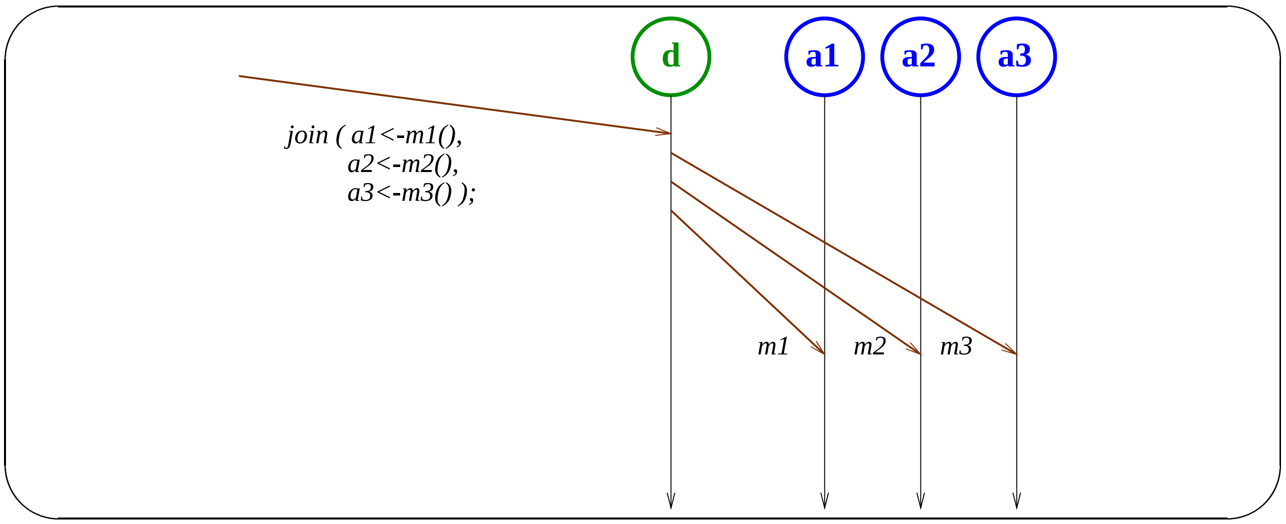


Figure 7.1: Multicast to three actors `a1`, `a2`, and `a3` with a single message to their director `d`. The messages `m1`, `m2`, and `m3` may be different

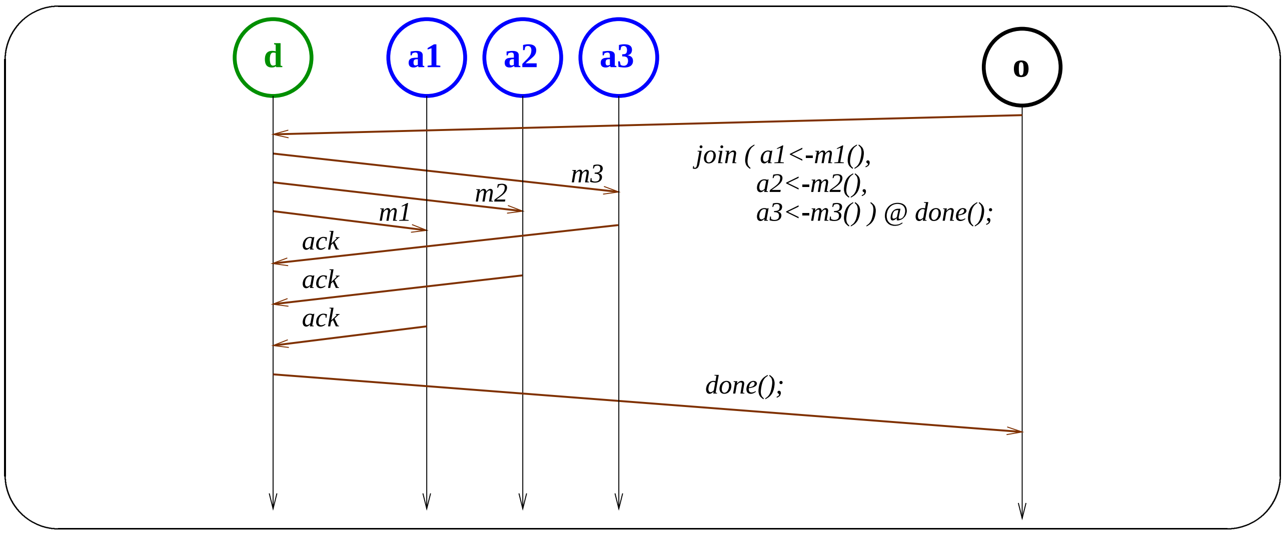


Figure 7.2: Acknowledged multicast: Actors acknowledge director on message receipt

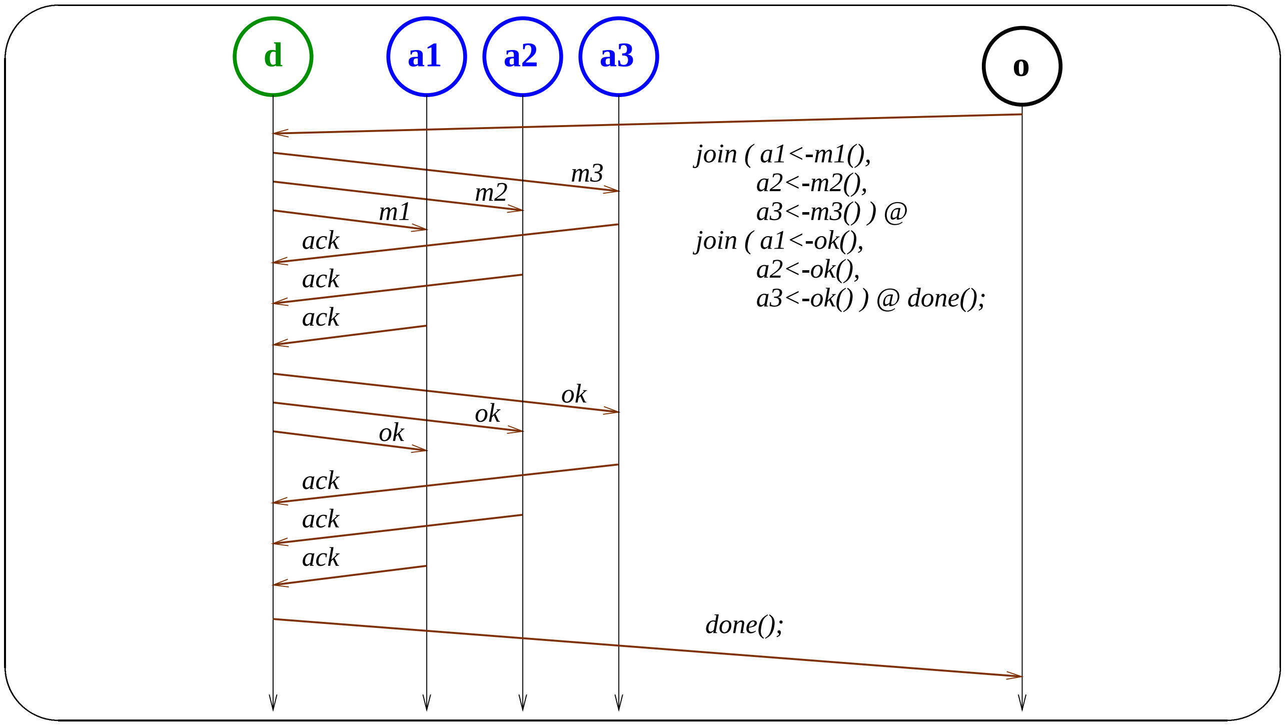


Figure 7.3: Group knowledge multicast: Actors `a1`, `a2`, and `a3` acknowledge message receipt and acknowledge `ok` message from director `d` signaling acknowledgment by everybody.

7.1.3 Group Knowledge Multicast

To implement *group knowledge* [34] – i.e., everybody in the cast knows that everybody in the cast got the message – we can use a two-phase acknowledged multicast protocol. In the first phase, the director sends a message to every actor in the cast. In the second phase, the director tells every actor in the cast that everybody acknowledged message receipt. Figure 7.3 shows a sample trace.

7.1.4 Performance Results

We have programmed the three multicast protocols with the Actor Foundry 0.1.9 [66]. The Foundry was executed on Solaris 2.5.1 and configured to use a reliable communication protocol implemented over UDP. Figure 7.4 shows the timing as measured from the outside actor, when all actors involved in the multicast protocols resided in the same host. We used an empty message (no data) and we averaged the time taken over multiple executions. Time grows linearly with the number of actors, and the slope is doubled with reliability: it takes twice as many messages to do acknowledged multicast than it takes to do basic multicast, and it takes twice as many messages to do group

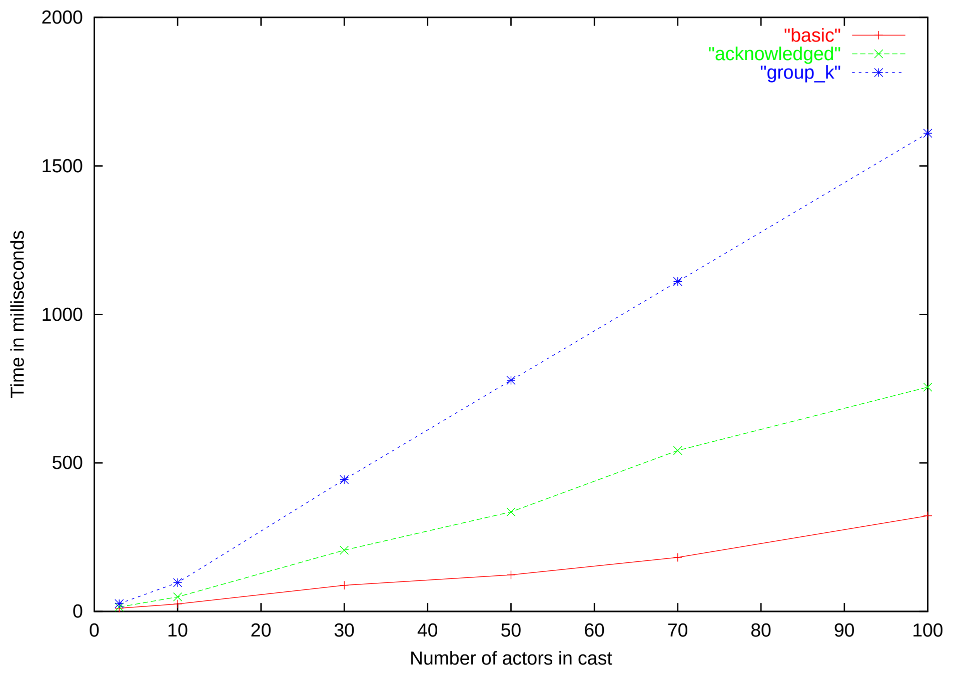


Figure 7.4: Performance of different multicast protocols (*in ms*) using the Actor Foundry 0.1.9

knowledge multicast than to do acknowledged multicast.

We have also implemented the same multicast protocols in `SALSA` 0.3.2. Figure 7.5 shows the results of timing the multicast protocols. The `SALSA` implementation results in better performance due to several factors. First, its recognition of locality of the participating actors and consequent use of local messages as opposed to reliable UDP. Second, the lightweight implementation of actors in `SALSA`'s library which is tightly integrated to the code generation process. Third, Actor Foundry's modular architecture (see Section 5.6.1) enables customization of the transport layer, the request handler, and so forth, yet paying a performance price. Note that the acknowledged multicast and the group knowledge multicast take a long time for the join director creation when the number of actors in the cast is small, however the time is amortized as the casts grow in size.

Perhaps more important than performance is the comparison in the size of these programs. Table 7.1 shows the number of lines of code in the programs (including the code for time measurements) using the Actor Foundry, using SALSA, and for reference, the lines of Java code generated by the SALSA preprocessor for these examples. Notice how join continuations and token-passing continuations can drastically reduce the size and complexity of concurrent programs – in this ex-

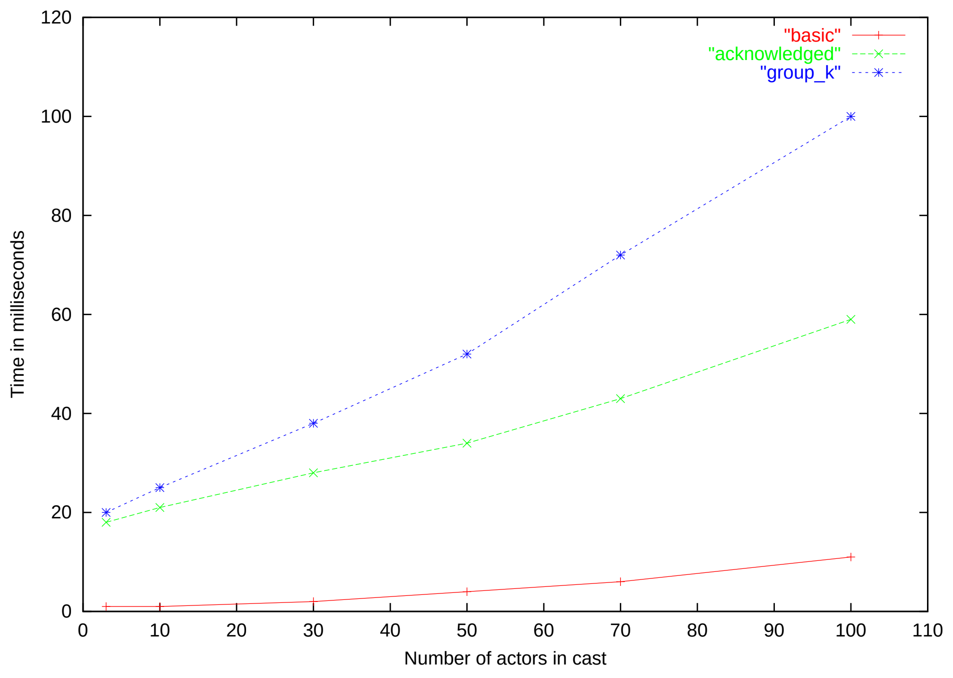


Figure 7.5: Performance of different multicast protocols (*in ms*) using **SALSA** 0.3.2

	Foundry	SALSA	Java
Shared Code	40	10	34
Basic Multicast	146	27	115
Acknowledged Multicast	60	21	134
Group-knowledge Multicast	73	24	183
TOTAL	319	82	466

Table 7.1: Lines of Code Comparison for Multicast Protocols

ample, a four-fold improvement.

7.2 Remote Exception Handling with a Messenger

In our second example, we use a *messenger* to improve locality in exception handling across remotely located casts.

Suppose we have a `sender` actor that wants to send a `letter` to a `recipient` for processing. Furthermore, let's suppose that the `sender` and `recipient` actors are in different hosts as shown in Figure 7.6. If the recipient cannot process the letter, an exception would be thrown from the recipient's host to the sender's host. After that, the sender may want to retry sending the letter at

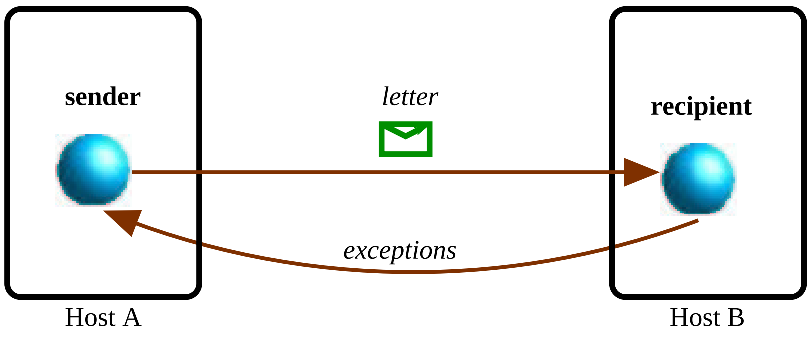


Figure 7.6: Passive message across hosts

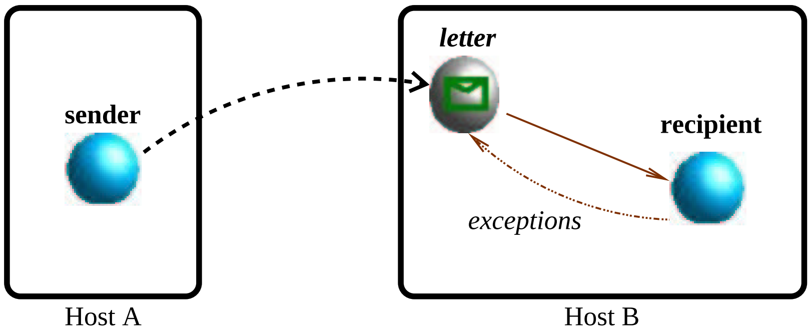


Figure 7.7: `Messenger` across hosts

a later time.

We could improve locality using messengers by creating a new actor in host A and migrating it to host B with the specific behavior of sending the original message and handling any associated exceptions that may arise remotely, as shown in Figure 7.7.

The `sender` actor embodies the main program (see Figure 7.8.) It creates a recipient and a letter and sends the letter with three different exception handling protocols. The first one sends a passive message and ignores failures in the target actor. The second and third invocations use a messenger to handle exceptions at the target host appropriately – with methods `retry` and `failed` respectively. The sender also provides a method `log` in case the messenger ultimately fails to deliver the letter.

The recipient is an actor that provides a method to process a letter. Once a letter has been processed, it returns `null`. If the letter needs to be delegated to a third actor, such actor reference is returned. If the actor cannot temporarily process the letter, the method returns its own reference

```
behavior Sender {

  Letter letter;
  Recipient recipient;

  void act(String[] arguments){

    /** Create letter messenger and recipient actor */
    letter = new Letter(this);
    recipient = new Recipient();

    /** Send a letter to a recipient, with possible failure */
    recipient<-process(letter);

    /** Migrate letter messenger to recipient's theater */
    letter<-migrate(recipient.getUAL());

    /** Send letter retrying until successful */
    recipient<-process(letter) @ letter<-retry;

    /** Send letter with special exception handling */
    recipient<-process(letter) @ letter<-failed;
  }

  void log(Recipient recipient){
    standardError<-println(recipient + "failed processing the letter." );
  }

}
```

Figure 7.8: Messenger Example in SALSA: Sender Code

```
behavior Letter {
  Sender sender;

  /** Letter constructor */
  Letter(Sender s){
    sender = s;
  }

  /** Retries sending the letter until successfully processed */
  void retry(Recipient recipient){
      if (recipient != null)
        recipient<-process(this) @ this<-retry;
    }

  /** Try once again after a second, else log an error */
  void failed(Recipient recipient){
      if (recipient != null){
        System.wait(1000);
        recipient<-process(this) @ this<-failedTwice;
      }
  }

  /** Tell the sender of the letter that it could not be processed */
  void failedTwice(Recipient recipient){
      if (recipient != null)
        sender<-log(recipient);
  }

}
```

Figure 7.9: Messenger Example in SALSA: Letter Code

for processing at a later time.

The `letter` messenger (see Figure 7.9) can either:

- retry forever until the letter is successfully processed, or
- retry once after one second and upon a second failure, send an exception message to the original sender to log an error.

Note that by migrating the letter actor to the recipient theater, we are able to handle exceptions much more efficiently at the target actor's remote host.

7.3 Worldwide Migrating Agent

Our third example, concerns a worldwide migrating agent, which keeps track of its itinerary and can print it in its current location on request. The SALSA code is shown in Figure 7.10.

The agent program receives a universal name and a list of theaters to visit as command-line

```
module migration;

import wwc.naming.*;

behavior Agent {

    StringBuffer itinerary; UAL initialLocation; long initialTime; int hops;

    void startTime(){ initialTime = System.currentTimeMillis(); }
    void printItinerary(){ standardOutput<-println(itinerary);  }

    void addLocation(String ual){
        if (itinerary == null){
            itinerary = new StringBuffer(ual);
            initialLocation = new UAL(ual);
        }
        else {
            hops++;
            itinerary.append(" " + ual);
        }
    }


    void printTime(){
        standardOutput<-println("Migrated "+hops+" times.");
        if (this.getUAL().equals(initialLocation)) {
            standardOutput<-println("Time ellapsed: "+ new Long
                (System.currentTimeMillis()-initialTime))@
            standardOutput<-println("Migration avg time: "+ new Long
                ((System.currentTimeMillis()-initialTime)/hops));
        }
    }

    void go(String[] args, Integer iObject){
        int i = Integer.intValue(iObject);
        if (i < args.length){
            addLocation(args[i]) @
                migrate(args[i]) @
                printItinerary() @
                printTime() @
                go(args, new Integer(++i));
        }
    }

    void act(String[] args){
        try {
            addLocation(args[1])@
                bind(args[0], args[1]) @
                startTime()@
                printItinerary() @
                go(args, new Integer(2));
        } catch (Exception e){
            standardOutput<-println(e);
            standardOutput<-println("Usage: java migration.Agent UAN UAL (<UAL>)*");
        }
    }
}
```

Figure 7.10: Worldwide Migrating Agent in SALSA

Machine Name	Location	OS-JVM	Processor
yangtze.cs.uiuc.edu	Urbana, IL, USA	Solaris 2.5.1-JDK 1.1.6	Ultra 2
vulcain.ecoledoc.lip6.fr	Paris, France	Linux 2.2.5-JDK 1.2pre2	PII, 350MHz
solar.isr.co.jp	Tokyo, Japan	Solaris 2.6-JDK 1.1.6	Sparc 20

Table 7.2: World-Wide Computer Test-bed

Local actor creation time	386 μs
Local message sending time	148 μs
LAN message sending time	30-60 ms
WAN message sending time	2-3 secs
LAN actor migration time	150-160 ms (minimal actor)
LAN actor migration time	240-250 ms (actor with 100 Kb of data)
WAN actor migration time	3-7 secs (minimal actor)
WAN actor migration time	25-30 secs (actor with 100 Kb of data)

Table 7.3: Local and Wide Area Network Time Ranges

arguments. The agent is initially bound to its universal name and home theater. Once the agent has migrated to its home theater it records the local time in that theater. For each new theater in the list, the agents migrates there (updating its itinerary of theaters visited) and prints the current itinerary. If and when the agent revisits its home theater (`initialLocation`) it prints the time which has elapsed since its first visit. Notice that the original initial time travels with the agent as part of its state, but it only makes sense to use it in the agent's home theater.

We have used the example to make a preliminary measure of performance of different aspects of remote communication and migration in `WWC` systems. We have used a test-bed with three LANs and one WAN, as depicted in Table 7.2.

Table 7.3 shows the ranges of values we have measured in our tests with minimal actors and empty messages, as well as with larger actors (up to 100 Kb of data).

We have not tried to optimize the current implementation of SALSA programs, the UAN service, or the RMSP server. [1] The main goal of this data is to understand the impact of the different factors in terms of orders of magnitude, not in terms of absolute values. These numbers give us an estimate of the difference in communication and migration times in local, local-area and wide-area

[1]To perform these tests, we used SALSA version 0.3.2. The local times were computed using JDK 1.1.8. on Linux 2.2.16 running on a Pentium III. Local message sending was two times faster than reported above using JDK 1.2 on Windows 2000, and three times faster using JDK 1.3 on Linux 2.2.16.

scenarios: local times are in microseconds, LAN times are in milliseconds and WAN times are in seconds.

7.4 A Buffering Messenger

Specialized implementations of messengers (subclasses) may enable multiple modes of remote communication. For example, a *buffering messenger* allows more efficient asynchronous remote communication, by storing messages locally at a given theater before actually delivering them to the final destination. In particular, when wide area networks are being used, the cost of remote message sending can be as expensive as actor migration. Even in a local area network, actor migration is only 3 to 8 times more expensive than remote message sending (see Table 7.3). Since actor communication is asynchronous and the actor model makes no guarantees on the ordering of arrival or processing of messages, we can use buffering messengers to improve the overall responsiveness and network usage of distributed actor systems.

The most obvious buffering messenger implementation creates a messenger for every target theater. New messages to actors in the theater represented by a messenger are buffered internally until a certain messenger size is reached or a given timeout expires. The most effective messenger size and delivery timeout are highly dependent on several factors. Among those factors are: application-dependent topology of communication, local or wide area system distribution, message sizes and frequencies, network traffic and actor migration rates. Since the nature of the World-Wide Computer is highly dynamic, we must devise adaptable strategies and heuristics for tuning a system's performance.

The performance of a buffering messenger is equivalent to the performance of actor migration – local message sending times are negligible. A buffering messenger is more efficient than sending n messages of size s, when messenger migration time is smaller than n remote message sending times:

$$T_{Migration}(n * s) < n * T_{MessageSending}(s)$$

In the worst case scenario tested, it is better to use a messenger in a wide area context, when the messenger can carry more than 10 messages to its final destination at once. Since messages may be directed to multiple actors which are resident in the same theater, it is not an unreasonable requirement at all.

7.5 Chapter Summary

We have introduced some applications of `SALSA` and the World-Wide Computer architecture: three multicasting protocols, a messenger application localizing exception handling, a worldwide migrating agent and a buffering messenger. We have also given some preliminary performance times for the architecture in a worldwide test-bed. Running code for these and other examples can be found at: `http://osl.cs.uiuc.edu/salsa`.

Chapter 8

A Hierarchical Model for Coordination

Application programmers have difficulties dealing with the increasing complexity of coordination of concurrent activities in distributed systems. Traditional coordination mechanisms rely on a shared space, and therefore do not scale to worldwide execution. Reflective approaches provide a lot of flexibility but often require complicated meta-level infrastructures.

The hierarchical model [86] is motivated by social organizations, where groups and hierarchical structures allow for effective information flow and coordination of activities. The basic idea is not entirely new. For example, Simon proposed hierarchies as an appropriate model to represent different kinds of complex systems – viz., social, biological and physical systems as well as systems carrying out symbolic computations [72]. The proposal was based on the observation that interactions between components at different levels in a hierarchy are often orders of magnitude smaller than interactions within the sub-components. Simon argues that the near-decomposability property of hierarchic systems also allows for their natural evolution and illustrates the argument with examples drawn from systems such as multi-cellular organisms, social organizations, and astronomical systems.

Regardless of the philosophic merits of Simon's proposition, we believe that a hierarchical organization of actors for coordination is not only fairly intuitive to use but quite natural in many contexts. By using a hierarchical model, we simplify the description of complex systems by repeatedly subdividing them into components and their interactions.

Coordination in the hierarchical model is accomplished by constraining the receipt of messages that are destined for particular actors. An actor can only receive a message when the coordination constraints for message reception are satisfied. The coordination constraints are checked for conformance by special actors designated as *directors.* The group of actors coordinated by a director is defined as a *cast.*

8.1 Directors and Casts

Although the director of a cast plays special roles – such as the "coordinator" of actors internal to the cast, an interface to other casts, a request broker for the service provided by the cast, and so forth – a director has no more "power" than other actors that are in the cast. That is, as far as the run-time system is concerned, a director is just another actor. It is also possible to have completely "uncoordinated" actors – i.e., actors which do not belong to a cast and which therefore have no external constraints on message receipt and processing.

An actor can have at most one director at a given point in time. However, a director may itself belong to a cast and thus be coordinated by another director. The strategy allows for dynamic reconfigurability of coordination constraints, as well as for modular constraint composition. The director-actor relationship forms a set of trees, named the *coordination forest.*

A message from a `sender` actor is received by a `target` actor only after *approval* by all the directors in the target actor's coordination forest path up to the first common director, if such a director exists, and otherwise, approval is required of all directors in the `target`'s coordination forest path up to the top level.

Figure 8.1 shows a sample actor configuration. We illustrate valid message paths by describing a few examples based on the sample configuration:

- A message from any actor can go directly to actor `a`, or to actor `k`.
- A message from actor `f` to actor `g` has to go through their first common director, actor `e`.
- A message from `c` to `f` has to go through directors `b` and `e`. However a message from `f` to `c` only needs to be approved by `b` – because `e` is *not* in `c`'s coordination forest path.
- A message from `k` to `b,c,...,j` has to be approved at least by `a`.

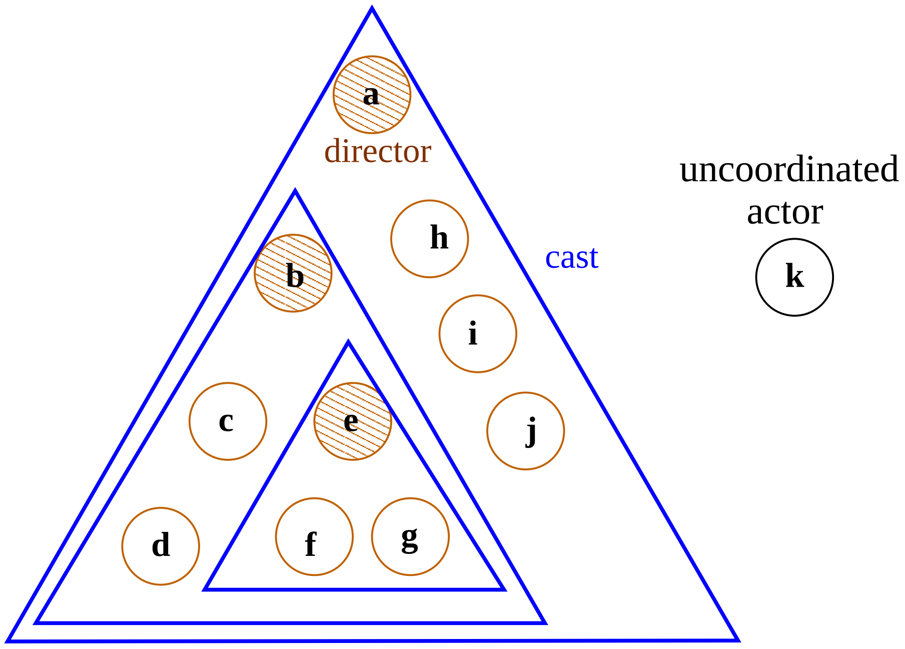


Figure 8.1: Hierarchical organization of *casts* coordinated by *directors*.

8.2 Operational Semantics

We present an operational semantics for the hierarchical model for coordination, with the goal of reducing ambiguities in message paths, to provide a basis for reasoning, and to guide future programming language and system implementations.

We extend the operational semantics formulated by Agha, Mason, Smith and Talcott [7] to capture the concept of casts and directors. Specifically, we add a primitive to the language for creating directed actors, we add a δ function which maps actors to directors, and we tag messages to show that they require approval. We perform a validation of hierarchies in the coordination structure, and we inductively define coordination forest paths, to be used for message redirection and reception in the single-step configuration transitions.

The following two sections introduce the language used and the reduction rules that define valid transitions between actor configurations.

8.3 A Simple Lambda Based Actor Language

Our actor language is a simple extension of the call-by-value lambda calculus that includes (in addition to arithmetic primitives and structure constructors, recognizers and destructors) primitives for creating and manipulating actors. An actor's behavior is described by a closure which embodies the code to be executed when a message is received, and the local store (variables and their values). The four actor primitives are:

`newactor`(e) creates a new uncoordinated actor with behavior e and returns its name.

`newdirectedactor`(e) creates a new actor with behavior e, directed by the creator, and returns its name.

`send`(v_0, v_1) creates a new message with receiver v_0 and contents v_1 and passes it to the message delivery system.

`ready`(v) signals the end of the current computation and the ability of the actor to process the next message with behavior v.

8.4 Operational Semantics for Coordinated Configurations

We give the semantics of actor expressions by defining a transition on coordinated open configurations. We first define values, expressions, messages, coordinated open configurations and coordination forest paths. Then, we proceed to define the single-step transition relation among actor configurations.

8.4.1 Values, Expressions and Messages

We take as given countable sets At (atoms) and X (variables). We assume At contains t, nil for booleans, as well as integers. F_n is the set of primitive operations of rank n, which includes arithmetic operations, branching, pairing and actor primitives `newactor`, `newdirectedactor`, `send` and `ready` (ranks 1,1,2 and 1).

Definition ($\mathsf{V}\quad\mathsf{E}\quad\mathsf{M}$): The set of *values*, V, the set of *expressions*, E, and the set of *messages*, M, are defined inductively as follows:

$$\mathsf{V} = \mathsf{At} \cup \mathsf{X} \cup \lambda\mathsf{X}.\mathsf{E} \cup \mathtt{pr}(\mathsf{V},\mathsf{V})$$

$$\mathsf{E} = \mathsf{V} \cup \mathtt{app}(\mathsf{E},\mathsf{E}) \cup \mathsf{F}_n(\mathsf{E}^n)$$

$$\mathsf{M} = \texttt{<}\mathsf{V} \Leftarrow \mathsf{V}\texttt{>}_{\mathsf{X}}$$

We use variables for actor names. An actor can be either ready to accept a message, written $\mathtt{ready}(v)$, where v is a lambda abstraction denoting its behavior; or busy executing an expression, written e. A message from a source actor a targeted to an actor with name v_0 and contents v_1 is written $\texttt{<}v_0 \Leftarrow v_1\texttt{>}_a$. We use the subscript a – the source actor – for bookkeeping to determine which directors need to approve the message.

8.4.2 Coordinated Actor Configurations

An actor configuration models an actor system with a set of actors, messages in transit and an interface to the external environment.

Let $\mathbf{P}_\omega[\mathsf{X}]$ be the set of finite subsets of X, $\mathbf{M}_\omega[\mathsf{M}]$ be the set of (finite) multi-sets with elements in M, $\mathsf{X}_0 \xrightarrow{f} \mathsf{X}_1$ be the set of finite maps from X_0 to X_1, $\mathrm{Dom}(f)$ be the domain of f and $\mathrm{FV}(e)$ be the set of free variables in e. We define actor configurations as follows.

Definition (Actor Configurations (K)): An *actor configuration* with actor map, α, multi-set of messages, μ, director map, δ, receptionists, ρ, and external actors, χ, is written

$$\left\langle \alpha \;\middle|\; \mu \;\middle|\; \delta \right\rangle_{\chi}^{\rho}$$

where $\rho, \chi \in \mathbf{P}_\omega[\mathsf{X}]$, $\alpha \in \mathsf{X} \xrightarrow{f} \mathsf{E}$, $\delta \in \mathsf{X} \xrightarrow{f} \mathrm{Dom}(\alpha)$, $\mu \in \mathbf{M}_\omega[\mathsf{M}]$, and let $A = \mathrm{Dom}(\alpha)$ and $D = \mathrm{Dom}(\delta)$[1], then:

(0) $\rho \subseteq A$, $A \cap \chi = \emptyset$, and $D \subset A$

(1) if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, and if $\texttt{<}v_0 \Leftarrow v_1\texttt{>}_a \in \mu$

[1] D stands for "Directed Actors", not "Directors"

then $\mathrm{FV}(v_i) \subseteq A \cup \chi$ for $i \in 0, 1$.

(2) if $a \in D$, then $\exists a_0, a_1, ..., a_n \in A$ such that $a_0 = \delta(a)$, $a_1 = \delta(a_0)$, $a_2 = \delta(a_1)$, ..., $a_n = \delta(a_{n-1})$, and $a_n \notin D$.

The last rule restricts actor configurations so that all directors in the coordination forest path for an actor belong to the same configuration and no cycles are allowed in the actor-director relationship.

8.4.3 Coordination Forest Paths

The coordination forest path for a given actor is defined as a set containing the actor itself and all its directors up to the top-most director. An external message – i.e., a message coming from an outside actor – needs to be approved by all the actors in a target actor's coordination forest path.

Definition (Coordination Forest Paths (Δ)): The *coordination forest path* of an actor a, $\Delta(a)$, in a configuration κ with director map δ, is defined as:

$$\Delta(a) = \begin{cases} \{a\} & \text{if } a \notin \mathrm{Dom}(\delta) \\ \{a\} \cup \Delta(\delta(a)) & \text{otherwise} \end{cases}$$

8.4.4 Single-step Transition Relation

There are three kinds of transitions between coordinated actor configurations:

1. **Local transitions** model actor behavior as in sequential functional programs

2. **Actor transitions** model basic actor primitive operations - actor creation, message sending and message reception

3. **Input/Output transitions** model interactions between the system and its environment. I/O transitions may change the system's interface to its environment.

The **local transition fun** is inherited from the purely functional fragment of our actor language. The transition represents progress inside a single actor.

The **actor transitions** are:

`newactor`: creation of an undirected actor, returning its address.

`newdirectedactor`: creation of an actor directed by the creator, returning its address.

`send`: message send, passes the message to the mail delivery system.

`redirect`: message redirection to the director of the target actor, if the message needs approval.

`receive`: message reception by an actor.

The **input/output transitions** represent interactions with external agents through the environment:

`out`: emitting a message to the environment, with target in χ. This transition may change the set of receptionists, ρ.

`in`: arrival of a message from the environment, with target in ρ. This transition may change the set of external actors, χ.

To describe the actor transitions between configurations other than message receipt and redirection, a non-value expression is decomposed into a reduction context filled with a redex. Reduction contexts are expressions with a unique hole, and serve the purpose of identifying the subexpression of an expression that is to be evaluated next. Reduction contexts correspond to the standard reduction strategy (left-first, call-by-value) of Plotkin [69] and were first introduced by Felleisen and Friedman [35]. We use the sign $\square$ for the hole occurring in a reduction context, and call such holes redex holes.

Definition ($\mathsf{E}_{\rm rdx}$ R): The set of *redexes*, $\mathsf{E}_{\rm rdx}$, and the set of *reduction contexts*, R, are defined by:

$$\mathsf{E}_{\rm rdx} = \mathsf{app}(\mathsf{V},\mathsf{V}) \cup \mathsf{F}_n(\mathsf{V}^n)$$

$$\mathsf{R} = \{\square\} \cup \mathsf{app}(\mathsf{R},\mathsf{E}) \cup \mathsf{app}(\mathsf{V},\mathsf{R}) \cup \mathsf{F}_{n+m+1}(\mathsf{V}^n,\mathsf{R},\mathsf{E}^m)$$

We let R range over R and r range over $\mathsf{E}_{\rm rdx}$.

An expression e is either a value or it can be decomposed uniquely into a reduction context filled with a redex. Thus, local actor computation is deterministic.

Lemma (Unique decomposition):

(0) $e \in \mathsf{V}$, or

(1) $(\exists! R, r)(e = R[r])$

Proof : An easy induction on the structure of e.□

The purely functional redexes inherit the operational semantics from the purely functional fragment of our actor language. The actor redexes are: $\texttt{newactor}(e)$, $\texttt{newdirectedactor}(e)$, $\texttt{send}(v_0, v_1)$, and $\texttt{ready}(v)$.

Definition ($\mapsto$): The single-step transition relation $\mapsto$, on actor configurations is the least relation satisfying the following conditions:[2]

<$\texttt{fun}$: a>

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \left\langle \alpha\{a \to e\} \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha\{a \to e'\} \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi}$$

<$\texttt{newactor}$: a, a'>

$$\left\langle \alpha\{a \to R[\texttt{newactor}(e)]\} \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \mapsto$$

$$\left\langle \alpha\{a \to R[a'], a' \to e\} \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

<$\texttt{newdirectedactor}$: a, a'>

$$\left\langle \alpha\{a \to R[\texttt{newdirectedactor}(e)]\} \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \mapsto$$

$$\left\langle \alpha\{a \to R[a'], a' \to e\} \;\middle|\; \mu \;\middle|\; \delta\{a' \to a\} \right\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

<$\texttt{send}$: a, v_0, v_1>

$$\left\langle \alpha\{a \to R[\texttt{send}(v_0, v_1)]\} \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \mapsto$$

$$\left\langle \alpha\{a \to R[\texttt{nil}]\} \;\middle|\; \mu, \texttt{<} v_0 \Leftarrow v_1 \texttt{>}_a \;\middle|\; \delta \right\rangle^{\rho}_{\chi}$$

[2]For any function f, $f\{\mathrm{x} \to \mathrm{x}'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{\mathrm{x}\}$, $f'(\mathrm{y}) = f(\mathrm{y})$ for $\mathrm{y} \neq \mathrm{x}$, $\mathrm{y} \in \mathrm{Dom}(f)$, and $f'(\mathrm{x}) = \mathrm{x}'$. $\texttt{msg}(v_0, v_1)$ is a value expression representing a message with target v_0 and contents v_1.

$\texttt{<redirect} : a, v_0, v_1\texttt{>}$

$$\left\langle \alpha \;\middle|\; \mu, \texttt{<}v_0 \Leftarrow v_1\texttt{>}_a \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha \;\middle|\; \mu, \texttt{<}\delta(v_0) \Leftarrow \texttt{msg}(v_0, v_1)\texttt{>}_a \;\middle|\; \delta \right\rangle^{\rho}_{\chi}$$

if $v_0 \in \mathrm{Dom}(\delta)$, $\delta(v_0) \neq a$, and $v_0 \notin \Delta(a)$

$\texttt{<receive} : v_0, v_1\texttt{>}$

$$\left\langle \alpha\{v_0 \to \texttt{ready}(v)\} \;\middle|\; \texttt{<}v_0 \Leftarrow v_1\texttt{>}_a, \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \mapsto$$

$$\left\langle \alpha\{v_0 \to \texttt{app}(v, v_1)\} \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi}$$

if $v_0 \notin \mathrm{Dom}(\delta)$, or $\delta(v_0) = a$, or $v_0 \in \Delta(a)$

$\texttt{<out} : v_0, v_1\texttt{>}$

$$\left\langle \alpha \;\middle|\; \mu, \texttt{<}v_0 \Leftarrow v_1\texttt{>}_a \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho'}_{\chi}$$

if $v_0 \in \chi$, and $\rho' = \rho \cup (\mathrm{FV}(v_1) \cap \mathrm{Dom}(\alpha))$

$\texttt{<in} : v_0, v_1\texttt{>}$

$$\left\langle \alpha \;\middle|\; \mu \;\middle|\; \delta \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha \;\middle|\; \mu, \texttt{<}v_0 \Leftarrow v_1\texttt{>}_{a'} \;\middle|\; \delta \right\rangle^{\rho}_{\chi \cup (\mathrm{FV}(v_1) - \mathrm{Dom}(\alpha))} \qquad a' \text{ fresh}$$

if $v_0 \in \rho$, $\mathrm{FV}(v_1) \cap \mathrm{Dom}(\alpha) \subseteq \rho$

The *redirect* and *receive* transition rules ensure that all target actor directors up to the least common director between the sender and the target actors (or the root director in the target's coordination forest path) get notified. The directors, thus, control when to actually send a message to a target actor. Notice the locality of information flow: if two actors belong to the same cast, outside actors and directors, even if in their coordination hierarchy, need not be notified/interrupted. For the configuration sample in Figure 8.1, a message from `c` to `f`, $\texttt{<}f \Leftarrow v\texttt{>}_c$, is redirected to `e`, $\texttt{<}e \Leftarrow msg(f, v)\texttt{>}_c$, and subsequently to `b`, $\texttt{<}b \Leftarrow msg(e, msg(f, v))\texttt{>}_c$. After checking coordination constraints, `b` and `e` send it to the final target `f`. Director `a` is not involved in this internal coordination.

The last two transitions account for the openness of actor configurations. The first transition, *out*, represents a message delivered to an external actor. The second transition, *in*, represents a message coming from the environment to one of the configuration's receptionists. Notice that the message will be redirected to the root of the receptionist's coordination forest path.

8.5 Related Work

Traditional models of coordination require reflective capabilities in the implementation architecture; such reflection is used to construct meta-level actors that can intercept and control base-level actors (e.g., see [63, 74, 89, 91]). The need for reflection creates two difficulties. First, it requires a specialized run-time system. Second, it complicates the semantics: correctness of a particular application not only depends on the semantics of application level actors, but also on the semantics of the meta-level actors implementing the reflective architecture.

The hierarchical model proposed, only uses abstractions that are themselves first-class actors. In particular, there is no requirement for meta-level actors which are able to intercept base-level actor messages. The cost of its simplicity results in at least two major disadvantages for the hierarchical model. First, the model does not enable the degree of transparency that can be afforded by defining coordination abstractions using reflective architectures. Second, because the hierarchical model is limited to customizing communication, it is not as flexible as a reflective model. However, observe that the actor model represents all coordination through its message-passing semantics; thus customizing communication is far more powerful than it may first appear to be.

There are two major differences between the hierarchical model's directors and the meta-actor stack of the onion skin model [3]. First, directors only manipulate messages, not behavior, so they are simpler to deal with. Second, directors are shared among multiple "base-level" actors, which enables multi-party coordination protocols to be more easily implemented.

While the hierarchical model's concepts of actor casts and directors can be implemented using CORBA for coordination of concurrent activities, the concept of mobile code is only realizable with an approach like Java's virtual machine [60] that allows actors and actor behaviors to be safely sent and interpreted across heterogeneous machines.

A number of coordination models and languages [31, 42] use a globally shared tuple space as a means of coordination and communication [26]. A predecessor of Linda, using pattern based data storage and retrieval was the Scientific Community Metaphor [57]. *Sprites*, the Scientific

Community's computational agents, share a monotonically increasing knowledge base; on the other hand, Linda allows communication objects (tuples) to be removed from the tuple space. Additional work in this direction includes adding types to tuple spaces for safety, using objects instead of tuples and making tuple spaces first-class entities [47, 52, 62].

In contrast to these approaches, the hierarchical model of actor coordination is based purely on point-to-point communication, which requires the sender to explicitly name the target of a message. Our use of the term coordination is different: following Wegner, we define coordination as *constrained interaction* [90] between actors. Constrained interaction does not require shared spaces, which limit scalability.

8.5.1 ActorSpaces

ActorSpaces [22] are computationally passive containers of actors. Messages may be sent to one or all members of a group defined by a destination pattern. *Casts* differ from ActorSpaces in that the management of messages coming to a group is handled by a director, which is a computationally active component, and can therefore explicitly implement multiple group messaging paradigms. On the other hand, the hierarchical model requires explicit description of how actors are grouped together, since policies such as pattern matching based on actor attributes are not directly provided.

8.5.2 Synchronizers

Synchronizers [38, 39] are linguistic constructs that allow *declarative* control of coordination activity between multiple actors. Synchronizers allow two kinds of specifications: messages received by an actor may be disabled and, messages sent to different actors in a group may be atomically dispatched. These restrictions are specified using message patterns and may depend on the synchronizer's current state. *Directors* differ from Synchronizers in that directors are not declarative – rather a director is an actor: a director can receive messages and a director may itself be subject to coordination constraints. Coordination of directors not only allows for dynamic reconfigurability of coordination policies but also for a hierarchical composition of constraints.

The hierarchical model is more restrictive in that it requires actors to belong to a single cast at a given time. By contrast, the groups controlled by synchronizers may overlap arbitrarily.

However, because the path for messages can be more rigid and the coordination more determinate in the hierarchical organization than in synchronizers, the former has the associated benefit of more predictable performance.

8.5.3 Interaction and Operational Semantics

Interaction semantics uses diagrams and paths for reasoning about composition in open distributed systems [79, 80]. Talcott's work concentrates on studying the sets of sequences of interactions of components (actor configurations) with their environment. The goal is to hide as much as possible from the details of a particular actor system, yet preserving a compositional semantics. Properties of groups of components can therefore be carried over from the properties of the components themselves. In the hierarchical model, we define an operational semantics based on work by Agha, Mason, Smith and Talcott [7]. We extend their transition system to handle a new primitive for directed actor creation, and to precisely define the path for messages within the coordination forest of a given configuration. We have not studied equivalence relations in the extended operational semantics.

8.6 Chapter Summary

Coordination of concurrent, asynchronous and potentially mobile activities in distributed systems is difficult to specify and implement. We introduced a hierarchical model for actor coordination where actors are grouped into *casts* and each cast is coordinated by a *director* actor. The hierarchical model guarantees scalability by hiding internal coordination within a cast from higher-level directors in the hierarchy. We presented an operational semantics for the model as a system of transitions between actor configurations. Internal transitions represent basic actor operations within a configuration, while external transitions model the openness of actor systems – i.e., interactions with other configurations.

Chapter 9

Discussion and Future Work

We discuss naming, remote communication, migration, and coordination issues in developing worldwide computing applications. We describe some limitations of our universal naming scheme. We comment on the design decisions behind our remote communication and migration strategies, and we compare the hierarchical model for coordination to reflective approaches. We conclude the chapter with potential future research directions in worldwide computing.

9.1 Naming

A major problem with our strategy for naming is its dependability on a unique name server, which entails the unreachability of potentially live actors, if their name server is down. The work on Domain Name Servers [64] has overcome the problem in the context of the Internet by providing a hierarchical organization of domain names, using a forwarding algorithm that updates names periodically, and having redundancy with multiple root servers in the domain name hierarchy.

Policies for handling temporary failures at naming server hosts need to be studied. Some possibilities include: to store messages to an unreachable actor in active `WWC` theaters for a reasonable amount of time, to retry periodically to contact the target actor's name host, and to notify the source of the message about partial and total failures. Another way is to create a representative to the actor that travels the `WWC` collecting messages to that actor. The last strategy avoids scattered undelivered messages and prevents from contact retrials from multiple points. The representative can also try to find traces of the target actor – temporary forwarding actors left in a departure theater after migration – that may lead to its current location.

Another major drawback of our naming strategy appears in the context of firewalls. If an actor migrates outside of an organizational firewall and the actor's naming server is inside the firewall, the naming server may not be possibly informed of such actor migration. Proxy actors forwarding messages into and out of the firewall may partially solve the reachability problems when security is not compromised. However, the naming service implementation needs to enable the utilization of proxies.

9.2 Remote Communication and Migration

Is location transparency a desired property?

Some researchers have argued that location transparency is an undesirable property in distributed systems: programmers need to be well informed about the location of their application objects before deciding on their interaction properties – in other words, different algorithms could be chosen depending on whether objects are co-located or not [50]. However, we believe that a programming model which provides both a high-level view of remote communication and mechanisms for migrating objects, is the best compromise; in such a model, not only do programmers have explicit control over the locality of interacting objects, they can also naturally express communication between them.

RMI vs RMSP: Asynchronous Communication Advantages

Often overlooked is an important fact: blocking communication, even though appropriate for centralized programs, is the wrong model for distributed systems. In a network with varying latencies and potential node failures, it does not make sense to invoke a method in a remote object and block the calling object indefinitely until getting an answer back. Note that synchronous, blocking communication between remote objects is the default behavior advocated by many distributed system architectures – e.g. Java's Remote Method Invocation [77].

It is more efficient to provide asynchronous communication primitives as the most basic interaction mechanism in distributed systems and use those primitives to build a number of synchronization mechanisms. For example, marshaling continuations [87] and futures [91] are two ways of enabling synchronization of activities in such loosely coupled, asynchronous communication-based distributed systems.

Java threads vs. `SALSA` actors migration

Some researchers have completely given up using remote communication in favor of migration [25]. However, the passive object model used in Java, limits migration in nontrivial ways:

- Java object migration requires ensuring that all threads concurrently and actively accessing an object should be in a coherent state before it can be migrated. This global synchronization requires the programmer's cooperation, inhibiting transparent object migration.
- The semantics for Java threads waiting on the lock for a migrating object need to be defined.
- Java objects holding references to the migrating object need to become aware of its migration. This is because there is no concept of globally unique references to objects.

On the other hand, an active object model naturally enables migration by completely encapsulating both the actor's state and the processing of the state. The encapsulation implies that migrating a universal actor is as simple as packing its state – including its mailbox of pending messages – and restarting the actor on a remote virtual machine – or *theater*. In our model, migration is performed in response to a message, thereby guaranteeing that the actor is not busy doing any other processing at the time of migration. Furthermore, Universal Actor Names persist beyond migration.

Fine-grained, coarse-grained and process migration

While universal actors go a long way towards improving fine-grained migration in distributed systems, we believe that a model for migration should consider coarser units of mobility, such as casts or cyborgs [6]. By moving groups of actors with high frequencies of internal communication and coordination, remote communication is decreased, thereby increasing overall system efficiency. Therefore, coordinated casts are a good compromise between fine-grained migration and process-based migration.

9.3 Coordination

Traditional models for coordination of actors require the run-time system to provide reflective capabilities in order to constrain certain types of messages from reaching their targets. A typical scenario is that of meta-level actors that are able to intercept message sending, receipt and processing for base-level actors.

The hierarchical model for actor coordination is less demanding of the run-time system in the sense that no special actor architecture is required for coordination of activities. Instead, actors are explicitly grouped in casts, which are in turn, coordinated by directors. Directors do not have more computational or communication capabilities than their coordinated counterparts. However, the model of communication ensures that directors are able to synchronize activities within their respective casts. Inter-cast communication can be performed with traditional asynchronous passive messages, or by using messengers.

A hierarchical structure of coordination may suggest possible bottlenecks at root directors in the coordination forest. However, because communications inside casts need not be coordinated by outside directors, the hierarchical model can be implemented efficiently. For directors above the common ancestor of two communicating actors, communications are internal transitions that need not interrupt outside actors.

It is possible to have actors that do not belong to any casts – such actors incur no unnecessary synchronization overhead at run-time. One strategy for avoiding potential bottlenecks – similar to the current one of replicating Internet domain name servers – is to co-locate directors on multiple nodes. The strategy would provide for limited cast state recovery in the case of failures in a cast's director.

Although the hierarchical model of actor coordination does not provide the sort of transparency and flexibility that reflective models can enable [51, 63, 74, 89, 91], we conjecture that despite its simplicity, it is powerful enough for many applications. Specifically, reflective models enable customization of all primitive operations in actor systems: actor creation, message sending, message reception, and internal state changes. The hierarchical model, on the other hand, only constraints message reception. Observe, however, that all communication and computation in actor systems happens as a result of receiving messages: therefore, restricting message reception is far more powerful than it may first appear to be.

9.4 Future Work

In order to realize an industrial-strength implementation of the World-Wide Computer, it is critical to deal with a few very important areas: security including authentication and safe remote code execution, support for multiple programming language and heterogeneous devices, and mechanisms for fault tolerance and reliability. In order to improve the programmability of worldwide computing applications, more research is needed in high-level programming language abstractions. Finally, it is fundamental to develop real-world applications of worldwide computing to evaluate their feasibility and practicality. These agenda items constitute the core of future worldwide computing research.

Security

Code mobility raises interesting questions regarding security, both for the host executing downloaded code, as well as for a mobile agent's state. Security restrictions are very important, for example to prevent malicious agents from starving nodes, or to prevent malicious nodes from starving agents, or to prevent malicious networks from damaging code en-route.

Security is important for services requiring user authentication, as well as to download and execute remote code in a safe manner, both for the hosting environment and for the code being executed. Different strategies include public key encryption technologies for strong user authentication with digital certificates; sandbox code execution limiting attacks from untrusted code (except denial of service attacks among others); signed code enabling less limited access to local computational and network resources; and proof-carrying code which can be verified against particular actions, such as reading and writing to local secondary memory.

Heterogeneity

Multiple programming language support is useful for enabling experimentation with new programming languages providing a range of diverse programming idioms and patterns. The universal naming scheme presented in the thesis is language independent. The remote communication and migration strategies are dependent on writing `SALSA` code, which in turn, uses the uniform object model present in Java. Programming language neutral protocols for remote communication and object and group migration are important topics of future work.

Another important area for future research includes better support for embedded systems and heterogeneous devices. We need dynamic, adaptable and extensible software components and network protocols. In particular, the Java virtual machine and TCP/IP impose more requirements than are actually necessary for small device communication and special multimedia contents such as audio and video streaming.

Failure handling

In the presence of failures or delayed communications due to the orders of magnitude of difference in network bandwidths worldwide, reliability is critical to the successful deployment of commercial applications on the World-Wide Computer. Algorithms such as replication, checkpointing and rollbacks, heart beating, and atomic commitment protocols are only partial solutions to the research problems in fault tolerant and real-time computing.

Programming language abstractions

A complete language implementation of the hierarchical model of actor coordination remains to be done. A complete implementation needs to include abstractions for creating director behaviors, which manipulate messages as first-class objects. Furthermore, a language should provide facilities enabling dynamic regrouping of actors into new hierarchies, and it should support the concept of *roles* – an actor can belong to multiple hierarchies, but switches its context from one to the other according to its current role. I believe that appropriate high-level linguistic constructs will enable the average programmer to write concurrent programs for distributed environments.

Applications

Last but not least, it is critical to build real-life large-scale worldwide computing applications. Applications in multiple domains – e.g., see Chapter 2 for a categorized group of examples – may help validate and improve the programming language abstractions and architectural run-time system provided by the `WWC` and `SALSA`.

A. SALSA 0.3.2 Grammar

We present the grammar used for SALSA v0.3.2. The SALSA grammar is based on the Java programming language's grammar [43]. The Java grammar used as a starting point, was taken from an example provided by the JavaCC lexical analyzer and parser generator.[1]

```
CompilationUnit   ::=  ( ModuleDeclaration )?
                       ( ImportDeclaration )*
                       ( TypeDeclaration )* <EOF>

ModuleDeclaration ::=  "module" Name ";"

ImportDeclaration ::=  "import" Name ( "." "*" )? ";"

TypeDeclaration   ::=  BehaviorDeclaration
                  |    InterfaceDeclaration
                  |    ";"

BehaviorDeclaration ::=  ( "abstract" | "final" | "public" )*
                         UnmodifiedBehaviorDeclaration

UnmodifiedBehaviorDeclaration ::=  "behavior" <IDENTIFIER>
                                   ( "extends" Name )?
                                   ( "implements" NameList )?
                                   BehaviorBody

BehaviorBody      ::=  "{" ( BehaviorBodyDeclaration )* "}"

NestedBehaviorDeclaration ::=  ( "static" | "abstract" | "final"
                               | "public" | "protected" | "private" )*
                               UnmodifiedBehaviorDeclaration

BehaviorBodyDeclaration ::=  Initializer
                        |    NestedBehaviorDeclaration
                        |    NestedInterfaceDeclaration
                        |    ConstructorDeclaration
                        |    MethodDeclaration
                        |    FieldDeclaration

MethodDeclarationLookahead ::=  ( "public" | "protected" | "private"
                                | "static" | "abstract" | "final"
                                | "native" | "synchronized" )*
                                ResultType <IDENTIFIER> "("

InterfaceDeclaration ::=  ( "abstract" | "public" )*
                          UnmodifiedInterfaceDeclaration
```

[1]Available at http://www.metamata.com/

```
NestedInterfaceDeclaration ::=  ( "static" | "abstract" | "final"
                           | "public" | "protected" | "private" )*
                           UnmodifiedInterfaceDeclaration

UnmodifiedInterfaceDeclaration ::=  "interface" <IDENTIFIER>
                                    ( "extends" NameList )? "{"
                                    ( InterfaceMemberDeclaration )* "}"

InterfaceMemberDeclaration ::=  NestedBehaviorDeclaration
                           |    NestedInterfaceDeclaration
                           |    MethodDeclaration
                           |    FieldDeclaration

FieldDeclaration ::=  ( "public" | "protected" | "private" | "static"
                      | "final" | "transient" | "volatile" )* Type
                      VariableDeclarator ( "," VariableDeclarator )* ";"

VariableDeclarator ::=  VariableDeclaratorId ( "=" VariableInitializer )?

VariableDeclaratorId ::=  <IDENTIFIER> ( "[" "]" )*

VariableInitializer ::=  ArrayInitializer
                    |    Expression

ArrayInitializer ::=  "{" ( VariableInitializer
                           ( "," VariableInitializer )* )? ( "," )? "}"

MethodDeclaration ::=  ( "public" | "protected" | "private" | "static"
                       | "abstract" | "final" | "native" | "synchronized" )*
                       ResultType MethodDeclarator
                       ( "throws" NameList )? ( Block | ";" )

MethodDeclarator ::=  <IDENTIFIER> FormalParameters ( "[" "]" )*

FormalParameters ::=  "(" ( FormalParameter ( "," FormalParameter )* )? ")"

FormalParameter ::=  ( "final" )? Type VariableDeclaratorId

ConstructorDeclaration ::=  ( "public" | "protected" | "private" )?
                            <IDENTIFIER> FormalParameters
                            ( "throws" NameList )? "{"
                            ( ExplicitConstructorInvocation )?
                            ( BlockStatement )* "}"

ExplicitConstructorInvocation ::=  "this" Arguments ";"
                              |    ( PrimaryExpression "." )?
                                   "super" Arguments ";"

Initializer ::=  ( "static" )? Block

Type ::=  ( PrimitiveType | Name ) ( "[" "]" )*

PrimitiveType ::=  "boolean"
              |    "char"
              |    "byte"
              |    "short"
              |    "int"
              |    "long"
              |    "float"
              |    "double"

ResultType ::=  "void"
           |    Type
```

```
Name ::=  <IDENTIFIER> ( "." <IDENTIFIER> )*

NameList ::=  Name ( "," Name )*

Expression ::= ConditionalExpression ( AssignmentOperator Expression )?

AssignmentOperator ::=  "="
                    |    "*="
                    |    "/="
                    |    "%="
                    |    "+="
                    |    "-="
                    |    "<<="
                    |    ">>="
                    |    ">>>="
                    |    "&="
                    |    "^="
                    |    "|="

ConditionalExpression ::=  ConditionalOrExpression
                           ( "?" Expression ":" ConditionalExpression )?

ConditionalOrExpression ::=  ConditionalAndExpression
                             ( "||" ConditionalAndExpression )*

ConditionalAndExpression ::=  InclusiveOrExpression
                              ( "&&" InclusiveOrExpression )*

InclusiveOrExpression ::=  ExclusiveOrExpression
                           ( "|" ExclusiveOrExpression )*

ExclusiveOrExpression ::=  AndExpression ( "^" AndExpression )*

AndExpression ::=  EqualityExpression ( "&" EqualityExpression )*

EqualityExpression ::=  InstanceOfExpression
                        ( ( "==" | "!=" ) InstanceOfExpression )*

InstanceOfExpression ::=  RelationalExpression ( "instanceof" Type )?

RelationalExpression ::=  ShiftExpression
                          ( ( "<" | ">" | "<=" | ">=" ) ShiftExpression )*

ShiftExpression ::=  AdditiveExpression
                     ( ( "<<" | ">>" | ">>>" ) AdditiveExpression )*

AdditiveExpression ::=  MultiplicativeExpression
                        ( ( "+" | "-" ) MultiplicativeExpression )*

MultiplicativeExpression ::=  UnaryExpression
                              ( ( "*" | "/" | "%" ) UnaryExpression )*

UnaryExpression ::=  ( "+" | "-" ) UnaryExpression
                |    PreIncrementExpression
                |    PreDecrementExpression
                |    UnaryExpressionNotPlusMinus

PreIncrementExpression ::=  "++" PrimaryExpression

PreDecrementExpression ::=  "--" PrimaryExpression

UnaryExpressionNotPlusMinus ::=  ( "~" | "!" ) UnaryExpression
                            |    CastExpression
                            |    PostfixExpression
```

```
CastLookahead ::=  "(" PrimitiveType
              |    "(" Name "[" "]"
              |    "(" Name ")" ( "~" | "!" | "(" | <IDENTIFIER>
                                | "this" | "super" | "new" | Literal )

PostfixExpression ::=  PrimaryExpression ( "++" | "--" )?

CastExpression ::=  "(" Type ")" UnaryExpression
               |    "(" Type ")" UnaryExpressionNotPlusMinus

PrimaryExpression ::=  PrimaryPrefix ( PrimarySuffix )*
                       ( MessageSendSuffix )?

PrimaryPrefix ::=  Literal
              |    "this"
              |    "super" "." <IDENTIFIER>
              |    "(" Expression ")"
              |    AllocationExpression
              |    ResultType "." "behavior"
              |    Name

PrimarySuffix ::=  "." "this"
              |    "." AllocationExpression
              |    "[" Expression "]"
              |    "." <IDENTIFIER>
              |    Arguments

MessageSendSuffix ::=  "<-" <IDENTIFIER> ( Arguments )?

Literal ::=  IntegerLiteral
        |    FloatingPointLiteral
        |    CharacterLiteral
        |    StringLiteral
        |    BooleanLiteral
        |    NullLiteral
        |    TokenLiteral

IntegerLiteral ::=  <INTEGER_LITERAL>

FloatingPointLiteral ::=  <FLOATING_POINT_LITERAL>

CharacterLiteral ::=  <CHARACTER_LITERAL>

StringLiteral ::=  <STRING_LITERAL>

BooleanLiteral ::=  "true"
               |    "false"

NullLiteral ::=  "null"

TokenLiteral ::=  "token"

Arguments ::=  "(" ( ArgumentList )? ")"

ArgumentList ::=  Expression ( "," Expression )*

AllocationExpression ::=  "new" PrimitiveType ArrayDimsAndInits
                     |    "new" Name ( ArrayDimsAndInits
                                     | Arguments ( BehaviorBody )? )

ArrayDimsAndInits ::=  ( "[" Expression "]" )+ ( "[" "]" )*
                  |    ( "[" "]" )+ ArrayInitializer
```

```
Statement ::=  Block
          |    EmptyStatement
          |    ContinuationExpression ";"
          |    StatementExpression ";"
          |    LabeledStatement
          |    SwitchStatement
          |    IfStatement
          |    WhileStatement
          |    DoStatement
          |    ForStatement
          |    BreakStatement
          |    ContinueStatement
          |    ReturnStatement
          |    ThrowStatement
          |    SynchronizedStatement
          |    TryStatement
          |    JoinStatement

Block ::=  "{" ( BlockStatement )* "}"

BlockStatement ::=  LocalVariableDeclaration ";"
               |    Statement
               |    UnmodifiedBehaviorDeclaration
               |    UnmodifiedInterfaceDeclaration

LocalVariableDeclaration ::=  ( "final" )? Type VariableDeclarator
                              ( "," VariableDeclarator )*

EmptyStatement ::=  ";"

StatementExpression ::=  PreIncrementExpression
                    |    PreDecrementExpression
                    |    PrimaryExpression
                         ( "++" | "--" | AssignmentOperator Expression )?

ContinuationExpression ::=  "currentContinuation"
                       |    PrimaryExpression ( "@" ContinuationExpression )?
                       |    JoinStatement

LabeledStatement ::=  <IDENTIFIER> ":" Statement

SwitchStatement ::=  "switch" "(" Expression ")"
                     "{" ( SwitchLabel ( BlockStatement )* )* "}"

SwitchLabel ::=  "case" Expression ":"
            |    "default" ":"

IfStatement ::=  "if" "(" Expression ")" Statement ( "else" Statement )?

WhileStatement ::=  "while" "(" Expression ")" Statement

DoStatement ::=  "do" Statement "while" "(" Expression ")" ";"

ForStatement ::=  "for" "("
                  ( ForInit )? ";" ( Expression )? ";" ( ForUpdate )?
                  ")" Statement

ForInit ::=  LocalVariableDeclaration
        |    StatementExpressionList

StatementExpressionList ::=  StatementExpression
                             ( "," StatementExpression )*

ForUpdate ::=  StatementExpressionList
```

```
BreakStatement ::=  "break" ( <IDENTIFIER> )? ";"

ContinueStatement ::=  "continue" ( <IDENTIFIER> )? ";"

ReturnStatement ::=  "return" ( Expression )? ";"

ThrowStatement ::=  "throw" Expression ";"

SynchronizedStatement ::=  "synchronized" "(" Expression ")" Block

TryStatement ::=  "try" Block
                  ( "catch" "(" FormalParameter ")" Block )*
                  ( "finally" Block )?

JoinStatement ::=  "join" "(" ContinuationExpressionList ")"
                   "@" ContinuationExpression

ContinuationExpressionList  ::=  ContinuationExpression
                                 ( "," ContinuationExpression )*
```

B. SALSA HelloWorld Example

The “Hello World” program in SALSA follows:

```
module helloworld;

behavior HelloWorld {
    void act(String arguments[]){
        standardOutput <- print("Hello ") @
        standardOutput <- println("World!");
    }
}
```

The Java code generated by SALSA for this example follows:

```
/* Generated by Salsa Compiler v 0.3.2 */

package helloworld;

import salsa.language.*;

public class HelloWorld extends UniversalActor{

        static StandardOutput standardOutput = new StandardOutput();
        static StandardInput standardInput = new StandardInput();
        static StandardError standardError = new StandardError();


        public void act(String arguments[]){

                {
                        Actor[] _targets = { standardOutput, standardOutput };
                        String[] _methodNames = { "print", "println" };
                        Object[][] _arguments = { {"Hello "}, {"World!"} };
                        int [] _tokens = { -1, -1 };
                        try {
                                standardOutput.send
                                        (Message.createMessage
                                        (this,
                                        _targets,
                                        _methodNames,
                                        _arguments,
                                        _tokens));
                        } catch (NoSuchMessageException _nsme){
                                _nsme.printStackTrace();
                        } catch (MessageCreationException _mce){
                                System.err.println("SALSA Internal Error:"+
                                        _mce.getMessage());
                        }
                }        }


        public static void main(String args[]){

                HelloWorld _helloWorld = new HelloWorld();

{
                        Actor[] _targets = { _helloWorld };
                        String[] _methodNames = { "act" };
                        Object[][] _arguments = { { args } };
                        int [] _tokens = { -1 };
                        try {
                                _helloWorld.send
                                        (Message.createMessage
                                        (null,
                                        _targets,
                                        _methodNames,
                                        _arguments,
                                        _tokens));
                        } catch (NoSuchMessageException _nsme){
                                _nsme.printStackTrace();
                        } catch (MessageCreationException _mce){
                                System.err.println("SALSA Internal Error:"+
                                        _mce.getMessage());
                        }
                }
        }
}
```

D. SALSA Acknowledged Multicast Example

The acknowledged multicast protocol example in SALSA follows:

```
module multicast;

behavior AcknowledgedMulticast{
    SimpleActor[] actors;
    long initialTime;
    long ack(){
        return System.currentTimeMillis() - initialTime;
    }
    void act(String[] args){
        int howMany = Integer.parseInt(args[0]);
        SimpleActor[] actors = new SimpleActor[howMany];
        for (int i = 0; i< howMany; i++){
            actors[i] = new SimpleActor();
        }
        initialTime = System.currentTimeMillis();
        standardOutput<-print("Time for acknowledged multicast: ") @
        join(actors<-m())@ack()@
        standardOutput<-print @
        standardOutput<-println(" ms.");
    }
}
```

The Java code generated by SALSA for this example follows:

```
/* Generated by Salsa Compiler v 0.3.2 */

package multicast;

import salsa.language.*;
```

```
public class AcknowledgedMulticast extends UniversalActor{

        static StandardOutput standardOutput = new StandardOutput();
        static StandardInput standardInput = new StandardInput();
        static StandardError standardError = new StandardError();

        SimpleActor[] actors;

        long initialTime;


        public long  ack(){


                return System.currentTimeMillis()-initialTime;
        }



        public void act(String[] args){

                int howMany=Integer.parseInt(args[0]);
                SimpleActor[] actors=new SimpleActor[howMany];

                for (int i=0;i<howMany;i++){

                actors[i]=new SimpleActor();
        }

                initialTime=System.currentTimeMillis();

                                MessageGroup _joinDirector1 = null;
                {
                        Actor[] _targets1 = actors;
                        String[] _methodNames1 = new String[actors.length];
                        int[] _tokenPositions1 = new int[actors.length];
                        Object [][] _arguments1 = new Object[actors.length][];
                        for (int i = 0; i< actors.length; i++){
                                _methodNames1[i] = "m";
                                _tokenPositions1[i] = -1;
                                _arguments1[i] = new Object[0];
                        }

                        try {
                        _joinDirector1 =
                                MessageGroup.createMessageGroup
                                (this,
                                        _targets1,
                                        _methodNames1,
                                        _arguments1,
                                        _tokenPositions1,
                                        null); // to set _continuation1
                        } catch (NoSuchMessageException _nsme){
                                _nsme.printStackTrace();
                        }
                        Message _continuation1 = null;
{
                        Actor[] _targets = { this, standardOutput, standardOutput };
                        String[] _methodNames = { "ack", "print", "println" };
                        Object[][] _arguments = { {}, {null}, {" ms."} };
                        int [] _tokens = { -1, 0, -1 };
                        try {
                                _continuation1 =
                                        (Message.createMessage
                                (this,
                                _targets,
                                _methodNames,
                                _arguments,
                                _tokens,
                                _joinDirector1.getReturnType(),
                                0));
                } catch (NoSuchMessageException _nsme){
                        _nsme.printStackTrace();
                } catch (MessageCreationException _mce){
                        System.err.println("SALSA Internal Error:"+
                                _mce.getMessage());
                }
        }
                _joinDirector1.setContinuation(_continuation1);
        }
{
                Actor[] _targets = { standardOutput, _joinDirector1 };
                String[] _methodNames = { "print", "process" };
                Object[][] _arguments = { {"Time for acknowledged multicast: "}, { } };
                int [] _tokens = { -1, -1 };
                try {
                        standardOutput.send
                                (Message.createMessage
                                (this,
                                _targets,
                                _methodNames,
                                _arguments,
                                _tokens));
                } catch (NoSuchMessageException _nsme){
                        _nsme.printStackTrace();
                } catch (MessageCreationException _mce){
                        System.err.println("SALSA Internal Error:"+
                                _mce.getMessage());
                }
        }       }


    public static void main(String args[]){

        AcknowledgedMulticast _acknowledgedMulticast = new AcknowledgedMulticast();

{
                Actor[] _targets = { _acknowledgedMulticast };
                String[] _methodNames = { "act" };
                Object[][] _arguments = { { args } };
                int [] _tokens = { -1 };
                try {
                        _acknowledgedMulticast.send
                                (Message.createMessage
                                (null,
                                _targets,
                                _methodNames,
                                _arguments,
                                _tokens));
                } catch (NoSuchMessageException _nsme){
                        _nsme.printStackTrace();
                } catch (MessageCreationException _mce){
                        System.err.println("SALSA Internal Error:"+
                                _mce.getMessage());
                }
        }
    }
}
```

References

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

[2] G. Agha, M. Astley, J. Sheikh, and C. Varela. Modular Heterogeneous System Development: A Critical Analysis of Java. In J. Antonio, editor, *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, pages 144–155. IEEE Computer Society, March 1998. http://osl.cs.uiuc.edu/Papers/HCW98.ps.

[3] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology*, May 1993.

[4] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III*, pages 345–363. International Federation of Information Processing Societies (IFIP), Elsevier Science Publisher, 1993.

[5] G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.*, chapter 12. MIT Press, 1999.

[6] G. Agha, N. Jamali, and C. Varela. Agent Naming and Coordination: Actor Based Models and Infrastructures. In A. Ominici, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*, chapter 9. Springer-Verlag, 2001. To appear.

[7] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[8] R. Allen and D. Garlan. Formalizing architectural connection. In *International Conference on Software Enginnering*, pages 71–80. IEEE Computer Society, 1994.

[9] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for Networks of Workstations: NOW. *IEEE Micro*, February 1995.

[10] M. Astley and G. A. Agha. Customization and composition of distributed objects: Middleware abstractions for policy management. In *Sixth International Symposium on the Foundations of Software Engineering (FSE-6, SIGSOFT '98)*, November 1998.

[11] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[12] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[13] T. Berners-Lee. World-Wide Computer. *Communications of the ACM.*, 40(2), Feb 1997.

[14] T. Berners-Lee, R. Cailliau, A. Luotenen, H. F. Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM.*, 37(8), Aug 1994.

[15] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. IETF Internet Draft Standard RFC 2396, August 1998. http://www.ietf.org/rfc/rfc2396.txt.

[16] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[17] J. Bosak. XML, Java, and the Future of the Web. *World Wide Web Journal*, XML Principles, Tools and Techniques, Oct. 1997. See http://www.w3.org/XML/ for more information.

[18] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml, February 1998. W3C Recommendation. Work in Progress.

[19] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[20] J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'89)*, pages 109–129. Cambridge University Press, 1989.

[21] Philip Buonadonna, Andrew Geweke, and David E. Culler. An implementation and analysis of the virtual interface architecture. In *Proceedings of Supercomputing '98*, Orlando, FL, November 1998.

[22] C. Callsen and G. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.

[23] N. Camiel, S. London, N. Nisan, and O. Regev. The POPCORN Project: Distributed Computation over the Internet in Java. In *Proceedings of the 6th International World Wide Web Conference*, April 1997.

[24] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[25] L. Cardelli and A.D. Gordon. Mobile ambients. In *Foundations of System Specification and Computational Structures*, LNCS 1378, pages 140–155. Springer Verlag, 1998.

[26] N. Carriero and D. Gelernter. *How to Write Parallel Programs.* MIT Press, 1990.

[27] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.

[28] K. M. Chandy, A. Rifkin, P. A. G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, New York, U.S.A., Aug 1996.

[29] A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs.* MIT Press, 1993.

[30] B. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, November 1997.

[31] P. Ciancarini and C. Hankin, editors. *First International Conference on Coordination Languages and Models (COORDINATION '96)*, number 1061 in LNCS, Berlin, 1996. Springer-Verlag.

[32] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, October 2000.

[33] W. Dally. *The J-Machine: System Support for Actors*, chapter 16, pages 369–408. MIT Press, Cambridge, Mass., 1990.

[34] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge.* MIT Press, 1995.

[35] M. Felleisen and D. Friedman. Control operators, the secd-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[36] J. Ferber and J. Briot. Design of a concurrent language for distributed artificial intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.

[37] I. Foster and C. Kesselman. The Globus Project: A Status Report. In J. Antonio, editor, *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, pages 4–18. IEEE Computer Society, March 1998.

[38] S. Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.

[39] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993*. Springer Verlag, 1993. LNCS 707.

[40] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant. Intra- and Inter-Object Coordination with MESSENGERS. In Ciancarini and Hankin [31].

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[42] D. Garlan and D. Le Metayer, editors. *Second International Conference on Coordination Languages and Models (COORDINATION '97)*, number 1282 in LNCS, Berlin, 1997. Springer-Verlag.

[43] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[44] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[45] C. Houck and G. Agha. HAL: A high-level actor language and its distributed implementation. In *Proceedings of the 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, August 1992.

[46] M. Hydari. Design of the 2K Naming Service. M.S. Thesis. Department of Computer Science. University of Illinois at Urbana-Champaign., February 1999.

[47] S. 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*, number 506 in LNCS. Springer-Verlag, 1991.

[48] S. Jagannathan. Communication-passing style for coordinated languages. In Garlan and Metayer [42], pages 131–149.

[49] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-grained mobility in the Emerald system. *TOCS*, 6(1):109–133, 1988.

[50] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, Sun Microsystems, November 1994.

[51] G. Kiczales, J. des Riviéres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[52] T. Kielmann. Designing a coordination model for open systems. In Ciancarini and Hankin [31], pages 267–284.

[53] W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1997.

[54] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, 1995.

[55] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[56] F. Kon, R. Campbell, M. Dennis Mickunas, and K. Nahrstedt. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1999.

[57] W. A. Kornfeld and C. Hewitt. The scientific community metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1), January 1981.

[58] D. Lange and M. Oshima. *Programming and Deploying Mobile Agents with Aglets.* Addison-Wesley, 1998.

[59] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison Wesley, 1997.

[60] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1997.

[61] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995. Special Issue on Software Architecture.

[62] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *ACM Conference Proceedings, Object Oriented Programming Languages, Systems and Applications*, pages 276–284, San Diego, CA, 1988.

[63] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of the European Conference on Object-Oriented Programming*, number 512 in LNCS, pages 231–250, 1991.

[64] P. Mockapetris. Domain Names - Concepts and Facilities. IETF Internet Draft Standard RFC 1034, November 1987. http://www.ietf.org/rfc/rfc1034.txt.

[65] Object Management Group. CORBA services: Common object services specification version 2. Technical report, Object Management Group, June 1997. http://www.omg.org/corba/.

[66] Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment, 1998. Work in Progress. http://osl.cs.uiuc.edu/foundry/.

[67] Open Systems Lab. SALSA: Simple Actor Language, System and Applications, 2000. Work in Progress. http://osl.cs.uiuc.edu/salsa/.

[68] R. Panwar and G. Agha. A methodology for programming scalable architectures. *Journal of Parallel and Distributed Computing*, 22(3):479–487, September 1994.

[69] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[70] S. Ren, G. A. Agha, and M. Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing*, 36:4–12, 1996.

[71] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Volume 2*, pages 21–36. Chapman & Hall, 1997.

[72] H. A. Simon. *The Sciences of the Artificial*, chapter The Architecture of Complexity: Hierarchic Systems. MIT Press, 3rd edition, 1996.

[73] D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996. TR UIUCDCS-R-96-1950.

[74] D. C. Sturman and G. Agha. A protocol description language for customizing failure semantics. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, October 1994.

[75] W.T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A New Major SETI Project based on project SERENDIP data and 100,000 Personal Computers. In *Proceedings of the Fifth International Conference on Bioastronomy*. Editrice Compositori, Bologna, Italy, 1997.

[76] Sun Microsystems Inc. – JavaSoft. Java Developers Kit (JDK) Documentation, 1995. Work in Progress. http://www.javasoft.com/products/jdk/.

[77] Sun Microsystems Inc. – JavaSoft. Remote Method Invocation Specification, 1996. Work in progress. http://www.javasoft.com/products/jdk/rmi/.

[78] Sun Microsystems Inc. – JavaSoft. JavaSpaces, 1998. Work in progress. http://www.javasoft.com/products/javaspaces/.

[79] C. Talcott. Interaction semantics for components of distributed systems. In *First IFIP workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '96)*, Paris, France, March 1996.

[80] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3), 1998.

[81] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in Carnot. *IEEE Parallel and Distributed Technology*, 1(2):16–20, May 1993.

[82] C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An object oriented concurrent system architecture. *Sigplan Notices*, 24(4):91–93, 1989.

[83] Amin Vahdat, Thomas Anderson, Michael Dahlin, David Culler, Eshwar Belani, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Computing*, July 1998.

[84] C. Varela. An Actor-Based Approach to World-Wide Computing. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 98), Doctoral Symposium, October 1998. http://osl.cs.uiuc.edu/~cvarela/oopsla98/.

[85] C. Varela and G. Agha. What after Java? From Objects to Actors. *Computer Networks and ISDN Systems: The International J. of Computer Telecommunications and Networking*, 30:573–577, April 1998. http://osl.cs.uiuc.edu/Papers/www7/.

[86] C. Varela and G. Agha. A Hierarchical Model for Coordination of Concurrent Activities. In P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models (COORDINATION '99)*, LNCS 1594, pages 166–182, Berlin, April 1999. Springer-Verlag. http://osl.cs.uiuc.edu/Papers/Coordination99.ps.

[87] C. Varela and G. Agha. Linguistic Support for Actors, First-Class Token-Passing Continuations and Join Continuations. Proceedings of the Midwest Society for Programming Languages and Systems Workshop, October 1999. http://osl.cs.uiuc.edu/~cvarela/mspls99/.

[88] J. Waldo. JINI Architecture Overview, 1998. Work in progress. http://www.javasoft.com/products/jini/.

[89] T. Watanabe and A. Yonezawa. An actor-based meta-level architecture for group-wide reflection. In J. W. deBakker, W.P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in LNCS, pages 405–425. Springer-Verlag, 1990.

[90] P. Wegner. Coordination as constrained interaction. In Ciancarini and Hankin [31], pages 28–33.

[91] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System.* MIT Press, Cambridge, Mass., 1990.

Vita

Carlos A. Varela, was born in Santafé de Bogotá, Colombia on December 28, 1972. He graduated from high school at age 14, and received a five-year fellowship to study Computer Science at the University of Los Andes, Colombia in January 1988. In August 1991, after getting a second place – among 168 two-person teams – in an international programming contest, Carlos moved to the University of Illinois at Urbana-Champaign to continue his undergraduate studies, obtaining a Bachelor of Science in Computer Science with Honors, in May 1992.

Subsequently, Carlos enrolled in the M.S./Ph.D. program, depositing his Master thesis on "Browsing Databases over the World-Wide Web" in October 1995 under the supervision of Professor Caroline Hayes. Carlos took a leave of absence from May 1995 to August 1996. During the leave, Carlos helped start up a software development company in Tokyo, Japan: International Systems Research. He also taught a graduate-level class on "World-Wide Web Programming" at the University of Los Andes, Colombia.

In August 1996, Carlos joined the Open Systems Laboratory, directed by Professor Gul Agha, where he did research on web-based computing, concurrent and distributed systems, and active object oriented programming languages. Carlos has over a dozen publications about his research and he has given invited plenary lectures, conference tutorials, paper presentations, and seminars in Switzerland, Japan, the U.S., Colombia, Canada, Germany, Australia, Netherlands, and France.

During his studies in Urbana-Champaign, Carlos was supported by the NSF-sponsored Research Experience for Undergraduates, by Research Assistantships from the National Center for Supercomputing Applications and the Department of Computer Science, and by a Beckman Institute Artificial Intelligence Fellowship. In August 2000, Carlos joined IBM's T.J. Watson Research Center as a Research Staff Member. In August 2001, Carlos will join Rensselaer Polytechnic Institute's Computer Science department as an Assistant Professor.