Contents

1 Worldwide Computing Middleware **1**

1.1 Middleware . . . 2
1.1.1 Asynchronous Communication . . . 2
1.1.2 Higher-level Services . . . 3
1.1.3 Virtual Machines . . . 4
1.1.4 Adaptability and Reflection . . . 4
1.2 Worldwide Computing . . . 5
1.2.1 Actor Model . . . 5
1.2.2 Language and Middleware Infrastructure . . . 6
1.2.3 Universal Actor Model and Implementation . . . 6
1.2.4 Middleware Services . . . 7
1.2.5 Universal Naming . . . 9
1.2.6 Remote Communication and Mobility . . . 10
1.2.7 Reflection . . . 12
1.3 Related Work . . . 16
1.3.1 Worldwide Computing . . . 16
1.3.2 Languages for Distributed and Mobile Computation . . . 16
1.3.3 Naming Middleware . . . 17
1.3.4 Remote Communication and Migration Middleware . . . 17
1.3.5 Adaptive and Reflective Middleware . . . 18
1.4 Research Issues and Summary . . . 18
1.5 Further Information . . . 19
1.6 Defining Terms . . . 19
1.7 Acknowledgments . . . 20

X-XXXX-XXXX-X/XX/$X.XX+$X.XX
© 2004 CRC Press LLC.

CONTENTS

1

Worldwide Computing Middleware

Gul A. Agha, *University of Illinois at Urbana-Champaign*

Carlos A. Varela, *Rensselaer Polytechnic Institute*

CONTENTS

1.1 Middleware 2
1.1.1 Asynchronous Communication 2
1.1.2 Higher-level Services 3
1.1.3 Virtual Machines 4
1.1.4 Adaptability and Reflection 4
1.2 Worldwide Computing 5
1.2.1 Actor Model 5
1.2.2 Language and Middleware Infrastructure 6
1.2.3 Universal Actor Model and Implementation 6
1.2.4 Middleware Services 7
1.2.5 Universal Naming 9
1.2.6 Remote Communication and Mobility 10
1.2.7 Reflection 12
1.3 Related Work 16
1.3.1 Worldwide Computing 16
1.3.2 Languages for Distributed and Mobile Computation 16
1.3.3 Naming Middleware 17
1.3.4 Remote Communication and Migration Middleware 17
1.3.5 Adaptive and Reflective Middleware 18
1.4 Research Issues and Summary 18
1.5 Further Information 19
1.6 Defining Terms 19
1.7 Acknowledgments 20

Abstract Widely distributed applications using Internet resources for computing require complex resource naming, discovery, coordination, and management policies. Software complexity is dealt with by developers with *middleware*, software layers dealing with distribution issues, such as naming, mobility, security, load balancing, and fault-tolerance. Middleware enables application developers to concentrate on their domain of expertise, reducing code and complexity by orders of

X-XXXX-XXXX-X/XX/$X.XX+$X.XX
© 2004 CRC Press LLC.

magnitude. We discuss different aspects of middleware infrastructures and we present the *World-Wide Computer*, our own worldwide computing infrastructure with naming, mobility, and coordination middleware layers, facilitating Internet-based distributed systems development.

1.1 Middleware

The wide variety of networks, devices, operating systems, and applications in today's computing environment create the need for abstraction layers to help developers manage the complexity of engineering distributed software. A number of models, tools and architectures have evolved to address the composition of objects into larger systems; some of the widely used *middleware* ranging in support from basic communication infrastructure to higher-level services includes CORBA [Object Management Group, 1997], DCOM [Brown and Kindel, 1996], Java RMI [Sun Microsystems Inc. – JavaSoft, 1996], and more recently Web Services [Curbera et al., 2002].

Middleware abstracts over operating systems, data representations and distribution issues thus enabling developers to program distributed heterogeneous systems largely as though they were programming a homogeneous environment. Many middleware systems accomplish this transparency by enabling heterogeneous objects to communicate with each other. Since the communication model for object-oriented systems is synchronous method invocation, middleware typically attempts to give programmers the illusion of local method invocation when they invoke remote objects. The middleware layers are in charge of low-level operations, such as marshalling and unmarshalling arguments to deal with heterogeneity, and managing separate threads for network communication.

Middleware toolkits provide compilers capable of creating code for the client (a.k.a. *stub*) and server (a.k.a. *skeleton*) components of objects providing remote application services, given their network-unaware implementation. Intermediate brokers help establish inter-object communication and provide higher-level services, such as naming, event and lifecycle services.

1.1.1 Asynchronous Communication

In order to give the illusion of local method invocation when invoking remote objects, a calling object is blocked—waiting for a return value from a remote procedure or method call. When the value returns the object resumes execution (see Figure 1.1). This style of communication is called *remote procedure call* or *RPC*. Users of middleware systems realized early on that network latencies make RPC much slower in nature than local communication and the consequence of its extensive use can be prohibitive in overall application performance. Transparency of communication may thus be a misleading design principle [Waldo et al., 1997].

Asynchronous (a.k.a. *event-based*) communication services enable objects to communicate in much more flexible ways [Agha, 1986]. For example, the result of invoking a method may be redirected to a third party or *customer*, rather than going back to the original method caller. Moreover, the target object need not synchronize with the sender in order to receive the message, thus retaining greater scheduling flexibility and reducing the possibilities of deadlocks.

Since asynchronous communication creates the need for intermediate buffers, higher-level communication mechanisms can be defined without significant additional overhead. For example, one can define a communication mechanism which enables objects to communicate with peers without knowing in advance the specific target for a given message. One such model is a shared memory abstraction used in Linda [Carriero and Gelernter, 1990]. Linda uses a shared tuple-space from which different processes (active objects) read and write. Another communication model is ActorSpaces [Callsen and Agha, 1994]; in ActorSpaces, actors use name patterns for directing messages to groups of objects (or a representative of a group). This enables secure communication that is transparent for applications. A more open but restrictive mechanism, called *publish-and-subscribe* [Banavar et al., 1999], has been used more recently. In publish-and-subscribe, set mem-

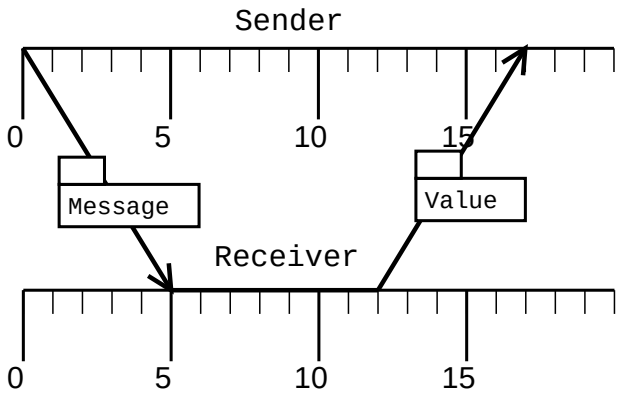

Figure 1.1: Synchronous communication semantics requires the sender of a message to block and wait until the receiver has processed the message and returned a value. In this example, the message processing takes only seven time units, while the network communication takes ten time units.

bership can be explicitly modified by application objects without pattern matching by an ActorSpace manager.

1.1.2 Higher-level Services

Beside communication, middleware systems provide high-level services to application objects. Such high-level services include, for example, object naming, lifecycle, concurrency, persistence, transactional behavior, replication, querying, and grouping [Object Management Group, 1997]. We describe these services to illustrate what middleware may be used to provide.

A *naming* service is in charge of providing object name uniqueness, allocation, resolution, and location transparency. Uniqueness is a critical condition for names so that objects can be uniquely found given their name. This is often accomplished using a name context. Object names should be object location-independent, so that objects can move preserving their name. A global naming context supports a universal naming space, in which context-free names are still unique. The implementation of a naming service can be centralized or distributed; distributed implementations are more fault-tolerant but create additional overhead.

A *life-cycle* service is in charge of creating new objects, activating them on demand, moving them, and disposing of them based on request patterns. Objects consume resources and therefore cannot be kept on systems forever—in particular, memory is often a scarce shared resource. Life-cycle services can create objects when new resources become available, can deactivate an object—storing its state temporarily in secondary memory—when the object is not being actively used and its resources are required by other objects or applications, and can also re-activate the object, migrate it, or can dispose (garbage collect) it if there are no more references to it.

A *concurrency* service provides limited forms of protection against multiple threads sharing resources by means of lock management. The service may enable application threads to request exclusive access to an object's state, read-only access, access to a potentially dirty state, and so on, depending on concurrency policies. Programming models, such as actors, provide higher-level support for concurrency management, preventing common errors, such as corrupted state or deadlocks, that can result from the use of a concurrency service,

A *persistence* or *externalization* service enables applications to store an object's state in secondary memory for future use, e.g., to provide limited support for transient server failures. This

service, even though high-level, can be used by other services, such as the life-cycle service described above.

A *transactional* service enables programming groups of operations with atomicity, consistency, isolation, and durability guarantees. Advanced transactional services may contain support for various forms of transactions, such as nested transactions, and long-lived transactions.

A *replication* service improves locality of access for objects by creating multiple copies at different locations. In case of objects with mutable state, a master replica is often used to ensure consistency with secondary replicas. In case of immutable objects, cloning in multiple servers is virtually unrestricted.

A *query* service enables manipulating databases with object interfaces using highly declarative languages such as SQL or OQL. Alternative query services may provide support for querying semi-structured data, such as XML repositories.

A *grouping* service supports creation of interrelated collections of objects, with different ordering and uniqueness properties, such as sets and lists. Different object collections provided by programming language libraries have similar functionality, albeit restricted to a specific programming language.

1.1.3 Virtual Machines

While CORBA's approach to heterogeneity is to specify interactions among object request brokers to deal with different data representations and object services, an alternative approach is to hide hardware and operating system heterogeneity under a uniform virtual machine layer (e.g., see [Lindholm and Yellin, 1997]). The virtual machine approach provides certain benefits but also has its limitations [Agha et al., 1998].

The main benefit of a virtual machine is platform independence, which enables safe remote code execution and dynamic program reconfiguration through bytecode verification and run-time object migration [Varela and Agha, 2001]. In principle, the virtual machine approach is programming language independent since it is possible to create bytecode from different high-level programming languages. In practice, however, bytecode verification and language safety features may prevent compiling arbitrary code in unsafe languages, such as C and C++, into Java bytecodes without using loopholes such as the Java native interface, which break the virtual machine abstraction.

The main limitations of the *pure* virtual machine approach are bytecode interpretation overhead in program execution and the inability to control heterogeneous resources as required in embedded and real-time systems. Research on just-in-time and dynamic compilation strategies has helped overcome the virtual machine bytecode execution performance limitations [Krall, 1998]. Open and extensible virtual machine specifications attempt to enable the development of portable real-time systems satisfying hard scheduling constraints and embedded systems with control loops for actuation [OVM Consortium, 2002; Schmidt et al., 1997; Bollela et al., 2000].

1.1.4 Adaptability and Reflection

Next-generation distributed systems will need to satisfy varying levels of quality of service, will require to dynamically adapt to different execution environments, will need to provide well-founded failure semantics, will have stringent security requirements, and will be assembled on-the-fly from heterogeneous components developed by multiple service providers.

Adaptive middleware [Agha, 2002] will likely prove to be a fundamental stepping stone to building next-generation distributed systems. Dynamic run-time customization can be supported by a reflective architecture. A reflective middleware provides a representation of its different components to the applications running on top of it. An application can *inspect* this representation and modify it. The modified services can be installed and immediately mirrored in further execution of the application (see Figure 1.2). We will describe the reflective model of actors in the next section.

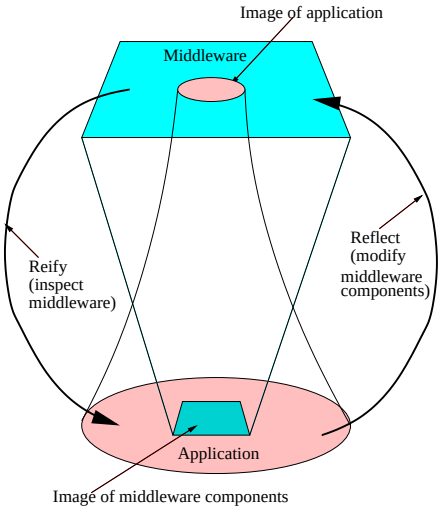

Figure 1.2: Using reflection, an application can inspect and modify middleware components.

1.2 Worldwide Computing

Worldwide computing research addresses problems in viewing dynamic networked distributed resources as a coordinated global computing infrastructure. We have developed a specific actor-based worldwide computing infrastructure, the *World-Wide Computer* (WWC), that provides naming, mobility, and coordination middleware, to facilitate building widely distributed computing systems over the Internet.

Worldwide computing applications view the Internet as an execution environment. Since Internet nodes can join and leave a computation at run-time, the middleware infrastructure needs to provide dynamic reconfiguration capabilities: in other words, an application needs to be able to decompose and re-compose itself while running – potentially moving its sub-components to different network locations.

1.2.1 Actor Model

In traditional object-oriented systems, the interrelationship between objects—as state containers—and threads—as process abstractions—is highly intertwined. For example, in Java [Gosling et al., 1996], multiple threads may be concurrently accessing an object creating the potential for state corruption. A class can declare all its member variables to be `private`, and all its methods to be `synchronized` to prevent state corruption because of multiple concurrent thread accesses. However, this practice is inefficient and creates potential for deadlocks (see e.g., citepvarela-agha-www7-98). Other languages, such as C++, do not even have a concurrency model built-in, requiring developers to use thread libraries.

Such passive object computation models severely limit applications reconfigurability. Moving an object in a running application to a different computer requires guaranteeing that active threads within the object remain consistent after object migration. Moving a thread in a running application to a different computer requires very complex invocation stack migration ensuring that references remain consistent, and that any locks held by the thread are safely released.

The actor model of computation is a more natural approach to application reconfigurability, since an actor is an autonomous unit abstracting over state encapsulation and state processing. Actors can only communicate through asynchronous message passing and do not share any memory. As a consequence, actors provide a very natural unit of mobility and application reconfigurability. Actors also provide a unit of concurrency by processing one message at a time. Migrating an actor is then as simple as migrating its encapsulated state along with any buffered un-processed messages.

Reconfiguring an application composed of multiple actors is as simple as migrating a subset of the actors to another computer. Since communication is asynchronous and buffered, the application semantics remains the same as long as actor names can be guaranteed to be unique across the Internet. The universal actor model is an extension to the actor model, providing actors with a specific structure for universal names.

1.2.2 Language and Middleware Infrastructure

Several libraries which support the Actor model of computation have been implemented in different object-oriented languages. Three examples of these are the Actor Foundry [Open Systems Lab, 1998], Actalk [Briot, 1989] and Broadway [Sturman, 1996]. Such libraries essentially provide high-level middleware services, such as universal naming, communication, scheduling, and migration.

An alternate is to support distributed objects in a sufficiently rich language which enables coordination across networks. Several actor languages have also been proposed and implemented to date, including ABCL [Yonezawa, 1990], Concurrent Aggregates [Chien, 1993], Rosette [Tomlinson et al., 1989], and Thal [Kim, 1997]. An actor language can also be used to provide interoperability between different object systems; this is accomplished by wrapping traditional objects in actors and using the actor system to provide the necessary services. There are several advantages associated with directly using an actor programming language, as compared to using a library to support actors:

- **Semantic constraints:** Certain semantic properties can be guaranteed at the language level. For example, an important property is to provide complete encapsulation of data and processing within an actor. Ensuring there is no shared memory or multiple active threads, within an otherwise passive object, is very important to guarantee safety and efficient actor migration.
- **API evolution:** 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. Furthermore, evolutionary changes to an actor API need not affect actor code.
- **Programmability:** Using an actor language improves the readability of programs developed. Often writing actor programs using a framework involves using language level features (e.g., method invocation) to simulate primitive actor operations (e.g., actor creation or message sending). The need for a permanent semantic translation, unnatural for programmers, is a very common source of errors.

Our experience suggests that an active object oriented programming language—one providing encapsulation of state and a thread manipulating that state—is more appropriate than a passive object oriented programming language (even with an actor library) for implementing concurrent and distributed systems to be executed on the Internet.

1.2.3 Universal Actor Model and Implementation

The universal actor model extends the actor model [Agha, 1986] by providing actors with universal names, location awareness, remote communication, migration, and limited coordination capabilities [Varela, 2001].

	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

Table 1.1: Comparison of `WWW` and `WWC` concepts.

We describe *Simple Actor Language System and Architecture* (SALSA), an actor language and system that has been developed to provide support for worldwide computing on the Internet. Associated with SALSA is a runtime system which provides the middleware necessary services [Varela and Agha, 2001]. By using SALSA, developers can program at a higher level of abstraction.

SALSA programs are compiled into Java bytecode and can be executed standalone or on the World-Wide Computer infrastructure. SALSA programs are compiled into Java bytecode to take advantage of Java virtual machine implementations in most existing operating systems and hardware platforms. SALSA-generated Java programs use middleware libraries implementing protocols for universal actor naming, mobility, and coordination in the World-Wide Computer. Table 1.1 relates different concepts in the World-Wide Web to analogous concepts in the World-Wide Computer.

1.2.4 Middleware Services

Services implemented in middleware to support the execution of SALSA programs over the World-Wide Computer can be divided into two groups: *core services* and *higher-level services* as depicted in Figure 1.3.

1.2.4.1 Core Services

An *actor creation* service supports the creation of new actors (which comprise an initial state, a thread of execution, and a mailbox) with specific behaviors. Every created actor is in a continuous loop, sequentially getting messages from its mailbox, and processing them. Concurrency is a consequence of the fact that multiple actors in a given program may execute in parallel.

A *transport* service supports reliable delivery of data from a computer to another. The transport service is used by the higher-level remote communication, migration, and naming services.

A *persistence* service supports saving an actor's state and mailbox into secondary memory, either for checkpoints, fault-tolerance or improved resource (memory, processing, power) consumption.

1.2.4.2 Higher-Level Services

A *messaging* service supports reliable asynchronous message delivery between peer actors. A message in SALSA is modelled as a potential Java method invocation. The message along with optional arguments is placed in the target actors mailbox for future processing. A message sending expression returns immediately after delivering the message—not after the message is processed as in traditional method invocation semantics. Section 1.2.6 discusses this service in more detail.

A *naming* service supports universal actor naming. A *universal naming* model enables developers to uniquely name resources worldwide in a location-independent manner. Location independence is important when resources are mobile. Section 1.2.5 describes the universal actor naming model and protocol used by this service.

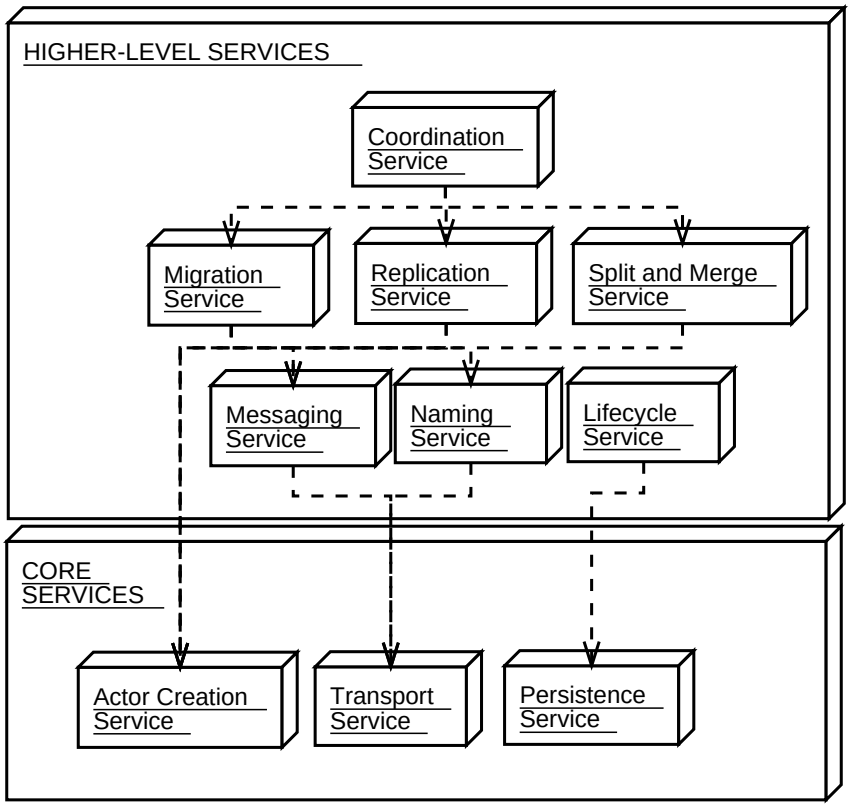

Figure 1.3: Core services include actor creation, transportation, and persistence. Higher-level services include actor communication, naming, migration, and coordination.

A *life-cycle* service can deactivate an actor into persistent storage for improved resource utilization. It can also reactivate the actor on demand. Aditionally, it performs distributed garbage collection.

A *migration* service enables actor mobility preserving universal actor names and updating universal actor locations. Migration can be triggered by the programmer using SALSA messages, or it may be triggered by higher-level services such as load balancing and coordination. Section 1.2.6 provides more details on the actor migration service.

A *replication* service can be used to improve locality and access times for actors with immutable state. It can also be used for improving concurrency in parallel computations when additional processing resources become available.

A *split-and-merge* service can be used to fine-tune the granularity of homogeneous actors doing parallel computations to improve overall system throughput.

Coordination services are meant to provide the highest-level of services to applications, including those requiring reflection and adaptation. For example, a load balancing service can profile resource utilization and automatically trigger actor migration, replication, and splitting and merging behaviors for coordinated actors [Desell et al., 2004].

1.2.5 Universal Naming

Since universal actors are mobile—their location can change arbitrarily—it is critical to provide a universal naming system that guarantees that references remain consistent upon migration.

Universal Actor Names (UAN) are identifiers that represent an actor during its life-time in a location-independent manner. An actor's UAN is mapped by a naming service into a *Universal Actor Locator* (UAL), which provides access to an actor in a specific location. When an actor migrates, its UAN remains the same, and the mapping to a new locator is updated in the naming system. Since universal actors refer to their peers by their name, references remain consistent upon migration.

1.2.5.1 Universal Actor Names

A *Universal Actor Names* (UAN) refers to an actor during its life-time in a location-independent manner. The main requirements on universal actor names are location-independence, worldwide uniqueness, human readability, and scalability.

We use the Internet's Domain Name System (DNS) [Mockapetris, 1987] to hierarchically guarantee name uniqueness over the Internet in a scalable manner. More specifically, we use Uniform Resource Identifiers (URI) [Berners-Lee et al., 1998] to represent Universal Actor Names. This approach does not require actor names to have a specific naming context, since we build on unique Internet domain names.

The universal actor name for a sample address book actor is:

```
uan://wwc.yp.com/~smith/addressbook/
```

The protocol component in the name is `uan`. The DNS server name represents an actor's *home*. An optional port number represents the listening port of the naming service—by default 3030. The remaining name component, the *relative UAN*, is managed locally at the home name server to guarantee uniqueness.

1.2.5.2 Universal Actor Locators

An actor's UAN is mapped by a naming service into a *Universal Actor Locator* (UAL), which provides access to an actor in a specific location. For simplicity and consistency, we also use URIs to represent UALs. Two universal actor locators for the address book actor above are:

```
rmsp://wwc.yp.com/~smith/addressbook/
```

and

```
rmsp://smith.pda.com:4040/addressbook/
```

The protocol component in the locator is `rmsp`, which stands for the *Remote Message Sending Protocol*. The optional port number represents the listening port of the actor's current *theater*, or single-node run-time system—by default 4040. The remaining locator component, the *relative UAL* is managed locally at the theater to guarantee uniqueness.

While the address book actor can migrate from the user's laptop to her personal digital assistant (PDA), or cellular phone; the actor's UAN remains the same, and only the actor's locator changes. The naming service is in charge of keeping track of the actor's current locator.

1.2.5.3 Universal Actor Naming Protocol

When an actor migrates, its UAN remains the same, and the mapping to a new locator is updated in the naming system. The *Universal Actor Naming Protocol* (UANP) defines the communication between an actor's theater and an actor's home, during its life-time: creation and initial binding, migration, and garbage collection.

UANP is a text-based protocol resembling HTTP with methods to create a UAN to UAL mapping, to retrieve a UAL given the UAN, to update a UAN's UAL, and to delete the mapping from the naming system. 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

A distributed naming service implementation can use consistent hashing to replicate UAN to UAL mappings in a ring of hosts and provide a scalable and reasonable level of fault-tolerance. The logarithmic lookup time can further be reduced to a constant lookup time in most cases [Tolman, 2003].

1.2.5.4 Universal Naming in SALSA

The SALSA pseudo-code for a sample address book management program is shown in Figure 1.4. The program creates an address book manager and binds it to a UAN. After the program successfully terminates, the actor can be remotely accessed by its name.

1.2.6 Remote Communication and Mobility

The underlying middleware used by SALSA-generated Java code uses an extended version of Java object serialization for both remote communication and actor migration.

1.2.6.1 Remote Message Sending Protocol

Universal actors communicate with peers by passing messages asynchronously. When actors are executing in remote theaters, an Internet-based protocol is used for such communication—the *Remote Message Sending Protocol* (RMSP).

```
behavior AddressBook {

      String getEmail(String name){...}

      void act(String[] args){
            AddressBook addressBook = new AddressBook();
            try {
                  addressbook<-bind("uan://wwc.yp.com/~smith/addressbook/",
                        "rmsp://wwc.yp.com/~smith/addressbook/");
            } catch (Exception e){
                  standardOutput<-println(e);
            }
      }
}
```

Figure 1.4: Universal Actor Name and Locator binding in SALSA.

RMSP is a protocol implemented as an extension to Java object serialization. An actor's theater contains an RMSP server that listens for incoming messages from actors in remote theaters. Such messages are targeted to an actor with a locator local to the receiving theater. The theater keeps track of hosted actors and their locators, so that incoming messages can be properly passed to the target actor.

Messages are represented as potential method invocations along with an optional continuation. Arguments are passed by value for primitive types, and Java *serializable* objects; and by reference for universal actors.

In addition to passing information by using object serialization, RMSP updates universal actor references so that most efficient access can be performed for peer actors that are hosted in the target theater. Interested readers are refered to [Varela, 2001] for details.

1.2.6.2 Universal Actor Migration Protocol

Universal actors can move from a theater to another by processing a migration request message. A universal actor implementation contains a thread of execution and encapsulated state. In response to a migration request, a universal actor's state—including buffered unprocessed messages—is serialized, and a new thread of execution is started at the receiving theater.

We reuse RMSP for universal actor migration. The theater's RMSP server accepts incoming objects and acts upon them based on their type. Currently there are two types: `Message`, for asynchronous message passing; and `UniversalActor`, for actor migration. An actor migration involves several steps: updating the naming service to reflect the actor's new locator; serializing the actor's state to the new theater; updating the actor's references to local resources—which we call *environment actors*; updating the theaters' meta-data; and restarting the actor's thread in the new location.

To avoid potential race conditions which arise because of messages en-route to migrating actors, we do not release the lock protecting the migrating actor's mailbox until the actor has completed the migration. Once the actor has acknowledged migration completion, the actor mailbox lock is released, and messages get re-routed as appropriate by the run-time system, with the assurance that the actor is already ready to receive them.

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

AddressBook addressBook = new
               AddressBook("uan://wwc.yp.com/~smith/addressbook/");
addressBook<-getEmail("David") @
  standardOutput<-println(token);
```

Figure 1.5: Remote Communication in SALSA.

```
//
//  Migrating an address book to a remote theater:
//

AddressBook addressBook =
               new AddressBook("uan://wwc.yp.com/~smith/addressbook/");
a<-migrate("rmsp://smith.pda.com/addressbook/");
```

Figure 1.6: Actor Migration in SALSA.

1.2.6.3 Remote Communication and Actor Migration in SALSA

The code for sending a `getEmail()` message to the address book manager created in the previous section, is shown in Figure 1.5. The code gets a reference to the address book manager using its UAN, sends a message requesting a user's email address, and prints it out in the console.

SALSA also enables migrating an actor to a given theater. For example, the code for migrating the address book manager above is shown in Figure 1.6. In this case, the actor is migrated to a new location given by a UAL (`rmsp://smith.pda.com/addressbook/`).

1.2.7 Reflection

We use an explicit representation of the implementation of actors to provide a mechanism for customization of middleware Astley and Agha [1998]. Such a representation is called the *meta-level* system. Thus, in a reflective architecture, a system is composed of two kinds of actors—base-level (application) actors and meta-level actors or *meta-actors*. Meta-actors are part of the middleware that manages system resources and implements the base-actor's runtime semantics.

In implementation terms, actors do not directly interact with one another. Instead, actors make *system calls* to the middleware—these calls correspond to invocations of methods in meta-actors. A system call by an actor is always blocking and the actor waits till the call returns. A meta-actor executes the method that is invoked by another actor and returns control on completion of the execution.

A meta-actor is capable of customizing the behavior of another actor by executing the method invoked by it. Multiple customizations may be applied to a single actor by building a *meta-level stack* (see Figure 1.7). A meta-level stack consists of a single base-actor and a stack of meta-actors on top of it, where each meta-actor customizes the actor which is just below it in the stack. Messages received by an actor in a meta-level stack are always delegated to the top of the stack so that the meta-actor always controls the delivery of messages to its base-actor. Similarly message sent by an actor passes through all the meta-actors in the stack.

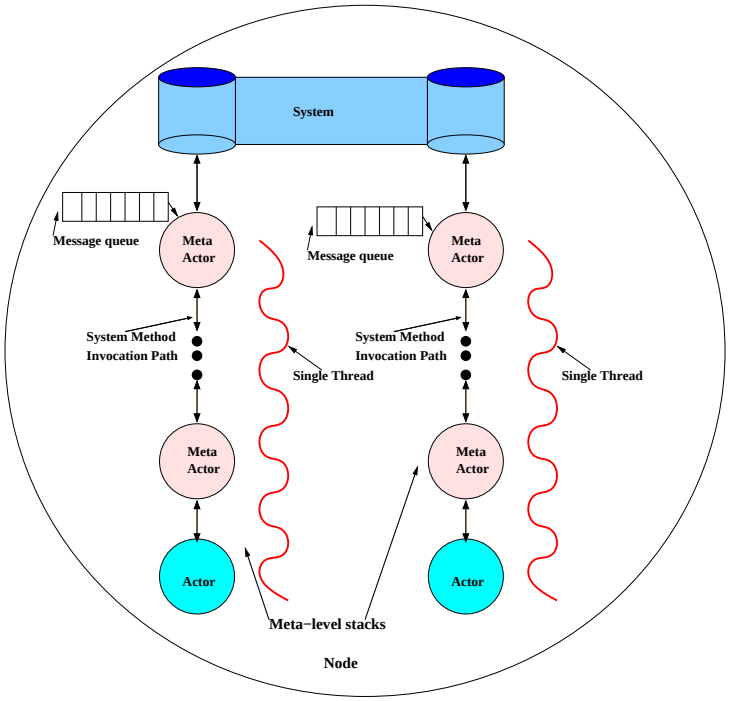

Figure 1.7: Meta-level stacks in an embedded node.

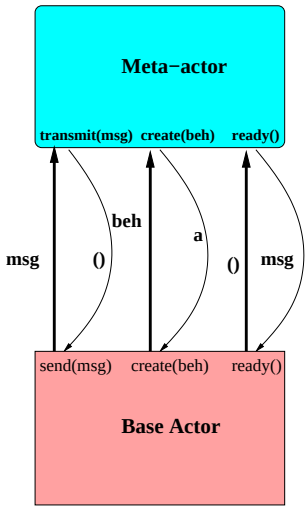

Figure 1.8: Interaction between meta-actor and base actor.

Conceptually, we can translate actor operations into method calls to a meta-actor. These operations are:

- **transmit(msg):** This method is invoked when an actor sends a message `msg`. If the actor has a meta-actor on its top it calls the transmit method of the meta-actor and wait for its return. The method returns without any value. Otherwise, if the actor is not customized by a meta-actor, it passes the message to the system for sending.

- **create(beh):** This method is invoked when the actor wants to create another actor with a given behavior `beh`. If there is a meta-actor on top of the actor, it calls the create method of the meta-actor and waits for it return. The method returns the address `a` of the new actor. Otherwise, the actor passes the create request to the system.

- **ready:** The ready method is invoked when an actor has completed processing the current message and is waiting for another message. If the actor has a meta-actor on its top it calls the ready method of the meta-actor and waits for its return. The method returns a message to the base-actor. Otherwise, the actor picks up a message from its mail queue and processes it. Notice, there is single mail-queue for a given meta-level stack.

Every meta-actor has a default implementation of the three system methods. They are given below:

- **transmit(msg):** If there is a meta-actor on its top, it calls `transmit(msg)` method of that meta-actor and waits for it to return. Otherwise, it asks the system to send the message to the target and returns.

- **ready():** If there is a meta-actor on top of it, it calls `ready()` method of that meta-actor and waits for it to return a message. Otherwise, the actor, by definition located at the top of the meta-level stack, dequeues a message from the mail queue. After getting the message, the actor returns the message to the base actor.

- **create(beh):** If the actor has a meta-actor at its top, it calls `create(beh)` method of that meta-actor and waits for the actor to return with an actor address. Otherwise, the actor passes the create request to the system and waits till it gets an actor address from the system. After receiving new actor address, the actor returns it to the base actor.

As an example of how we may customize actors under this model, consider the encryption of messages between a pair of actors. Figure 1.9 gives pseudo-code for a pair of meta-actors which may be installed on each endpoint. The `Encrypt` meta-actor implements the `transmit` method which is called by the base-actor while sending a message. Within `transmit`, a message is encrypted before it is sent to its target. The `Decrypt` meta-actor implements the `ready` method which is called when the base actor is ready to process a message. Method `ready` decrypts the message before returning the message to the base-actor.

Thus the abstraction of the middleware in terms of meta-actors gives the power of dynamic customization. Meta-actors can be installed or pulled out dynamically (see Figure 1.10). The installation and removal of meta-actors by the application itself makes it capable of customizing the middleware. Applications can thus control adaptive middleware by reification and reflection. This means that while middleware can provide default policies for resource profiling, secure communication, load balancing, and coordination; applications can override default mechanisms and provide their own resource management policies.

```
actor Encrypt() {
    actor server;

    // Instantiated with name of server
    init (actor S) {
        server := S;
    }

    // Encrypt outgoing messages if they
    // are targeted to the server
    method transmit(Msg msg) {
        actor target = msg.dest;
        if (target == server)
            target ← encrypt(msg);
        else
            target ← msg;
        continue();
    }
}
actor Decrypt() {
    // Decrypt incoming messages targeted for
    // base actor (if necessary)
    method rcv(Msg msg) {
        if (encrypted(msg))
            deliver(decrypt(msg));
        else
            deliver(msg);
    }
}
```

Figure 1.9: The `Encrypt` policy actor intercepts **transmit** signals and encrypts outgoing messages. The `Decrypt` policy actor intercepts messages targeted for the server (i.e., the **rcv** method) and, if necessary, decrypts an incoming message before delivering it.

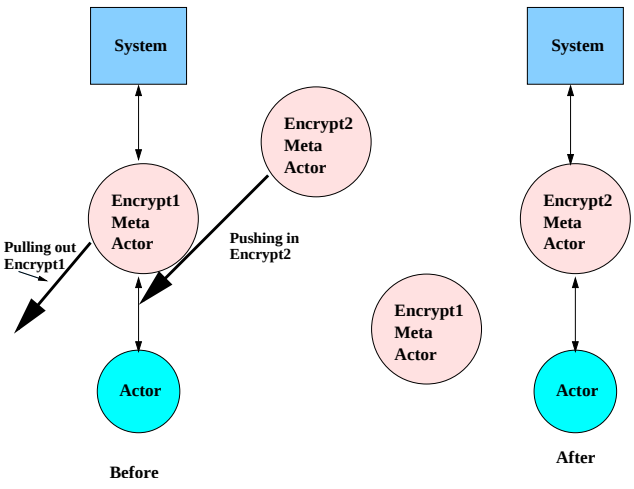

Figure 1.10: Dynamic customization is enabled by pulling out and pushing in new meta-actors for the implementation of the encryption algorithm.

1.3 Related Work

1.3.1 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 [Anderson et al., 1995], and Berkeley's Millennium project is exploiting a hierarchical cluster structure to provide distributed computing on a "campus-wide" scale [Buonadonna et al., 1998]. The Globus project seeks to enable the construction of larger *computational grids* [Foster and Kesselman, 1998]. 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 [Chandy et al., 1996]. WebOS seeks to provide operating system services, such as client authentication, naming, and persistent storage, to wide area applications [Vahdat et al., 1998]. 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 [Kon et al., 1999].

Most approaches to worldwide computing are either operating system-dependent, application-dependent, or require a set of computers under the administrative control of its users.

In application-dependent worldwide computing, a domain-specific program is downloaded by users to provide their computing power to the data analysis task at hand. For example, the Search for Extra Terrestrial Intelligence (SETI) project at Berkeley uses idle computers from participants around the world to analyze space data in search of patterns and potential "signals" from intelligent life in outer space [Sullivan et al., 1997]. Interesting extensions include: a separation between office (SETI@Work) and home computers (SETI@Home) in the global computing task; and another project started at Stanford to understand how proteins fold (Folding@Home).

In other middleware approaches to worldwide computing, such as the Grid [Foster and Kesselman, 1998], it is required of users to have accounts (logins and passwords) in the systems that take part in the global computation. While this approach may work well for certain groups of users with multiple supercomputer accounts, it does not properly scale to Internet computing. On the Internet, there is a wide variety of hardware architectures, operating systems, and domains of administrative control.

The WWC attempts to provide middleware for distributed system developers to use a heterogeneous network of Internet-connected computers, under multiple administrative domains, in an application-independent manner.

1.3.2 Languages for Distributed and Mobile Computation

The ABCL family of languages has been developed by Yonezawa's research group [Yonezawa, 1990] to explore an object-oriented concurrent model of computation, based on Actors. ABCL has been developed in Common Lisp. One significant 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's more general continuation passing style can be used to implement *now* and *future* message passing.

THAL, an extension to HAL (High-level Actor Language) [Houck and Agha, 1992], was developed by Kim [Kim, 1997] 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 shown that with proper compilation techniques, parallel actor programs can run as efficiently as their

non-actor counterparts. Future research includes studying optimizations of SALSA actor programs, in particular, the actor model's data encapsulation enables eliminating most of Java's synchronization overhead.

Gray et al. present a very complete survey of mobile agent systems [Gray et al., 2000] categorized by the programming languages they support. Agent systems supporting multiple programming languages include: Ara, D'Agents, and Tacoma. Java-based systems include Aglets [Lange and Oshima, 1998], Concordia, Jumping Beans, and Voyager. Other systems supporting a non-Java single programming language include: Messengers, Obliq, Telescript, and Nomadic Pict [Wojciechowski and Sewell, 1999].

Obliq [Cardelli, 1995] 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.

Emerald [Jul et al., 1988], one of the first systems supporting fine-grained migration, includes 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.

1.3.3 Naming Middleware

ActorSpaces [Callsen and Agha, 1994] 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 [1998], 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 [Vahdat et al., 1998]. 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.

The 2K distributed operating system [Kon et al., 1999] builds upon and enhances CORBA's naming system [Hydari, 1999]. 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.

1.3.4 Remote Communication and Migration Middleware

The Common Object Request Broker Architecture (CORBA) [Object Management Group, 1997] has been designed with the purpose of handling heterogeneity in object-based distributed systems.

Sun's JINI [Waldo, 1998] 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) [Sun Microsystems Inc. – JavaSoft, 1996]. JavaSpaces [Sun Microsystems Inc. – JavaSoft, 1998] is another important component of JINI that uses a Linda-like [Carriero and Gelernter, 1990] model to share, coordinate, and communicate tasks in Jini-based distributed systems.

Java [Gosling et al., 1996] 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 [Lange and Oshima, 1998] is a more recent framework for the development of Internet agents which can migrate, preserving their state.

1.3.5 Adaptive and Reflective Middleware

Adaptive middleware in distributed systems has been studied by several researchers. Gul Agha et al. [1993] have introduced meta-actors to implement different interaction services such as fault tolerance, security, and synchronization. Fabio Kon et al. [2002] have presented a model of reflective middleware that allows dynamic inspection and modification of the execution semantics of running applications as a response to changing resources in a distributed environment in order to improve performance. Research has also been done at the level of middleware security. Venkatasubramanian [2002] discussed the safe composibility of reflective middleware services to ensure the trustworthiness of systems. The Two-Level Actor Machine (TLAM) model [Venkatasubramanian and Talcott, 1995] is a reasoning framework for specifying and proving properties about interactions of middleware components.

Varela and Agha [1999] introduced a hierarchical model that groups actors into *casts* coordinated by *directors*. The cast directors are meta-level actors that filter incoming messages to group members. The hierarchical model is more general than the stack model presented in this paper in that a single meta-actor can coordinate more than one base actor. The hierarchical model does not restrict actor creation or message sending, only message reception. This restriction is valid in that actors are reactive entities, i.e., all computation proceeds in response to messages.

1.4 Research Issues and Summary

Worldwide computing is the coordinated use of large scale network-connected resources for global computations and human collaboration. In this chapter, we defined the universal actor model and linguistic abstractions for coordination of globally distributed actors. We also described middleware services, such as naming and mobility, that are needed to implement actors on the World-Wide Computer.

There are still several open research problems before worldwide computing will become more common. These problems include:

- A security model that enables participants to safely volunteer their resources without risking loss of their information, and that enables worldwide computing users to trust the validity of global computations and the privacy of their data.

- A fine-grained resource management framework, which enables participants to control and be compensated for the computing resources they provide. For example, a participant may wish to volunteer only a fixed percentage of their computing, communication, and storage capabilities.

- Higher-level coordination abstractions to facilitate the programming of worldwide computing systems in a way that is largely transparent to systems issues such as load balancing and fault-tolerance.

A worldwide computing infrastructure including a universal actor programming language (SALSA) and several middleware services at different stages of development (WWC/IO) is freely available at http://www.cs.rpi.edu/wwc/salsa/. This infrastructure can be used as a modular starting point for implementing additional middleware services, or for developing distributed computing applications to be executed over the Internet or over Grid-like environments.

Worldwide computing will enable new classes of applications, where the dividing line between the physical world of communicating devices and the logical world of computing actors will get thinner and thinner. Physical devices are becoming more powerful and inter-connected and logical actors are becoming more mobile and autonomous.

Mobility of devices and actors with different granularities in heterogeneous networks will induce ad-hoc emergent coordination and interaction behavior patterns. These self-coordinated actor systems will enable efficient use of scarce resources, and will clear the way for the creation of complex computing systems at very large scales. The availability of virtually unlimited storage, communication, and data processing capabilities will open a new door for applications in many domains, including science, education, business, government, and technology.

1.5 Further Information

Worldwide computing research is published in several computer science journals, including *ACM Transactions of Internet Technologies* and *IEEE Internet Computing*. There are several conferences devoted to special topics, critical to worldwide computing, e.g., the International Conference on Coordination Models and Languages (COORDINATION), the ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE), the IEEE International Symposium on Cluster Computing and the Grid (CCGRID), and the International World Wide Web Conference (WWW) series. Online bibliographies for many of these conferences and journals can be found on the World Wide Web.

1.6 Defining Terms

Common Object Request Broker Architecture (CORBA): Suite of protocols, languages and software systems for object-based distributed computing.

HyperText Markup Language (HTML): Language used to write Web content, including hypertext references or anchors.

HyperText Transfer Protocol (HTTP): Light-weight network protocol designed for information exchange between web servers and clients.

Local Area Network (LAN): Computer network physically colocated, which is characterized by low latencies and high bandwidth.

Middleware: Software layers in between applications and operating systems dealing with distributed computing issues, such as naming, mobility, security, load balancing, and fault-tolerance.

Remote Method Invocation (RMI): Java programming language API and framework for RPC-style synchronous interactions between objects.

Remote Message Sending Protocol (RMSP): Network protocol for asynchronous message exchange between WWC actors. Used by Java code generated from code written in the SALSA programming language.

Remote Procedure Call (RPC): Protocol for invoking remote procedures and marshalling and unmarshalling arguments across a network.

Simple Actor Language System and Applications (SALSA): Programming language facilitating the development of WWC applications using the universal actor model.

Uniform Resource Identifier (URI): Generic term uniformly denoting names, locators, or citations for worldwide resources.

Uniform Resource Locator (URL): URI specifying the location of a Web resource.

Universal Actor Locator (UAL): URI specifying the location of a universal actor.

Universal Actor Name (UAN): URI specifying the name of a universal actor, transparent to its location.

Universal Actor Naming Protocol (UANP): Network protocol to exchange information with a naming service regarding universal actor names and locations.

Wide Area Network (WAN): Computer network spread by a distance larger than a single building, potentially across continents. Characterized by higher latencies and lower bandwidths.

World-Wide Computer (WWC): Suite of protocols, languages, and software systems for worldwide computing using universal actors.

Worldwide computing: Area in computer science and engineering that studies all aspects related to using a wide area network as a computing and collaboration platform.

World-Wide Web (WWW): Suite of protocols, languages and software systems for worldwide information exchange.

1.7 Acknowledgments

Many ideas presented here are the result of countless discussions in the Open Systems Laboratory at UIUC; in particular, we would like to express our gratitude to Mark Astley, Nadeem Jamali, Yusuke Tada, Prassanna Thati, Koushik Sen, James Waldby, and Reza Ziaei for helpful discussions about this work, and for some of the figures. We also thank members of the Worldwide Computing Laboratory at Rensselaer Polytechnic Institute for many discussions about worldwide computing research; in particular, we would like to express our gratitude to Travis Desell, Kaoutar El Maghraoui, and Abe Stephens for continued development of the SALSA programming language and IO middleware framework. The work described here has been supported in part by DARPA IXO NEST Program under contract F33615-01-C-1907, by the DARPA IPTO TASK Program under contract F30602-00-2-0586, and by ONR under MURI contract N00014-02-1-0715.

References

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

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.

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.

Gul A. Agha. Introduction: Adaptive middleware. *Communications of the ACM*, 45(6):30–32, June 2002. ISSN 0001-0782.

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

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.

G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *19th International Conference on Distributed Computing Systems (19th ICDCS'99)*, Austin, Texas, May 1999. IEEE.

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.

G. Bollela, J. Gosling, B. Brosgoland, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specication for Java*. Addison-Wesley, Reading, Massachusetts, June 2000. Available from http://www.rtj.org/rtsj-V1.0.pdf.

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.

N. Brown and C. Kindel. Distributed component object model protocol – dcom/1.0. Technical report, Microsoft, May 1996. http://ds1.internic.net/internet-drafts/draft-brown-dcom-v1-spec-00.txt.

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.

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

L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995. URL `http://research.microsoft.com/Users/luca/Papers/Obliq.A4.pdf`.

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

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.

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

Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Distributed Systems Online*, 3(4), 2002. URL `http://dsonline.computer.org/0204/features/wp2spot.htm`.

Travis Desell, Kaoutar El Maghraoui, and Carlos Varela. Load balancing of autonomous actors over dynamic networks. In *Proceedings of the Adaptive and Evolvable Software Systems: Techniques, Tools, and Applications Minitrack of the Software Technology Track of the Hawaii International Conference on System Sciences (HICSS'37)*, January 2004.

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.

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

R. Gray, D. Kotz, G. Cybenko, and D. Rus. Mobile agents: Motivations and state-of-the-art systems. Technical report, Darmouth College, April 2000. Available at ftp://ftp.cs.dartmouth.edu/TR/TR2000-365.ps.Z.

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.

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

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.

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

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.

F. Kon, F. Costa, G. Blair, and Roy H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002.

Andreas Krall. Efficient JavaVM just-in-time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, 1998. North-Holland. URL `citeseer.nj.nec.com/krall98efficient.html`.

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

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

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

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

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

OVM Consortium. OVM An Open RTSJ Compliant JVM. http://www.ovmj.org/, 2002.

Douglas C. Schmidt, Aniruddha Gokhale, Timothy H. Harrison, and Guru Parulkar. A high-performance endsystem architecture for real-time CORBA. *IEEE Communications Magazine*, 14(2), 1997. URL `citeseer.nj.nec.com/schmidt97highperformance.html`.

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.

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.

Sun Microsystems Inc. – JavaSoft. Remote Method Invocation Specification, 1996. http://www.javasoft.com/products/jdk/rmi/.

Sun Microsystems Inc. – JavaSoft. JavaSpaces, 1998. http://www.javasoft.com/products/javaspaces/.

Camron Tolman. A Fault-Tolerant Home-Based Naming Service for Mobile Agents. Master's thesis, Rensselaer Polytechnic Institute, April 2003. http://www.cs.rpi.edu/wwc/theses/fhns/cam_thesis_final.pdf.

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.

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.

C. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, April 2001.

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.

Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001. ISSN 0362-1340. http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf.

N. Venkatasubramanian. Safe composibility of middleware services. *Commun. ACM*, 45(6):49–52, 2002.

Nalini Venkatasubramanian and Carolyn Talcott. Meta-architectures for resource management in open distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 144–153, New York, August 1995. ACM Press.

J. Waldo. JINI Architecture Overview, 1998. Work in progress. http://www.javasoft.com/products/jini/.

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997. URL `citeseer.nj.nec.com/waldo94note.html`.

Pawel Wojciechowski and Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999. URL `citeseer.nj.nec.com/article/wojciechowski99nomadic.html`.

A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.

Index

ABCL, 16
actors, 2, 6
  languages, 6
  libraries, 6
  migration, 11
  names, 9
  reflection, 12
  universal, 6
ActorSpaces, 2, 17
  capabilities, 17
  patterns, 17
agents
  mobile, 17
asynchronous communication, 2

CORBA, 2, 17
  services, 3

DCOM, 2

grid computing, 16

Java, 18
  RMI, 2, 18
  virtual machine, 4

Linda, 2, 18

middleware, 2
  adaptive, 4, 18

publish and subscribe, 2

reflection, 4
RPC, 2

SALSA, 7, 19
  migration, 12
  universal naming, 10
services, 7
  actor creation, 7
  concurrency, 3
  coordination, 9
  externalization, 3
  grouping, 4
  life-cycle, 3, 9
  messaging, 7
  migration, 9
  naming, 3, 7
  persistence, 3, 7
  query, 4
  replication, 4, 9
  split and merge, 9
  transactional, 4
  transport, 7

THAL, 16

universal naming, 9

Web Services, 2
World-Wide Computer, 5, 7, 19
worldwide computing, 5, 16, 18

X-XXXX-XXXX-X/XX/$X.XX+$X.XX
© 2004 CRC Press LLC.