

# Telepresent Agents: A New Paradigm for Networked Systems

## Russ Abbott

The Aerospace Corporation  
El Segundo, California  
and  
Department of Computer Science  
California State University, Los  
Angeles  
Los Angeles, California  
RAbbott@CalStateLA.edu

## Behzad Parviz

Department of Computer Science  
California State University, Los  
Angeles  
Los Angeles, California  
BParviz@CalStateLA.edu

## Chengyu Sun

Department of Computer Science  
California State University, Los  
Angeles  
Los Angeles, California  
CSun@CalStateLA.edu

**Abstract.** *Telepresent Agents are software Agents that are instantiated simultaneously within two or more distinct computer applications (called Locales) which are operating at distinct nodes of a computer network. Agents are composed of Capabilities—often a separate Capability for each Locale in which the Agent operates. As an Agent exchanges information among and within its Capabilities, information passes implicitly among the Locales as if through a wormhole communication channel.*

*One way to think about SimX—our name for the mechanisms described here—is that it does for data shared among Locales and Remote Procedure Calls what object-oriented programming did for instance variables and subprogram calls. By providing an entity-like construct within which such calls and data can be encapsulated, a construct that cuts across Locale boundaries, SimX eliminates the need for explicit mechanisms to allow Agents and Locales to communicate directly.*

**Keywords:** agent-based system, capability, distributed system, locale, SimX, synchronization, telepresent agents.

## I. INTRODUCTION

Network-based processing is vital to the fields of agent-based and evolutionary computation.

- In agent-based computations, agents are often understood as operating in two environments simultaneously. For example in economic modeling, an agent may participate in both a market environment and a production environment.

Agents may also be understood as moving from one environment to another; when one source of a resource depletes, the agent may move to another source. Each of these sources may be modeled as a separate environment.

- In evolutionary computation, one often encourages diversity through the creation of multiple loosely linked evolving populations. Individuals are permitted to migrate—at controlled rates—from one to the other.

This paper discusses SimX, a conceptual framework for network-based systems. The SimX approach to networked systems is formulated in terms of what are called *Locales*—self-contained computer applications—and *Telepresent Agents*—software agents that may be understood as being instantiated in multiple Locales simultaneously. Section VIII reviews a number of the existing approaches and discusses how SimX differs from them.

A Java implementation of SimX has been developed. In the discussion to follow, we make free use of concepts from object-oriented programming in general and Java in particular.

## II. NETWORKS, LOCALES, AND AGENTS

This section sets the context with some basic definitions.

### A. Networks, Nodes, and Sockets

A network is a collection of nodes linked by communication channels. A *Node* is a source and destination for messages. Although separate

Nodes typically reside on separate physical computers, there is no reason to disallow multiple Nodes from co-existing on a single computer.

SimX implements communication channels between Nodes with Java Sockets using the TCP protocol. Thus a SimX network consists of some number of Nodes linked by Java Sockets. In SimX, Nodes are relatively invisible. We do not discuss them further.

### B. Agents

A software agent is typically understood as a computer program that operates with a certain degree of autonomy. From the perspective of object-oriented programming one may usefully think of an agent as any object that runs as a Thread. Intuitively, an agent is an instance of a Thread—or of a class that implements Runnable.

A feature of Agents that has received less attention than it deserves is the requirement that agents control their own boundaries, i.e. that they are containers of a sort with boundaries that cannot be violated except in specified ways. In object-oriented programming this property is typically implemented through the use of method and instance variable modifiers such as *public*, *protected*, and *private*.

SimX Agents make very few methods public. The public methods exposed by SimX Agents fall into two broad categories.

1. Methods that allow the outside world to present information to the Agent.
2. Methods that reveal one or more of the agent's properties or aspects of the agent's state.

None of the methods in the above categories cause an Agent to act. All of the actions taken by Agents are driven by the Agent's internal thread. In general it is not possible for an external thread to reach into an Agent and cause it to act. This is a stricter restriction than that typically seen in object oriented programming.

We will call this constraint the *boundary integrity* property. Agents have an inside and an outside; they control strictly what passes over the boundary that separates the inside from the outside.

Agents with the boundary integrity property communicate by passing information to each other and by placing information in the environ-

ment but not by operating on each other. This form of interaction is called *stigmergy*, see [1].

### C. Locales

A Locale corresponds to a computer application that can run in a non-distributed environment, i.e., on a single computer.

A Locale Realization is the execution of a Locale on a Node. In object-oriented programming terms, a Locale is to a Locale Realization as a Class is to an Instance.

SimX defines an *abstract* Locale class. All other Locales are subclasses of the Locale class. A Locale Realization is the instantiation of one of these subclasses. A legacy computer application can be re-implemented as a Locale by writing a special wrapper Locale that runs that application within it. From here on we use the term *Locale* to refer to any single-computer application.

From an Agent-based perspective, a Locale may be understood as an environment within which Agents may act and interact.

Each Node supports and provides communication services for exactly one Locale Realization.

## III. A MOTIVATING EXAMPLE

### A. Distributed Simulations

Imagine a distributed simulation of a military campaign. We will suppose that the simulation is not interactive; it runs on its own.

Suppose that the simulation consists of three more or less independent simulations:

- a battlefield simulation in which personnel and other military assets move and interact,
- a weather simulation that had been developed independently of its possible military implications, and
- a simulation of the communication network used by the military, but developed independently of the battlefield simulation.

For simplicity, suppose that each of these simulations is implemented as a single Locale, i.e., that there are three locales.

Let's focus on the following two types of Agents.

1. A commander is an Agent that operates in the battlefield, interacting with his troops and equipment as well as with the enemy and with the environment. In addition, such an Agent communicates with his superiors, peer, and subordinates by using the communication network.

Such an Agent must be *present* in two of the three Locales, the battlefield Locale and the communication network Locale.

2. A weather reporter Agent that provides relevant information about the weather to the battlefield simulation. It gets that information from the weather simulation by operating as an observer within that simulation, gathering information that is relevant to the battlefield simulation. It makes that information available to the battlefield simulation, which changes the environment within which the Agents in the battlefield operate.

Such an agent must also be *present* in two of the three Locales, the battlefield Locale and the weather simulation Locale.

Let's assume that each Locale has just one Realization, e.g., on its own node in the network. Thus we have three Locales, two Agents, and three Locale Realizations. Each Locale has one Realization, and each Agent is present in two Locales—and therefore in two Locale Realizations.

#### IV. DISTRIBUTED VS CENTRALIZED SYSTEMS

When an Agent is present in two Locale Realizations, we distinguish between the Agent's *home* Realization and the remote Locale Realization(s) where the Agent also appears. It is the representation of the Agent in its home Realization that reflects the true state of the Agent. The representations of Agents in their home realizations are called *home agent instances*, or just *home agents*. The representations of an Agent in remote locale realizations are called *proxy agent instances*, or just *proxy agents*.

Because different Agents may have different home Locale Realizations, the true state of the system as a whole, i.e., the true states of the set of all Agents, may be distributed.

In contrast, most multiplayer online games have a centralized state. The single home Lo-

cale Realization is the one on the game company's computer. This approach is not impossible from our perspective, but it is not necessary.

#### V. HOW AGENTS FIND LOCALES: THE REGISTRY

Any web-based system needs some sort of resource locator. We call ours a Registry. The Registry exists at a known Internet address and is listening for connections on a known port.

Both Agents and Locale Realizations register with the Registry, which serves as a sort of classified ad repository, listing both Agent needs and existing Locales and their Realizations.

When a Locale Realization is created, it sends a message to the Registry identifying itself by its Locale type. Recall that every Locale is a subclass of the Locale class. Thus a Locale Realization identifies itself to the Registry in terms of its subclass of the class Locale.

When an Agent seeks to enter one or more Locales, it (really its home Locale Realization) sends to the Registry a request for an invitation to the type(s) of Locale the Agent is seeking. The Registry stores that information. It also informs all Locale Realizations of the requested type(s) that an Agent is seeking entry. Those Locale Realizations may then invite the Agent to send them proxies.

In addition, whenever a new Locale Realization joins the network, the Registry will inform it of all Agents seeking known to be entry to its Locale type.

This mechanism allows Agents and Locale Realizations find each other.

- Locale Realizations register their types with the Registry.
- Agents register their needs for Locale types with the Registry.
- When a Locale type matches an Agent need, Locale Realizations of that type are offered the opportunity to invite the Agent to send a *proxy*.
- Once an Agent proxy is established at a Locale Realization all further communication is directly between the Agent and its proxy. Neither the Registry, the Agent's home Locale Realization, nor the proxy's host Locale Realization provides any

middleman communication services between an Agent and its proxies.

## VI. AGENTS AND CAPABILITIES

If an agent is to operate in multiple Locales, it typically needs a number of different capabilities. In SimX, Agents are defined in a way to facilitate just such a structure. SimX Agents are collections of what are called Capabilities. A capability is simply a Class that is a subclass of the SimX Capability class. An Agent has as its most important component a collection of instances of Capability subclasses.

It is the capabilities associated with an Agent that defines its application-level functionality. There is only one Agent class; none of its methods implement any application-specific functions. All the application-specific methods are in the Capability objects associated with an Agent. Every Agent must have at least one Capability. Simple Agents may have just one Capability.

### A. *The Need for a new Programming Language Construct*

Earlier we described the boundary integrity property of Agents, which requires Agents to expose very few methods in its API. Yet if Agent consist of a collection of capabilities and if those capabilities are to interact, sibling Capabilities must expose methods—methods that should not be accessible to objects outside the Agent. We need a programming language construct that allows Capability instances of the same Agent to call methods in each other but that prevents these methods from being called from elsewhere.

This calls for a second level of encapsulation. Objects are one level of encapsulation; Agents, which themselves are made up of Capability objects, are a second. There is currently no mechanism in Java to declare this restriction.

## VII. AGENT SYNCHRONIZATION

To illustrate how SimX implements Agent synchronization, lets consider how time is synchronized among the three simulations in the battlefield simulation example. First we must discuss TimeGenerators and the TimeAwareLocale class.

### A. *The TimeGenerator Locale*

Besides the three Locales already described, there is a fourth Locale, a TimeGenerator Locale. A TimeGenerator can be as basic as a simple

time stepper, a task that sleeps for a period of time and then issues a time signal. Our objective is to have the three operational Locales in the simulation synchronized by time signals generated by a single TimeGenerator.

When the TimeGenerator joins the network, it declares itself to the Registry. When the other simulations join the network, they each generate what we will call a *Clock Agent* that seeks a TimeGenerator Locale. The three simulations are thus each invited to send proxy Clock Agents to the TimeGenerator. Consequently, there are three Clock Agents. Each has as its home locale one of the operational simulations. Each has a proxy Agent at the TimeGenerator Locale.

These three Agents function like the weather reporter Agent described earlier. They report time ticks generated at the TimeGenerator to their home Locales. These Agents each have a single Capability, a Clock Capability that knows how to report time events to its home Locale.

To make this concrete consider the `timeChanged()` method shown in Figure 1. It illustrates one mechanism SimX Agents use to synchronize themselves. This is a method in the Clock Capability.

TimeGenerators are defined so that whenever a new Time (tick) is generated, all Clock capabilities of resident proxy Agents will have their `timeChanged()` methods called. To understand how this works, we must first discuss TimeAwareLocales.

### B. *TimeAwareLocales*

The class TimeGenerator is a subclass of the class TimeAwareLocale. TimeAwareLocales are defined to pass on the time signals that they receive to all resident Agents (and proxies). That transmission is done by placing the time signal on the Agent's input queue. When the Agent processes a time signal, it passes that signal on to those of its capabilities that implement the TimeListener interface (i.e., that define a `timeChanged()` method) by calling `timeChanged()` in the Capability. This chain of transmission preserves the Agent's boundary integrity property.

### C. *Home Methods for Agent Synchronization*

As Figure 1 shows, the `timeChanged()` method in a Clock Agent's Clock Capability is declared to be a *home* method. When a *home* method is called in a proxy it is not executed

```

public home void timeChanged(Time newTime) {
    ((TimeAwareLocale)getLocale()).timeChanged(newTime);
}

```

Figure 1

there. Instead the corresponding method (with the same parameters) is called in the home Agent.<sup>1</sup> Such a call is *not* a traditional remote procedure call; the proxy does not block waiting for the call to complete. Instead, one can think of such calls as spawning a new thread in the (home) agent but as having no effect in the proxy. (When called in a home Agent, such methods are executed normally.)

When executed (in the home Agent’s Clock Capability), the `timeChanged()` method calls `getLocale()` to get the Locale Realization where the home Agent is situated. The method then calls that Locale Realization’s `timeChanged()` method, passing on the new Time object.

As described above, all of the Agents in a TimeAwareLocale Realization are told about each new time tick. Since the Clock Agent itself is an Agent in its home Locale Realization, this mechanism leads to this same `timeChanged()` method being executed a second time. That second execution has no effect; the Locale already knows about this Time step and does not process it a second time.

#### D. TimeGenerators

Given this framework, a TimeGenerator can now be understood as a TimeAwareLocale that has internally generated time signals. That is, a TimeGenerator doesn’t rely on a Clock Agent to call its `timeChanged()` method. It calls its own `timeChanged()` method whenever a

new time step occurs. Everything else, including updating of all other TimeAwareLocales, follows from the definitions of Agent and TimeAwareLocale and from the SimX Agent synchronization mechanisms.

#### E. Automatic Generation of Clock Agents

The TimeAwareLocale class is defined so that whenever it is instantiated, it automatically generates a Clock Agent, which then searches for a TimeGenerator to which to send a proxy. As a result, every Locale that is declared to be a subclass of TimeAwareLocale automatically receives time step signals. Thus if each of the three simulations in the battlefield example are defined as subclasses of TimeAwareLocale, they each have Clock Agents created for them automatically. These Clock Agents then seek out a TimeGenerator for time signals.

All a subclass of TimeAwareLocale need do to make use of the a Clock Agent’s time signals is to override the TimeAwareLocale’s `timeChanged()` method. When doing so, it is important to call

```
super.timeChanged(newTime)
```

to allow the TimeAwareLocale itself to update its time information and to pass the time signals on to other Agents in the Locale.

## VIII. COMPARING APPROACHES TO LOCALE-TO-LOCALE COMMUNICATION

The Clock Agent example illustrates how Agent synchronization allows two Locale Realizations to exchange information. When a *home* method is called in a proxy it is not executed there. Instead the corresponding method (with the same arguments) is called in the home Agent.

This mechanism provides a transparent wormhole communication channel between Locale Realizations. Methods called in a proxy in one Realization are executed in the Agent’s home instance.

In the remainder of this section, we briefly review a number of other approaches that have

---

<sup>1</sup> The current SimX implementation does not include a language preprocessor; there is no actual *home* modifier. The semantics of such a modifier is implemented more directly. We use the modifier notation because it is a better reflection of how the concepts are to be understood.

It would be a major step to define extensions to Java. Yet there are circumstance when syntactic extensions seem warranted. Aspect-oriented programming (see [2]) seems to be one such case. Agents, especially networked agents, may be another.

been taken to allow Locales to pass information from one to another.

1. *Message passing.* The simplest form of message passing is the transmission of information over connections between Nodes in a network. Although this mechanism is very useful (and serves as the basis of the underlying SimX communication system), it forces one to focus on the fact that communication is occurring.
2. *Message services.* An extension of message passing is the addition of message services such as publish, broadcast, and subscribe. (See, for example, [3]) Message services make it somewhat easier to use message passing, but they do not change the basic conceptual model. One is still thinking about sending, receiving, and processing messages.
3. *Remote procedure calls.* A remote procedure call (or Remote Method Invocation in Java) is a specialized form of message passing. Under this approach objects make method calls on other objects in other Locale Realizations. Remote procedure calls are specialized paired messages: the call and the returned value. Implementations of this approach include CORBA [4], and JINI [5].

When what one wants is to make a method call on a remote object, this sort of service is very useful. The disadvantage is that it provides very little additional conceptual leverage and offers only relatively low level conceptual services.

4. *Agent Communication Languages (ACL).* The ACL approach takes message passing in another direction. Instead of turning a message into a method invocation, an ACL defines a semantics (or at least a framework for a semantics) in which one can express possibly sophisticated communication content in messages. This is the approach taken by the Web services framework (see [6]), the Knowledge Query and Manipulation Language (KQML) (see, for example, [7]) and the Foundation for Intelligent Physical Agents (FIPA) (see, for example, [8]).

The Open Agent Architecture (OAA) project (see [9]) provides one of the more

ambitious implementations of this approach.

As with the other approaches, the ACL approach still requires one to focus on sending messages. Indeed, under the ACL conceptual framework the message and the expression of its content semantics in the ACL becomes even more central.

5. *Shared Databases.* An alternative to message passing is shared databases. If Locale Realizations can store and query a shared database, they can use that database to pass messages and to share information. There are still messages, but under this approach, the messages are communications between the Locale Realizations and the shared database. Again, the problem is that one is still focusing on the communication between nodes and the expression of the information in a form that can be stored in the shared database.

An interesting variant of this approach is Linda [10] in which the shared database, which consists of prolog-like assertions, supports prolog-like unification.

6. *Interoperability Standards.* A somewhat different approach is that of defining standards that allow Locale Realizations to interact. A current effort in this tradition is IEEE 1516, a Standard for Federated Simulations (see [11]), an outgrowth of the Defense Modeling and Simulation Office (DMSO) High Level Architecture (HLA) initiative. Under this approach all participating Locale Realizations must implement a defined set of services, which are to be made available to the other Locale Realizations. This is similar to the remote procedure call mechanism but with predefined APIs at each Locale. Again, the weakness of this approach is that it focuses on communication between Locale Realizations. Another difficulty is that the set of services required of each Locale Realization tends to be quite burdensome.

Most of the approaches sketched above might be considered conceptually heavyweight; they involve significant conceptual overhead on the part of software developer/users. The SimX approach differs in that it minimizes the conceptual overhead price users must pay. Instead of forcing users to focus on constructing communica-

tion transmissions, it allows users to focus on the task at hand, i.e., what the agents are doing with the information.

## IX. OTHER SIMX AGENT COMMUNICATION MECHANISMS

Earlier we discussed the *home* modifier for methods. SimX provides a number of other Agent synchronization mechanisms.

- One may declare an instance variable in a Capability to be a *state* variable. Whenever such a variable has its value changed, that value is changed in both the home Agent and all the proxy Agents.
- Parallel to declaring a method a *home* method, it is also possible to declare a method a *proxy* method. Such methods are executed only in proxy Agents and not in home Agents.
- Methods may be declared to be *parallel* methods. No matter where these methods are called, they are executed both in the home Agent and in all the proxy Agents.

SimX implements *parallel* methods and *state* variables similarly. Whenever a *parallel* method is invoked or a *state* variable is changed, no change takes place immediately. Instead a message is sent to the home Agent, which executes the method or changes the variable. The home Agent also sends messages to all its proxy Agents (including the one that originated the action) instructing them to execute the method or change the variable. Of course, all these messages are invisible to the software developer and occur at what might be called the SimX virtual machine level. Information is transmitted when making these method calls because the arguments to the calls are included in the remote method invocation.

## X. CONCLUSIONS

SimX provides a new framework for networked systems that allows developers to focus on how information is used in multiple Locales rather than on how that information is transmitted among the Locales. The SimX notion of Telepresent Agents with multiple Capabilities allows information to flow back and forth among Locales as it flows back and forth among and within the Capabilities of an Agent.

SimX simplifies the architecture of networked systems because it eliminates the need to construct messages to transmit information among the system components. In so doing, it reduces complexity by eliminating an unneeded level of formality and indirection.

## ACKNOWLEDGMENTS

The authors thank Ric Cowan, Ted Davey, Ivan Filippenko, Greg Mullert, Alan Quan, Debora Shuger, Bob Weber, and Farhad Zaerpour for insightful discussions. Ivan and Greg also contributed significantly to the development of the SimX software.

## REFERENCES

- [1] Bonabeau, Eric, "Editor's Introduction: Stigmergy," *Artificial Life*, Vol. 5, issue 2, MIT Press, Spring 1999.
- [2] AspectJ Team, *The AspectJ Programming Language*, <http://eclipse.org/aspectj/>, 2/03.
- [3] Monson-Haefel, R. and D. Chappell, *Java Message Service*, O'Reilly, 2000.
- [4] Object Management Group (OMG), "CORBA/IIOP 2.2 Specification," February 1998. Available online at <http://www.omg.org/corba/corbiiopt.html>.
- [5] Sun Microsystems, "JINI™ Architecture Specification," 2001. Available at [http://www.sun.com/software/jini/specs/jini1\\_2.pdf](http://www.sun.com/software/jini/specs/jini1_2.pdf)
- [6] Chinnici, R. et. al. "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," W3C, 2003. Available at <http://www.w3.org/TR/2003/WD-wsdl20-20031110/>.
- [7] Labrou, Y. and T. Finin, "A Proposal for a New KQML Specification, TR-CS-97-03, Dept. of Computer Science, University of Maryland, Baltimore County, 1997. Available at <http://www.cs.umbc.edu/kqml/papers/kqml97.pdf>
- [8] FIPA, FIPA "Ontology Service Specification," 2000, available at <http://www.fipa.org/specs/fipa00086/XC00086D.pdf>
- [9] Cheyer, A. and D. Martin, "The Open Agent Architecture", *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 4 , no. 1, pp. 143-148, March 2001, Available at <http://www.ai.sri.com/~oaa/>.
- [10] Carriero, N. and D. Gelernter. "Data parallelism in Linda." *Languages and Compilers for Parallel Computing*, edited by U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Lecture Notes in Computer Science, Springer Verlag, Berlin, 1993.
- [11] IEEE, "1516.3-2003 IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP)," 2003