ROAM: Room-based Object-based Agent Middleware

Guido J. van't Noordende, Frances M.T. Brazier, Andrew S. Tanenbaum, Maarten R. van Steen

Division of Mathematics and Computer Science, Faculty of Sciences
Vrije Universiteit, Amsterdam, The Netherlands
{guido,frances,ast,steen}@cs.vu.nl

Keywords: Multi-Agent Systems, Mobile Agents, Distributed Systems, Middleware, Virtual Environments

Abstract

In this paper we present work in progress on a worldwide, scalable multi-agent system, based on a paradigm of hyperlinked rooms.[1] The framework offers facilities for managing distribution, security and mobility aspects for both active elements (agents) and passive elements (objects) in the system. Our framework offers separation of logical concepts from physical representation and hooks for security architectures.

1 Introduction

Multi-Agent Systems (MASs) offer an infrastructure in which agents can do work autonomously for their owner while moving along a physical or logical path. Existing MASs [1, 2, 3, 4, 5, 6, 7] often deal with mobile agents, sometimes with heterogenous agents, and occasionally with secure agents. Current solutions are rarely scalable and frequently lack location and distribution transparency. Very few systems offer any kind of paradigm or framework to help application designers produce coherent, well-structured environments. In this paper we propose a scalable multi-agent middleware system that solves these problems.

2 The ROAM Framework

Our framework consists of a world (or possibly multiple disjoint worlds), each containing a set of hyperlinked rooms. Each room contains agents and objects. At any instant, an agent is in one room, but agents can move from room to room and they can take objects with them.

In essence, a room forms a shared data-space for agents with regard to visibility. Agents can interact only with objects in the same room, but can send messages to agents anywhere in the world. However, normally an agent will do most of its business with other agents in the same room.

Entities in a room can be agents, objects, or hyperlinks. Each agent is a (possibly multithreaded) process running on one host. No part of the internal process state of an agent can be accessed from the outside by other agents. Objects are strictly passive: they consist of data and code hidden by an interface. Hyperlinks determine how the rooms are connected.

An agent enters a world by entering a room. Certain rooms in a world may be used as starting points; we call these 'entry rooms.' Once in an entry room, an agent may move to any other room to which that room is hyperlinked. Directly moving to internal rooms (behind an entry room) is not allowed; agents must always go through an entry room, and follow hyperlinks from there. A world may have multiple entry rooms.

When an agent enters an entry room for the first time, the agent is registered in the world. A component is provided in each world, called the *basement*, which keeps track of the information needed to make the world function, such as the location of the agents. The basement is not visible to agents.

The first thing an agent must access when entering a room is a special object, called *Room Monitor Object (RMO)*. The RMO registers all content in its room, and provides an interface for agents to interact with the room. Descriptions of entities in a room (e.g., agents, objects and hyperlinks to other rooms) are specified in *attribute sets (ASs)* which can be obtained through the RMO interface.

Each entity has a (possibly empty) attribute set, placed in the RMO, using which the entity is described. Example attributes are the name of an agent or the coordinates of an agent in a room. Attribute sets are (object,attribute,value) triples. The attributes of ASs may be fixed in a world, but they may also be extensible; in this case, a definition of the AS can be

[1] Submitted to ASCI 2001, May 30 - June 1, 2001, Heijen, The Netherlands. This version slightly revised, Feb. 8, 2001.

obtained through a method of the RMO interface. An agent can alter an attribute set only if it has permission from the owner of the entity. Each object in a room can also contain an attribute set internally, besides the one in the RMO, which specifies additional (private) attributes.

Event notification can be provided by the RMO and other objects that have an attribute set. An object that provides event notification has a method *notify(event-type)* in its interface. This method blocks (i.e., does not return) until the event occurs. The event-type is a template AS, in which the values of attributes that an agent is interested in are specified. As soon as an attribute in the object matches the corresponding template AS's value, notify returns. The event-type is comparable to a template tuple as in JavaSpaces [8]; in some worlds, the event-type may also include expressions. The evaluation mechanism of the event-type is hidden inside the object.

Every world can also have an *attic*. The attic contains global services and is directly accessible to agents in any room. Through the attic, an agent can obtain world-scoped information, for example, the topology (hyperlink layout) of a world, directory services, or a bulletin board service (e.g., for publishing agent information.)

3 Example Applications

As an example of the ROAM paradigm, consider a world designed for buying and selling raw materials for industry. An entry room is set up where interested parties can obtain information about the products for sale. Hyperlinks from this room lead to rooms for specific products, such as ore, water, and electricity.

Agents for users that want to buy or sell certain products can be launched into the system and go to an appropriate room where they can meet other agents that offer or want products. An offer may be negotiated, after which an agent can either return to its owner with the current offer, or communicate with other agents to try to negotiate a package deal (e.g., optimizing for the cheapest combination of ore, water, and electricity). Some global information such as up-to-date currency exchange rates, freight rates, etc., may be available to all agents through objects in the room or in the attic.

Other examples are: Multi-User Dungeons (MUDs), in which players have to find their way through a maze of rooms, in which they can find items and may meet many adversaries; a virtual learning environment, where users can move among classrooms; a shopping mall, where the world is divided up into departments, each with stores selling a certain category of products; and libraries, with rooms for different topics.

In short, the ROAM paradigm replaces the World Wide Web paradigm of a collection of hyperlinked documents that users can inspect with that of a collection of hyperlinked rooms in which agents can meet to do business.

4 Distribution Aspects of ROAM

A world may be spread over many machines. At any instant, each agent is a process running on some particular host computer. As agents move from room to room, they may also move from machine to machine. As an agent wanders from room to room in a world, it may not be aware of what machine it is on, unless it cares.

A mechanism is provided that allows agents to connect to objects transparently, using a location-independent identifier. This mechanism is called *binding.* Binding results in a binary interface being loaded into the address space of the agent, hiding the exact implementation and location of the object. After the interface has been loaded, the agent can invoke the object's methods.

Our model supports distributed objects, whose state may be replicated on a number of hosts simultaneously [9]. However, our model can also be built using other object architectures, such as those supported by CORBA [10], DCOM [11], or Java RMI [12].

In a world, logical location is decoupled from physical location. Logical components (e.g., rooms, objects and agents) as well as components that make a world work (e.g., basement) may be physically located on a number of hosts, but this is not visible to an agent in a world.

We offer a way to group hosts in terms of shared properties. This grouping is called a *zone.* A zone can consist of multiple hosts, and can be related to one or more properties. Examples of those properties are an administrative authority, or a common operating system. A host can be a member of multiple zones.

Zones can be used for location and distribution policies. A distribution policy can be used for a distributed object to determine to which hosts its data may be distributed. For example, a policy may dictate that only hosts in one zone may hold the full state of a distributed object, whereas agents in another zone may access the object only through a proxy (which provides a mechanism to invoke methods on an object remotely, but has no state). Non-distributed components of a world such as agents may be located on any host in a zone of a world.

5 Security in ROAM

Protection of data 'on the wire' and point-to-point authentication can be achieved using well-known cryptographic mechanisms [13]. In a distributed sys-

tem, trust-based mechanisms are needed for authorization. Zones offer an elegant abstraction by which hosts can be grouped on security properties, that can be used for authentication and authorization.

As a practical example of how zones can be used for authentication, each zone may be associated with a public key, called the zone key. A certification authority can be used to sign this zone key and associated security properties, which can then be made available to others through a Public Key Infrastructure (PKI). For authentication of a host as part of a zone, a shared secret key may be used by the hosts in a zone, but usually some more advanced mechanism, for example using a group signature mechanism [14], will be used. The associated security properties, as published in the PKI, can be used as the basis for authorization.

Protecting a host against an agent running on it is usually possible (e.g., using sandboxing techniques as is done in Java [15] and SafeTcl [16].) Additionally, trust-based mechanisms may be used by a host before it accepts an agent to execute on it, to improve host security. Trust mechanisms can be based on several properties of an agent, such as who wrote its code, the agent's owner, or which zones it has passed through on its itinerary. A code certification scheme may be used to increase trust in an agent by a host that does not know the agent. The zone key can be used to check an agent's code certificate. Based on the above mentioned properties of an agent, a host can assign permissions to an incoming agent, or reject it. Protecting an agent against a malicious host on which it is executing is not possible in general [17].

Zones are visible only at the physical level, not on the logical level: zones are orthogonal to logical concepts such as agents, objects, and rooms. However, zone-related behavior may become apparent on the logical level, for example because some object may not be accessed due to zone-based security policies. In many cases, therefore, a world should provide mechanisms for an agent to query zone information and act on it.

6 Implementation

An agent must load in the runtime system of a world to access that world. This runtime system uses a middleware layer, which may provide general functionality such as communication libraries and a bind mechanism for a world. This middleware layer is aware of most ROAM concepts, such as zones and the basement. Using runtime system and middleware, an agent can interact with a world.

The basement is an essential part of any world. It contains among other things a registry of rooms and a location service (lookup service) for agents that are bound to a world. This information is among others needed to guarantee logical consistency of a world (e.g., an agent may not be in two rooms at a time), and for interagent communication based on location-independent agent identifiers.

To start an agent on some host, a *spawn service* must be present on that host, which is capable of starting up the agent in its proper execution environment (e.g., sandbox). Several Agent Programming Languages (APLs) may be supported by a spawn service on a host; the APL is the programming language in which the agent is written. Typically, only a few APLs are supported in a world. For example, a spawn service might execute a Java agent by executing a JVM with this agent in it, but it might also support binary agents that can run as native code on this host. Agents in a world are usually not allowed to access resources outside a world, such as the local filesystem or the Internet.

At any instant, an agent always runs on one host in one zone. An agent can move to a host in a different zone, or to another host in the same zone, by invoking a method enter_zone, specifying a target zone and preferences.

Preferences can be given by the agent, to facilitate proper selection of a host within the target zone. Examples of preferences that can lead to the selection of a particular host in a zone are closeness to resources (e.g., a database object), or some specific hosting OS. An agent that wants to move to another host in the same zone, must reenter its current zone using different preferences. Enter_zone is atomic: the move succeeds or the agent remains where it is. Changing zone does not change the room the agent is in. Enter_zone fails if the current room cannot be accessed from a target zone.

An *Agent Transfer Protocol (ATP)* is used to physically move agents from one host to another. The ATP negotiates the preferences of the agent with the receiving zone, which either results in the agent being sent to a host in this zone, or in the agent being rejected.

An *agent container (AC)* represents an agent in a marshalled format. An AC is divided into a number of segments, preceded by a Table Of Content (TOC). An AC is and sent over the network as a contiguous byte-stream when an agent is moved to another host. Segments are typed: types include shared agent code + data segment, agent code segment, agent data segment, object segment (to contain serialized objects), and data segment (to contain general data).

An AC is extensible by an agent: segments in the agent container can be changed, deleted and added; it contains at least one TOC and a segment containing the agent. The language in which the agent is written is indicated in the AC. A cryptographic TOC mechanism may be used that allows for inspection of what changes were made to an AC during its itinerary, and where. This allows for detection of malicious alter-

ations to the AC in most cases [18]. An agent is initialized on a target host by the spawn service that is present there, using the serialized agent contained in the AC.

Some components of a world must be present at world deployment. A world deployer must at least provide the basement and one entry room. Additional (hyperlinked) rooms, and initial content, i.e., objects contained in rooms, may also be provided by a world deployer.

Usually, a world is extensible: objects and rooms may be created dynamically. Adding object-based components to a world is done using an *Object Creation Service (OCS)*. For example, to create an object in the context of a room, the OCS makes an instance of a class object (dependent on the object architecture supported by a world), the identifier (object reference) of which is placed in a room. Similarly, to create a new room, a RMO is created by the OCS, the identifier of which is placed in one or more rooms if it concerns a hyperlinked (internal) room, or in the basement if it concerns an entry room. Policies in rooms or basement, subject to zone policies, determine whether a new object or room may be added or deleted.

Agents can create, move, copy, delete, and replace objects in a world (if security permits.) Usually, when moving an object from one room to another, moving the object's identifier to the new room is sufficient. However, sometimes it is necessary to copy (or move) an object physically to another zone, for example because it has to be placed in a room located in a zone with restrictive security policies. In this case, the object can be serialized by the OCS and placed in the AC of an agent. The OCS in the receiving zone instantiates a new object from a class object in this zone, and initializes it with the serialized state from the agent's AC. If an agent places an object in a room, the agent automatically becomes the owner. In some worlds, ownership may be transferable, for example when picking up items in a MUD.

7 World Design Language

We are currently developing a World Design Language (WDL) for specifying properties of a world, which facilitates designing worlds. Through the WDL, both high-level design issues about the world structure (e.g., constraints on room topology), as well as low-level issues (e.g., reliability of communication) are addressed.

Using the WDL, a world designer can specify items such as whether new rooms may be added dynamically; whether events are supported (and the mechanism for event-type evaluation); whether a particular mechanism is used to manage zone membership (and a mechanism for defining security policies); whether certain methods have to be provided by all objects in a world, and whether this object interface is extensible; what methods the interface for components such as the RMO has (some methods are mandatory for all worlds); whether attribute sets can be dynamically defined per room (and what language is used to describe attribute sets); whether there is a naming scheme for entry rooms; whether all agents should support a common Agent Communication Language (ACL) within this world, e.g., to communicate with agents provided by the world designer; which APLs are allowed (e.g., Java or SafeTcl agents); what architecture is used for objects (e.g.,Globe or Java RMI). This list is neither exhaustive nor final.

We are currently studying a middleware-based design for world systems, in which a runtime system is built specifically for each world, that makes use of functionality offered by the middleware. This middleware should be extensible, so that functionality specified in the WDL can be integrated in this middleware, if needed. It may be that this middleware layer always offers some standard functionality for all worlds, for example providing best-effort, unreliable, unordered interagent communication.

8 Discussion

Current multi-agent frameworks do not separate logical location from physical location. In our system the logical, virtual environment in which an agent resides uses concepts that are completely orthogonal to physical location. This is a major improvement over current systems. Furthermore, our framework imposes no restrictions on the APLs of agents using the system and it provides interagent communication independent of an agent's logical location. The zone concept allows for flexible and scalable deployment of worlds, using for example location and distribution policies to place (distributed) logical components on physical locations, and security policies to allow secure deployment of a world over multiple security domains.

9 Acknowledgements

Thanks to Niek Wijngaards for valuable discussions on the model.

References

[1] J. Baumann, F. Hohl, M. Strasser, K. Rothermel. *Mole - Concepts of a Mobile Agent System*, August 1997. Technical Report, Universität Stuttgart.

[2] F.M.T. Brazier B. Dunin Keplicz M. Jennings J. Treur. DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework.

Int'l Journal Coop. Information Systems, 6:67–94, 1997.

[3] R.S. Gray, D. Kotz, G. Cybenko, D. Rus. D'Agents: Security in a Multiple-language, Mobile-agent System. *Mobile Agents and Security*, pages 154–187, 1998. LNCS 1419, Springer-Verlag.

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

[5] L.C. Lee H.S. Nwana, D.T. Ndumu. Zeus, a collaborative agents toolkit. *Proc. of the 3d Int'l Conference on Practical Applications of Agents and Multi-agent Technology, London, UK*, pages 377–392, March 1998.

[6] T. Stolpmann H. Peine. The architecture of the ara platform for mobile agents. *Proc. First Int'l Workshop on Mobile Agents*, 1997. LNCS 1219, Springer-Verlag.

[7] H.C. Wong and K. Sycara. Adding Security and Trust to Multi-Agent Systems. *Proceedings of Autonomous Agents Workshop on Deception, Fraud and Trust in Agent Societies*, pages 149–161, May 1999.

[8] K. Arnold E. Freeman, S. Hupfer. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.

[9] A.S. Tanenbaum M. van Steen, P. Homburg. Globe: A wide-area distributed system. *IEEE Concurrency*, January-March 1999.

[10] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

[11] Microsoft Corporation. Dcom technical overview. November 1996.

[12] Sun Microsystems. *Java RMI. A New Approach to Distributed Computing*, 1998. White paper, http://java.sun.com/products/javaspaces/whitepapers/dcpaper.pdf.

[13] M. Speciner C. Kaufman, R. Perlman. *Network Security*. Prentice Hall, 1995.

[14] D. Chaum and E. van Heyst. Group signatures. *Advances in Cryptology - EUROCRYPT '91*, pages 257–265, 1991. LNCS 547, Springer-Verlag.

[15] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, June 1999.

[16] B.B. Welch J.K. Ousterhout, H.Y. Levy. The safe-tcl security model. *Mobile Agents and Security*, 1998. LNCS 1419, Springer-Verlag.

[17] C.F. Tschudin T. Sander. Protecting mobile agents against malicious hosts. *Mobile Agents and Security*, 1998. LNCS 1419, Springer-Verlag.

[18] N. Karnik and A. Tripathi. Security in the ajanta mobile agent system. *To appear in: Software - Practice and Experience*, 2000.