A Software Toolkit for Prototyping Distributed Applications*

(Preliminary Report)

James C. French
Charles L. Viles

University of Virginia Technical Report
CS-92-26
29 September 1992

Abstract

We describe a set of software tools for rapid prototyping of distributed applications. The toolkit is based upon a model of computational agents that are distributed in a developer specified manner throughout a host network of workstations. The model makes no assumptions concerning network heterogeneity. The distribution and functionality of agents completely specifies the communication patterns and topology of the distributed system. Agents that follow the communication protocol may be written in any language. The intended usage of this toolkit is for both education and rapid prototyping of distributed applications.

1. Introduction

For the distributed systems neophyte, it is often difficult to understand the issues that arise in developing a distributed application or porting an application from a centralized to a distributed environment. While these issues are well known (Mullender, 1989, Tanenbaum and van Renesse, 1985), the fundamental problems they cause can be better appreciated through actual involvement in a distributed application. For example, transparent naming is often cited as a requirement in a distributed application. Though a

*. This work was supported in part by grants from Simpson Weather Associates, Virginia Center for Innovative Technology grant no. VCIT CAE-91-012, and Department of Energy grant DEFG05-88ER25063

simple idea conceptually, it turns out to be a more difficult task to actually provide such a service. We believe that the implementation effort itself is sometimes the best way to develop an understanding of the larger issues.

Even for seasoned developers, a facility for rapid prototyping of a distributed computation is extremely useful. For many applications, a prototype for a distributed computation may end up being computationally slow, but will still help expose and highlight issues and problems that might otherwise be ignored until later in the development process (when, presumably, considerably more time and effort has been expended).

In this paper, we describe a set of software tools for rapid prototyping of distributed applications. The toolkit is based upon the idea of computational agents that are distributed in a developer specified manner throughout a host network. The distribution and functionality of agents completely specifies the communication patterns and topology of the distributed system. We believe these software tools may be of use in at least the two scenarios described previously: 1) as a prototyping framework for developing applications in a loosely-coupled distributed system like a network of workstations, and 2) as a learning tool for students in an advanced undergraduate or graduate level systems class. High performance is not a goal of these tools though it is not necessarily precluded either.

In section 2, we describe the idea of computational agents and their linkage in a Virtual Agent Network (VAN). We then show how they are used to frame a distributed computation. Section 3 is a description of the software tools themselves. In section 4, we discuss the details of agent to agent message routing, including a discussion of how certain distributed systems issues were handled in the development of the toolkit itself. In Section 5, we describe three example computations using the functions of Section 3. In section 6 we summarize and conclude.

2. Computational Agents and the Virtual Agent Network

A computational agent is defined as a disk-resident executable program that, when instantiated as a process, may communicate with other instantiated agents. This family of communicating agents together forms a distributed application. Agents may have some information about other agents that communicate with it. A sending agent will at least know the name of the agent it is sending to (though it may not know the location), and a receiving agent can find out who sent the message and where it came from. In general, this information is not necessary for an agent, though some kinds of agents may use it in their computations.

Making agents disk-resident rather than memory resident provides some advantages. First, it means that agents can be coded in any number of languages and in fact allows multilingualism on a single application, as long as the agent conforms to the listed communication patterns. Secondly, from an implementation point of view, it means that quiescent agents (those that are not currently involved in a computation) do not take up scarce system resources like memory and slots in the process table.

2.1. Definitions

Before going further we must define a few terms. The ***agent site*** is the physical location where an instance of an agent is created and its computation is performed. For a particular distributed application, a developer defines a number of ***agent types*** that in total can perform the entire distributed computation. An agent type is a template for an agent. It describes the computation the agent performs but is not an actual instance of the agent. In particular an agent type has no data or site associated with it. Agent types are identified by a string (e.g. “VectorMultiply”) that we will sometimes refer to as the ***agent name***. We hasten to add that the use of the term ***agent name*** is informal and does not connote any

greater ability to identify a particular process at a particular site. Finally, we define an ***agent instance***, as an executing process that is bound to a particular agent site and a particular set of data. We will often refer to an agent generically, and the particular kind of reference (type, or instance) should be apparent from context.

2.2. Virtual Agent Network

Agents are linked logically in a ***Virtual Agent Network*** (VAN). A VAN is composed of a number of sites, each of which may execute some subset of the computational agents that have been defined for a particular distributed computation. In the VAN, the developer specifies the particular agent types that may be instantiated at each site. We refer to this specification of the distribution of agents among sites as the ***agent configuration***. There is no a-priori constraint on the number of sites or on the nature of the agent configuration. Some sites may instantiate agents of all types, others may instantiate only some types of agents.

Each site in the VAN has a message router. The router's job is to deliver messages from one agent to one or more other agents while hiding all details of the message delivery from the participants. From the sending agent's view, a message is simply delivered to an instance of the destination agent type, whereupon the receiver performs some computation. The VAN takes care of message routing and the instantiation of a receiving agent. In some cases, only a single message is needed to perform the computation. In other cases, sustained bidirectional communication is needed.

In the VAN depicted in Figure 1, there are three types of agents defined, A1, A2, and A3. The distributed computation has five sites associated with it. The agent configuration shows that A1 may execute at every site, A2 at three sites, and A3 at two sites.

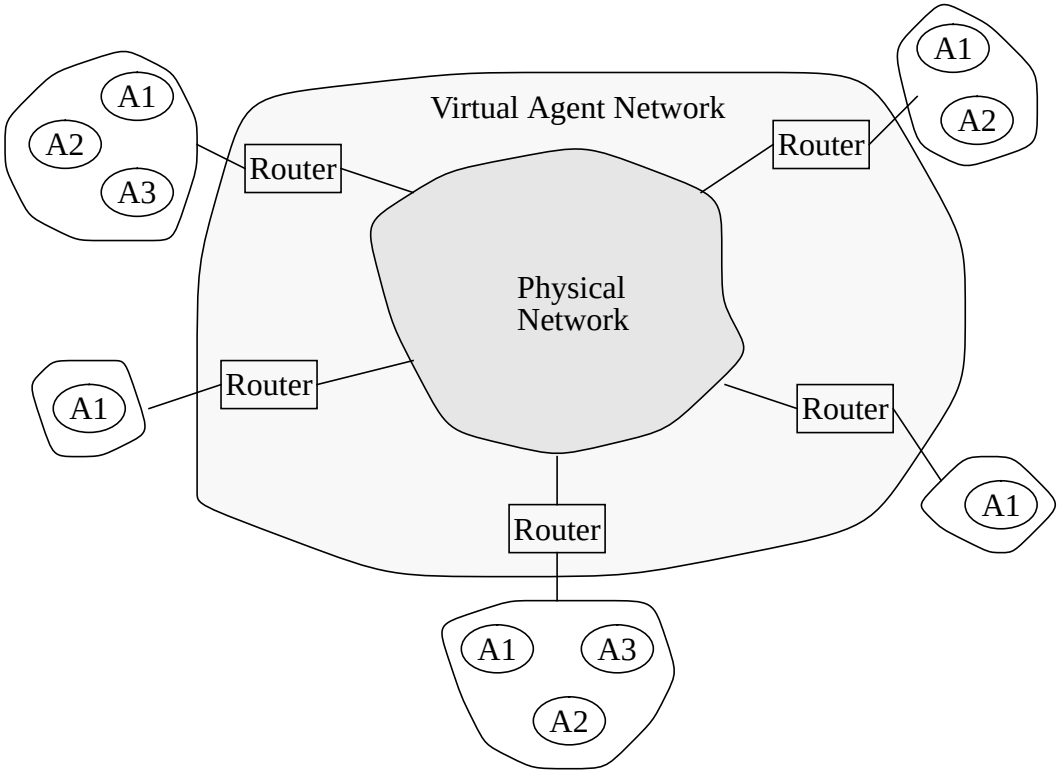


Figure 1

2.3. Heterogeneity

This distributed agent model has no constraints in terms of the heterogeneity of the VAN. Any site with the facility to route messages may participate in the VAN. In principle, any agent may execute at any site, provided the site has the resources needed to conduct the agent's computation. Of course, a developer would likely configure the agents to best match computation requirements with available site resources.

2.4. Distributed Computation

The general idea behind a computation is that each agent receives some data, performs its part of the work needed and passes the transformed data on to the next agent or agents in line. We make no requirement that agents be local or that the computation be

linear. Depending upon the agent configuration, the computation may be entirely local, entirely remote, or some mix of the two. The topology of the agent interaction may be linear or may form some more arbitrary graph with a single agent communicating with multiple agents of different types or multiple instances of the same agent type.

As a simple example, Figure 2 illustrates two agents. The MatrixEntry agent gets two matrices from the user and sends them to a MatrixMultiply agent to perform the multiplication. If MatrixEntry is invoked again, a new instance of MatrixMultiply will

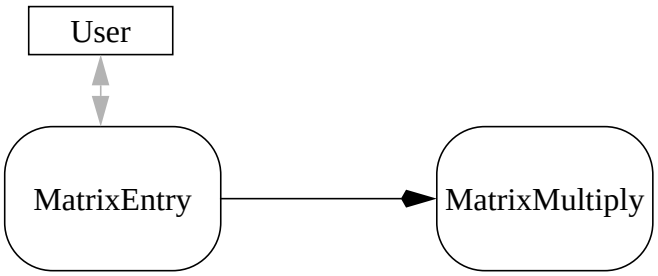


Figure 2

receive the two matrices.

The granularity of computation is defined by how the developer chooses to formulate his agents. We could easily have reformulated our first example so that the MatrixEntry agent sent a row and column to the DotProduct agent. The DotProduct agent would perform the appropriate computation on the given row and column and send the result back to the original MatrixEntry agent. MatrixEntry would then send the next row

and column, continuing until the multiplication was performed. Figure 3 illustrates this formulation.

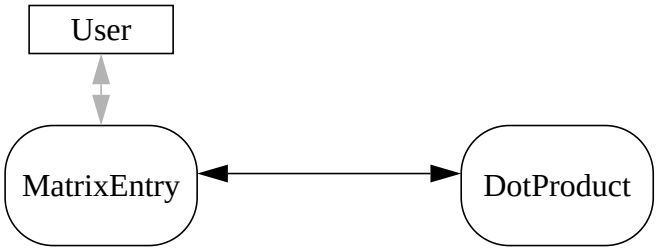


Figure 3

The agent model does not require any particular computation granularity and in fact, granularity is completely under user control. However, our present implementation is best suited for larger grained applications, because each message transmission and reception involves a number of process instantiations and possibly a remote link.

Agents may carry on simple 1-way communication with other agents (as in Figure 2) or they may have sustained, bidirectional conversations (as in Figure 3). The facilities for each are described in the next section.

3. Available Software Tools

Within the VAN model described in the previous section, agents communicate solely by message passing. In the following subsections, we describe the routines for both 1-way and bidirectional communication. We have found it convenient to define separate routines for the two types of communication. When only a single message needs to be sent, then a 1-way routine can be used. In all function specifications, we have used a C-like syntax.

3.1. Agent Naming

We are careful to differentiate between an agent name, an agent site and an agent address. An agent name is a string that identifies the agent type e.g. *"MatrixMultiply"*. An agent site is the physical location where an agent may be instantiated and executed e.g. *"ruby.cs.virginia.edu"*. An agent address is the concatenation of name and site, e.g. *"MatrixMultipy@ruby.cs.virginia.edu"*. The agent address identifies a particular site where an agent computation should be performed.

3.2. Sending

3.2.1. One way Communication

An agent can send a 1-way message in three ways *SendAny()*, *SendSpec()*, and *SendAll()*. The function interfaces are similar:

*int SendAny (char *AgentName, int AgentMsgType, int Length, char *MsgBody)*

*int SendAll (char *AgentName, int AgentMsgType, int Length, char *MsgBody)*

*int SendSpec (char *AgentAddress, int AgentMsgType, int Length, char *MsgBody)*

Each of the Send routines will return the value of *Length* if *MsgBody* is successfully delivered to the message passing system (VAN). They return a value less than 0 if the message cannot be delivered to the system. *AgentName* identifies the type of the destination agent i.e. the receiver. For *SendSpec(), AgentAddress* is a well-formed agent address as defined previously. *MsgBody* is the actual data the sender is transmitting to the agent(s) targeted by *AgentName* or *AgentAddress*. *MsgBody* must be contiguous and communicating agents must agree on the format of the data. *AgentMsgType* is provided as a convenience to the sender. If a sender wants to send several kinds of messages to the same agent type, then *AgentMsgType* is an easy way to notify the receiver of this fact. The

recipient can look at the message type to decide how to interpret the message. A similar effect could be achieved by embedding the type in *MsgBody*, but at some loss of convenience. *Length* is the length of *MsgBody* in bytes.

The semantics of who will receive the message are different for each of the functions. *SendAny()* is used when *MsgBody* can be sent to any site that can instantiate an agent of the named type. For example, the MatrixEntry agent of Figure 1 might send the following message

*SendAny ("MatrixMultiply", MATRIX_ENTRY, Length, (char *) matrixStruct);*

where *MatrixStruct* has been packed in a standard manner with the matrices and their dimensions. As far as MatrixEntry is concerned, it does not matter where the MatrixMultiply agent is instantiated and performs its computation, only that the computation is performed somewhere. In fact, when using *SendAny()* (a one-way communication primitive), MatrixEntry cannot know which of the agents in the agent configuration ends up performing the matrix multiplication. Agent scheduling is left up to the VAN, which (in some manner) examines the agent configuration and chooses a site to perform the computation. The scheduling policy is chosen to achieve system goals like load balancing and is independent of the agent computation.

The *SendSpec()* function expects *AgentAddress* to be a well-formed agent address. Continuing with our example, if MatrixEntry knows that the site *uvacs.cs.virginia.edu* can handle MatrixMultiply agents, then it could send the data there

SendSpec ("MatrixMultiply@uvacs.cs.virginia.edu",
*MATRIX_ENTRY, Length, (char *) matrixStruct);*

For now, we finesse the issue of how an agent finds the location of another agent by stating that either a user or another agent has supplied that information.

The final send routine is *SendAll()*. *SendAll()* is a multicast message, essentially a broadcast message within a group, where the group is all agents of a particular type. This routine is the only available mechanism for a sender to transmit a message to a group of agents, since there is no guarantee that an agent will know anything about the overall agent configuration, much less the sites where all agents of a given type can be instantiated. In section 5, we will see how *SendAll()* is used.

It is important to emphasize that the above primitives are for 1 time, 1-way communication. If a sending agent does two SendSpec() calls, then there will be a destination agent instance created for each call, even if the destination site is the same in both calls.

3.2.2. Bidirectional Communication

An agent can engage in bidirectional communication in three ways *SendDirectAny()*, *SendDirectSpec(),* and *Respond()*. The function interfaces for the first two are similar to their 1-way counterparts:

*int SendDirectAny (char *AgentName, int AgentMsgType, int Length, char *MsgBody)*

*int SendDirectSpec (char *AgentAddress, int AgentMsgType, int Length, char *MsgBody)*

SendDirectAny() and *SendDirectSpec()* set up a direct, 2-way channel between sender and receiver. The parameters and their semantics are exactly the same as for *SendAny()* and *SendSpec()*, but the return semantics are slightly different. Each of these 2-way routines returns the value of *Length* if *MsgBody* is successfully delivered to the input queue of the destination agent. They return a value less than 0 if the message cannot be delivered to the agent. Both routines will block until the message is delivered to the destination agent. The semantics of who will receive a message sent by either of the SendDirect routines are exactly the same as their 1-way counterparts.

The third 2-way routine is *Respond()*:

*int Respond (AgentMessage *PrevMsg, int AgentMsgType, int Length, char *MsgBody)*

This routine allows an agent to respond to a particular agent instance that has sent the message, *PrevMsg*. Accordingly, *Respond()* may only be used after a direct connection has been established. A direct connection is established when an agent is a sender and executes a *SendDirect()* routine or an agent is a receiver and receives a message from a sender that has used a *SendDirect()* routine. The recipient of *MsgBody* is the agent instance that sent *PrevMsg*. *Respond()* returns *Length* on successful delivery of the message to the input queue of the sender of *PrevMsg*, or less than zero if delivery is unsuccessful or if there is no open connection between the invoking agent and the sender of *PrevMsg*. The *AgentMessage* structure is described below.

Currently, we restrict agents to one open bidirectional channel at a time. If an agent does consecutive SendDirect calls, then the connection to the destination agent of the first SendDirect will be severed, and a new one will be opened to the destination agent of the second call.

3.3. Receiving

The VAN packages the parameters from the Send routines into a standard message format and arranges for delivery of a message to the destination agent(s). We call this standard message format an AgentMessage. An AgentMessage is what is delivered to receiving agents.

An agent receives a message using *AgentReceive()*.

*AgentMessage *AgentReceive();*

If the message is successfully read, then *AgentReceive()* returns a pointer to an *AgentMessage* structure. The fields of this structure are readily accessible via the assorted *GetMsg()* routines described below. If there is some problem, then *AgentReceive()* returns NULL.

An agent can discern a message originated by one of the 2-way routines from a message sent by one of the 1-way routines in one of two ways. It can either determine this information at the application level from the context in which the message was received, or the information itself can be explicit in the contents of the *MsgBody* or the *AgentMsgType*.

3.4. The AgentMessage Structure

A receiving agent has no knowledge of the details of the AgentMessage structure. *AgentReceive()* returns a handle for the message which may be used to access the components of the AgentMessage.

An agent has the following routines available to it once it has received an AgentMessage.

*char *getMyAddress (char *address);*

Returns the full agent address of the invoking agent

*char*getMsgDestAgent (AgentMessage *msg);*

Returns the name of the destination agent of the given message.

*char *getMsgDestSite(AgentMessage *msg);*

Returns the site of the destination agent of the given message.

*char *getMsgDestAddr (AgentMessage *msg);*

Returns the agent address of the destination agent of the given message.

*char *getMsgSrcAgent (AgentMessage *msg);*

Returns the name of the sending agent of the given message.

*char *getMsgSrcSite (AgentMessage *msg);*

Returns the site of the sending agent of the given message.

*char *getMsgSrcAddr (AgentMessage *msg);*

Returns the agent address of the destination agent of the given message.

*char *getMsgBody (AgentMessage *msg);*

Returns a pointer to the body of the message, that part containing application specific data.

*int getMsgType (AgentMessage *msg);*

Returns the type of the message. Useful when a receiving agent handles multiple kinds of messages.

*int getMsgSize (AgentMessage *msg);*

Returns the size (in bytes) of the message body.

4. The message router

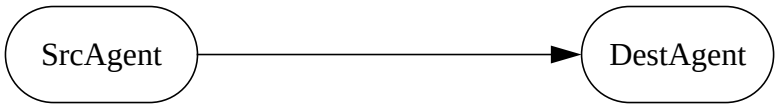
4.1. Message delivery

4.1.1. One way

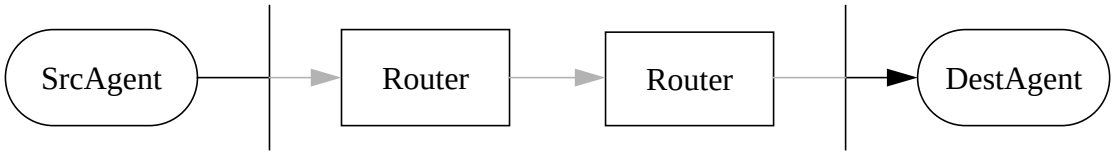
A typical message delivery for the 1-way, 1-time routines is as follows. An agent invokes one of the send routines. The message is passed to the local router as an OUTGOING message. If the message was sent using *SendAny()*, then the router chooses one of the agents it knows about and sends the message to that agent. If *SendAll()* was used, then the router broadcasts the message to all agents of that type. With *SendSpec()*, the message is transferred without any address translation. In all cases, an OUTGOING message is marked as INCOMING so that the receiving router knows how to handle it.

When a router receives a message marked INCOMING, then it invokes the specified agent and passes the message along to that agent. This message delivery involves an agent process instantiation whereupon the new process receives the message. In practice, a copy of the executable code for each agent type is cached at each site capable of instantiating that agent as specified in the agent configuration. This is purely an implementation efficiency, since the runtime system could easily transport the code from

some central site if it was not already present at the specified site. This message routing is represented graphically in Figure 4. The top portion illustrates the abstraction and the bottom portion shows the implementation.



Abstraction



Implementation

Figure 4

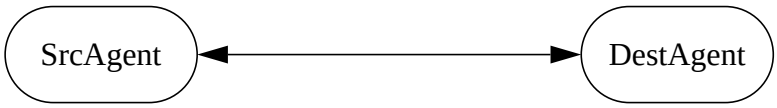
It is important to note that routers know absolutely nothing about the application specific content of a message - it is treated as a simple stream of bytes that is interpretable by some agent. Routers know only administrative details concerning a message i.e. length, type (INCOMING or OUTGOING), source, and destination. This very careful division ensures that the message routing system is completely separate from the application.

4.1.2. Bidirectional

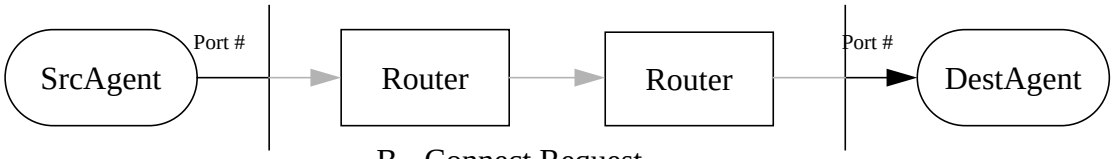
Message delivery in the bidirectional case is somewhat different. The routers are used to set up the connection between agents. Once the connection is established, all messages are direct agent-to-agent and the routers are completely out of the picture. This situation is illustrated in Figure 5. Figure 5A illustrates the abstraction and the Figures 5B, C and D show the implementation. To establish the connection, the sender sends a DIRECT_CONNECT message to the destination agent and also tells that agent a channel

number where the sender will be waiting for a connection request (Figure 5B). The receiver can then connect to that channel directly (Figure 5C). Once the connection is established, 2-way communication is possible over that channel (Figure 5D), completely bypassing the routers.

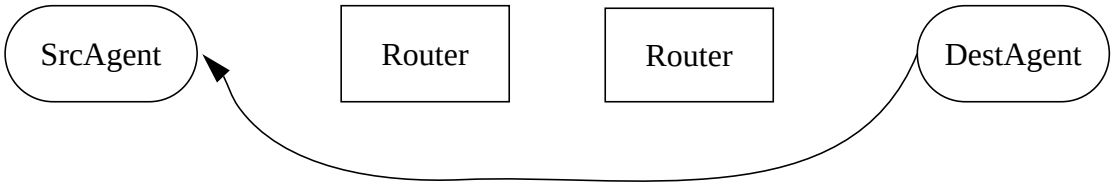
.



A - Abstraction



B - Connect Request



C - Connection Establishment

D - 2-way Communication

Figure 5

4.2. Reliability

A successful return from a 1-way send routine indicates that a message has been successfully delivered to the VAN. It does not guarantee that the message is actually delivered to its destination agent. If the router at the destination site is down, then the message cannot be delivered. At this point, the message may be queued for later delivery, it may be returned, or it may simply be discarded. The agent model does not specify which of these protocols to use. Similarly, if an agent is not present at the address produced by the router's translation function, then the message cannot be delivered. In either of these cases, one of the routers can catch the error and record the problem.

The bidirectional routines offer a greater degree of reliability. Successful return from these routines means that the message has been delivered successfully to the destination agents incoming message queue. It does not guarantee that the agent will actually read the message.

In our current implementation, we assume that sites do not go down, and that the translation function used by the router is accurate i.e. that agents may be instantiated at the sites specified in the agent configuration. At this time, we make no effort to recover from violations of these assumptions. In the future, we will relax the first assumption by allowing sites to go down temporarily without loss of message traffic. Pending messages to downed sites will be delivered when the site is operational again.

4.3. Global and Local Information Requirements

A group of routers needs two pieces of global information in order to cooperate on an application. They are

- ***Agent Configuration.*** In general, we require that a router must know where all agents may be instantiated or have the ability to find out.

- ***Communication Link.*** All routers on an application must agree on a common communication port upon which to listen for messages.

In our current implementation, the router also requires one local piece of information, namely the location of the cached copies of executable code for all agents that the agent configuration has specified as executable at the local site.

4.4. Naming

For correct operation, agents must be named in such a way that there are no conflicts within a particular application or between different applications. It turns out that this is not too difficult to enforce.

Since agents are disk-resident and must appear in the same directory on the file system, each type must have a distinct identifier (the choice of *meaningful* identifiers is left to the developer). Agents of the same type located at separate sites have distinct names when one properly views their names as a concatenation of type identifier and site (e.g. MatrixMultiply@ruby.cs.virginia.edu). Selection of a unique port for each application and unique directories for each application at a particular site forestalls any name conflicts between applications. Thus it is fine for two applications to have an agent called ParseInfo, since the agents for each application are in separate directories and the associated routers are listening on different ports.

In summary, unique naming is handled with a combination of developer decisions, file system semantics, and the naming protocol of the Internet. The (type, site, port, directory) tuple uniquely specifies all agents for all applications.

To illustrate the formulation of computations using agents as the building blocks, we now present an example.

5. An Example - Passive Awareness

In this section, we describe a more complex distributed application. The context of this example is a passive system for information dispersal. This is commonly known as a Selective Dissemination of Information (SDI) system.

5.1. Description

The idea is that there is some group of interested parties that wish to share information. The information takes the form of various electronic documents e.g. mail, technical reports or news articles. The system must have the following capabilities:

- ***User registration.*** An interested user must be able to register his desire to see documents as they come available. Ideally, he would also specify the kinds of documents he is interested in, and the system would apply this user profile as a filter to incoming documents.
- ***Document entry.*** Members of the system must be able to enter documents that they wish to make available.
- ***Document archiving.*** Entered documents should be archived for some period of time so that they can be retrieved.
- ***User Notification.*** If a document 'matches' with a particular users profile, then that user should be notified of the documents existence.
- ***Document acquisition.*** Once notified that a document exists, a member should be able to acquire the full text of the document.

We call this a passive system because members do not have to do a periodic, active search of the database to see what new documents have been entered in the system. The documents are the active components, finding their way to interested users. We have deliberately avoided some of the interesting information retrieval aspects of the application (for instance, how exactly to compare a document with a user profile to determine whether the user is interested in the document) in order to concentrate on the functionality of the system and its expression in terms of distributed computational agents.

5.2. Solution

We express the solution to the above problem as two separate agent interactions, one that handles the user registration, and the other that handles document entry, archiving, and user notification. Document retrieval is a third agent interaction that we do not address here.

Both agent interactions are illustrated in Figure 6, user registration on the top and document entry, archiving and user notification on the bottom.

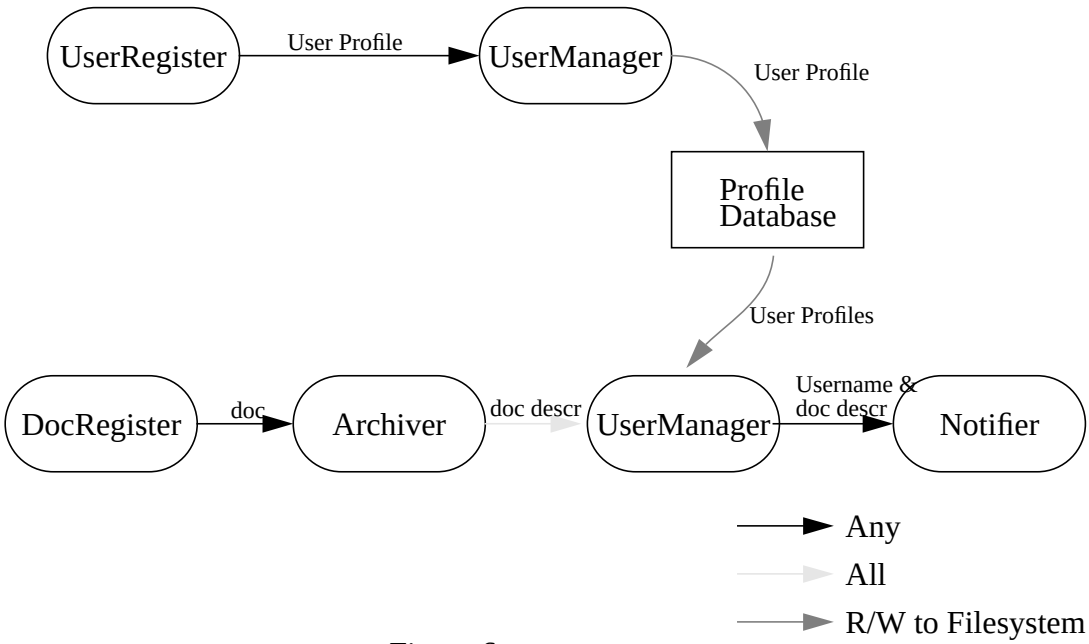


Figure 6

To register, a UserRegister agent (invoked by the user) takes a user's name, location, and a description of the user's interests and does a *SendAny()* to a UserManager. There may be several UserManager agents throughout the VAN, and each site maintains only a portion of a database of user profiles. When a document comes on-line, the UserManager at a particular site can then compare the document to the site's portion of the

user profiles. As an aside, we note that because UserRegister does a *SendAny()*, there is no redundancy in the distributed database, at least as we have formulated it here. The user profile is sent to a UserManager at only one site.

When a member desires to contribute a document, he invokes DocRegister. DocRegister takes the document and sends it to an Archiver. The Archiver's job is twofold, it creates a description of the document that is suitable for user profile comparison, and it archives the document locally in a known location. The Archiver then sends the document descriptor to UserManager agents at all sites that run UserManagers by doing a *SendAll()*. Since the user profiles are distributed, *SendAll()* must be used to ensure that all members have a chance to see the document. When the UserManager receives a message from an Archiver, it knows that it is getting a document descriptor and must compare that document against its portion of the user profiles. If the document and profiles 'match', then the UserManager sends the document descriptor and the matching user's name and address to a Notifier. The Notifier can then (somehow) let the user know that the document is there.

As this example demonstrates, it is possible for a receiving agent to get messages from more than one kind of agent, and in fact there is no constraint in this regard. The message router ensures that any properly addressed message will be delivered if that site is operating at the time of delivery. Of course, it is up to the programmer to ensure that the receiving agent knows what to do with the message once it has arrived.

6. Summary

The Virtual Agent Network concept provides a conceptual framework within which developer's may construct a distributed application. Such an application can be viewed as distributed in at least two ways. First, any particular computation can be split into components and performed by agents at different sites. Secondly, when the components of

a large application are represented by agents and the agents are spread over many sites, then the application itself can be considered distributed.

The routines described in Section 3 provide concrete tools with which a developer can implement a distributed prototype. Because agents are disk resident, they do not take up process table space, memory, or CPU time except when they are executing. It also means that agents can be written in any programming language that has an interface to the agent library routines.

The VAN model is well-suited for loosely-coupled systems like a LAN running on Ethernet. Because workstation to workstation communication is costly, agents are best formulated to perform fairly large-grained computation, though there is no conceptual problem in having fine-grained agents. There is also no conceptual problem in running the VAN model on a wider area network, and in fact our current implementation will work in such a situation with no modification.

7. References

Tanenbaum, AS and R. van Renesse. 1985. “Distributed Operating Systems”. Computing Surveys 17(4): 419-470.

Mullender, S. 1989. *Distributed Systems*. ACM Press, New York, NY.