Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

AN AGENT-BASED FRAMEWORK FOR LINKING DISTRIBUTED SIMULATIONS

Linda F. Wilson
Daniel Burroughs
Jeanne Sucharitaves
Anush Kumar

Thayer School of Engineering
8000 Cummings Hall
Dartmouth College
Hanover, NH 03755–8000, U.S.A.

ABSTRACT

Simulations often operate on static datasets and data sources, but many simulations would benefit from access to dynamic data. This paper describes our work developing a software agent-based framework for dynamically linking distributed simulations and other remote data resources. The framework allows independently-designed simulations to communicate seemlessly with no *a priori* knowledge of the details of other simulations and data sources. In this paper, we discuss our architecture and current implementation developed using the D'Agents mobile agent system. To demonstrate the feasibility of our system, we present a prototype for a hypothetical search and rescue mission.

1 INTRODUCTION

Operational simulations are typically designed to use static datasets and data sources. Many simulations would produce more-accurate results if they could access dynamically-changing data from other sources. Furthermore, many interactive simulations require dynamic data, possibly from multiple sources. From the perspective of one simulation, other simulations are data resources, producing information possibly relevant to the past, present, or future of the system being modeled.

This paper presents the fundamental framework for using software agent technology to link distributed simulations. Specifically, we use software agents to coordinate distributed operational simulations and efficiently communicate data between simulations and other data resources. Such simulation agents will allow simulations to enter and exit a global simulation "cloud" (Figure 1) asynchronously without requiring recompilation and constant re-coordination among all participating sites and datasets. The networked

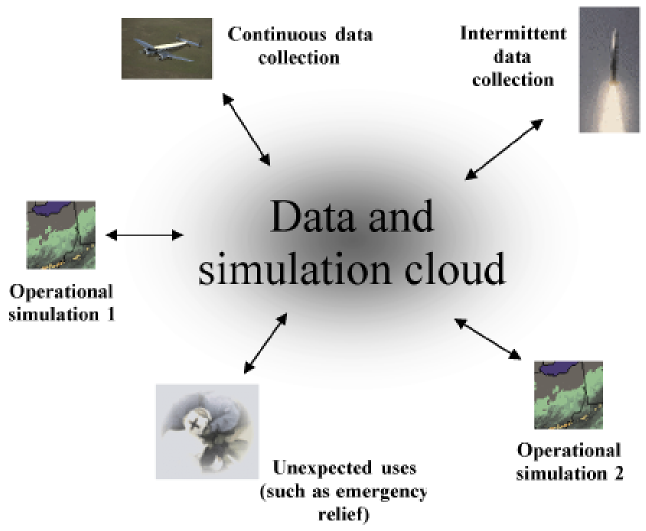
Continuous data collection
Intermittent data collection
Data and simulation cloud
Operational simulation 1
Unexpected uses (such as emergency relief)
Operational simulation 2

Figure 1: Cloud Linking Various Simulation and Data Resources

data and simulation cloud consists of dynamically changing data and computational resources available on a network to one or more simulations. Note that the networked resources may consist of other simulations, datasets, active probes, and sensors.

Each simulation in the cloud is designed independently with little or no knowledge of the other simulations in the cloud. While a simulation must be able to specify what resources it needs and what it provides to the cloud, it does not need to know any specifics about the other simulations (e.g. language, file formats, etc.). Instead, our agent-based framework provides the interface for linking the simulations.

Simulations and other resources may join or leave the cloud at any point in time. For example, consider a scenario in which some of the communication within the cloud occurs via a wireless network. A forest fire simulation could communicate with various sensors out in the field as well as with a weather simulation running at a remote site. Naturally, sensors located in hostile environments may communicate sporadically with the rest of the network.

In this paper, we describe the software architecture for our system and discuss a prototype example involving a hypothetical search and rescue operation. A discussion of some of the challenges involved with this system can be found in Wilson, Cybenko, and Burroughs (1999).

2 RELATED WORK

Perhaps the greatest challenge in this project lies in the desire to have multiple independently-designed simulations be able to comprehend the data that they are passing back and forth between one another. In describing a framework for communications between intrusion detection systems, Kahn et. al. state three conditions that must be met for strong interactivity between independently-developed systems.

1. Configuration interoperability, which refers to the ability of two systems to discover one another and communicate data back and forth.
2. The ability to parse the data being transferred (e.g. agree on data types, byte ordering, etc.).
3. Inter-comprehension, or agreement on the meaning and definition of the data descriptors.

When all of these conditions are met, it is possible for two independently-developed systems to interact closely even though they were designed independently.

One approach to solving this problem is to require that all participants in the simulation cloud be designed to a strict specification. This is the approach used in the High Level Architecture (HLA) (Dahmann, Fujimoto, and Weatherly 1998). As described in various specification documents on the HLA web page, HLA defines the structure of objects in the simulation, grouping of objects, and the communication between these objects. This leads to a set of highly reusable and interactive simulation objects. However, the disadvantage of such a system is that all participants in the system must be written to meet the specifications. While a well-designed specification will not pose a great burden to the development of new software, rewriting old software to a new specification may be undesirable or impossible.

Another approach to this problem is to require that the inputs and outputs of the system be described in a predefined manner, without any specifications for the internal operation of the software. The advantage of this approach is that existing systems could have their data translated into this common format without a great deal of modification to the existing software. In order to accomplish this, a robust system for communicating and describing data is required.

Knowledge Interchange Format (KIF), a development of the Logic Group at Stanford as part of the ARPA Knowledge Sharing Effort, is an attempt to develop this. KIF is based on first order logic with extensions to support non-monotonic reason and definitions (Finin et al. 1994). As described in the draft proposal, KIF is designed for the interchange of knowledge in disparate computer systems. It is not designed as a human interaction language, nor is it designed to be the internal representation of data within a system. Instead, KIF is designed to facilitate the independent development of software that will eventually communicate.

Another product of the ARPA Knowledge Sharing Effort is KQML (Finin et al. 1994). This is described as a “language that is designed to support interactions among intelligent software agents” (KIF draft proposed standard). KQML is concerned with knowing who to talk to and how to maintain a conversation. This includes the ability to initiate a conversation. It is designed to control the structure on the conversation while tools such as KIF define the language of the conversation. It enables programs to identify, connect to, and exchange information with other programs. However, the meaning and description of the data is not necessarily defined by KQML.

3 FRAMEWORK ARCHITECTURE

3.1 Overview

We have developed an agent-based software framework to facilitate the dynamic exchange of data between distributed simulations and other data resources. The goals of this framework are to allow existing simulations to join the cloud with a minimal amount of code modification and to provide a system where simulations can interact without any *a priori* knowledge of each other’s interfaces.

The architecture of our system consists of four basic components: user objects (e.g. simulation entities), generic local agents, mobile helper agents, and a broker agent. The user objects are often described as simulations but can actually be any producers or consumers of data. For example, a sensor that generates data used as input to a simulation would be a data producer. Visualization tools that are used to collect and display output of various simulations would be consumers. Simulations, of course, can be both producers and consumers of data. The generic local agent (GLA) is a user object’s interface to the simulation cloud, and all communications and commands are routed through the object’s GLA. Thus, the user object needs to communicate directly only with its GLA. Within the simulation cloud, the broker establishes the necessary links between the user objects by connecting their GLAs. Finally, the helper agents (HAs) are mobile agents which are used to minimize the network traffic by performing computations at the data source. The various components in this system are shown in Figure 2 and described in detail in the following subsections.

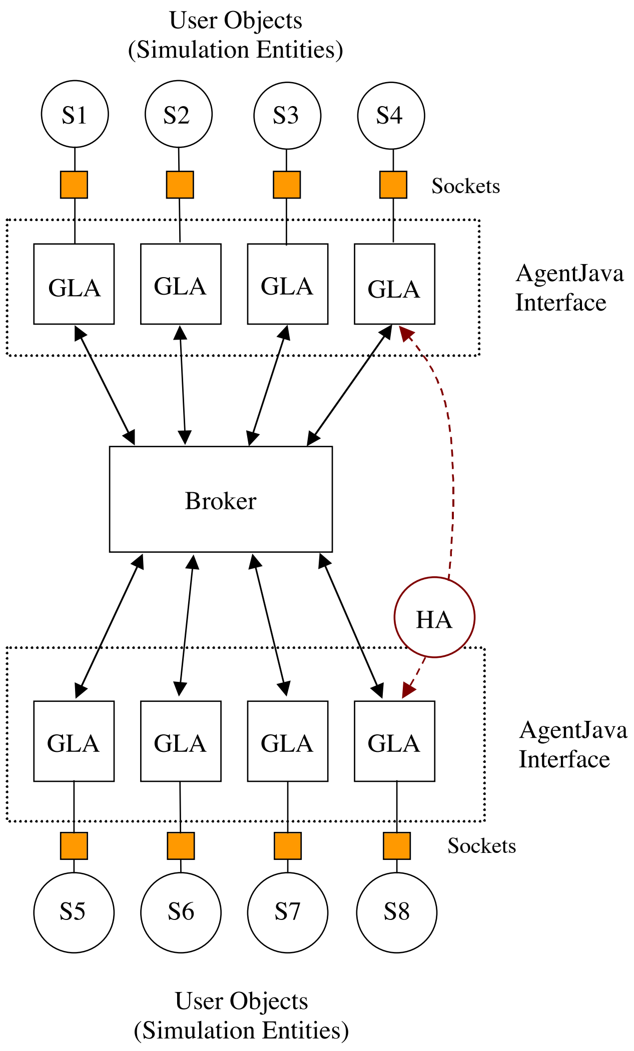

Figure 2: Basic Framework Connecting the Components in the Cloud

3.2 User Objects and GLAs

In order to participate in the simulation cloud, a user object (such as a simulation) needs some basic functionality. At a minimum, the user object must be able to connect to and disconnect from the simulation cloud. Data producers will also need to advertise their services to the outside world. Data consumers will need to be able to discover advertised services and invoke those services. Other functionality includes the ability to generate and use helper agents and respond to changes in the availability of other user objects in the simulation cloud.

The basic functionality is provided to the user object through its generic local agent (GLA). The GLA is the user object's sole connection to the simulation cloud, and there is a one-to-one relationship between a user object and a GLA. The user object, through commands sent to and from the GLA, has the ability to connect to the cloud, disconnect from it, advertise its services, and look up available services. Essentially, the GLA is the front end of the user object. However, the user object must be able to communicate with its GLA. This communication capability may be built into the simulation or provided via a separate interface. Our prototype system described in Section 5 discusses the use of such an interface.

The architecture of the GLA can be described with three main components: the agent communication system, the command interpreter, and the dynamic command table. (These components are shown in Figure 3, which is discussed in Section 5.) The communication system controls all communication between the GLA and other objects in the simulation cloud, including the local simulation, other GLAs, and the broker. We use sockets for communication since they provide a platform- and language-independent method for communication.

Any incoming messages are passed to the command interpreter, which processes the messages and carries out their instructions as appropriate. Messages fall into two general categories: broker interaction commands and execution commands. Broker commands cause the GLA to interact with the broker, either to advertise or discover services. Execution commands cause the GLA to execute functions from the local simulation or invoke commands on a remote GLA. Broker interaction functions are built into the GLA, are static, and are identical across all GLAs. Execution commands, on the other hand, are dynamic in nature.

The abilities available to a GLA are determined by what services it has advertised to the simulation cloud or discovered from the broker. That is, the GLA knows what services or functions are provided by the local user object (simulation) and what remote services have been requested by the local simulation.

The dynamic command table provides a data structure for maintaining the necessary information regarding these abilities. This information includes the name of the service, a detailed description of the service, a description of the service's input and output parameters, and the location and execution method of the service. When a service is either advertised or discovered, its specifics are written into the GLA's dynamic command table. Once the service is listed in the GLA's table, the GLA can access that service without going through the broker.

For an advertised service (one provided by the local user object), the execution request will come from a remote GLA that needs information from the local service. For a discovered service (provided by a remote user object), the execution request will originate from the local simulation.

Finally, the GLA provides the ability for a user object to disconnect itself from the simulation cloud. The GLA of the disconnecting object reports this to the broker (who reports the disconnection to other remote GLAs) and then terminates itself. Note that the simulation may continue to run even though it no longer participates in the cloud.

In our current system, the GLAs support only blocking requests from a simulation. When a simulation passes an execution command to its GLA, the simulation must wait until this command has been completed before continuing. Alternatively, it must provide its own mechanism for generating the request in a separate thread and polling for the completion of the request. In future implementations, non-blocking requests will be allowed, and the GLA will be able to interrupt or otherwise inform the simulation when a request is completed. In certain situations, the GLA will also be able to prefetch data before it is needed by the simulation.

3.3 The Broker

The broker acts as a database for cloud participants and their advertised services. When a user object advertises a service to its GLA, the GLA reports this advertisement to the broker. Later, when a remote user object searches for a particular service, its GLA will query the broker. Upon finding a match for the query, the broker will inform the remote GLA of the name, location, parameter list, and execution method of the service. This information will be transfered to the remote GLA, who will write it into its dynamic command table. Once the remote GLA has this entry, it can directly access the local GLA of the service without any further communication through the broker.

In the case where a desired service is not found, it will be possible for a user object to tell its GLA to leave a standing request at the broker. If at some future time the service does become available, the broker will inform the appropriate GLA of this, and the GLA will in turn inform the simulation.

A similar approach is used when an incomplete service is found. If the broker later finds a better match, it will inform the GLA of this change in the state of the simulation cloud, and the GLA can begin using the new service.

If multiple matches are found for a query, the broker will inform the GLA of all possible matches so the GLA can determine which source is most appropriate. Furthermore, the GLA will be able to use a second source if the connection to the primary source is lost.

A central part of the brokering system is the ability to describe services accurately and unambiguously so that consumers and producers of data can be matched correctly. This difficult problem is fundamental to the desire to have various simulations and data resources interacting seemlessly through the cloud. We are currently investigating several paths for solving this problem, and we will report our work in future papers.

In order to be scalable and robust, the broker should be designed as a distributed system rather than as a large central database. For example, a group of user objects located at one facility could have their own broker. This broker would know the locations of other brokers, and upon receiving a request which it could not fulfill, it would simply pass this request to another broker. This system is similar to domain name server (DNS) lookup, in which a higher-level DNS is queried only when a query to the local DNS fails.

3.4 Helper Agents

The final component of the simulation cloud is the helper agent, which is used to extract or generate the desired data while minimizing network traffic. Since a producer does not necessarily know the exact needs of its consumers, it is unrealistic to expect the producer to provide all possible forms of its data. Thus, data may need to be collected from the producer and then manipulated to fulfill the needs of the local consumer.

For example, suppose one simulation needs the average value of a set of data provided by a remote producer. One possibility is to send all of the data from the remote source to the simulation and then compute the average. This is inefficient in terms of network usage since it requires a large amount of data to be transferred when only a small amount (i.e. the average) is required. A better approach would be to perform the averaging calculation at the data source. In order to do this, we introduce the helper agent.

In our software architecture, a helper agent is a mobile agent capable of traveling to a remote location and executing there. Through its GLA, a simulation may create a helper agent which travels to the remote GLA in order to preprocess the data before transferring it over the network. From the remote GLA's viewpoint, the fact that the helper agent has moved itself across the network is invisible. The helper agent and the GLA communicate over a socket just as if they were still on remote machines. In fact, the helper agent issues the same commands to the remote GLA that would have been sent from the originating GLA. Once the helper agent collects the results, it processes them and then returns the final result across the network. Figure 2 shows a helper agent transferring data between the GLAs for simulations S4 and S8.

The two primary goals of this architecture are to allow existing simulations to join the cloud with a minimal amount of code modification and to provide a system where user objects can interact without any *a priori* knowledge of each other's interfaces. By using a standard, platform and language-independent protocol, sockets, we are able to provide an interface to which it is relatively easy to connect. Furthermore, by decoupling the user objects from one another, there is a great deal of flexibility in the system. A user object is not required to know anything about the other user objects; it needs only to talk with its local GLA, and all further communication is handled through the agent system.

4 CURRENT IMPLEMENTATION

We have developed an initial implementation to demonstrate the concept of using an agent-based system to exchange data dynamically between simulations. This section describes the basics of our implementation, while the next section discusses a search-and-rescue prototype which demonstrates the operation of our system.

We are using Agent Java from the D'Agents system (D'Agents web page, Brewington et al 1999). Our current implementation provides basic functionality for the generic local agents (GLAs). Implementation of the broker system is part of our future work.

As discussed earlier, every simulation has its own GLA. Each GLA is developed as a Java application providing end-to-end connectivity with certain built-in functionality, and the GLAs communicate with each other through sockets. Note that sockets shield the programmer from the low-level details of the network, like media types, packet sizes, packet re-transmission, network addresses, and many other lower-level implementation details. Most importantly, sockets allow platform-independent communication.

At a minimum, each GLA provides five standard functions that are invoked through commands sent from a simulation to its GLA. The commands are as follows:

- **Lookup**: With this command, the GLA takes the function name and function description (which can possibly include expected input and output parameters) from the simulation and sends this information to the broker. The broker searches the simulation cloud to find a match and then returns the contact information for the entity (simulation, sensor, etc.) that provides the requested service.
- **Advertise**: This command is used by an entity in the simulation cloud to advertise its producer capabilities. When an entity invokes this command, it supplies the name of the function or service it can execute along with a description of the service.
- **Execute**: A consumer in the simulation cloud invokes the execute command to initiate a request for data. The consumer provides the local function name and description to its GLA. If a match is found in the GLA's table, the request will be processed by the appropriate producer and the results will be channeled back to the consumer via the respective GLAs. If a match is not found, the GLA contacts the broker to find a match. If no match is found by the broker, the GLA tells the consumer that it will not be able to satisfy the request.
- **Table**: The table command is used to access the contents of the dynamic table data structure maintained by each GLA. This table is an integral part of the system because it is involved in the execution of each command. When a user object or entity first advertises its capabilities to the simulation cloud, the capabilities are recorded in the table by its GLA. The table entry includes the function name and location, where the location is "local" for an advertisement. When a lookup command is invoked by an entity, the broker checks the tables of the other GLAs in the cloud, finds a match, and sends the appropriate information to the requesting GLA. In this case, the table entry includes the local function name, the location of the service, and the remote function name of the service. to be invoked) in the table. Upon encountering the execute command, the GLA checks the table to see if a match already exists; if it does then the request is channeled to the location specified in the table. Otherwise, the broker tries to find a match.
- **Disconnect**: This command is used by an entity to exit from the simulation cloud. The GLA informs the broker of the disconnection and then terminates itself. Note that the simulation may continue to execute after it leaves the simulation cloud.

The agent system has two communication interfaces: the internal interface between each GLA and the broker, and the external interface between the GLA and its simulation or simulation interface. Java sockets are used to implement both the internal and external communication interfaces. With different simulations entering the cloud from different platforms and using various programming languages, the platform-independent Java socket provides the communication flexibility for interoperability.

Communication error control is handled through the use of acknowledgments. When the simulation sends the first command to the GLA, the GLA validates the command and sends back a "valid" acknowledgment. Once the simulation receives an acknowledgment, it sends other information such as the function name and function description. All of the commands and descriptions are acknowledged individually.

5 SEARCH AND RESCUE EXAMPLE

5.1 Description

Suppose that a Coast Guard search and rescue unit receives calls for help on an unpredictable (random) basis. Each call indicates the estimated location and time of an accident. The Coast Guard needs to obtain a forecast of the trajectory of the survivors. The information must be sufficient to dispatch a vessel from one or more locations and deploy a search pattern in and around the predicted trajectory. Furthermore, the temperature history along that trajectory is needed to determine the likelihood of survival. Since the probability of survival depends on the number of accumulated degree-hours, survivors must be intercepted before they accumulate too much exposure to cold.

Coincidentally, Dartmouth's Numerical Methods Laboratory (NML) provides an ocean forecasting service which archives the latest forecasts for ocean velocity and temperature. Given a suitable request in terms of estimated location and time, the service can compute a trajectory (i.e. location and temperature vs. time). This is exactly the service needed by the Coast Guard unit.

To demonstrate the basic idea of using a software agent-based system to exchange data between remote simulations, we developed a simple prototype for this search and rescue scenario. The Coast Guard simulation can be visualized as a client or consumer that is making requests, and the ocean forecasting service is the server or producer that is servicing these requests dynamically. The communication is facilitated by our agent-based system and is completely transparent to both simulations.

5.2 Implementation

For this prototype, we developed GLAs with basic capabilities and basic communication interfaces for the Coast Guard and ocean simulations. However, the broker which is responsible for linking the simulations has not yet been developed; thus, the broker's functionality is hard-coded in the prototype.

Our prototype consists of four basic components: the Coast Guard simulation, the ocean forecasting simulation, the NML server interface for the ocean simulation, and the individual GLAs. The components are shown in Figure 3 and described in the following paragraphs.

The Coast Guard implementation in this prototype is a Java application that enables the user to enter the commands to the GLA as if they were sent from a simulation. This application is similar to the interface that a simulation needs to implement to be able to join the simulation cloud and talk to its GLA. The Coast Guard simulation and its GLA are shown in the left side of Figure 3.

The ocean forecasting simulation was developed by Dartmouth's Numerical Methods Laboratory (NML) without any knowledge of our simulation cloud or GLAs. By itself, the simulation knows only to take an input file containing the required parameters and generate the corresponding trajectory. Thus, we created a simple interface called the *NML server* to handle communication betwen the ocean simulation and its GLA. The NML server interface receives the request from the GLA, creates the formatted input file needed by the ocean simulation, and sends the file to the simulation to execute. The interface then waits for the result and sends it back to the GLA. Since the NML server is necessary for the ocean simulation's participation in the simulation cloud, we consider the "user object" to be the ocean simulation and its NML server interface. The right side of Figure 3 shows the ocean simulation/NML server combination and the corresponding GLA.

In our prototype, we can visualize the GLA acting as a client or a server, depending on the command issued, using sockets to listen to port 1610. The GLA listens to this port for incoming requests from consumers and also for processed results from producer entities. A single listening socket is used to handle requests from the local simulation object and from other objects internal to our system (e.g. other GLAs, brokers, etc.). The socket can therefore be viewed as a pipe transporting streams of data between various GLAs and the entities. The NML server listens to port 2812 for incoming requests from its GLA, communicates with the simulation, and sends the required result set back to the GLA, who then sends it to the remote GLA that needs the data. The port numbers are chosen arbitrarily.

During an advertise operation, the GLA takes the function name and description of the service the simulation would like to provide and stores it in the dynamic function table with a "local" service location. In Figure 3, the ocean simulation advertises that its service named Proc1 takes parameters X, Y, and Z and returns an output T. Thus, the local service Proc1 is listed in its GLA table. In future implementations containing a functional broker, the advertise function will be responsible for sending this information to the broker to store in the broker's database for further lookups.

During a typical lookup operation, the consumer gives its GLA the description and local function name (to be used in the GLA's table), and the GLA calls the broker's lookup method to find a match. In this prototype, the broker's lookup method returns hard-coded information on the producer's location (i.e. we have simulated the broker's actions). Given the producer's information, the GLA enters the local function name, remote service location, and remote function name into its dynamic function table. In Figure 3, the Coast Guard simulation requests information for three services that it labels Func1, Func2, and Func3. Matches for these services were apparently found since all three are listed in the GLA table. Note that the Coast Guard GLA's table shows the local name for each service along with the remote location and remote name for each service. Thus, the first entry confirms that Func1 is executed by calling Proc1 on the NML machine. The entries for Func2 and Func3 are for demonstration purposes only; they were not used in the prototype.

When an execute operation occurs, the GLA takes the function name and description, looks it up in the dynamic function table, and continues only if the function exists in the table. If the function is a local service, the GLA will

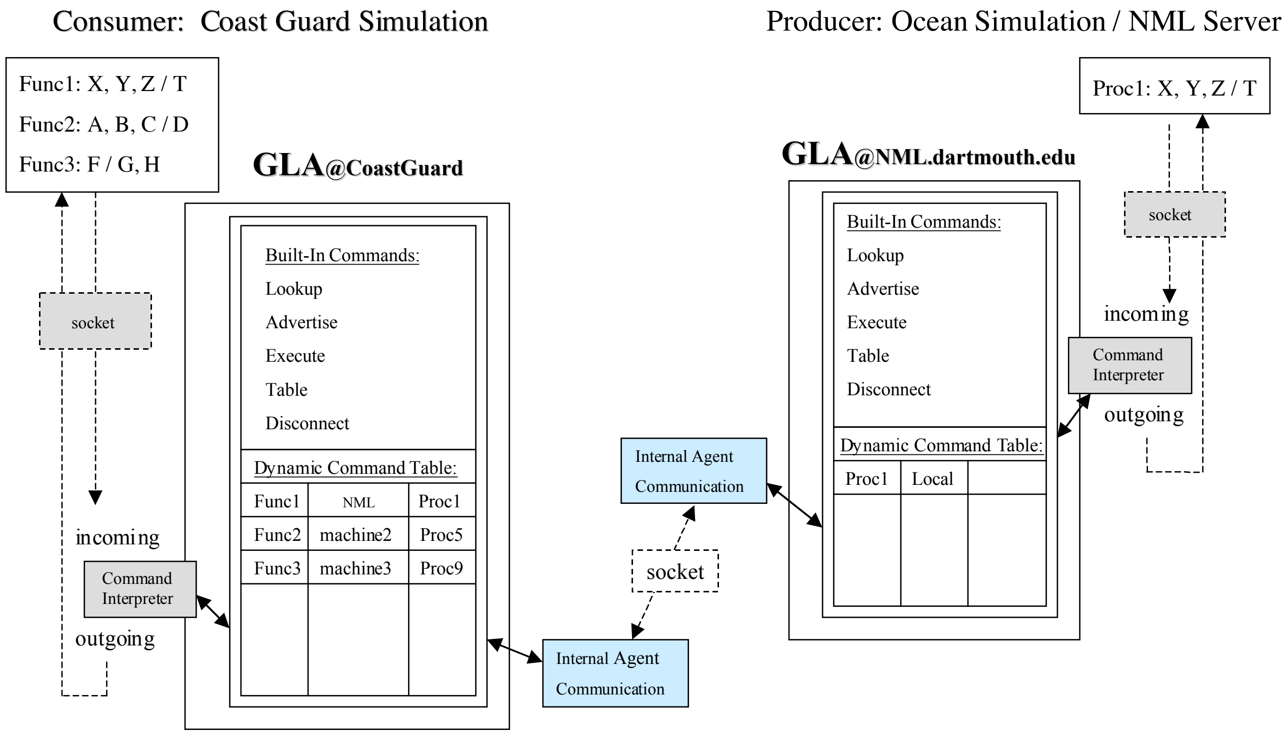

Figure 3: Search and Rescue Prototype Implementation

open a connection with its local simulation or interface, send the information, wait for the result, and send the result back to the enquirer. In Figure 3, a request to the ocean simulation GLA to execute Proc1 will result in execution of the local Proc1 service. If the function is a remote service, the GLA will open a connection with the appropriate remote GLA object and ask that GLA to execute this function. In Figure 3, a request to the Coast Guard GLA for Func1 will cause a request to the ocean simulation GLA for Proc1.

A sample run of our prototype can be visualized as follows:

1. The Coast Guard simulation and the ocean simulation/NML server enter the simulation cloud. The GLAs are activated and wait for incoming requests.
2. The NML server performs an advertise operation to inform its GLA of its capabilities. Information on Proc1 is stored in the GLA's table.
3. The Coast Guard simulation performs a lookup operation for the function that it will need to execute when a rescue call occurs. The broker returns the name and location of the required procedure that matches the criteria, and this information (i.e. the NML location, etc.) is entered into the Coast Guard GLA's table. Note that Func1 is the Coast Guard's name for the service named Proc1 located at the NML machine.
4. The Coast Guard simulation then issues an execute operation whenever it needs information from the ocean forecasting simulation. The GLA verifies that the required functionality is present, and the Coast Guard simulation gives its GLA the necessary parameters to be passed to the ocean simulation.
5. The Coast Guard's GLA then sends the data provided by the Coast Guard simulation to the ocean simulation's GLA.
6. The ocean simulation's GLA receives the execute command and parameters from the Coast Guard's GLA and sends the information to the NML server which acts as an interface between the GLA and the ocean simulation.
7. The ocean simulation receives this information from the NML server and runs the required executable to produce the output data file.
8. This output file is then sent to the ocean simulation's local GLA, which then sends it to the Coast Guard's GLA, which in turn sends it to the Coast Guard simulation.
9. The Coast Guard simulation uses the trajectory information to find and rescue the survivors.

Using a real-world scenario, our prototype demonstrates a transparent, dynamic, and real-time integration of simulations using an agent-based framework.

6 CONCLUSIONS

We have presented the framework and initial implementation of our agent-based system for linking distributed simulations. Our simple prototype, though incomplete with respect to the development of all the components, demonstrated the effectiveness of using an agent framework to link simulations together dynamically at run time. In addition, it showed that an existing simulation written with no knowledge of our system can be added to it.

In order to enter the simulation cloud, an entity must have an interface to the GLA. This interface is the single component which must be customized at the end-user level. We have kept this interface as simple as possible so that minimal effort will be required on the entity's part to use our system. For the ocean forecasting simulation, we added a simple server interface to enable communication between the ocean simulation and its GLA.

In the traditional approach involving interaction between different entities, any simulation wishing to communicate with another simulation or data provider must know in advance where and when it will be required to do so. In many cases, such communication requires that the participating simulations be written to meet a given standard. In addition, a considerable amount of network bandwidth is typically used in order to transfer data from one entity to another.

Our approach allows an entity to request data dynamically at any time through its local GLA. The broker will facilitate the matching and the resulting data will be transferred to the requesting entity. Although our system requires some overhead in the form of the broker and local GLAs, this overhead places little burden on the CPU and memory systems. The network bandwidth used will be comparable to the traditional approach, and the performance of our system will increase when a mobile helper agent is used. The fact that the simulations can request information on the fly, without being aware of the location of the provider, overrides the minimal overhead incurred by the simulation cloud.

When it is developed, the broker will play the central role in making the system dynamic at run time. We are currently investigating various approaches to develop this module and also the use of a universal description language by which every entity can describe what it produces and consumes without any ambiguity.

Our agent-based framework for linking distributed simulations opens new avenues of possibilities to dynamically integrate simulation systems and data providers at run time. We look forward to developing more-advanced implementations that include a full-fledged broker agent, sensors as data producers, etc.

ACKNOWLEDGMENTS

This work is supported by National Science Foundation KDI Grant 9873138. We would like to thank Daniel R. Lynch and Miroslaw (Mirek) Kiczko for their work with the ocean forecasting simulation.

REFERENCES

Brewington, B., R. Gray, K. Moizumi, D. Kotz, G. Cybenko, and D. Rus. 1999. Mobile agents in distributed information-retrieval. In Matthias Klusch, editor, *Intelligent Information Agents*, Spring-Verlag.

D'Agents Web Page. `<http://agent.cs.dartmouth.edu>`

Dahmann, J. S., R. M. Fujimoto, and R. M. Weatherly. 1998. The DoD high level architecture: An update. *Proceedings of the 1998 Winter Simulation Conference*, pp 797–804.

Finin, T., R. Fritzson, D. McKay, and R. McEntire. 1994. KQML as an agent communication language. *Proceedings of the Third International Conference on Information and Knowledge Management*, pp 456–463.

High level architecture web page. `<http://hla.dmso.mil>`

Kahn, C., P. Porras, S. Staniford-Chen, and B. Tung. A common intrusion detection framework. Submitted to the *Journal of Computer Security*.

Knowledge interchange fFormat, draft proposed American National Standard (dpANS) NCITS.T2/98-004. `<http://logic.stanford.edu/kif/dpans.html>`

Knowledge sharing effort web page. `<www.cs.umbc.edu/kse/>`

Wilson, L. F., G. Cybenko, and D. Burroughs. 1999. Mobile agents for distributed simulation. *Proceedings of the High Performance Computing Symposium (HPC '99)*, pp 53–58.

AUTHOR BIOGRAPHIES

LINDA F. WILSON is the Clare Boothe Luce Assistant Professor of Engineering at Dartmouth College. She received her BS degree from Duke University and MSE and PhD degrees from the University of Texas at Austin. Her email and web addresses are `<Linda.F.Wilson@dartmouth.edu>` and `<http://thayer.dartmouth.edu/~lwilson>`.

DANIEL BURROUGHS is a PhD student at Dartmouth College. He received his BS degree from the University of Central Florida in 1995. His email address is `<Daniel.Burroughs@dartmouth.edu>`.

JEANNE SUCHARITAVES is a master's student at Dartmouth College. She received her AB degree from Dartmouth College in 1999. Her email address is `<Jeanne.Sucharitaves@dartmouth.edu>`.

ANUSH KUMAR is a master's student at Dartmouth College. He received his BE degree from Venkateshwara College of Engineering (University of Madras, India) in 1999. His email address is `<Anush.Kumar@dartmouth.edu>`.