Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

THE ABELS SYSTEM: DESIGNING AN ADAPTABLE INTERFACE FOR LINKING SIMULATIONS

G. Ayorkor Mills-Tettey
Greg Johnston
Linda F. Wilson
Joseph M. Kimpel
Bin Xie

Thayer School of Engineering
Dartmouth College
Hanover, NH 03755-8000, U.S.A.

ABSTRACT

The Agent-Based Environment for Linking Simulations (ABELS) provides a framework to facilitate the dynamic exchange of data between distributed simulations and other remote data resources. Specifically, the framework allows the formation of a dynamic "data and simulation cloud" that links a heterogeneous collection of networked resources. ABELS consists of three major components: user entities that serve as data producers and/or consumers, a brokering system for organizing and linking the various participants, and generic local agents that connect simulations and data resources to the cloud of participants. This paper describes the major redesign and implementation of the generic local agent, which serves as the adaptable interface between the user and the ABELS system.

1 INTRODUCTION

Many simulations and other applications need dynamic access to a variety of data resources. For example, a simulation predicting the severity of a flood needs rainfall and weather predictions (from weather simulations), current water levels (from sensors), and information regarding the existing drainage infrastructure (from databases). Researchers in a particular field (e.g., medicine or arctic research) may form a consortium to exchange data and provide services.

In the traditional approach involving interaction between different entities, any simulation desiring to communicate with another simulation or data resource must know in advance where, when, or how it will be required to do so. Thus, simulations typically must be hardwired to specific resources or developed according to a certain standard. This traditional approach is not always desirable or even feasible.

We have developed a framework for using software agent technology for linking distributed simulations and other data resources. The Agent-Based Environment for Linking Simulations (ABELS) framework uses software agents to coordinate distributed simulations, communicate data efficiently between simulations, and retrieve data from other sources such as sensors and datasets. ABELS allows simulations to enter and exit a global simulation "cloud" consisting of dynamically changing data and computational resources. 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 services it provides to the cloud, it does not need to know any specifics about the other simulations. The simulation also does not need to be written to a particular specification; instead, our agent-based framework provides the interface for linking the simulations and adapts to their needs.

Data resources may join or leave the cloud at any time. The simulations and other resources do not need to be perfectly synchronized in time, but the resources needed by a particular simulation must generally cover the same time frame or have been previously generated for the appropriate time period. Currently, we are working with loosely-coupled simulations, but we will add synchronization mechanisms for tightly-coupled systems at a later date.

Previous papers (Kumar et al. 2002, Sucharitaves et al. 2002, Wilson et al. 2001) described the ABELS framework and its primitive prototype implementation. This paper describes the current design and implementation of the interface between resources and the cloud which links and coordinates the resources. This work represents a major redesign of some of the ABELS components, and it provides examples that demonstrate the flexibility of our framework in adapting to the needs of users.

2 RELATED WORK

The ABELS framework is not the first that is designed to allow information exchange between autonomous entities that benefit from shared information. Two notable architectures that have similar goals to the ABELS system are the *High Level Architecture* (HLA) and the *Web Services* architecture. In this section, we discuss these two frameworks and their relationship to the ABELS system.

The *High Level Architecture* (HLA) allows tightly-coupled simulations that conform to a set of rules and share a common Federation Object Model (FOM) to interact at runtime using a common runtime infrastructure (RTI) (Dahmann 1998). In the HLA, all individual simulations, known as federates, must conform to the Federation Object Model (FOM) in order to exchange information with other federates. This requires legacy systems to be rewritten to conform to the HLA standard, and does not easily facilitate spontaneous information exchange between simulations, since the Federation Object Model must be agreed upon before the formation of the federation. The ABELS system described in this paper targets information exchange between loosely-coupled information producers and consumers, including entities that do not share a data representation format.

There are other architectures being developed that also allow a more loosely-coupled communication paradigm between participating entities. The *Web Services* framework, backed by industry players such as Microsoft, IBM, and Sun, is an emerging framework for application-to-application interaction built on existing Web protocols and open XML standards (Curbera et al. 2002). In the *publish-find-bind* paradigm employed in the web services architecture, businesses *publish* their services to a directory system where other businesses can then *find* these services and *bind* to them in order to use them (Knutson and Kreger 2002). The three protocols that form the backbone of the current web services architecture are the Simple Object Access Protocol (SOAP), the Web Services Description Language (WSDL), and the Universal Description, Discovery and Integration (UDDI) protocol.

SOAP (Mitra 2001, Gudgin et al. 2001a, Gudgin et al. 2001b) is an XML-based protocol that provides a platform- and language-independent means of remote method invocation. To use a remote web service, a program sends a SOAP request specifying the method to invoke, as well as input and output parameters, to the remote web service, often over HTTP. The web service performs the required action or computes the desired result, and sends back a SOAP response encapsulating the reply.

In order to be able to successfully send a SOAP request to a web service, a client needs to know what methods the web service supports, what the expected input and output parameters are, the location of the service, and the protocols it understands. This is achieved by having businesses describe their services using the Web Services Description Language (WSDL) (Christensen et al. 2001). The WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. In the WSDL, the operations and messages are first described abstractly and then bound to a specific network protocol and address.

The final component in the web services equation is Universal Description, Discovery, and Integration (UDDI), which defines a means of publishing and discovering information about web services (McKee et al. 2001). UDDI defines the interface exposed by a distributed business registry that stores business registrations written in a common XML format. The information included in the registration includes address and contact information, industrial categorizations based on standard taxonomies, and technical information about services that are exposed by the businesses. This technical information includes specifications which can be WSDL descriptions of supported web services. In a typical usage scenario, programs and programmers use UDDI to locate information about services, and a programmer would then use this information to create client programs that use the published web services, or would use UDDI to publish web services that can be used by others.

The web services architecture is an important step in the direction of enabling interoperability between services provided by different businesses and organizations. It reduces the time required to create programs that use the services of other businesses by eliminating the need to design custom data exchange protocols. This in turn enables the re-use of services as components in creating more complex services provided to an end-user. Legacy services could conceivably utilize the web services architecture by creating SOAP wrappers that translate SOAP requests and responses to and from the particular protocol that is understood by the service.

However, an important functionality that is not directly supported by the web services architecture is that of runtime brokering or matching of clients and services. The idea of runtime brokering of clients and services shifts focus from the concrete implementation of a service to the abstract service or information that is provided. It allows the transparent replacement of one service provider with another that provides similar functionality, without having the services be written to conform to a specific standard. It is this runtime brokering and matching of clients and services that forms the core of the ABELS system described in this paper.

3 ABELS OVERVIEW

The ABELS framework uses software agents and a brokering system to allow independently designed, distributed simulations and data resources to communicate with each other with

no *a priori* knowledge of the implementation details of other simulations or data resources. As shown in Figure 1, the ABELS system architecture consists of user entities, generic local agents (GLAs), and a brokering system.

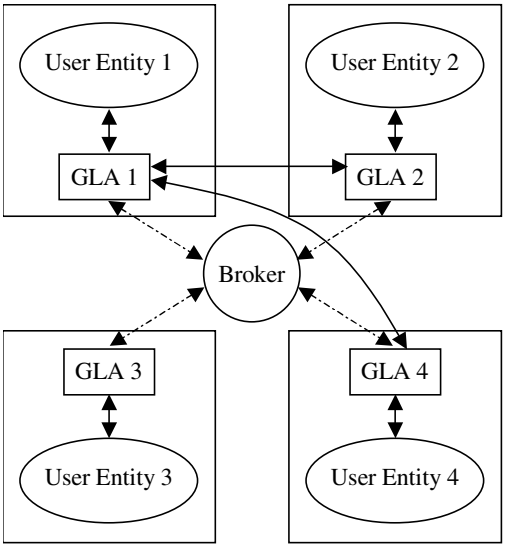

Figure 1: Basic ABELS Framework

The user entities are often described as simulations but can actually be any producers or consumers of data. For example, a sensor that generates data would be a data producer. Visualization tools that are used to collect and display the output of various simulations would be consumers. Simulations, of course, can be both producers and consumers of data. In general, we say that consumers make requests or queries for data and producers provide services that generate data.

The generic local agent (GLA) serves as the interface between a user entity and the data and simulation cloud. It allows a producer to describe its services, register the services with the cloud, and provide services as resources for others. Similarly, it allows a consumer to make requests for resources, send the request to the brokering system for matching, and connect to the corresponding producers to use those services of interest. Furthermore, the GLA handles any necessary data format and unit conversions. Although the GLA is implemented in Java, it does not require the user entity to use a specific language or platform. Section 4 covers the GLA in more detail.

The brokering system is responsible for managing all the resources in the data and simulation cloud. Specifically, it stores descriptions and references for all the resources in the system and matches requests with corresponding services, based on textual descriptions. Once the brokering system establishes links between two GLAs, the GLAs communicate directly without going again through the broker. In Figure 1, the broker has established one link between GLAs 1 and 2 and another link between GLAs 1 and 4. The brokering system also notifies the users and updates the list of services when new services arrive or existing services become unavailable

Logically, the brokering system consists of the broker, the matching and ranking system, and the keyword and unit databases. For better accuracy and efficiency, there are two levels of matching in the ABELS system. The broker stores the services and performs the first-level matching according to high-level categories or groups such as "medical simulation" and "weather simulation". The matching and ranking system performs the second-level matching and ranks all the matching services according to their descriptions. For efficiency and possible user interaction, the matching and ranking system is local to each GLA. (As discussed in Section 4, the matching and ranking system is actually implemented as part of the GLA, even though it logically belongs to the brokering system.) The keyword and unit databases provide users with keywords and units to accurately describe services and requests. The unit database also contains conversion factors between related units. The broker is implemented using Sun Microsystems' Jini technology. For more information on the brokering system, see Kumar et al. (2002).

4 THE GENERIC LOCAL AGENT

4.1 Overview

The ABELS *generic local agent* (GLA) is a user entity's portal into the ABELS cloud. Every user entity, whether a consumer or producer, connects to and interacts with other entities in the cloud through its generic local agent. Several user entities, administered by the same person or organization, may use the same GLA to connect to the ABELS cloud.

A GLA that acts as a portal to a producer user entity is referred to as a *producer GLA*. Conversely, the GLA for a consumer entity is a *consumer GLA*. Every GLA has the potential to be a producer, a consumer, or both, and its designation may change over its lifetime, based on the entities that use it to connect to the ABELS cloud.

The main functions that the GLA performs on behalf of its user entities are the following:

- The GLA maintains service registrations in the cloud on behalf of producer user entities.
- It performs service lookups with the broker on behalf of consumer entities.
- It refines service matches from the broker on behalf of consumer entities.
- It initiates remote service requests on behalf of consumer entities.
- It relays incoming service requests to producer entities, and returns the response to the requestor.

In order to perform these functions, the GLA must interact with the other entities in the ABELS cloud, namely, its own user entities, the brokering system, and other generic local agents. Figure 1 illustrates these interactions. The GLA uses a TCP/IP socket to communicate with its user entity and RMI to communicate with other GLAs and the brokering system.

As illustrated in Figure 2, the functionality of the GLA is divided into four modules, a detailed description of which forms the bulk of the remaining sections of this paper. These modules are briefly described below.

The *service module* stores information on services provided by producer entities using this GLA. It is responsible for all interactions between the GLA and its producer entities.

The *query module* stores information on queries being made by consumer entities using this GLA. It is responsible for all interactions between the GLA and its consumer entities.

The *matching and ranking module* is responsible for refining the high-level matching between producers and consumers that is performed by the broker, to better suit the needs of the consumer GLA. It is used by the query module of a consumer GLA.

The *communication module* is responsible for all communication between the GLA and the brokering system. Its services are used by the other three GLA modules.

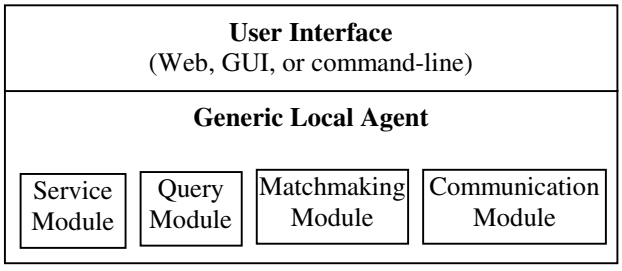

Figure 2: Conceptual View of the GLA and Its User Interface

A final component that is closely associated with the GLA, but is not a module of the GLA, is the user interface. While the GLA interacts directly with its producer or consumer user entities, it interacts with a human user or administrator through a user interface. As described in Section 4.2, the user interface assists the human user in setting up and editing descriptions of services or queries, that are then stored and used by the GLA.

Figure 2 illustrates the conceptual view of the GLA, its modules, and its user interface. The implementation of the user interface is in reality decoupled from the generic local agent, as shown in Figure 3. This is to provide more flexibility to the user in terms of which kind of user interface to use. It also reflects the fact that after the one-time description of services and/or queries is complete, the GLA will often be used without human interaction.

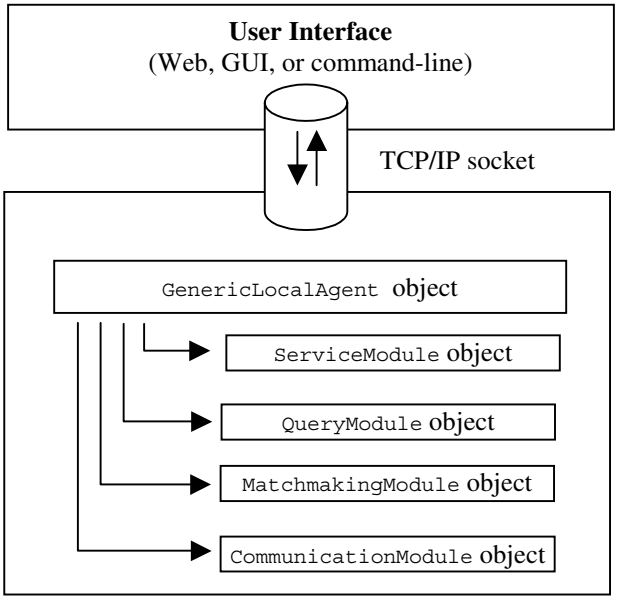

Figure 3: Implementation Overview of the GLA and Its User Interface

4.2 Defining Services and Queries

One of the primary goals of the ABELS system is the ability to exchange data between different entities with little or no changes in the implementation of the user's simulation. With a variety of simulations running on a variety of platforms, socket communication was chosen to be the most common platform- and programming-independent way of passing data between entities. However, in order to achieve true communication (i.e., a seamless integration of data between heterogeneous simulations), a standardized way of describing a simulation through its inputs and outputs was developed. Since the producer and consumer GLAs will exchange data through sockets in a byte stream, the simulation inputs and outputs will be defined in a one-dimensional format. To assist the user, the GLA provides interfaces specifically for the purpose of formatting services and queries. We have developed a graphical web-based interface using Java Server Pages, and we are currently developing a Java graphical user interface based on Swing. A command-line interface will be developed in the future.

The ABELS user interface attempts to provide a very adaptable way of describing simulations regardless of their platform, programming language, and content. By viewing simulations as black boxes and only focusing on defining their inputs and outputs, the interface can abstract away the details of the simulation implementation.

Inputs and outputs are further generalized in terms of a sequence of variables having a specific data type, definition, and possibly additional qualifiers (e.g., units). To reduce the amount of repetition that may occur when defining identical variables in an input or output, the user begins by defining components (i.e., definitions, subsets, ranges,

units) and then later assembles them into variables. Figure 4 gives some examples of components.

Definitions:	Family name, Age, First name, GPA, Pets
Units:	Years, Meters
Subsets:	{Dog, Cat}, {Man, Woman}
Data types:	String, Float, Integer, Double

Figure 4: Example Components

Once all of the components are defined, the interface will guide the user through the process of assembling the input and output sequences for the simulation, one variable or pattern at a time. Table 1 demonstrates a theoretical one-dimensional input sequence.

Table 1: Input Sequence with Variables Defined through Components

Data Type	Definition	Qualifier	Type
String	Family name		Variable
Float	GPA		Variable
String	First name		
Float	GPA		Pattern
Integer	Age	Unit: Years	
String	First name		
Float	GPA		Pattern
Integer	Age	Unit: Years	
String	Pets	Subset: Dog, Cat	Variable
String	Pets	Subset: Dog, Cat	Variable

The user assembles the input sequence by defining variables and using patterns. Patterns are repeating sequences of variables that contain exactly the same components. Therefore, the user can define patterns (if any) and then use them in the same way as components (e.g., as a qualifier to a variable). Table 2 demonstrates how the user would define the example above.

Table 2: Example Input Sequence Defined

Input Location	Input Type	Input Data Type	Input Description	Input Qualifier	Input Repeat
1 – 1	Variable	String	Username		1
2 – 2	Variable	Float	GPA		1
3 – 8	Pattern			Person	2
9 – 10	Variable	String	Pets	Dog, Cat	2

Once all inputs and outputs to a simulation are defined, the graphical user interface presents a build page where the user can verify previously-defined service information and select the input/output combinations that will be offered by the service. A service can support several functions (i.e., a specific input/output combination located on a specific port). For example, a simulation can have a function that expects an input of two integers and outputs one integer as the sum. The same simulation can have another function on a separate port that receives the same two input integers but outputs one float as the average.

The user interface will prompt the user to further describe the simulation by selecting keywords and ABELS groups to join. All producer simulations are required to join at least one ABELS group. As discussed in Section 4.5, a consumer begins the search for producer entities by performing a first-level lookup to locate entities belonging to specified groups. This grouping helps reduce the search time for consumers.

Once simulations have been retrieved through the first-level lookup, keywords can be used to broaden a consumer's search through these simulations. For example, a consumer searching for a service about cats can further broaden the search by adding keywords such as "feline" and "kitten". This will not only retrieve all producer services with the word "cat" in the definition but also retrieve those potentially useful services with the words "kitten" or "feline" in the definition. In the same way, producers can also add keywords to the definition to make the service more likely to be found by consumers.

Both consumers and producers will use the interface to connect or disconnect to the ABELS cloud, describe their simulations, and/or search for other simulations. Once a user has used the interface to describe the simulation, this information will be passed to the GLA for use in registration and stored locally for future updates.

The graphical user interface (GUI) provides the following advantages to the ABELS system:

- User-friendliness: The GUI provides clear instructions for the user to register services and requests. It allows the user to navigate backward and forward between each registration step to view or alter previously entered information. It also has describes data in terms of reusable components.
- Robustness: The GUI provides a way of error checking data before it gets sent to the ABELS cloud. In this way, user inputs can be validated throughout the registration process and errors that may occur due to inconsistent simulation descriptions can be caught ahead of time.
- Scalability: The GUI communicates with the GLA through sockets and therefore can be launched from the same computer as the GLA or from a completely separate computer. This allows multiple users to access the GLA from different machines. Furthermore, the web-based interface allows users to access the GLA from anywhere in the world through a basic web browser.
- Security: The GUI will have an initial login page to validate users. Since potentially sensitive data may be exchanged, this provides an extra layer of security to the ABELS system. The interface se-

curity is only one of many security mechanisms that will be used in ABELS.

4.3 The GLA Service and Query Modules

After the interface gathers the service information from the user, the service module (for producers) or query module (for consumers) is responsible for saving and managing this information. They both act on behalf of their user during the data exchange process and are primarily involved in the following areas:

- Local storage
- Request/Response
- Statistical data.

As described in Section 4.2, both consumers and producers define their simulations through a common user interface. Once a user finishes defining a simulation, the information is saved in a Java object and, in the case of a producer, passed to the communication module for registration with the broker. This registration object contains the following information:

Main Information
- Simulation name
- Simulation description
- Simulation location (IP address)
- Simulation functions (input/output combinations)
- Simulation keywords
- ABELS groups joined or requested by the simulation

Helper Information
- Component information (i.e., definitions, ranges, subsets, units, patterns).

This registration object fully describes the simulation and is the common link between a consumer's request for data and a producer's advertisement of its data. For a producer of data, it is a description of the service to be provided to the entities in the ABELS cloud (i.e., given a certain input, what the service will produce as an output). For a consumer, it is the description of the simulation data that is desired from the cloud (i.e., the input that can be provided and the output that is desired). On registration, a producer's service module and consumer's query module are responsible for storing this Java object locally. For future updates to a previously-defined simulation, the service module or query module retrieves the Java object so that the user can edit the information through the interface.

The data exchange begins on registration by placing all user simulation descriptions (consumer and producer registration objects) into specific groups. When a consumer makes a first-level lookup request (as discussed in Section 4.5), the broker returns all producer simulations that are part of the same group. The query module receives the first-level lookup results and then passes them to the matching module for the second-level matching and ranking. The second-level matching and ranking, as described in Section 4.4, is a detailed comparison between the given service and the query, resulting in a ranking between 0 and 1.0 that represents the degree of compatibility between the service and the query. As a result, the query module stores a sorted list of the ranked matching services with each query, and will use the highest ranked service for data exchange.

A human user, via a user-interface, can also override the second-level matching and ranking by designating a particular matching service as "preferred" or "unsuitable". These designations are used to further classify services and give the user more control over the service(s) used by his simulation. Therefore, the query module will maintain for each query three lists of services: "preferred", "regular" and "unsuitable". Resolving a query then follows a path of selecting the highest-ranked service in the "preferred" list, then if necessary (i.e., there are no services in "preferred" or it cannot communicate with them), choosing one from the "regular" list. A service will never be selected from the "unsuitable" list.

Once the query module has selected a service, it uses the service proxy sent with the service description to contact the simulation and begin transferring data. It takes the input data from the local simulation, performs any conversion needed (dictated by the matching and ranking module described in Section 4.4), and then sends it through the consumer GLA to the receiving remote simulation. The query module waits for the eventual output response from that remote simulation and again handles the conversions before channeling it back to the consumer simulation. By handling each request in a separate thread of execution, the query module can handle multiple requests from its consumer entities simultaneously.

When a request is made by a consumer, it is the producer service module that handles the execution of the simulation. It retrieves the input data from the requesting remote consumer simulation, channels it to the appropriate local producer simulation (using the IP address and port contained in the request), and waits for output. Finally, when the output is eventually returned, the service module sends it back to the requesting remote consumer simulation. Figure 5 displays a high-level view of the data exchange process.

When simulations locate each other through the cloud and begin exchanging data, the ABELS system will also maintain a variety of statistical data. The producer service module and consumer query module will be responsible for generating and storing this statistical data since all requests and responses are handle by these two modules. The following statistics will be collected:

Service module
- Average wait time in the queue

- Average execution time of each simulation
- Total number of users accessing each simulation

Query Module

- Average wait time per request per service
- Ranks of all services.

The data collected can then be used to improve the response time in future applications where the emphasis is placed on execution speed (e.g., real- time applications).

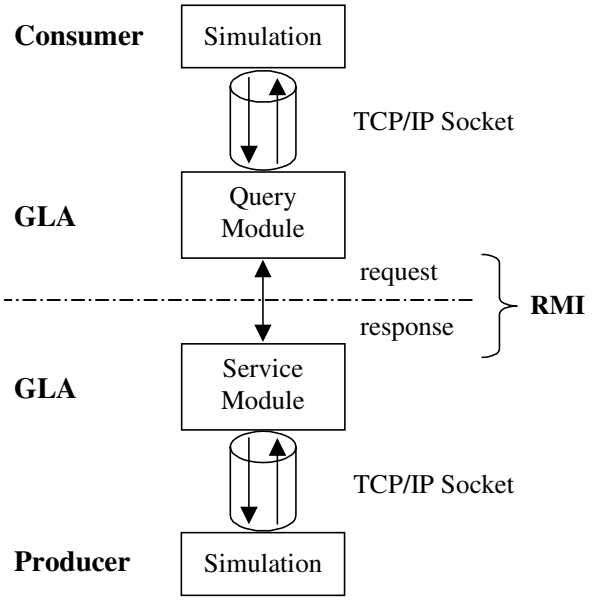

Figure 5 : Data Exchange between Generic Local Agents

4.4 The GLA Matching and Ranking Module

At the heart of the ABELS system is a sophisticated matching and ranking system that facilitates the difficult task of finding services which might be of interest to a particular consumer. In the first-level look up, explained in Section 3, the service descriptions for all services in the current group are retrieved from the broker. After this process, each description is sent to the matching module, where it is compared with the query specification and receives a rank from 0 to 1.0. This rank will give some indication of how consistent a particular service is with the given query.

The architecture of the matching module, as shown in Figure 6, is a multi-component system that communicates with the broker and GLA. Although logically the work of the matching module falls under the scope of the brokering system, it is implemented as part of the GLA to avoid bottlenecks. The matching module contacts the *unit database*, a centralized repository for information on possible units (e.g., miles, degrees Celsius, etc.) and conversions between exchangeable units. The database is stored centrally because applicable conversions can be added at any time by other entities. During the matching process, the matching module finds any unit information available though RMI calls to the database, and checks for the convertibility of query units to service units.

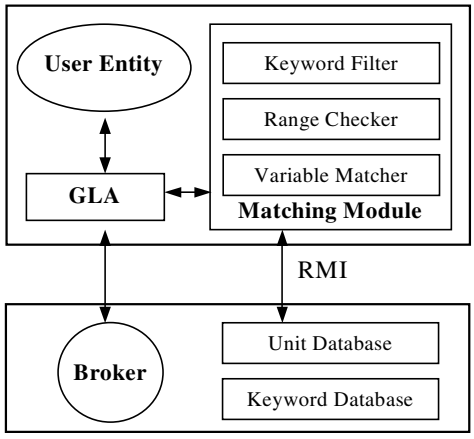

Figure 6: Matching Module Components and Interactions with Other Modules

The *keyword database*, although not specifically used by the matching module, is important for ranking. It allows users to select possible synonyms for their keywords during the initial description of their service or their construction of a query. This will assure that all possibly relevant services are found by the ranker. Only synonyms that the user selects will be added to a list of keywords, to ensure that superficially similar words that are not relevant to the particular description are not included. For example, we would not want a search that included the keywords "Puget Sound" to match with a service containing the keywords "interference through communication systems", despite the fact that both descriptions contain a synonym of the word "noise".

The *keyword filter*, a component of the matching module, will look at the keywords and descriptions specified in either a query or a service description and filter out words (e.g., prepositions) that will be of little use in finding matches. A static list of words deemed useless is stored locally for comparison. Unless they have been specifically designated to be kept, these words are removed from the descriptions for matching. After this filtering, descriptions are compared for relevancy.

The *range checker* will check possible values that input/output parameters of a desired service can take, as specified in either ranges or subsets. These are compared with those of the query to determine whether there is any possible overlap. This process also takes into account possible unit conversions. In the case of no overlap, a service's rank is decreased.

A third component of the matching module is the *variable matcher,* which will compare two sets of variables, one specified in the query and another present in the service. The variable matcher will evaluate all the possible

variable mappings and determine the optimal mapping of variable descriptions from the query to the service. This ensures that regardless of the order in which a query specifies its variables, the best match will be found. In its most basic form, this process is a perfect matching on a complete bipartite graph, and the variable matcher utilizes the Kuhn-Munkres algorithm (Bondy and Murty 1976).

Once the rank of a service is computed, the module returns the rank, any unit conversions, and the variable mapping. The rank will be used in the selection of a service, and the additional information is cached to avoid further time-consuming RMI calls to the broker when a service is chosen.

4.5 The GLA Communication Module

The communication module of the GLA encapsulates all communication between the GLA and the broker. It exposes the functionality of the brokering system to the rest of the GLA. It also hides the implementation of the broker communication technology from the other GLA modules, allowing the communication technology or the implementation of the brokering system to undergo changes in the future without repercussions on the entire GLA.

The main functions that the communication module exposes to the rest of the GLA are the following:

- Joining and exiting the ABELS cloud
- Querying the broker for the list of groups currently supported by the ABELS cloud
- Submitting a request to the broker for a new group to be supported by the ABELS cloud
- In a producer GLA, performing service registration with the broker and maintaining leases on all service registrations
- In a consumer GLA, performing the *first-level lookup* with the broker and maintaining leases on all first-level lookup subscriptions
- In a consumer GLA, relaying first-level lookup notifications from the broker to the query module.

Like the ABELS brokering system, the GLA communication module is implemented using Sun's Jini technology. Jini's purpose is to federate groups of devices and software components in a dynamic distributed system. The *lookup service* forms the heart of the Jini architecture, and it is the central repository of objects that can be searched based on what interfaces they support or other criteria. The prototype distributed brokering system used in the current implementation of ABELS consists of several Jini lookup services, each running on a separate machine and supporting specific groups of services.

The communication module of the GLA registers each service supported by a producer GLA with one or more of the Jini lookup services in the brokering system, based on the group(s) that the service wishes to join. The registration information stored with the broker includes a description of the service, formatted as described in Section 4.2, as well as a service proxy object that will be used by clients to communicate back to the producer GLA in order to execute the service. In the current implementation, this service proxy is a remote reference to the service module, implemented using Java remote method invocation. After registering each service, the communication module receives and persists the service IDs assigned by the lookup service to each service, and maintains leases on these service registrations for as long as the GLA is part of the ABELS cloud. When the GLA exits the ABELS cloud, all the services are de-registered with the broker. When the GLA later re-enters the ABELS cloud, the services are re-registered with the same service IDs that the communication module saved to persistent storage. This behavior conforms to the Jini *join protocol*, and makes it possible to uniquely identify a service across several registrations and de-registrations in its lifetime.

On the consumer's side, the communication module performs a *first-level lookup* with the broker to find all services registered with the cloud in a specific group or set of groups associated with a particular query. The service registration information and the service proxy of each service returned from the broker are passed to the query module, which is responsible for managing the consumer entity's queries. When performing the first-level lookup for a query, the communication module also subscribes to notifications from the broker for changes in service registrations in the groups of interest. The notifications are received, for example, when a new service in the groups of interest enters the cloud, or when a previously discovered service exits the cloud. The communication module conveys these notifications to the query module as needed. Some performance optimizations implemented by the communication module include consolidating notifications representing the same event but sent from different lookup services, and caching first-level lookup results for one query to be used for a different query interested in the same groups. The communication module maintains leases on the first-level lookup notification subscription for as long as the GLA is part of the ABELS cloud.

5 CONCLUSIONS AND FUTURE WORK

This paper presented the redesign of the generic local agent, which serves as the portal to the ABELS system. It also demonstrated how the system adapts to the needs of the users.

For future work, there are many improvements and enhancements that can be made. First of all, we will continue to develop and improve the user interfaces. In particular, we will extend the functionality of the user interfaces to expose more administration capabilities to the human user. For example, we will allow the user to de-

velop lists of preferred services and unsuitable services, to be used by ABELS in dynamically switching between matching services. We will continue to improve both the first- and second-level matchings. The system will be expanded to allow not only one-to-one matchings but also many-to-one matchings of services to queries so that the results from several services can be combined to generate the desired response. We are developing a test deployment scenario to experiment with the matching and ranking of actual services to study how well the system performs. Finally, there are several security features that must be added. In addition to the basic user validation mechanism described in this paper, we will add encryption capabilities and authentication mechanisms to restrict access to sensitive resources.

ACKNOWLEDGMENTS

This work is supported by National Science Foundation KDI Grant 9873138 and U.S. Army Corps of Engineers contract DACA42-01-P-0288. We would like to thank Min (Cecilia) Zhang for her work on the graphical user interface.

REFERENCES

Bondy, J. A., and U. S. R. Murty. 1976. *Graph Theory with Applications.* London, England : MacMillan.

Christensen, E., F. Curbera, G. Meredith, and S. Weerawarana (editors). 2001. Web services description language (WSDL) 1.1. W3C Note 15 March 2001. Available online via <http://www.w3.org/TR/wsdl> [accessed May 06, 2002]

Curbera F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. 2002. Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing, March/April 2002 86-93.

Dahmann, J. 1998. Standards for simulation: as simple as possible but not simpler: the high level architecture for simulation. *Simulation* 71(6) 378-387.

Gudgin, M., M. Hadley, J.-J. Moreau, and H. F. Nielson (editors). 2001a. SOAP Version 1.2 Part 1: Messaging Framework. W3C Working Draft 17 December 2001. Available online via <http://www.w3.org/TR/2001/WD-soap12-part1-20011217/> [accessed May 06, 2002]

Gudgin, M., M. Hadley, J.-J. Moreau, and H. F. Nielson (editors). 2001b. SOAP Version 1.2 Part 2: Adjuncts. W3C Working Draft 17 December 2001. Available online via http://www.w3.org/TR/2001/WD -soap12-part2-20011217/ [accessed May 06, 2002]

Knutson, J., and H. Kreger. 2002. Web Services for J2EE, Version 1.0. Public Draft v0.3. IBM. Available online via <http://www-3.ibm.com/software/solutions/webservices/pdf/websvcs-0_3-pd.pdf> [accessed May 09, 2002]

Kumar, A., L. F. Wilson, T. B. Stephens, and J. Sucharitaves. 2002. The ABELS brokering system. In *Proceedings of the 35th Annual Simulation Symposium,* 63–71. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

McKee, B., D. Ehnebuske and D. Rogers (editors). 2001. UDDI Version 2.0 API Specification. UDDI Open Draft Specification 8 June 2001. Available online via <http://www.uddi.org/pubs/ProgrammersAPI-V2.00-Open-20010608.pdf> [accessed May 06, 2002]

Mitra, N. (editor). 2001. SOAP Version 1.2 Part 0: Primer. W3C Working Draft 17 December 2001. Available online via <http://www.w3.org/TR/2001/WD-soap12-part0-20011217/> [accessed May 06, 2002].

Sucharitaves, J., L. F. Wilson, and A. Kumar. 2002. The generic local agent: gateway to the ABELS system. In *Proceedings of the High Performance Computing Symposium – HPC 2002*, ed. A. Tentner, 147–154. San Diego, California: The Society for Modeling and Simulation International.

Wilson, L. F., D. J. Burroughs, A. Kumar, and J. Sucharitaves. 2001. A framework for linking distributed simulations using software agents. *Proceedings of the IEEE*, 89 (2): 186–200.

BIOGRAPHIES

G. AYORKOR MILLS-TETTEY is a master's student at Dartmouth's Thayer School of Engineering. She received her AB degree from Dartmouth College in 2001.

GREG JOHNSTON is a research associate at Dartmouth's Thayer School of Engineering. He received his AB degree from Dartmouth College in 1999.

LINDA F. WILSON is Associate Professor of Engineering at Dartmouth's Thayer School of Engineering. She received her MSE and PhD degrees from The University of Texas at Austin in 1991 and 1994, respectively.

JOSEPH M. KIMPEL is a senior engineering major at Dartmouth College.

BIN XIE is a doctoral student at Dartmouth's Thayer School of Engineering.