MATCHMAKING IN THE ABELS SYSTEM FOR LINKING DISTRIBUTED SIMULATIONS

Joshua O. Peteet, John P. Murphy, and Linda F. Wilson
Thayer School of Engineering
Dartmouth College
Hanover, NH 03755-8000 USA
Email: Firstname.Lastname@dartmouth.edu

KEYWORDS

Distributed simulation, brokering systems, matchmaking, dynamic information exchange, software agents, simulation tools.

ABSTRACT

Large-scale simulations often need dynamic access to heterogeneous data resources such as sensors, databases, or other simulations. The Agent-Based Environment for Linking Simulations (ABELS) is designed to facilitate the dynamic formation of a "cloud" of independent simulations and other data resources for the exchange of information. Participants in the data and simulation cloud join and exit the cloud as needed and have no prior knowledge of the other cloud participants. The formation of the cloud is achieved using a distributed brokering system that matches data consumers in the cloud with appropriate data producers, based on registration information submitted by the various participants in the cloud. This paper describes in detail the process used to match and rank prospective data producers for a given data consumer.

1. INTRODUCTION

Suppose you have a large simulation or similar application that needs access to data from multiple networked resources. One approach is to hardwire those connections so that your application knows exactly where to find the data, what formats are used, etc. One downside of this, of course, is that you must change your code if any of the resources change in location or format. Another approach is to write your application to a particular standard (e.g., HLA, Dahmann et al. 1998), so that you can communicate with other resources that conform to the standard. However, you may use only those resources that conform to the standard, and in many cases, you must still know information about the resources that are used by your application. Yet another approach is provided by the web services architecture, which uses the SOAP, WSDL, and UDDI protocols to provide interoperability between independent services (Curbera et al. 2002). The web services architecture, however, requires that you know in advance which specific resources will be used.

Networked resources come and go, so problems can occur if the resources you have specified become unavailable when you are running your simulation. It would be helpful if your simulation could use the best resources available and determine what those resources are at runtime. Furthermore, it would be even better if you could take advantage of all resources available, rather than only those written for a particular standard.

The Agent-Based Environment for Linking Simulations (ABELS) system is designed to facilitate the rapid formation of a distributed "cloud" of autonomous data resources (Mills-Tettey et al. 2002; Mills-Tettey and Wilson 2003a; Mills-Tettey and Wilson 2003b; Murphy et al. 2003; Wilson et al. 2001; Wilson et al. 2002; Wilson et al. 2003). Individual cloud participants join and exit the data and simulation cloud as needed and have no prior knowledge of the other cloud participants. A distributed brokering system is used to match data producers to consumers and initiate communication between cloud participants, but it does not control the independently-designed participants in any way. Each participant in an ABELS cloud is responsible for determining what resources it makes available to the cloud and for describing those resources accurately.

Cloud participants may be data producers, data consumers, or both. Each data producer is said to provide a service, while a data consumer is said to make requests or queries for information. A service definition includes the name, location, and description of the service along with detailed information about the functions it provides. A query is defined as the ideal function desired by the consumer. Note that a particular query may match functions from multiple producers, so the ABELS system must perform a detailed matching and ranking of the candidate services and functions.

This runtime matching of data producers and consumers is a key feature of ABELS. While a simulation or other application 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 participants (e.g., language, units, file formats, etc.). This feature allows the transparent replacement of one service provider with another that provides similar

Proceedings 15th European Simulation Symposium
Alexander Verbraeck, Vlatka Hlupic (Eds.)
(c) SCS European Council / SCS Europe BVBA, 2003
ISBN 3-936150-28-1 (book) / 3-936150-29-X (CD)

functionality, without having the services be written to conform to a particular standard.

This paper describes in detail the matching and ranking system used to match consumers with producers. Section 2 provides an overview of the ABELS system and mentions related work. Section 3 discusses the goals of the matching and ranking system, and Section 4 describes the process of defining services and queries. Section 5 discusses the ranking process by which the individual service functions are evaluated in terms of fitness to a given query. Section 6 examines the query resolution process, while Section 7 presents conclusions and areas of future work.

2. BACKGROUND

2.1. The ABELS System

The Agent-Based Environment for Linking Simulations (ABELS) is a software framework that allows independent simulations and other data resources to exchange information with no prior knowledge of each other. An ABELS cloud is a federation of communicating participants, where each participant produces and/or consumes some type of data. As shown in Figure 1, the ABELS system architecture consists of three basic types of components: user entities, generic local agents (GLAs), and a distributed brokering system. An optional user interface is provided to permit human interaction with the system via the GLAs.

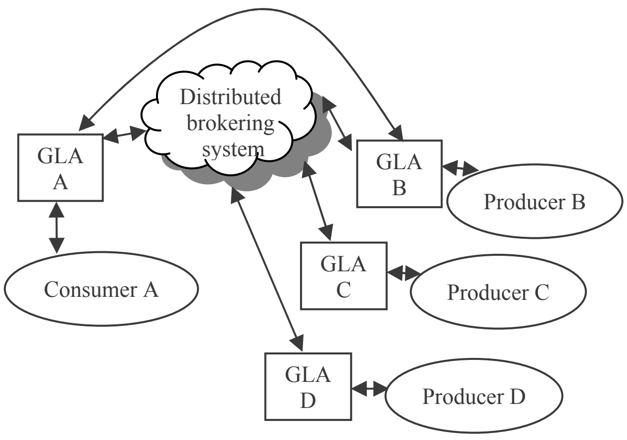

Figure 1: Basic Framework Connecting Elements in the Cloud

A user entity is any producer or consumer of data, and simulations often serve as both producers and consumers. A producer is considered to be a service that provides one or more service functions. A consumer makes requests or queries for the information it desires from the cloud. Figure 1 shows four user entities: Consumer A, Producer B, Producer C, and Producer D.

The ABELS system is designed for loosely-coupled interactions between cloud participants. That is, user entities are independent and are not written to a particular standard, and consumers are not statically linked to particular producers. Furthermore, there are no tight interdependencies between cloud participants; ABELS is not appropriate for participants that are tightly-coupled to one another.

Each user entity communicates with the cloud via its general local agent (GLA). The user entity uses its GLA to join or exit the cloud, register its services and service functions, and make queries for data. A data producer is said to interface to the cloud via a producer GLA, while a data consumer communicates via a consumer GLA. Although a particular GLA may serve both producers and consumers for a single organization, it is still useful to discuss the GLA by separating its capabilities into producer actions and consumer actions. For example, a consumer GLA is responsible for handling any data format, unit, and file conversions that are necessary between the consumer and the producer that is serving it. A producer GLA is responsible for passing input data to a desired service, executing the desired service function, and returning the output data to the corresponding consumer GLA. In Figure 1, GLA A is a consumer GLA, while GLAs B, C, and D are producer GLAs. The GLAs are implemented in Java.

The distributed brokering system is responsible for managing all of the cloud participants and matching consumers with suitable data producers. Once the brokering system establishes links between two GLAs, the GLAs communicate directly without going again through the broker. In Figure 1, there is a link between GLA A and GLA B, indicating that a match has been made between a query of Consumer A and a service function of Producer B. Logically, the brokering system consists of the broker, the matching and ranking system, and the keyword and conversion databases. The broker is implemented using Java and Sun Microsystems' Jini technology (Kumaran 2002), while the other components are implemented using Java. (Additional information on Jini can be found at http://www.sun.com/jini).

The broker manages all of the resources in the cloud. It uses a system of leases to determine which participants are in the cloud and detect when someone has left the cloud unexpectedly. It also stores descriptions and remote references or proxies for all of the resources in the system. When a producer GLA registers a service with the brokering system, the GLA sends a proxy object that is used later by a corresponding consumer GLA to communicate with the producer GLA via Java Remote Method Invocation (RMI). When new services arrive or existing services become unavailable, the broker also notifies potential consumer GLAs of these changes, so that the best service function available can be used to resolve a consumer's query.

For better accuracy and efficiency, there are two levels of matching in the ABELS system. The broker stores the service information and performs the first-level matching according to high-level categories or groups such as "medical simulations" and "ocean simulations". That is, the broker determines which services belong to the groups of interest. As indicated in Figure 1, there may be multiple producers eligible for matching with the query of a consumer. The matching and ranking system, which is the focus of this paper, performs the second-level matching by comparing the query with all of the service functions belonging to the services returned by the broker's first-level matching. Note that each service may have multiple functions, only some of which may be relevant to the particular query. Thus, the matching and ranking system examines the query and all of the corresponding service functions in detail, and ranks the matching service functions according to their descriptions, data types, keywords, and measurement units. For efficiency and potential user interaction, the matching and ranking system is local to each GLA and is actually implemented as part of the GLA. The matching and ranking process will be described thoroughly in the remainder of this paper. Additional information on the ABELS framework can be found at http:// thayer.dartmouth.edu/~abels.

The keyword and conversion databases provide users with keywords, units, and file types to accurately describe services and queries. The conversion database also contains conversions between related units or file types; thus, a consumer can receive data in meters even if its matching service function produces data in feet.

There may be multiple clouds running at a particular point in time, and each cloud has an administrator that sets the policies of the cloud. For example, one cloud may be set up for general use by anyone at a particular location, such as a college or university. Another cloud could be set up for use by researchers in a particular field, such as cancer research or oceanographic modeling. Although it is possible for consumers and producers to belong to multiple clouds, queries and service functions do not cross cloud boundaries. That is, a query defined in a particular cloud will match only those service functions belonging to the same cloud.

2.2. Related Work

The ABELS system is not the only one with the goal of enabling the interoperability and reuse of simulations. Other systems include the High Level Architecture (HLA) (Dahmann et al. 1998), the web services framework (Curbera et al. 2002), the First All Modes All Sizes (FAMAS) project (Boer et al. 2002; http://www.famas.tudelft.nl), and the Extensible Modeling and Simulation Framework (XMSF) (Brutzman et al. 2002).

In HLA, participants called federates participate in a federation of interacting simulations. Participants must be written to meet the federation object model (FOM) standard, and all interactions between federates occur through the runtime infrastructure (RTI), which acts as a distributed operating system for the federation. HLA is designed for tightly-coupled interactions, while ABELS is designed for loosely-coupled interactions.

The web services framework (Curbera et al. 2002) consists of a collection of protocols and standards for communication (SOAP), service description (WSDL), and service discovery (UDDI). Although it was not designed specifically for simulation, the web services framework can be applied to simulation interoperability. However, it does not directly support the runtime brokering or matching of data consumers with producers, and this runtime matching ability is a key feature of the ABELS system.

The Extensible Modeling and Simulation Framework (XMSF) applies web-based technologies, such as XML-based languages and the web services framework, to create standards which allow modeling and simulation interoperability between distributed systems (Brutzman et al. 2002). Like the application of the web services framework to simulation interoperability discussed above, the XMSF does not support runtime matching and brokering of resources.

The FAMAS Simulation Backbone (Boer et al. 2002; http://www.famas.tudelft.nl), like ABELS, is designed to support interoperability between different and distributed simulation models without requiring adherence to HLA standards. However, the FAMAS approach requires a predetermined scenario script to control simulation runs. In contrast, ABELS is adaptable to changing simulation resources at runtime, allowing flexibility to changing circumstances.

3. GOALS OF MATCHING AND RANKING

The goal of matching and ranking is to find the best available service to resolve a given query at runtime, with no prior knowledge of the data format that a particular service uses, or even where the resource is located. Data producers and consumers have no information about each other in advance of the matching and ranking process, and need not conform to any particular standards. Because several very similar services may suit a given query, finding the best service is often a difficult task. To optimize their performance, the matching and ranking processes are implemented in the generic local agent, although both are logically part of the brokering system.

Each service is defined in terms of what functionality it offers in the form of one or more service functions. Each service function is defined as a sequence of input and output variables, with information on data types, units, ranges, and subsets. A query, seeking to find a single service function, also defines these parameters, and both query and service define a short description. These variables and descriptions are the sole basis for matching and ranking, and therefore consistency when registering services and queries is essential.

Two characteristics distinguish the ABELS matching and ranking system from similar approaches to linking simulations and other data services: its loosely-coupled nature and its capacity for runtime brokering. Unlike the High Level Architecture (HLA), ABELS is designed as a loosely-coupled system. A service is matched to a query solely on the basis of inputs, outputs, and user-defined descriptions, allowing ABELS to abstract both service and query from their implementation details. Unlike the web services framework, ABELS allows for runtime matching of services to queries, allowing the system to adapt to changes in the availability of networked resources.

4. DESCRIBING SERVICES AND QUERIES

The user entities in the ABELS cloud can produce data, consume data, or both produce and consume data. Data is produced through registered services, and requests for data are specified through queries. Each service and query is defined with information specifically provided for the matching and ranking process.

The first step in the matching and ranking process is the definition of the service in question. Generally this will be done only once for any given service, as the definition persists and can be automatically re-registered if the service goes offline and later comes back online again. If the service itself changes, so must its definition, though it need not be wholly rewritten.

A service consists of one or more related functions that are offered from the same computer. For example, a service may be defined to offer access to a weather database, and the individual functions would offer particular information from that database, such as the temperature or wind speed on a certain day at a certain location. Each function has its own data flow, taking in the input information needed to retrieve or calculate its output information. These service functions are matched against the individual queries.

The service represents everything the user entity offers to the cloud, and its description should include details that are common to all of its functions. This includes group membership, keyword information, and a text description. This text description contains information about the origin of the service, the computations performed, any relevant equipment (such as a sensor setup), and anything else needed to describe the service. This description should ideally be as detailed as possible to facilitate the most informed matches.

Each function in a service has its own name and description, and has a precise description of the input data the function takes, the output data it gives, and details of variables, the individual data items. These details include variable order and grouping, data range, and unit or file type. Figure 2, below, shows an example set of variables, split into input (top) and output (bottom) for a sample function. A variable could be a number, a date, a string of text, or even a file. Each variable has a name and a description, both for matching and for human readability.

An example might better illustrate the distinction between a service and its service functions. If we were to have a database of tide measurements in a local bay going back a century, there would be certain functions we could offer based on that database. One function might calculate the average high tide mark over a given span of time. Another might simply give a table of tide values over a given month. Another might give the lowest tide reading for an entire year. Each function shares the database but retrieves different information from it and performs different calculations.

In a similar manner, the query is written as though specifying an ideal function, from the name of the ideal service providing the function to the specifics of the data flow. The user might specify a preferred system or sensor setup, for example, and the ranges on its variables would be based on custom factors.

The process of defining a service is done through the user interface to the GLA. It is done on a partly *a la carte* style, where the component pieces of a service are defined first, and then the service functions are built from them.

Description	Repeat	Type	Subset	Range	Units
Year	1	int		[1907,2003]	year
Month	1	string	{"jan", …}		

Description	Repeat	Type	Subset	Range	Units
Num Rows	1	int		(0, inf)	unitless
Tide Mark	Num Rows	float		(0.2, 40.3)	inches

Figure 2: Sample List of Variables

The process starts with the name and definition of the service. Individual clouds may have guidelines for how these are written, such as to correspond to a certain schema, but ABELS itself requires only that they be plain ASCII text. The information here is common to all the functions defined as part of this service, such as location, database specifications, or contact person.

In the tidal measurements example, the service description needs all of the general information about the database, regardless of the individual functions it offers. That description might consist of the following text: "This service offers tide information for [the fictional] Wheelock Bay, from a database maintained at Dartmouth College using information collected by the National Weather Service from March 1907 to present. All measurements were taken at Hanover, New Hampshire."

The process continues by defining variable parts. Variable name/description pairs are written at this stage without associating them with an actual variable. There are three additional parts that can be defined here. The range is defined as a single numerical range in the style (min,max) or [min, max], allowing "inf" to indicate no defined upper bound beyond that of the type. The range is unitless as defined, and may be matched with different units. For example, the range "non-negative", defined as [0,inf), could be useful for many units. Defining ranges separately reduces repetition in creating the variables themselves.

A subset represents a finite list of allowable values that a variable may take. This will most often be found with string variables, such as {"north", "south", "east", "west"} or {"true", "false"}. Numbers are also allowed as subsets, such as the set of odd integers between 1 and 20. Numerical subsets, like ranges, are unitless.

The next step in our example, then, would be to define the variable parts to use. Variable names would include "month", "year", and "tide mark" with short text definitions such as "Tide height at the Hanover measuring station". Because the years are constrained by the measurement period, we could define a range "years period" as [1907,2003]; however, this would necessitate changing the service definition once 2004 data is available, so [1907,inf) is another possibility if the database interface can accept and properly deal with dates after the last entry. Similarly, we could define a "non-negative" [0,inf) range for use with the tide mark, or search the database to see what the actual global high and low values are to give a more exact output range. Using an actual range such as (0.2, 40.3) would be more likely to properly match query definitions, but would have to be updated if the database ever receives an entry higher or lower than that.

Month values could of course be integers, but if our particular database requires the three-letter month abbreviations, that can be accomplished using a subset "three-letter months" of the strings {"jan", "feb", "mar", …, "dec"}.

A unit or file type can be either user-defined or selected from a list of pre-defined units and file types. User-defined units are allowed, but may not allow conversion to other units or file types; a match in this case would have to be identical, which is generally acceptable for files. The pre-defined units and file types, however, are defined in the conversion database maintained on a cloud-wide basis by the broker, and have conversion routines defined to translate data from one unit to another (e.g., inches to meters) or one file type to another (e.g., MS Word to LaTeX). These conversion routines may considerably expand the pool of possible matches to a given query.

When these variable parts are defined, the user may build variables from them and define the input and output variable lists for each function. A variable must have a name, description, and type such as integer, floating point value, date, file, or text string. Units, ranges, and subsets may be associated with a variable here but are not required. Each variable also has a repeat value, which can be either a number or a reference to the value of a previously defined variable. The single variable, then, becomes a column of data of either fixed length or of a varying length to be specified at runtime.

The variable order matters; when defining a service, the input variables should be in the order that the service expects the data, and the output variables should be in the order in which the service returns the data. In defining its ideal service, a query would assume that the service takes the input data in exactly the order that the consumer entity gives its data, and the output in the order that the consumer expects its results.

Defining variables in this way is perfectly acceptable, but may become repetitious if similar variables are to be defined for multiple functions in the service. To avoid this repetition, the user may save individual variables or groups of variables as patterns. The first use of patterns is simple reuse: once a pattern is defined it may be used multiple times in multiple functions. The pattern name and a repeat value are all that is necessary to add a variable or group of variables to the list. Because patterns can contain multiple variables, a pattern can be used to quickly and easily define multi-column tables of data.

Returning to the example, we know that the variable representing a tidal reading will be used multiple times, so we define a pattern named "tidal reading". We use the name/definition pair "tide mark", declare it of type floating-point number, with the "non-negative" range, and we select the pre-defined unit "inches". We also give the pattern a repeat value of 1 because this is a single-variable pattern. Because year and month are also variables we will use repeatedly, we define appropriate patterns for them as well, "year" as an integer with range "years period" and pre-defined unit "years", and "month" as a text string with subset "three-letter months" and unit "months".

Defining these patterns will make it easier to define input and output lists. One service function offers the lowest tide reading for a year, so we define an input list with just the "year" pattern and an output list with just the "tidal reading" pattern, with one repetition of each. Another function offers a table of tidal values for a given month. The input list for that function would just be one instance of "year" and another of "month". For the output list, the database gives a column of readings preceded by the number of rows. To define this, we can go back and add a name/definition pair "number of rows"/"The number of rows in the following table", and add a variable to the output list with that name, of type integer, range "non-negative" (i.e., (0,inf)), and unitless. After that, we add the pattern "tidal reading" with a repeat value listing the variable we just defined. This is interpreted as repeating this pattern a number of times to be determined at runtime by the value of the variable "number of rows".

The next step in registering a service is to define each service function in terms of its description and its input and output lists, and save both lists to the service definition. The function descriptions should describe the calculations or data retrieval performed and more general information not included in the service description, such as average response time. The service keywords are then selected, and the service can then be registered in any available groups the user desires.

Concluding the example, we assemble the functions from the input and output lists we just defined, together with a short description for each function, such as "This function returns the lowest tide mark over the entire given year." Just like the variable parts and patterns, we can reuse the input and output lists. For instance, we defined an input containing the year and an output containing a single tide mark, which we could easily reuse for a function to offer the highest tide mark for the year. Once the functions are assembled, we select the keywords "oceanography", "New Hampshire", "tidal measurements", etc., and register the whole service in the "oceanography" and "Dartmouth College" groups.

The service definition will persist as long as the producer GLA is connected to the cloud. When a subsequent query is registered in any of this service's groups, the service description will be returned to the consumer GLA for matching and ranking. This group-based approach is the first-level lookup that is performed by the broker. Each service that is returned to the consumer GLA is examined and its functions ranked in order to determine which service functions to use for resolving queries.

5. THE RANKING PROCESS

The ranking process provides a basis for quantitative comparison among services by assigning a numerical rank to every service function that might satisfy a particular user-defined query. This numerical rank, a number between 0 (a non-match) and 1 (a perfect match), is the weighted average of several factors, each of which reflects some aspect of the fitness of a particular function for a particular query. Each individual factor has a value between 0 and 1.

The ranking process begins when a query is registered with the cloud. The broker performs a first-level lookup based on the groups of interest defined in the query specification, and sends service information to the consumer GLA for every service that has joined any of the groups of interest. This first-level lookup is the first of the two steps in the matching and ranking process.

This collection of services will vary greatly in terms of the functionality actually being offered. Some of the services may contain functions that match the query, but many will not. The goal of the second-level matching and ranking process is two-fold; it must determine which service functions are appropriate matches for the query, and it must determine which of the appropriate matches is the best match.

In determining the rank for a service function, the matchmaker in theory achieves both goals. The rank, a number between 0 and 1, indicates the relative fitness of the function in satisfying the query. In order to best satisfy its goals, the ranking process will be carefully tested and adjusted so that the ranks of inappropriate services are all clustered near 0, and the ranks of the appropriate services near 1, with very few services in between. In that way, the rank distribution will be considered a general indicator of whether a given service function is suitable or unsuitable.

In second-level matching and ranking, we first consider the keywords defined for both query and service. We compute the percentage of key words in the query description that are also contained in the function description. This percentage is one of the weighted factors in our comprehensive rank.

Second, we consider the groups defined for both query and service. For one of the weighted factors in our comprehensive rank, we compute the percentage of groups in the query description that are also contained in the function description, which is also one of the weighted factors in our comprehensive rank.

Finally, we assess the mapping success between function and query, that is, the degree to which the query specification is consistent with the function specification. In this stage of ranking, we consider the input and output variables for both service function and query. A function that is well-mapped to a particular query will contain the input and output variables specified in the query, measured in units compatible with the units specified in the query definition.

The mapping itself results from a variable-to-variable comparison where, in principle, each variable in the query is compared to each variable in the service in an effort to determine which service variable, if any, corresponds to it. On the left side of Figure 3, each directed edge represents such a comparison between variables for the query (Q) and service function (S). In practice, there is no reason to compare a number to a text string or to a file, or a file to a date, or any single variable to a table of variables. We can exploit this to reduce the number of comparisons we make by assigning to each variable a compound type, where all of the numbers (whether integer or floating point) are taken together, all the files, strings, dates, and arrays are each taken together, for a total of five compound types. Only variables of the same compound type are compared.

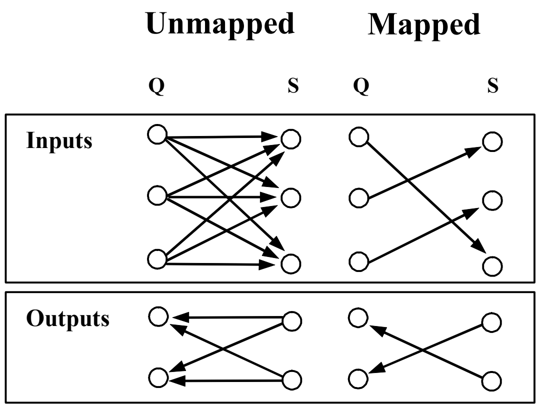

Figure 3: Unmapped and Mapped Variable Sets

The right side of Figure 3 represents the ideal outcome of the mapping process: Each variable in the query has been linked to a corresponding variable in the output. However, most of the functions being evaluated will not be perfect or even good matches; they are simply not matches suited to the query. These functions will often have different numbers of variables than the query, and so a one-to-one matching will not be possible or even desirable. Even if the query and service function have the same number of variables, if those variables are not describing the same data, they should be not be considered to correspond. It would be better to leave variables "orphaned" or unmapped than to map a query variable incorrectly as an indication of suitability.

When each variable pair is compared, we consider several factors. First, the name and description of the variables are compared for similarity. Then, the units of the two variables are compared. If they are the same, that is a fair indication of a match, but if they are different they may still match; for example, one may be in centimeters and the other in inches. The conversion database comes into play here, determining whether a conversion path exists between the two units, and if so, how long it is. Because the conversion database is linked internally according to scale, a long conversion path would indicate a likely mismatch, such as from micrometers to nautical miles, which are both measures of length but several orders of magnitude away from each other.

The data flow indicates whether a match exists for the range or subset. The chief criterion here is whether one entity may be producing data that is out of range for the other. Data flows from consumer input to service input, and from service output to consumer output. The ranges for the consumer inputs, then, should match or fit inside the ranges of the service inputs, and the consumer output subsets should be a subset of those of the service output. For example, the consumer could offer {"north", "northeast", "east", "southeast", etc.} but if the service accepts only {"north", "south", etc.} then the consumer could be providing input data that the producer does not recognize. A service that does not recognize data provided by the consumer would be penalized in the ranking process.

Any function that is not well mapped to a particular query faces a two-tiered penalty to its rank. Functions that do not produce output variables specified in the query are penalized heavily, as are functions that require input variables not specified in the query. Similarly, functions that produce output variables not specified in the query are penalized to a lesser degree, as are functions that do not require input variables specified in the query, and functions that contain units not convertible to units specified by the query. All such penalties are assessed on a perfect rank of 1.0, and the result is the final, and most important, factor in our weighted average.

By design, any function that is ranked has been returned by the first-level matching and therefore must define at least one group that is also defined in the query description. Accordingly, any ranked function must have a positive rank. We reserve the negative and zero ranks for ranking errors.

In addition to the automatic ranking, the user has an opportunity to examine the service functions and their assigned ranks. The first level of control that a user has is to designate certain services as more or less desirable than others. By labeling a service as *preferred*, *deprecated*, or just *unsuitable*, the user can determine the order in which the GLA will select services during query resolution. The second level gives the user control over the ranking weights themselves, such as the weight of the penalty given to missing variables or missing keywords.

6. THE QUERY RESOLUTION PROCESS

Once the query has been defined and registered, and the services returned from the first-level lookup have been ranked, the GLA is ready to resolve queries. The resolution process, which is shown in Figure 4, begins when the consumer entity connects to the GLA, provides the name of the query to be resolved, and sends the service input as a single data stream (Step 1).

The first task in the resolution process is to select the service to use. The GLA first looks at all of the services that the user has marked preferred, and selects from that list the service function with the highest rank. If that service is unavailable, it steps through the list of preferred services until it has exhausted them, and only then goes to the unmarked services, then to the list of deprecated services. If no services can be found in these three lists, the resolution fails rather than select an unsuitable service.

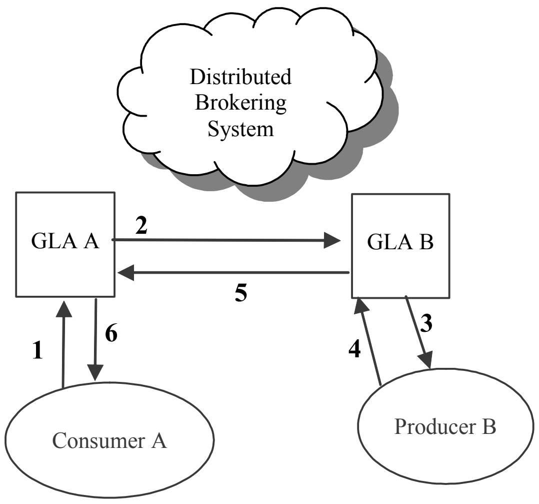

Figure 4: Data Flow Between Producer and Consumer During Query Resolution

When a service function is selected to satisfy the query resolution, the consumer GLA examines the mapping that was generated during the ranking process. It uses this information to determine how to rearrange variables and determine which variables to omit. The GLA extracts the data from the consumer's data stream, divides it into variables, rearranges as needed, and performs any necessary unit and file conversions. It formats this into a single data stream appropriate for the producer software, and sends it to the producer GLA (Step 2). The producer GLA contacts the producer entity (Step 3) with the data stream formed by the consumer GLA, and waits for the producer to return its output data stream (Step 4). This data is passed back to the consumer GLA (Step 5). Note that the work done by the producer GLA is kept to a minimum, to keep the burden on the producer side as small as possible.

The consumer GLA disassembles the output data stream according to the variable list in the service definition, and uses the mapping to reassemble the data for the consumer. It performs unit and file conversions, and reforms the data into a stream to return to the consumer entity, which may be waiting for the data (as a blocking query) or planning to contact the GLA at a later time to collect the data (as a non-blocking query).

7. CONCLUSIONS AND FUTURE WORK

The current implementation of the matching and ranking system is only partially complete; currently, it ranks on the basis of the service description, keywords, text descriptions, and number of variables. In the near future, the mapping process will be implemented, as will the conversion database. Further work will be required in several areas, such as finding more-sophisticated ways to match the text descriptions, adding user control, and determining the optimal weighting for different aspects of the ranking process.

Individual user groups may require a more customized system. Researchers whose simulations adhere to the SEDRIS format, for example, would require unit conversion to and from its Environmental Data Coding Specification (EDCS) standard for units. (Additional information on SEDRIS can be found at http://www.sedris.org.) Some unit schemes will be included in the standard ABELS conversion database, but new standards or ones that are not widely used may also be needed by certain users. To accomplish this, a flexible administrative interface for the conversion database will allow those groups managing an ABELS cloud to specify any number of units and file types and the conversion routines among them, without worrying about inconsistencies among service and query definitions.

ACKNOWLEDGMENTS

This work is supported by National Science Foundation KDI Grant 9873138 and U.S. Army Corps of Engineers contract DACA42-01-P-0288.

REFERENCES

Boer, C.; Y. Saanen; H. Veeke; A. Verbraeck. 2002. "Final Report, Project 0.2 - Technical Design, Simulation Backbone FAMAS.MV2." TRAIL Research School, Delft.

Brutzman, D.; M. Zyda; J. M. Pullen; K. Morse. 2002. "Extensible Modeling and Simulation Framework (XMSF), Challenges for Web-Based Modeling and Simulation", Findings and Recommendations Report: Technical Challenges Workshop, Strategic Opportunities Symposium (Fairfax, VA, 22 October, 2002), 1-52.

Curbera, F.; M. Duftler; R. Khalaf; W. Nagy; N. Mukhi; S. Weerawarana. 2002. "Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI." *IEEE Internet Computing* 6, No.2, (Mar./Apr.), 86-93.

Dahmann, J.; F. Kuhl; and R. Weatherly. 1998. "Standards for Simulation: As Simple as Possible but Not Simpler, the High Level Architecture for Simulation." *Simulation*, 71, No.6 (Dec.), 378-387.

Kumaran, S.I. 2002. *Jini Technology: An Overview*, Prentice-Hall, Upper Saddle River, N.J.

Mills-Tettey, G.A.; G. Johnston; L.F. Wilson; J.M. Kimpel; and B. Xie. 2002. "The ABELS system: designing an adaptable interface for linking simulations". In *Proceedings of the 2002 Winter Simulation Conference*, Volume 1 (San Diego, CA, December 8-11), 832-840.

Mills-Tettey G.A. and L.F. Wilson. 2003a. "Security issues in the ABELS system for linking distributed simulations". In *Proceedings of the 36th Annual Simulation Symposium*, (Orlando, FL, Mar. 30 – Apr. 2). IEEE, Picataway, N.J., 135-144.

Mills-Tettey, G.A. and L.F. Wilson. 2003b. "A Security Framework for the Agent-Based Environment for Linking Simulations (ABELS)". *Simulation*, to appear.

Murphy, J.P.; G.A. Mills-Tettey; L.F. Wilson; G. Johnston; and B. Xie. 2003. "Demonstrating the ABELS system using real-world scenarios". In *Proceedings of the 2003 SAINT Conference*, (Orlando, FL, Jan. 27-31). IEEE, Picataway, N.J., 74-83.

Wilson, L.F.; D.J. Burroughs; A. Kumar; and J. Sucharitaves. 2001. "A framework for linking distributed simulations using software agents". In *Proceedings of the IEEE* 89, no. 2, (Feb.), 186-200.

Wilson L.F.; B. Xie; J.M. Kimpel; G.A. Mills-Tettey; and G. Johnston. 2002. "The Design of the Distributed ABELS Brokering System". In *Proceedings of the Sixth IEEE International Workshop on Distributed Simulation and Real-Time Applications* (DS-RT) (Fort Worth, TX, Oct. 11-13). IEEE, Picataway, N.J., 151-158.

Wilson, L. F.; W. R. Lochridge; and G. A. Mills-Tettey. 2003. "The Secure ABELS Brokering System". In *Proceedings of the 15th European Simulation Symposium* (Delft, The Netherlands, Oct. 26 - 29), SCS, San Diego, CA, to appear.

AUTHOR BIOGRAPHIES

JOSHUA O. PETEET is a master's student at Dartmouth's Thayer School of Engineering. He received his AB degree in computer science from Bowdoin College in 2002. His email address is `Joshua.O.Peteet@dartmouth.edu`.

JOHN P. MURPHY is a PhD student at Dartmouth's Thayer School of Engineering. He received his BS degrees in computer engineering and electrical engineering from West Virginia University in 2001. His email address is `John.P.Murphy@dartmouth.edu`.

LINDA F. WILSON is an associate professor at Dartmouth's Thayer School of Engineering. She received her BS degree in mathematics from Duke University in 1988 and her MSE and PhD degrees in electrical and computer engineering from the University of Texas at Austin in 1990 and 1994, respectively. Her email address is `Linda.F.Wilson@dartmouth.edu` and her web page can be found at `http://thayer.dartmouth.edu/~lwilson`.