5-30-2001

Improving a Brokering System for Linking Distributed Simulations

Thomas B. Stephens
Dartmouth College

Dartmouth College Computer Science Technical Report

TR2001-389

Improving a Brokering System for Linking Distributed Simulations

By

Thomas B. Stephens

Advisor: Linda F. Wilson

Department of Computer Science

Dartmouth College

Hanover, NH 03755

May 30, 2001

Abstract

The Agent Based Environment for Linking Simulations (ABELS) is a software framework designed to provide disparate simulations with dynamically updated data sources. It allows simulations and other agents to join a "cloud" of interacting producers and consumers of data. Once they have joined the cloud, they can publish services to other members and use methods published by others. This paper presents the initial design of a set of matchmaking components for the ABELS framework. These components dictate how services describe their abilities and requirements to ABELS. Furthermore, they help ABELS successfully match data producing services to the requests of data consuming clients.

We begin by describing a system for a data producing service to describe itself to the ABELS cloud, as well as a corresponding system for a data consumer to describe its needs. We then describe in detail the three components that make up the ABELS matchmaking system: the match ranker, which ranks a data producer's ability to fill the request of a data consumer; the thesaurus, which helps the match ranker recognize closely related terms; and the unit database, which allows participants in the ABELS system to translate between related data units. We also discuss how these basic components can be built upon and improved in future versions of the ABELS framework.

1 Introduction

Computer simulation models of real world systems have become extremely important as a means of testing theories and predicting future events. Transportation authorities use weather simulations to predict where icy road conditions may occur so that they can fight the hazards more effectively; firefighters use complex simulations to predict the paths of forest fires. As these simulations become more prevalent and complex, it will become increasingly useful to be able to access information generated by other simulations. For example, a simulation purporting to predict icy roadways will need information about weather conditions, such as air temperature, wind speed, and precipitation, that could easily be generated by another simulation or a remote sensor. Furthermore, many time-sensitive simulations could benefit greatly from a continuously updated repository of accurate information. Consider a simulation designed to predict the path of a forest fire. What if the simulation had access to information about wind speed that was updated in real time? From a programmer's perspective, it would be nice to be able to "plug in" to an information source that could generate accurate information as needed without having to hard-code the simulation so that it is forever tied to a single information source.

The Agent Based Environment for Linking Simulations (ABELS) is designed to provide disparate simulations with dynamically updated data sources [9 – 12]. It allows simulations and other agents to join a "cloud" of interacting producers and consumers of data. Once they have joined the cloud, they can publish services to other members and use methods published by others. To use the example provided above, the icy roadway simulation could find a producer of weather data to use in its own calculations, while a

highway traffic simulation might use the icy roadway simulation to help predict trouble spots on the roads that would affect traffic movement.

The design of ABELS is intended to be as easy on the simulation programmer as possible. A simulation that wants to take advantage of the cloud needs only to be able to communicate with its generic local agent (GLA) and be able to describe itself to the cloud; the ABELS system does the rest. This paper describes the components of ABELS that handle matching data consumers to appropriate data producers.

1.1 ABELS Overview

The ABELS system is composed of 4 main parts: user simulation objects, GLAs, helper agents, and the brokering system [9-12]. User simulation objects are the actual simulation applications that produce and consume data in the simulation cloud. Producers of data (sometimes referred to as "services") are those simulations or sensors that offer some data producing service to the cloud. Consumers of data are those programs that use some producer in the cloud to get the data they need. Most simulations in the cloud will probably act as both producers and consumers – that is, they will use some members of the cloud to obtain data while producing data for other members. Generic local agents (GLAs) are the user objects' window into the simulation cloud. Each user object has its own GLA which handles all interactions between the user object and the cloud. Helper agents are small programs that are capable of traveling across a network to perform some service for its owner in a remote location. The broker is the central authority of the simulation cloud. It monitors who joins and leaves the cloud and when.

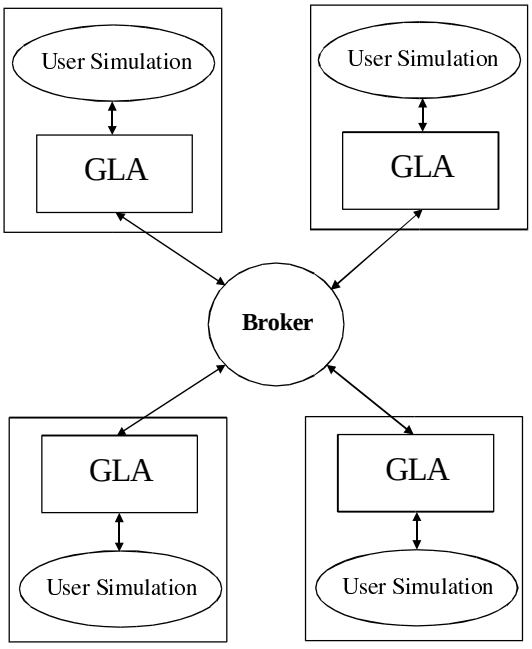

Figure 1: The ABELS Framework

In order to accommodate a wide range of platforms, the ABELS GLA and broker components are implemented using the Java programming language and Sun Microsystems' Jini architecture. Java is designed to run on virtually any computing platform, so a single version of the GLA can be used for user simulations on any platform. The broker uses Jini and Remote Method Invocation (RMI) to manage remote communications between itself and the GLAs. Figure 1 shows the interactions in the ABELS system.

1.1.1 User Simulation Objects

A user object can be nearly any application that produces or consumes information. Any agent that can generate some useful data can join the ABELS cloud as a producer. Examples of producers might include computer simulations of real world events or remote field sensors that can report real-time data. Similarly, any program that needs data may join the cloud as a consumer object. Some types of consumers could be computer event simulations that require externally generated data for processing. The only requirement of a user object is that it must be capable of socket communication with a GLA. In fact, custom interface additions could be used to provide this socket communication. Typically, user objects will be both producers and consumers.

1.1.2 Generic Local Agents

Generic local agents, or GLAs, are the user objects' interface to the ABELS simulation cloud. A user object communicates only with its own GLA, and the GLA handles all communications with the rest of the ABELS system. A GLA can publish information about data that a producer object can generate for other members of the cloud, or it can search the cloud for producers that can provide necessary information to a data consumer. Perhaps most importantly, a GLA can communicate with other GLAs to exchange information – so when a consumer wants to use a producer's published service, the consumer's GLA contacts the producer's GLA and requests the desired data. In order to accommodate the widest range of user simulations, the ABELS GLA is implemented in the Java programming language, so it is usable on any platform that can run a Java virtual machine.

1.1.3 Helper Agents

Helper agents, or HAs, are small executable routines that can perform some service on behalf of their owners. They are designed to be mobile on a network, so that an owner can send a helper agent to a remote location to perform some calculation, as opposed to bringing the data across the network to process locally. For example, an application that needs to know the median score of a list of grades might send a helper agent to the location that has the list so that the data can be processed at its source. Then only a single grade needs to be sent back across the network instead of the whole list. GLAs can use helper agents on the behalf of their simulation objects.

1.1.4 The Broker

The ABELS broker is a central lookup service for the simulation cloud. A producer's GLA can publish information about the services it offers by contacting the broker and submitting a description of itself. Similarly, a consumer's GLA can find the data needed by its user object by contacting the broker and searching for what it needs.

The broker uses Sun's Jini architecture to handle the complications of communication between the different entities on the network. Because the GLAs are implemented in Java, they can easily use Jini and RMI to communicate with each other and the broker. Jini offers some rudimentary query matching capabilities but nothing sophisticated enough to fit the requirements of the ABELS project. The current broker implementation organizes all of the known services into logical groups or categories based on the types of services they offer. For example, a service that generates data on ocean currents might be organized in the "ocean" group, while an air temperature simulation might be organized in the "weather" group. Currently, the broker can only search services by group, so a GLA that needs information about surface currents in the Indian Ocean can query the broker for a list of "ocean" simulations, but it cannot ask for anything more specific.

1.2 Matchmaking Components

The basic service searching functionality provided by the broker is not robust enough by itself to meet the functional requirements of the project. In the example used above, we would like the user to be able to ask specifically for a service that can produce data on water currents in the Indian Ocean. It would be better if ABELS could do some of the work by narrowing down the list of possible matches that the user must sift through to find an appropriate service. Therefore, the ABELS system uses a group of helper components, known as the matchmaking components, to aid users in the often difficult task of determining what producer services might be able to fill a consumer's request for data. The matchmaking components can examine the description of a data producer and determine how useful it would be to a particular consumer. The three matchmaking components are the match ranker, the thesaurus, and the unit database.

1.2.1 Match Ranker

The match ranker component is responsible for examining consumer queries and comparing them to the descriptions of data producing services. It helps users determine which producer services can best fill the requests of their consumer objects. The match ranker uses the unit database and thesaurus to determine a relevance score for each producer based on how closely it matches the needs of the consumer. Currently, each GLA has its own instance of a match ranker that is responsible for refining the broker's high level search. This design distributes some of the processing work away from the central broker to prevent a performance bottleneck.

1.2.2 Thesaurus

The thesaurus module handles lists of keywords and synonyms for those words.

The matchmaker uses the thesaurus to help expand its matching abilities. The thesaurus keeps track of associations between closely related words so that the matchmaker can match these closely related terms in a search. For example, the keyword "cat" might be associated with the words "kitty", "kitten", and "feline". In that case, a consumer object searching for services that are related to cats would get matching results that have to do with kitties, kittens, and felines as well.

1.2.3 Unit Database

The unit database is a knowledge base of data units used in the simulation cloud. Users can enter and edit information about the units that their simulations use and provide conversion information for the units of interest. The matchmaker uses the unit database to determine if the information returned from a particular service could be used to fill a consumer's query. For example, users might specify that x degrees Fahrenheit is equal to $(5/9)*(x-32)$ degrees Celsius. In that situation, any user searching for a service that reports a temperature in degrees C could receive matches that reported the temperature in degrees F, along with appropriate conversion modules to seamlessly apply the transformation to get the information into the correct format.

1.3 Goals

The goal of this project was to take significant steps toward developing a fully functional matchmaking module for the ABELS system. Realistically speaking, it will take years to achieve the level of sophistication and independent matching ability that would be ideal for a matchmaking module of this type. Therefore, the focus of this project was to design an extensible and flexible system that can easily evolve and mature as the ABELS project moves forward. We set out to put enough thought into the design of the components that they will be able to handle significant improvements without requiring many design changes. Specifically, we intended to design a limited matchmaker that could rank service descriptions according to relevance to a consumer query. The ranking system was to take advantage of simple unit conversion information and keyword lookup functionality. Most importantly, the matchmaker was to be designed in a way that would allow significant improvements to be made within the current design; in other words, improving the system should not necessitate redesigning it.

We believe that we have achieved, and in some cases surpassed, each of the goals of the project. The current matchmaker implementation is very modular and flexible to allow for easy improvements to the module. The current matchmaker uses a unit database module to provide sophisticated unit conversions and a thesaurus module to provide keyword lookups. Since each of these components is designed modularly, they are individually and independently upgradable. Furthermore, the implementation of the match ranker makes it easy to change the ranking criteria or the service description format, or even the ranking algorithm itself, all independently of each other.

2 Background Work

We considered several pre-existing technologies and algorithms for describing and

ranking services in the ABELS system. We investigated three paradigms for service descriptions: structured description languages, a custom XML specification, and custom Java data structures. Each option had benefits and drawbacks, but in the end we decided to use a custom XML specification with a complementary set of custom Java data structures. We also examined methods for ranking and indexing text descriptions, although no researched algorithms were implemented in this version.

2.1 Description Languages

Our first investigation was into so-called "description languages" such as KQML/KIF [2,4] and FLBC [3,4]. These languages were designed to represent knowledge and beliefs in a generic way to facilitate communication between disparate computer systems. KQML is a messaging language that uses KIF to describe the content of its messages. FLBC is an attempt to improve upon the capabilities of KQML/KIF as described below.

KIF (Knowledge Interchange Format) is a predicate calculus based language for generic representation of knowledge between different computer programs. It is designed to facilitate interaction between computer programs that were developed in different ways and that do not follow the same conventions. According to the designers, it was intended to provide a language that would be a mediator in the translation of other languages. Basically, if you could translate an expression in language A into KIF and vice versa, and you could translate an expression from language B into KIF and vice versa, then you could translate expressions from A to B and from B to A via KIF.

KIF is fairly robust in its quest to generically describe information. It is capable of describing relationships between pieces of data. For example we could relate the price of 2 different cars as follows:

```
( > (price Porsche) (price Honda))
```

This asserts that the price of a Porsche is greater than the price of a Honda. The language is also capable of expressing interest in a particular type of information or relationship. If we were looking for information on the price of Porsche cars, we could represent the interest as follows:

```
(interested tom '(price Porsche))
```

Furthermore, the language is capable of defining and representing simple procedures for agents to follow. For example, the following KIF message tells the recipient to print the message "Hello" followed by a new line [8].

```
(progn (fresh-line t)
       (print "Hello!")
       (fresh-line t))
```

KQML (Knowledge Query and Manipulation Language) [2] is a language protocol intended to allow communication between disparate computer systems. KQML uses KIF to describe much of the information that it passes.

The Knowledge Interchange Format specification explicitly states that it is not

intended to be used as an internal representation of information within a computer program, or for communication between closely related programs [8]. It seems that the broker – GLA pair would qualify as "closely related programs". In other words, KIF does not seem to be appropriate for use within the broker or matchmaker.

KIF is designed to handle the publishing of data to an unknown system. In other words, if our computer simulation could generate data about ocean currents in the Gulf of Mexico, we might want to report that information using KIF so that other KIF capable applications could read the data and understand it. However, it is unclear how KIF could be used to easily tell the other applications exactly what kind of data it was reporting. The GLA / broker system of the ABELS project is designed to eliminate the need for this type of generic data format. The standard data format is determined via Java functions. It would be feasible to use KIF definitions to explain the structure and content of data available in the cloud, but the technology is targeted at addressing a different problem.

Scott Moore's FLBC (Formal Language for Business Communication) [3,4] is an attempt to improve upon DARPA's KQML/KIF efforts. FLBC provides a relatively robust messaging system but, much like KQML and KIF, its intended use is connecting diverse applications. Scott Moore goes to great lengths to show that his FLBC language is better and more robust than KQML. In fact, he provides translations for several of KQML's standard performatives to show that FLBC can do everything that KQML can do. However, the technology is still not entirely appropriate for describing simulation specifications within a controlled system such as ABELS.

As mentioned above, the investigated description languages are slightly misdirected for use in describing an ABELS service. It is possible to have users describe their data producing services and specify their consumer queries using one of these languages. Such an implementation would have the benefit of a plain ASCII representation of the specification of the service so that the data could be easily saved, modified, and passed across socket connections to be read on multiple computer architectures. However, because of the Jini-based design of the ABELS system, all communications between the GLA and broker are over RMI connections. This eliminates the need for socket-friendly ASCII description languages; it is just as easy to pass a Java object across RMI as it is to pass an ASCII stream. Furthermore, the aforementioned languages are not widely supported. In fact, the only projects in which they seem to have been used are those written by the authors of the respective languages. Finally, because these languages are not widely used, there are no pre-existing tools to manipulate descriptions written in these languages. Therefore, we would have to parse the descriptions anyway in order to use them. In short, using a description language such as FLBC or KQML / KIF would impose many extra restrictions on the system and necessitate more work for the broker and GLA, but would not provide any real benefits that we could not get using custom Java data structures.

2.2 Custom XML Specification

Another option available is to design a custom XML (Extensible Markup Language) specification to represent the data we need to describe in a service. This

approach involves developing a customized vocabulary of tags (sometimes referred to as a data type definition or DTD) that would describe all of the information needed about each service, and populating an XML file with the appropriate data. The XML description can be passed across the network in a relatively efficient way, and examined either directly or after parsing. This approach has the advantage of easy long term storage of description parameters, and easy editability of those parameters without the help of a user interface. Furthermore, it would be easy to augment the DTD to accommodate future functionality in the system. There are many existing tools available to manipulate XML files so generating and parsing the descriptions is relatively easy. The only potential drawback is that we might require an extra parsing step before examining the data, but it would also be possible to examine the XML string directly without parsing.

2.3 Custom Java Data Structures

The most straightforward representation would be to use custom Java classes that define fields for each piece of information needed about a service. A graphical user interface could collect each piece of information needed and store it in an instance of a custom class. The object would be easily passed via RMI to and from the broker, and would be easily manipulated by the ABELS components. The object could be serialized to disk for long term storage. On the down side, it would be difficult to recover the serialized objects after a version change. That is, if there were ever any changes to the structure of the custom class (i.e. to add functionality to the system) it would be very difficult to recover the serialized instances of the old class. Changing data in the saved description would require the graphical user interface. Furthermore, it would probably be less efficient to pass Java objects across the network as opposed to plain text.

2.4 Current Solution

Our current solution is to use a custom XML specification with complimentary Java data structures. The service descriptions are stored and passed around the network as XML strings to take advantage of the size efficiency and persistence of the ASCII representation. The matchmaking components parse the XML into a custom Java object for processing. In other words, service descriptions are stored and passed across the network as strings of ASCII text, but they are parsed into a custom Java data structure before they are ranked by the matchmaking components. Therefore we take advantage of the benefits of both the designs; we get storability and size efficiency from using XML, and straightforward processing algorithms from using Java classes.

3 Architecture

As shown in Figure 2, the original ABELS design placed all of the matchmaking components together as a helper module for the broker. It dictated that rankings would be calculated on the broker's virtual machine as part of the initial registration process. However, this design imposed several potentially serious bottlenecks on the system. At startup, a simulation cloud will have no registered services; in other words, the broker is empty. Slowly GLAs will begin connecting to the broker, some to publish producer

descriptions and others to search for consumer data. When a consumer connects, the broker would initially rank all known data producing services and return the list to the GLA. However, when a new producer joins the cloud, the broker is responsible for notifying each of the registered consumers. In order to rank this new service for the consumers, each registered consumer must contact the matchmaker to re-submit its query. This means many virtually simultaneous remote method calls to the matchmaker, which would consume significant network bandwidth and processing resources on the broker machine. In terms of performance, each of *n* consumers would make simultaneous remote requests to the matchmaker to re-submit their queries, so the network bandwidth cost of registering a producer would be O(*n*). Since each ranking request requires the submitted query to be compared to each of the *m* registered producer methods, the registration of a new producer is O(*m***n*) with respect to processing resources on the broker machine. Even on a high bandwidth network with a fast server to run the broker, this design would not scale well.

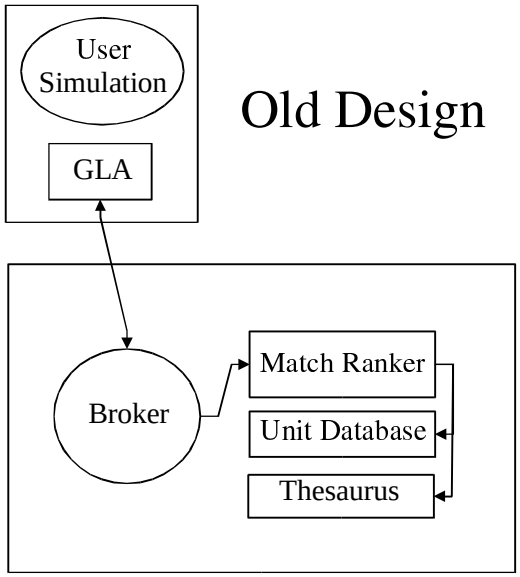

Figure 2: Original Matchmaker Design

Alternatively, we could have required that the broker rank each new producer before notifying the consumers of its arrival. This would only cut down on the network traffic by a constant factor because each of the *n* consumers would still have to be notified of the new service via a remote method call, so network usage would still be O(*n*). Processing time on the broker would be significantly reduced because each of the *n* consumer queries would need to be compared only to a small set of producer methods. If the new producer service registers *p* new methods, it would take O(*n***p*) processing time to rank the new methods for all of the consumers. The drawback of this design is that the broker must store all of the consumer queries for use each time a new service registers.

To help spread its processing load, the match ranking module was redesigned to run as a local service to each GLA as shown in Figure 3 . In other words, each GLA has

its own copy of the match ranker. Currently, when a new consumer registers its query with the broker, it receives an unranked list of the available producer services in the cloud. The services are then ranked using the resources of the local GLA so that the central broker is not burdened with the task of calculating each rank.

This design alleviates the network bandwidth and processing bottlenecks of the original system. While the network bandwidth cost of registering a new producer is still O(*n*) because of the required *n* remote notifications, it is a constant factor better than the original design because it doesn't require a callback to the broker, and the broker must send less information to the consumer because only the producer descriptions, and not ranking information, need to be sent across the network. The registration operation requires very little processing on the broker; instead, the required computations are spread across all of the consumer machines. For a registration of p new consumer functions, each of the *n* consumers will require O(*p*) processing time. This is much improved over the centralized computation of the old design.

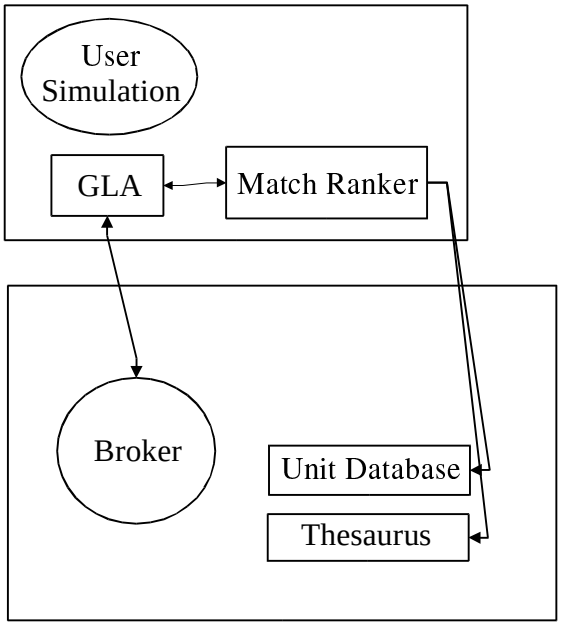

Figure 3: Current Matchmaker Design

The thesaurus and unit database modules are used as centralized knowledge repositories and are accessed by the matchmakers via RMI calls. This is necessary because the knowledge contained in these two components must be shared among all participants in the cloud. While this strategy creates network traffic problems of its own, they are minimized by caching significant information from the central databases in the match rankers. The localized match rankers will be comparing the same query to different service descriptions over and over again. Therefore, they will using the same unit conversions and keyword descriptions for each comparison. Because of this pattern we can limit the RMI calls to the central database components to be relative to the number of

changes made to the database during the life cycle of a GLA. The match ranker needs only to look up each query keyword and each query unit once until the next time the database changes. Each GLA can be notified by the central databases of any changes so that it can clear the matchmaker's cache.

4 Matchmaker Implementation

The current matchmaker implementation describes ABELS services as follows. A service is described by the set of methods that it publishes to the cloud. Each method is described by a natural language description as well as the set of input parameters required and the output returned. Each parameter is described by the following:

1. A natural language description of the data represented by the object. This description is used primarily by the matchmaker for keyword matching and will be available to users if human intervention is required to select a match.
2. A data type definition. This could be a simple data type such as an integer or floating point number, or a complex type definition that includes several nested objects. In the case of the latter option, nested objects can be described using the same structure as parameters.
3. An optional unit specification to clarify the meaning of numerical measurements. The unit specification is used in conjunction with a database of unit conversions to allow close matches to succeed. For example, with the appropriate knowledge in the conversion database, a service that produces rainfall data in inches can be seamlessly matched to a query for rainfall information in centimeters.

For example, a method that reports wind speed at a given latitude and longitude could be described as follows:

- Description: Returns the last measured wind speed at the specified location.
- Input – Latitude: The latitude of the location of interest. Measured in degrees.
- Input – Longitude: The longitude of the location of interest. Measured in degrees.
- Output: The wind speed at the specified location. Measured in kilometers per hour.

The match ranker takes a query from a data consumer and ranks each available data producing method based on the descriptions provided. Consumer queries are described in the same way as producer methods; that is, they have a description and input and output parameters as described above. When a data consumer submits a query, the match ranker examines each available method of each related service. The broker uses high level categories to limit the number of services ranked, so that only methods that fit the requested category are examined. Each of these methods is ranked according to its similarity to the query.

This implementation assumes that an ABELS service is merely the sum of its available methods and that a method can be precisely described by its inputs and outputs. (Note that it is very likely that future implementations will want to take other information

into account in its ranking process, such as the average response time or network latency. The design of the current matchmaker implementation makes it easy to add or modify the way in which services are ranked).

The matchmaker is modularized in a way that complements the structure of the information that it examines. The three central components that handle most of the functionality of the matchmaker are described by the Java modules: `MatchRanker`, `UnitDB`, and `Thesaurus`. The `Thesaurus` module handles keyword aliasing, just as a real thesaurus would. The `MatchRanker` uses the thesaurus to look up synonyms of keywords so that a user searching for "cat" will get back matches that have associated keywords "cats", "kitten", "kitty", etc. The `UnitDB` is responsible for unit translations. Users can define commonly used units such as "meters", "feet", "centimeters", or "kilometers", and conversion equations to go from one unit to another. The matchmaker uses these conversion rules to match objects of differing but compatible units. The match ranker is the central component of the matchmaker; it implements the ranking function that generates a numeric rank to represent the appropriateness of the service for the query. It uses the thesaurus and unit database modules to help determine if a given service method can fill a particular request.

4.1 Data Structures

The implementation of the ABELS matchmaking components makes use of several data structures that merit special explanation. The following utility classes are central to the functionality of the matchmaking components.

4.1.1 PersistentHash

The `PersistentHash` class is a hash table implementation that actively saves all changes to disk as they happen. It is an easy solution for situations where database persistence would be nice but would be too much trouble to implement. We use a variation of this persistent hashtable to implement the graph structure that holds the unit database. We also use the persistent hashtable to implement the thesaurus component. It provides database-like functionality without requiring as much initialization.

The `PersistentHash` object writes all changes to permanent storage on the hard disk, but also caches data in an in-memory read cache to improve efficiency. Because of the cache, the performance is asymptotically similar to that of a `java.util.Hashtable`, with O(1) performance for most operations.

A directory specified at creation time stores the key/value pairs in long-term persistence. When a key / value pair is added to the hash table, it is stored as a pair of files. The hash code of the key (as determined by the `Object.hashCode()` function) is used as the base filename. Since the hash code is an integer, there is no need to worry about file name encoding issues. The key object is written in the file "<hashcode>.key" and the value is written in the file "<hashcode>.val", both as serialized Java objects.

Since the `PersistentHash` object extends a standard Java hash table, we use the base object as the read cache. At construction time, the hashtable reads in all file

names in the directory to cache the known keys. These keys are assigned a value of `CACHE_MISS` to indicate that the true value has not yet been loaded. When a `CACHE_MISS` is encountered in a get operation, the appropriate file is read from the disk and cached for future use. The creation of a persistent hashtable is the only operation that requires O(*n*) processing time. For a new table, *n* is zero, so this is not a problem, but re-creating a table with existing information requires O(*n*) time to read all of the keys into the cache.

As noted above, this persistent hash table implementation is capable of O(1) performance on all basic operations (put, get, remove). Constant time overhead is incurred from the disk I/O associated with writing a new value in a put operation or a cache miss in a get operation; however, these do not affect the asymptotic performance of the data structure. "Put" operations incur the extra overhead of a disk write, but the overhead is proportional to the size of the value object being added to the table and not to the number of entries in the table. Some "get" operations also incur the added cost of a disk read when the requested object is not in the cache, but again, this added cost is proportional to the size of the data value being read and not to the size of the data structure.

4.1.2 QuickGraph

The `QuickGraph` object is a simple and fast implementation of a weighted, directed graph. Each vertex of the graph is designated by a unique object, and each directed edge can hold a weight object. Vertices are stored as keys in a hash table. The value associated with each vertex key is another hashtable that represents the edges originating at the respective vertex. Such an edge hash table has keys corresponding to destination vertexes, and associated values corresponding to the weight of the edge. Two variations of the `QuickGraph`, the `PersistentGraph` and the `CachingGraph`, enhance the graph's functionality slightly. The `PersistentGraph` uses `PersistentHash` objects for storage instead of standard hash tables, so that all data in the graph is seamlessly persisted to permanent storage. The `CachingGraph` uses an in-memory cache to remember the result of expensive computations. Specifically, all information generated by breadth first searches on any node are cached so that further operations that require a breadth first search on the node will take constant time. To ensure that the data in the cache is accurate, the cache is cleared whenever the graph is modified. The caching capabilities of this variant can improve the performance of the graph greatly in situations where there are many more breadth first search related operations than there are modifications to the graph, as is the case with our unit database component.

We use the caching implementation of the persistent graph for our unit database. The graph structure provides a combination of performance and functionality that would be difficult to get from a relational database system such as MySQL.

4.1.3 Equation

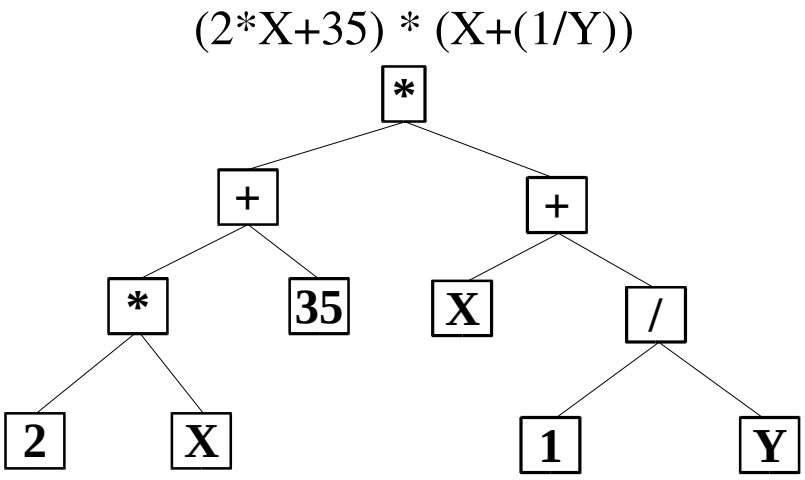

*Figure 4: The Equation (2*X+35)*(X+(1/Y))*

An `Equation` object is a tree-based representation of a mathematical expression as shown in Figure 4. Currently, our math library supports addition, subtraction, multiplication, division, parentheses, and arbitrarily named variables. Standard infix representations of mathematical expressions can be parsed into an equation tree. Each node of the tree contains either a number (implemented as a `java.lang.Number`), an operator (+,-,*,/,()), or a variable. Anything that is not recognized as a number or an operator is treated as a variable. Equations can be evaluated by specifying a hash table of values for the variables in the expression. The tree is recursively evaluated as follows: numerical nodes take on the value of the number they contain. Variable nodes take on the value of the variable they contain, as specified by the table of input parameters. Operator nodes take on the value of the operator they contain applied to the values of their child nodes. In other words, a node containing a "+" with the two child nodes "2" and "3" would have a value of 5. Equations can be chained together by replacing variable nodes with other equations. This effectively amounts to replacing the tree node that has the variable with the root of the equation that should be chained. The result is to replace the variable with the value of the equation. This comes in handy for handling multiple step unit conversions, as described below.

4.2 Thesaurus

The thesaurus interface defines simple methods to store and retrieve lists of relevant similar words associated with a given keyword. For example the keyword "cat" might have the list "kitten","kitty","feline" associated with it in the thesaurus. The default implementation in version 1.0 of the matchmaker is a simple table of keywords where each

word in the table has an associated list of similar words. This implementation takes advantage of the `PersistentHash` class described above to automatically handle persistent storage of the data in the thesaurus. It provides fast performance – O(1) time for most data retrieval operations – at the cost of size efficiency. Since each keyword has its own list of associated words, there will probably be a lot of duplicate storage of words. For example, the word "strong" might be associated with many keywords, in which case a copy of the string "strong" would be stored for **each** of those keywords.

A more size-efficient implementation would be a graph structure with keywords stored at the vertices and edges connecting words that are associated with each other. This way very little word duplication would occur. Keyword lookups could also find "deep" matches that crossed multiple graph edges by using a breadth first search on the keyword node of interest. On the other hand, the performance cost of doing these "deep" searches on the thesaurus would probably outweigh the benefits. If the graph grew to more than a few thousand keywords, the breadth first search could be prohibitively slow; furthermore, deep keyword matches would not necessarily be desirable in the context of the matchmaker. It would be expected that each level of the breadth first tree generated in a traversal of the thesaurus graph would be decreasingly similar to the original word. In other words, by the time the tree is three or four levels deep, the words being connected could have very little to do with the original word.

Another potential implementation could take advantage of a relational database system, such as MySQL. Keywords could be stored in a table along with a unique integer id, and a join table could be used to string together lists of associated words. The KEYWORD table would look something like: `TABLE KEYWORD (ID INT PRIMARY KEY, WORD CHAR[40] NOT NULL UNIQUE)`, and the join table would be: `TABLE WORDJOIN (KEY_ID INT NOT NULL, JOINED_ID INT NOT NULL)`. Then we could generate keyword lists for a given word by issuing the following SQL command: `"SELECT K2.WORD FROM KEYWORD K1, KEYWORD K2 ,WORDJOIN WJ WHERE K1.WORD LIKE 'foo' AND K1.ID = WJ.KEY_ID AND WJ.JOINED_ID = K2.ID"`. This implementation would make good use of storage space, and performance, although greatly dependent on the database, would be expected to be good. Deep searches would not be very efficient for the database, but as mentioned above, they are not necessarily desirable for the matchmaker application.

4.3 Unit Database

The unit database module allows the matchmaker to translate between closely related units. Specifically, it stores conversion information about the real world units that are used in function parameters. It stores names of units and equations that it can use to convert between them. For example, the unit database might know that 1 foot = 12 inches, in which case a user simulation that needs some measurement in feet would be able to use a service that gave the measurement in inches. Furthermore, the matchmaker can use the conversion rules to determine compatible units while matching service queries. The unit database uses the equation structure described above to represent simple mathematical equations that are used to convert units. Equations can be parsed from

standard infix algebraic expressions (for example: "18*(x+2)") The equation engine supports addition, subtraction, multiplication, division, and infinite levels of parentheses to control the order of operations. Furthermore, any tokens not recognized as operators or numbers are treated as variables that can be initialized when the equation is evaluated. The unit conversion engine expects each conversion equation to have a single variable "X" that designates the amount of the starting unit. For example, a conversion equation from centimeters to meters would be "X/100"; similarly, an equation to convert from kilometers to meters would be "1000*X". When asked to convert 143 kilometers into meters, the equation simply plugs the number of kilometers (143) into the variable "X" and the output (143,000) is the corresponding number of meters. Furthermore, variables can be used to nest equations inside each other. For example, if we know that "1000*X1" converts kilometers to meters, and we know that "100*X2" converts meters to centimeters, then we can assign to "X2" the value "1000*X1" to compose an equation ("100*(1000*X1)") that will convert from kilometers to centimeters.

The conversion and equation data structures are designed to be easily incorporated into small agents that would be able to handle conversions seamlessly for a user simulation. The required functionality of such a conversion agent would be merely to execute a specific remote method and apply specified conversions to the necessary input and output parameters. Such an agent could be instantiated by the GLA for each service match that needed unit conversions. Conversion equations could be pulled out of the unit database module and stored by the agent for use whenever the host simulation wanted to use the remote service. Currently, the local GLA handles unit conversions for consumer objects.

The current implementation of the unit database uses a graph to store units and conversions. A "unit" object consists of nothing more than a name for that unit, so for simplicity we store only the string name of each unit in the vertices of the graph. Conversions are represented by weighted, directed edges connecting the vertices. The weight of each edge is a stored equation that will convert from the starting node unit to the ending node unit. With this design we can perform infinitely deep searches on the graph. In other words, if we know a conversion from A->B, and we know a conversion from B->C, then using a breadth first search and the equation composition described above, we can generate a conversion from A->C.

The graph implementation uses a persistent hash table to store vertex values as keys. Each vertex has an associated hash table that stores edge weights hashed with a key value of the destination node. This double level hash table performs similarly to an adjacency matrix implementation because edge lookups can be performed in O(1) time. We can therefore perform the aforementioned "deep searches" on the unit database in linear time with respect to the number of conversions in the database [1]. Furthermore, we get constant time inserts and deletes from the database as well as constant time shallow searches.

The implementation of the class `matchmaker.unit.DeepUnitDB` uses the cached persistent graph implementation described above to ensure data persistence. All data in the unit database is automatically persisted to long term storage as soon as it is

updated in memory. The persistent graph implementation reduces performance slightly because of the overhead required for disk writes, and because of occasional cache misses during get operations. The performance reduction is by a constant factor, so the module still scales well. The CachingGraph structure used for the module caches the information generated by each breadth first search, so in the absence of modifications to the database, we can perform deep searches in constant time. In the context of the matchmaker, this improves performance significantly because for each query to the matchmaker we will typically generate several unit searches. If the information is saved so that the breadth first tree only needs to be calculated once, we save considerable processing time.

4.4 Match Ranker

The match ranker module is responsible for generating a numeric rank that represents the pertinence of a single service method with respect to a query. When a GLA is notified of a new service, it uses the match ranker to compare its consumer query to each of the new method descriptions. The match ranker returns a floating point rank valued from 0-1 representing the similarity of the two inputs, and information describing any unit conversions needed to plug the two services together.

The current match ranker implementation uses simple string comparisons to rank possible query matches. We start with a total score of zero and a comparison count of zero, and apply the following algorithm: For two given service method objects (call them "service" and "query" to denote the possible service match and the input query parameters, respectively) we compare each word in the query description to the service description. If the word from the query (or any of the word's synonyms) exists in the service description then we have a match. For each comparison we increment the comparison count by 1, and for each match we increment the total score by 1. Next we compare each input parameter in the query to the input parameters of the service. The descriptions of each object are compared just as the method descriptions were compared, and unit specifications are compared (using the unit database) to see if the service is compatible with the query. Since input and output parameters seem to be more significant attributes of a method than a textual description, they count for more in the final rank of the method. When input and output parameters are compared the comparisons and matches count double towards the final score. In other words, each comparison adds 2 to the comparison count, and each match adds 2 to the total score.

The match ranker uses both the thesaurus and unit database modules to assist in ranking queries. When description parameters are compared, each keyword in the query specification is looked up in the thesaurus to find similar, related keywords that might appear in the description of the service object. If any of the related words are found, the comparison is scored as a match, just as it would if the original keyword had been found. When parameter objects are compared, the query unit is looked up in the UnitDB and a list of all compatible units is generated. If the service object uses a compatible unit, the comparison is scored as a match.

At the end of the comparison algorithm, we have a total score that is less than or equal to the total number of comparisons. The rank for the method is calculated as the

total score divided by the total number of comparisons.

5 Conclusions and Future Work

The final vision of the ABELS matchmaking system is one which can seamlessly rank and match services to queries without significant user intervention, and can learn from the choices that users make when intervention is required. Although we are still a long way from that goal, we have made significant progress with version 1.0 of the matchmaking components. The system cannot yet make informed enough decisions to act completely without user intervention, but with improvements to the ranking algorithms this functionality is certainly possible. The matchmaker could simply decide on the highest ranked service whenever a decision is needed. The thesaurus and unit database components take steps towards the goal of a "smart" matchmaker that would learn from user input. The thesaurus and unit database grow to learn appropriate conversions and keywords as users add them. The user interface could automatically prompt users for updates to the databases whenever they make choices that the matchmaker did not expect. For example, if a user choses a service with a relatively low rank, the interface could prompt her to explain the decision, perhaps by specifying a new conversion or keyword association.

As mentioned above, the match ranker's ranking algorithm could be improved by more carefully weighting different pieces of service descriptions. The current implementation weights certain criteria more than others; for example, the textual description of a method as a whole counts for less than the description of a specific parameter. This is because one underlying assumption of the design of the description format is that a method's functionality can be described by its inputs and outputs. In other words, the information in the general description is expected to be duplicated in the descriptions of the inputs and outputs. There could be better ways of weighting these query criteria. In fact, different groups of simulations might work better with different ranking algorithms and weights. This is another area that the match maker could be able to learn from user choices. The future matchmaker might be able to dynamically adjust its ranking weights based on user feedback.

Version 1.0 of the match maker ranks services based solely on relevance; however, it would be better to provide more information about service matches than just a relevance rating. For example, it would be good to also rate the performance of different services. Most users would probably prefer a service whose average response time is 10 ms over an equivalent service that needs 15 seconds to process. In future versions of the match maker we could provide such performance rankings in terms of network lag, server load, or average response time. This functionality could be easily incorporated by extending the match maker to include a module that would keep track of these performance issues and report them. Also, the thesaurus module could track relevance ratings of words in the association list. The match ranker could use these ranks to more accurately gauge the relevance of certain keywords. Instead of scoring the same rank for any keyword match, some keywords could count for more than others. For example, matching the word "to" in a query should probably count for less then matching the word "Atlantic" or

"Temperature". Instead of building our own terminology database from scratch, we could use a pre-populated lexical database such as WordNet [13]. Such a product would simplify the process of building a thesaurus for use in searches because it comes pre-populated with relationships between thousands of words in the English language.

The unit database could be improved to rate unit conversions based on aspects such as conversion complexity or precision. Most users would rather use a conversion that preserved 10 significant digits over one that preserved only 2. This could be implemented by adding a "precision" field to data descriptions and unit conversions. If the precision of a conversion was less than the precision required by a query, then the match would be ranked lower than if the precision requirement was met.

References

[1] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press 1990.

[2] KQML Advisory Group. "An overview of KQML: A Knowledge Query and Manipulation Language". March 2, 1992.

[3] Scott A. Moore and Steven O. Kimbrough. "Message Management Systems at Work: Prototypes for Business Communication". *Journal of Organizational Computing* 5:2, pp. 83-100.

[4] Scott A. Moore. "KQML & FLBC: Contrasting Agent Communication Languages". Proceedings from the 32nd Hawaii International Conference on System Sciences 1999.

[5] Gerard Salton and Chris Buckley "Term Weighting Approaches in Automatic Text Retrieval". November 1987. http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR87-881

[6] Gerard Salton and R. K. Waldstein. "Term Relevance Weights in On-Line Information Retrieval". July 1977. http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR77-316

[7] Greg Travis. "The Persistent Hashtable: a Quick-and-Dirty Database". http://softwaredev.earthweb.com/java/sdjjavaee/article/0,,12396_600531,00.html

[8] UMBC Knowledge Sharing Effort web site. http://www.cs.umbc.edu/kse/kif

[9] Linda F. Wilson, George Cybenko, and Daniel Burroughs, "Mobile Agents for Distributed Simulation", Proceedings of the High Performance Computing Symposium (HPC '99), pp. 53-58, 1999.

[10] Linda F. Wilson, Daniel Burroughs, Jeanne Sucharitaves, and Anush Kumar, "An Agent-Based Framework for Linking Distributed Simulations", Proceedings of the 2000 Winter Simulation Conference, pp. 1713-1721, 2000.

[11] Linda F. Wilson, Daniel J. Burroughs, Anush Kumar, and Jeanne Sucharitaves, "A Framework for Linking Distributed Simulations Using Software Agents", Proceedings of the IEEE, vol 89, no. 2, pp. 186-200, February 2001.

[12] Linda F. Wilson, Jeanne Sucharitaves, Anush Kumar, and Daniel J. Burroughs, "The ABELS Framework for Linking Distributed Simulations Using Software Agents", Proceedings of the IASTED International Symposia Applied Informatics, Symposium 2: Networks, Parallel and Distributed Processing, and Applications, pp. 51-58, 2001.

[13] WordNet web site. http://www.cogsci.princeton.edu/~wn/