KQML - A Language and Protocol for Knowledge and Information Exchange

Tim Finin and Rich Fritzson
Computer Science Department
University of Maryland, UMBC
Baltimore MD 21228

Don McKay and Robin McEntire
Valley Forge Engineering Center
Unisys Corporation
Paoli PA 19301

Abstract. This paper describes the design of and experimentation with the Knowledge Query and Manipulation Language (KQML), a new language and protocol for exchanging information and knowledge. This work is part of a larger effort, the ARPA Knowledge Sharing Effort which is aimed at developing techniques and methodology for building large-scale knowledge bases which are sharable and reusable. KQML is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to share knowledge in support of cooperative problem solving.

KQML focuses on an extensible set of *performatives*, which defines the permissible operations that agents may attempt on each other's knowledge and goal stores. The performatives comprise a substrate on which to develop higher-level models of inter-agent interaction such as contract nets and negotiation. In addition, KQML provides a basic architecture for knowledge sharing through a special class of agent called communication *facilitators* which coordinate the interactions of other agents The ideas which underlie the evolving design of KQML are currently being explored through experimental prototype systems which are being used to support several testbeds in such areas as concurrent engineering, intelligent design and intelligent planning and scheduling .

Introduction

Many computer systems are structured as collections of independent processes these are frequently distributed across multiple hosts linked by a network. Database processes, real-time processes and distributed AI systems are a few examples. Furthermore, in modern network systems, it should be possible to build new programs by extending existing systems; a new small process should be conveniently linkable to existing information sources and tools (such as filters or rule based systems).

The idea of an architecture where this is easy to do is quite appealing. (It is regularly mentioned in science fiction.) Many proposals for intelligent user-agents such as Knowbots [Kahn] assume the existence of this type of environment. One type of program that would thrive in such an environment is a mediator [Wiederhold]. Mediators are processes which situate themselves between "provider" processes and "consumer" processes and perform services on the raw information such as providing standardized interfaces; integrating information from several sources; translating queries or replies. Mediators (also known as "middleware") are becoming increasingly important as they are commonly proposed as an effective method for integrating new information systems with inflexible legacy systems.

However, networks environments which support "plug and play" processes are still rare, and most distributed systems are implemented with ad hoc interfaces between their components. Many Internet resources, such as library catalog access, *finger*, and menu based systems are designed to support only process-to-user interaction. Those which support process-to-process communication, such as ftp or the Mosaic world wide web browser, rely on fairly primitive communication protocols. The reason for this is that there are no adequate standards to support

complex communication among processes. Existing protocols, such as RPC, are insufficient for several reasons. They are not all that standard; there are currently several successful and incompatible RPC standards (e.g. ONC and DCE). They are also too low level; they do not provide high level access to information, but are intended only as "remote procedure calls."

Nor are there standard models for programming in an environment where some of the data is supplied by processes running on remote machines and some of the results are needed by other similarly distant processes. While there are many ad hoc techniques for accomplishing what is needed, it is important that standard methods are adopted as early as is reasonable in order to facilitate and encourage the use of these new architectures. It is not enough for it to be possible to communicate, it must be easy to communicate. Not only should low level communication tasks such as error checking be automatic, but using and observing protocol should be automatic as well.

KQML is a language and a protocol that supports this type of network programming specifically for knowledge-based systems or intelligent agents. It was developed by the ARPA supported Knowledge Sharing Effort [Neches 91, Patil 92] and separately implemented by several research groups. It has been successfully used to implement a variety of information systems using different software architectures.

The Knowledge Sharing Effort

The ARPA Knowledge Sharing Effort (KSE) is a consortium to develop conventions facilitating the sharing and reuse of knowledge bases and knowledge based systems. Its goal is to define, develop, and test infrastructure and supporting technology to enable participants to build much bigger and more broadly functional systems than could be achieved working alone.

Current approaches for building knowledge-based systems usually involve constructing new knowledge bases from scratch. The ability to efficiently scale up AI technology will require the sharing and reuse of existing components. This is equally true of software modules as well as conceptual knowledge. AI system developers could then focus on the creation of the specialized knowledge and reasoners new to the task at hand. New systems could interoperate with existing systems, using them to perform some of its reasoning. In this way, declarative knowledge, problem solving techniques and reasoning services could all be shared among systems. The ability to build, manage and use sharable and reusable knowledge resources is thought to be a key to the realization of large-scale intelligent systems. The definition of conventions enabling sharing among collaborators is the essential first step toward these goals.

The KSE is organized around four working groups each of which is addressing a complementary problem identified in current knowledge representation technology:

- The Interlingua Group is concerned with translation between different representation languages, with sub-interests in translation at design time and at run-time.
- The KRSS Group (Knowledge Representation System Specification) is concerned with defining common constructs within families of representation languages.
- The SRKB Group (Shared, Reusable Knowledge Bases) is concerned with facilitating consensus on the contents of sharable knowledge bases, with sub-interests in shared knowledge for particular topic areas and in topic-independent development tools/methodologies.
- The External Interfaces Group is concerned with run-time interactions between knowledge based systems and other modules in a run-time environment, with sub-interests in communication protocols for KB-to-KB and for KB-to-DB.

The KQML language is one of the main results which have come out of the external interfaces group of the KSE.

KQML

We could address many of the difficulties of communication between intelligent agents described in the Introduction by giving them a common language. In linguistic terms, this means that they would share a common syntax, semantics and pragmatics.

Getting information processes, especially AI processes, to share a common syntax is a major problem. There is no universally accepted language in which to represent information and queries. Languages such as KIF [Genesereth et. al. '92], extended SQL, and LOOM [McGreggor] have their supporters, but there is also a strong position that it is too early to standardize on any representation language. As a result, it is currently necessary to say that two agents can communicate with

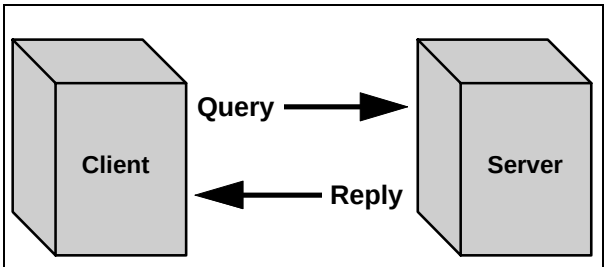

Figure 1 - In this example of a synchronous communication query, a blocking query waits for an expected reply.

each other if they have a common representation language or use languages that are inter-translatable.

Assuming a common or translatable language, it is still necessary for communicating agents to share a framework of knowledge (i.e. a shared structured vocabulary) in order to interpret the messages they exchange. This is not really a shared semantics, but a *shared ontology*. There is not likely to be one shared ontology, but many. Shared ontologies are under development in many important application domains such as planning and scheduling, biology and medicine.

Pragmatics among computer processes includes

1) knowing who to talk with and how to find them

2) knowing how to initiate and maintain an exchange.

KQML is concerned primarily with pragmatics (and secondarily with semantics). It is a language and a set of protocols which support computer programs in identifying, connecting with and exchanging information with other programs.

KQML Protocols

There are a variety of interprocess information exchange protocols. There is the simple case of one process (a client) sending a query to another process (a server) and waiting for a reply as is shown in Figure 1. This occurs commonly when a backward-chaining reasoner retrieves information from a remote source. As it needs data, it places queries and waits for the replies before attempting any further inferences. A far as protocol is concerned, this case includes those where the server's reply message actually contains a collection of replies.

Another common case is when the server's reply is not the complete answer but a handle which allows the client to ask for the components of the reply, one at a time as shown in Figure 2. A common example of this type of exchange is a simple client querying a relational database or a reasoner which can produce a sequence of instantiations in response to a query. Although this exchange requires that the server maintain some internal state, the individual transactions are each the same as in the single reply case. I.e., each transaction is a "send-a-query / wait / receive-a-reply" exchange. We refer to these transactions as being synchronous because messages arrive

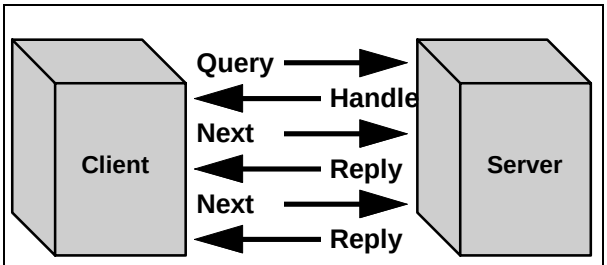

Figure 2 - The remote server can maintain state by remembering the partial answer. Replies are sent individually, each at the request of the client.

at the client only when they are expected.

It is a different situation in real-time systems, among others, where the client subscribes to a server's output and then an indefinite number of replies arrive at irregular intervals in the future, as shown in Figure 3. In this case,

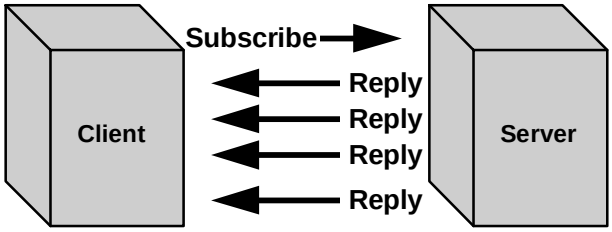

Figure 3 - Using an asynchronous communication protocol, a non-blocking subscribe request can result in an irregularly spaced, indeterminate number of incoming messages.

the client does not know when each reply message will be arriving and may be busy performing some other task when they do. We refer to these transactions as being *asynchronous*.

There are other variations of these protocols. For example, messages might not be addressed to specific hosts, but broadcast to a number of them. The replies, arriving synchronously or asynchronously have to be collated and,

optionally, associated with the query that they are replying to.

The KQML Language

KQML supports these protocols by making them an explicit part of the communication language. When using KQML, a software agent transmits messages composed in its own representation language, wrapped in a KQML message.

KQML is conceptually a layered language. The KQML language can be viewed as being divided into three layers: the content layer, the message layer and the communication layer. The content layer is the actual content of the message, in the programs own representation language. KQML can carry any representation language, including languages expressed as ASCII strings and those expressed using a binary notation. All of the KQML implementations ignore the content portion of the message except to the extent that they need to determine its boundaries.

The *communication level* encodes a set of features to the message which describe the lower level communication parameters, such as the identity of the sender and recipient, and a unique identifier associated with the communication.

The *message* layer forms the core of the language. It determines the kinds of interactions one can have with a KQML-speaking agent. The primary function of the *message* layer is to identify the protocol to be used to deliver the message and to supply a *speech act* or *performative* which the sender attaches to the content. The

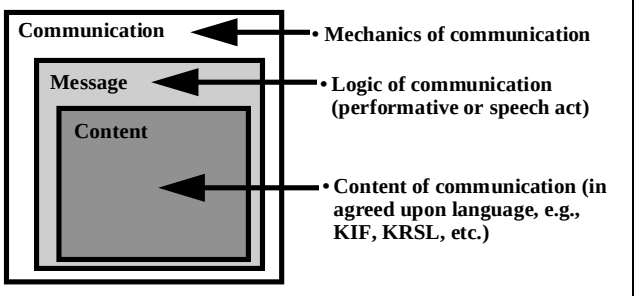

Figure 4 - The KQML language can be viewed as being divided into three layers: the content layer, the message layer and the communication layer.

performative signifies that the content is an *assertion*, a *query*, a *command*, or any of a set of known performatives. Because the content is opaque to KQML, this layer also includes optional features which describe the content: its language, the ontology it assumes, and some type of more general description, such as a descriptor naming a topic within the ontology. These features make it possible for KQML implementations to analyze, route and properly deliver messages even though their content is inaccessible.

Conceptually, a KQML message consists of a performative, its associated arguments which include the real content of the message, and a set of optional arguments which describe the content in a manner which is independent of the syntax of the content language. For example, a message representing a query about the location of a particular airport might be encoded as:

```
(ask-one    :content (geoloc lax (?long ?lat))
            :ontology geo-model3)
```

In this message, the KQML **performative** is ask-one, the **content** is *(geoloc lax (?long ?lat))* and the assumed **ontology** is identified by the token *:geo-model3.* The same general query could be conveyed using standard Prolog as the content language in a form that requests the set of all answers as:

```
(ask-all    :content "geoloc(lax,[Long,Lat])"
            :language standard_prolog
            :ontology geo-model3)
```

The syntax of KQML is based on a balanced parenthesis list. The initial element of the list is the performative and the remaining elements are the performative's arguments as keyword/value pairs. Because the language is relatively simple, the actual syntax is relatively unimportant and can be changed if necessary in the future. (The current syntax was selected because most of the original implementation efforts were done in Common Lisp.)

The set of KQML performatives is extensible. There is a set of reserved performatives which have a well defined meaning. This is not a required or minimal set; a KQML agent may choose to handle only a few (perhaps one or two) performatives. However, an implementation that chooses to implement one of the reserved performatives must implement it in the standard way. A community of agents may choose to use additional performatives if they agree on their interpretation and the protocol associated with each.

Some of the reserved performatives are shown in Figure 5. In addition to standard communication performatives such as ask, tell, deny, delete, and more protocol oriented performatives such as *subscribe*, KQML contains performatives related to the non-protocol aspects of pragmatics, such as *advertise* - which allows an agent to

announce what kinds of asynchronous messages it is willing to handle; and *recruit* - which can be used to find suitable agents for particular types of messages.

For example, agent B might send the following performative to agent A:

Basic query performatives:
evaluate, ask-if, ask-in, ask-one, ask-all
Multi-response query performatives:
stream-in, stream-all
Response performatives:
reply, sorry
Generic informational performatives:
tell, achieve, cancel, untell, unachieve
Generator performatives:
standby, ready, next, rest, discard, generator
Capability-definition performatives:
advertise, subscribe, monitor, import, export
Networking performatives:
register, unregister, forward, broadcast, route

Figure 5 - There are about two dozen reserved performative names which fall into seven basic categories.

```
(advertise
   :language KQML
   :ontology K10
   :content (subscribe :language KQML
                       :ontology K10
                       :content (stream-about
                                  :language KIF
                                  :ontology motors
                                  :content motor1)))
```

to which agent B might respond with:

```
(subscribe      :reply-with s1
                :language KQML
                :ontology K10
                :content (stream-about
                           :language KIF
                           :ontology motors
                           :content motor1))
```

Agent A would then send B a stream of tell and untell performatives over time with information about *motor1*, as in:

```
(tell     :language KIF
          :ontology motors
          :in-reply-to s1
          :content (= (val (torque motor1) (sim-time 5))
                      (scalar 12 kgf))
(tell     :language KIF
          :ontology structures
          :in-reply-to s1
          :content (fastens frame12 motor1))
(untell   :language KIF
          :ontology motors
          :in-reply-to s1
          :content (= (val (torque motor1) (sim-time 5))
                      (scalar 12 kgf))
```

KQML Semantics. Currently there are no formal semantics defined for the basic KQML performatives or for the protocols associated with them. A semantic model is under development that assumes that a KQML-speaking agent has a virtual knowledge base with two separate components: an information store (i.e., "beliefs") and a goal store (i.e., "intentions"). The primitive performatives are defined in terms of their effect on these stores. A TELL(S), for example, is an assertion by the sending agent to the receiving agent that the sentence S is in its virtual belief store. An ACHIEVE(S) is a request of the sender to the receiver to add S to its intention store.

The protocols that govern the allowable responses when an agent receives a KQML message must also be defined. These are currently defined informally in English descriptions, but work is underway to provide formal definitions in terms of a grammar using the definite clause grammar (DCG) formalism.

KQML Internal Architectures

KQML was not defined by a single research group for a particular project. It was created by a committee of representatives from different projects, all of which were concerned with managing distributed implementations of systems. One project was a distributed collaboration of expert systems in the planning and scheduling domain. Another was concerned with problem decomposition and distribution in the CAD/CAM domain. A common concern was the management of a collection of cooperating processes and the simplification of the programming requirements for implementing a system of this type. However, the groups did not share a common communication architecture. As a result, KQML does not dictate a particular system architecture, and several different systems have evolved.

Our group has two implementations of KQML. One is written in Common Lisp, the other in C. Both are fully interoperable and are frequently used together.

The design of these two implementations was motivated by the need to integrate a collection of preexisting expert systems into a collaborating group of processes. Most of the systems involved were never designed to operate in a communication oriented environment. The communication architecture is built around two specialized programs, a router and a facilitator, and a library of interface routines, called a KRIL.

KQML Routers. *Routers* are content independent message routers. Each KQML-speaking software agent is associated with its own separate router process. All routers are identical; each is just an executing copy of the same program. A router handles all KQML messages going to and from its associated agent. Because each program has an associated router process, it is not necessary to make extensive changes to the program's internal organization to allow it to asynchronously receive messages from a variety of independent sources. The router provides this service for the agent and provides the agent with a single point of contact for communicating with the rest of the network. It provides both client and service functions for the application and can manage multiple simultaneous connections with other agents.

The router never looks at the content fields of the messages it handles. It relies solely on the KQML performatives and its arguments. If an outgoing KQML message specifies a particular Internet address, the router directs the message to it. If the message specifies a particular service by name, the router will attempt to find an Internet address for that service and deliver the message to it. If the message only provides a description of the content (e.g. query, :ontology "geo-domain-3", :language "Prolog", etc.) the router *may* attempt to find a server which can deal with the message and it will deliver it there, or it may choose to forward it to a smarter communication agent which may be willing to route it. Routers can be implemented with varying degrees of sophistication -- they can not guarantee to deliver all messages.

In the C implementation, a router actually is a separate UNIX process. It is a child process which is forked by the application. The communication channel between the router and the application carries KQML messages but may carry more than is specified by the formal protocol. That is, since it is a private channel between the router and application it does not have to observe KQML protocol. The router only has to observe the formal KQML rules when speaking to the outside world. The communication channel is currently implemented by a UNIX pipe, but we are planning on experimenting with a higher bandwidth channel which can be implemented with shared memory.

The Lisp implementation uses Lucid's multitasking primitives to implement the router as a separate Lisp task within the application's Lisp image. It would be too inefficient to fork a separate Lisp image for the router. However, we are planning on experimenting with using the C router with Common Lisp applications.

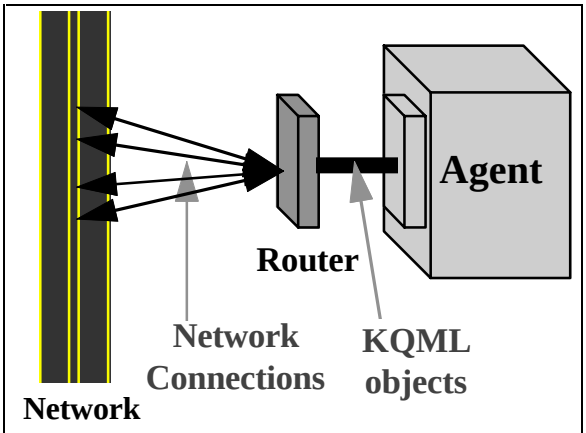

Figure 6 - A router gives an application a single interface to the network, providing both client and server capabilities, managing multiple simultaneous connections, and handling some KQML interactions autonomously.

KQML Facilitators. To deliver messages that are incompletely addressed, routers rely on *facilitators*. A facilitator is a network application which provides useful network services. The simplest service it provides is to maintain a registry of service names; routers rely on facilitators to help them find hosts to route information to. In this role, facilitators serve only as consultants to the communication process.

However, facilitators can provide many other communication services. On request, a facilitator may forward messages to named services. Or, it may provide matchmaking services between information providers and consumers. They include

content based routing of information between agents,

brokering of information between an advertising supplier and an advertising consumer,

recruiting suppliers to deal directly with advertising consumers,

smart multicasting of information to interested agents

These activities can be performed in a relatively simple manner (as shown in Figure 8) or they may be performed by an intelligent agent capable of synthesizing information from multiple sources.

Facilitators are actual network software agents; they have their own KQML routers to handle their traffic and they deal exclusively in KQML messages. There is typically one facilitator for each local group of agents. This can translate into one facilitator per local site or one per project; there may be multiple local facilitators to provide redundancy. The facilitator database may be implemented in any number of ways depending on the number of hosts served and the quality of service required. An early

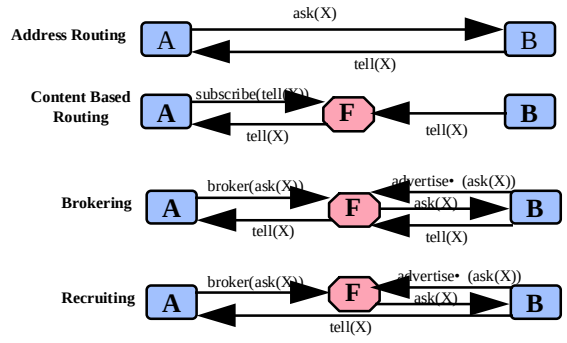

Figure 7 - Facilitators are agents that deal in knowledge about the information services and requirements of other agents. They can offer services such as forwarding, brokering, recruiting and content-based routing.

implementation of a facilitator replicated the database on every machine in the local net, to reduce communication overhead for routing. This was replaced with a more centralized implementation which is supplemented by caching of information in the routers. For larger networks, and for facilitators serving multiple networks, a distributed implementation (analogous to the Internet domain name service) may be more appropriate.

When each application starts up, its router announces itself to the local facilitator so that it is registered in the local database. When the application exits, the router sends another KQML message to the facilitator, removing the application from the facilitator's database. In this way applications can find each other without there having to be a manually maintained list of local services.

KQML KRILs. Since the router is a separate process from the application, it is necessary to have a programming interface between the application and the router. This interface is called a KRIL (KQML Router Interface Library). While the router is a separate process, with no understanding of the content field of the KQML message, the KRIL is embedded in the application and has access to the application's tools for analyzing the content. While there is only one piece of router code, which is instantiated for each process, there can be various KRILs, one for each application type or one for each application language. The general goal of the KRIL is to make access to the router as simple as possible for the programmer.

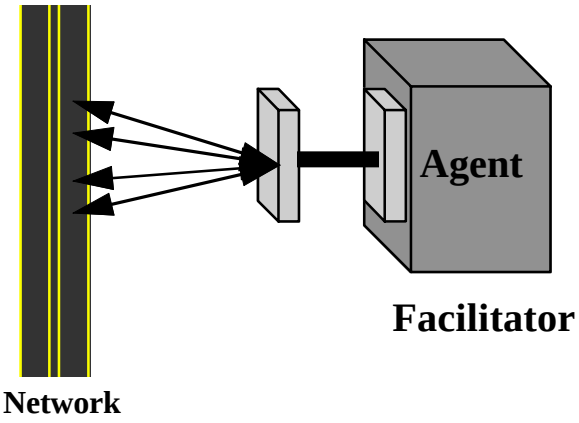

Figure 8 - A communication Facilitator is an agent that performs various useful services, e.g. maintaining a registry of service names, forwarding messages to named services, routing messages based on content, providing "matchmaking" between information providers and clients, and providing mediation and translation services.

To this end, a KRIL can be as tightly embedded in the application, or even the application's programming language, as is desirable. For example, an early implementation of KQML featured a KRIL for the Prolog language which had only a simple declarative interface for the programmer. During the operation of the Prolog interpreter, whenever the Prolog database was searched for predicates, the KRIL would intercept the search; determine if the desired predicates were actually being supplied by a remote agent; formulate and pose an appropriate KQML query; and return the replies to the Prolog interpreter as though they were recovered from the internal database. The Prolog program itself contained no mention of the distributed processing going on except for the declaration of which predicates were to be treated as remote predicates. Figure 9 shows an example of this together with a facilitation agent which provides a central content-based routing service.

It is not necessary to completely embed the KRIL in the application's programming language. A simple KRIL for a language generally provides two programmatic entries. For initiating a transaction there is a **send-kqml-message**

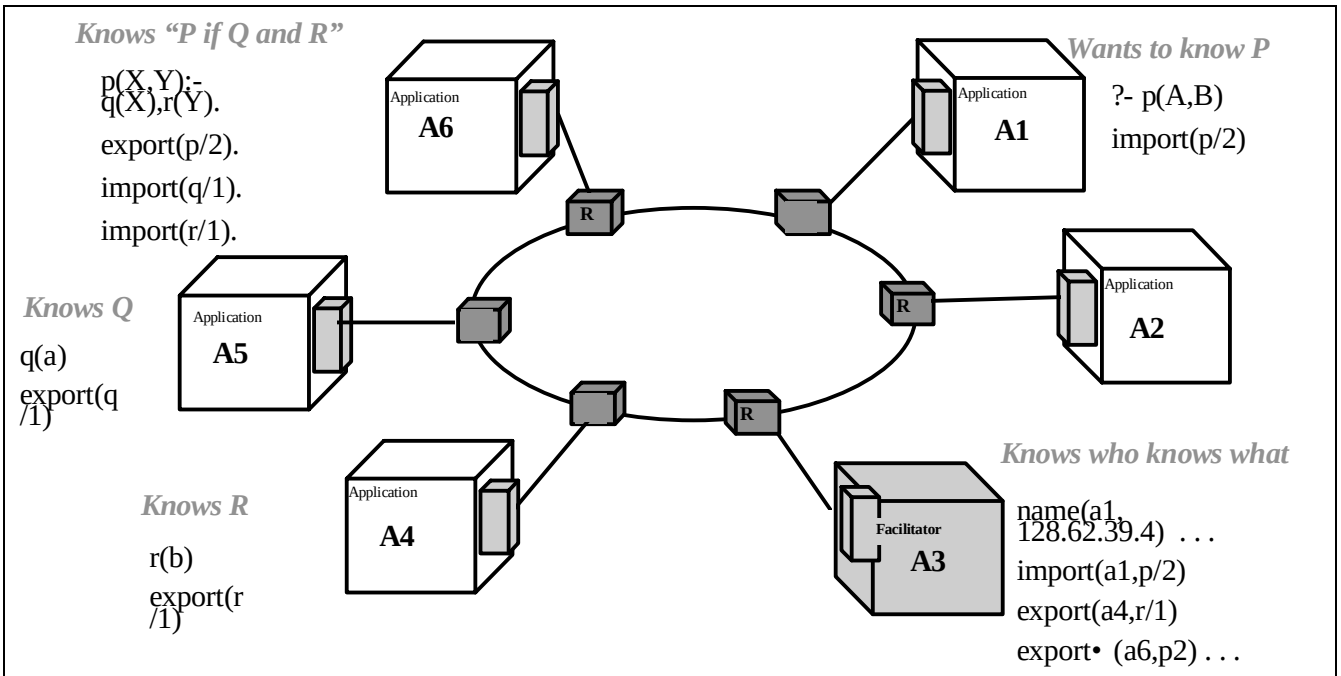

Figure 9 - This example shows the use of a facilitator to do content-based routing allowing a set of Prolog-based agents to work together to prove goals.

function. This accepts a message content and as much information about the message and its destination as can be provided and returns either the remote agent's reply (if the message transmission is synchronous and the process blocks until a reply is received) or a simple code signifying the message was sent. For handling incoming asynchronous messages, there is usually a **declare-message-handler** function. This allows the application programmer to declare which functions should be invoked when messages arrive. Depending on the KRIL's capabilities, the incoming messages can be sorted according to *performative*, or *topic*, or other features, and routed to different message handling functions.

In addition to these programming interfaces, KRILs accept different types of declarations which allow them to register their application with local facilitators and contact remote agents to advise them that they are interested in receiving data from them. Our group has implemented a variety of experimental KRILs, for Common Lisp, C, Prolog, Mosaic, SQL, and other tools.

KQML Performance. We have developed a simple performance model, shown in Figure 11, for KQML communication which has allowed us to analyze the efficiency of communication and to identify and eliminate bottlenecks by tuning the software and adding additional capabilities. For example, various compression enhancements have been added which cut the communication costs by reducing the message sizes and also by eliminating a substantial fraction of symbol lookup and string duplication.

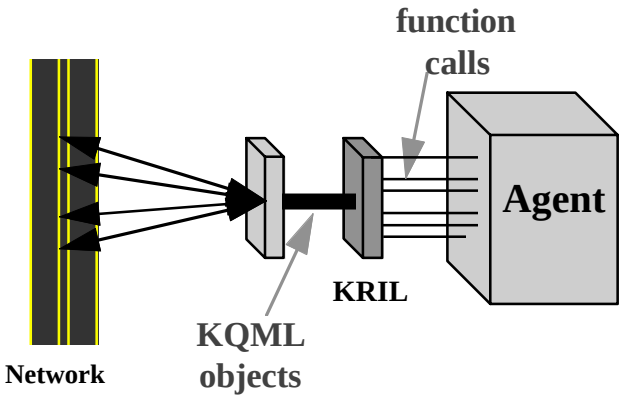

Figure 10 - The KRIL is part of the application and has access to its internals. It provides internal access points to which the router delivers incoming messages, analyzes outgoing messages for appropriate domain tagging and routing, and provides application specific interface and procedures for communication access.

Experiences with KQML

We have used KQML as the communication language in several technology integration experiments in the

ARPA/Rome Lab Planning Initiative. These experiments linked a planning agent (in SIPE), with a scheduler (in Common Lisp), a knowledge base (in LOOM), and a case based reasoning tool (in Common Lisp). All of the components integrated were preexisting systems which were not designed to work in a distributed environment.

We have also successfully used KQML in demonstrations for the ARPA-supported Integrated Weapons Systems Database, integrating distributed clients (in C) with mediators which were retrieving data from distributed databases. Additional work was done under this project using KQML to link a World Wide Web browser with mediators designed to locate documents for them.

The Computer Systems Division of the Aerospace Corp. has used KQML to integrate commercial off-the-shelf software into systems by wrapping them in KQML-speaking shells.

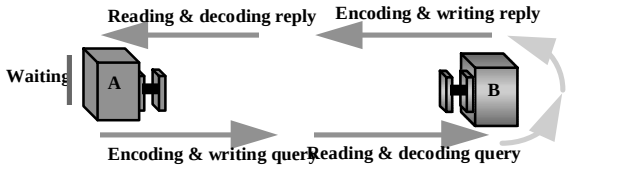

Figure 11 - A simple performance model for KQML communication has allowed us to analyze the efficiency of KQML-based communication and to eliminate bottlenecks.

The Lockheed AI Center and the Palo Alto Collaboration Testbed have also made extensive use of KQML to decompose and distribute problems in the CAD/CAM domain.

Conclusion

This paper has described KQML -- a language and associated protocol by which intelligent software agents can communicate to share information and knowledge. We believe that KQML, or something very much like it, will be important in building the distributed agent-oriented information systems of the future. One must ask how this work is to be differentiated from the work in two related areas -- *distributed systems* (DS) and *distributed AI* (DAI).

KQML and DS. KQML offers an abstraction of an information agent (provider or consumer) at a higher level that is typical in other areas of Computer Science. In particular, KQML assumes a model of an agent as a knowledge-based system (KBS). Although this will not seem to be surprising or profound in our AI community, it is a significant advance (we hope!) for the general CS community. The KBS model easily subsumes a broad range of commonly used information agent models, including database management systems, hypertext systems, server-oriented software (e.g. finger demons, mail servers, HTML servers, etc), simulations, etc. Such systems can usually be modeled as having two virtual knowledge bases -- one representing the agent's information store (i.e., beliefs) and the other representing its intentions (i.e., goals).

We hope that future standards for interchange and interoperability languages and protocols will be based on this very powerful and rich model. This will avoid the built-in limitations of more constrained models (e.g., that of a simple remote procedure call or relational database query) and also make it easier to integrate truly intelligent agents with simpler and more mundane information clients and servers.

In addition to having something to offer, KQML also has something it seeks from distributed systems work -- the right abstractions and software components to provide basic communication services. Current KQML-based systems have been built on the most common transport layers in use today -- TCP/IP and EMAIL. The real contributions that KQML makes are independent of the transport layer. We anticipate that KQML interface implementations will be based on whatever is seen as the best transport mechanism.

K**QML and DAI.** The contribution that KQML makes to Distributed AI research is to offer a standard language and protocol that intelligent agents can use to communicate among themselves as well as with other information servers and clients. We believe that permitting agents to use whatever content language they prefer will make KQML appropriate for most DAI research. In designing KQML, our goal is to build in the primitives necessary to support all of the interesting agent architectures currently in use. If we have been successful, then KQML should prove to be a good tool for DAI research, and, if used widely, should enable greater research collaboration among DAI researchers.

KQML and the Future. The ideas which underlie the evolving design of KQML are currently being explored through experimental prototype systems which are being used to support several testbeds in such areas as concurrent engineering [Cutkowski, McGuire, Tenenbaum, Kuokka], intelligent design [Genesereth] and intelligent planning and scheduling. Figure 12 shows the

architecture of a system in which KQML is being used to support the interchange of knowledge among a planner, a plan simulator, a plan editor and a knowledge server, which is the repository for the shared ontology and access point to common databases through the *Intelligent Database Interface* [McKay, Pastor].

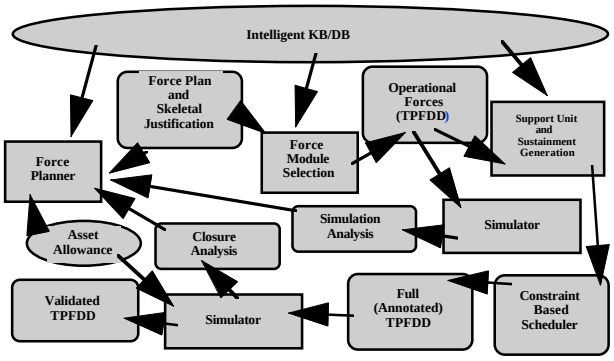

Figure 12 - KQML has been used in the ARPA Rome Planning Initiative to support communication between components of an intelligent planning system.

The design of KQML has continued to evolve as the ideas are explored and feedback is received from the prototypes and the attempts to use them in real testbed situations. Furthermore, new standards for sharing persistent object-oriented structures are being developed and promulgated, such as OMG's CORBA specification and Microsoft's OLE 2.0. Should any of these become widely used, it will be worthwhile to evolve KQML so that its key ideas — the collection of reserved performatives, the support for a variety of information exchange protocols, the need for an information based directory service — can enhance these new information exchange languages.

Bibliography

External Interfaces Working Group ARPA Knowledge Sharing Effort. KQML Overview. Working paper, 1992.

External Interfaces Working Group ARPA Knowledge Sharing Effort. Specification of the KQML agent-communication language. Working paper, December 1992.

S. Bussmann and J. Mueller. A communication architecture for cooperating agents. Computers and Artificial Intelligence, 12:37--53, 1993.

M. Cutkosky, E. Engelmore, R. Fikes, T. Gruber, M. Genesereth, and W. Mark. PACT: An experiment in integrating concurrent engineering systems. 1992.

Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. Trends in cooperative distributed problem solving. IEEE Transactions on Knowledge and Data Engineering, 1(1):63--83, March 1989.

Dan Kuokka et. al. Shade: Technology for knowledge-based collaborative. In AAAI Workshop on AI in Collaborative Design, 1993.

James McGuire et. al. Shade: Technology for knowledge-based collaborative engineering. Journal of Concurrent Engineering: Research and Applications, to appear.

Tim Finin, Rich Fritzson, and Don McKay et. al. An overview of KQML: A knowledge query and manipulation language. Technical report, Department of Computer Science, University of Maryland Baltimore County, 1992.

Tim Finin, Rich Fritzson, and Don McKay. A language and protocol to support intelligent agent interoperability. In Proceedings of the CE& CALS Washington `92 Conference. June 1992.

Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML: an information and knowledge exchange protocol. In International Conference on Building and Sharing of Very Large-Scale Knowledge Bases, December 1993.

M. Genesereth and R. Fikes et. al. Knowledge interchange format, version 3.0 reference manual. Technical report, Computer Science Department, Stanford University, 1992.

Mike Genesereth. Designworld. In Proceedings of the IEEE Conference on Robotics and Automation, pages 2,785--2,788. IEEE CS Press.

Mike Genesereth. An agent-based approach to software interoperability. Technical Report Logic-91-6, Logic Group, CSD, Stanford University, February 1993.

Carl Hewitt and Jeff Inman. DAI betwixt and between: From ``intelligent agents" to open systems science. IEEE Transactions on Systems, Man and Cybernetics, 21(6), December 1991. (Special Issue on Distributed AI).

Michael N. Huhns, David M. Bridgeland, and Natraj V. Arni. A DAI communication aide. Technical Report ACT-RA-317-90, MCC, Austin TX , October 1990.

R. E. Kahn, Digital Library Systems, Proceedings of the Sixth Conference on Artificial Intelligence Applications CAIA-90 (Volume II: Visuals), Santa Barbara CA, pp. 63-64, 1990.

Robert MacGregor and Raymond Bates, The Loom Knowledge Representation Language, Proceedings of the Knowledge-Based Systems Workshop, St. Louis, Missouri, April, 1987.

Don McKay, Tim Finin, and Anthony O'Hare. The intelligent database interface. In Proceedings of the 7th National Conference on Artificial Intelligence, August 1990.

R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. AI Magazine, 12(3):36 -- 56, Fall 1991.

Jeff Y-C Pan and Jay M. Tenenbaum. An intelligent agent framework for enterprise integration. IEEE Transactions on Systems, Man and Cybernetics, 21(6), December 1991. (Special Issue on Distributed AI).

Mike P. Papazoglou and Timos K. Sellis. An organizational framework for cooperating intelligent information systems. International Journal on Intelligent and Cooperative Information Systems, 1(1), (to appear) 1992.

Jon Pastor, Don Mckay and Tim Finin, View-Concepts: Knowledge-Based Access to Databases, First International Conference on Information and Knowledge Management, Baltimore, November 1992.

R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The darpa knowledge sharing effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92), San Mateo, CA, November 1992. Morgan Kaufmann.

J. R. Searle. What is a speech act? In M. Black, editor, From Philosophy in America, pages 221--239. Allen & Unwin, Ort??, 1965.

Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers, C-29(12):1104--1113, December 1980.

Reid G. Smith and Randall Davis. Framework for cooperation in distributed problem solving. IEEE Transactions on System, Man, and Cybernetics, SMC-11(1):61--70, January 1981.

M.Tenenbaum, J. Weber, and T. Gruber. Enterprise integration: Lessons from shade and pact. In C. Petrie, editor, Enterprise Integration Modeling. MIT Press, 1993.

Gio Wiederhold Peter Wegner and Stefano Ceri. Toward megaprogramming. Communications of the ACM, 33(11):89--99, November 1992.

Steven T. C. Wong and John L. Wilson. COSMO: a communication scheme for cooperative knowledge-based systems. IEEE Transactions on Systems, Man and Cybernetics, to appear.