A Language and Protocol to Support Intelligent Agent Interoperability*

Tim Finin
University of Maryland
Baltimore MD
finin@cs.umbc.edu

Rich Fritzson
Paramax Systems Corp.
Paoli PA
fritzson@prc.unisys.com

Don McKay
Paramax Systems Corp.
Paoli PA
mckay@prc.unisys.com

April 1992

Abstract

We describe a language and protocol intended to support interoperability among intelligent agents in a distributed application. Examples of applications envisioned include intelligent multi-agent design systems as well as intelligent planning, scheduling and replanning agents supporting distributed transportation planning and scheduling applications. The language, KQML for Knowledge Query and Manipulation Language, is part of a larger DARPA-sponsored Knowledge Sharing Initiative focused on developing techniques and tools to promote the sharing on knowledge in intelligent systems. We will define the concepts which underlie KQML and attempt to specify its scope and provide a model for how it will be used.

A version of this paper will appear in the *Proceedings of the CE & CALS Washington '92 Conference*, June 1992. Address comments to Tim Finin, Computer Science, University of Maryland Baltimore County, Baltimore MD 21228. finin@cs.umbc.edu. 410-455-3522 -3969 (fax).

1 Introduction

Many computer systems are now structured as collections of independent processes, frequently on distributed hosts linked by a network. Database processes, real-time processes and distributed AI systems are a few examples. Although this architecture is becoming increasingly popular, there are no standards for many of the important protocols needed to support this architecture. Nor are there standard models for programming in an environment where some of the data is supplied by asynchronous 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 the use of these new architectures. Among the issues which need to be dealt with are:

- What language does a process use to formulate a query? In what language is the reply made?
- What protocol does a process use to send a query and produce an answer?

*This work was supported in part by DARPA and Rome Laboratory under USAF contract F30602-91-C-0040 and by the Air Force Office of Scientific Research under contract F49620-92-J-0174.

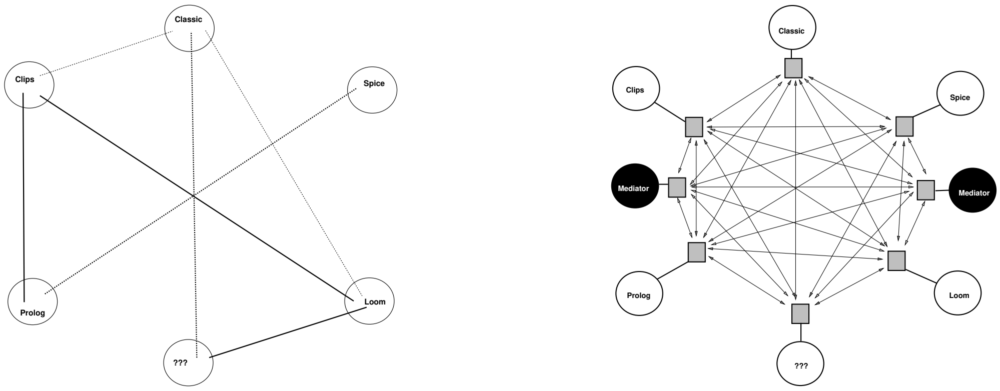

Figure 1: Not only do systems operate and communicate in different ontologies and languages, but they follow different protocols for communicating. This makes communication difficult, since each application has to deal with it independently. Having a multitude of protocols makes a hard problem worse. The introduction of communication facilitator agents simplifies the communication problem.

- What is an effective way to present a communication protocol to users of various programming disciplines such as procedural languages, backward chaining engines, forward chaining engines, classifiers, etc?
- How does a process know to which process to send queries? How can it find out? What protocol does it use in finding out?

This paper discusses the problem and describes the design of a simple knowledge transfer protocol that addresses some of the issues. This is part of a related effort to develop an new style of interface for knowledge-based systems based on the KQML [?] specifications being explored by the DARPA Knowledge Sharing Initiative [?].

The problem of coordinating many agents who must communicate with one another is a difficult one. These problems are compounded when we are trying to coordinate a multitude of "intelligent agents" for several reasons. The first diagram in Figure 1 shows a hypothetical situation in which agents implemented in different AI systems need to share knowledge.

We divide this problem into two general aspects: problems having to do with the mechanics of establishing reliable communication with the right agent and problems involving the mutual understanding and utility of the content of the communication. We believe that this problem decomposition enhances the scalability of our our approach. To solve the first set of problems we introduce additional agents – *communication facilitators* – to help with the process. Our approach to address the second set of problems employs two ideas. The first is to use a simple model for what can be said from one agent to another and to establish some simple standards, conventions and protocols for saying it. The second idea is to provide for a class of *communication mediators* whose role is to effect the semantic and ontologic transformation which are required to map one agents knowledge into another's.

The second diagram in Figure 1 shows our hypothetical situation with the introduction of facilitators and mediators. It might seem that we have complicated the situation and made the problem worse since we have more than doubled the number of agents and show each facilitator talking to every other facilitator. However, we have simplified things considerably since:

- Facilitators and Mediators communicate among themselves using well-defined protocols which are independent of the language or languages which the agents use to exchange knowledge.

- Communication between processes is handled by facilitators and (optionally) mediators, offloading most of the details from the agents.
- Each application system only has to know how to communicate with a facilitator to transmit messages to another system. These messages can be expressed in in any knowledge representation language that the two communication agents find convenient. The use of mediators provides additional translation and mapping services to allow applications to interoperate.

The Knowledge Query and Manipulation Language (KQML) is a language and a protocol to support the high level communication among intelligent agents. It can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to interact cooperatively in problem solving.

SKTP, a Simple Knowledge Transfer Protocol, supports KQML interactions and is defined as a protocol stack with at least three layers: content, message and communication. Additional layers will appear below these three to supply reliable communication streams between the processes. The content layer contains an expression in some language which encodes the knowledge to be conveyed. The message layer adds additional attributes which describe attributes of the content layer such as the language it is expressed in, the ontology it assumes and the kind of speech act it represents (e.g., an assertion or a query). The final communication layer adds still more attributes which describe the lower level communication parameters, such as the identity of the sender and recipient and whether or not the communication is meant to by synchronous or asynchronous.

2 Knowledge Query and Manipulation Language

The Knowledge Query and Manipulation Language (KQML) is a language and an associated protocol to support the high level communication among intelligent agents. It can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to interact cooperatively in problem solving. We argue that KQML should be defined as more than a language with a syntax and semantics, but must also include a protocol which governs the use of the language (e.g., a pragmatic component).

2.1 Design Issues and Assumption

Architectural assumptions. Agents will typically be separate processes which may be running in the same address space or on separate machines. The processes will have a reliable information transport mechanism (e.g. TCP/IP streams) connecting them. We need a protocol that is simple and efficient to use to connect a few pre-defined agents on a single machine or on several machines on the same local area network. We also need the protocol to be an appropriate one to scale up to a scenario in which we have a large number (i.e. hundreds or even thousands) of communicating agents scattered across the global Internet and who are dynamically coming on and off line.

Communication Modes. KQML will support several modes of communication among agents along several independent dimensions. Along one dimension, it supports interactions which differ in the number of agents involved – from a single agent to a single agent (i.e., point-to-point), as well as messages from one agent to a set of agents (i.e., multicasting). Along another dimension, it permits one to specify the recipient agents either explicitly (e.g., by Internet address and port number), by symbolic address (e.g., to "to the *theTRANSCOMMapServer"* or even by a declarative description form of broadcast (e.g., "to any KIF-speaking agents interested in airport locations"). A final dimension involves synchronicity – the protocol must support *blocking* as well as *non-blocking* communication.

Syntactic assumptions. The actual representation of information at the most underlying level, is not of great importance; current implementations utilize Lisp s-expressions transmitted between processes in the form of ascii streams. The forms at the content layer depend on the content-language being used and may be represented as strings, if necessary. The forms at the message and communication layer will be ascii representations of lists with a symbol as the first element and whose remaining elements use the Common Lisp keyword argument convention. However, newly emerging standards, such as those from the Object Management Group, may provide an entirely new basis for representing the content of the messages.

Security. Security is an issue in any distributed environment. We will need to develop conventions and procedures for authentication which will allow an agent to verify that another agent is who it purports to be.

Transactions. Interactions among knowledge-based systems have a different kind of transaction processing which will require something other than the now standard two-phase commit. That is because interacting agents may use information and knowledge gained from one information source for longer periods of time than read/write locks support. In one way, knowledge-based systems are similar to other advanced systems such as software engineering or CAD/CAM design environments (see Computing Surveys, 1991). Further, interactions among knowledge-based systems may better be cast in terms of belief spaces and/or logics of belief than in terms of low level transactions. The development of a good model to support transactions among intelligent agents is a research topic for the KQML group to consider sometime in the future. Developing a workable solution which is incrementally implementable may prove key to the ultimate success of the KQML effort.

Protocol Approach. KQML should be defined as more than a language with a syntax and semantics, but must also include a protocol which governs the use of the language (i.e., a pragmatic component). Using a protocol approach is common in modern communication and distributed processing. The first diagram in Figure 2 shows a simplified version of the standard protocol stack for network communication over an Internet. At the top of the stack is the application-level protocol, in this case SMTP (*Simple Mail Transfer Protocol*) and at the bottom is the low level protocol in which data is actually exchanged. From a mailer's point of view, it is communicating with another mailer using the SMTP protocol. It need not know any of the details of the protocols which support its communication.

We are developing a similar approach to support communication among intelligent agents – defining a protocol stack for transferring knowledge across the Internet. The second diagram in Figure 2 shows a simple protocol stack we are using for the model of KQML. We assume that the KQML protocol stack is an application protocol layer of the standard OSI model and thus assume reliable communication. SKTP, a Simple Knowledge Transfer Protocol, supports KQML interactions and is defined as a protocol stack with at least three layers: content, message and communication. These layers are built upon some reliable network transport mechanism.

2.2 KQML Layers

KQML expressions consist of a content expression encapsulated in a message wrapper which is in turn encapsulated in a communication wrapper, as shown in Figure 3. Thus the language is

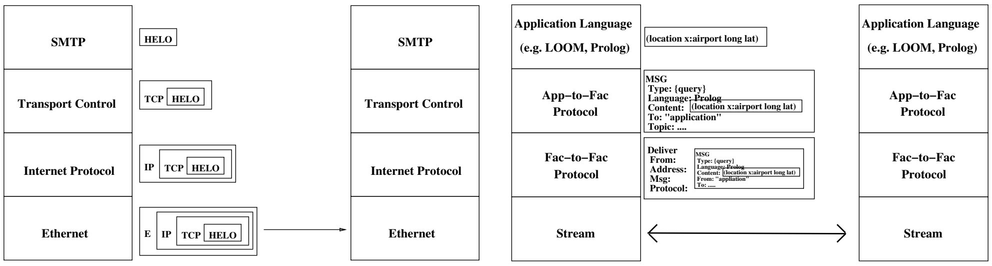

Figure 2: Modern Internet communication is governed by a "protocol stack" with distinct, well-defined layers. Communication between intelligent agents should also be governed by a protocol stack with distinct, well-defined layers.

thought of as being divided into three layers: content, message and communication. The content layer contains an expression in some language which encodes the knowledge to be conveyed. The format of this expression is unimportant to KQML; it can carry any type of content. However, there are emerging standards for Knowledge Representation (e.g. Interlingua, KIF [?], etc) and standards for *persistent objects pointers* (e.g. the OMG Object Request Broker) which may prove to be very valuable in the near future.

Because the content is opaque to KQML, the message layer adds a set of features which describe the content, e.g. the language it is expressed in, the ontology it assumes and the kind of speech act it represents (e.g., an assertion or a query). These features make it possible for the protocol implementation to analyze, route and properly deliver messages even though their content is inaccessible.

The final communication level adds a second layer of features to the message which describe the lower level communication parameters, such as the identity of the sender and recipient and whether the communication is meant to be synchronous or asynchronous. These are used by the network layer which provides reliable transfer of bytes between processes on a network.

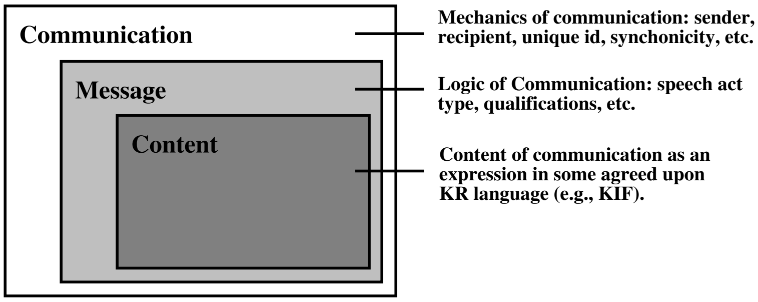

Figure 3: KQML expressions can be thought of as consisting of a content expression encapsulated in a message wrapper which is in turn encapsulated in a communication wrapper.

2.3 KQML Content Layer

KQML makes no commitments about the content layer. One should be able to use it with any number of content languages, such as KIF [?] or LOOM [?]. All the two intelligent agents need to do is to agree on a language to use for communication. This does not preclude the use of or diminish the need for an *interlingua* such as KIF to support knowledge sharing. It does allow two agents who are using the same internal language to use it as the communication language in the protocol.

2.4 KQML Message Layer

The message layer is used to encode a message that one application would like to have transmitted to another application. These messages are of two general types — content messages and declaration messages. A "content" message contains a description of a piece of knowledge being offered or sought. "Declaration" messages are used to announce the presence of an agent, register its name and provide descriptions of the general types of information that the agent will send and would like to receive and the actual content baring messages sent between agents.

The message layer can also be thought of as a "speech act layer". One of the most important attributes to specify about the content is what kind of "speech act" it represents – an assertion, a query, a response, an error message, etc.

2.4.1 Content Messages

A content message is used to describe a query, assertion or other speech act involving some sentence in the content language. It is represented as a list whose first element is the atom MSG and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. Possible keyword arguments are:

:TYPE - <*Speech Act*>. The speech act type of the message (e.g., query, assert, retract, etc.).

:QUALIFIERS - <*keyword list*>. A keyword tagged list of qualifiers appropriate to the message type.

:CONTENT-LANGUAGE -<*language name*>. A term naming the language in which the CONTENT field is expressed.

:CONTENT-ONTOLOGY - <*ontology name*>. A term or list of terms chosen from a standard list naming the ontologies assumed.

:CONTENT-TOPIC - <*topic name*>. A term or list of terms describing the topic of the knowledge within the given ontology.

:CONTENT - <*content language sentence*>. The actual knowledge to by conveyed expressed as a sentence in the content-language.

The following example message is a query expressed in KIF for which exactly one answer is sought.

```
(MSG
  :TYPE query
  :QUALIFIERS (:number-answers 1)
  :CONTENT-LANGUAGE kif
  :CONTENT-ONTOLOGY (blocksWorld)
  :CONTENT-TOPIC (physical-properties)
  :CONTENT (color snow ?C))
```

2.4.2 Declaration Messages

A declaration message is used to provide meta-information about the content messages that the agent will generate and would like to receive. These declarations can be used to register a service (e.g., "I'll answer questions about the physical properties of blocks") and to register a need for a service (e.g., "I want to be keep current on the location of every block").

Syntactically, a declaration is a list whose first element is the atom DCL and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. Possible keyword arguments are:

- **:TYPE** - <*Speech act*>. The speech act type of the embedded (MSG ...) expression (e.g., assert, query).
- **:DIRECTION** - <*OneOf(import, export)*>. Specifies whether the information is to be imported or exported.
- **:COMM** - <*Oneof(block, nonblock)*>. Specifies whether the service is being offered or sought in a blocking or nonblocking communication mode.
- **:MSG** - <*Message*>. A (MSG ...) expression which specifies the content level information that is to be imported or exported.

The following example announces that agent *a001* is willing to export assertions expressed in KIF about the color properties of things in a blocks world ontology.

```
(DCL
  :TYPE assert
  :DIRECTION export
  :MSG (MSG
         :TYPE assert
         :CONTENT-LANGUAGE KIF
         :CONTENT-ONTOLOGY (blocksWorld)
         :CONTENT-TOPIC (physical-properties)
         :CONTENT (color ?X ?Y)))
```

2.5 KQML Communication Layer

At the communication layer, agents exchange *packages*. A package is a wrapper around a message which specifies some communication attributes, such as a specification of the sender and recipients. A package is represented as a list whose fist element is the atom PACKAGE and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. Possible keyword arguments are:

- **:TYPE** - <*Message type*>. This is the type of the embedded message, i.e. either a *content* message or a *declaration* message.
- **:FROM** - <*Agent ID*>. The unique identifier of the sending agent.
- **:TO** - <*Agent ID*>. The unique identifier or identifiers of the recipient agent(s).
- **:ID** - <*Package ID*>. A unique identifier for this message. This should be generated at this layer (e.g. by the facilitator agent if one is being used) and is used to refer to the message later.
- **:COMM** - <*Oneof(block, nonblock)*>. Specifies whether or not the communication is to be carried out in a *blocking* or *nonblocking* mode.
- **:IN-RESPONSE-TO** - <*Package ID*>. A list of one or more package IDs which refer to earlier messages that this package is in response to.
- **:CONTENT** - <*Message*>. An (DCL ...) or (MSG ...) expression.

In the following example, application *ap001* is sending a synchronous query to application *ap002*:

```
(PACKAGE
  :FROM ap001
  :TO ap002
  :ID DVL-f001-111791.10122291
  :COMM block
  :CONTENT
    (MSG
      :TYPE query
      :QUALIFIERS (:NUMBER-ANSWERS 1)
      :CONTENT-LANGUAGE KIF
      :CONTENT (color snow _C)))
```

2.6 KQML Performatives

Message types play an important part in this protocol. They appear at the message level in both content and declaration messages and are akin to a "speech act" type in the theory of natural language communication. Message types determine what one can "do" or "perform" with the sentences in the content language.

The definition of the various KQML performatives described in [**?**] is based on the following model of a knowledge base: A *knowledge base* (KB) is a set of sentences in a language L, which can be the object language of the knowledge base, or a set of sentences of another language for which a computable mapping into L exists. Candidates for languages other than the object language of a KB are, for example, the Interlingua, or, if the object language for a KB are graphs, a linear notation describing these graphs. Since KQML is not assumed to be a superset of the Interlingua, it must to identify sentences of the KB by way of quoted sentences of a language that can be translated into the object language of the KB. This language is called the *content language*.

3 SKTP

SKTP is a design for an implementation of the KQML protocol stack. The design follows the layered organization of the protocol. One section of the code handles the encapsulation and labeling of content expressions (implementing the message layer). Another section determines the destination of the messages and arranges (via some standard transport mechanism) for their delivery and the return of any immediate responses. An important feature of SKTP is its tight integration with the implementation language of an application. This provides a nearly seamless interface between the application and the communication protocols, significantly reducing the difficulty of programming communicating agents and allowing a much tighter collaboration between processes than has been easily achievable before.

A preliminary implementation of SKTP has been written in Common Lisp and currently links applications written in a dialect of Prolog. We are designing interfaces to additional languages and systems.

There are four protocol layers shown in Figure 2. Each has a matching component in the implementation design shown in Figure 4. The overall communication is between *applications* written in an *application language*. Applications exchange expressions which have some meaning. This is the *content layer*. Expressions are selected for transmission to remote sites and wrapped in *messages*. This is the *message layer* and is implemented in Figure 4 by the module labeled **Facilitator Interface Library (FIL)**.

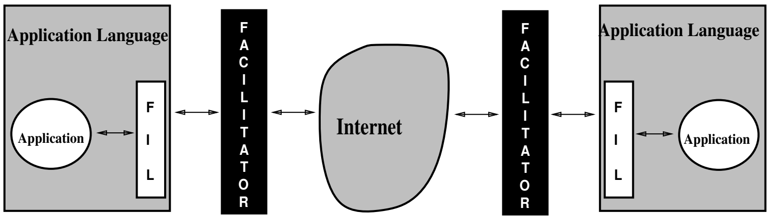

Figure 4: The implemented SKTP architecture has a component for each of the major protocol layers in KQML.

Messages might not have unambiguously specified destinations; they may have multiple destinations; they may require special handling. The layer which handles this "routing" of messages is the *communication layer* and is implemented by a separate agent called a **facilitator**. The underlying stream which carries the structured data between facilitators is, of course, the TCP/IP protocols provided by the Internet.

Each application is associated with a facilitator. Figure 5 shows an imagined collection of application communicating via SKTP over an Internet. While the diagram looks complex, the important gain made by the communication layer (implemented by the *facilitators*) is that all network communication is made using the same protocol, instead of a different protocol for each pairing of systems.

3.1 Facilitator Interface Library

The *Facilitator Interface Library* (FIL) is the code which connects the varying worlds of different AI languages and systems to the communication world of KQML. The FIL performs three functions

- It interprets a set of declarations which describe the internal knowledge base transactions (e.g. definitions, queries, assertions) should be imported from or exported to remote systems.
- It contains code which monitors those internal transactions and arranges for the appropriate expressions to be transmitted as messages to a facilitator which will route them appropriately.
- It contains code which provides access points for a facilitator to deliver messages to the application (e.g. queries to be answered, assertions to be stored, etc.)

Because the FIL is tightly integrated with the application system, it is partly implemented in the underlying implementation language. For example, in our current prototype we have applications written in a dialect of Prolog which is implemented in Common Lisp. The FIL for these applications is written partly in Prolog and partly in Common Lisp.

Declarations. When using SKTP, an application program does not need to be modified to make "calls" on communication primitives. Instead, it is written as though the information that it needs was available locally (or, if it is primarily a supplier of information, as though there was no need to communicate). The program is augmented by a set of declarations which describe the internal transactions (assertions, queries, etc.) that are to be exported to remote sites and what types of transactions it is willing to process from remote sites. Declarations describe the following attributes of expression:

- Whether the expression is to be exported (sent to a remote site) or imported (accepted from a remote site)
- The type of the expression (e.g. assertion, query, definition, etc.)

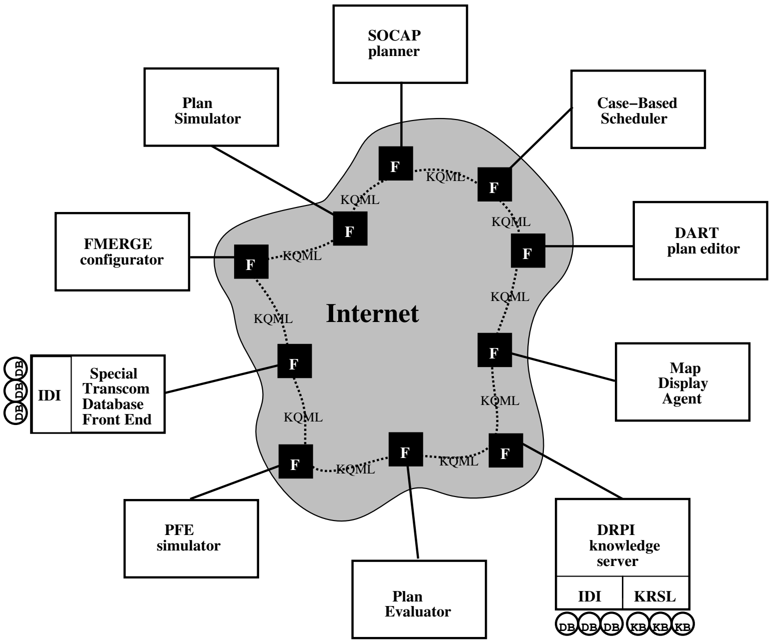

Figure 5: A network of processes communicating using the SKTP architecture is envisioned as a part of the DRPI testbed. Communication among the processes is handled by *communication facilitators.*

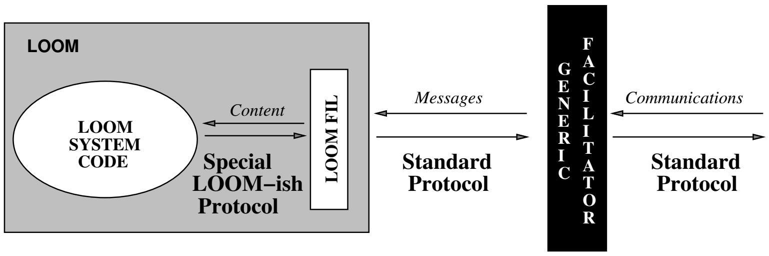

Figure 6: The *Facilitator Interface Library* or FIL is the code which connects an AI language or system such as LOOM to a generic Facilitator agent.

- A characterization of the expressions of this type which are to be selected. For example, in a relational system, the description might contain the name of the relation and the number of arguments, or in an object-oriented system it might be the class of the object.

Declarations which describe exports have to result in code which monitors the internal flow of expressions, selects appropriate ones for encapsulation, and is prepared to insert any replies into the the applications internal flow as though they originated locally.

Programming Models. The basic idea of this approach is to completely hide the communication primitives from the application programmer. This is why the FIL will frequently need to be partly written in the underlying implementation language: it needs access to the internal routines of the language itself to help determine *when* expressions need to be transmitted and *which* expressions should be selected.

While this is a difficult job for the implementor of the FIL, it has a couple of significant advantages. The first is that it makes application programming **much easier**. The application programmer doesn't have to think about communication issues while writing the application, just prepare a set of declarations to go with it. The declarations themselves are written at a higher level of abstraction than communication code and so are easier to write. The second advantage is that it relieves the implementor of the FIL from having to design and implement a creative and clever way to integrate the communication primitives with the non procedural languages used in AI systems.

Tighter Collaboration. This approach also makes it possible for applications to collaborate at a much tighter level of coupling than the simple "pipe" model of communication which is the only model currently used in the current testbeds we are using[1]. For example, in the current SKTP implementation, if an application's internal processes require a particular goal to be satisfied remotely, the system will transmit that goal to a particular remote system and the answers will be seamlessly integrated into the local system's inferencing cycle. The system answering the remote query may also generate additional remote queries (possibly back to the originating system). All of this is transparent to the originating application which operates as though all the necessary information was being provided locally. Because the library intercepts *internal* transactions, two processes can actively collaborate, in parallel, on a single goal, without explicitly programming that collaboration. This greatly elevates the state-of-the-practice for collaboration among separately written processes.

Exporting Messages. Declarations which state the the application is going to be exporting expressions require that the FIL contain code which will monitor the generation of these expressions, and act on appropriate ones.

When an application declares that it needs to export some of its queries to remote agents, the FIL creates code which monitors the internal generation of queries, queries which are normally generated for use by an internal inference engine, looking for ones which match the declared description. Queries which match the declarations are encapsulated as *messages* and passed to a *facilitator*. Depending on the application, the language, and the designer of the FIL, the FIL might wait for answers, or it might not; it might merge the answers from remote sites with local answers, or replace the local answers outright. These and other design decisions are made by the designer of the FIL and may be passed on as declaration options to the application programmer.

[1] The DARPA/Rome Planning Inititive (DRPI) and the Palo Alto Collaborative Testbed (PACT) [?].

Similar considerations apply for declarations which state that the application is going to be exporting assertions. The primary difference is where in the implementation the FIL has to look for the expressions and what to do with any replies that are received.

Importing Messages. If an application is willing to answer queries for remote agents, or it is interested in receiving assertions from remote agents, it declares this in the same way as it would declare a need to export expressions. However, in this case the FIL has to establish a set of properly advertised functions or entry points to which a facilitator can deliver the queries or assertions.

The actual implementation of this connection depends on the design of the facilitator and the type of connection it has with its FILs. In various implementations the facilitator might be part of the same lisp image, or it might be a separate process connected by shared memory or some type of interprocess communication channel. Naturally the kind of "advertisement" needed to let the facilitator know how to deliver messages would depend on the type of connection between the two modules.

3.2 Facilitators

Facilitators bridge the gap between KQML messages and the Internet world of host names and TCP/IP streams. Using the metaphor of the Internet protocol stack, they are the KQML equivalent of Internet routers.

Facilitator accept messages from FILs and rely on the information in the message's fields to determine the appropriate destinations for the message. In some cases an application may identify a particular site as being the target of a message, either by host name (e.g. To: louise.vfl.paramax.com) or more symbolically (e.g. to: whichever machine is currently advertising itself as "geosys server A"). In other cases, the application may not know what an appropriate site is; the facilitator must rely on values of other message fields and a knowledge of what other sites are available in order to decide where to route the message.

Routing. Among the fields of the a KQML messages are:

- language - The language in which the encapsulated expression is written.
- type - "query", "assertion", "definition", etc.
- ontology - The general "framework" or "context" which the sender of the messages assumes and which the receiver must share.
- topic - The specific subject matter of the message. This field can only be interpreted in the context of a given *ontology*.

The declarations written for a program must provide sufficient information to allow the FIL to provide values for these fields. The *facilitator* uses them to search a database of remote agents who have declared that they are suitable targets for these messages. For example, if a facilitator receives (from a FIL) a message which is described as being:

- **Type:** *query*
- **Language:** *relational*
- **Ontology:** *DRPI-93*
- **Topic:** *Airports:Location*

it must look for one or more systems, somewhere on its connected network, which have advertised that they are willing to *import* queries of this type and answer them (By having matching "import" declarations.). It does this by searching a database of declarations looking for entries which match those of the question. When it finds them, it delivers the message to facilitators which are "representing" them and, waits for either an acknowledgment of receipt or actual replies.

While this process does not seem difficult on the surface, there are several problems which will require extensive work, especially as the number of agents available on a network increases and as the complexity of the information being exchanged increases.

Ontology and Topic Matching. The task of matching the declared *ontology* and *topic* of a message against a database of similar declarations is not well defined. While it is not difficult to develop simple examples and simple implementations to handle them it is also not difficult to create complex examples with no obvious implementation strategy.

Consider the case of a small and simply structure *ontology* which is divided into a small and shallow class hierarchy, such as **travel**, divided into fewer than ten possible subclasses such as *air*, *rail*, *car*, etc. Queries may be tagged as having one of these classes as their *topic*; knowledge bases can choose to advertise that they are willing to answer queries about one or more of these classes. As long as all of the participants understand which queries are about which topic and abide by the rules implicit in the simple ontology, the problem of matching messages with remote systems is reduced to simple string matching.

However, if the ontology is not quite as trivial, for example if it is described by a class hierarchy of moderate depth, such as the animal kingdom, then the problem is not so trivial. For example, if a knowledge base advertises that it is willing to IMPORT QUERIES about the class of *mammals* and a facilitator has a client trying to EXPORT a QUERY about *cows*, making the match is more difficult. The routing task must be relatively simple in order to keep the facilitators relatively small and fast. The design of *ontologies* to be used for this purpose must be made with these problems and constraints in mind.

Database of Knowledge Based Services. The second problem to be overcome is how a database of currently available applications is to be maintained. Actually gathering the data is not difficult. An assumption of this design is that all applications provide their FILs with declarations of the queries they can import and the assertions they can export, and that their associated *facilitators* will transmit these announcements over the network. The question is where and how should the database be implemented; there are several alternatives.

The database can take a variety of forms. It may be replicated in every facilitator, it may be centralized on an advertised machine, it may be stored in a distributed form across the network. Implementation strategies are based on the requirements of a particular environment.

For small sets of machines, a replicated implementation may be easiest. That is, each machine maintains its own complete copy of the list of network services. Maintenance of this list has to be performed in realtime; whenever a service begins or ends operation it has to be added to or removed from the list. With a small number of machines the overhead for each machine is not too great.

However, for even modest collections of machines (e.g., more than ten or twenty) the burden of broadcasting service announcements to every known machine, and the burden of processing such announcements from every known machine becomes noticeable, making a centralized approach is more suitable. One machine could serve as a repository for a single database. All processes would send both assertions of services they are making available and queries for needed services to this single machine.

In a very large network, e.g. a large campus network or the Internet itself, any central server will be both a bottleneck and a single point of failure. On this scale, a distributed approach is needed. A good example of this is the Internet distributed name service.

3.3 Implementation

Prototype versions of the components described above have been implemented. We have implemented

- A *facilitator interface library* for an implementation of the language Prolog.
- A *facilitator* which runs as a separate process within the same Common Lisp image as the Prolog language.
- A TCP/IP based communication package which links multiple Common Lisp images on different machines.

An implementation of a Facilitator Interface Library for Prolog has to handle the following events:

- A declaration by the local application of its communication status (what it needs and what it can provide)
- An assertion by the local application which needs to be transmitted to a remote application.
- A query by the local application which needs to be transmitted to a remote application.
- An assertion by a remote application which has been received by the local facilitator
- A query by a remote application which has been received by the local facilitator and needs to be answered

Declarations by the Local Application. This facilitator provides routing for four types of application declarations:

- *Export Queries.* Applications which want to send queries to remote sites.
- *Export Assertions* Applications willing to transmit new assertions to remote sites..
- *Import Queries.* Applications willing to receive (and answer) queries from remote sites.
- *Import Assertions.* Applications which want to receive assertions from remote sites.

Each declaration is accompanied by a description of the type of assertion or query to be exported or imported.

Declaration: Exporting Queries. When a Prolog application declares that it needs to export some of its queries to remote sites, the facilitator interface creates code which will automatically transmit queries of the appropriate type to the facilitator.

The current implementation handles this by generating a Prolog rule of the form:

```
(<query> . <args>) :-
   =(L, (call-lisp (remote-solve (<query> . <args>))))
   member(<args>, L)
```

For example, if an application declares that it wishes to export queries of the form:

```
(color X Y)
```

The facilitator interface will assert the following rule:

```
(color X Y) :-
   = (L, (call-lisp (remote-solve (color X Y))))
   member((X Y), L)
```

The function *remote-solve* transmits the query (with substitution performed on bound variables) to the facilitator which arranges for it to be answered by a remote site. The result is expected to be a list of variable bindings, e.g.

```
((sky blue) (emerald green))
```

This method of dealing with locally generated queries is simple and provides an effective way of dealing with the fact that the remote site returns a list of all solutions while the local site only expects one at a time while it backtracks through them and solves the problem of merging local answers with remote ones in a simple way. It is also very easy to implement.

Declaration: Exporting Assertions. *Exporting assertions* is a declaration primarily used by forward chaining applications and not those implemented in Prolog, but we have included it here for the sake of completeness.

When a Prolog application declares that it is willing and able to export assertions of a particular type, it needs to create code to arrange that assertions which match those described by the declaration are forwarded to the facilitator. The current implementation has modified the low level "assert" and "retract" functions in the Prolog implementation to intercept and transmit matching assertions (and retractions) to the facilitator.

Declaration: Importing Queries and Assertions. When a Prolog application declares that it is willing to accept queries or receive assertions of a particular type, all it needs to do is transmit that declaration to its local facilitator. The facilitator is responsible for insuring that other applications are aware of this service. The transmission is performed by a simple function call provided by the facilitator package.

When a Prolog system is willing to support this type of activity it it needs to provide the facilitator with functions to call whenever remote queries or assertions arrive from a remote site. This *registration* is made by a call to a function provided by the facilitator package.

Handling Locally Generated Queries and Assertions. When the local Prolog generates a query or assertion which needs to be transmitted to a remote site, the preliminary work of the declaration handling (see above) has already arranged for the expression (the query or assertion) to reach the facilitator interface code. The next step is to package the expression into a *message*.

The facilitator provides a function for making *messages*. The interface package simply provides values for the following *message* fields:

- **content.** The expression itself.
- **language.** In this case, the name of the particular Prolog dialect, *Frolic*.
- **type.** *query*, *assertion*, *retraction*, ...
- **ontology.** This is a name which signifies the shared assumptions that the programs have about the knowledge they are using. It is a keyword shared among programs to keep other programs with the same **topic** from answering.

- **topic.** For one simple ontology, this is simply the particular predicate used in the expression, e.g. COLOR/2. For another we used a list which represented the predicate and its arguments which were represented by either constants or untyped variables, e.g. (available JohnSmith ?Time ?Date ?Duration).

Prolog is not a good language in which to experiment with *ontology* definitions because most "real" ontologies tend to be object oriented while Prolog is relation oriented. For example, a service which can answer queries of the form:

```
(location ?x:airport ?y:coordinates)
```

which can be translated as "What is the location of a particular airport?" is not likely to advertise itself as a "location" server, providing information about the location of various objects. It is more likely to be an "airport" server, able to answer questions about various characteristics about airports, including their location. That is, it is more likely to be able to answer

```
(number-of-runways ?x:airport ?n:number)
```

than

```
(location ?x:museum ?y:coordinates)
```

A given application is likely to be able to provide some information about a set of objects in some "knowledge space". What kind of knowledge space is described by the *ontology* field. But the task of describing which objects in that space, and what relations about those objects, falls to the *topic* field.

A second function, *send-msg-to-facilitator* passes it on to the facilitator for routing.

Handling Remotely Generated Queries and Assertions. To handle remotely generated queries and assertions all the interface package has to do is provide a function for the facilitator to call when it needs to pass a query to be processed or an assertion to be added/retracted from the database.

Common Lisp Facilitator. The facilitator's role is to route messages to appropriate recipients. Messages are not usually addressed to a specific individual site but to either a symbolically named service (e.g. Shipping-database or Planning-System-7) or a service which has advertised that it is willing to accept messages of this type. The facilitator is responsible for tracking which remote applications are interested in receiving assertions or are willing to answer queries on various topics.

To accomplish this, each facilitator maintains its own database of remote applications. Each entry in the database provides the Internet address of the host that the application is running on and a TCP/IP port address for the facilitator on that host. The entries are indexed by the types of messages the applications are willing to accept. Messages are characterized, as described earlier, by the same fields used to construct them: *type*, *language*, *ontology*, *topic* and also *communication style*.

The database is maintained jointly by all active facilitators using the following rules:

- When a local application declares that it is willing to import queries or assertions, the facilitator broadcasts that to all sites which may be running a facilitator.
- When a facilitator receives a declaration from another facilitator it acknowledges it by sending a list of imports that its applications are willing to accept.

The first rule lets everyone know about any new services. The current implementation is awkward in that it requires a list of machines where facilitators might be running. We will be replacing this with a separate service which tracks running facilitators and distributes new messages to them. The second rule insures that new facilitators which announce their services are immediately apprised of other facilitators on the net and can build their own database.

Common Lisp TCP/IP. The facilitator is implemented using a locally written TCP/IP interface which allows Common Lisp applications to act as TCP/IP stream clients or servers. It provides client functions to open streams to remote TCP/IP ports using hostnames (or Internet addresses) and service names (or numbers). It also creates a separate process (within a Lucid Common Lisp image) which monitors a specified port and will spin off additional subprocesses when remote system communicate with that port. (That is, it implements a standard UNIX server program.)

4 Conclusions

KQML is a language which supports the high level communication among intelligent agents. It can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to interact cooperatively in problem solving. SKTP, a Simple Knowledge Transfer Protocol, supports KQML interactions and is defined as a protocol stack with at least three layers: content at the application level, message at the application to facilitator level, and communication at the facilitator to facilitator level. Additional layers appear below these three to supply reliable communication streams between the processes. The content layer contains an expression in some language which encodes the knowledge to be conveyed. The message layer adds additional attributes which describe attributes of the content layer such as the language it is expressed in, the ontology it assumes and the kind of speech act it represents (e.g. an assertion or a query). The final communication layer adds still more attributes which describe the lower level communication parameters, such as the identity of the sender and recipient and whether or not the communication is meant to by synchronous or asynchronous.

We have implemented an experimental prototype of SKTP which uses *communication facilitators* as intelligent "routers" to simplify the application interface and realize the protocol. Facilitators provide a declarative framework in which applications specify their knowledge needs and the knowledge services they offer, establish communication channels between appropriate agents, and mediate the resulting dialogue.

KQML is part of a larger DARPA-sponsored Knowledge Sharing effort focused on developing techniques and tools to promote the sharing on knowledge in intelligent systems. The next steps in this research will be apply this integration approach in several distributed testbeds. Examples of applications envisioned include intelligent multi-agent design systems supporting collaborative designs of complex circuits and devices by multiple design teams as well as intelligent planning, scheduling and replanning agents supporting distributed transportation planning and scheduling applications.

5 Acknowledgements

The concepts and ideas in this paper are the result of contributions from a great many people. Major contributions were made by Hans Chalupsky, Mike Genesereth, Stu Shapiro, and Gio Wiederhold.