Revision 3.0 March 1995

SOFTWARE DESIGN DOCUMENT

FOR

KQML

(KNOWLEDGE QUERY and

MANIPULATION LANGUAGE)

CONTRACT NO. F30602-93-C-0177
CDRL SEQUENCE NO. A009

Prepared for:

United States Air Force
Advanced Research Projects Agency (ARPA)

Prepared by:

Unisys Corporation
70 East Swedesford Road
Paoli, PA 19301

Table of Contents

1. Scope 1

1.1 Identification 1

1.2 System Overview 1
1.2.1 What is KQML? 1
1.2.2 Using KQML in End-User Applications 1
1.2.3 Using the TCP/IP API 1

1.3 Document Overview. 2

2. Referenced Documents 3

2.1 Government Documents 3

2.2 Non-Government Documents 3

3. KQML Preliminary Design 4

3.1 Overview 4
3.1.1 Architecture 4
3.1.2 System States and Modes 6

3.2 Design Description 9
3.2.1 TCP/IP interface 9
3.2.2 TCP/IP interface for C 9
3.2.3 KQML Facilitator 10
3.2.4 KQML Router 10

4. Data Elements in the KQML Implementation 12

4.1 TCP/IP Interface Components 12
4.1.1 TCP/IP Interface Components for Common Lisp 12
4.1.2 TCP/IP Interface Components for C 12

4.2 Facilitator 12

4.3 Router Components 12
4.3.1 KQML messages 13
4.3.2 The router's host cache 13

5. Notes 14

1. Scope

1.1 Identification

This document provides a detailed description of the Common Lisp and C implementations of KQML.

1.2 System Overview

1.2.1 What is KQML?

Modern computing systems often involve multiple intergenerating computations/nodes. Distinct, often autonomous nodes can be viewed as agents performing within the overall system in response to messages from other nodes. There are several levels at which agent-based systems must agree, at least in their interfaces, in order to successfully interoperate:

- Transport: how agents send and receive messages;
- Language: what the individual messages mean;
- Policy: how agents structure conversations;
- Architecture: how to connect systems in accordance with constituent protocols.

KQML is primarily concerned with the transport and language levels. It complements work on representation languages concerning domain content, including the ARPA Knowledge Sharing Initiative's Knowledge Interchange Format (KIF). KQML has also been used to transmit object-oriented data. KQML is a language for programs to use in order to communicate attitudes about information, such as querying, stating, believing, requiring, achieving, subscribing, and offering. KQML is indifferent to the format of the information itself, thus KQML expressions will often contain subexpressions in different content languages.

KQML is most useful for communication among autonomous, asynchronous, agent-based programs.

A KQML message is called a performative, a term from speech theory. The message is intended to perform some action by virtue of being sent. A substantial number of KQML performatives can be found in the *Specification of the KQML Agent-Communication Language*. (See section 2, **Referenced Documents**.)

1.2.2 Using KQML in End-User Applications

Programs written in C can become KQML agents by choosing to accept messages from other programs or choosing to send messages to other programs, or choosing to do both. There is no constraint that a program be either a server or a client. Programs are viewed as agents which are free to initiate communication or respond to communication.

The KQML implementation creates additional processes to handle incoming messages asynchronously. The user's program is free to execute code or respond to local events (e.g. input from the user at the console).

1.2.3 Using the TCP/IP API

The KQML specification doesn't specify the architecture of the environment it is used in. It is possible to use KQML in a TCP/IP network of multiprocessing systems (such as UNIX

workstations). But it is also possible to transmit KQML expressions over RS-232 lines or even send them via email. The agents sending them do not have to be multitasking; they can be more primitive computing systems (e.g. computers running MS-DOS). However, each implementation has to make certain assumptions about the environment in which they work.

The C and Lisp implementations are designed to work in Sun ANSI C and Lucid Common Lisp environments respectively and they assume that communication will be via a network which implements UNIX sockets.

The implementation also assumes that an agent called a facilitator will be running on an accessible and well-known host. The facilitator keeps track of which services are available and at which hosts and IP port they can be reached.

The TCP/IP primitives used by both the C and Lisp KQML implementations provide:

- A client facility for opening a connection to a remote TCP/IP service.
- A server facility which listens to multiple TCP/IP ports and transfers incoming data to a function associated with the port. It will also monitor open streams and invoke a specified function when new data becomes available.

1.3 Document Overview.

This document provides a description of the functionality, architecture and implementation of both the Common Lisp and C implementations of KQML

2. Referenced Documents

2.1 Government Documents

Software User's Manual for the Common Lisp Implementation of KQML. Contract No. F30602-91-C-0040, CDRL Seq. No. A0005

Software Design Document for the Initial Common Lisp Implementation of the KQML Knowledge Router and Knowledge Router Interface. Contract No. F30602-91-C-0040, CDRL Seq No. A006

2.2 Non-Government Documents

Mediated Information Systems Technology, http://louise.vfl.paramax.com/

Draft Specification of the KQML Agent-Communication Language, Finin, Weber, et al.

Contact Tim Finin, Computer Science, University of Maryland Baltimore County, Baltimore MD for current status. (http://www.cs.umbc.edu/kqml/kqmlspec.ps)

KQML as an Agent Communication Language, Finin, Fritzson, McKay and McEntire. *The Proceedings of the Third International Conference on Information and Knowledge Management*, ACM Press, November 1994. (http://www.cs.umbc.edu/kqml/papers/kqml-acl.ps)

Mediators in the Architecture of Future Information Systems, Wiederhold, *IEEE Computer*, vol. 25, no. 3, March 1992, pages 38-49.

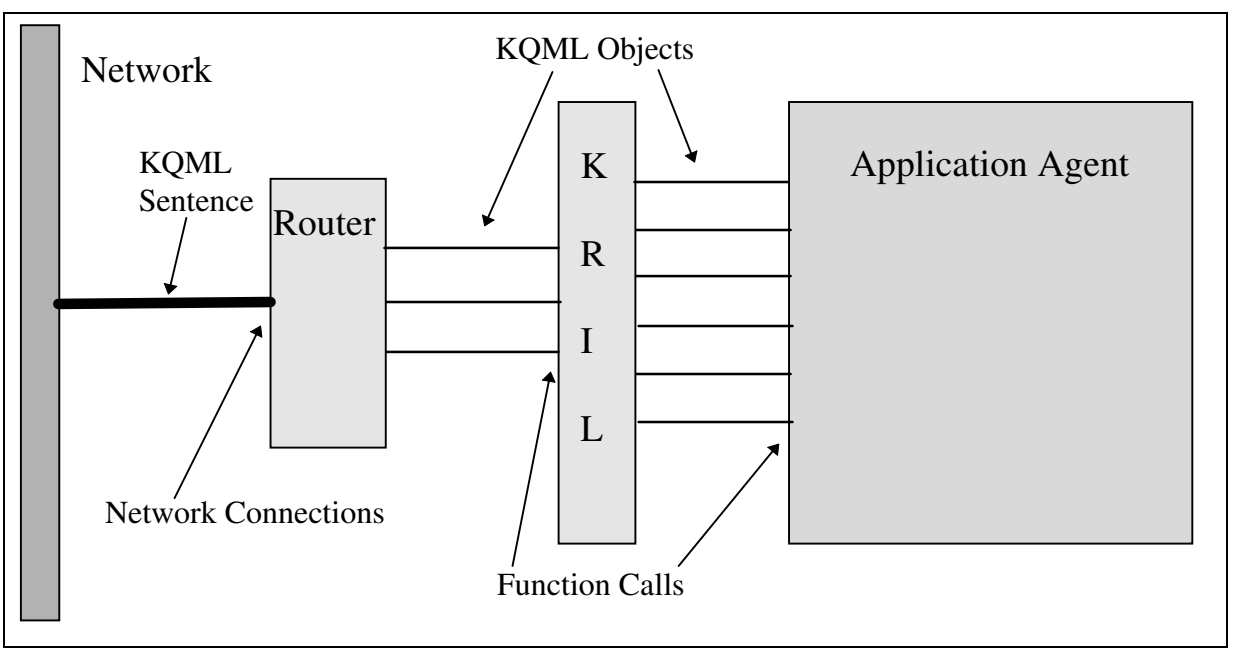
3. KQML Preliminary Design

3.1 Overview

The KQML Router, Facilitator and Knowledge Router Interface Library (KRIL) provide application programs with a relatively simple way to communicate across a network which supports TCP/IP connections. The KRIL has a three layered architecture with access functions at each layer providing access to:

- TCP/IP steams and server functions
- full KQML messages and packets and their component fields
- a simplified interface to KQML for Common Lisp and C applications

3.1.1 Architecture



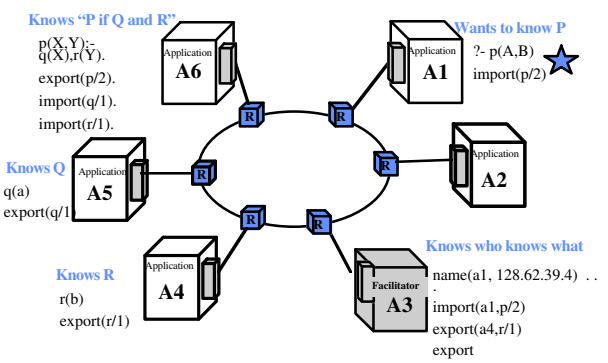
The KQML layer implements the KQML protocol as defined in the current version of the **Draft Specification of the KQML Agent-Communication Language** {Finin, Weber, et al]. An application programmer can create KQML messages or packets and access any of the fields of those data structures.

The two Unisys implementations of KQML, Common Lisp and C, are fully interoperable and are frequently used together. The design of these implementations was motivated by the need to integrate a variety 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 design is built around two specialized programs, a *router* and a *facilitator*, and a library of interface routines, an API, called a *KRIL*.

3.1.1.1 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 each 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 the rest of the network. It provides both client and service functions for the application and manages multiple simultaneous connections with other agents.

The router never looks at the content fields of the messages it handles. It relies 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, 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.



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.

3.1.1.2 KQML Facilitators.

To deliver messages that are incompletely addressed, routers rely on *facilitators*. A facilitator is a network application which provides useful network services. It maintains a registry of service names;it will forward messages on request to named services. It may provide matchmaking services between information providers and consumers. 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.

3.1.1.3 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.

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 10 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 generally provides two programmatic entries. For initiating a transaction there is a **send-kqml-message** function in the Lisp API and **kqml_send** in the C API. Each of these functions 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 a function to register message handlers, the **declare-message-handler** function in the Lisp API and **register_handler** in the C API. This allows the application programmer to declare which functions should be invoked when messages arrive. Depending on the KRILs 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.

3.1.2 System States and Modes

3.1.2.1 Outgoing Expressions - Lisp

make-msg *(perf content &rest arglist)*

The function **make-msg** is used to create a standard *KQML* message data structure. The perf argument is the performative that will be used in the *KQML* message that is constructed. The content argument will appear as the content of the message created. These two fields are enough to construct a valid *KQML* expression; however, the user may add other keyword/value pairs, as described in the *KQML* specification or of their own choosing, in the body of the message by simply including additional arguments to this function. In order to construct a valid *KQML* message the number of arguments after the content must be an even number so that every keyword will have a corresponding value.

send-msg *(msg &optional host)*

The function **send-msg** is used to transmit the message to appropriate hosts. The value of **send-msg** is a list of messages returned by the remote sites which identify themselves with the given symbolic name. **send-msg** examines the message structure, and queries a facilitator (a remote process which maintains a database of available hosts), for hosts

which are advertising themselves with the specified symbolic name. For each matching host, **send-msg** transmits the message to the host, and waits for and reads the response. The reply messages are concatenated together to form a single list which is returned to the calling function. **send-msg** transmits the *KQML* packets to the remote sites by using the TCP level function **connect-to-service**. This function accepts a host name and port identifier as its two required arguments and, if successful, returns a bidirectional Common Lisp stream as a result. The stream is a connection to the process monitoring the specified port.

msg-field *(msg key)*

The function **msg-field** is used to extract the content field of the *KQML* message from the reply messages. The *msg* argument is a complete *KQML* message. This is the message that will be searched for the specified field. *key* is the argument used to search the *KQML* message, and is a valid *KQML* keyword. The *KQML* message will be searched for the keyword provided and will return the value for that keyword.

3.1.2.2 Outgoing Expressions - C

kqml_send_msg *char *perf char *content char *rec_agent int reply kqml_message **reply_msg ...*

This function sends an expression to the appropriate destination, i.e. receiving agent using the content and the KQML performative supplied. *perf* is a KQML performative to be used as the performative of the KQML message that will be sent, *content* is the content message of the KQML message that will be sent, and rec_agent is the name of the receiving agent that is the destination of this KQML message. The fourth argument, *reply,* should be TRUE if the routine should block until a result is returned, and FALSE if the routine should not block. The fifth argument, *reply_msg,* is the address of a pointer to a kqml_message structure and will contain the reply to the message sent when the call to **kqml_send_msg** completes. Finally, this function allows for other arguments to be passed in, using varargs to handle this. The purpose of these final arguments is to provide the API user the ability to specify additional keyword/value pairs to be included in the KQML message sent out. If no additional keyword/value pairs are to be specified, then this argument should be NULL.

kqml_send_msgv *char *perf char *content char *rec_agent int reply kqml_message **reply_msg char *kv[]*

This function identically to the kqml_send_msg function describe above. The single difference is that whereas **kqml_send_msg** allows for additional keyword/value pairs by using varargs, **kqml_send_msgv** allows the user to specify these keyword/value pairs using a character string array. This will be useful to users who will be calling **kqml_send_msgv** without knowing beforehand the number of additional arguments to be passed.

kqml_deliver_msg *char *perf char *content ...*

kqml_deliver_msg is used by an application program which is supporting the monitor and subscribe KQML performatives. Messages sent using kqml_deliver_msg will be sent to all appropriate subscribers. The KQML KRIL will automatically handle all incoming requests by clients which contain the monitor or subscribe performative. Whenever an application calls **kqml_deliver_msg**, the KRIL will determine which clients are listening for the information in the message and route that message to them.

build_msg *char *perf char *content ...*

The **build_msg** function is used to construct new KQML messages. **build_msg** returns a pointer to a freshly allocated kqml_message structure which contains the performative and fields provided. It is not necessary to use this function to construct a message, because kqml_send_msg will construct one for you, but it is provided for programmers who are extending the KRIL or need to use the message structure for some other purpose. The *perf* argument is a KQML performative to be used as the performative of the new KQML message. *content* is the content field of the message constructed. Additional keyword/value pairs, for inclusion in the KQML message being built, may be included after the *content* argument.

get_kqml_field *char *field_name kqml_message *message*

get_kqml_field returns the value of the specified field, if it exists, and NULL if no such field exists in the message. *field_name* is the name of the field whose value is being requested, and *message* is the KQML message that will be searched for that field. KQML fields are identified by their associated keywords or by the word *:performative*.

put_kqml_field *char *field_name kqml_message *message char *new_value*

put_kqml_field will modify the current value of the field/keyword if that field already exists, or it will add the keyword/value pair to the KQML message, if that field/keyword does not yet exist. *field_name* is the name of the field whose value is being changed or added, *message* is the KQML message to be modified with the new value, and *new_value* is the new value of the field/keyword. KQML fields are identified by their associated keywords.

3.1.2.3 Incoming Expressions - Lisp

Incoming expressions are first received by the function **stream-watcher** which immediately spawns a new process which applies the function **handle-stream** to the stream linking the local process with the remote one. **handle-stream** uses a table of incoming file descriptors and port numbers to select a specific function to read the data from the stream and process it. For *KQML* streams, the function called is always **kqml-server.**

kqml-server reads a complete *KQML* message from the stream. It determines which function (if any) is the appropriate one to apply to the message. It applies the function and, if a result is expected, it writes the result back to the stream as a *KQML* message.

If the application is using the *KRIL* interface, it can register functions to handle different types of incoming messages using the **register-handler** function. This function accepts a function name, a language, and a *KQML* performative and assigns the function to handle incoming messages whose performative and language specification match the provided ones.

3.1.2.4 Incoming Expressions - C

Incoming expressions are first received by the listener function in the router module. The C version of *KQML* uses a router which is a separate process from the application process. The router immediately parses the *KQML* expression into a C structure and then writes that structure, using XDR, to the pipe which connects the router to it's application.

The application's *KRIL* receives this structure from the router using the function **fetch_kqml_message**. At this point the *KRIL's* dispatch function, **handle_router**, will do the following;

1. Check for a handler function registered by the application for the performative in the *KQML* message being processed. If there is no handler function associated with this performative, then a *KQML* error message will be sent back to the sender of the message with a content of "Can't handler performative". Otherwise, processing will continue.
2. Execute the application's handler function that is associated with this performative, passing to the handler function the content field of the *KQML* message along with a pointer to the complete *KQML* message.
3. Upon return from the application's handler function, **handle_router** will determine whether or not a reply to the current message should be sent. If so, a reply message will be created and sent to the sender of the message using the function **send_kqml_message**. This response will be received by the *KQML* router module which will then process the message in the same manner as it handles all outgoing *KQML* messages.

3.2 Design Description

These implementations of KQML, the C implementation and Lisp, contain several components; namely, an underlying TCP/IP interface, a KQML router (krouter) and a separate facilitator, written in C.

3.2.1 TCP/IP interface

3.2.1.1 TCP/IP interface for Lucid Common Lisp

This component consists of a set of lisp functions (tcp.lisp), a set of functions written in C (tcp.c) and foreign-function declarations connecting the two (contained in tcp.lisp). It contains Lisp functions which provide access to the UNIX implementation of TCP/IP communications. The services provided are;

`connect-to-service`

This provides the standard client service of opening a stream to a remote TCP/IP listener.

`connect`

This provides the same service as connect-to-service, but it maintains a cache of open connections and returns previously opened connections instead of opening a new one.

`register-service`

This allows a Lisp programmer to create a TCP/IP server which listens to a port for connections by remote clients, creates a stream connecting to the remote client, and passes the stream to a specified function for processing. A program can register multiple services and listen, simultaneously, to multiple TCP/IP ports. The program also monitors previously established streams for new data and provides the newly reactivated streams to the registered functions.

The current TCP/IP interface automatically assigns a TCP/IP port to a service. This is adequate for KQML purposes since the port is then advertised to other agents on the network automatically. However, for applications which must work on specific ports, this function must be modified.

3.2.2 TCP/IP interface for C

This component consists of a set of C functions built on top of the UNIX implementation of TCP/IP communications. The services provided are;

`tcp_to_service`

This provides the standard client service of opening a stream to a remote TCP/IP listener.

`get_connection`

Find or wait for the next socket on which there is data.

`kqml_register`

Register a function to receive and handle *KQML* messages. This function is not itself built on top of the UNIX TCP/IP implementation, but when used in conjunction with get_connection, provides a service analagous to the register-service function of the Lisp implementation, and so is included for completeness.

3.2.3 KQML Facilitator

The *KQML* facilitator is a simple software agent which maintains a database of active *KQML* speaking agents. It accepts *KQML* tell and untell performatives to maintain its database and responds to ask-one and ask-all queries about the contents of the database. Each item in the database is a tuple containing a symbolic agent name , an internet host name , and a TCP/IP port address

3.2.4 KQML Router

3.2.4.1 KQML Router for Lucid Common Lisp

This module provides the functions needed to construct a *KQML* message, to send it to a remote *KQML* agent, to examine the fields inside the message and to define which application functions are to handle incoming *KQML* messages.

The functions provided by this module are:

`message object manipulation functions`

These functions include make-msg msg-field and the generic function print-content which allow an application to build a *KQML* message with specified fields, examine the contents of a message, and control how the content field of a message is printed.

`initialization functions`

The function start-krouter establishes a TCP/IP listener for *KQML* expressions and relies on a defined interface to a local facilitator to register the process with the local facilitator. The function stop-krouter unregisters the applicaton with the local facilitator and shuts down the TCP/IP interface.

`sending KQML expressions`

The function send-msg will transmit a given *KQML* expression to an appropriate host (or hosts) and return any reply messages as the value of the function call.

`receiving KQML expressions`

The function define-interface allows the user to specify a function which will handle incoming *KQML* messages. A different function can be specified for each *KQML* performative.

3.2.4.2 KQML Router for C

This module provides the functions needed to receive a *KQML* message, to send it to a remote *KQML* agent, to parse the incoming *KQML* message, and to send that message to and receive it from the application's KRIL.

The functions provided by this module are:

`initialization functions`

There are four primary functions for initialization procedures, which are **init_router**, **init_app**, **init_listener** and **init_facilitator**. **init_router** initializes the router's cache of remote agents and it's table of remote connections. The function **init_app** establishes a connection to the router's application using the router's standard in and standard out. **init_listener** creates a listener for this router (and thereby, indirectly, for it's associated application) on some port. If the router's application is a *KQML* facilitator, then it will establish a listener on designated facilitator port. In all other cases, the port will be the next available port after the facilitator's port offset. Finally, the function **init_facilitator** will find a facilitator agent and create a connection between that facilitator and the router.

`sending KQML expressions`

In the C router there is a difference between sending messages to the router's associated application and sending messages to other *KQML* agents. The function **send_parsed_msg** is used to transmit a given *KQML* expression, that exists in a C structure in the router, to an appropriate remote *KQML* agent (or agents). The function **send_kqml_message** sends a *KQML* message, that exists in a C structure in the router, to the router's associated application using XDR streams.

`receiving KQML expressions`

In the C router there is a difference between receiving messages from the router's associated application and receiving messages from other *KQML* agents. The function **fetch_kqml_message** reads in a *KQML* message from an XDR stream that connects the router to its associated application. The function **parse_kqml** will read (and parse) *KQML* messages from remote *KQML* agents, and convert those messages into a C structure containing the message.

4. Data Elements in the KQML Implementation

4.1 TCP/IP Interface Components

4.1.1 TCP/IP Interface Components for Common Lisp

The TCP/IP interface is primarily a foreign function interface to the TCP capabilities of UNIX. The primary internal data components are those which make up the cache of open streams which support the connect function. This cache consists of a single array, indexed by UNIX file descriptor integers. Each element of the array contains a structure with the following fields:

- a LISP bidirectional stream
- a lisp function which is applied to the stream whenever data is available on it
- the name of the nost at the other end of the stream
- the TCP/IP port number that the stream is attached to (at the remote end)

When a file descriptor is being used by a UNIX listener, its stream and host fields are empty.

Other global variables in the TCP/IP interface are **inet-config** which simply holds a list of server functions and **next-port** which holds an integer identifying the next available TCP/IP port to be allocated to a new service (by the register-service) function.

4.1.2 TCP/IP Interface Components for C

The TCP/IP interface is one level of API provided in the C implementation of *KQML*, which currently provides access to the UNIX implementation of TCP/IP The primary internal data components for this interface are those that make up the cache of open stream which support the connect function. This cache consists of a arrays indexed by UNIX file descriptor integers. The following data is stored in these arrays for each file descriptor;

- an input stream
- an output stream
- the symbolic name of the remote agent
- hostname of the machine on which the remote agent resides
- port number of the port at which the remote agent is reachable
- information on the state of the connection; i.e., last I/O operation on the connection and whether or not a response is pending for that connection

4.2 Facilitator

The KQML facilitator is written in C and relies uses the components of the C implementation of KQML. Internally, the facilitator is a simple single file database, stored in an array, which maintains tuples of symbolic name, internet host name, and TCP/IP port

4.3 Router Components

The router has two major data types, the *KQML* messages themselves and a small host cache that mirrors the cache of the facilitator. In fact the router's cache is built from responses from the facilitator for queries issued by the router as to the locations of various agents.

4.3.1 KQML messages

In the Lisp implementation *KQML* messages are defined to be syntactically LISP lists, so they are stored internally in that form. A message is a list whose first element is a performative and whose remaining elements are pairs of *KQML* parameters (keywords) and values. A simple lookup function msg-field is used to examine individual fields of a message.

In the C implementation *KQML* messages are stored in a C structure which contains the file descriptor number, the performative for the expression, and an array of keyword/value pairs.

4.3.2 The router's host cache

The KQML router maintains a cache of the elements of the facilitator's database that it has used. The eliminates the need for it to consult the facilitator for each transaction.

In the Lisp implementation, the cache is a Lisp list whose elements are exactly the same as the elements of the facilitator database: symbolic name, host name, port address.

The C implementation of KQML contains the same information as that in the Lisp implementation, described above, but holds that information in C arrays.

5. Notes

KQML is under active development. New implementations will implement larger subsets of the KQML specification but should remain substantially compatible with existing usage.