KQML$_{\text{UofT}}$ Users' Guide

Version 1.1, August 11, 1993

J. Christopher Beck

Department of Industrial Engineering, University of Toronto
4 Taddle Creek Road, Toronto, Ontario M5S 1A4 Canada

tel: +1-416-978-7321, fax: +1-416-978-3453, internet: chris@ie.utoronto.ca

1.0 Introduction

This is a description of the implementation of Knowledge Query and Manipulation Language created at the Enterprise Integration Laboratories at the Department of Industrial Engineering, University of Toronto. Please note that this document is written assuming an understanding of the metaphors, vocabulary, and constructs specified in the KQML document.[1]

The rest of this document is organized as follows: Section 2 presents an overview of the system architecture; Section 3 goes into more depth about the C++ interface to be used by the agent-builders; Section 4 discusses the TCP/IP interface that is written in C; Section 5 is a short note on the conformity of this implementation to the KQML design documents; and finally Section 6 is a list of known bugs and to whom bug reports should be sent.

2.0 System Architecture in Broad Strokes

Two main classes have been created for KQML$_{\text{UofT}}$: class performative and class mbox (declared in `performative.h` and `mbox.h` respectively). There is also a net_address class that is declared in `performative.h`. This is also extensively used, but is quite straightforward and will not be discussed here.

2.1 class performative

The performative class is best conceptualized as a frame with slots. Each parameter in a KQML performative corresponds to a slot in the frame. The usual slot access functions (get and set) are provided for each slot. Please see the file `performative.h` together with the KQML documentation to understand the available parameters and their semantics.

As defined in the KQML document, each performative has a type (`achieve`, `ask-if`, etc.). It was decided that creating subclasses for each type or even for subsets of types of performative would be

1. The latest KQML design document is available by anonymous ftp at eitech.com:kqmlspec.ps.

overkill. As a consequence there is a single class performative and every instance of that class must have a type. There are no restrictions due to this type in the sense that a performative of any type can have a value for any of the parameters. Enforcement of parameter restrictions such as these is left for the agent-builder, if such a restriction is desired.

A performative also has an address. This is not strictly speaking a slot in the frame because it is not one of the parameters of a KQML performative. It specifies the destination of a performative that is written to an mbox and it specifies the origin of an performative that is read from an mbox. Further discussion can be found in the Section 3.

2.2 class mbox

mbox is the "mailbox" class and is best conceptualized as just that: a black box that performatives can be written to and read from.

There is an extremely important limitation to the mbox class: **there can be only one instance of an mbox per agent**. There is no way to ensure a compile-time error if a second instance is created, but a run-time error will occur. Additionally, there is no assignment or copying of an mbox instance. Obviously, given this restriction, if the mbox instance is used as a parameter to some function, it must be passed by reference (i.e. - `send_letter(mbox& mailbox)`).

The mbox also puts restrictions on a performative that is written to it. If the performative does not meet these requirements, a run-time error (with a hopefully helpful error description) is produced. The requirements are as follows: **every performative written to an mbox must have a non-null value in the following slots: sender, receiver, reply-with. Additionally, the performative must have a non-null address.** These restrictions are to aid successful delivery of the performative to the appropriate destination.

The meaning of the address in a performative has different interpretations depending on the history of the performative. If the performative is created by the local agent, the address that the local agent specifies is the destination address of the performative (using an envelope/postal service metaphor, this address is the "mailing address" for the performative). If the performative has be received via an mbox from another agent, the address is the address of the agent that sent the performative (it is the "return address").

2.3 The Name Server

We have found it necessary in our agent-building to have a special agent that all other agents are created knowing. We refer to this special agent as the **name-server** and it's main purpose is to allow agents to find other agents. A typical scenario is for an agent "wake up" and send it's name and abilities to the name-server and then ask the name-server for the names of some agents that can provide a service that is needed. Upon receipt of this information, the agent can then talk directly to the service-providing agent.

This is an implementation detail and can be ignored if another way is found for agents to find each other. We found, however, that if our technique is used there must be some low-level support for the name-server concept. This support consists of two things:

- the name and address of the name-server are global values (within an agent). The definition of these, which can be found in `mbox_static.h`, also depends on definitions in `comm_interface.h`.
- there is a separate constructor for the mbox of the name-server. The normal mbox constructor takes no arguments. The name-server mbox constructor takes a single argument, the address of the name-server. If a name-server is built, use this constructor within the name-server agent, if a name-server is not built, this constructor should never be used.

3.0 The C++ Interface

As discussed above, there are two main C++ classes created in $\text{KQML}_{\text{UofT}}$. This section will consist of a more in-depth discussion of these two classes and the public member functions that allow manipulation.

3.1 class performative

Please refer to `performative.h` for more information.

Since performatives have a type which comes from a specified set of types, an enumeration has been defined for these types in the file `kqml_enum.h`. This enumeration (`kqml_type`) specifies all and only the types that a performative can have. A performative cannot exist without a type (presently, the type can be `kqml_undefined`. We have encountered no instances where having a performative of undefined type is useful. If it is determined to be harmful, this may be removed in future releases). In order to create a performative a type must be specified. However, the functionality to change the type (`set_type`) after the creation is provided. The `set_type` and `get_type` functions both use the `kqml_type` enumeration. Additionally, two functions (`cook_kqml_type` and `uncook_kqml_type`) are declared in `kqml_enum.h` to translate from a character string to the enumeration and vice versa, respectively.

An interface with the ostreams library is also provided in order to display the slot values. This simply allows the syntax: "`cout << perf_instance;`" (where `perf_instance` is a performative frame) to be used to print out a performative.

The `set_*` and `get_*` functions are provided for each slot of a performative frame and for the address of the performative. In all cases but three the `set_*` function accepts a constant character string as a parameter and the corresponding `get_*` function returns a pointer to a character string as a result. The three anomalies are with the address, with the force slot, and with the language and content slots.

3.1.1 address

The set_address and get_address functions manipulate instances of the class net_address. As stated above, there are no surprises (hopefully) with this class and the reader is referred again to performative.h.

3.1.2 force

The force slot uses a enumeration type that is declared in kqml_enum.h. The set_force function takes a parameter of type constant force_level and the get_force function returns a value of type force_level. This was done because the values the force slot can take on are tentative or permanent. These values are specified in the KQML document and are not user defined. (This is contrasted with slots such as ontology and language where the agent-builder will define the languages and ontologies that the agent can "understand"). Two more functions are supplied with the force_level enumeration: cook_force_level (translates from a character string to a force_level value) and uncook_force_level (does the reverse).

3.1.3 language and content

These slots present a complication because of the possibility that the language slot value will be "kqml" and therefore, the content slot will actually be another performative frame embedded within the outer frame. In performatives where the language value is not "kqml," the content slot will simply be a character string. Because of this dependence, there are two set_language_and_content functions. Both take the language value as a character string and take the content value as the second parameter. This second parameter can be either a character string or a pointer to a performative object. If the language value is not "kqml" and the second parameter is a performative pointer, a runtime error will result. Similarly, if the language value is "kqml" and the content value is not a performative pointer.

The get_* functions provided are slightly different as well. The get_language function is like the other slot get_* functions in that it returns a character string. There are two get_content_* functions: get_content_c returns the character string value of the content slot if it is a character string and NULL if it is not a character string; get_content_p returns a pointer to the embedded performative frame if it exists, or NULL otherwise.

3.2 mbox

See mbox.h for more information.

The name-server mbox constructor has been discussed above (Section 2.3).

The mbox functions provide the main interface to the communications infrastructure. As described below in Section 4, this infrastructure consists of three forked processes and a number of pipes of communication between these processes. This is all created when an instance of an mbox is created. That is why only one instance of mbox is allowed per agent. The destruction of this instance handles the killing of the subprocesses (**NOTE:** This can have impact on agent functionality. Since the subprocesses are killed when the mbox is destroyed, it is possible that a performative will have been written to the mbox instance, but not yet delivered to the destination agent. If the mbox is destroyed before it is empty, the performatives that it contained will be lost. See Section 4 for further details).

There are four functions provided with mbox. The first is the trivial `get_my_address` which returns the `net_address` of the agent. The other are: `send_mail`, `receive_mail`, and `send_and_wait`. These act as specified in `mbox.h` and offer few surprises. A more convenient interface to `send_mail` and `receive_mail` is through the overloading of the '>>' and '<<' operators. Given this the following are equivalent:

- `mailbox.send_mail(perf);` and `mailbox << perf;` (where `perf` is an instance of a performative).
- `p = mailbox.receive_mail();` and `mailbox >> &p;` (where `p` is a pointer to a performative).

Cascading of the operators is possible. To send three performatives, the following is valid:

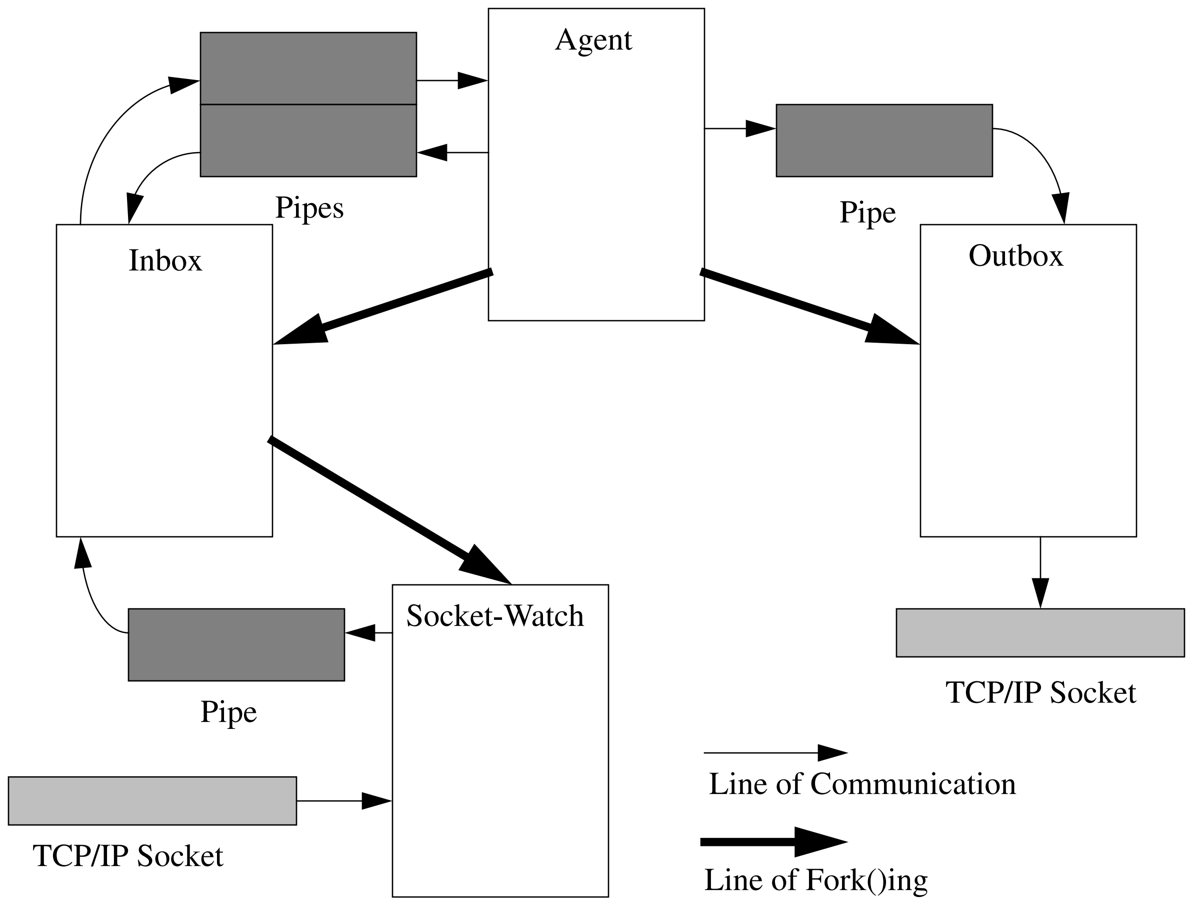
- `mailbox << perf1 << perf2 << perf3;`

4.0 The Underlying TCP/IP Interface Functions

4.1 The Details of the System

This sub-section provides a description of the set of processes that is created to send and receive character strings via UNIX sockets and some description on how the sending and receiving is accomplished.

First, a diagram:



This is the process hierarchy that the C functions `initialize_inbox` and `initialize_outbox` create. This is all encapsulated in the mbox class. In detail, the function of each of these processes is as follows:

INBOX - keeps a queue of incoming performatives and upon the agent's request pops the queue, writing the top performative to the pipe. It forks the socket-watch process with a connecting pipe. The inbox periodically checks this pipe to see if there is an incoming performative. If there is, the pipe is read and the new performative is placed at the end of the performative queue. When the agent calls `wait_for_reply`, it passes the inbox two character strings (the sender field and the reply-with field). Inbox searches for the reply-with value in the in-reply-to fields of all the queued performatives. If it is found, that performative is moved to the top of the performative queue and the agent is notified. If the string is not found, the inbox processes each new incoming performative, looking for the string. If the string is not found, the performative is placed on the bottom of the queue as usual but if it is found the performative placed at the top of the queue and the agent is notified. If the time specified by BLOCK_TIME_OUT expires, the inbox process constructs an error performative from the agent *to itself* with the comment field equal to "block_error." This performative is put on the top of the queue and the agent is notified *just as if a responding performative had been received*. Upon exit from the `wait_for_reply` function, the next performative that the agent will read is this error performative, indicating that the block has timed-out. To reiterate, when the blocking function returns, the next performative that the agent reads is either a genuine reply from the non-local agent, or it is an error message describing the undeliverability of the original performative (see the outbox section below) or the block error that had occurred.

SOCKET-WATCH - this is the process that actually watches the incoming communications socket, waiting for someone to request to connect and transfer a message. When socket-watch detects a request, it gets the incoming performative and then writes it to the pipe to the inbox. Socket-watch then pauses until inbox acknowledges receipt of the performative and then goes back to listening for another delivery request.

OUTBOX - keeps a queue of out-going performatives. When the queue is non-empty it pops the top performative and attempts to connect to the specified address and port number. If it cannot connect after a number of attempts (currently 12 attempts with an increasing (but random) pause between tries) then it constructs its own error performative and sends it to the inbox of its *own agent*. It sets the in-reply-to field, therefore there should not be a problem of an agent blocking for an undeliverable performative and never becoming unblocked. Whenever the agent sends a request, the outbox reads the pipe and puts the out-going performative on the bottom of the queue.

4.2 The C Functions

There are 6 underlying C functions that provide the interface to TCP/IP. Agent-builders should not have to use these functions at all since they are hidden by the C++ interface discussed above. Briefly, however, these functions are:

`initialize_inbox` sets-up the various sub-processes and signal catchers that enables the agent to receive performatives via a UNIX socket. The return value is 0 if no errors were found, -1 otherwise. NOTE: initialize_inbox must be called before initialize_outbox!

`initialize_outbox` initializes the sub-processes and signal catchers that enable the agent to send performatives via UNIX sockets to other agents.

`send_performative` sends a character string to the agent specified by the `port` and `address` parameters (not shown). The returned value is a -1 if the performative could not be delivered and 0 if everything went well.

`receive_performative` receive a performative. Space for the characters read from the pipe will be allocated inside the `receive_performative` function, so the value of the `*performative` parameter (not shown) before the function call should be NULL. The return value is -1 if there is no performative ready to be read and 0 if a performative has been successfully stored in performative.

`wait_for_reply` This function implements the self-blocking function that is described above in the description of the inbox process. ***NOTE****: the performa-*

tive in the `performative` *parameter must have been just sent with a* `send_performative` *function call*. The agent must call `receive_performative` upon exiting from `wait_for_reply` in order to receive the waited-for performative.

`graceful_exit` terminates the sub-processes and returns.

5.0 KQML Conformity

$KQML_{UofT}$ is a very slight super-set of the KQML draft specification found in the KQML document: *Specification of the KQML Agent-Communication Language*, The DARPA Knowledge Sharing Initiative External Interfaces Working Group, December 10, 1992. All of the performative types and parameters specified in that document can be found in this implementation.

The area that is beyond the KQML specification of December 10, 1992 is the **transport-address** performative. This performative has been proposed on the KQML-USERS mailing list and has a tentative syntax as follows:

```
(transport-address  :sender name-server
                    :receiver A1
                    :name A2
                    :language s-expressions
                    :ontology tcp-host-port
                    :content (128.100.1.20 6667))
```

This performative is designed so that the transport-address of other agents could be distributed. It is expected that most of the uses of the `transport-address` performative will be as an embedded performative inside a `tell`, `ask-if`, or other performative. (Obviously, the values of the parameters are specified by the agent-builders. The above values are just examples).

6.0 Bugs and To Whom To Complain

There are currently no known bugs in the $KQML_{UofT}$ implementation.

Questions, bugs reports, suggestions, complaints, etc. should be directed to Chris Beck at the University of Toronto (chris@ie.utoronto.ca).