# Extending ACL
# to Support Communication
# in a
# Real-Time Multi-Agent System

BY

**Lekshmi S. Nair**

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

**UNIVERSITY OF RHODE ISLAND**

**2000**

## Abstract

Agents communicate with their peers by exchanging messages in an expressive agent communication language. An Agent Communication Language (ACL) defines the types of messages (and their meanings) that agents may exchange. Knowledge Query Manipulation Language (KQML) is a high-level, message oriented communication language and protocol for information exchange and knowledge sharing among agents. It focuses on an extensible set of performatives, to describe the kinds of communication that agents can have. In addition, KQML provides a basic architecture for knowledge sharing through a special class of agents called communication facilitators, which coordinate the interactions of other agents. A real-time agent must meet its objectives within specified timing constraints, possibly trading-off the quality of its results. For example, a real-time agent might be employed to monitor stock prices to look for certain changes in the market, and report on these changes within a deadline. In order to express timing constraints in a real-time multi-agent system, the agent communication language should be suitably extended. In this thesis we develop extensions to an ACL to support real-time communication. We also implement the extended ACL in KCOBALT, an agent communication tool-kit based on KQML and CORBA. The extensions we have made to KQML are equally applicable to FIPA ACL .

## Acknowledgement

It has been my good fortune to have Dr. Lisa Cingiser DiPippo as my advisor while I worked on this thesis. This work would not be possible without her guidance, encouragement, and support. I thank her for all the valuable help and advice she has given me.

I would like to thank Dr. Dominique Benech for providing me with the KCobalt toolkit and helping me install the software.

I am also grateful to Dr. Fay Wolfe and the members of the real-time research group at the University of Rhode Island for the valuable discussions that have enriched my understanding and widened my exposure to this interesting subject.

I thank Lorraine for making my experience at URI a more enjoyable one.

Finally, I wish to thank Ranjit, my parents and my brothers for providing me with their support and patience through this research.

## 1.0 Introduction

The software world is one of great richness and diversity. Many thousands of software products are available today, providing a wide variety of information and services in a wide variety of domains. While most of these programs provide their users with significant value when used in isolation, there is increasing demand for programs that can interoperate, that is exchange information and services with other programs and thereby solve problems that cannot be solved alone.

Part of what makes interoperability difficult is heterogeneity. Agent-based software engineering was invented to facilitate the creation of software able to interoperate in such settings. In this approach to software development, application programs are written as software agents. These software components communicate with their peers by exchanging messages in an expressive agent communication language.

Agent-based software engineering is often compared to object-oriented programming. Like an object, an agent provides a message-based interface independent of its internal data structures and algorithms. The primary difference between the two approaches lies in the language of the interface. In general object-oriented programming, the meaning of a message can vary from one object to another. In agent-based software engineering, agents use a common language with an agent-independent semantics [11].

Agent technology is beginning to be used to solve real-world problems in a range of industrial and commercial applications. Businesses and consumers are relying more and more on automated processes to handle the buying and selling of goods and services. Many of these systems such as real-time auction systems, stock-market quoting systems, and goods pricing systems, have inherently autonomous features as well as tight constraints on when and how they can execute specific tasks. These types of applications could benefit from a real-time multi-agent system in which agents communicate,

coordinate and negotiate to meet their goals, within specified timing and quality constraints.

In this thesis we present extensions to support timing and quality constraints in an agent communication language, KQML to support real-time communication between agents and implement the extensions in KCobalt, an agent communication tool-kit based on KQML and CORBA. Section 2 provides the background and related work that has formed the foundation for the work of this thesis. It introduces Agent and Muti-Agent System and contains a review of KQML and FIPA ACL. It also describes the KCobalt system and provides the background on the real-time system. Section 3 presents our model for Real-Time Multi-Agent System (RTMAS) on which our architecture and real-time agent services are based. It then describes an example electronic commerce application to illustrate the features of the model and the RTMAS architecture. Section 4 presents the real-time extensions made in KQML. Section 5 describes the implementation and integration details of the extensions in the KCobalt system. Section 6 concludes the thesis with a summary of the contributions of this work and a discussion of future work.

## 2.0 Background

This section provides foundational definitions and discusses related work in the areas of multi-agent systems and real-time agents. It gives background information about the various features of KQML and compares it with another existing agent communication language, FIPA ACL.

## 2.1 Agents & Multi-Agent System (MAS)

An agent is a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives [1]. Situatedness, in this context, means that the agent receives sensory input from its environment and that it can perform actions which change the environment in some way. Examples of environments in which agents maybe situated include the physical world or the Internet. An autonomous system

should be able to act without the direct intervention of humans (or other agents), and it should have control over its own actions and internal state. Examples include any process control system, which must monitor a real-world environment and perform actions to modify it as conditions change, such systems range from very simple thermostats to extremely complex nuclear reactor control systems. Another example is a software daemon, which monitors a software environment and perform actions to modify the environment as conditions change. A simple example is the UNIX xbiff program, which monitors a user's incoming email and obtains their attention by displaying an icon when new, incoming email is detected. An agent should respond to its environment, and should exhibit opportunistic, goal-directed behavior and interact with other agents to solve its own problems as well as helping others with their activities.

A multi-agent system is designed and implemented as several interacting agents. Multi-agent systems are ideally suited to representing problems that have multiple problem solving methods or multiple problem solving entities. Most multi-agent systems provide a specialized agent, called a `facilitator`, which is tasked with finding agents to fulfill services required by requestor agents. There are two types of facilitators that can be employed. In both models, each agent registers with the facilitator and advertises the services that it can perform on behalf of other agents. When an agent requests a service from a `broker` facilitator, the broker passes the request along to the agent that provides the requested service. If no such agent exists, that is if the requested service has not been advertised to the broker by any agent, the broker responds to the requesting agent with a message. A `matchmaker` facilitator works in much the same way as an agent broker. However, when a request for service is made to a matchmaker, the matchmaker agent passes along a reference to the agent that can provide the service. That is, the matchmaker puts together, or matches, agents that can work together. Once this match is made, the agents can communicate with each other directly without the use of the matchmaker.

## 2.2 KQML

Knowledge Query and Manipulation Language (KQML) is a language that is designed to support interactions among intelligent software agents [2]. It was developed by the DARPA (Defense Advanced Research Projects Agency) supported Knowledge Sharing Effort. The DARPA Knowledge Sharing Effort (KSE) is a consortium to develop conventions facilitating sharing and reuse of knowledge bases and knowledge based systems. Its goal is to define, develop, and test infrastructure and supporting technology to enable participants to build much bigger and more broadly functional systems than could be achieved working alone [2].

KQML focuses on an extensible set of performatives, which defines the permissible “speech acts” agents may use in communicating with each other. It is a language and a set of protocols that support computer programs in identifying, connecting with and exchanging information with other programs. The performative signifies that the content is an assertion, a query, a command, or any other mutually agreed upon speech act. It also describes how the sender would like any reply to be delivered, that is what protocol will be followed.

### 2.2.1 KQML message Three-layer organization

A KQML message consists of three layers[6] : content, communication, and message. The content layer bears the actual content of the message in the program’s own representation language. KQML can carry any representation language, including languages expressed as ASCII strings and those expressed using binary notation. Every KQML implementation ignores the content portion of the message, except to determine where it ends.

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

The message layer, which encodes a message that one application would like to transmit to another, is the core of KQML. This layer determines the kinds of interactions one can have with a KQML-speaking agent. A primary function of the message layer is to identify the protocol to be used to deliver the message and to supply a speech act or performative that the sender attaches to the content. The speech act indicates whether the message is an assertion, a query, a command, or any of a set of known performatives.

In addition, since the content is opaque to KQML, the message layer also includes optional features that describe the content language, the ontology (set of term descriptions for a specific domain of discourse) it assumes, and some type of description of the content, such as a descriptor naming a topic within the ontology. Every agent incorporates some view of the domain (and the domain knowledge) it applies to. The technical term for this body of background knowledge is ontology. More formally, an ontology is a particular conceptualization of a set of objects, concepts, and other entities about which knowledge is expressed, and of the relationships among them [6]. These features make it possible for KQML implementations to analyze, route, and properly deliver messages whose content is inaccessible.

### 2.2.2 Syntax and Performatives

The syntax of KQML is based on a balanced parenthesis list [2]. The initial element of the list is the performative; the remaining elements are the performative's arguments as keyword / value pairs.

For example, a KQML message from agent `joe` representing a query about the price of a share of IBM stock can be encoded as:

```
(ask-one
     :sender      joe
     :content    (PRICE IBM ?price)
     :receiver    stock-server
     :reply-with  ibm-stock
     :language    LPROLOG
     :ontology    NYSE-TICKS)
```

In this message, the KQML performative is `ask-one`. It sends a query asking about the price of a share of IBM stock. The content which encloses the actual message is (`PRICE IBM ?price`). The ontology assumed by the query is identified by the token `NYSE-TICKS`. This indicates that the message uses the tickers in the New York Stock Exchange. The receiver of the message is to be a server identified as `stock-server` and the query is written in a language called `LPROLOG`. The value of the `:content` keyword is the content level, the values of the `:reply-with`, `:sender`, and `:receiver` keywords form the communication layer; and the performative name with the `:language` and `:ontology` form the message layer. In due time, the `stock-server` might send to `joe` the following KQML message:

```
(tell
     :sender      stock-server
     :content    (PRICE IBM 14)
     :receiver    joe
     :in-reply-to ibm-stock
     :language    LPROLOG
     :ontology    NYSE-TICKS)
```

This message binds the value of a share of IBM stock to the variable price and sends it as a response to the query message received with the identifier `ibm-stock`.

Altogether KQML proposes a set of 36 performatives divided into three domains :

1) discourse - normal conversation between agents
   e.g. ask-if, ask-one, ask-all, advertise, tell, stream-all
2) intervention - error & information processing messages
   e.g. error, sorry, standby, discard
3) facilitation - messages to facilitators & network information
   e.g. broker, recruit, recommend, broadcast

Table 1 gives a summary of reserved performatives for :sender S and :receiver R [3]. The KQML Reserved Parameter Keywords are given in the Table 2 [3].

| Name | Meaning |
|---|---|
| ask-if | S wants to know if the :content is in R's VKB |
| ask-all | S wants all of R's instantiations of the :content that are true of R |
| ask-one | S wants one of R's instantiations of the :content that is true of R |
| stream-all | multiple-response version of ask-all |
| eos | the end-of-stream marker to a multiple-response (stream-all) |
| tell | the sentence is in S's VKB |
| untell | the sentence is not in S's VKB |
| deny | the negation of the sentence is in S's VKB |
| insert | S asks R to add the :content to its VKB |
| uninsert | S wants R to reverse the act of a previous insert |
| delete-one | S wants R to remove one matching sentence from its VKB |
| delete-all | S wants R to remove all matching sentences from its VKB |
| undelete | S wants R to reverse the act of a previous delete |
| achieve | S wants R to do make something true of its physical environment |
| unachieve | S wants R to reverse the act of a previous achieve |
| advertise | S wants R to know that S can and will process a message like the one in a :content |
| unadvertise | S wants R to know that S cancels a previous advertise and will not process any more messages like the one in the :content |
| subscribe | S wants updates to R's response to a performative |
| error | S considers R's earlier message to be malformed |
| sorry | S understands R's message but cannot provide a more informative response |
| standby | S wants R to announce its readiness to provide a response to the message in :content |
| ready | S is ready to respond to a message previously received from R |
| next | S wants R's next response to a message previously sent by S |
| rest | S wants R's remaining responses to a message previously sent by S |
| discard | S does not want R's remaining responses to a previous (multi-response) message. |
| register | S announces to R its presence and symbolic name |
| unregister | S wants R to reverse the act of a previous register |
| forward | S wants R to forward the message to the :to agent ( R might be That agent) |
| broadcast | S wants R to send a message to all agents that R knows of |
| transport-address | S associates its symbolic name with a new transport address |
| broker-one | S wants R to find one response to a <performative> (some agent other than R is going to provide that response) |
| broker-all | S wants R to find all responses to a <performative> (some agent other than R is going to provide that response) |
| recommend-one | S wants to learn of an agent who may respond to a <performative> |
| recommend-all | S wants to learn of all agents who may respond to a <performative> |
| recruit-one | S wants R to get one suitable agent to respond to a <performative> |
| recruit-all | S wants R to get all suitable agents to respond to a <performative> |

**Table 1. Summary of reserved performatives for :sender S and :receiver R**

| Keyword | Meaning |
|---|---|
| :sender | name of the agent sending the performative |
| :receiver | name of the agent receiving the performative |
| :reply-with | unique identifier for a message, to be referenced in later communications |
| :in-reply-to | the expected label in a response to a previous message (extracted from :reply-with) |
| :language | name of the representation language used in :content |
| :ontology | name of the ontology used in :content |
| :content | the information carried within the performative, for which it expressed an attitude. |

**Table 2. Summary of reserved parameter keywords and their meanings**

### 2.2.3 Semantics

The semantics of KQML is provided in terms of `preconditions`, `postconditions`, and `completion conditions` for each performative. Assuming a sender A and a receiver B, preconditions indicate the necessary states for an agent to send a performative, **Pre(A)**, and for the receiver to accept it and successfully process it, **Pre(B)**. If the preconditions do not hold, the most likely response is `error` or `sorry`.

Postconditions describe the states of the sender after the successful utterance of a performative, and of the receiver after the receipt and processing of a message but before a counterutterance. Postcondition **Post(B)** holds unless a `sorry` or an `error` is sent as response to report the unsuccessful processing of the message.

A completion condition for the performative, **Completion**, indicates the final state. For example, after a conversation has taken place and the condition indicates whether the intention associated with the performative that started the conversation has been fulfilled.

Preconditions, postconditions, and completion conditions describe states of agents in a language of mental attitudes (belief, knowledge, desire and intention) and action descriptors (for sending and processing a message). For example the semantics for `tell` performative suggests that an agent cannot offer unsolicited information to another agent. Suppose that an agent A wants to tell agent B the truth about X, tell(A,B,X). Then Pre(A) is that A believes X to be true and agent A knows that agent B wants to know the truth about X. Pre(B) is that agent B intends to know the truth of X. Post(A) will be that A knows that B knows A believes X to be true. Post(B) is that B knows that A believes X to be true. The Completion condition for the tell performative is that agent B knows that agent A believes X to be true.

### 2.2.4 Suitability of KQML for Cooperative Multi-Agent interactions

The different interaction needs existing in the case of cooperative intelligent management agents can be divided into 7 types: affirmation, negation, question, action request, acceptation, denial and error [5]. Therefore KQML must cover each of these cooperation interactions through its performatives.

It can be verified that KQML is able to answer to these seven interactions by establishing a correspondence between them and one or several KQML performatives that support each interaction in an inter-agents message as given in table below.

| **Cooperation Interactions** | **KQML performatives** |
|---|---|
| Affirmation | tell, insert |
| Negation | deny, untell, uninsert |
| Question | ask-if, ask-all, ask-one, stream-all |
| Action request | achieve |
| Acceptation | tell |
| Denial | sorry |
| Error | error |

**Table 3. KQML response to needs for cooperation interactions**

Thus KQML proposes at least one performative for each interaction that can take place between cooperative intelligent agents. Therefore it is suitable to offer a layer allowing high-level interactions between agents. KQML also provides the notion of a facilitator that provides indispensable base services for a multi-agent system. Considering these features of KQML, we used it as the language to extend to support real-time communication in a multi-agent system.

## 2.3 FIPA ACL

FIPA's (The Foundation for Intelligent Physical Agents) agent communication language (ACL), like KQML, is based on speech act theory [4]. Messages are actions or communicative acts, and they are intended to perform some action by virtue of being sent. FIPA ACL is superficially similar to KQML. Its syntax is identical to KQML's except for different names for reserved primitives. KQML uses performatives to refer to communication primitives whereas in FIPA ACL, the communication primitives are called `communicative acts`, or CAs for short.

The FIPA ACL specification document[4] claims that FIPA ACL (like KQML) does not make any commitment to a particular content language. This claim holds true for most primitives. However to understand and process some FIPA ACL primitives, receiving agents must have some understanding of Semantic Language, or SL which is the language used to define FIPA ACL's semantics. More details are given in Section 2.3.1.

Given below is an example of an ACL message:

```
(request
     :sender      test-agent
     :receiver    ping-agent
     :content     (inform
                    :sender      ping-agent
                    :receiver    test-agent
                    :content     (alive)
                    :language    simple)
                    :language    fipa-acl)
```

In this message the test-agent is sending a request to the ping-agent to inform test-agent whether ping-agent is alive or not. The ACL message that the test-agent expects to receive in response to its request is shown below:

```
(inform
     :sender   ping-agent
     :receiver test-agent
     :content  (alive)
     :language  simple)
```

The semantics of the `request` communicative act do not guarantee that the ping-agent will act upon the request made by the test-agent. It is therefore possible that the test-agent will not receive the inform message as expected even though the ping-agent is in fact alive. The impact of such a result is that the test-agent is still unaware of the ping-agent's status.

### 2.3.1 Semantics

SL is the formal language used to define FIPA ACL's semantics. In FIPA ACL, the semantics of each communicative act is specified as sets of SL formulae that describe the act's feasibility preconditions and its rational effect. For a given CA 'a', the feasibility preconditions FP(a) describe the necessary conditions for the sender of the CA. That is for an agent to properly perform the communicative act 'a' by sending a particular message, the feasibility preconditions must hold for the sender. The agent is not obliged to perform 'a' if FP(a) holds, but it can if it chooses. A communicative act's rational effect represents the effect that an agent can expect to occur as a result of performing the action. It also typically specifies conditions that should hold true of the recipient. The FPs and the REs involve agents state descriptions that are given in SL.

Consider the CA `inform`, in which agent `i` informs agent `j` of content $\phi$, <`i,inform(j,` $\phi$)> . The content of `inform` is a proposition, and its meaning is that

the sender informs the receiver that a given proposition is true. Here the FP is similar to the precondition and RE is similar to the postcondition for the `tell` performative of KQML. According to FIPA ACL semantics the FP for `inform` is that agent `i` holds the proposition is true and does not believe that agent `j` has any knowledge of the truth of the proposition. RE for the communicative act is, agent `i` intends that agent `j` should also come to believe that the proposition is true.

## 2.4 Comparing the ACLs

KQML and FIPA ACL are almost identical with respect to their basic concepts and the principles they observe. The two languages have the same syntax. They differ semantically at the level of what constitutes the semantic description: preconditions, postconditions, and completion conditions for KQML, feasibility conditions and rational effect for FIPA ACL.

Both languages assume a basic noncommitment to a reserved content language. However, in the FIPA ACL, an agent must have some limited understanding of SL to properly process a received message.

Another difference between the two ACLs is in their treatment of the registration and facilitation primitives. These primitives cover a range of important pragmatic issues, such as registering, updating registration information, and finding other agents that can be of assistance in processing requests (register, unregister, broker, recommend, recruit). In KQML, these tasks are associated with performatives that the language treats as first-class objects. FIPA ACL does not consider these tasks CAs. Instead, it treats them as requests for action and defines a range of reserved actions that cover the registration and life cycle tasks. In this approach, the reserved actions do not have formally defined specifications or semantics and are defined in terms of natural-language descriptions.

## 2.5 KCOBALT

KCobalt [5] is an agent communication tool-kit based on KQML and CORBA. CORBA (Common Object Request Broker Architecture) is a standard software specification for distributed environments in which all entities are treated as distributed objects [14]. Its main role is to provide message transport between agents. CORBA also offers standard services to facilitate distribution. These services can be used to simplify the implementation of the `facilitator` concept at the KQML level. The KQML layer contacts the CORBA services it needs to provide services of the `facilitator`. The CORBA architecture allows for location, language and operating system transparency. To provide such functionality, it uses interface definitions, written in the CORBA Interface Definition Language (IDL), specifying the methods, parameters, types and exceptions a CORBA object supports [14].

The KCobalt cooperation framework offers communication abstraction between intelligent agents to allow them to exchange information and messages. This communication abstraction, represented in the case of interaction between two intelligent agents, is shown in Figure 1[12].

KCobalt offers to intelligent agents a communication interface to allow them to exchange messages as well as interaction content through KQML performatives. Message transport itself is provided by CORBA. By combining KQML and CORBA, KCobalt offers the communication and interaction framework to cooperative intelligent agents. Intelligent agents using it have to be seen as KQML agents at the KQML level and as CORBA objects at the CORBA level.

In order to go from the agent level to the KQML level, and express messages of cooperative intelligent agents under the KQML textual form, there is a parser/generator. To go from the KQML level to the CORBA level, an interface is provided written in IDL language. It describes all messages the intelligent agents (seen as CORBA objects at this

level) can exchange. Here we move from asynchronous communication in KQML to synchronous client/server exchanges in CORBA requests.

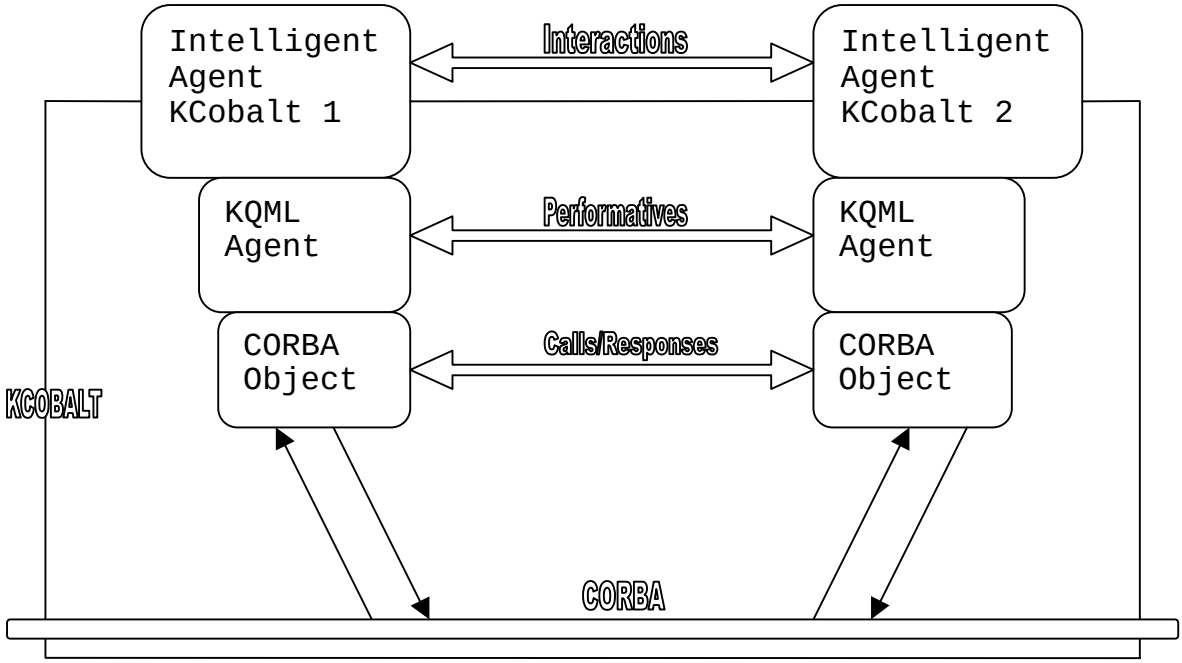


**Figure 1. Communications between two KCobalt agents**

## 2.5.1 Base architecture

The functional architecture depicted in Figure 2[12] for KCOBALT consists of the following modules:

• Parser/ Generator module:

This module helps to go from the agent level to the KQML level, and express messages of agents under the KQML textual form. The parser intercepts the KQML textual messages emitted by agents. It decomposes the messages, identifying the nature of the performatives and the fields matching associated parameters. The generator rebuilds a textual message from the received data. The intelligent agent can then process this textual message.

- Services calls processing/ redirection module:

This module intercepts all KQML messages transmitted to the KCOBALT framework and identifies the ones addressing services and especially the KQML facilitator. Depending on the service asked for, it may process it locally or transfer (possibly after translating it) the request to the suitable service.

- KQML/ CORBA interface module:

This module identifies KQML performatives, translates the parameters to types CORBA can deal with, and calls methods matching the performatives on the CORBA object . In the KCobalt architecture, inspired from Y.Labrou's work in [3] (Table 5), the KQML performatives are divided into three categories.

1) All agents category:

These are performatives all KQML agents can exchange. They form the Core performatives. There are 27 core performatives.

2) Facilitators category:

These are performatives used for communication with KQML facilitators (being specific KQML agents, providing services). There are only three pure facilitation performatives : `register`, `unregister`, `transport-address`

3) Optional category:

These are performatives available only if published. There are six optional performatives (these also are facilitation performatives as Y.Labrou described). They are `broker-one`, `broker-all`, `recruit-one`, `recruit-all`, `recommend-one`, `recommend-all`

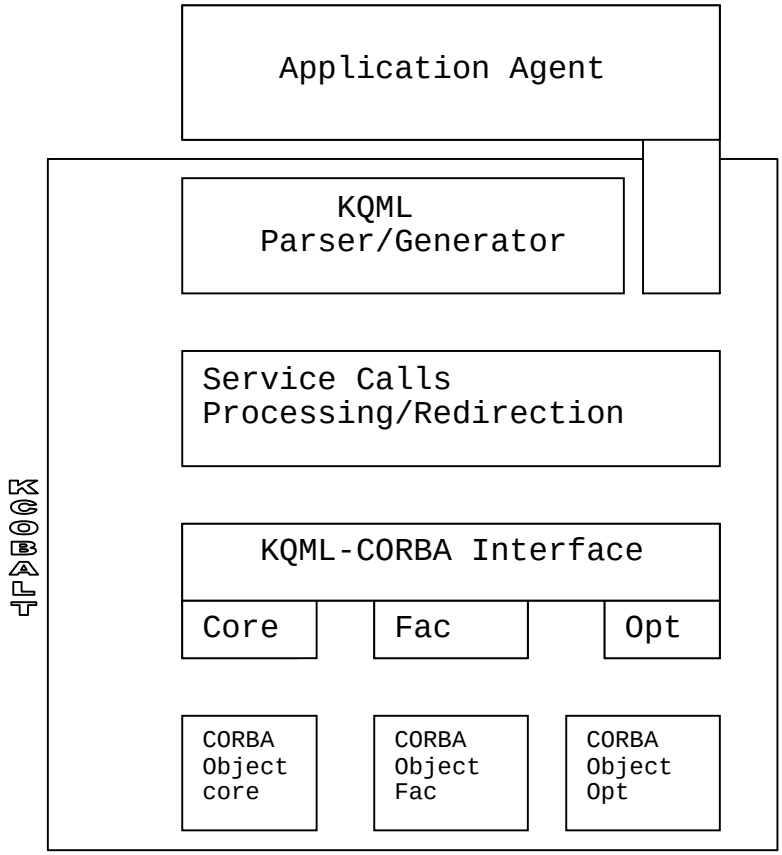


**Figure 2. KCOBALT Functional Architecture**

Depending on the performative, the interface uses Core, Facilitation or Optional functions.

• CORBA object modules:

These modules are independent of each other and they support message marshaling and transport. Depending on the concerned performatives, the Core, Facilitation or Optional module will be used to transport messages from one KCobalt infrastructure to another one, and finally from one intelligent agent to another one.

• Performative Exchanger:

This is a simple user interface provided with KCobalt for testing. This GUI can be used for sending and receiving performatives one after the other to simulate inter-agent communication.

KCobalt, based on the combination of KQML and CORBA, offers an open and generic cooperation framework between intelligent agents. It is therefore a suitable framework to develop any cooperative intelligent system. So KCobalt was used as the basis for implementing real-time extensions to KQML for inter-agent communication.

## 2.6 Real-Time Systems

A real-time system is a system in which the time at which the output is produced is significant [13]. The input corresponds to some "movement" in the physical world, and the output has to relate to the same movement. Real-time systems interact with the external world in a way that involves time. When a stimulus appears, the system must respond to it in a certain way and before a certain deadline. If it delivers the correct answer, but after the deadline, the system is regarded as having failed.

Real-time systems are generally split into two types [9] depending on how serious their deadlines are and the consequences of missing one. These are:

1) Soft real-time systems.
2) Hard real-time systems.

Soft real-time means that missing an occasional deadline is all right. For example, a telephone switch might be permitted to lose or misroute one call in $10^5$ under overload conditions and still be within specification. In contrast, even a single missed deadline in a hard real-time system is unacceptable, as this might lead to loss of life or an environmental catastrophe. e.g : nuclear reactor. There are intermediate systems with firm real-time where missing a deadline is not catastrophic, but there is no value in continuing with operation if timing constraint is not met. For example, if a robot in an assembly line

conveyor belt missed picking up the object being passed, there is no point in attempting to pick it up later.

### 2.6.1 Real-Time Extensions to SQL

SQL (Structured Query Language) is a standard relational database language. RTSQL [7] includes extensions to support real-time databases that specify timing constraints on execution along with other constraints on data and transactions. The most developed support in RTSQL is for timing constraints. RTSQL specifies time constrained execution by placing timing constraints on individual data manipulation statements or a block of statements.

The START BEFORE and COMPLETE BEFORE clauses are used to express the latest start time and latest finish time for the execution of the statement. The START AFTER and COMPLETE AFTER clauses are used to express the earliest start time and earliest finish time for the execution of the statement. The SQL standard, SQL2 provides three datetime data types: DATE, TIME, TIMESTAMP. There is also an interval data type called INTERVAL, that can be used to express a period of time, such as 5 minutes. SQL also supports three datetime valued functions: CURRENT_DATE returns the current date, CURRENT_TIME returns the current time, and CURRENT_TIMESTAMP returns the current date concatenated with the current time.

The IMPORTANCE LEVEL directive allows for the specification of the relative importance of an action. A scheduling algorithm may use relative importance of tasks as a parameter in determining scheduling priority of the tasks. The basic data manipulation operations SELECT, INSERT, UPDATE and DELETE were implemented with timing constraints specified for each action.

This work helped in deciding the appropriate places to include timing constraints and to do parallel extension of an agent communication language in order to support real time communication between agents.

### 2.6.2 Related Real-Time Agent Work

Here we discuss related real-time agent work that has been done.

TAEMS (Task Analysis, Environment Modeling, and Simulation) is a domain independent task modeling framework used to describe and reason about complex problem solving methods [15]. TAEMS models are used in multi-agent coordination research and are used in many projects. TAEMS models are hierarchical abstractions of problem solving processes that describe alternative ways of accomplishing a desired goal. All primitive actions in TAEMS, called *methods*, are statically characterized via discrete probability distributions in three dimensions: quality, cost and duration. Quality is a deliberately abstract domain-independent concept that describes the contribution of a particular action to overall problem solving. Duration describes the amount of time that the action modeled by the method will take to execute and cost describes the financial or opportunity cost inherent in performing the action.

Soft constraints in TAEMS take the form of soft commitments made with other agents and soft interactions between tasks. In the model, a client may specify an overall soft deadline, soft cost limit, or soft quality requirement. These soft constraints are members of a package of client preferences called *design criteria* that describes for the scheduler the client's objective function. The scheduler then makes trade-off decisions as needed to best address the client's needs.

AMSIA is another agent architecture that supports real-time requirements. It combines the possibility of using different reasoning methods with a mechanism to control the resources needed by the agent to fulfill its high level objectives [16]. Agent missions in this model have as relevant parameters their priority (importance for the agent) and deadline. In this architecture it is the control mechanism which is in charge of deciding which tasks to execute. The goal of the control mechanism is to maximize the profit of the line of activity of the agent. To do so, it always tries to favor the sequences of tasks corresponding to more important plans and to use tasks to carry out those plans

which offer more quality. The control mechanism can remove sequences of tasks corresponding to less important plans to favor the introduction of those corresponding to more important ones. To decide between sequences of tasks the control mechanism scores them taking into account both the importance of the objectives of the plan and the quality offered by the tasks used to follow it.

These works helped in coming up with the appropriate constraints needed to be incorporated in our RTMAS model. As of now, no published work is available that describes extending an agent communication language like KQML to handle real time requirements on agents.

## 3.0 Real-Time Multi-Agent System

A real-time agent must meet its objectives within specified timing constraints, possibly trading-off the quality of its results. For example, a real-time agent might be employed to monitor stock prices to look for certain changes in the market, and report on these changes within a deadline. In order to express and enforce the timing and other quality of service (QoS) constraints of individual agents, a real-time multi-agent system (RTMAS) must provide services that allow the real time agents to communicate, coordinate, and cooperate to meet the goals of their particular application and the specified QoS constraints.

This section describes an example Real-Time Multi-Agent application and presents our model for RTMASs on which our architecture and real-time agent services are based.

## 3.1 An Example Real-Time Multi-Agent Application

A stock trading system is an example of an application for which a real-time multi-agent system(RTMAS) would be useful. Here the real-time agents work together to meet their

goals, and specified QoS requirements and coordinate to make intelligent recommendations, purchases and sales of stocks.

Suppose there are four different types of agents: a *UserAgent*, a *QuotingAgent*, a *TrendWatchingAgent*, and a *BuySellAgent*. The *UserAgent* communicates with the human user to determine her requirements, such as risk level, amount of money to spend, and market sector preferences. The *UserAgent* also communicates with the other agents in the system to be able to make recommendations to the user. Each *QuotingAgent* has the ability to get quotes on stocks on a particular sector of the market. It can also monitor a particular stock for a particular price range. *QuotingAgents* communicate with the other agents that require information about stock prices. They can also communicate with each other if a request is made to one *QuotingAgent* for a stock that it cannot quote.

The *TrendWatchingAgent* looks for particular trends in the market. Each specific *TrendWatchingAgent* may be responsible for a particular kind of trend, such as a long-term increase in biotechnology stocks. When a *TrendWatchingAgent* recognizes a trend that might be interest to other agents, it notifies them. The *TrendWatchingAgent* communicates with a *UserAgent* if the *UserAgent* has expressed interest in a particular trend. It communicates with the *QuotingAgent* in order to get quotes on specific stock prices.

The *BuySellAgent* is responsible for actual purchases and sales of stocks. This kind of agent can act autonomously if the human user has expressed to the *UserAgent* that transactions can be made automatically. In this case, the *UserAgent* utilizes the user's profile and information from the other agents in the system to specify buy and sell transactions to the *BuySellAgent*. If the user wants to be involved in each transaction, then the *UserAgent* can make recommendations to the user, get her approval, and then notify the *BuySellAgent* to perform transaction.

The timing constraints on this RTMAS stem from the volatility of prices in the stock market. For example, if the *UserAgent* determines that the user should purchase

100 shares of Techno stock because the price is currently relatively low, then it must specify a deadline to the *BuySellAgent* by which the transaction must be made in order to realize the desired result.

## 3.2 RTMAS Model

The Real Time Multi-Agent System Model proposed by the Real Time Research group in URI [8] embodies the features and functionality required to express and enforce timing constraints on real-time agent interactions. The model is based on the assumption that many agents can perform their tasks in multiple ways. Each of the methods of execution of an agent's task is associated with a worst case execution time and an expected amount of quality returned. The model is made up of a set of real-time agents(*RTAgent*) and a set of communications among the real-time agents(*Request*).

### 3.2.1 RTAgent

A real-time agent can be defined as follows:

$RTAgent = \{S_1, S_2, ...., S_n\}$

Each *RTAgent* is comprised of a set of *solvables*, $\{S_1, S_2, \ldots S_n\}$, where a solvable is a problem that the agent is designed to solve. Each solvable within the agent is represented by an optimal result (O) and a set of execution strategies (ES):

$S_i = < O, ES >$

The optimal result for a solvable may vary from environment to environment depending upon the developer, the user, and the intended use of the agent. This is an objective, system-specific definition of what is considered to be the absolute best result for the problem. For instance, in the case of stock trading , the *BuySellAgent* may have a solvable, *BuyStock,* to purchase a particular stock. The optimal solution in this scenario might be to buy the stock at the current price with no fee.

In the model of a solvable, ES represents a set of execution strategies that can be used to produce a result for the solvable:

$$ES = \{es_1, es_2, es_n\}$$

For example, the solvable *BuyStock* may have an execution strategy, $BS_1$, that uses a discount broker with a low fee. This execution strategy may come close to the no fee requirement of the optimal result, but if the discount broker typically has a longer turn around time, then the deadline of the *BuyStock* request may be violated and the price of the stock may have changed. On the other hand, an execution strategy, $BS_2$, that uses a more expensive broker may be able to handle the request more quickly.

Each execution strategy of a solvable is comprised of three elements:

$$es_i = < ex, q, tv >$$

The execution time (*ex*), represents the amount of time it takes a strategy to run. Quality (q) is calculated as a percentage of the optimal result such that *q = (strategy result / optimal result )*. This definition of quality is conditional upon the ability to quantify the result of a task. In the example above, we quantify the optimal result of the *BuyStock* solvable by specifying zero fee for the transaction. While this optimal result may be impossible to achieve, it provides a metric by which to measure the results of the actual execution strategies.

In this model, the quality of an execution strategy must be known ahead of time. In some cases, an average quality will have to be used to represent the actual quality returned by an execution strategy in a specific scenario. For example, in the *BuyStock* solvable, it may not be possible to know the exact quality returned by its execution strategies if the fee is based on a percentage of the exact stock price. Instead, we can determine a statistical delta from the requested stock price for the particular broker, and

compute the fee based on this estimate. This estimate may be updated based on the actual use of the system in which the real-time agent exists.

The last component of an execution strategy is the tradeoff value (*tv*). This parameter provides a measure of how much value will be lost by reducing the execution strategy of a solvable. The tradeoff value is defined as the change in quality between two execution strategies, divided by the change in time. More precisely, for any $es_i$, we have :

$$tv_i = \frac{\frac{q_i - q_{i+1}}{q_i}}{ex_i - ex_{i+1}}$$

### 3.2.2 Real-Time Agent Request

Communication among agents in this model is performed through requests for service from one agent to another. The formal specification for a request R is

$$R = < A, V, I, D, H >$$

*A* represents the name of the real-time agent to which the request is directed. *V* is the name of the solvable that the client is requesting to be performed. *I* is the level of importance of the request. This value is based on some system–wide scale of importance agreed-upon by all agents. *D* represents the deadline by which the request must be completed. In the model, this deadline can be either a soft deadline, or a firm deadline, depending upon the requirements of the application. *H* specifies the quality threshold for the request. That is, the requesting agent expresses through *H*, the minimum quality returned by the request. If the servicing agent cannot provide this amount of quality, then the requesting agent may choose to abort the request.

As an example of a real-time agent request, consider a *UserAgent* in the stock trading example. It may send a request to the *QuotingAgent* for the price of IBM stock *GetPrice(IBM)*. In the real-time agent request *A* gets the value *QuotingAgent* and *V* gets the value *GetPrice(IBM)*.The deadline that the *UserAgent* specifies on this request may be based on the requirements of some other transaction that the *UserAgent* is performing. The *UserAgent* may specify a quality threshold that allows for a quarter of a point difference from the actual stock price in order to meet its deadline. The importance of the request depends upon the overall transaction that the *UserAgent* is attempting to perform. If the transaction involves spending a few hundred dollars, then the importance is low. But if it involves thousands of dollars, the importance may be higher. This model of a request forms the basis for our real-time extension to KQML described in the next section.

## 4.0 Real-Time Extensions to KQML

In order to express timing constraints in a real-time multi-agent system, we must extend the expressibility of the agent communication language used. Such a language is a protocol for information exchange and knowledge sharing among agents. These languages provide “communicative acts” or “performatives” to describe the kinds of communication that agents can have. Due to the similarities between the languages, most of the extensions we are considering for KQML are equally applicable to FIPA-ACL.

This thesis extends the expressibility of the agent communication language KQML in order to express timing constraints in a RTMAS. There are two types of communication that we have identified as requiring extension: (1) a request from one agent to another, and (2) an advertisement of capabilities from a servicing agent to a facilitator.

The KQML performatives are extended to include the constraints such as deadline, importance, and quality to represent *RTAgent* requests. For example, consider a

request from a *UserAgent* to a *TrendWatchingAgent*, to report on current trends in internet stocks within 15 seconds. The KQML request will look like the following:

```
(ask-one
     :sender        UserAgent
     :receiver      TrendWatchingAgent
     :reply-with    Trend
     :Qos_requirement (dl=15,imp=4,qual=75)
     :language      Java
     :ontology      Stock
     :content       Watch(internet))
```

The `Qos_requirement` parameter is added to the KQML `ask-one` performative to allow for the expression of the deadline (`dl`), the importance (`imp`) and the quality threshold (`qual`) for this request. All these constraints are included as part of a `Qos_requirement` parameter in order to allow for the addition of further quality of service constraints in the future.

Communication between agents and facilitators must also be extended to allow for expression of timing capabilities. All agents that provide services to other agents must advertise with a facilitator. For example, the *BuySellAgent* has a solvable to buy a stock (*BuyStock*). It has two execution strategies, each with a specific execution time and quality. The facilitator message to advertise the capabilities of this agent is as follows:

```
(advertise
     :sender        BuySellAgent
     :receiver      Facilitator
     :Qos_capabilities   (
                       (ex=5,qual=85)
                       (ex=2,qual=65) )
     :language      Java
     :ontology      Stock
     :content       BuyStock(A))
```

In this example, the *BuySellAgent* specifies through the `Qos_capabilities` parameter that it has two execution strategies, one can execute in 5 seconds with a

returned quality of 85, and the other that can execute in 2 seconds with a returned quality of 65. The `Qos_capabilities` used to express the quality of service characteristics can be easily extended.

In most real-time applications deadlines or other time constraints are present on the agent's problem solving. We can extend KQML by either adding new performatives, adding new parameters or creating new ontologies. For real-time communication between agents extending KQML by adding new parameters is the best choice. Our real-time extension to an agent request which includes `Qos_requirement` parameter in the KQML performatives takes into account the deadline, importance and quality constraints that the user inputs for the particular message. These constraints can then be used by the real-time scheduling algorithm run by the real-time scheduler which will be incorporated in future. The scheduler determines the scheduling priority and execution time values for each agent request depending on the constraint values specified by the user. The `Qos_capabilities` parameter added to the `advertise` performative allows an agent communicating with the facilitator to specify the execution time as well as the quality of the result returned by each execution strategy for a solvable of that agent. The real-time facilitator which will be incorporated into the system in future, communicates with the real-time scheduler to determine which agent can best satisfy the user's request using alternate solution methods, or trading-off quality depending on what is available.

## 5.0 System Implementation

The extended Real-Time Agent Communication Language is implemented as part of a Real-Time Multi-Agent System prototype being developed at URI [8]. This prototype is based on the RTMAS architecture depicted in Figure 3. It is a multi-layered architecture. At the lowest layer, there is a POSIX-compliant real-time operating system. Above that is the real-time ORB layer. The real-time middleware services layer consists of the Scheduling Service and the RT Trader Service. The Scheduling Service assigns priority to servers and to client requests. The RT Trader Service receives requests for service from clients with the specified timing constraints, such as deadline and importance. It

then determines which server object can respond to the request within the specified timing constraints. Finally, the real-time agent services layer extends the Scheduling Service and the RT Trader Service of the previous layer to provide a Real-Time Agent Scheduling Service and a Real-Time Facilitator Service. The agent services layer also provides a service for Real-Time Agent Communication.

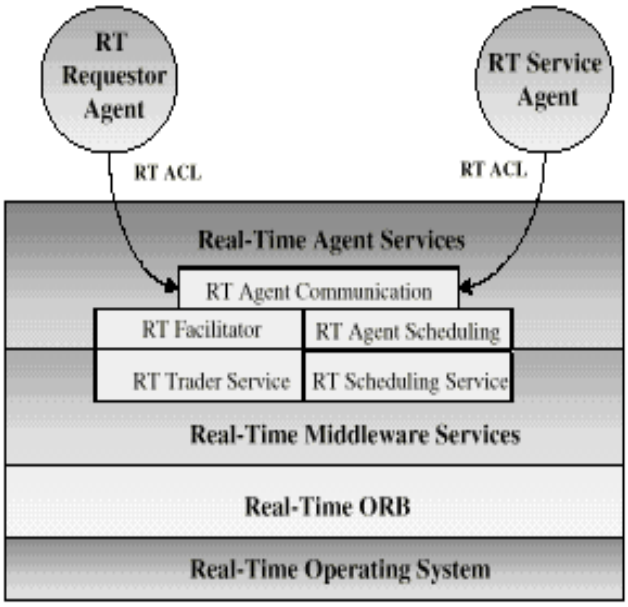


**Figure 3. Real-Time Multi-Agent System Architecture**

The initial design is based on the implementation of the KCobalt system [5] that maps KQML messages to CORBA IDL.

## 5.1 KCobalt Implementation

The KCobalt architecture was implemented using the JavaBeans technology [5]. JavaBeans are independent software components that can be assembled and parameterized statically at compile time or dynamically during Java program execution. Beans are concepts similar to object programming, but applied to high-level components offering advanced services. The implementation architecture based on JavaBeans is shown in Figure 4[12].

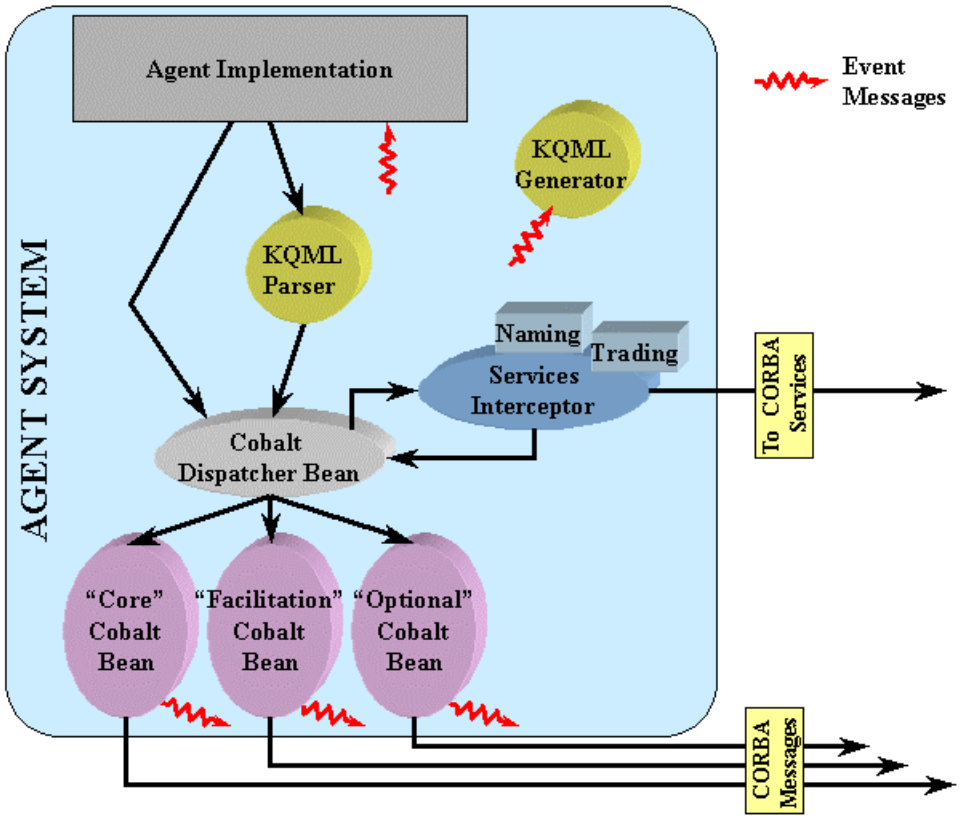


**Figure 4. KCobalt Implementation Architecture**

In the KCobalt implementation, agents are represented as CORBA objects. The IDL for each of the objects in cobalt.idl includes the following specification. Here we show only two performatives, `ask-one` and `tell`, but the interfaces for the other performatives are similar.

```
module Core {

interface CoreS {

//ask-one//
void askOne (  in  string  sender,
               in  string  receiver,
               in  string  inReplyTo,
```

```
                in  string  replyWith,
                in  string  language,
                in  string  ontology,
                in  string  content) ;
//tell //
void tell   (   in  string  sender,
                in  string  receiver,
                in  string  inReplyTo,
                in  string  replyWith,
                in  string  language,
                in  string  ontology,
                in  string  content) ;
. . .

  }
}
```

The interface for the agent object includes a CORBA method for each KQML performative, with parameters to represent the KQML parameter. There are three distinct interfaces, corresponding to the three performative categories: `Core`, `Facilitation` and `Optional`.

Thus the only mechanism, for communication with an agent object is through its KQML performative methods. Agent communication is expressed as a KQML string. That is, when an agent wishes to communicate with another agent object, it expresses a KQML string with the desired performatives. The string is sent to a `KQMLParser object` that parses the string and creates a `CobaltPerformative class` object to represent the specific performative being requested. The parser is based on a public domain tool initially developed by Sun Microsystems, JavaCC [10], functioning like Yacc and Lex and used to describe KQML grammar and to call Java functions matching each keyword. The parser then calls the `Dispatch(CobaltPerformative)` method of the `CobaltDispatcher` which determines to what agent the performative should be directed. The dispatcher object is responsible for calling the method on the agent object that corresponds to the performative in the message.

The interceptor module deals with performative interceptions, calling the right internal class when required. This provide agents KQML facilitator's capabilities. It

intercepts KQML messages and redirects them to the right CORBA service. The interceptor uses a configuration file `perf-intercept.map` defining :

- performatives to intercept,
- content of the `receiver` parameter required for performative interception,
- Java class to call in case of interception.

The interception module offer to agents the ability to register, locate and discover each other dynamically, during the multi-agents system operation.

## 5.2 Real-Time Agent Communication Implementation

Our current implementation is based on the KCobalt implementation described in Section 5.1. As is the case in KCobalt, all agents in our implementation are represented as objects. The IDL for an agent object in our implementation includes the following specifications :

```
interface CoreS {

//ask-one//
void askOne (  in  string  sender,
               in  string  receiver,
               in  string  inReplyTo,
               in  string  replyWith,
               in  string  language,
               in  string  ontology,
               in  string  content,
               in  string  Qos_info);
. . .

  }
```

The interface for an agent object is similar to the interface for agent objects in KCobalt. Each performative is represented as a method on the interface. The main difference is that our performative methods provide parameters for the expression of QoS constraints. The `Qos_info` parameter includes the priority and execution time allotted to the agent by the scheduler to execute this request.

Figure 5 displays the implementation design. To specify its capabilities, a servicing agent sends a RT KQML "advertise" string to the parser object, through its `parse()` method. The parser parses the message and creates a performative object to send to the dispatcher object. The dispatcher calls the `advertise()` method on the RT Agent Facilitation Service object to be integrated into the system . When a requesting agent requires a service from a servicing agent, the requesting agent sends a RT KQML string with the specified performative to the parser object (1). The parser parses the string and sends the associated performative object to the dispatcher object (2). For example, assume that a real-time agent specifies the following RT KQML string message:

```
(askOne
      :sender           UserAgent
      :receiver         TrendWatchingAgent
      :replyWith        Trend
      :Qos_requirement  (dl=15,imp=4,qual=75)
      :language         Java
      :ontology         IBM
      :content          Watch(internet)   )
```

The parser object parses this string and passes it to the dispatcher. The dispatcher extracts the specified agent, solvable and QoS information, and makes the following method call to the RT Agent Scheduling Service object (3):

```
schedule("TrendWatchingAgent", "Watch(internet)",15,4,75)
```

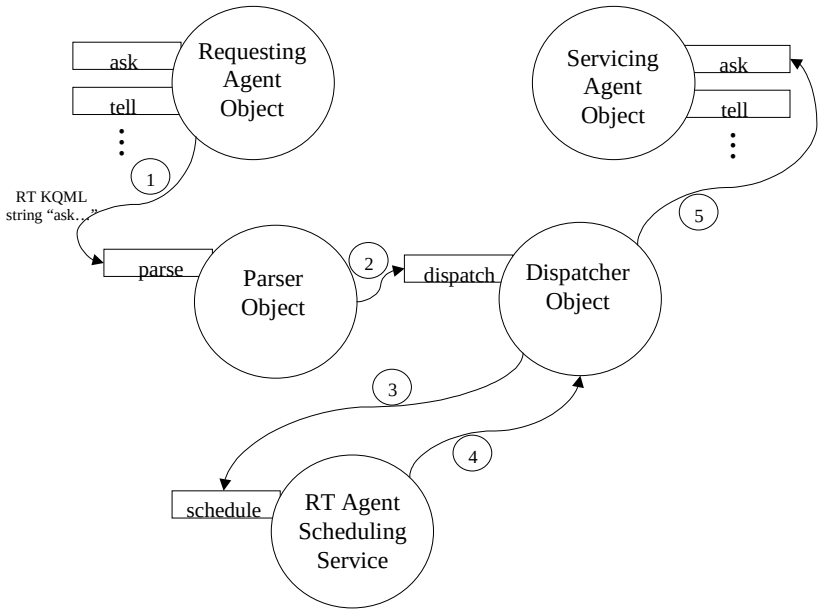


**Figure 5. RTMAS Prototype Implementation.**

For the implementation developed for this thesis the `schedule` function assigns priority and execution time values in the form of `Qos_Info` string for each agent request. A scheduling algorithm for a Real-Time Multi-Agent System has been developed by the Real-Time research group at URI which will soon be integrated into the system. For this implementation, the real-time parameters in `Qos_Info` are not based on any real-time scheduling algorithm. The scheduling service returns dummy values (4). Finally, the dispatcher calls the method on the servicing agent corresponding to the requested performative with the QoS parameters determined by the scheduling service (5).

In our implementation the interface for an agent object is similar to the interface for agent objects in KCobalt except that the performative methods represented on the interface were extended to include the Qos constraints. This makes it possible for agents to communicate using the performative methods defined on the agent interface in KCobalt.

## 5.3 Example Agent Implementation

To demonstrate the Real-Time Agent communication , I implemented in this thesis two example agents as shown in Figure 7. A *BuyStock* agent with two solvables `GetPrice(stock)` and `Buy(stock,num)` and a *UserAgent* with two solvables namely `TellPrice()` and `ConfirmSale()`.When the *BuyStock* agent gets a "`GetPrice`" request, it calls the method `GetPrice(searchStr)`. The `searchStr` is the stock name specified by the *UserAgent.*

The `GetPrice()` method of the *BuyStockAgent* object reads in a datafile containing records with fields stock name and prices. This method will search through a vector of strings holding raw input lines from the datafile, breaking each line into fields and comparing the stock name field against the searchStr passed to the method. If a match is found the entire line in the datafile entered for that stock, namely the stock name and its price, is returned. If no match is found then a string with the message "No match found" is returned. The *BuyStock_wrapper* object then sends a reply `TellPrice()` to the *UserAgent_wrapper* with the returned string from `GetPrice().`The *BuyStock_wrapper* and the *UserAgent* wrapper provide interface functions for the solvables of the *BuyStockAgent* and *UserAgent* respectively.

When the *UserAgent_wrapper* gets the stock price information it calls `TellPrice()` on the *UserAgent*. The *UserAgent* extracts the price from the content parameter value field and checks it. If the price is less than or equal to a particular limit value then it will send a message `Buy(stock,num)` to the *BuyStockAgent* to buy that number of stocks specified in the value `num` of the particular stock name specified by the argument `stock`. On getting the buy request the *BuyStock_wrapper* calls `Buy()` on the *BuyStockAgent* which sends back a `ConfirmedSale()` reply.

In order to test the extended performatives, the Performative Exchanger GUI was used. It acts as a simple graphical agent interface and allows a user to select the

performative to send, then to enter parameters depending on this performative. The test graphical interface is presented in Figure 6.

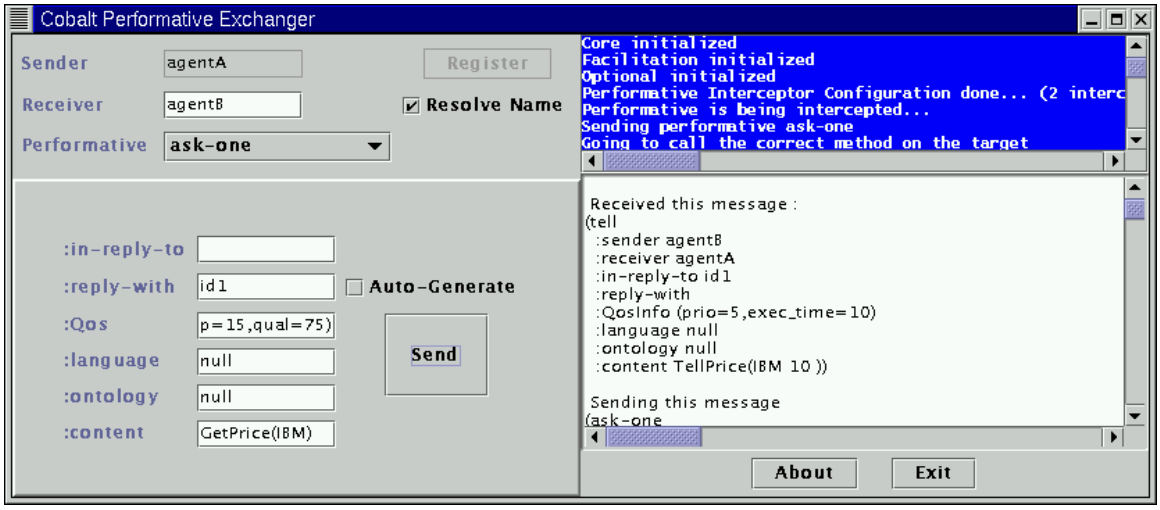


**Figure 6. Graphical Interface of the test application for KCobalt**

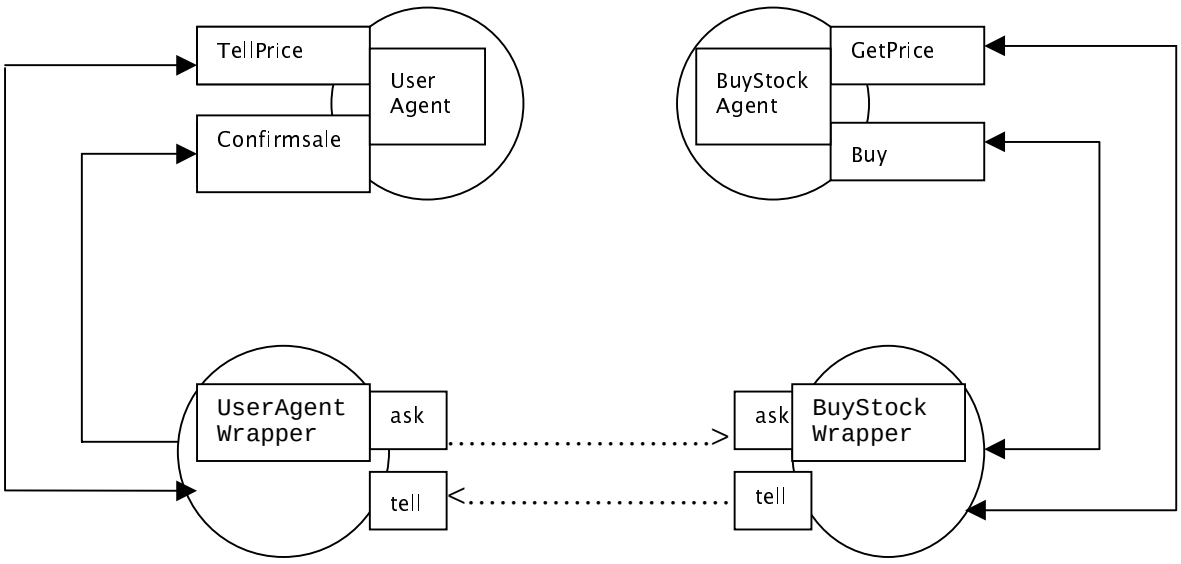


**Figure 7. Example Agents**

## 6.0 Conclusions

This chapter summarizes the completed work and discusses some key work to be done in the future.

## 6.1 Summary of Completed Work

Our RT Agent Communication Service extends the KQML agent communication language with the ability to express QoS parameters in each performative. This extension makes it possible for the supporting environments to analyze and deliver messages based on its Qos parameters, even though the `:content` itself is inaccessible. The design of these extensions is extensible in that new QoS features, such as security, network latency, and periodic execution of a task can easily be added in the future. We have used the KCobalt implementation since the model is very close to the model being developed by the Real Time Research group at URI. In order to extend the KCobalt implementation to allow for the expression of QoS constraints we extended the parser to recognize the additions that we have made to KQML, and we extended the implementation of agent objects to allow for QoS parameters in the performative methods. This provides agents with the ability to express the kinds of real-time and QoS constraints that occur in many real-world applications.

## 6.2 Future Work

Some suggestions for extending this work are listed below.

**1 Improve performance**

This implementation is not efficient for a real-time system because each time a RT KQML performative is expressed as a string, it must be parsed on the fly. While we could determine a worst case bound on the parsing time, this technique severely impedes real-time performance.

The performance of the system can be improved if new techniques similar to the KCobalt parsing techniques can be developed that do not require on-the-fly parsing of strings. The new technique can allow real-time agent programmers to implement their agent objects in exactly the same way as in the current implementation. It can employ a pre-processor that performs the role of the parser object and the dispatcher object. That is, the pre-processor will parse the KQML strings in the real-time agents, and builds the calls to the RT Agent Scheduling Service directly into the real-time agent objects. Thus, the final code for a real-time agent will not send a string to a parser, but rather, it will make a call to the `schedule()` method of the RT Agent Scheduling Service, bypassing the parser and the dispatcher altogether.

**2 Apply To FIPA ACL**

Syntactically both KQML and FIPA ACL messages are almost identical. The similar syntax guarantees that a developer will not have to alter the code that receives, parses and sends messages. The code that processes the primitives should change depending on whether the code observes the proper semantics. When compared to KQML, FIPA ACL is more powerful with composing new primitives. In FIPA ACL the “administration primitives” such as `register`, `unregister` etc., are treated as requests for action with reserved (natural language) meaning. There are no “facilitation primitives”, e.g., broker, recommend, recruit, etc., in FIPA ACL. The power stems from the power of the SL language as a content language to describe agents’ states. KQML’s weakness is its religious non-commitment to a content language.

**3 Integrate implementation of RT Scheduling Service and RT Facilitator**

This work should be integrated with a RT Scheduling Service which uses a real-time scheduling algorithm to schedule the various user requests coming into the system. Each agent in the system advertises its capabilities with the facilitator. A RT Facilitator provides the ability to request real-time agent services without specifying the exact agent that will perform the service. The RT Scheduling Service communicates with the facilitator in order to determine the best agent to provide a particular service using

alternate solution methods, or trading-off different resources, depending on what is available.

## References


[1] Nicholas R. Jennings, Katia Sycara, Michael Woolbridge. A Roadmap of Agent Research and Development. In *Autonomous Agents and Multi-Agent Systems*, 1, 275-306 (1998), Kluwer Academic Publishers, Boston.

[2] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML as an Agent Communication Language. In *The Proceedings of the Third International Conference on Information and Knowledge Management* (CIKM), ACM Press, November 1994.

[3] Y.Labrou and T.Finin, *A Proposal for a New KQML Specification*, Tech.Report TR-Cs-97-03, computer Science and Electrical Engineering Dept., Univ. of Maryland, Baltimore County, baltimore, Md., 1997.

[4] FIPA. FIPA 98 Specification. 1998.
`http://www.fipa.org/spec/fipa98.html`

[5] D.Benech, T.Desprats. A KQML-CORBA based Architecture for Intelligent Agents Communication in Cooperative Service and Network Management. In *Proceedings of IFIP/IEEE International Conference on Management of Multimedia Networks and Systems '97* July 1997.

[6] T.Labrou, T.Finin, and Yun Peng, Agent Communication Languages: The Current Landscape, University of Maryland, Baltimore County. ( IEEE Intelligent Systems, March/April 1999)

[7] Janet Jamie Prichard, RTSQL : Extending The SQL Standard to Support Real-Time Databases. University of Rhode Island Technical Report , July 1995.

[8] Lisa Cingiser DiPippo, Victor Fay-Wolfe, Lekshmi Nair, Ethan Hodys, Oleg Uvarov. A Real-Time Multi-Agent System Architecture for E-Commerce Applications, Submitted to ISADS( International Symposium on Autonomous Decentralized Systems) 2001.

[9] Tanenbaum, Andrew S. Distributed Operating Systems. New Jersey: Prentice-Hall, Inc., 1995.

[10] JavaCC, The Java Parser Generator- Metamata (initially developed by Sun Microsystems) – pages web : `http://www.metamata.com/JavaCC/`

[11] Communications Of The ACM , Intelligent Agents July 1994.

[12] Dominique Benech, Interaction Frameworks for Distributed and Cooperative Paradigms of Intelligent Systems and Networks Management. University Paul Sabatier, Toulouse, Technical Report, November 1999.

[13] Liu, C.L., and Layland, J.W.: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM, vol. 20, Jan. 1973.

[14] The Common Object Request Broker: Architecture and Specification – Object Management Group- Revision 2.2, February 1998.

[15] Thomas Wagner, and Victor Lesser, Design-to-Criteria Scheduling: Real-Time Agent Control. UMass Computer Science Technical Report 1999.

[16] I. Soto, M. Ramos, M. Garijo and C. Iglesias. An Agent Architecture to fulfill Real-Time Requirements. To appear in Fourth International Conference on Autonomous Agents, Agents 2000.

## Bibliography

Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML as an Agent Communication Language. In *The Proceedings of the Third International Conference on Information and Knowledge Management* (CIKM), ACM Press, November 1994.

D.Benech, T.Desprats. A KQML-CORBA based Architecture for Intelligent Agents Communication in Cooperative Service and Network Management. In *Proceedings of IFIP/IEEE International Conference on Management of Multimedia Networks and Systems '97* July 1997.

T.Labrou, T.Finin, and Yun Peng, Agent Communication Languages: The Current Landscape, University of Maryland, Baltimore County. ( IEEE Intelligent Systems, March/April 1999)

Janet Jamie Prichard, RTSQL : Extending The SQL Standard to Support Real-Time Databases. University of Rhode Island Technical Report , July 1995.

Lisa Cingiser DiPippo, Victor Fay-Wolfe, Lekshmi Nair, Ethan Hodys, Oleg Uvarov. A Real-Time Multi-Agent System Architecture for E-Commerce Applications, Submitted to ISADS( International Symposium on Autonomous Decentralized Systems) 2001.