A Real-Time Multi-Agent System Architecture for E-Commerce Applications[*]

Lisa Cingiser DiPippo, Victor Fay-Wolfe,
Lekshmi Nair, Ethan Hodys and Oleg Uvarov
The University of Rhode Island
Kingston, RI USA 02881
lastname@cs.uri.edu

Abstract

This paper describes an architecture for real-time multi-agent systems (RTMAS) that builds upon an existing real-time CORBA architecture. The RTMAS architecture provides real-time agent services for real-time agent communication, real-time agent scheduling and real-time agent facilitation. These services work together to allow for the expression and enforcement of real-time agent interactions. The paper describes the design of these services, along with a prototype implementation of the RTMAS architecture that is based upon an existing agent communication implementation.

1. Introduction

Electronic commerce systems have become increasingly pervasive in the business world. 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.

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]. 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 paper presents an architecture for RTMASs that provides for the expression and enforcement of QoS constraints among real-time agent communications. This architecture is based on a RTMAS model that allows agents to specify QoS capabilities and requirements on their communications. Our RTMAS architecture relies on an underlying real-time CORBA infrastructure to provide seamless distributed communication among agents. Using a CORBA framework to provide this functionality in a multi-agent system relieves the agent developer from providing low-level interagent communication. Our architecture builds on dynamic real-time CORBA middleware services to provide real-time communication, real-time scheduling and real-time facilitation services to the agents in the system.

The remainder of this paper is organized as follows: Section 2 provides background in the areas of agent communication, real-time agents, existing agent architectures, and the dynamic real-time CORBA

[*] This work is partially supported by the U.S. Office of Naval Research grant N00014-00-1-0060.

infrastructure that forms the foundation for our RTMAS architecture. Section 3 presents our model for RTMASs on which our architecture and real-time agent services are based. We describe an example electronic commerce application to illustrate the features of the model and the RTMAS architecture. Section 4 presents the RTMAS architecture and describes the layers and the services in detail. Section 5 describes our preliminary implementation of this architecture, along with some ideas for future improvements to the implementation. Finally, Section 6 concludes with a summary of the contributions of this work and a discussion of future work.

2. Background

In this section we present some key areas that form the background for our RTMAS development. We present work in agent communication and facilitation that defines the languages and mechanisms that agents utilize to coordinate to meet their goals. We go on to describe previous work in the area of real-time agents. This work has developed some of the defining characteristics of real-time agents that we have built into our model. We also discuss existing agent architectures and how they relate to our RTMAS architecture. Finally, we describe the dynamic real-time CORBA architecture on which our RTMAS architecture is built. This underlying architecture provides the means for enforcement of timing constraints in the scheduling and communication among CORBA objects.

2.1. Agent Communication and Facilitation

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.

Communication among agents and facilitators is typically achieved through an agent communication language, such as the Knowledge Query Manipulation Language (KQML) [2]. KQML provides *performatives* to define the kind of interactions a KQML-speaking agent can have. A KQML message consists of three layers: the content layer, the message layer and the communication layer. The content layer bears the actual content of the message in the agent's own representation language. The communication layer encodes a set of message features, which describe the lower-level communication parameters, such as the identity of the sender and recipient, and a unique identifier associated with the communication. The message layer is used to encode a message that one application would like to transmit to another. The message layer forms the core of the KQML language, and determines the kinds of interactions one can have with a KQML-speaking agent. The performatives of a KQML message include those to request that an agent perform a task (`ask-one`), to provide other agents with certain information (`tell`), to watch another agent for a particular condition (`monitor`), and to register capabilities with another agent (`advertise`).

The Foundation for Intelligent Physical Agents (FIPA) [3] is developing an agent communication language known as FIPA-ACL. This language is very similar to KQML, with some differences in its semantics. We have chosen to use KQML in our architecture because it is more well-developed. However, most of the work that we have done in real-time agent communication can easily be applied to FIPA-ACL as well.

2.2. Existing Agent Architectures

Many agent architectures have been developed to support multi-agent systems. Here we highlight a few and describe how they relate to our work. The DECAF (Distributed, Environment-Centered Agent Framework) Agent Framework [4] is a Java-based multi-agent system. It provides a matchmaker agent that accepts KQML performatives to allow for agent communcation. SRI International has developed an agent architecture called Open Agent Architecture (OAA) [5]. Agent interaction in OAA is done through an agent facilitator using a language called Interagent Communication Language (ICL). Both of these architectures have been developed from the ground up. That is, they each provide their own underlying communication mechanism. Our work relies on a CORBA infrastructure to provide this.

Several agent architectures have been developed using CORBA [6] as the underlying communication framework. Broadcom has developed the Agent Services Layer (ASL) as a layer on top of a CORBA framework [7]. The ASL provides services specific to agent interaction and coordination, and uses CORBA as the underlying distributed communication middleware. KCobalt [8] is an agent framework based on KQML and CORBA. The implementation of KQML on CORBA provides a complete communication layer that can support cooperation in multi-agent systems. The system provides a mapping from the agent language KQML to the CORBA interface definition language (IDL). We have chosen to expand on the KCobalt implementation to build the real-time agent communication module of our system because the KQML to IDL mapping is straightforward and easily extensible with real-time features.

There has been recent work towards developing mechanisms to support real-time agents. Several architectures have been proposed for real-time agents, as well as research in scheduling agent tasks within the architecture. Much of the real-time agent scheduling work relies on the assumption that in order to perform a task, an agent or set of agents may have multiple ways of solving the same problem, each with varying time required to compute the result, and varying quality of the result produced. Typically, the more time available to solve the problem, the higher the quality of the result. This becomes very useful in real-time agent scheduling because it allows for a trade-off between the quality of the result and the amount of time required in order to meet specified timing constraints.

In [9], a design-to-time scheduling algorithm for incremental decision-making is described. It presents a model that provides a hierarchical abstraction of the problem solving processes which describe alternative methods for solving a specific goal depending upon the available amount of time. The allocation of resources can be dynamically adjusted in order to meet system-wide timing constraints. Design-to-criteria scheduling [10] is an extension of the design-to-time algorithms. The model here is more general in that it can take into account any scheduling criteria, such as time, cost, and quality, and it can use uncertainty as part of the decision-making process. The DECAF architecture described above [4] is incorporating scheduling algorithms based on the design-to-criteria model. ASTRO [11] is a model for real-time agents in which each goal is divided into subgoals that each have a deadline. The scheduling algorithm used by the ASTRO schedules all subgoals with their minimal duration. It then increases the duration of the highest priority subgoal to result in maximal satisfaction (highest quality result for the subgoal). Soto et. al. [12] have developed the AMSIA agent architecture which provides a representation of plans allowing different reasoning activities to create plans that guide the future behavior of an agent. When deciding among sequences of tasks to schedule, the control mechanism scores them based on importance, deadline and the quality offered by the tasks

2.3. Dynamic Real-Time CORBA

Our RTMAS architecture relies on a dynamic real-time CORBA architecture that we are developing in a concurrent project. This section describes the ongoing standardization of real-time CORBA by the OMG, and our current dynamic real-time CORBA implementation.

2.3.1. OMG Standard. The Object Management Group (OMG) has recently begun the specification of a standard for real-time distributed object management. In 1999 a specification for Realtime CORBA for static scheduling was published [13]. This specification defines the required features that a CORBA implementation must have to support hard real-time applications in which timing constraints must be met. These features include priority, bounding priority inversion and protocol selection. The specification also includes an optional scheduling service that uses the primitives of the RT ORB to achieve a uniform scheduling policy in the CORBA system. Currently the OMG is working on a specification for Dynamic Realtime CORBA. This specification will be an extension of the static specification to support dynamic scheduling, where clients and servers dynamically enter and exit the system and priorities may change over time. The Dynamic RT CORBA standard will support the specification of *policy* and *parameters* to be used to schedule each task. That is, instead of each task carrying with it a CORBA Priority, as is required in the current RT CORBA standard, tasks will carry a richer description that includes parameters, such as deadline, importance, delay, and period.

2.3.2. Our Dynamic Real-Time CORBA Architecture. We have been involved in the development of an implementation of a dynamic real-time CORBA system that is based on some of the ideas that have come from the standardization process of the OMG [14]. This implementation relies on a real-time ORB based on the developing dynamic standard, and it focuses on the development of services that support dynamic real-time requirements. These services include a Scheduling Service and a real-time (RT) Trader Service. These two services work together to ensure that client requests are scheduled to meet their deadlines.

Scheduling Service. The Scheduling Service assigns priority to servers and to client requests using an earliest deadline first (EDF) priority assignment policy. It employs a load shedding heuristic when all requested tasks cannot be scheduled to meet deadlines. That is, whenever a new request enters the system, the scheduling service performs an EDF schedulability analysis to determine if it can be scheduled. If it cannot, the load shedding heuristic determines, based on a weighted sum of importance and remaining execution time, which currently scheduled task(s) can be shed in order to maintain schedulability of the system [14]. When a task is shed, the Scheduling Service reduces the priority of the task below a specified threshold and raises an exception to indicate to the client that requested the task that it has been shed.

RT Trader Service. In a CORBA system, the Trader Service assigns bindings to server objects based on the requirements of a client's request. Our RT Trader service extends this capability by allowing clients to specify timing parameters, such as deadline and importance. When the RT Trader Service receives a request for service from a client, it determines which server object can respond to the request within the specified timing constraints. Further, the RT Trader Service employs a probabilistic algorithm to find the server object that can meet the current request and the next likely request. See [15] for more details on this algorithm.

Figure 1 displays how our dynamic real-time CORBA implementation works. Any server that provides a service must first register its capabilities (including execution times) with the RT Trader Service (1). The Scheduling Service receives a request from a client to execute a method on a server (2). The request includes a deadline and an importance parameter. The Scheduling Service and the RT Trader Service work together to schedule the request (3). First the RT Trader Service attempts to find a server object that can best handle the request. If it cannot find any servers on which the task can be scheduled, then the Scheduling Service performs its load shedding heuristic on the server suggested by the RT Trader Service as most likely to allow the request. If a task is shed, an exception is raised to the corresponding server object. This is not depicted in Figure 1, which shows the process of a specific request that has not been shed. If the requested task is found to be schedulable, the Scheduling Service responds to the client with a priority at which to execute the method call (4). Finally, the client calls the method on the server (5) with the specified deadline and priority.

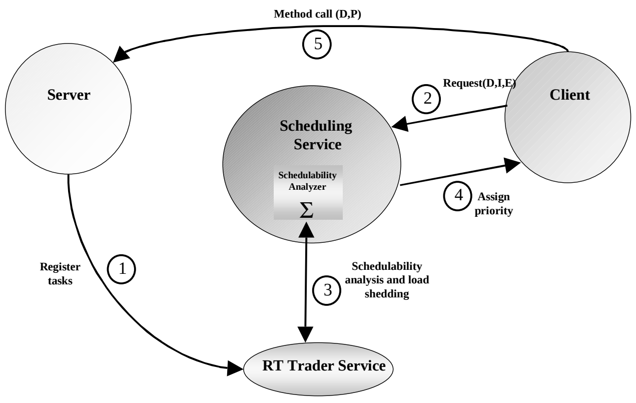

Figure 1 - Dynamic Real-Time CORBA

3. Real-Time Multi-Agent System Model

In this section we describe our model of RTMASs on which our RTMAS architecture and its real-time agent services are based. We first present an example electronic commerce application in which a RTMAS would be useful, and then we present the elements of the model, using the example to illustrate important points.

3.1. Example Stock Trading RTMAS

Here we present an example RTMAS that allows multiple real-time agents to coordinate to make intelligent recommendations, purchases and sales of stocks. Each of the agents involved employs some degree of intelligence to perform its tasks. However, for the purposes of this paper, we will not be concerned with exactly how the agents perform their tasks, but rather, how they are designed so that they can work together to meet their goals, and their specified QoS requirements.

We will discuss 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.

A *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 of interest to another agents, it notifies them. The *TrendWatchingAgent* communicates with a *UserAgent* if the *UserAgent* has expressed interest in a particular trend. It communicates with the *QuotingAgents* 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 the recommendation to the user, get her approval, and then notify the *BuySellAgent* to perform the 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 benefit

3.2. RTMAS Model

Our RTMAS model 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. This assumption follows from much of the previous work discussed in Section 2 [9,10].

Our RTMAS model made up of a set of real-time agents (*RTAgent*) and a set of communications among the real-time agents (*Request*).

RTAgent. A real-time agent can be defined as follows:

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

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 this problem. For instance, in the stock trading example, 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:

$$ES = \{es_1, es_2, \ldots, es_f\}$$

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. The level of quality, *q*, is a rating of the result of an execution strategy. Quality is calculated as a percentage of the optimal result such that *q = (strategy result / optimal result).* This definition for 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:

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

The tradeoff value for the execution strategy with the shortest execution time (es_f) is undefined. This is because reducing from the shortest execution strategy amounts to shedding the task altogether.

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 our model, this deadline can be either a soft deadline, or a firm deadline, depending upon the requirements of the application. Finally, *H* specifies the quality threshold for the request. That is, the requesting agent expresses through *H*, the minimum quality required 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 Intel stock (*GetPrice*). 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 may be low. But if it involves thousands of dollars, the importance may be higher.

4. Real-Time Multi-Agent System Architecture

Our RTMAS architecture is a multi-layered architecture as depicted in Figure 2. At the lowest layer, we assume a POSIX-compliant real-time operating system [16]. Above that, the real-time ORB layer consists of the required features of the developing dynamic real-time CORBA standard. Since this standard is not yet completely defined, we rely on the real-time ORB standard of the current static Realtime CORBA standard. The real-time middleware services layer consists of the Scheduling Service and the RT Trader Service described in Section 2.3. 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 Facilator Service. The agent services layer also provides a service for Real-Time Agent Communication.

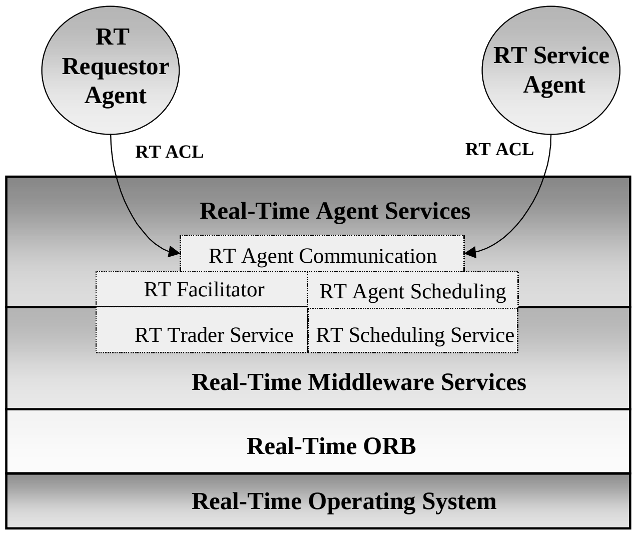

Figure 2 - RTMAS Architecture

The key to our architecture lies in building real-time agent services on top of existing services at the RT CORBA layer. While other agent architectures have been built upon CORBA [6], our work is unique in its reliance on RT CORBA and RT CORBA services as a foundation for the RTMAS architecture. In this section we will describe each of the services that we have developed. We will then describe a preliminary prototype implementation that we have designed based on our RTMAS architecture.

4.1. Real-Time Agent Communication Service

In order to express timing constraints in a RTMAS, we have extended the expressibility of the agent communication language KQML. There are two types of communication that we have currently 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. We have extended the language to allow for expression of QoS requirements and capabilities through new performative parameters. We have chosen not to include this information as part of the content of the KQML performative because expression of Qos is independent of content, and should be treated as part of the communication between agents, and not as part of the specific message being sent to the receiving agent. Also, the QoS information needs to be used by the scheduling mechanism, and would not be available if it were embedded in the content.

Agent Request. In our RTMAS model described in Section 3, there are three kinds of constraints that are expressed: deadline, importance, and quality. We extend the KQML performatives to include these constraints to produce a real-time KQML (RT KQML). For example, consider a request from the *UserAgent* to a *TrendWatchingAgent*, to report on current trends in internet stocks within 15 seconds. The KQML request may look like the following:

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

The `QoS_requirement` parameter (highlighted in bold) 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. We have included all of these constraints as part of a single `QoS_requirement` parameter in order to allow for the addition of further quality of service constraints in the future.

The *TrendWatchingAgent* will respond to this request with a message like the following:

```
(tell
  :sender           TrendWatchingAgent
  :content          +35%
  :receiver         UserAgent
  :in-reply-to      Trend
  :QoS_requirement  (dl 5, imp 4, qual 75)
  :language         Java
  :ontology         Stock)
```

The `tell` message also expresses a `QoS_requirement` parameter. This is because agent communication is asynchronous, and therefore all messages must be sent explicitly. This message must be scheduled in order to meet the requirements specified in the `ask-one` request. The QoS parameters may be derived from the parameters specified in the original message. For example, the `tell` message above has a deadline of 5 seconds, indicating that there are only 5 seconds left to meet the 15 second deadline of the original `ask-one` message. All KQML performatives may express QoS constraints in the form of the `QoS_requirement` parameter, so that they can be scheduled to meet their constraints. For a full description of the RT KQML extension, see [17].

Facilitator Advertisement. Communication between agents and facilitators must also be extended to allow for expression of timing capabilities. The facilitator and the scheduler are responsible for determining which execution strategies of which agents will provide a solution to a requesting agent. Therefore, 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
  :ontology      Stock
  :language      Java
  :content       BuyStock(A)
  :QoS_capabilities(
             (ex 5, qual 85)
             (ex 2, qual 65)))
```

In this example, the *BuySellAgent* specifies through the `QoS_capabilities` parameter that it has two execution strategies, one that 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. Again, we use a single parameter, `QoS_capabilities`, to express the quality of service characteristics so that it can be easily extended.

4.2. Real-Time Agent Scheduling Service

The scheduling of agent requests is the key to enforcing expressed timing and other QoS constraints. Our scheduling model relies on the fact that real-time agents can have multiple execution strategies that provide varying levels of quality given varying amounts of time to execute. Our real-time agent scheduling algorithm is similar to the algorithm described in Section 2.3 for dynamic real-time CORBA. It uses earliest deadline first (EDF) scheduling of agent tasks, but rather than load shedding, it relies on a load reduction heuristic that determines which currently scheduled tasks to reduce in quality and time, in order to maintain schedulability of the system.

Scheduling Algorithm. Our real-time agent scheduling algorithm processes requests for service between agents. It performs schedulability analysis, and reduces the execution time of one or more agent execution strategies if necessary to maintain schedulability.

To illustrate how the algorithm works, assume that there is currently a set of scheduled tasks. A requesting agent makes a request for a solvable in a servicing agent. The request is accepted by the RT Agent Scheduling Service and the task is created and added to the list of tasks to be scheduled. The execution time for the task initially defaults to the execution time of the execution strategy with the highest quality.

The Scheduling Service performs an EDF schedulability analysis on all tasks including the new request. If all the tasks are found to be schedulable, then the requesting agent is given permission to make the request to the servicing agent. If the tasks are not schedulable, then the schedulability analysis returns a set of critical points representing the tasks that miss their deadlines. This critical point information is provided as input to the load reduction heuristic that will identify tasks for load reduction to yield a feasible schedule.

Load Reduction Heuristic. For each critical point, there is a set of tasks with shorter or equal deadline that are candidates for reduction. That is, reducing the execution time of these candidate tasks may allow the task at the critical point to become schedulable. For each critical point, the load reduction heuristic sorts all candidate tasks by cost. It reduces the task with the lowest cost and then determines if any more reduction is necessary. That is, if the difference between the amount of time required at this critical point, and the amount of time gained through this reduction is greater than zero, then further reduction is required for this critical point.

The key to this heuristic is in how we calculate cost. Our current implementation uses the tradeoff value of the current execution strategy multiplied by the importance of the request to determine the cost. Thus, the more important the request, and the higher the tradeoff value, the less likely the task will be reduced. However, other factors could be used in calculating cost. For example, we could use parameters, such as remaining execution time or time gained by performing a reduction in the computation. Further, we could weight each of these parameters in varying ways to get different results. Details of this scheduling algorithm, and results of performance tests can be found in [18].

4.3. Real-Time Agent Facilitation Service

Agent facilitation provides a mechanism for agents to find out about each other's capabilities. In a RTMAS, this includes timing and other QoS capabilities such as number of execution strategies, and the execution time and quality returned by each of the execution strategies. In our RTMAS architecture, there is a tight coupling between the RT Agent Facilitation Service and the RT Agent Scheduling Service because the facilitator knows about the agent capabilities, which the scheduler needs to use in order to make scheduling decisions.

Our RT Agent Facilitation Service implements a matchmaker type of facilitator. However, each request for service must go through the RT Agent Scheduling Service which may need to communicate with the facilitator to assign priority. The facilitator also provides other information to aid in the determination of which agents to schedule. When a request for a solvable on a servicing agent is made, there may be more than one agent that can provide the required service. For example, in the stock trading system described above, there are likely multiple agents that can monitor stock prices. The RT Agent Facilitation Service provides mechanisms to help determine which of these agents would provide the best solution. Of course, if only one of the servicing agents would provide a schedulable solution, then the facilitator would choose that agent. However, if more than one of the agents could provide a feasible solution, the facilitator could use its knowledge of the system to find the best agent to solve the problem. For instance, if enough statistical information is available about usage of the agents in the system, the facilitator uses an algorithm similar to the RT Trader Service algorithms described in

Section 2.3 to determine which of the candidate agents could service the current request as well as certain likely future requests.

The specifics of the real-time facilitation algorithms are not complete at the time of this writing. We are currently considering other properties that the real-time agent facilitator might use to make the selection of a servicing agent more intelligent, such as quality of service properties like available bandwidth, security, and fault tolerance.

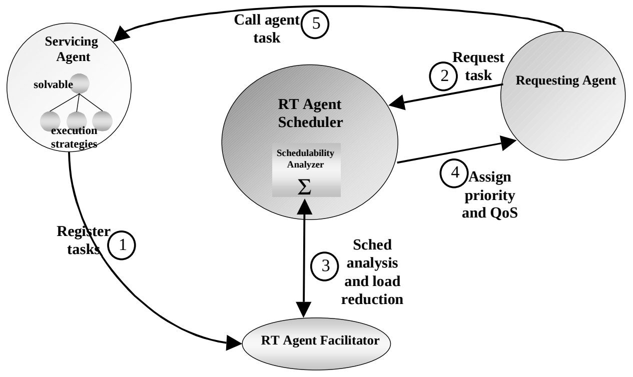

Figure 3 - RT Agent Scheduling and Facilitation

Figure 3 illustrates how the RT Agent Facilitation Service and the RT Agent Scheduling Service work together, with the scheduler using information available in the facilitator, such as timing capabilities and current load, to make scheduling decisions. Note the similarities with the dynamic real-time CORBA system described in Section 2.3. The Servicing Agent first registers its capabilities with the RT Agent Facilitation Service (1). When a Requesting Agent makes a request (2), the RT Agent Scheduling Service communicates with the RT Agent Facilitation Service to determine if any registered agents can respond to the request, within the specified QoS constraints. If not, the RT Agent Scheduling Service performs load reduction using the servicing agent that the facilitator determined to be most likely to be able to handle the request (3). The load reduction is implemented through exceptions to the agents involved, and is not depticted in Figure 3. The RT Agent Scheduling Service returns to the Requesting Agent the priority assigned to the request, and any other necessary QoS parameters, such as the execution time of the chosen execution strategy (4). Finally, the Requesting Agent makes the request to the Servicing Agent (5).

5. RTMAS Prototype Implementation

This section describes the implementation of a prototype RTMAS based on the architecture that we have described in Section 4. The implementation is preliminary. Some parts are complete while other parts are still in early phases of implementation. The initial design is based on the implementation of the KCobalt system [8] that maps KQML messages to CORBA IDL. In this section we give a brief description of the Kcobalt system. We then go on to describe our current implementation. And we finish with a discussion of how we will enhance the implementation for improved real-time performance.

5.1. KCobalt Implementation

In the implementation of the KCobalt system [8], agents are represented as CORBA objects. The IDL for each of the objects includes the following specification. Here we show only two performatives, but the IDL for the other performatives is similar.

```
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 method for each KQML performative, with parameters to represent each KQML parameter. 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 object wishes to communicate with another agent object, it expresses a KQML string with the desired performatives. The string is sent to a *parser object* that parses the string and creates a *performative class object* to represent the specific performative being requested . The parser then sends the performative object to a *dispatcher object* 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. For details about this implementation, see [8].

5.2. Real-Time Agent Communication Implementation

Our current implementation is based on the KCobalt implementation described above. 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:

```
struct QoSInfo {
    int priority;
    int exec_time;
}
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 QoSInfo 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 expression of QoS constraints, in the form of the `QoSInfo` structure. When a real-time agent object expresses a RT KQML string, the dispatcher object parses the string and determines the performative and its parameters. It then calls a method (`schedule()`) on the RT Agent Scheduling Service object to determine the schedulability of the request. For example, assume that a real-time agent specifies the following RT KQML string message:

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

The dispatcher object parses this string and produces the following method call to the RT Agent Scheduling Service object:

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

The scheduling service returns a priority and the execution time allotted to the agent to execute this request. With this information, the dispatcher calls the method on the servicing agent that corresponds to the requested performative. In the above example, if the RT Agent Scheduling Service returns with a priority of 20 and an execution time of 10, the dispatcher calls the ask-one method on the TrendWatchingAgent as follows:

```
ask-one("UserAgent",
         "TrendWatchingAgent",
        "","Trend","Java","Stock",
        "Watch(internet)",qos_Info);
```

where `qos_Info` contains the priority and execution time information provided by the RT Agent Scheduling Service.

5.3. Real-Time Agent Scheduling Service Implementation

The RT Agent Scheduling Service object has a single method on its interface.

```
QoSInfo schedule(
    RTAgent ServicingAgent,
    Solvable RequestedSolvable,
    int deadline, int importance,
    int quality)
```

The parameters for this method include the servicing agent on which the request is being made, the solvable being requested from the servicing agent, and the QoS parameters for the request (deadline, importance and quality). In the example above, the servicing agent is *TrendWatchingAgent*, the solvable is *Watch(internet)*, and the QoS parameters are 15, 4 and 75. Notice that the specified performative is not included as a parameter in this method. This is because the RT Agent Scheduling Service is not concerned with what performative is being used, only with the solvable on the agent that is being requested. The dispatcher object is responsible for determining which solvable is being specified by the RT KQML performative.

The return type of the `schedule()` method is `QoSInfo`, as specified in the IDL shown above. The scheduling service takes the QoS information specified in the method, communicates with the RT Agent Facilitation Service, and performs its scheduling algorithm and load reduction heuristic if necessary. The `schedule()` method returns to the dispatcher the priority at which the requested solvable should execute, and the amount of time that the servicing agent has been allotted to execute the solvable in the form of the `QoSInfo` structure.

5.4. Real-Time Agent Facilitation Service Implementation

The implementation of the RT Agent Facilitation Service is not yet complete because we are still working on the details of some of the algorithms. The implementation will have features of both a real-time agent object, and the RT Trader Service described in Section 2.3. On the interface of the RT Agent Facilitation Service will be a method for each RT KQML performative, like all other RT agents. These performatives will include special facilitator performatives such as `broker()` and `recruit()`. These will allow the RT Agent Scheduling Service to communicate with the facilitator in order to determine the best agent to provide a particular service. The implementation of these methods will include the algorithms that we are developing based on the RT Trader Service work.

Currently, our implementation does not employ the RT Agent Facilitation Service. Instead, the RT Agent Scheduling Service performs schedulability analysis on the agent specified in its `schedule()` method. However, once the facilitation service is complete, it will easily fit into the system implementation with minor changes.

5.5. Implementation

Figure 4 displays our implementation design. To specify its capabilities, a servicing agent sends a RT KQML "advertise" string to the parser object, through its `parse()` method (1). The parser parses the message and creates a performative object to send to the dispatcher object (2). The dispatcher calls the `advertise()` method on the RT Agent Facilitation Service object (3). 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 (4). The parser parses the string and sends the associated performative object to the dispatcher object (5). The dispatcher extracts the specified agent, solvable and QoS information, and calls the RT Agent Scheduling Service `schedule()` method with these parameters (6). The scheduling service communicates with the RT Agent Facilitation Service through its `recruit()` method to determine which servicing agent to consider in the schedulability analysis (7). After the RT Agent Scheduling Service performs the scheduling algorithm and determines the priority and execution time for the required solvable, it returns this information to the dispatcher object (8). Finally, the dispatcher calls the method on the servicing agent corresponding to the requested performative with the QoS parameters determined by the scheduling service (9).

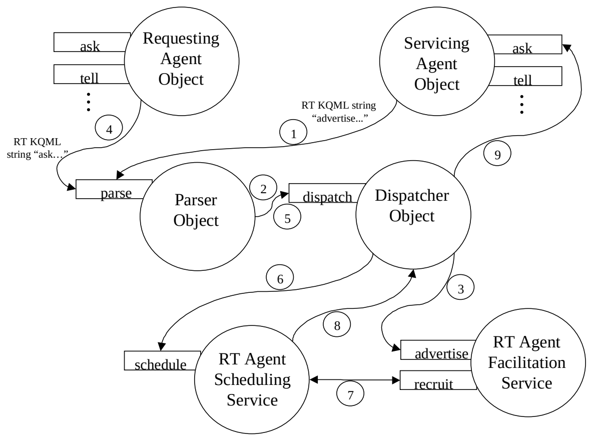

Figure 4 – RTMAS Prototype Implementation

5.6. Future Implementation Improvements

Our initial implementation is a prototype system to demonstrate the utility of our RTMAS architecture. We have built it upon the existing KCobalt implementation for several reasons. First, the model is very close to our model of building the agent layer on top of the CORBA layer. Second, it was relatively easy to extend the KCobalt implementation to allow for the expression of QoS constraints. It was a matter of revising the parser to recognize the additions that we have made to KQML, and extending the implementation of agent objects to allow for QoS parameters in the performative methods. However, 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.

We are currently working on developing techniques that are similar to the KCobalt parsing techniques, but that do not require on-the-fly parsing of strings. This new technique will allow real-time agent programmers to implement their agent objects in exactly the same way as in the current implementation. It will 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 specified in the real-time agents, and build 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.

6. Conclusion

In this paper we have presented a RTMAS architecture that builds on the strengths of an existing real-time CORBA architecture, and adds a layer of real-time agent services. These agent services work together to provide support for the expression and enforcement of QoS constraints in the RTMAS, while relying on the RT CORBA services and ORB layers to provide the underlying communication support.

Our RT Agent Communication Service extends the KQML agent communication language with the ability to express QoS parameters in each performative. The design of these extensions is extensible in that new QoS features, such as security and network latency, can easily be added in the future. The RT Agent Scheduling Service builds on the scheduling mechanism that we have developed for real-time CORBA. It uses the same EDF scheduler and schedulability analysis, but it employs a load reduction heuristic when overload conditions exist. This heuristic capitalizes on the ability of our real-time agents to perform a task in more that one way, with varying execution times and levels of quality. While the design and implementation of our RT Agent Facilitation Service are not complete, its placement in the RTMAS architecture is sound. It provides the ability to request real-time agent services without specifying the exact agent that will perform the service.

Our prototype implementation of the RTMAS architecture correctly implements all of the services that we have designed. It provides a mechanism to implement real-time agents as CORBA objects. We are developing updates to this implementation to improve its real-time performance.

The RTMAS architecture described in this paper is a general architecture for real-time agent applications. It will provide a platform upon which to develop real-time e-commerce agent applications such as the stock trading system described throughout the paper. It provides agents with the ability to express and enforce the kinds of real-time and QoS constraints that occur in many real-world applications. It also provides a platform for experimenting with new techniques in real-time agent development that will further the advancement of many applications in the realm of electronic commerce.

References

[1] Nicholas R. Jennings, Katia Sycara, Michael Wooldbridge. 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, Richard Fritzson, Don McKay, 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] FIPA. *FIPA 98 Specification*. 1998. http://www.fipa.org/spec/fipa98.html.

[4] John Graham and Keith Decker. Towards a Distributed, Environment-Centered Agent Framework. In *Proceedings of the 1999 Intl. Workshop on Agent Theories, Architectures, and Languages [ATAL-99]*, Orlando, July 1999.

[5] David L. Martin, Adam J. Cheyer, Douglas B. Moran. The Open Agent Architecture: A Framework for Building Distributed Software System. *Applied Artificial Intelligence*. vol. 13, pp. 91-128. Jan-Mar 1999.

[6] OMG. *Common Object Request Broker Architecture – Version 2.2*. OMG, Inc., 1998.

[7] Fergal Somers, Richard Evans, David Kerr. Scalable Low-Latency Network Management Using Intelligent Agents. *Communicate: Broadcom's Technical Journal*. vol. 3, issue 2.

[8] 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 Services '97* July 8-10, 1997.

[9] Alan Garvey, Victor Lesser. Design-to-time Real-Time Scheduling. IEEE Transactions on Systems, Man and Cybernetics – Special Issue on Planning, Scheduling and Control. vol. 23, no. 6, 1993.

[10] Thomas Wagner and Victor Lessor. Design-to-Criteria Scheduling: Real-Time Agent Control. *Proceedings of the 2000 AAAI Spring Symposium on Real-Time Systems*.

[11] Michel Occello, Yves Demazeau and Christof Baeijs. Designing Organized Agents for Cooperation with Real-Time Constraints. *First International Collective Robotics Workshop* (CRW'98), July 4-5, 1998, Paris, *Collective Robotics*, pp. 25-37, Springer-Verlag, 1998.

[12] Ignatio Soto, Mercedes Garijo, Carlos A. Iglesias, Manuel Ramos. An Agent Architecture to fulfill Real-Time Requirements. In *Proceedings of the Fourth International Conference on Autonomous Agents*, June 2000.

[13] OMG. *Realtime CORBA*. Electronic document at http://www.omg.org/docs/orbos/98-10-05.pdf.

[14] Oleg Uvarov. *Dynamic Real-Time Scheduling and Load Shedding for QoS Middleware*, University of Rhode Island – Department of Computer Science thesis proposal.

[15] Steven Wohlever, Victor Fay-Wolfe, Bhavani Thuraisingham, R. Freedman, John Maurer. CORBA-based Real-time Trader Service for Adaptable Command and Control Systems. In *Proceedings of the Second IEEE International Symposium on Object-oriented Real-time Distributed Computing* (ISORC 99). May 1999.

[16] IEEE, *IEEE Standard Portable Operating System Interface for Computer Environments (POSIX) 1003.1*, IEEE, New York, 1990.

[17] Lekshmi S. Nair. *RT KQML – A Real-Time Agent Communication Language*, University of Rhode Island Technical Report – Department of Computer Science thesis proposal.

[18] Ethan Hodys. *A Scheduling Algorithm for a Real-Time Multi-Agent System*, University of Rhode Island Technical Report – TR00-275.