1

BEEHIVE: GLOBAL MULTIMEDIA DATABASE SUPPORT FOR DEPENDABLE, REAL-TIME APPLICATIONS

John Stankovic,
Sang Son, and Jörg Liebeherr

Department of Computer Science

University of Virginia

Charlottesville, VA 22903

1 INTRODUCTION

The Next Generation Internet (NGI) will provide an order of magnitude improvement in the computer/communication infrastructure. What is needed is a corresponding order of magnitude improvement at the application level. One way to achieve this improvement is through global virtual databases. Such databases will be enterprise specific and offer features along real-time, fault tolerance, quality of service for audio and video, and security dimensions. Support of all these features and tradeoffs between them will provide an order of magnitude improvement in performance and functionality over browsers, browsers connected to databases, and, in general, today's distributed databases. Such global virtual databases will not *only* be enterprise specific, but also interact (given proper protections) with the worldwide information base via wrappers. Such wrappers may be based on Java and Java Data Base Connectivity standards.

There are many research problems that must be solved to support global, real-time virtual databases. Solutions to these problems are needed both in terms of a distributed environment at the database level as well as real-time resource management below the database level. Included is the need to provide end-to-end guarantees to a diverse set of real-time and non-real-time applications over the current and next generation Internet. The collection of software services that support this vision is called BeeHive.

The BeeHive system that is currently being defined has many innovative components, including:

- real-time database support based on a new notion of *data deadlines*, (rather than just transaction deadlines),
- parallel and real-time recovery based on semantics of data and system operational mode (e.g., crisis mode),
- use of reflective information and a specification language to support adaptive fault tolerance, real-time performance and security,
- the idea of security rules embedded into objects together with the ability for these rules to utilize profiles of various types,
- composable fault tolerant objects that synergistically operate with the transaction properties of databases and with real-time logging and recovery,
- a new architecture and model of interaction between multimedia and transaction processing,
- a uniform task model for simultaneously supporting hard real-time control tasks and end-to-end multimedia processing, and
- new real-time QoS scheduling, resource management and renegotiation algorithms.

The BeeHive project builds upon these results and combines them into a novel design for a global virtual database.

In this chapter we discuss the high-level BeeHive system and sketch the design of a native BeeHive site showing how all the parts fit together. We also present technical details on two of the main functional ingredients of BeeHive which include Resource Management and QoS, Real-Time Databases, Adaptive Fault Tolerance, and Security. Due to lack of space we do not discuss adaptive fault tolerance and security. A summary of the work concludes the paper.

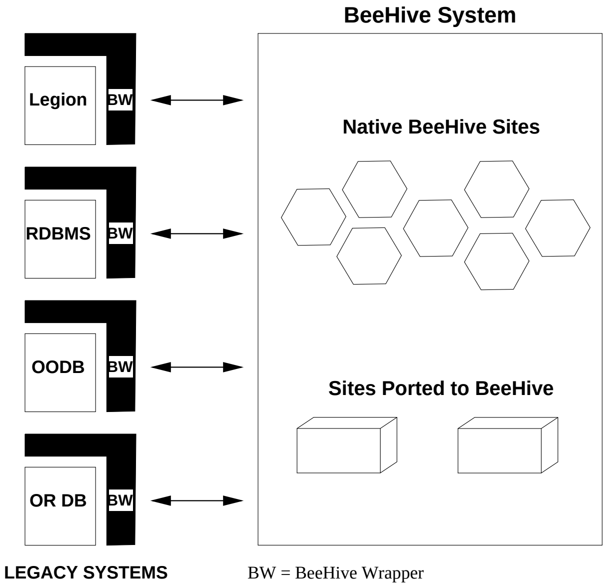

Figure 1 BeeHive.

2 GENERAL BEEHIVE DESIGN

2.1 An Overview of the Design

BeeHive is an application-focussed global virtual database system. For example, it could provide the database level support needed for information technology in the integrated battlefield. BeeHive is different than the World Wide Web and databases accessed on the Internet in many ways including BeeHive's emphasis on sensor data, use of time valid data, level of support for adaptive fault tolerance, support for real-time databases and security, and the special features that deal with crisis mode operation. Parts of the system can run on fixed secure hosts and other parts can be more dynamic such as for mobile computers or general processors on the Internet.

The BeeHive design is composed of native BeeHive sites, legacy sites ported to BeeHive, and interfaces to legacy systems outside of BeeHive (see Figure 1).

The native BeeHive sites comprise a federated distributed database model that implements a temporal data model, time cognizant database and QoS protocols, a specification model, a mapping from this specification to four APIs (the OS, network, fault tolerance and security APIs), and underlying novel object support. Any realistic application will include legacy databases. BeeHive permits porting of these databases into the BeeHive virtual system by a combination of wrappers and changes to the underlying software of these systems. It is important to mention that BeeHive, while application focussed, is *not* isolated. BeeHive can interact with other virtual global databases, or Web browsers, or individual non-application specific databases via BeeHive wrappers. BeeHive will access these databases via downloaded Java applets that include standard SQL commands. In many situations, not only must information be identified and collected, but it must be analyzed. This analysis should be permitted to make use of the vast computer processing infrastructure that exists. For example, BeeHive will have a wrapper that can utilize a distributed computing environment, such as the Legion system [8], to provide significant processing power when needed.

2.2 Native BeeHive Design

The basic design of a native BeeHive site is depicted in Figure 2. At the application level, users can submit transactions, analysis programs, general programs, and access audio and video data. For each of these activities the user has a standard specification interface for real-time, QoS, fault tolerance, and security. At the application level, these requirements are specified in a high level manner. For example, a user might specify a deadline, full quality QoS display, a primary/backup fault tolerance requirement, and a confidentiality level of security. For transactions, users are operating with an object-oriented database invoking methods on the data. The data model includes timestamped data and data with validity intervals such as is needed for sensor data or troop position data. As transactions (or other programs) access objects, those objects become active and a mapping occurs between the high level requirements specification and the object API via the mapping module. This mapping module is primarily concerned with the interface to object wrappers and with end-to-end issues. A novel aspect of our work is that each object has semantic information (also called reflective information because it is information about the object itself) associated with it that makes it possible to simultaneously satisfy the requirements of time, QoS, fault tolerance, and security in an adaptive manner. For example, the information might include rules or policies and the action to take when the underlying system cannot guarantee the deadline or level of fault tolerance requested. This semantic information also includes code that makes calls to the resource management subsystem to satisfy or negotiate the resource requirements. The resource management subsystem further translates the requirements into resource specific APIs such as the APIs for the OS, the network, the fault tolerance support mechanisms, and the security subsystem. For example, given that a user has invoked a method on an object with a deadline and primary/backup requirement, the semantic information associated with the object makes a call to the resource manager requesting this service. The resource manager determines if it can allocate the primary and backup to (1) execute the method before its deadline and (2) inform the OS via the OS API on the modules' priority and resource needs.

In terms of this design, the main tasks to be undertaken include

- the full development of the high-level specification including how these requirements interact with each other,
- the implementation of real-time object-oriented database support,
- the design and implementation of our semantics enhanced objects,
- the design and implementation of the object-oriented wrappers,
- the development of the mapping module,
- the design and implementation of the resource management, fault tolerance, and security subsystems.

3 RESOURCE MANAGEMENT AND QOS

A critical component for the success of BeeHive is its ability to efficiently manage a vast amount of resources. BeeHive requires end-to-end resource management, including physical resources such as sensors, endsystems resources such as operating systems, and communications resources such as link bandwidth.

We assume that low-level resource management is available for single low-level system resources, such as operating systems and networks. For networks, resource reservation signaling is based on the RSVP [3] and UNI 4.0 [2] protocols for IP networks and ATM networks, respectively. Likewise, we assume that all operating systems are provided with a resource management entity.

Based on these comparatively primitive resource management systems, BeeHive will implement a sophisticated end-to-end adaptive resource management

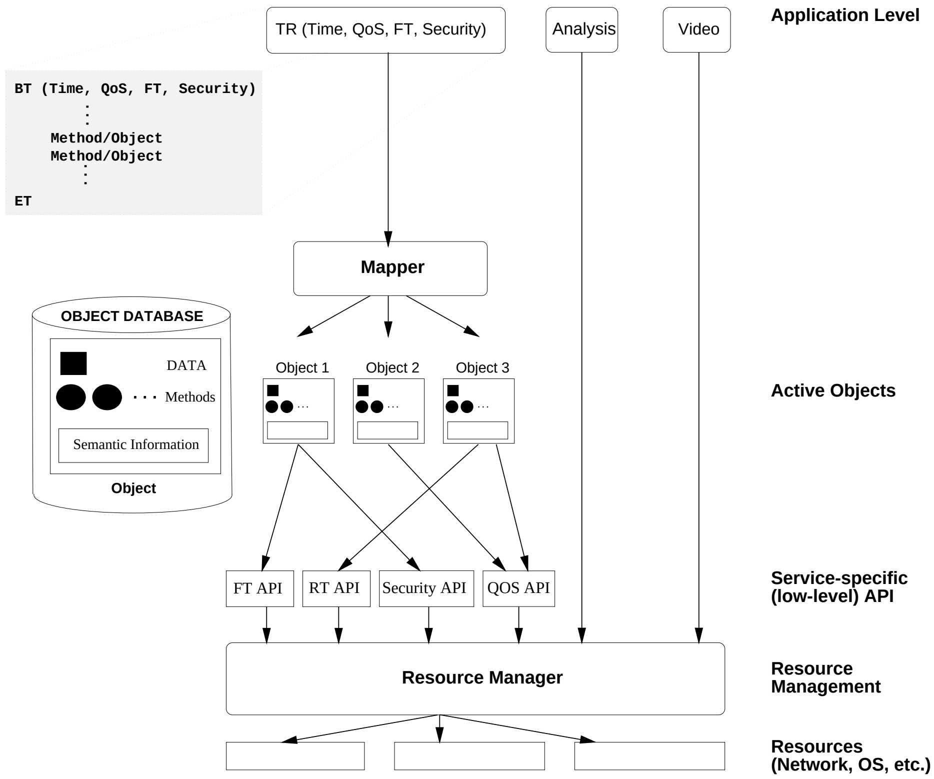

Figure 2 Native BeeHive Site.

system that supports applications with widely varying service requirements, such as requirements on timeliness, fault tolerance, and security. The resource management in BeeHive offers the following services:

- Provide service-specific application programming interfaces (APIs) that allow application programmers to specify the desired QoS without requiring knowledge of the underlying low-level resource management entities. The QoS can be a function of mode (normal or crisis mode).
- Map qualitative, application-specific service requirements into quantitative resource allocations.
- Dynamically manage network and systems resources so as to maximize resource utilization.

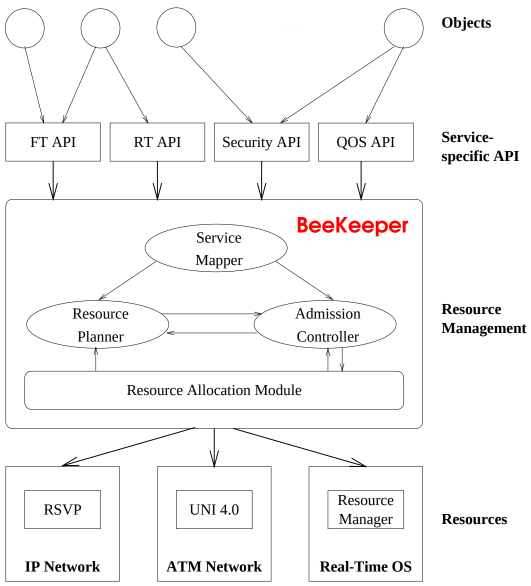

Figure 3 Resource Management in BeeHive.

By providing service-specific APIs, we allow application programmers to specify the QoS requirements of an application in an application-specific fashion. The resource management entities of BeeHive are responsible for mapping the QoS requirements into actual resource needs. The advantage of this approach is a significant increase of reusability. Specifically, an application need not be modified if the underlying resource infrastructure changes. (It may be convenient to think of our approach as "Resource Hiding").

To maximize resource utilization in BeeHive, we will enhance resource management with a planning component. The planning component keeps track of the dynamic behavior of resource usage. By maintaining information not only of the current state of resource utilization, but also of past and (predicted) future usage, the resource management scheme can adapt to the changing resource demands in the system especially during crisis.

In Figure 3 we illustrate the main components of the resource management system in BeeHive. These components will be discussed below.

Service-Specific Application Programming Interfaces

The application programming interface (API) provided by BeeHive must satisfy several constraints. On the one hand, the API must allow the application programmer to access the full functionality of the system without being burdened with internal details. On the other hand, the API must be simple enough as to provide a simple (and extensible) internal representation. The API of the resource management system in BeeHive is a trade-off between the requirements for application specificity and internal simplicity.

- Since BeeHive operates in a heterogeneous distributed computing environment, application programmers should not be required to have knowledge of the underlying resource infrastructure on top of which the application is being executed.
- Rather than adopting a "one-size-fits-all" approach, we provide a set of different APIs. More specifically, we design a separate API for each of the value-added services provided by BeeHive. In this project, we will build four APIs for the following services:
  - Applications with Real-Time Requirements.
  - Applications with QoS Requirements.
  - Applications with Fault-Tolerance Requirements.
  - Applications with Security Requirements.

  For example, the service-specific API allows application developers to specify the QoS requirements of a fault-tolerant application in terms of MTTF (Mean Time To Failure). The resource manager maps these service-specific QoS requests into actual resource requests. Of course, applications and/or individual tasks or transactions may require more than one or even all services. The BeeKeeper (described next) is responsible for tradeoffs between these services.

"BeeKeeper" – The Resource Manager of BeeHive

The resource manager of BeeHive, referred to as the "BeeKeeper", is the central entity of the resource management process. The main function of the BeeKeeper is the mapping of service-specific, possibly qualitative, QoS requirements into actual, quantitative, resource requests. The following are the main components of the BeeKeeper:

- The *Service Mapper* performs the mapping of qualitative resource requests into quantitative requests for physical resources. The service mapper generates a uniform internal representation of the multiple (service-dependent) QoS requests from applications. The uniform presentation is derived from a novel task model discussed below.
- The *Admission Controller* performs the tests that determine if BeeHive has sufficient resources to support the QoS requirements of a new application without compromising the QoS guarantees made to currently active applications.
- The *Resource Allocation Module* is responsible for managing the interface of BeeHive to underlying resource management systems of BeeHive components, i.e., the resource management entities of an ATM network, an RSVP managed IP network, or a real-time operating system, such as RT-Mach. It maintains a database on resources allocated to the BeeHive application.
- The *Resource Planner* attempts to globally optimize the use of resources. The Admission Controller of the BeeKeeper merely decides whether a new application is admitted or rejected. Obviously, such a binary admission control decision leads to a greedy and globally suboptimal resource allocation. (Note that all current resource allocation methods, e.g., for ATM networks or real-time operating systems, are greedy.) The Resource Planner is a module that enhances the admission control process in order to provide globally optimal resource allocations. The Resource Planner prioritizes all current and incoming requests for resources. Based on the prioritization, it devises a resource allocation strategy.

  The Resource Planner obtains from the Resource Allocation Module information on the state of current resource allocations. As much as possible, the Resource Planner should be provided with information on future resource usage. The Resource Planner processes this information and provides input to the Admission Controller.

  In BeeHive, the Resource Planner of the BeeKeeper plays a central role for adaptive resource allocation. If an incoming request with high-priority cannot be accommodated, the Resource Planner initiates a reduction of the resources allocated to low-priority applications. If necessary, the Resource Planner will decide upon the preemption of low-priority applications.

Internal Uniform Task Model. An important part of resource management in BeeHive is a new task model which yields a uniform presentation of applications within the BeeKeeper. The model, referred to as *unified task model*, is flexible enough to express the stringent timeliness requirements of all applications that run in BeeHive. On the other hand, the scheme is sophisticated enough to cope with the complexity of multimedia tasks, such as variable bit rate compression of video streams. The unified task model will provide us with a single abstraction for reasoning about multimedia and real-time systems, thereby, overcoming the traditional separation of multimedia and real-time control systems.

Let a(t) denote the execution time necessary for a task to complete the workload that arrives for the task at time t. Let $A[t, t+\tau] = \int_{t,t+\tau} a(x)dx$ denote the execution time necessary to complete the workload that arrives to the system in the time interval $[t, t+\tau]$.

Then the workload that arrives for a task can be characterized by a so-called *task envelope* A^* which provides a time-invariant upper bound on A, that is, for all times $\tau \geq 0$ and $t \geq 0$ we have [5]:

$$A[t, t+\tau] \leq A^*(\tau) \tag{1.1}$$

A task envelope A^* should be *subadditive*, that is, it should satisfy

$$A^*(t_1) + A^*(t_2) \geq A^*(t_1 + t_2) \qquad \forall t_1, t_2 \geq 0 \tag{1.2}$$

If a task envelope A_1^* satisfies (1.1) but is not subadditive, it can be replaced by a subadditive envelope A_2^* such that $A_2^*(t) \leq A_1^*(t)$ for all $t \geq 0$. The notion of task envelopes is powerful enough to describe all hard real-time and soft-real time tasks.

The processing requirements of a task are described as follows. Let $S[t, t+\tau]$ denote the amount of service time that a processor can devote to processing one or more instances of the task in time interval $[t, t+\tau]$. With the task envelope A^* and a deadline for completing an instance of that task, say D, a task always meets its deadline if for all $\tau \geq D$ we have: $\min_\tau\{A^*(t-\tau) = S[0,t])\} \leq D$. We use $S^*(t) = A^*(t-D)$ to denote the *service envelope* of the task.

4 REAL-TIME DATABASES

In applications such as the integrated battlefield or agile manufacturing, the state of the environment *as perceived by the controlling information system* must be consistent with the actual state of the environment being controlled or within prescribed limits. Otherwise, the decisions of the controlling system may be wrong and their effects disastrous. Hence, the timely monitoring of the environment, the timely processing of the sensed information, and the timely derivation of needed data are essential. Data maintained by the controlling systems and utilized by its actions must be up-to-date and temporally correlated. This *temporal consistency* must be maintained through the timely scheduling of the actions that refresh the data.

In these applications, actions are triggered by the occurrence of events. An event triggers an action only if certain conditions hold. For instance, the occurrence of the event corresponding to a temperature reading would trigger an emergency reaction only if the temperature value is above a threshold value. The Event-Condition-Action (ECA) paradigm of active databases is convenient to enforce these constraints and also to trigger the necessary actions. Rules can be designed to trigger entry into specific modes; to trigger the necessary adaptive responses to time constraint violations – to effect recovery, to trigger actions if temporal data is (about to become) invalid; and to shed loads as well as adjust deadlines and other parameters, e.g., importance levels and QoS, of actions, when overloads occur, and to help support security. The ECA paradigm will be a core component of the BeeHive system.

Transactions that process data with validity intervals must use timely and relatively consistent data in order to achieve correct results. We have developed the ideas of data deadline and forced delay for processing transactions that use (sensor) data with temporal validity intervals. Data read by a transaction must be valid when the transaction completes, which leads to another constraint on completion time, in addition to a transaction's deadline. This constraint is referred to as *data-deadline*. Within the same transaction class, the scheduling algorithm should be aware of the *data-deadline* of a transaction, that is, the time after which the transaction will violate temporal consistency[1]. The scheduling algorithm should account for data-deadlines when it schedules transactions whenever a data-deadline is less than the corresponding transaction deadline. To do this, we have developed earliest data-deadline first (EDDF) and data-deadline based least slack first (DDLSF) policies. Since EDDF and DDLSF policies do not consider the *feasibility* of validity intervals of data objects that a transaction accesses, they are combined with the idea of *Forced-Wait*. With *Forced-Wait*, whenever a temporal data object is read by a transaction, the system checks if this transaction will commit before the validity of the data object expires. If the validity could expire before the commit then the transaction is made to wait until the data object is updated, else the transaction is allowed to continue.

[1] Note that a transaction can violate temporal consistency without missing its deadline.

Another important area in real-time database systems is recovery. As a basis for supporting real-time database recovery, we assume four-level memory hierarchy. The first level consists of main memory that is volatile. At the second level is non-volatile RAM (NV-RAM). The third level consists of the persistent disk storage subsystem, and at the fourth level is archival tape storage.

The motivation for the use of NV-RAM stems from the fact that maintaining large amounts of data in main memory can be very expensive while disk I/O times might be unacceptable in certain situations. For instance, writing to disk for the purpose of logging data touched by critical transactions and reading from disk to undo critical transactions might be too expensive. It is not difficult to conceive situations where writing to disk may result in missing deadlines, but by writing to NV-RAM, deadlines can be met. NV-RAM can be used purely as a disk cache where the data moved to NV-RAM later migrates to the disk or it can be used as a temporary stable storage where the data is stored for performance reasons and later may or may not not migrate to the disk depending on the (durability) characteristics of the data.

Persistence of system data structures such as global lock tables, and rule bases that represent the dependencies between transactions could become potential bottlenecks in real-time databases. Making these system data structures persistent in NV-RAM results in better performance.

In our solution, the characteristics of a particular type of data will determine where data is placed in the four-level memory hierarchy, where the logs are maintained and how the system recovers from transaction aborts. Data is characterized by temporality, frequency of access, persistence, and criticality. We then tailor the data placement and logging and recovery techniques that are needed to the data characteristics.

5 SUMMARY

We have described the design of BeeHive at a high level. We have identified novel component solutions that will appear in BeeHive. More detailed design is continuing an a prototype system is planned. Success of our approach will provide major gains in performance (and QoS), timeliness, fault tolerance, and security for global virtual database access and analysis. The key contributions would come from raising the distributed virtual system notions to the transaction and database levels while supporting real-time, fault tolerance, and security properties. In application terms, success will enable a high degree of confidence in the usability of a virtual database system where a user can obtain secure and timely access to *time valid data* even in the presence of faults. Users can also dynamically choose levels of service when suitable, or the system can set these service levels automatically. These capabilities will significantly enhance applications such as information dominance in the battlefield, automated manufacturing, or decision support systems. However, many research questions remain.

Acknowledgements

This chapter is a short version of a similar invited paper presented at the Second Workshop on Active Real-Time Databses, Lake Como, Italy, Sept. 1997.

REFERENCES

[1] R. Abbott and H. Garcia-Molina, Scheduling Real-Time Transactions: A Performance Evaluation, *ACM Transactions on Database Systems*, Vol. 17, No. 3, pp. 513-560, September 1992.

[2] ATM Forum, ATM User-Network Interface Specification, Version 4.0, 1996.

[3] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification, Internet Draft, November 1996.

[4] M. J. Carey, R. Jauhari and M. Livny, On Transaction Boundaries in Active Databases: A Performance Perspective, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 3, pp. 320-336, September 1991.

[5] R. L. Cruz, A Calculus for Network Delay, Part I: Network Elements in Isolation, *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[6] U. Dayal et. al., The HIPAC Project: Combining Active Databases and Timing Constraints, *SIGMOD Record*, Vol. 17, No. 1, pp. 51-70, March 1988.

[7] M. Di Natale and J. Stankovic, Dynamic End-to-End Guarantees in Distributed Real-Time Systems, *Real-Time Systems Symposium*, Dec. 1994.

[8] A. Grimshaw, W. Wulf, and the Legion Team, The Legion Vision of a Worldwide Virtual Computer, *CACM*, Vol. 40, No. 1, January 1997, pp. 39-45.

[9] J. Huang, J.A. Stankovic, D. Towsley and K. Ramamritham, Experimental Evaluation of Real-Time Transaction Processing, *Real-Time Systems Symposium*, pp. 144-153, December 1989.

[10] T. Kuo and A. K. Mok, Real-Time Data Semantics and Similarity-Based Concurrency Control, *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[11] J. Liebeherr, D. E. Wrege, and D. Ferrari, Exact Admission Control in Networks with Bounded Delay Services, *IEEE/ACM Transactions on Networking*, Vol. 4, No. 6, pp. 885-901, December 1996.

[12] R. M. Sivasankaran, K. Ramamritham, J. A. Stankovic, and D. Towsley, Data Placement, Logging and Recovery in Real-Time Active Databases, *Workshop on Active Real-Time Database Systems,* Sweden, June 1995.

[13] J. Stankovic and K. Ramamritham, Reflective Real-Time Operating Systems, *Principles of Real-Time Systems*, Sang Son, editor, Prentice Hall, 1995.

[14] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham and D. Towsley, Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics, *Real-Time Systems Symposium*, December 1996.

[15] J. A. Zinky, D. E. Bakken, and R. Schantz, Overview of Quality of Service for Objects, In *Proceedings of the Fifth IEEE Dual Use Conference*, May 1995.