Architecture and Object Model for Distributed Object-Oriented Real-Time Databases

John A. Stankovic *Sang H. Son*
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

The confluence of computers, communications, and databases is quickly creating a global virtual database where many applications require real-time access to both temporally accurate and multimedia data. This is particularly true in military and intelligence applications, but these required features are needed in many commercial applications as well. We are developing a distributed database, called BeeHive, which could offer features along different types of requirements: real-time, fault-tolerance, security, and quality-of service for audio and video. Support of these features and potential trade-offs between them could provide a significant improvement in performance and functionality over current distributed database and object management systems. In this paper, we present a high level design for BeeHive architecture and sketch the design of the BeeHive Object Model (BOM) which extends object-oriented data models by incorporating time and other features into objects.

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 globally distributed databases. Such databases could be enterprise specific and offer features along real-time, fault tolerance, quality of service (QoS) for audio, video and images, and security dimensions. Support of these features and potential trade-offs between them could provide a significant improvement in performance and functionality over current distributed database and object management systems.

There are many research problems that must be solved to support the global, real-time 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, i.e., in both the operating system and network layers. 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 scheduling, resource management and renegotiation algorithms for sets of services.

In the remainder of this paper we discuss the high-level BeeHive system architecture and sketch the design of a native BeeHive site showing how all the parts fit together. We also present technical details on the BeeHive Object Model (BOM) and its properties. A summary of the work concludes the paper.

2 General BeeHive Design

2.1 System Overview

BeeHive is an application-focussed distributed database system. For example, it could provide the database level support needed for information technology in the integrated battlefield. BeeHive is different from 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, interfaces to legacy systems outside of BeeHive, and an interface to the Legion system[1] [11] (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 sophisticated database 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

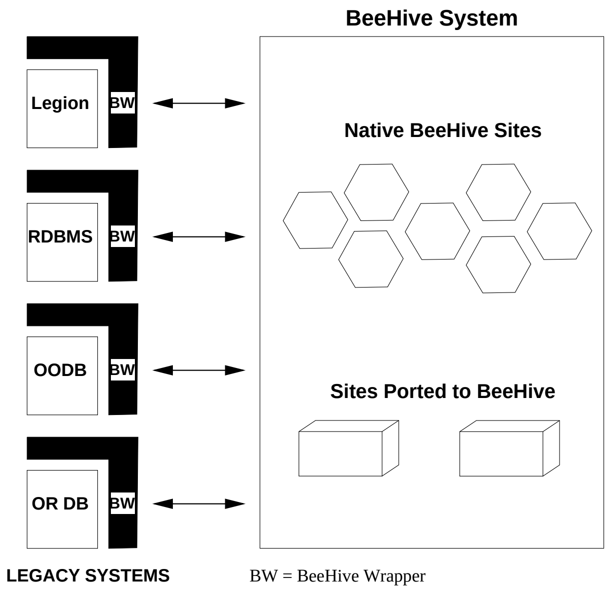

Figure 1: BeeHive.

not isolated. BeeHive can interact with other virtual global databases, or Web browsers, or individual non-application specific databases via BeeHive wrappers[2]. 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. Here, BeeHive will have a wrapper that can utilize the Legion system to provide significant processing power when necessary.

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, video, and image data. For each of these activities the user has a standard specification interface for real-time, QoS for audio, video, and images, fault tolerance, and security. The collection of these four requirements is generically referred to as services. QoS is restricted to mean QoS for audio, video, and images. At the application level, these requirements

[1]Legion is a distributed global execution platform for executing parallel programs.

[2]An initial implementation of BeeHive wrappers that provide an interface to the open information sources of the Internet has been implemented. However, this aspect of BeeHive is not the focus of this paper.

are specified in a high level manner. For example, the 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.

3 Resource Management

A critical component for the success of BeeHive is its ability to efficiently manage distributed 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.

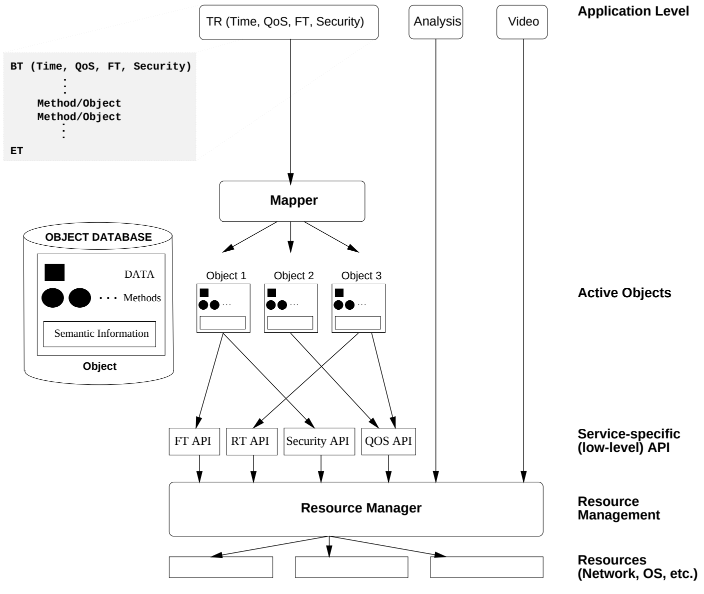

Figure 2: Native BeeHive Site.

We assume that low-level resource management is available for single low-level system resources, such as operating systems and networks. Based on these comparatively primitive resource management systems, BeeHive will implement a sophisticated end-to-end adaptive resource management system that supports applications with widely varying service requirements, such as requirements on timeliness, QoS, fault tolerance, and security. The resource management in BeeHive offers the following services:

- Provide service-specific application programming interfaces (APIs) that allow developers to specify the desired services without requiring knowledge of the underlying low-level resource management entities. The services 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.

By providing service-specific APIs, we allow application programmers to specify the service requirements of an application in an application-specific fashion. The resource management entities of BeeHive are responsible for mapping the service requirements into actual resource needs. For example, for the QoS service, a high quality video requirement might map to 10% of

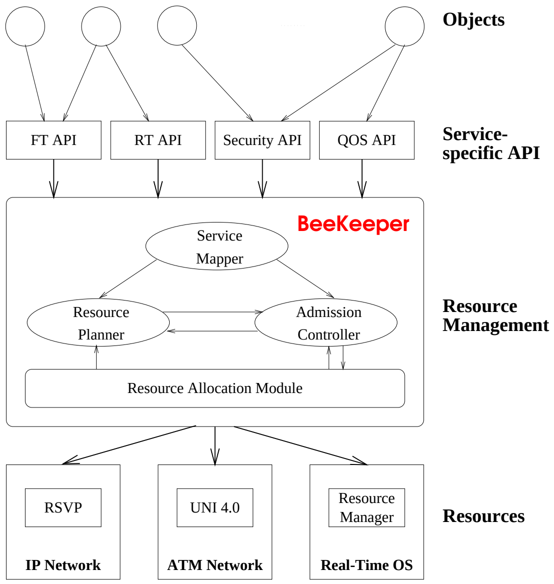

Figure 3: Resource Management in BeeHive.

the CPU and 10 Mbytes of network bandwidth. 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 developers 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 audio and video 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 the 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 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) requests from applications.

- The *Admission Controller* performs the tests that determine if BeeHive has sufficient resources to support the service requirements of a new application without compromising the service 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 most current resource allocation methods, e.g., for ATM networks or real-time operating systems, are greedy.) The Resource Planner is a module to enhance the admission control process and to yield globally improved 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 allocation. As much as possible, the Resource Planner should be provided with information on future resource usage. The Resource Planner processes this information and provides it 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 or abortion of low-priority applications.

4 BeeHive Object Model

In the BeeHive architecture, the user sees the database as a set of BeeHive objects, a set of transactions, and a set of rules. Objects are used for modeling entities in the real world, transactions are used to specify application requirements and to execute the functionality of the application, and rules are used for defining constraints and actions to be taken. Our BeeHive object model (BOM) extends traditional object-oriented data models by incorporating semantic information regarding real-time, fault-tolerance, importance, security, and QoS requirements. This information includes worst-case execution time, resource requirements, and other constraints that must be considered in resource management, scheduling, and trading-off among different types of requirements. In this section we present each component of the BOM, and then discuss how they can be supported by the BeeHive architecture. The BOM has some similarity in terms of the structure of objects to the RTSORAC object model [12]. One of the main differences is that while RTSORAC model supports only real-time and approximation requirements, BOM supports a rich set of types of requirements and their trade-offs.

4.1 Objects

A BeeHive object is a tuple $< N, A, M, CF, T >$, in which N is an object identifier, A is a set of attributes (composed of a name, domain information and values), M is a set of methods, CF is a compatibility function that determines how method invocations can coexist, and T is a timestamp of the latest update. If no updates have been done to this object, the value of T is the object creation time. The data stored in an object can have temporal consistency specified by a validity interval. The value part of each attribute is represented by a value and a validity interval. Each attribute has a timestamp of the latest update time of the attribute value. The timestamp is used to determine the temporal consistency of the value. Other semantic characteristics may be added in the future. For example, if an attribute X contains some amount of imprecision in its value, the field $X.MaxImp$ represents the maximum amount of imprecision that can exist in $X.value$.

Every object belongs to a class which is a collection of objects with a similar structure. Classes are used to categorize objects on the basis of shared properties and/or behavior. Classes are related in a subclass/superclass hierarchy that supports multiple inheritance. Complex objects, object identity, encapsulation, object classes, class hierarchies, overloading, overriding, and late binding can be supported by the BOM.

The functional interface of an object is defined as the set of methods M. A method is modeled by $< MN, P, ET, R, SI >$, in which MN is the name of the method that represents the executable code, P is the set of parameters to be used by the method, ET is the execution time requirements of the method (in many cases this will be the WCET[3] of the method), R is the set of resources required by the method other than CPU time (includes memory, I/O, data, etc.), and SI is a set of semantic information associated with the method. For example, if the method M1 can be applied to either single copy or primary/backup copies, SI should specify different constraints to be considered in each case (such as availability of both copies in the case of primary/backup copies). It is important to note that this functional interface is extensible in the set of resource requirements and semantic information associated with methods.

The compatibility function is defined between all method invocations. It uses semantic information specified in methods as well as policies and system state information to specify compatibility between each pair of methods of the object. The function can be specified in the form:

CF(M1, M2) = $< BooleanExpression >$

where the boolean expression can either be a simple *true* or *false*, or it may contain predicates involving semantic information of the object, system state, and policies. For example, if M1 and M2 can be executed in parallel only when the object is being used in primary/backup mode and two instances of the resource O is available, the compatibility function can be stated as follows:

```
boolean Compatibility_M1.M2
if ((Obj_Mode = PB) and (Nr_avail(O) = 2))
then return true
else return false;
```

4.2 Transactions

A transaction is one element of the BeeHive system that brings information from applications to objects, and returns information from objects to applications. A transaction is associated with requirements, policies for trade-offs, and importance, in addition to usual method invocations with required parameters. A transaction is modeled by $< TN, XT, I, RQ, P >$, in which TN is a unique identifier of the transaction, XT is the execution code, I is the importance, RQ is a set of requirements, and P is the policy for trade-offs. The set of requirements, RQ, consists of requirements for each property (real-time, fault-tolerance, QoS, security), and optional PreCond and PostCond. PreCond represents preconditions that must be met before the transaction can be executed, and PostCond represents postconditions that must be satisfied upon completion of the transaction. For example, it may be appropriate for a transaction to execute only if some specified event has occurred, such as after the successful execution of a related transaction. The real-time requirements of a transaction are specified in terms of a deadline, start time, and its period if it requires periodic execution. The fault-tolerance requirements can be specified by the degree and form of redundancy. The security requirements are specified by the level of security of the transaction and types of encryption and authentication to be used.

The policies can be stated by specifying which requirements can be reduced to a lower level. For example, the requirement of 3 replicated copies for fault-tolerance can be reduced to 2 copies if the resource manager cannot assign 3 copies of the object for the transaction. The policies can be stated in several different ways to show the order in which the requirements are to be reduced. For example, one policy can state that if the system cannot allocate resources to satisfy all the requirements, first reduce the fault-tolerance requirement from 3 copies to 2 copies. If the requirements cannot still be satisfied, reduce the security requirement from the secret level to the classified level and omit all the encryption involved, etc. Other policies might state that the fault-tolerance and deadline requirements can be reduced simultaneously, but security requirements cannot be reduced.

4.3 Rules

Rules are used to specify constraints that define correct states of an object and inter-object relationships as well as actions to be taken on events. A rule is modeled by $< RN, O, E, C, A, CM >$, in which RN is the name of the rule, O is the set of objects that are referenced in this rule, E is a set of events that can invoke the rule, C is a set of conditions, A is a set of ordered actions that need to be taken if any, and CM is a coupling mode that describes when the action A is to be executed.

[3] WCET is the worst case execution time.

Any component can be omitted in a rule specification. For example, if the coupling mode CM is missing, the rule only supports the default coupling mode. If the action part A is missing, events are detected and conditions are evaluated, but no action is taken. Depending on the definition of conditions, the action may be a side effect of condition evaluation. If the condition part C is missing, an action is taken every time an event is taken. And if the event part E is missing, conditions are evaluated every time any event occurs in the system, which could be very expensive.

The set of events E describes all events that can trigger this rule. An event in E has two components: $< EN, AE >$, in which EN is the name of the event and $AE \subseteq AtomicEvents$ is a set of atomic events that constitute the event. AtomicEvents represents the set of all possible atomic events that can be detected (e.g., update of a specified data object). Although it is possible to extend our model to support composite events for detecting complex situations, BOM in the current design supports only atomic events in rule specification, since the detection of composite events can be very expensive.

The set of conditions C describes the possible conditions that can trigger an action once an event is detected. A condition is specified by a predicate which can include attributes of objects and system states. Both logical and temporal consistency constraints can be expressed by these conditions with their corresponding actions to take to correct the violated constraints. For example, assume that an object X has an attribute *Position*. If a temporal consistency requirement of the $X.Position$ requires that it is not more than 3 seconds old, a condition $X.Position > Now() - 3$ captures that situation and corresponding action $Update(X.Position)$ will enforce the temporal consistency.

The action A is a pair $< Ty, Op >$ where Ty is action *type* and Op is action *operation*. The action type Ty describes what type of an action can be done. Actual operation of an action depends on action type Ty. In BOM, we support three types of actions: *reject, method invocation*, and *transaction*. The action type definition goes from simple to complex. The type *reject* is the simplest form. Reject means that the current activity that triggered the rule is rejected. An example could be a security violation or failure in authentication. The second type *method invocation* is useful when rules are meant to be used inside an object. Using transactions is a waste of resources in such a case because all operations occur inside an object. Finally, the last type *transaction* can be used with the full functionality of the triggering mechanisms of an active database model.

The Coupling Mode CM describes when the invoked action should be performed. The most common coupling modes to be supported in BOM are immediate, deferred, and detached. In immediate coupling mode, the transaction is triggered immediately; if the event was triggered within a transaction, the triggered transaction is executed as a subtransaction of the triggering transaction. In the deferred coupling, the triggered transaction is executed right before the commit of the triggering transaction. In detached coupling, the triggered transaction is executed as a separate transaction regardless of the triggering condition.

4.4 BeeHive Object Manager

To support the object model described above, the BeeHive architecture will provide modules to manage schemas, objects, transactions, and rules. The functions are separated into the schema manager module, object manager module, storage manager module, transaction manager module, recovery manager module, and rule manager module. Here we only briefly discuss the object manager module (OMM).

The OMM provides the organization of semantic information of objects, creation and deletion of objects, and concurrency control and recovery of objects. All the persistent objects are created and stored in permanent storage using the storage manager. At system startup time, a shared main memory segment is created and an object table is instantiated at a well-known location in the shared segment. The table associates each object's name with the object entry in the shared segment. Semantic information of each object is preprocessed and compiled into method calls and rules associated with the object.

Objects can be shared by concurrent transactions and, hence, concurrency control must be exercised by the OMM. The compatibility function specified for an object should be utilized for concurrent access of objects and invocations of methods. When a transaction requests a method invocation of an object, OMM evaluates the compatibility function of the method with each currently invoked method. Depending on the outcome, a transaction is either allowed to execute the requested method or suspended until the conflicting lock is released.

5 Adaptive Fault Tolerance

Given the large and ever-growing size of databases together with their potential use in applications such as information dominance on the battlefield, faults may occur frequently and at the *wrong times*. For the system to be useful and efficient and to protect against common security breach points, we must have adaptive fault tolerance. In this section, we briefly discuss adaptive fault-tolerance features of BeeHive.

Our approach is to design adaptive and database centric solutions for non-malicious faults. Any system that deals with faults must first specify its fault hypotheses. In particular, we will consider the following fault hypotheses: processors may fail silently (multiple failures are possible); transient faults may occur due to power gliches, software bugs, race conditions, etc.; and timing faults can occur where data is out of date or not available in time.

In our approach we propose a service-oriented fault tolerance and support it with underlying model based on adaptive fault tolerance.

Service-Oriented FT: For service-oriented fault tolerance we consider how typical users operate with BeeHive and consider the fault tolerance aspects of these services. The services are:

- *Read Only Queries:* These can be dynamically requested by users or automatically triggered by the actions in the active database part of BeeHive. These queries can have soft deadlines and can retrieve data of all types including text, audio, video, etc.

- *Update Transactions:* These transactions can be user invoked or automatic. When permitted, they can update any type of data including temporal data.

- *Multimedia Playout and QoS:* When data that is retrieved is audio and video, the playout itself has time constraints, is large in volume, must be synchronized, can be degraded if necessary, etc.

- *Analysis Tools:* Retrieved data may be fed to analysis tools for further processing including having this processing itself be distributed by using Legion [11].

The user-level fault tolerance interface includes features for each of the four service classes for each fault type. For example, the FT service for read-only queries allows queries to proceed when processors fail, be retried if transient faults occur, and can produce partial results prior to the deadline to avoid a timing fault. For multimedia playout, processing can be shifted to other processors when processors fail. A certain degree of transient faults is masked, and degraded service is used to avoid some timing faults. Similar fault tolerance services can be defined for the other combinations.

Support for Adaptive Fault Tolerance: Queries, update transactions, multimedia playout, and analysis tools may access any number of objects. In order to support these fault tolerant services, we propose an underlying system model based on adaptive (secure) fault tolerant (real-time) objects. Since fault tolerance can be expensive, we must be able to tailor the cost of fault tolerance to a user's requirements for it. In our solution, each object in the system represents data and methods on that data and various types of semantic information that supports adaptive (secure) fault tolerance in real-time. Briefly, this works as follows.

Input to an object can be, in addition to the parameters required for its functionality, the time requirement, the QoS requirement, the degree of fault tolerance needed, and the level of security. Inside the object and hidden from the users are control modules which attempt to meet the incoming requirements dynamically based on the request and the current state of the system. This is a form of admission control. For example, a user of an object may want to execute a method on this database object with a passive backup, have all outputs from the object encrypted and have results within 3 minutes. The control module inside the object dynamically interacts with the system schedulers, resource allocators, and encryption objects to perform admission control, make copies and encrypt messages. The admission control calling the schedulers decides whether this can all be done within 3 minutes. If not, its control strategies indicate how to produce some timely result based on the semantics of the object. In this way (in this simple example), the users obtain the fault tolerance, security and time requirements that they want on this invocation subject to the current system state. Another user or this same user at a different time may request different levels of service from this object and the system adapts to try and meet these requirements. Note that crisis mode may trigger changes to *sets of objects* based on the embedded tradeoff strategies.

One key research issue to be investigated is mapping the service level fault tolerance request to the underlying objects. This research question is one of composition. That is, given the underlying object mechanisms that support adaptive fault tolerance how can objects be composed to meet the service level requirements. Similar mapping questions exist for fault tolerance, real-time, and security, and their interaction.

6 Related Work

We are not aware of any efforts to design and build a system with the same capabilities as BeeHive, that is, a global virtual database with real-time, fault tolerance, QoS, and security properties in heterogeneous environments. Of course, standards such as ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity) are being used to get better access to databases, but these are not addressing the research questions posed here. There are several research projects, past and present, that have addressed one or more of the issues of real-time databases, QoS at the network and OS levels, multimedia, fault tolerance, security, and distributed execution platforms. We briefly describe a few of these projects.

The QuO architecture [19] being developed at BBN aims at supporting QoS at the CORBA object level. It provides mechanisms for measuring and enforcing QoS agreements. Its goal is to make distributed applications more predictable and adaptive, even if end-to-end guarantees cannot be provided. It addresses the issues in supporting QoS at the object level, such as synthesizing information about system properties and providing a framework to support code reuse. It uses the QoS Description Language (QDL) to specify an application's expected usage patterns and QoS requirements for a connection to an object. To help the application adapt to different system conditions, QuO supports multiple behaviors for a given functional interface, each bound to the contract for which it is best suited. One of its main objectives is the reduction of variance in system properties, which in the current implementation are performance issues. Although it is planned to broaden the scope to include security and fault-tolerance, it is not clear how such extension can fit into the architecture. Further, they do not consider issues such as transactions and database management.

STRIP (STanford Real-Time Information Processor) [1] is a database designed for heterogeneous environments and provides support for value function scheduling and for temporal constraints on data. Its goals include high performance and the ability to share data in open systems. It does not support any notion of performance guarantees or hard real-time constraints, and hence cannot be used for the applications we are envisioning in this project.

The Distributed Active Real-Time Database System (DEEDS) [2] prototype is an event-triggered real-time database system, using dynamic scheduling of sets of transactions, being developed in Sweden. The reactive behavior is modeled using ECA rules. In the current prototype they do not support temporal constraints of data and multimedia information.

To allow applications to utilize multiple remote databases in dynamic and heterogeneous environments, the notion of mediator was introduced and a prototype was implemented in the PENGUIN system [17]. A mediator is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications. It mainly deals with the mismatch problem encountered in information representation in heterogeneous databases, but no real-time and fault-tolerance issues are pursued as in BeeHive.

While commercial database systems such as Oracle [10] or Sybase [9] allow for the storage of multimedia data, it is usually done as a BLOB (binary large object). These systems are not integrated with real-time applications. Also developed in industry is the Mercuri project [8] where data from remote video cameras is transferred through an ATM network and displayed using X windows, but they provide only best effort services.

In recent years, considerable progress has been made in the areas of QoS support for operating systems, networks, and open distributed systems. However, no existing system can give end-to-end QoS assurances in a large-scale, dynamic, and heterogeneous distributed system. Note that none of the existing QoS network architectures supports an integrated approach to QoS that contains the network as well as real-time applications.

The Tenet protocol suite [4] developed within the context of the BLANCA Gigabit testbed networks presented the first comprehensive service model for internetworks. The work resulted in the design of two transport protocols (CMTP, RMTP), a network protocol (RTIP), and a signaling protocol (RCAP) to support a diverse set of real-time services. The protocols of the Tenet Group have not been tailored towards hard real-time applications, and focused on support of multimedia data. The Tenet protocols do not provide a middleware layer that can accommodate the needs of applications with special requirements for security or fault tolerance.

Several QoS standardization efforts are being undertaken by several network communities. The ATM Forum recently completed a traffic management specification [3] which supports hard-real time applications via peak rate allocations in the CBR service class. All other ATM service classes only give probabilistic QoS guarantees. The IntServ working group of the IETF is working towards a complete QoS service architecture for the Internet, using RSVP [7] for signaling. The draft proposal for a *guaranteed service* definition will support deterministic end-to-end delays; however, an implementation is not yet available.

The Open Software's Foundation Research Institute is pursuing several efforts to build configurable real-time operating systems for modular and scalable high-performance computing systems. An important effort in respect to fault-tolerance is the CORDS [15] system. CORDS develops an extensible suite of protocols for fault isolation and fault management in support of dependable distributed real-time applications. The project is targeted at military embedded real-time applications and focuses on operating systems solutions, in particular IPC primitives.

7 Summary

We have described the design of BeeHive at a high level. We have identified novel component solutions that will appear in BeeHive. We have presented the architecture and the object model of BeeHive. More detailed design is continuing and a prototype system is being developed. Success of our approach will provide major gains in performance, timeliness, fault tolerance, QoS, and security for global distributed database access and analysis. The key contributions would come from raising the distributed system notions to the transaction and database levels while supporting real-time, fault tolerance, QoS, and security properties. In application terms, success will enable a high degree of confidence in the usability of a distributed 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 that must be resolved remain. They include

- developing an overall *a priori* analysis on the performance and security properties of the system, given a collection of adaptive objects,
- developing efficient techniques for on-line dynamic composition of these new objects,
- analyzing interactions and tradeoffs among the myriad of choices available to the system,
- determining if the fault models are sufficient,
- creating time bounded resource management and admission control policies,
- determining if there is enough access to legacy systems to achieve the security, functionality, timeliness, and reliability required,
- determining how the system works in crisis mode, and
- determining how the system scales.

References

[1] B. Adelberg, B. Kao, and H. Garcia-Molina, An Overview of the STanford Real-time Information Processor, *ACM SIGMOD Record*, 25(1), 1996.

[2] S.F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring, DeeDS: Towards a Distributed and Active Real-Time Database Systems, *ACM SIGMOD Record*, 15(1):38–40, March 1996.

[3] ATM Forum, *ATM Traffic Management Specification 4.0*, April 1996.

[4] A. Banerjea, D. Ferrari, B. A. Mah, M. Moran, D. C. Verma, and H. Zhang, The Tenet Real-Time Protocol Suite: Design, Implementation, and Experiences, *IEEE/ACM Transactions on Networking*, 4(1):1–10, February 1996.

[5] A. Bondavalli, J. Stankovic, and L. Strigini, Adaptive Fault Tolerance for Real-Time Systems, *Third International Workshop on Responsive Computer Systems*, September 1993.

[6] A. Bondavali, J. Stankovic, and L. Strigini, Adaptable Fault Tolerance for Real-Time Systems, *Responsive Computer Systems: Towards Integration of Fault Tolerance and Real-Time*, Kluwer, 1995, pp. 187-205.

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

[8] A. Guha, A. Pavan, J. Liu, A. Rastogi, and T. Steeves, Supporting Real-Time and Multimedia Applications on the Mercuri Testbed, *IEEE Journal on Selected Areas in Communications*, Vol. 13, No. 4, May 1995.

[9] J. E. Kirkwood, *Sybase Architecture and Administration*, Prentice-Hall, 1993.

[10] G. Koch and K. Loney, *Oracle: The Complete Reference*, Mc Graw-Hill, 1997.

[11] M. J. Lewis and A. Grimshaw, The Core Legion Object Model, In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[12] J. Prichard, L. DiPippo, J. Peckham, and V. Wolfe, RTSORAC: A Real-Time Object-Oriented Database Model, *Database and Expert System Applications Conference (DEXA'94)*, August 1994.

885-901, Ramamritham,

[13] J. Stankovic and K. Ramamritham, The Spring Kernel: A New Paradigm for Hard Real-Time Operating Systems, *IEEE Software*, 8(3):62-72, May 1991.

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

[15] F. Travostino and E. Menze III, The CORDS Book, OSF Research Institute, September 1996.

[16] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, *IEEE Communications Magazine*, 14(2), February 1997.

[17] G. Wiederhold, Mediators in the Architecture of Future Information Systems, *IEEE Computer*, Vol. 25, No. 3, March 1992, pp. 38-49.

[18] M. Xiong, J. Stankovic, K. Ramamritham, D. Towsley and R. M. Sivasankaran, Maintaining Temporal Consistency: Issues and Algorithms, *The First International Workshop on Real-Time Databases*, March, 1996.

[19] J. Zinky, D. Bakken, and R. Schantz, Architectural Support for Quality of Service for CORBA Objects, *Theory and Practice of Object Systems*, 3(1):1-20, April 1997.