BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications

John A. Stankovic, Sang H. Son, and Jörg Liebeherr

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. We are developing a global virtual database, called BeeHive, which is enterprise specific and offers 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 significant improvement in performance and functionality over browsers, browsers connected to databases, and, in general, today's distributed databases. We present a high level design for BeeHive and various novel component technologies that are to be incorporated into BeeHive.

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 the remainder of this paper 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 the main functional ingredients of BeeHive which include Resource Management and QoS, Real-Time Databases, Adaptive Fault Tolerance, and Security. A brief description of the state of art is given. A summary of the work concludes the paper.

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

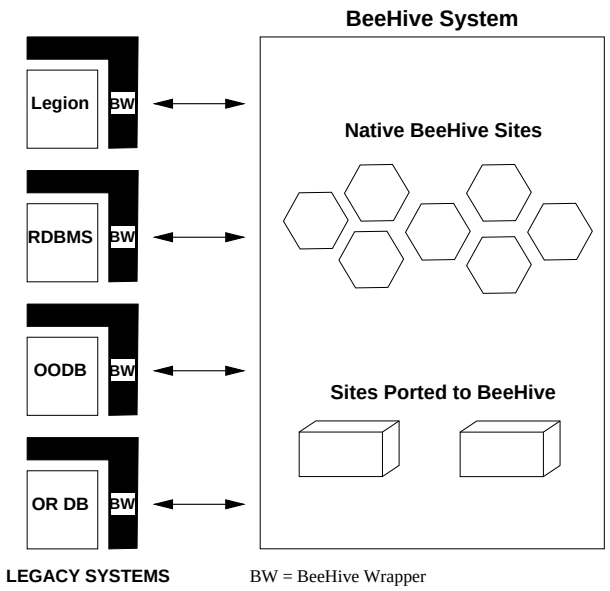

Fig. 1. BeeHive.

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 [20], 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.

In the following sections, some of our ideas on resource management, real-time databases, adaptive fault tolerance, and security are described.

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 [13] and UNI 4.0 [8] protocols for IP networks and ATM networks, respectively. Likewise, we assume that all operating systems are provided with a resource management entity.

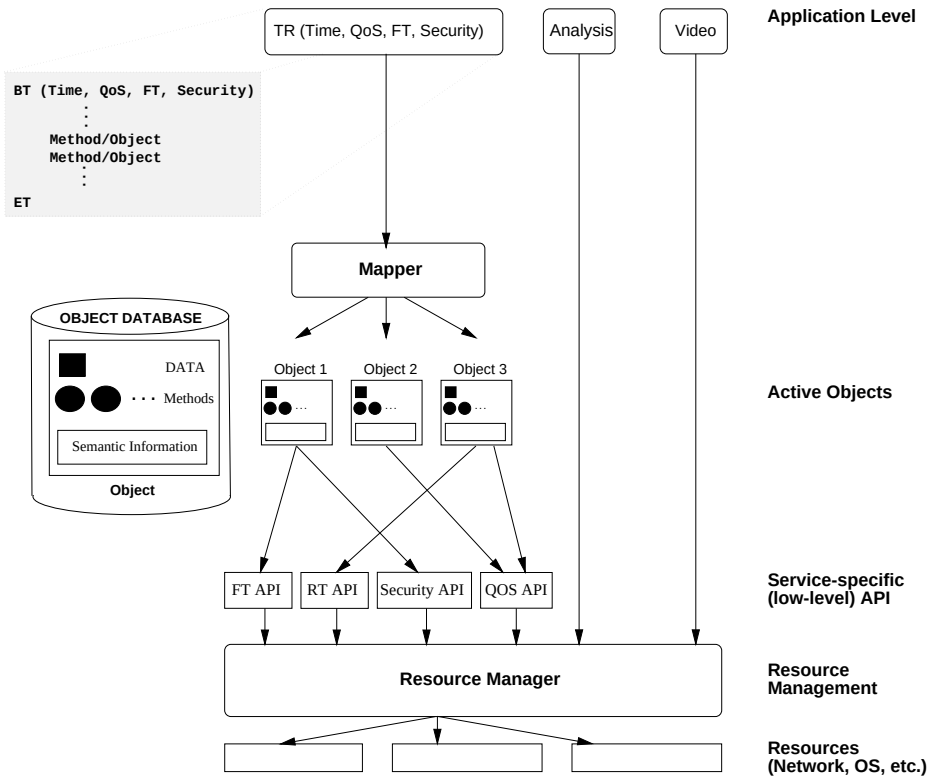

Fig. 2. Native BeeHive Site.

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, 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.

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

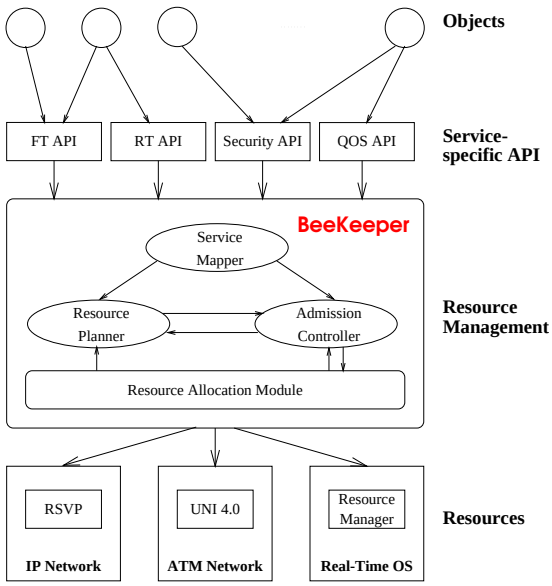

Fig. 3. Resource Management in BeeHive.

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 [15]:

$$A[t, t+\tau] \leq A^*(\tau) \tag{1}$$

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

$$A^*(t_1) + A^*(t_2) \geq A^*(t_1 + t_2) \qquad \forall t_1, t_2 \geq 0 \tag{2}$$

If a task envelope A_1^* satisfies (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 [46], 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. As an example, assume we have data which has a short time validity, high frequency of access, non-persistent, and is critical. Positions of flying aircraft are examples of such data. This kind of data will be placed in main memory. For space reasons, the data might temporarily migrate to NV-RAM if the validity is long enough for a NV-RAM write and read. No-steal buffer policy will be used and so there is no need to undo. In addition, given the short validity, redos will also not be needed or feasible. If some data has long validity, high frequency of access, persistent, and is critical (reactor temperatures in a chemical plant have this property), then this kind of data will be placed in main memory for performance reasons. No-steal buffer policy along with the force policy will be used where data is forced to NV-RAM and subsequently to disk. Similar tailored solutions exist for the other possibilities.

The frequency of access attribute dictates where one should place the data such that I/O costs are minimized. In traditional databases, disk pre-fetching is a technique that is used to minimize the I/O delay. In our context, an analog of this technique can be used, also to ensure that valid data is available when needed. Specifically, a similar technique, namely, *pre-triggering*, can be used to acquire temporal data that is going to be accessed, but is invalid or will become invalid by the time the data is needed. Instead of triggering a transaction to acquire the data just before it is needed, the transaction can be triggered earlier, at some

opportune time.

5 Adaptive Fault Tolerance

Given the large and ever-growing size of databases and global virtual databases, faults may occur frequently and at the *wrong times*. For the system to be useful and to protect against common security breach points, we must have adaptive fault tolerance.

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 glitches, software bugs, race conditions; and timing faults can occur where data is out of date or not available in time. If the global virtual database can handle these faults and operate efficiently, then it should prove to be robust and useful under typical scenarios. With these fault tolerance mechanisms in place, it is possible to consider adding support for malicious faults at a future time, especially since our solutions (outlined below) will support adaptive fault tolerance. However, malicious faults are beyond the scope of work at this stage of the project.

In our solution 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; this processing itself can be distributed.

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 user's requirements. In our solution, each object in the system represents data and methods on that data and various types of semantic information that support 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, 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 a database object with a passive backup, have all outputs from the object encrypted and have results within three minutes. In such a case 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 three minutes. If not, its control strategies indicate how to produce some timely result based on the semantics of the object. In this way the user obtains the fault tolerance, security and time requirements desired 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 is the mapping of 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 Security

Security is an integral part of the system and is one component in the integrated interface to our adaptive, fault tolerant, real-time, and secure objects. Our primary goal is to create a security architecture that is consistent with also meeting real-time, fault tolerance and QoS requirements. In the architecture, users can specify the level of security required and a secure mapping level takes this requirement and maps it to the underlying security API. This underlying API can

change as results from our project or other projects become known. However, because of our novel underlying object paradigm, some novel security support techniques have been identified. In BeeHive, classes (in object oriented programming language terms) have security rules associated with them. Security rules can be inherited by subclasses or overridden depending on the security model in effect for that object. Rules can belong to each method or to the object as a whole. Each object can have its own security API which the mappping layer will utilize, but the rules themselves are hidden. The security rules can be if-then-else rules as well as utilize the notion of an *encrypted profile* to either look for patterns of illegal access or, alternatively, to certify a good pattern of access. Standard security features such as passwords, encryption, and bytecode verifiers can be part of this architecture. Legacy systems, subsystems, proprietary systems, etc. can be surrounded by a firewall which is an object wrapper with a particular security API.

7 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, and security properties in heterogeneous environments. However, there are several 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.

STRIP (STanford Real-Time Information Processor) [2] 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 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.

DeeDS (Distributed Active Real-Time Database System) [6] 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 [64]. 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 [31] or Sybase [30] allow for the storage of multimedia data, it is usually done as BLOBs. These systems are not integrated with real-time applications. Also developed in industry is

the Mercuri project [21] where data from remote video cameras is transferred through an ATM network and displayed using X windows, but they provide only best effort services. The Presto project deals with providing session-based QoS guarantees to continuous multimedia database applications, and does not address the coexistence of control data with the continuous media data.

Projects such as Legion [36, 20] concentrate on distributed execution platforms but do not deal with multimedia databases and end-to-end QoS guarantees. By providing BeeHive wrappers to Legion we will be able to support Legion objects within BeeHive that satisfy QoS guarantees. Commercial systems with similar goals as Legion, however, focused exclusively on a client-server model, are OSF/DCE [40] and CORBA [63]. The CORBA standards and products based on them also do not have the functionality nor real-time properties that we are developing, although a real-time CORBA is emerging.

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 support an integrated approach to QoS that contains the network as well as real-time applications.

The Tenet protocol suite [10] 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 rather 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.

The Extended Integrated Reference Model (XRM) [34] that is being designed and implemented at Columbia University provides a resource management and control systems for multimedia applications over ATM networks. XRM is based on previous work on the Magnet-II testbed and shares with it the restriction to a small number of fixed QoS classes.

Several QoS standardization efforts are being made by several network communities. The ATM Forum recently completed a traffic management specification [7] 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 [13] for signaling. The draft proposal for a *guaranteed service* definition will support deterministic end-to-end delays; However, an implementation is not yet available. Our work will take full advantage of the framework provided by ATM. Also, any output that comes from the IntServ group [49] at IETF will be applicable to our work.

The RT Mach project [35, 61] has built distributed real-time operating system services supported by a guaranteed end-to-end resource reservation paradigm.

The RT Mach paradigm is complemented by a functionally scalable microkernel along with a performance monitoring infrastructure. RT Mach is applicable to hard and soft real-time applications, but the services provided are not intended to scale to large geographical areas.

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 [62] 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.

The Globus [18] project is developing basic software infrastructure for computations that integrate geographically distributed computational and information resources. Globus creates a parallel programming environment that supports the dynamic identification and composition of resources available on large-scale internets, and provides mechanisms for authentication, authorization, and delegation of trust within environments of this scale. Globus emphasizes the importance of heterogeneity and security; however, it does not offer solutions for fault-tolerance and real-time.

BBN's Corbus [68] is a distributed, object-oriented system that facilitates the development of distributed applications. Corbus provides the middleware that closes the gap between QoS offered by real-time operating systems and networks and the communications researchers and object-oriented applications. Corbus is based on CORBA [63] and its object model.

8 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, since there are key research questions that must be resolved, there is risk involved with this approach. Fundamental research questions 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. 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. B. Adelberg, B. Kao, and H. Garcia-Molina, An Overview of the STanford Real-time Information Processor, *ACM SIGMOD Record*, 25(1), 1996.
3. B. Adelberg, H. Garcia-Molina and B. Kao, Applying Update Streams in a Soft Real-Time Database System, *Proceedings of the 1995 ACM SIGMOD*, pp. 245 - 256, 1995.
4. B. Adelberg, H. Garcia-Molina and B. Kao, Database Support for Efficiently Maintaining Derived Data, Technical Report, Stanford University, 1995.
5. T. E. Anderson, D. E. Culler, and D. A. Patterson, A Case for NOW (Networks of Workstations), *IEEE Micro*, 15(1):54–64, February 1995.
6. 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.
7. ATM Forum, *ATM Traffic Management Specification 4.0*, April 1996.
8. ATM Forum, ATM User-Network Interface Specification, Version 4.0, 1996.
9. N. Audsley, A. Burns, M. Richardson and A. Wellings, A Database Model for Hard Real-Time Systems, Technical Report, Real-Time Systems Group, Univ. of York, U.K., July 1991.
10. 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.
11. A. Bondavalli, J. Stankovic, and L. Strigini, Adaptive Fault Tolerance for Real-Time Systems, *Third International Workshop on Responsive Computer Systems*, September 1993.
12. 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.
13. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification, Internet Draft, November 1996.

14. 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.
15. 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.
16. U. Dayal et. al., The HIPAC Project: Combining Active Databases and Timing Constraints, *SIGMOD Record*, Vol. 17, No. 1, pp. 51-70, March 1988.
17. M. Di Natale and J. Stankovic, Dynamic End-to-End Guarantees in Distributed Real-Time Systems, *Real-Time Systems Symposium*, Dec. 1994.
18. I. Foster and C. Kesselman, Globus: A metacomputing infrastructure toolkit, SIAM (to appear), 1997.
19. N. Gehani and K. Ramamritham, Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems, *Real-Time Systems*, Vol. 3, No. 4, December 1991.
20. 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.
21. 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.
22. J.R. Haritsa, M.J. Carey and M. Livny, On Being Optimistic about Real-Time Constraints, *Proc. of 9th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, April, 1990.
23. J.R. Haritsa, M.J. Carey and M. Livny, Earliest Deadline Scheduling for Real-Time Database Systems, *Proceedings of the Real-Time Systems Symposium*, pp. 232-242, December 1991.
24. J.R. Haritsa, M.J. Carey and M. Livny, Data Access Scheduling in Firm Real-Time Database Systems, *The Journal of Real-Time Systems*, Vol. 4, No. 3, pp. 203-241, 1992.
25. 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.
26. J. Huang, J.A. Stankovic, K. Ramamritham and D. Towsley, Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes, *Proceedings of the 17th Conference on Very Large Databases*, pp. 35-46, September 1991.
27. J. Huang, J.A. Stankovic, K. Ramamritham, D. Towsley and B. Purimetla, On Using Priority Inheritance in Real-Time Databases, **Special Issue** of *Real-Time Systems Journal*, Vol. 4. No. 3, September 1992.
28. M. Humphrey and J. Stankovic, CAISARTS: A Tool for Real-Time Scheduling Assistance, *IEEE Real-Time Technology and Applications Symposium*, June 1996.
29. B. Kao and H. Garcia Molina, Subtask Deadline Assignment for Complex Distributed Soft Real-Time Tasks, *Technical Report STAN-CS-93-1491*, Stanford University, Oct. 1993.
30. J. E. Kirkwood, *Sybase Architecture and Administration*, Prentice-Hall, 1993.
31. G. Koch and K. Loney, *Oracle: The Complete Reference*, Mc Graw-Hill, 1997.
32. T. Kuo and A. K. Mok, SSP: a Semantics-Based Protocol for Real-Time Data Access, *IEEE 14th Real-Time Systems Symposium* , December 1993.
33. T. Kuo and A. K. Mok, Real-Time Data Semantics and Similarity-Based Concurrency Control, *IEEE Transactions on Knowledge and Data Engineering*, 1996.

34. A. A. Lazar, S. Bhonsle, and K. S. Lim, A Binding Architecture for Multimedia Networks, In *Proceedings of COST-237 Conference on Multimedia Transport and Teleservices*, Vienna, Austria, 1994.
35. C. Lee, R. Rajkumar, and C. Mercer, Experiences with Processor Reservation and Dynamic QoS in Real-Time Mach, In *Proceedings of Multimedia Japan*, March 1996.
36. 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.
37. 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.
38. Y. Lin and S.H. Son, Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order, *Proceedings of the Real-Time Systems Symposium*, pp. 104-112, December 1990.
39. M. Livny, *DeNet Users Guide*, version 1.5, Dept. Comp. Science, Univ. of Wisconsin, Madison, WI 1990.
40. H. W. Lockhart, *OSF DCE Guide to Developing Distributed Applications*, McGraw-Hill, New York, 1994.
41. E. McKenzie and R. Snodgrass, Evaluation of Relational Algebras Incorporating the Time Dimension in Databases, *ACM Computing Surveys,* Vol. 23, No. 4, pp. 501-543, December 1991.
42. D. Niehaus, K. Ramamritham, J. Stankovic, G. Wallace, C. Weems, W. Burleson, and J. Ko, The Spring Scheduling CO-Processor: Design, Use and Performance, *Real-Time Systems Symposium*, Dec. 1993.
43. H. Pang, M.J. Carey and M. Livny, Multiclass Query Scheduling in Real-Time Database Systems, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995.
44. B. Purimetla, R. M. Sivasankaran, J.Stankovic and K. Ramamritham, Network Services Databases - A Distributed Active Real-Time Database (DARTDB) Applications, *IEEE Workshop on Parallel and Distributed Real-time Systems*, April 1993.
45. K. Ramamritham, Real-Time Databases, *Distributed and Parallel Databases* 1(1993), pp. 199-226, 1993.
46. K. Ramamritham, Where Do Deadlines Come from and Where Do They Go? *Journal of Database Management*, Spring, 1996.
47. K. Ramamritham, J. Stankovic and P. Shiah, Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems, *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184-94, April 1990.
48. K. Ramamritham, J. Stankovic and W. Zhao, Distributed Scheduling of Tasks with Deadlines and Resource Requirements, *IEEE Transactions on Computers*, 38(8):1110-23, August 1989.
49. S. Shenker, C. Partridge, and R. Guerin, Specification of Guaranteed Quality of Service, IETF, Integrated Services WG, Internet Draft, August 1996.
50. R.M. Sivasankaran, J.A. Stankovic, D. Towsley, B. Purimetla and K. Ramamritham, Priority Assignment in Real-Time Active Databases, *The International Journal on Very Large Data Bases*, Vol. 5, No. 1, January 1996.
51. 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.

52. X. Song and J. W. S. Liu, How Well Can Data Temporal Consistency be Maintained? *IEEE Symposium on Computer-Aided Control Systems Design*, 1992.
53. X. Song, Data Temporal Consistency in Hard Real-Time Systems, Technical Report No. UIUCDCS-R-92-1753, 1992.
54. X. Song and J. W. S. Liu, Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 5, pp. 786-796, October 1995.
55. 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.
56. J. Stankovic, K. Ramamritham, and D. Towsley, Scheduling in Real-Time Transaction Systems, in *Foundations of Real-Time Computing: Scheduling and Resource Management*, edited by Andre van Tilborg and Gary Koob, Kluwer Academic Publishers, pp. 157-184, 1991.
57. J. Stankovic, SpringNet: A Scalable Architecture For High Performance, Predictable, Distributed, Real-Time Computing, Univ. of Massachusetts, Technical Report, 91-74, October 1991.
58. J. Stankovic, and K. Ramamritham, *Advances in Hard Real-Time Systems*, IEEE Computer Society Press, Washington, DC, September 1993.
59. J. Stankovic and K. Ramamritham, Reflective Real-Time Operating Systems, *Principles of Real-Time Systems*, Sang Son, editor, Prentice Hall, 1995.
60. J. Stankovic, Strategic Directions: Real-Time and Embedded Systems, *ACM Computing Surveys*, Vol. 28, No. 4, December 1996.
61. H. Tokuda, T. Nakajima and P. Rao, Real-Time Mach: Towards a Predictable Real-Time System, *Proc. Usenix Mach Workshop*, October 1990.
62. F. Travostino and E. Menze III, The CORDS Book, OSF Research Institute, September 1996.
63. S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, *IEEE Communications Magazine*, 14(2), February 1997.
64. G. Wiederhold, Mediators in the Architecture of Future Information Systems, *IEEE Computer*, Vol. 25, No. 3, March 1992, pp. 38-49.
65. D. E. Wrege, E. W. Knightly, H. Zhang, and J. Liebeherr, Deterministic Delay Bounds for VBR Video in Packet-Switching Networks: Fundamental Limits and Practical Tradeoffs, *IEEE/ACM Transactions on Networking*, 4(3):352–362, June 1996.
66. 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.
67. 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.
68. 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.

This article was processed using the LaTeX macro package with LLNCS style