The Cogency Monitor: An External Interface Architecture For a Distributed Object-Oriented Real-Time Database System

John A. Stankovic, Sang H. Son, and Chi D. Nguyen
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

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. The BeeHive system however, must be able to interact with existing databases and import such data. In this paper, we present a high level design for this external interface and introduce the concept of a cogency monitor which acts as a gateway between BeeHive and the external world. The cogency monitor filters, shapes, and controls the flow of information to guarantee specified properties along the real-time, fault tolerance, quality of service, and security dimensions. We also describe three implementations of the cogency monitor and present some preliminary performance results that demonstrate the value of the cogency monitor.

Key Words: Real-time, object-oriented, database security, fault-tolerance, quality of service

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 for audio and video, 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 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* [15], (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 [5], real-time performance and security [13],
- 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,
- 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 [10], and
- new real-time QoS scheduling, resource management and renegotiation algorithms [9].

One of the interesting issues that is encountered is how to interact with the external Internet world. There are many sources of data that are publicly available and for many enterprises this data should be accessed and incorporated into their own enterprise database. In our case, the external data is incorporated into BeeHive and we support new functionality in accessing the data along real-time, fault tolerance, QoS, and security dimensions. Common examples of external data are weather information and stock market price quotes. Additionally, there may be external sensors and actuators that contain vital information, which needs to be monitored and stored within BeeHive. In the remainder of this paper we present a brief overview of the BeeHive system and discuss the issues concerning the external interface. We then present the cogency monitor architecture, describe its implementation, and provide some performance results.

2. BeeHive System

2.1. Brief Overview For Perspective

BeeHive is an application-focussed distributed database system. For example, it could provide the database level support needed for information dominance 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 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 open Internet systems.

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 real-time, QoS, fault tolerance and security APIs), and underlying novel object support. 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.

2.2. 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 policies. Our BeeHive object model (BOM) extends traditional object-oriented data models by incorporating semantic information regarding real-time, fault-tolerance, importance, security, and quality of service 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 briefly describe each component of the BOM. The BOM has some similarity in terms of the structure of objects to the RTSORAC object model [11]. 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. The reason for presenting the BOM in this paper is to give a flavor of how objects are represented internally in BeeHive. Any external data, that is to be imported into BeeHive, must adhere to the BOM and must map to this uniform view.

A BeeHive object is modeled as a tuple <N, A, M, CF>, 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, and CF is a compatibility function that determines how method invocations can coexist. The data stored in an object can have temporal consistency specified by a validity interval. A value and a validity interval represent the value part of each attribute. 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. In addition to the structure, all objects that belong to the object class can be accessed from it. Complex objects, object identity, encapsulation, class hierarchies, overloading, and late binding can be fully or partially supported.

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 worst case execution time 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 (e.g., 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:

$$CF(M1, M2) = < \textit{Boolean Expression} >$$

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.

The implementation of a native BeeHive node is currently underway and is being built using the Shore object store from the University of Wisconsin.

3. The BeeHive External Interface

3.1. Overview

BeeHive as a database must be able to interact with the vast resources available from open information sources. This data could be from other databases, from the World Wide Web, and from sensors that are important to Beehive. The main objective of the BeeHive external interface is to be able to handle and retrieve such data in such a manner as to:

1. Retrieve the data as quickly as possible to minimize temporal staleness.
2. Guarantee specified properties along the dimensions of real-time, QoS, fault-tolerance, and security.
3. Add value to the data by filtering and possibly integrating it into BeeHive.

The "data" that is external to BeeHive is extremely heterogeneous in content and must be handled accordingly. From the viewpoint of BeeHive the interactions with external databases are query-centric. Additionally, the query mechanisms to retrieve this data range from standard SQL queries to information retrieval search methods used by the WWW search engines. In order to deal with this heterogeneous data we have chosen to classify the data into three distinct categories (1) structured (2) unstructured and (3) raw data. We believe that this taxonomy of data is logical, consistent and complete with respect to the types of data that are available. Additionally, segregating data into these three categories allows for an easier implementation of our external interface.

Structured data is data pertaining to a collection of objects that have been returned from a database. The interface to this type of data would be through some type of query language like SQL and through some form of database connectivity such as ODBC or JDBC. Structured data is indexed and usually returned as unique tuples for a relational database or objects for object oriented databases. Tuples or objects can be readily transformed into an object, which means it is easier to integrate into BeeHive.

The major hurdle involved in inserting structured data into an object-oriented database is that the schema or object definition must be previously defined or available. The other issue is that not all of the attributes of the object are present in the data (e.g., the query might return the projection of a schema or class type) and thus there must be some method of dealing with these "null" attributes. This implies that whatever method is used for integrating the structured data into BeeHive requires some semantic information about the actual data that is being received.

Unstructured data is data pertaining to a collection of objects that is not indexed. Unstructured data is the results of queries from common search engines, such as Infoseek, and Altavista. Other forms of unstructured data are frequently asked questions (FAQ) listings, usenet, etc. Unstructured data is hard to deal with since the mapping of the data to an object, which can be integrated into an object oriented database, is not well defined. The data is awkward to deal with and is extremely diverse ranging from text files to multimedia clips. Often there is not a direct mapping of data to tuple or object as in structured data. For example, do we want to insert the entire HTML file as an object or are we to insert only the hyper-links? To make matters worse the query mechanisms used by information retrieval engines are rarely repeatable; it is often the case that the same query to an Internet search engine will return different results. The results of unstructured data queries are thus ranked in terms of "relevance". This ranking that is provided is not always accurate and this leads to the question of what data is of interest and what data should be entered into BeeHive. In order to deal with such data one must have some semantic information to make these decisions, similar to the case of structured data. An important difference here is that some form of filtering must be done on unstructured data prior to integration into BeeHive. This filtering performs the mapping of data to object that was previously mentioned.

Lastly, raw data is data that inherently pertains to only one object. This would include sensor data, individual web pages, video clips, files, etc. Similar to unstructured data, raw data is very diverse in nature, but in this case there is usually more of a direct mapping between raw data and an object. There is still the need to perform some filtering on the raw data however, similar to unstructured data. The difference here is that raw data would be periodic in nature as in the case of sensor data. The issues that are involved with handling this data include how often to poll the sensor for new data and how much of a buffer to keep prior to integration into the database. (Polling data too frequently leads to wasted resources since it may not be required if the temporal validity of the data is larger than the polling rate.)

3.2. The Cogency Monitor

The key concept about the different types of data is that once we have the data it must be verified and shaped prior to its incorporation into BeeHive. (By verified we mean that the data meets the real-time, fault tolerance, quality of service, and security characteristics that the user specified.) At the very least since BeeHive is an object oriented database we must be able to transform this data into unique objects. Our overall view of the BeeHive external interface is shown in Figure 1.

The heart of our architecture is the cogency monitor. Any BeeHive node can execute one or more cogency monitors. A cogency monitor can (i) support value added services that are explicitly specified, (ii) monitor the incoming data for “correctness” and possibly make decisions based on the returned data, (iii) execute client supplied functionality, and (iv) map incoming data into BeeHive objects. The term cogency is used since cogency means validness, strong argument, ability to forcefully compel or constrain; this is exactly what we would like the external interface to do.

The planned value added services include real-time, fault tolerance, QoS, and security features and are shown in Table 1. This is by no means an exhaustive list; as we gain more experience with building cogency monitors more services will be added.

The cogency monitor checks correctness by verifying that incoming data should be entered into BeeHive. This aspect is a form of firewall. The cogency monitor can execute general functions provided by clients such as queries, transactions, or workflows. The cogency monitor needs semantic information that the user provides so that the external data can be checked, controlled, and transformed into BeeHive objects properly. The semantic information can be supplied as schema definitions.

The cogency monitor can have little to no functionality, for efficiency, or it is able to perform processing and filtering on data prior to integration into BeeHive. The cogency monitor utilizes its own files that exist outside BeeHive as a useful technique to support many of its capabilities.

Our use of a cogency monitor assures an architecture that is extensible and flexible and that can be used with or without BeeHive. We have identified several key services that a cogency monitor can provide as shown in Table 1. The vision is to allow users to mix and match the capabilities available from our library to construct a cogency monitor that gains this added value and that is precisely what is required by a client, and no more. The full design of the GUI is being developed.

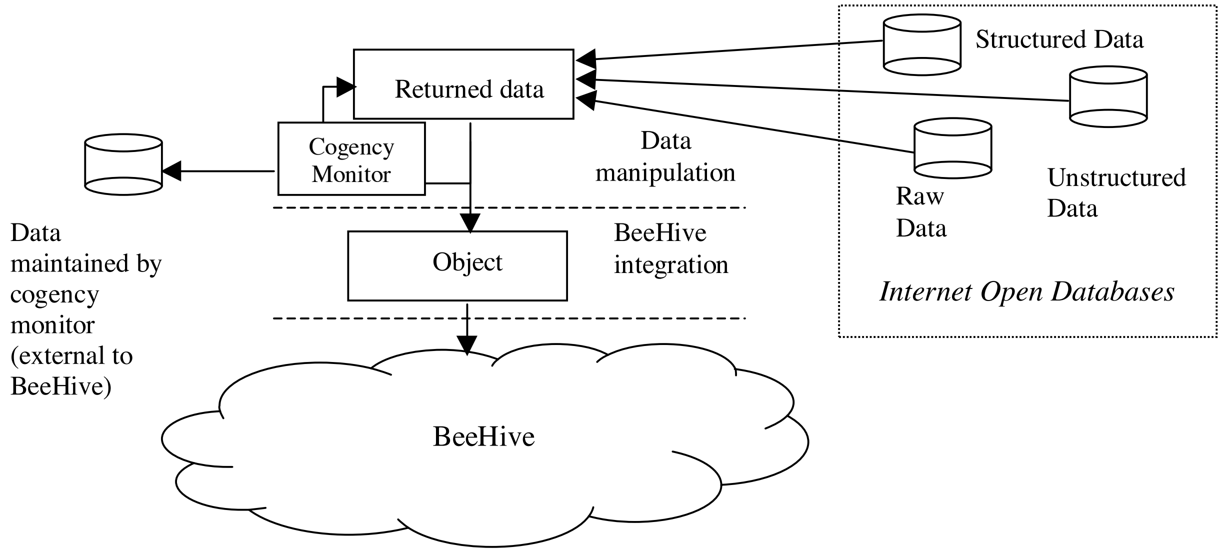

Figure 1: *BeeHive External Interface Architecture*

Research questions still must be resolved including how to ensure compatibility in automatically linking and loading the various features found in Table 1. Our current idea is to use a compatibility function that identifies valid combinations of services.

A cogency monitor is not a firewall or an intelligent agent. A firewall [6] is a device that secures a connection between a trusted network and another network. A firewall might simply filter packets based on headers or include a proxy that acts to limit operations, keep an audit trail, or apply security based rules. Intelligent agents [4] are proxies that try to provide some value added service in very general ways, perhaps even learning or using inference. Our cogency monitor is different from firewalls in that it extends significantly beyond firewalls, but is more focused on specified behavior than an intelligent agent is. In particular, cogency monitors add real-time, quality of service, fault tolerance, and security as well as allowing user-added transformations and mappings into BeeHive objects. The philosophy of the cogency monitor is that which is explicitly specified along the four dimensions is guaranteed, but nothing else. Of course, if a user tries to specify a requirement that is not feasible, then it is denied and no such cogency monitor is created.

4. Experiments

We have implemented three simple cogency monitors, one for each of the three data source types: structured, unstructured, and raw, respectively. We have chosen these three cogency monitors to demonstrate that our architecture is applicable and valid for all three types of data. In each implementation we demonstrate some of the value added services that were presented in Table 1. For the structured data type we interfaced with a small relational database (mSQL) and developed some real-time (timeout with old data, timestamp, monitor response times), and fault tolerance (return previous/old results) properties. For the unstructured data type we have implemented a meta-searcher that possesses fault tolerance (primary and N backup sites, return previous/old results) and real-time (timestamp) properties. And lastly, for the raw data types we have implemented a cogency monitor that will "add functionality" by filtering raw data for the user.

4.1. Structured data cogency monitor

Assume that a BeeHive user requires external data and that his query result needs to be returned within a pre-specified deadline. If this deadline is not met then the user is willing to settle for a result that is temporally stale. He would also like response time monitoring support to help ascertain the typical response time for his query. The user should then be able to use a GUI to collect these features from a library and create a cogency monitor for this particular set of requirements. While we do not yet have a general GUI, we implemented this particular cogency monitor.

An example of where this cogency monitor is useful is aboard a US Navy carrier. The database might contain intelligence information about unfriendly forces that are of interest (vessel type, position, capabilities, etc.) At present the carrier is more interested in an unfriendly ship that is in the nearby vicinity. As such the commanding officer wants to get data about this

Real-time features	Fault tolerance features	Quality of Service features	Security features
✓ Real-time timeout ✓ Periodic activation ✓ Start time ✓ Return partial results ✓ Timestamp incoming data ✓ Broadcast in parallel to multiple sites for faster response ✓ Monitor response times (and/or adjust deadlines dynamically)	✓ Retries ✓ Return previous/old results ✓ Primary and N backup sites ✓ Filters (check returned values for reasonableness) ✓ Keep history log	✓ Reserve bandwidth ✓ Compress data ✓ Keep multiple resolution versions of data	✓ Encryption ✓ Access control ✓ Maintain a firewall (place cogency monitor external to BeeHive) ✓ Keep audit trail

Table 1: *Cogency monitor value added services*

unfriendly ship with timely and fault tolerant guarantees. If the commanding officer cannot get the most up to date information, he will settle for stale information, but he must have the information in time.

The cogency monitor consists of a daemon process that queries the mSQL database periodically at a predetermined rate. The data that is returned is stored locally on the client side and timestamped. (This allows the commanding officer to have a local copy of the data at all times.) To handle a client request the cogency monitor attempts to connect to the mSQL database. If the deadline is reached prior to getting a response the cogency monitor returns the most recent result that was retrieved by the daemon process. By varying the periodic rate at which the daemon queried the mSQL database we are able to evaluate how "stale" the local data is relative to a given query.

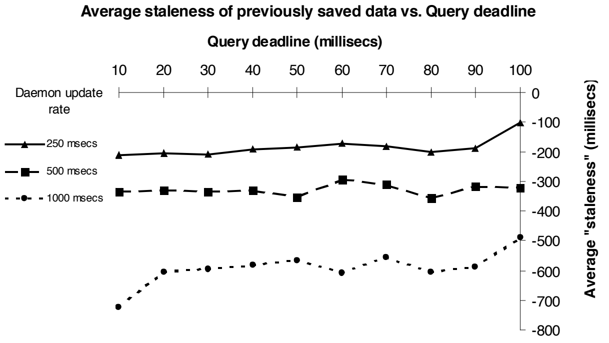

Figure 2: Average staleness of previously saved data vs. Query deadline

Figure 2 plots the average staleness of previously saved data versus the query deadline for different daemon update rates. It is important to note that the staleness of the data is bounded by the daemon update rate in milliseconds. This is important since it gives an upper bound of how stale the data can be. For our navy carrier example, if the daemon update rate is set at 250 milliseconds, then the captain can be assured that the data that he gets back will not be stale by more than 250 milliseconds. And on average the staleness will be approximately be 200 milliseconds as explained below.

The mean and standard deviations for the different update rates are given in Table 2. The average time it took to return a query result from the mSQL database was 75 milliseconds. Using this fact it is easy to explain the mean staleness results obtained. For example the expected staleness for a daemon update rate of 250 ms would be -(250/2 + 75) = -200 milliseconds. This value agrees with the actual value of –185 milliseconds. Figure 3 shows the average staleness over all queries. As can be expected there is a sharp decrease in staleness at a deadline of 75+ milliseconds, since deadlines larger than this would have results provided directly from the mSQL database instead of a previously stored result.

Daemon update rate (msecs)	Mean staleness (msecs)	Staleness standard deviation (msecs)
250	-185	32
500	-329	19
1000	-591	58

Table 2: *Mean and Standard deviation for staleness for different daemon update rates*

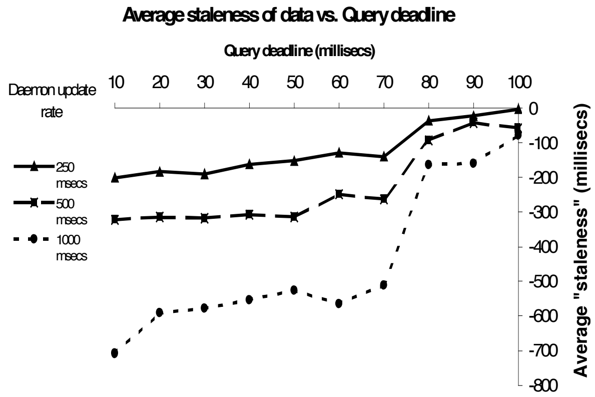

Figure 3: *Average staleness of data vs. Query deadline*

The results of the experiments on our cogency monitor are promising since it allows data to be retrieved with a known staleness by changing the daemon update time accordingly. The cogency monitor has added value by providing a soft guarantee of how long a query should take. An important implication of this result is that the cogency monitor can be used to provide the client with an expected query execution time. This allows for feedback to be provided to the client concerning the average time it takes to return a query result. The client using this information is able to set a reasonable deadline time.

In summary, the client now has a cogency monitor that accesses a remote database with some real-time and fault tolerance properties.

4.2. Unstructured data cogency monitor

For this experiment, assume that a user requires that a query sent to three unstructured remote data sources are to be returned within a pre-specified deadline. As in the structured experiment, the user is willing to settle for temporally stale data, however for this experiment there is more than just one site that can be queried. The scenario that we envision for this cogency monitor is a stockbroker who requires a stock price within a prescribed amount of time. The stockbroker will settle for old data, as long as he knows exactly how old it is. (The cogency monitor that we describe below actually connects to general information search engines instead of stock market search engines, but the two are logically equivalent.)

The setup for the unstructured data type cogency monitor is very similar to the structured cogency monitor. The differences are (1) the cogency monitor now queries three different sites for the data and (2) the cogency monitor does not have a daemon that periodically queries the information sources (network etiquette requires that we do not flood the Internet search engines).

The client is a WWW graphical browser (Netscape, Internet Explorer) which submits a common gateway interface (CGI) query to our cogency monitor with a given deadline. The cogency monitor passes this query to three Internet search engines (Infoseek, AltaVista, and Yahoo) for their response. This setup is similar to other meta-searchers like MetaCrawler [12], in that we are collating data from several other search engines versus actually querying our own data locally. The cogency monitor waits for the returned results from the search engines. If a result is returned prior to the time deadline, the cogency monitor timestamps the HTML file and presents it to the client. Additionally, the cogency monitor saves the timestamped file locally. If however, the result is not received from a search engine prior to the deadline, the cogency monitor will try to return a previously saved copy of the results if it is available locally. If this is not possible, the cogency monitor will inform the user that no result is available.

What we are trying to demonstrate with this cogency monitor is not that BeeHive is a meta-searcher like the ones that are prevalent on the WWW. We are not competing with such groups and their tools. As a matter of fact, we would like to leverage their technology. It is very easy to have the cogency monitor use MetaCrawler as one of the search engines that it will query. What the cogency monitor is demonstrating is the added value that could be obtained from unstructured data dealing mainly with real-time and fault-tolerance characteristics. Our cogency monitor is able to select many different sites in order to retrieve the desired information and it provides a previous result if possible. (This follows the old fault-tolerant adage of “old data is better than no data”). Additionally, this cogency monitor is able to provide a bounded time limit for which a user has to wait for a response through the use of a deadline. The use of different search engines for fault tolerance is acceptable since these information retrieval techniques, as mentioned before, seldom return the same results. Thus we argue that any result provided by one of the search engines is just as likely to produce “valid” results.

4.3. Raw data cogency monitor

Another key capability of the cogency monitor is the ability to attach user supplied functions. In this third implementation we apply this capability to raw data. Our raw data cogency monitor is a simple prototype for searching web pages on the Internet. The purpose of this experiment was to demonstrate how a client would be able to add functionality to raw data through the use of a cogency monitor. The cogency monitor filters incoming data based on a query provided by the user. The client as in the unstructured example, is a WWW graphical browser that submits a CGI query to our cogency monitor along with a specified URL. The cogency monitor then searches the specified HTML page residing at the URL and also recursively searches links from that URL for the keywords that the client entered. The cogency monitor only recursively checks links to a maximum depth as selected by the client. For, example this cogency monitor could be used for retrieving HTML pages concerning Bill Clinton at the CNN website.

This cogency monitor demonstrates the added value that could be obtained for the user. The user initially only has a piece of raw data, the CNN URL, but now the cogency monitor returns information containing unstructured data that pertains to semantic information (his query). In addition to the added functionality the client could have also specified real-time, fault-tolerance, quality of service, or security requirements.

5. 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, quality of service for audio and video, and security properties in a heterogeneous environment. In the research area 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, and security. There have been even fewer that have dealt with trying to incorporate external data into a database with these (real-time, fault tolerance, quality of service, and security) characteristics. Due to space limitations we briefly describe just one of these projects.

The QuO architecture [16] 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 help distributed applications be more predictable and adaptive even if end-to-end guarantees cannot be provided. It addresses the issues to support QoS at object level, such as synthesizing information about system properties and providing a framework to support code reuse. 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.

6. Summary

We have presented the design of BeeHive at a high level and have identified the external interface as an important piece of the BeeHive architecture. We have described an architecture for the external interface that is powerful and flexible. Our novel concept of using a cogency monitor as the gateway between the external world and BeeHive provides a clean and standardized way of dealing with heterogeneous data. A cogency monitor does not rely on cooperating information services as some other projects do and as such BeeHive can leverage off of those projects. The most powerful argument for a cogency monitor however, is that it adds value to and control on the data, rather than just passing it on to BeeHive. The value that we are mainly concerned with pursuing deals with real-time, fault-tolerance, quality of service for audio and video, and security. Our future work will entail expanding the services (see Table 1). We are also developing a standard graphical user interface that will enable BeeHive users to create their own cogency monitors using standard libraries that we provide.

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] M. Balabanovic, An Adaptive Web Page Recommendation Service, *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[4] R. J. Bayardo Jr., W. Bohrer, and et. al., InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments, *ACM SIGMOD*, 1997.

[5] A. Bondavali, J. Stankovic, and L. Strigini, Adaptable Fault Tolerance for Real-Time Systems, *Third International Workshop on Responsive Computer Systems*, Sept. 1993; full version *Proc. PDCS*, September 1993.

[6] W. Cheswick and S. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*, Addison-Wesley Publishing Company, 1994.

[7] T. Finin, R. Fritzson, D. McKay, R. McEntire, KQML as an Agent Communication Language, Proceedings of the Third International Conference on Information and Knowledge Management, *ACM Press*, November 1994.

[8] J. Hammer, H. Garcia-Molina, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom, Information Translation, Mediation, and Mosaic-Based Browsing in the TSIMMIS System, *ACM SIGMOD International Conference on Management of Data*, San Jose, CA., June 1995.

[9] H. Kaneko, J. Stankovic, S. Sen and K. Ramamritham, Integrated Scheduling of Multimedia and Hard Real-Time Tasks, *Real-Time Systems Symposium*, December 1996.

[10] 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.

[11] 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.

[12] E. Selberg, and O. Etzioni, The MetaCrawler Architecture for Resource Aggregation on the Web, *IEEE Expert*, January/February 1997.

[13] S. H. Son and C. Chaney, Supporting the Requirements for Multilevel Secure and Real-Time Databases in Distributed Environments, *Annual IFIP WG 11.3 Conference of Database Security*, Lake Tahoe, CA, Aug. 1997, pp. 57-71.

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

[15] 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.

[16] 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.