A high performance Distributed Database System for enhanced Internet Services

A.J.H. Peddemors, L.O. Hertzberger

University of Amsterdam, Department of Computer Science
Computer Architecture and Parallel Systems Group
Kruislaan 403, 1098 SJ, Amsterdam, The Netherlands
e-mail: {arjan,bob}@wins.uva.nl

Abstract. Using a distributed database system as part of a distributed web server architecture has obvious advantages. It is shown that a first phase distributed database system can be build by extending an existing object oriented database system with application specific additions. A web database is implemented, as part of a traditional HTTP based distributed web server, using this distributed database system.

1 Introduction

Users of the Internet impose increasing demands on network bandwidth, web server performance and reliability of services. The number of new users and different kinds of services is still increasing every day as well. In only a few years time, the Hypertext Transfer Protocol (HTTP) [1] based World Wide Web has become the main service bypassing older services like ftp, e-mail and telnet. As the Internet is maturing, it is very likely that new services, for instance audio and video streaming, telephony, secure payment transactions and news casting, will make the spectrum more diverse and may even challenge the current WWW dominance. Next to that, it is also likely that the definition of the WWW services changes over time: originally, the WWW was strictly bound to the HTTP, but nowadays most web browsers support the addition of components for dedicated services, like audio streaming, and are able to run Java applets. This enables web clients to use a wide range of protocols including UDP and TCP itself. Consequently, the distinction between services is fading.

The current ad hoc web server architecture is hardly able to meet the increasing demands. The traditional way of accessing data on the web is through a web server machine running an HTTP daemon that reads the data from the local file system, which was later enhanced with the Common Gateway Interface (CGI) [2]. The CGI standard defines an interface between external gateway elements and the HTTP daemon. While the HTTP daemon retrieves static data, such as an HTML document, from the file system, a CGI element is invoked every time it is addressed by a web client, allowing for dynamic data generation and database access. This architecture does have some inherent inefficiencies:

1. **lack of data management:** As data on the web server becomes more complex and inter-dependent, data management gets more important. Take for instance a set of HTML documents pointing to each other: if one document is removed, the referring

documents must be updated as well. In this case, storing data on the file system does not enforce HTML link consistency.
2. **CGI is stateless:** No state information is preserved while making calls to CGI elements. Suppose a CGI element is used to connect to a database module and suppose that a web client makes multiple transactions on the same database module. Then it is necessary to connect, do authentication and disconnect with every transaction separately.

The standard implementation of this architecture has the following drawbacks:

1. **CGI diminishes performance:** Normally, CGI elements are implemented in such a way that for every request sent to them a new process is created. From the underlying operating system point of view, this is costly.
2. **no scalability:** No specific provisions are made for parallelism to enable scalability.

Currently, the usage of frameworks supporting the implementation of distributed object oriented Internet applications, such as the Common Object Request Broker Architecture (CORBA) [3], [12] and Java's Remote Method Invocation (RMI) [11], is becoming more and more accepted. Such frameworks would enable a web client to conveniently connect to, for instance, database server objects, thereby bypassing the CGI mechanism. In general, however, traditional HTTP based web servers do not fit well the client server paradigm those frameworks rely on, making them less suitable to host (server) objects for a distributed object oriented application. A sign that a framework like CORBA is gaining momentum is the inclusion of a Java based Object Request Broker (ORB) in the latest release of the Netscape Communicator web browser.

Given the previous considerations, an object oriented, high performance, scalable and fault tolerant database system embedded in the web server architecture may dissolve some of the inefficiencies associated with the traditional web server, especially the lack of data management and scalability, and may enable to support the employment of distributed object frameworks.

This paper describes the first phase realization of a distributed database system as part of a number of possible next generation web server architectures. An iterative process is followed to build the distributed database system. Every phase in this building process has a clearly defined set of objectives, spans a limited amount of time, adds functionality, and the output of every phase serves as input for the next phase. The first phase will implement a minimum level of functionality, concentrating on inter-node communication and not on general distributed transaction and query management mechanisms. This paper starts with a brief discussion on possible parallel web server architectures in which an object oriented database can be used, followed by a description of the first phase distributed database design and implementation.

2 Distributed Web Server architectures

It is expected that future WWW services will not only be based on HTTP but also on a framework that supports the deployment of distributed object oriented applications in a wide area network such as the Internet. Therefore, at this point, two approaches are discerned to introduce parallelism in the web server architecture: take the traditional HTTP based server and add parallelism or make a shift to the object oriented paradigm and use the concept of distributed objects. Both approaches can benefit from an embedded distributed database system.

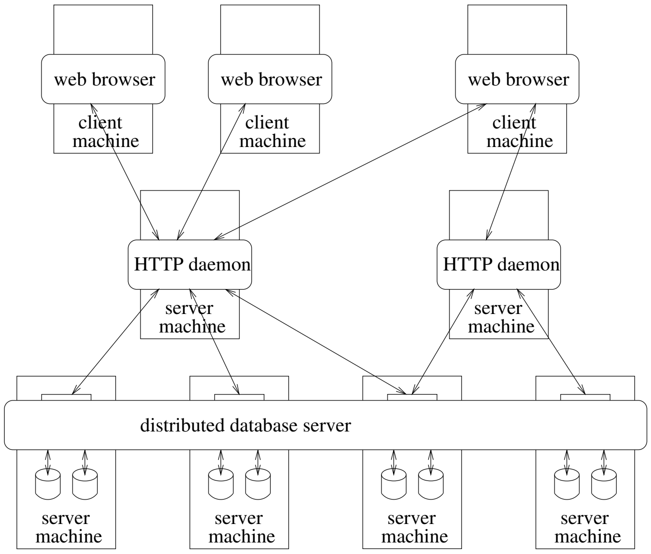

Fig. 1. Distributed Web Server - Traditional

Figure 1 depicts a possible traditional distributed web server. It consists of one or more front end machines running HTTP daemons and one or more database machines running the distributed database system. The server machines are connected by a fast network. The front end machines may be extended with a caching system. The HTTP daemon uses an addition (plug-in) to conduct file system access to the database system. CGI executables are run on the front-end machines. A web cache may be added to hold frequently requested data. Load distribution of incoming HTTP requests over the front-end machines may be realized by means of the Domain Name Service (DNS) round-robin lookup mechanism. This mechanism traditionally has been used to add scalability to a web server (see [13]).

Figure 2 shows a possible object oriented distributed web server. In this setup, client objects connect to server objects by means of some underlying object framework like CORBA. Instances of server objects exist in the distributed object runtime environment available on the server machines. Server object persistence can be realized by the distributed database system. For performance reasons, multiple instances of the same server object may exist; the runtime environment takes care of balancing incoming requests over the instances.

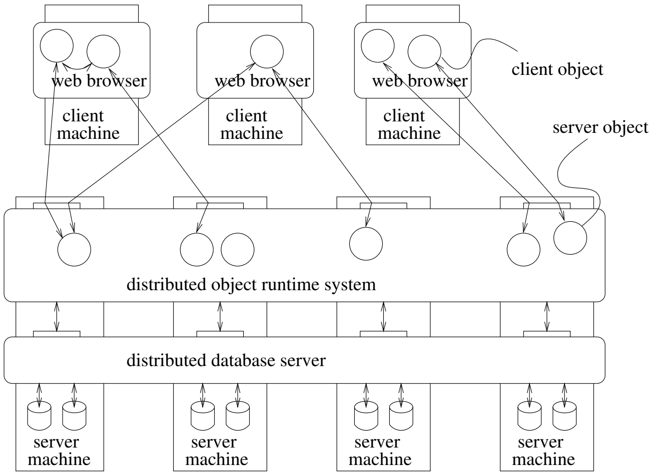

Fig. 2. Distributed Web Server - Object oriented

3 Requirements

The objectives for the distributed database system in general are to make it specifically suited to be embedded in a high-performance and scalable web server architecture. The web server may consist of a heterogeneous set of machines. Database clients must be able to connect to a single node of the distributed database system to execute all its database actions. This allows a client to go only once through the process of setting up a connection with the database and it reduces the total number of connections to the database. The following requirements are set for the object oriented distributed database system:

1. **portability:** In order to allow for a heterogeneous set of server machines, the database must be able to run in a heterogeneous environment. This means that the database must be portable to a wide range of platforms.
2. **multi-threaded node server:** The distributed database must consist of a number of nodes each running a node server. The node server must be multi threaded to allow for Symmetric Multi Processor (SMP) kind of scalability.
3. **full database functionality per node server:** Every node server must offer all the database functionality. This allows clients to connect to a single arbitrary database node to perform all its actions. It also allows for load balancing by spreading connections over the available node servers.

For the first phase, a distributed database system offering an adequate level of functionality in order to be able to use it in a traditional distributed web server (as described in the previous section) must be build.

Given these requirements, the obvious way to go is to select an existing object oriented database system and extend it to a distributed system. This allows for the realization, within limited time, of enough functionality to use it in a traditional distributed web server while keeping the implementation effort limited. One of the key elements in such a setup is a component that is specifically suited to support efficient and high speed inter-node communication. Using a distributed object computing framework like CORBA for this task is considered, for current CORBA implementations, too inefficient (see [14]), and therefore must be implemented here. The Matisse database system [4] was chosen because of its larger emphasis on high performance and scalability compared to other object oriented database systems and its good match with the requirements stated above.

4 Matisse

The Matisse database system is based on the client-server paradigm: client applications connect via a network to the server, where the data is stored, in order to perform actions like object search, data retrieval, data updates, etc. The server, called MTS, supports these actions with a relatively simple set of possible operations on a limited number of server data structures, as defined by the Server Engine Services (SES) API [5]. A client application may need services at a higher level, for instance services which allow for Object Oriented database access. These services are provided by a number of Matisse client libraries and associating APIs. The most important one is the Object Oriented Services (OOS) C API [7], [8] extended with a C++ binding [9] and an SQL interface [10]. The client libraries use the low-level server interface to implement their services.

Matisse has, amongst others, the following features:

1. **multi-threaded server:** For every client connection, a separate thread is started. Also, writing to separate disks is handled by separate threads, which implements I/O parallelism.
2. **multi platform server availability:** The server and client APIs are available for a wide range of operating systems including Microsoft Windows NT, Sun Solaris and IBM AIX.
3. **server extensibility:** A special API, called OpenMTS [6], enables the creation of extensions to the server. This facilitates the implementation of additional functionality needed for distributed database system. It is a low level API: no object oriented services like relationship and attribute management are available.
4. **high and low level client side APIs and language bindings:** Different client APIs allow to access the server, as described above, at different levels in different programming languages.
5. **fault tolerance:** The server is able to replicate objects over different disks: when a disk crashes, no data is lost.

5 Design

The main idea behind the design is to run an extended Matisse database server as a node server in the distributed database system. The extension must offer functionality to let the node server cooperate in such a way that together they form a distributed database system. This means that the extension must offer a communication mechanism.

The Matisse server is extensible but was not otherwise designed for a distributed environment. An object is referred to by an object identifier (oid) but each server assigns object identifiers to new objects itself, which results in globally non-unique object identifiers. It is necessary, however, that each object in the distributed database has a unique object identifier if the generic server interface must be preserved (i.e. a client cannot tell the difference between connecting to a distributed or a normal system; all higher level APIs can be used). When generic interface preservation is not an issue, an application specific interface may be constructed which reckons with non-global object identifiers. In either case, there is a need for an efficient communication mechanism between the nodes of the database.

For the first phase, it was decided that the distributed database will be based on application specific server extensions on top of OpenMTS dealing with non-global object identifiers. Compared to globally unique object identifiers, it has a number of advantages and disadvantages.

Advantages:

1. No need to change the Matisse server core: implementation effort will be limited. This also delivers hands-on experience and performance measurements within reasonable time.
2. A dedicated server interface increases performance, because it can be designed to reduce client/server network traffic and to do query processing on the server. This makes it especially suitable for high performance web applications like search engines.

Disadvantages:

1. The generic server interface is not usable.
2. No object oriented services are available: OpenMTS is a low level API.
3. For every application, a new server interface must be written.

The design of the distributed database system is depicted in figure 3. A client is granted full database access by connecting to a single server, the so called entry server. Each server in the system can perform the role of entry server for the clients that are currently connected to it. The communication handler takes care of efficient communication between the nodes. It supports broadcasting requests and gathering of replies over the high speed network connecting the nodes. From an entry server point of view, all other servers are so called internal servers. This means that each server in the system also performs the role of internal server.

Suppose a server extension defines a read operation and data may reside on any single node in the system. The following actions take place on the entry server when a client reads an object:

1. The client connects to the entry server.
2. The client sends the read request to the entry server.
3. The entry server identifies the request as a read request.
4. The read request is broadcasted to all other nodes (if necessary, the object may be available locally) by the entry server using the communication handler.

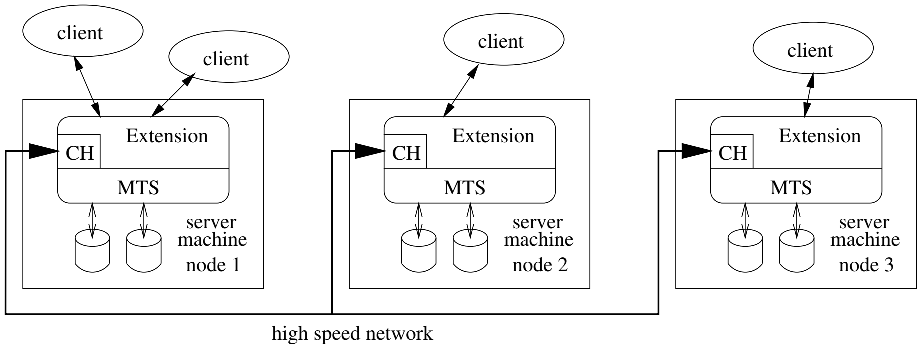

Fig. 3. First phase distributed database system: clients connect to an application specific extension to the Matisse server (MTS). Part of the extension is a communication handler (CH) supporting communication between distributed database nodes.

5. Reply data is gathered by the entry server using the communication handler.
6. The entry server sends the object data to the client.
7. The client sends the next request or closes the connection.

The internal server uses a dispatch mechanism offered by the communication handler to process incoming broadcast requests from entry servers. For the read request, this would look like:

1. The entry server sends the read request.
2. The internal server identifies the request as a read request.
3. The internal server tries to read the object locally.
4. The result is returned to the entry server.

Obviously, multiple clients may read objects at the same time connecting to any server in the system. This means that the network between the nodes is used for multiple read actions at the same time. Therefore, the capacity of this 'internal' network imposes a maximum throughput restriction. From a throughput perspective, two other parameters are important as well: the maximum throughput of the network to the outside world (to the clients) and the maximum throughput of all the disks in the system together. These three parameters should be in balance. The expectation is that the system is in balance for reading when all three throughput parameters have approximately the same value. Measurements must indicate if this is valid.

In general, client requests may involve sending back data from multiple internal servers to the entry server. In that case, broadcasting the request to all the internal servers is likely to be the most efficient mechanism. For client requests such as the object read action described above, where only one internal server sends back data, broadcasting the request may be expensive. Being able to determine to which relevant nodes a client request must be sent, however, means administering which client requests are important to which nodes. This introduces a considerable amount of complexity that would hamper a straightforward implementation.

6 Implementation

The server extension is application dependent: every application using the distributed database imposes its own functional requirements on the extension. Two different extensions were implemented. A simple object store, called MPPCRUD, allowing for create, read, update and delete actions on database objects associated with a unique name is used for testing and performance measurements. It is based on Matisse CRUD. A simple web object store (WEBDB) is used in a first phase parallel web server. The same communication handler is used for both extensions.

The Matisse server is multi threaded. This means that every part of the server extension must be thread safe, including the network layer used by the communication handler. Figure 4 shows the main threads in the extended server involved in handling a request that cannot be handled by the entry server. The client listener and client request handler threads are started by the Matisse server; the node listener and internal request handler threads are started by the communication handler. The following actions take place while handling the request:

1. The client connects to the entry server (extended server 1).
2. The client listener thread starts a client request handler thread to handle requests for the connected client.
3. The client sends a request.
4. When the client request handler cannot handle the request locally, the request is broadcasted to the internal servers (extended servers 2 and 3).
5. The node listener thread on the internal server puts the client request in a request queue.
6. The internal request handler thread reads the request from the queue and sends back the reply to the client request handler on the entry server over a channel. A channel represents a network connection resource used to send data.
7. After gathering all replies from the internal servers, the client request handler sends back a reply to the client.

A node listener thread accepts requests send by another server (resulting in n - 1 node listener threads per server process, where n is the total number of nodes) and inserts them in a single request queue. A configurable number of worker threads, the internal request handlers, read requests from the queue and handle it.

The distributed database system is implemented on a Parsytec CC Massively Parallel Processor (MPP) machine on AIX. This machine features a proprietary internal high speed network (HSlink, 33MBps node-to-node sustained throughput) and eight i/o nodes. Every i/o node runs AIX and Parsytec's EPX. The communication handler is implemented using TCP/IP on a 10Mbps Ethernet and using MPI on the HSlink. Current MPI implementations, however, are not thread safe. Therefore, the MPI communication handler can not be used in the extended server. Work on a thread safe implementation of MPI is in progress.

The traditional distributed web server configuration consists of a dual-processor Sun UltraSPARC SMP as front-end machine and the Parsytec CC as distributed database server machine running WEBDB. The front-end runs a Netscape Enterprise HTTP daemon (multi-threaded) with a dedicated plug-in that is used to redirect file access to the

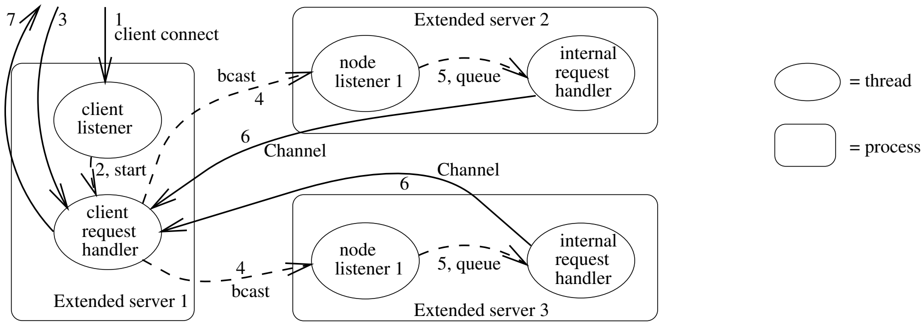

Fig. 4. Main threads within the extended server.

distributed database. This plug-in uses the WEBDB server extension to retrieve objects associated with the URI of the request. Retrieval of these objects is, from the plug-in point of view, transparent, i.e. the plug-in does not need to know at which node the object is stored.

7 Results

Performance tests are executed for both the communication handler and MPPCRUD. The most important parameters for these tests are the number of database nodes and the object size. No tests are run for the traditional distributed web server configuration (yet).

The communication handler is tested for MPI on the HSlink under EPX (single thread of control) and TCP/IP on a 10Mbps Ethernet. It indicates a lower throughput and a lower latency for the TCP/IP implementation.

The MPPCRUD object read performance test shows that for larger objects (> 64 KB) the theoretical upper limit is approached based on the capacity of 10Mbps Ethernet. In that case, the fact that requests are broadcasted to all the internal servers does not compromise performance (for small object sizes, it does). The network capacity clearly is not in balance with the disk capacity; as nodes are added to the database server, the disk capacity becomes larger but the network capacity remains the same. Tests with faster network hardware must be conducted to get a better impression of the database performance.

8 Future work

To be able to describe the execution characteristics of the system, more measurements must be executed on faster networking hardware. Additionally, performance test will be run for the communication handler and server extensions based on thread safe MPI on the HSlink under AIX.

While the work presented here is a first phase implementation, a future distributed database system may be based on globally unique object identifiers to allow for full object oriented database services. This next phase system would require a change in the server core and a higher implementation effort. New elements such as replication of objects over several nodes to reduce internal network requirements, general distributed query processing and general distributed transaction management must be investigated. The distributed database system integrated within the parallel web server must be tested and web benchmarks must be run.

Applications such as a web cache and a text search engine will be implemented on top of the web server architecture.

9 Conclusions

By building an initial implementation of the distributed database system, hands-on experience has been acquired. It shows that a first phase distributed database system can be build with an acceptable level of functionality, by extending the existing Matisse database server. It also indicates that the Matisse database system offers good support for server extensions. The system can be used in both a traditional and a distributed object based parallel web server architecture.

10 Acknowledgments

We would like to thank Edouard Duvillier for his support and contributions to the discussions we had on the design of the system. The work presented in this paper is part of the JERA project funded by the Dutch HPCN foundation.

References

1. T. Berners-Lee et al., "Hypertext Transfer Protocol – HTTP/1.0", RFC 1945, May 1996
2. National Center for Supercomputing Applications, "The Common Gateway Interface", http://hoohoo.ncsa.uiuc.edu/cgi/overview.html
3. Object Management Group, "The Common Object Request Broker: Architecture and Specification 2.0", http://www.omg.org/, 1995
4. Matisse Software, "MATISSE Software Home Page", http://www.matisse.com/
5. Matisse Software, "MATISSE Server Engine Services (SES) API Reference", 1996
6. Matisse Software, "MATISSE Open MTS API Reference", 1996
7. Matisse Software, "MATISSE OOS C API Overview", 1997
8. Matisse Software, "MATISSE OOS C API Reference", 1997
9. Matisse Software, "MATISSE C++ Binding Programming Guide", 1996
10. Matisse Software, "MATISSE SQL Programming Guide", 1994
11. Sun Microsystems, "Java Development Kit 1.1.4 Documentation", http://java.sun.com/products/jdk/1.1/docs/
12. Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments", IEEE Communications Magazine, Vol. 14, No. 2, February 1997
13. E.D. Katz, M. Butler, R. McGrath, "A Scalable HTTP Server: The NCSA Prototype", http://www.ncsa.uiuc.edu/InformationServers/Conferences/CERNwww94/www94.ncsa.html
14. T.H. Harrison, D.C. Schmidt, "Evaluating the Performance of O-O Network Programming Toolkits", C++ Report, SIGS, Vol. 8, No. 7, July/August, 1996