# Aquarius: A Data-Centric approach to CORBA Fault-Tolerance

Gregory Chockler* Dahlia Malkhi* Barak Merimovich* David Rabinowitz*

## 1 Introduction

The Internet provides abundant opportunity to share resources, and form commerce and business relationships. Key to sharing information and performing collaborative tasks are tools that meet client demands for reliability, high availability, and responsiveness. Many techniques for high availability and for load balancing were developed aiming at small to medium clusters. These leave much to be desired when facing highly decentralized settings.

In order to take the existing, successful approaches a step forward towards large scale distributed systems, we identify two core challenge areas related to information technology tools.

The first is attention to scale and dynamism, in order to exploit reliability and survivability techniques in the networks of today and the future. In this domain, there is a growing understanding that replication based on techniques borrowed from the group communication world fail to scale beyond a few dozens of servers, and incur a serious cost of cross-server monitoring for failures. In the past few years, several protocols were developed around quorum replication in a *data-centric* approach, demonstrating both in theory and in practice that it is a viable alternative for fault tolerance. Advances to quorum systems, e.g., [18, 21, 3, 1, 2, 14] can be employed according to a wide range of parameters, e.g., to mask server corruption due to malicious penetration, thus offering service developers flexibility to choose the replication framework most suitable for the application needs. The Fleet project [19, 20] and the Agile store [16] are prototype systems demonstrating some of these techniques. Several recent works [11, 7, 5, 6, 17] further enhance our fundamental understanding of quorum-based replication. Our work embraces this approach and provides a concrete and detailed implementation and performance assessment.

The second is deployment in real settings and providing an evolution path for legacy software. While we strive to keep the work general, we focus on CORBA [24] as a development platform. This choice is made so as to provide for inter-operability and uniformity, and comply with state-of-the-art heterogeneous middlewares. CORBA is a leading standard for bridging object oriented and distributed systems technologies, simplifying development of distributed applications. Our results provide important insights that may impact the emerging Fault-Tolerant CORBA (FT-CORBA) standard [25].

In this paper we introduce *Aquarius*, a fault tolerant CORBA software that answers the two challenges mentioned above. First, Aquarius employs replication techniques from [7, 5] for survivability and scalability. Second, the design seamlessly wraps legacy software to provide

*School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem 91904, ISRAEL. Email: {grishac,dalia,barakm,dar}@cs.huji.ac.il

a smooth migration path for robustifying existing services. The architecture is demonstrated using a test-case replicated SQL database.

## 1.1 Data-centric replication

Consider a typical service program that is accessible by many clients over the network. The challenge is to robustify the service for high availability and load balancing with little or no intervention to existing client or server code.

The *data-centric* approach regards the service as a shared object which is manipulated by multiple clients. Copies of the object reside on a collection of persistent storage servers, accessed by an unbounded universe of transient client processes. In order to coordinate updates to different copies of an object, clients perform a three-phase commit protocol on the object copies. Their first action is to write their intended update next to the object; then they attempt to commit the update at a quorum; and finally, they commit it and actually invoke the update method on all copies.

The design puts minimal additional functionality on data servers, who neither communicate with one another, nor are aware of each other. Essentially, each server needs a thin wrap that provides a facility for storing and retrieving temporary 'meta-data' per object. Clients are also not heavy. Their interaction is through rounds of remote invocations on quorums of servers. The approach offers great simplicity for constructing fault-tolerant distributed systems featuring a high degree of decentralization and scale. In addition, it faithfully models Storage Area Network (SAN) settings where persistent storage elements are directly accessible to clients via a high-speed network.

More concretely, this approach has several important advantages. First, it alleviates the cost of monitoring replicas and reconfiguration upon failures. Second, it provides complete flexibility and autonomy in choosing for each replicated object its replication group, failure threshold, quorum system, and so on. In contrast, most implementations of state machine replication pose a central total-ordering service which is responsible for all replication management (see, e.g., [27, 4] for good surveys). Third, it allows support for Byzantine fault tolerance to be easily incorporated into the system by employing Masking Quorum systems [18] and response voting. Finally, limiting redundancy only to the places where it is really needed (namely, object replication) results in an infrastructure with only a few necessary components (cf. Section 3) thus simplifying the system deployment and reducing its code complexity. (Our CORBA implementation uses 4K lines of java code for each of the client and server implementations).

## 1.2 Fault tolerance in CORBA

CORBA fault-tolerance has received significant attention in recent years, both in research and standardization, culminating with the recently adopted Fault-Tolerant CORBA (FT-CORBA) standard [25]. The existing fault-tolerant CORBA implementations rely on group communication services, such as membership and totally ordered multicast, for supporting consistent object replication. The systems differ mostly at the level at which the group communication support is introduced. Felber classifies in [8] existing systems based on this criterion and identifies three design mainstreams: *integration*, *interception* and *service*. Below we briefly discuss these approaches and give system examples.

With the integration approach, the ORB is augmented with proprietary group communication protocols. The augmented ORB provides the means for organizing objects into groups and supports object references that designate object groups instead of individual objects. Client requests made with object group references are passed to the underlying group communication layer which disseminates them to the group members. The most prominent representatives of this approach are Electra [15] and Orbix+Isis [13].

With the interception approach, no modification to the ORB itself is required. Instead, a transparent interceptor is over-imposed on the standard operating system interface (system calls). This interceptor catches every call made by the ORB to the operating system and redirects it (if necessary) to a group communication toolkit. Thus, every client operation invoked on a replicated object is transparently passed to a group communication layer which multicasts it to the object replicas. The interception approach was introduced and implemented by the Eternal system [23].

With the service approach, group communication is supported through a well-defined set of interfaces implemented by service objects or libraries. This implies that in order for the application to use the service it has to either be linked with the service library, or pass requests to replicated objects through service objects. The service approach was adopted by Object Group Service (OGS) [9, 8].

Among the above approaches, the integration and interception approaches are remarkable for their high degree of object replication transparency: It is indistinguishable from the point of view of the application programmer whether a particular invocation is targeted to an object group or to a single object. However, both of these approaches rely on proprietary enhancements to the environment, and hence are platform dependent: with the integration approach, the application code uses proprietary ORB features and therefore, is not portable; whereas with the interception approach, the interceptor code is not portable as it relies on non standard operating system features.

The service approach is less transparent compared to the other two. However, it offers superior portability as it is built on top of an ORB and therefore, can be easily ported to any CORBA compliant system. Another strong feature of this approach is its modularity. It allows for a clean separation between the interface and the implementation and therefore matches object-oriented design principles and closely follows the CORBA philosophy.

Two more recent proposals, Interoperable Replication Logic (IRL) [22] and the CORBA fault-tolerance service (FTS) of [10], do not clearly fall in any one of the above categories. IRL [22] proposes to confine the replication logic support (such as total ordering, membership, etc.) within a separate tier for which stronger timing properties can be assumed or enforced. This decouples the replication support from dealing with asynchrony inherent to wide area network environment thus removing constraints on individual object replica placement. IRL provides for high degree of modularity and transparency by allowing the replication support to be introduced and evolved with only minimal changes to the existing clients and object implementations. Aquarius borrows the idea of the three tier architecture from IRL. However, in contrast to IRL, the middle tier of Aquarius consists of independent entities that are not aware of each other and do not run any kind of distributed protocol among themselves. The advantages of this architecture are further discussed in Section 4.

The core idea of the FTS [10] proposal is to utilize the standard CORBA's Portable Object Adaptor (POA) for extending ORB with new features such as fault-tolerance. In particular, FTS introduces a Group Object Adaptor (GOA) which is an extension of POA that provides necessary hooks to support interaction of the standard request processing mechanism and an external group communication system. The resulting architecture combines efficiency of the integration approach with portability and interoperability of the service approach. Aquarius utilizes the object adaptor approach for implementing the server side of the replication support (see Section 4).

### 1.3 Organization

This document is structured as follows: Section 2 describes the data-centric methods we employ. Section 3 describes the overall design of *Aquarius*, and details of the implementation are provided in Section 4. Section 5 presents performance measurements of *Aquarius*. Section 6 describes a replicated database built using *Aquarius*. Section 7 outlines possible future developments.

## 2 Replication methodology

Our methodology for supporting consistent, universal object replication utilizes the ordering protocol of [5]. The algorithm follows the general Paxos framework to ensure consistent operation ordering at all times (even in presence of failures and timing uncertainty) and guarantee progress when the system stabilizes. Similar to Paxos, the stability assumptions are encapsulated into a separate leader election module.

We assume that a single application object is replicated at $n > 2t$ servers up to $t$ of which can crash. We make use of an RPC communication facility that supports asynchronous invocations and guarantees that an operation issued by a correct client eventually reaches all its correct targets. The algorithm tolerates any number of client failures. We first give a brief description of the ordering algorithm. We then outline the implementation of leader election. Further details and the algorithms and their correctness proofs can be found in [5] and [7], respectively.

### 2.1 Operation ordering

The operation ordering is carried out by the client side of the algorithm whose pseudocode is depicted in Figure 7 in the appendix. The client utilizes replicated servers for storing application requests and ordering decisions. The implementation employs two separate threads: one for disseminating application requests to the servers (the *dissemination* thread), and the other one for ordering previously submitted requests (the *ordering* thread). At the core of the ordering thread is a three-phase Consensus protocol whose decision value is an ordering of operations, represented by a sequence of operation identifiers (prefix). The first phase is used to discover the latest decision value which is then extended with newly submitted operations to obtain the next decision value. This new value is then proposed and committed to the servers. The clients employ unique ranks (similar to the Paxos ballots) to prevent concurrent leaders from proposing conflicting decisions and to reliably determine a decision value to extend.

To ensure progress, the ordering thread employs a simple backoff based probabilistic mutual exclusion mechanism similar to that of [7] that works as follows: Whenever an ordering attempt fails because of an intervening ordering attempt by a higher ranked client (PROPOSE returns *nack*), the ordering thread backs off and then repeats its ordering attempt. The backoff period is chosen randomly from the interval $[\Delta, \Delta f(attempt)]$, where $\Delta$ is the upper bound on the time required to order a single operation, *attempt* counts the number of unsuccessful ordering attempts, and $f$ is a function monotonically increasing over *attempt* (e.g., $f = 2^{attempt}$ for exponential backoff). By choosing backoff times from monotonically increasing intervals, the method ensures exclusion among a priori unknown (but eventually bounded) number of simultaneously contending clients. Note however, that the mechanism does not guarantee starvation freedom if the clients keep submitting new operations. The Aquarius implementation overcomes this problem by introducing persistent client side agents (proxies) and extending the basic backoff protocol to support long-lived leader election.

The server side of the ordering algorithm (see Figure 8 in the appendix) is very simple: It supports only three operations: GET to read the replica's state; PROPOSE to store a possibly non-final ordering proposal whose rank is the highest so far; COMMIT to finalize the order and apply the operations to the application object.

## 3 System Architecture

### 3.1 Overview

Our implementation of the data-centric approach forms a middle-tier of client-proxies, whose role is to act on behalf of client requests. Proxy modules may be co-located with client processes, but need not be so. There are several reasons for logically separating between clients and proxies.

**Persistence:** Proxies can be administered with prudence, and thus a proxy that becomes a leader of the ordering protocol may prevail for a long period. This entails considerable savings in the bootstrap of the ordering scheme. In contrast, clients might enter and leave the system frequently, generating contention for the leadership position.

**Efficiency:** A proxy may serve multiple clients, as well as multiple objects. Several important optimizations result from this (as described in more detail below), e.g., batching dissemination requests and ordering operations.

**Transparency:** The client code requires little or no change. For legacy applications this means that they will simply use the remote CORBA reference as they did before, without any changes.

**Extendibility:** The proxy is an ideal location for extended functionality, as it is in the critical path of the protocol. For example, we envision running in the future monitoring and profiling tools on the distributed application.

On the server side, the data-centric approach requires simple functionality. This simplicity allows us to use the *Object Adapter* approach, introduced in [10]. CORBA defines an object adapter, called the *Portable Object Adapter (POA)*, for use in client-server applications. This adapter can be extended and customized according to the application's specific needs. Aquarius defines the *Quorum Object Adapter (QOA)* which adds the functionality required by the ordering protocol without modifying the application server code.

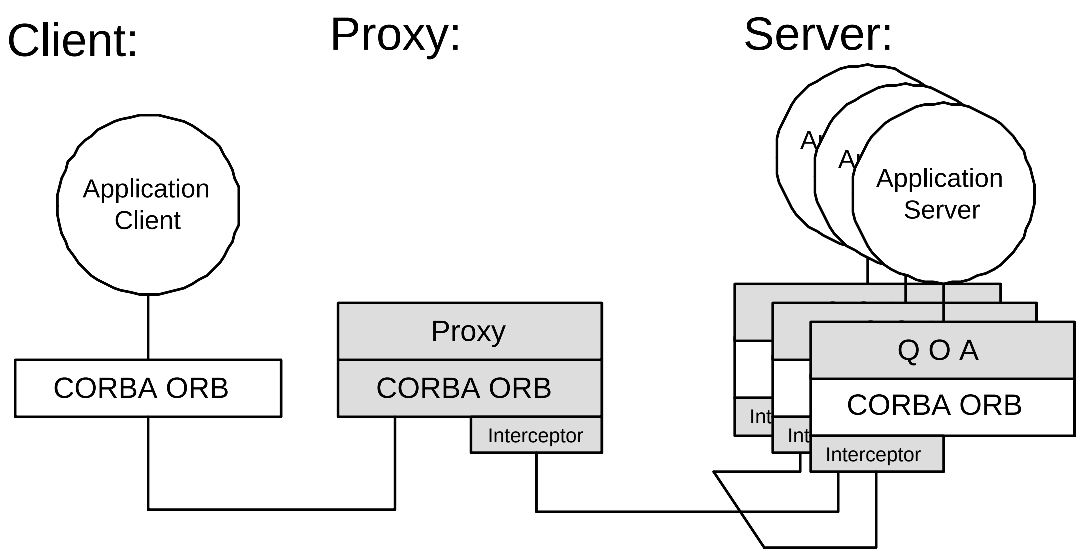

Figure 1: The *Aquarius* Architecture

On the client side, each request is assigned a unique ID used in the ordering protocol. This unique ID is added transparently using the CORBA Portable Interceptor mechanism. This request is sent to the proxy and forwarded to the application servers. No change is required in the client's code.

These then are the main components of the Aquarius system: the QOA that supports the additional server functionality, the proxy that handles dissemination and ordering, and Portable Interceptors that transparently transfer the unique IDs. This architecture is in the spirit of the data-centric approach, maintaining state on the servers and executing the protocol on the proxy, while enjoying the advantages of a three-tier architecture requiring minimal changes to existing code.

## 3.2 Proxy

The *Aquarius* proxy is a stateless server - it holds no persistent data, and requires no stable storage. This allows a backup proxy to assume the leadership position in case of a proxy failure without reconfiguration. The new leader automatically acquires all relevant information in the process of executing the ordering protocol.

The proxy consists of two parallel threads of execution, each with its own separate data and interface. The first, called the *dissemination thread*, is responsible for disseminating client requests to all replica servers, and to process their responses. The second thread, called the *ordering thread*, is responsible for creating a total order of all client requests. The proxy receives client requests via the CORBA Dynamic Skeleton Interface (DSI) which allows it to receive client calls from any application. The message is then forwarded asynchronously via the CORBA Dynamic Invocation Interface (DII) to all replica servers. Once the results from the replicas return, the proxy returns the agreed upon result to the client.

CORBA Object Request Brokers (ORBs) commonly support two threading paradigms: THREAD-PER-REQUEST and THREAD-POOL. In the THREAD-PER-REQUEST model, a new

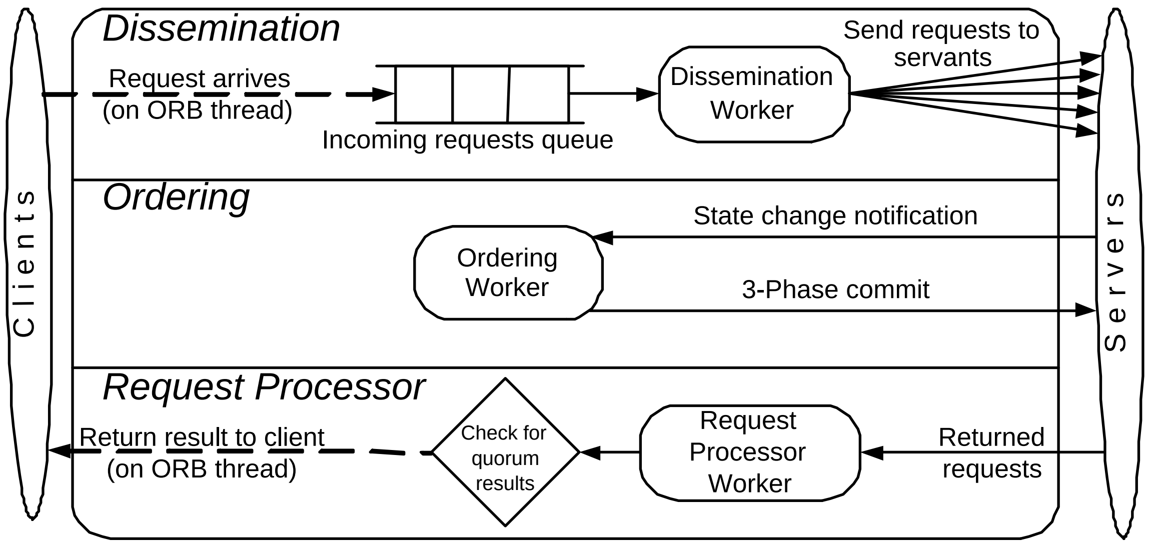

Figure 2: The *Aquarius* Proxy Architecture

thread is created to dispatch each request. In the THREAD-POOL model, a pool of threads is allocated during bootstrap. For each incoming request a thread is removed from the pool and is used to dispatch the request. Once the request is completed, the thread returns to the pool. If no thread is available for a request, it is queued until a thread is ready. Both these models are inadequate for the tasks required of the Aquarius proxy. Since requests can block until they are ordered a THREAD-POOL may easily be exhausted, halting all future operations in the proxy. The THREAD-PER-REQUEST model is not affected by this problem, but it does not scale well. Therefore a different approach is employed.

The Aquarius proxy uses a fixed number of threads which run fast, non-blocking operations. All incoming requests are queued and dispatched by the dissemination thread. Ordering requests are sent and received by the ordering thread, and a third thread, called the *request processor*, is responsible for collecting replica responses and returning results to the clients. This model uses minimal resources while ensuring the proxy does not halt.

The Aquarius system can manage any number of replicated objects, where each object can have any number of replicas, and be accessed by any number of proxies. For each object, one of the participating proxies is designated as the leader proxy that runs the ordering protocol for that replication group. The other proxies are considered followers, and implicitly rely on the leader to order their requests.

## 3.3 Quorum Object Adapter (QOA)

In a standard, non fault-tolerant CORBA application, the server object factory instantiates an implementation object and activates it in a CORBA Portable Object Adapter (POA). It then publishes the reference to this object. In *Aquarius*, the object factory must activate the object in a QOA. This is the only change required in the server. Note that the logic of the implementation remains unchanged. Only the factory class, which is usually a separate and much simpler module, is changed.

The QOA acts as a wrapper to the implementation object and an additional CORBA

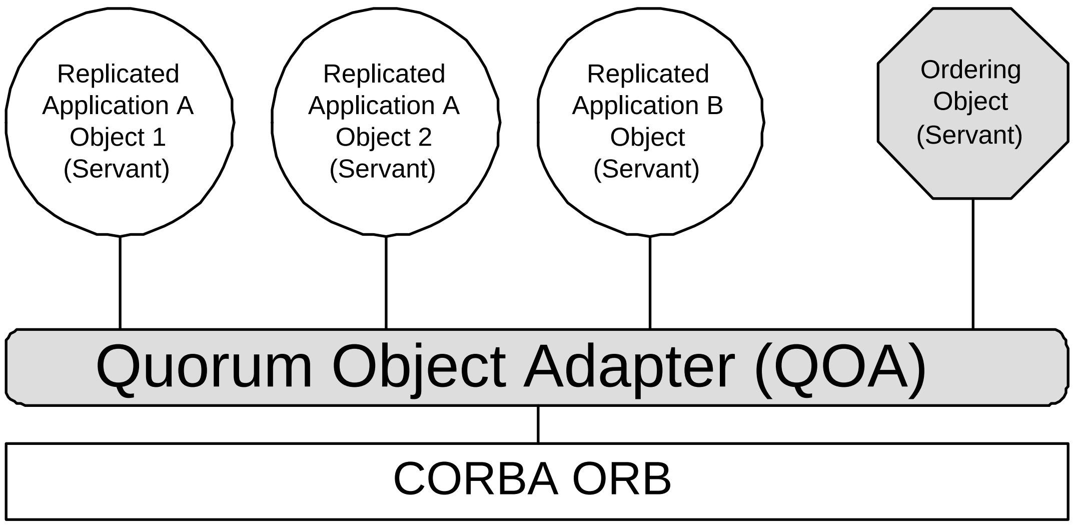
Replicated Application A Object 1 (Servant)
Replicated Application A Object 2 (Servant)
Replicated Application B Object (Servant)
Ordering Object (Servant)
Quorum Object Adapter (QOA)
CORBA ORB

Figure 3: The *Aquarius* Quorum Object Adapter Architecture

object, which is responsible for handling the server side of the total order protocol. This additional object simply maintains the ordering state of the replica, and changes it according to the ordering calls. It is the simplicity of these calls that allows us to embed them in an object adapter.

Dispatching of client calls in the QOA is handled using the CORBA *RequestDispatcher* mechanism: ordering operations are dispatched by the ordering object, and any other operation is kept in storage until they are ready to be executed by the application server object. Operations are ready to be executed once their unique id is committed by the QOA and there is no other unique id prior to it in the total order which has not been executed.

Note that the execution of operations is delegated to a separate thread, so as not to block the ordering object from processing further ordering requests.

## 4 Implementation

### 4.1 Bootstrap

An *Aquarius* proxy is responsible for creating or accessing object replicas. These replicas can be created by a call from the proxy, in which case they are created within the context of an *Aquarius* QOA or they can be existing replicas, created previously by an *Aquarius* proxy. The proxy then creates an object group reference, which looks like a standard CORBA reference, and that can used by any client.

All bootstrap operations use the interfaces defined in the FT-CORBA specification [25]. Specifically, the *GenericFactory* interface is used for creating replicas and groups, and the *PropertyManager* interface is used for specifying configuration parameters.

Once replicas are created and a group reference exists for them, the proxy is responsible for disseminating any requests made on the object group reference to all participating replicas. Any number of proxies can access the object replica-group. Each proxy creates its own object group reference, but all use the same group of replicas. One of these proxies is considered the leader, and is responsible for executing the ordering protocol. The proxies are independent of each other, and are unaware of any other proxies in the system.

## 4.2 Ordering

The ordering protocol is a three phase commit protocol, as defined in [7]. The parameters of each phase in the protocol are arrays of unique IDs which detail the total order of the operations, and pointers to the current position of the total order.

Forming the ordering on message IDs is an optimization of the data-centric approach, as it decouples the dissemination and ordering messages. Each message is assigned a unique ID. The ordering protocol does not require the actual message contents, only its ID, serving to minimize the size of the messages in the ordering protocol. In Aquarius this is supported transparently using CORBA *Portable Interceptors*, which add the unique ID to the message header without any change to the application's client or server code.

A asynchronous notification mechanism between servers and proxies allows the proxy ordering thread to be idle most of the time. As request messages arrive at the replicas, they generate a state change in the server QOA, which notifies one proxy – the current presumed leader – of the change. This initiates the ordering protocol at the proxy leader. Once the message has been committed at the QOA, it is dispatched to the application code, which then executes the request and returns its reply to the proxy that sent it.

In order to support these notifications without registering the proxy at each of the QOAs, a proxy leader calls a remote operation on each of the replicas which only returns if a request is pending at this QOA. If no such operation is pending, the request blocks until a request arrives. This operation is equivalent to the `get()` operation of the ordering protocol, except that it blocks until it can return useful information. The additional remote call adds very little overhead at each of the QOAs while saving the bandwidth required for continuous execution of the ordering protocol.

## 4.3 Handling Proxy Failures

In order to ensure fault-tolerance all components in the system must be replicated, and the Aquarius proxy is no exception. With multiple proxies in place, a client can overcome a proxy failure by simply switching to any other proxy which is a client for the object, or by instructing a proxy to join this group (this can be implemented transparently with client-side Portable Interceptors, as described in [10]). The situation is more complicated if the proxy that fails is also the leader. In this case, ordering operations stops and the QOAs will not execute any application code. Therefore a new leader must be elected.

A proxy that is part of a group can suspect that the proxy leader for the group has failed if a user-defined timeout has expired since it disseminated a client request to the object replicas. All proxies that suspect such a failure will attempt to become leaders by executing the ordering protocol. Only one will succeed, and it becomes the new leader, while the others will fail on a *RankException* and remain followers. The statelessness of the ordering protocol makes this possible, since a new leader can simply resume where the last one failed.

## 4.4 Summary of optimizations and enhancements

While implementing the system we found some practical improvements of value:

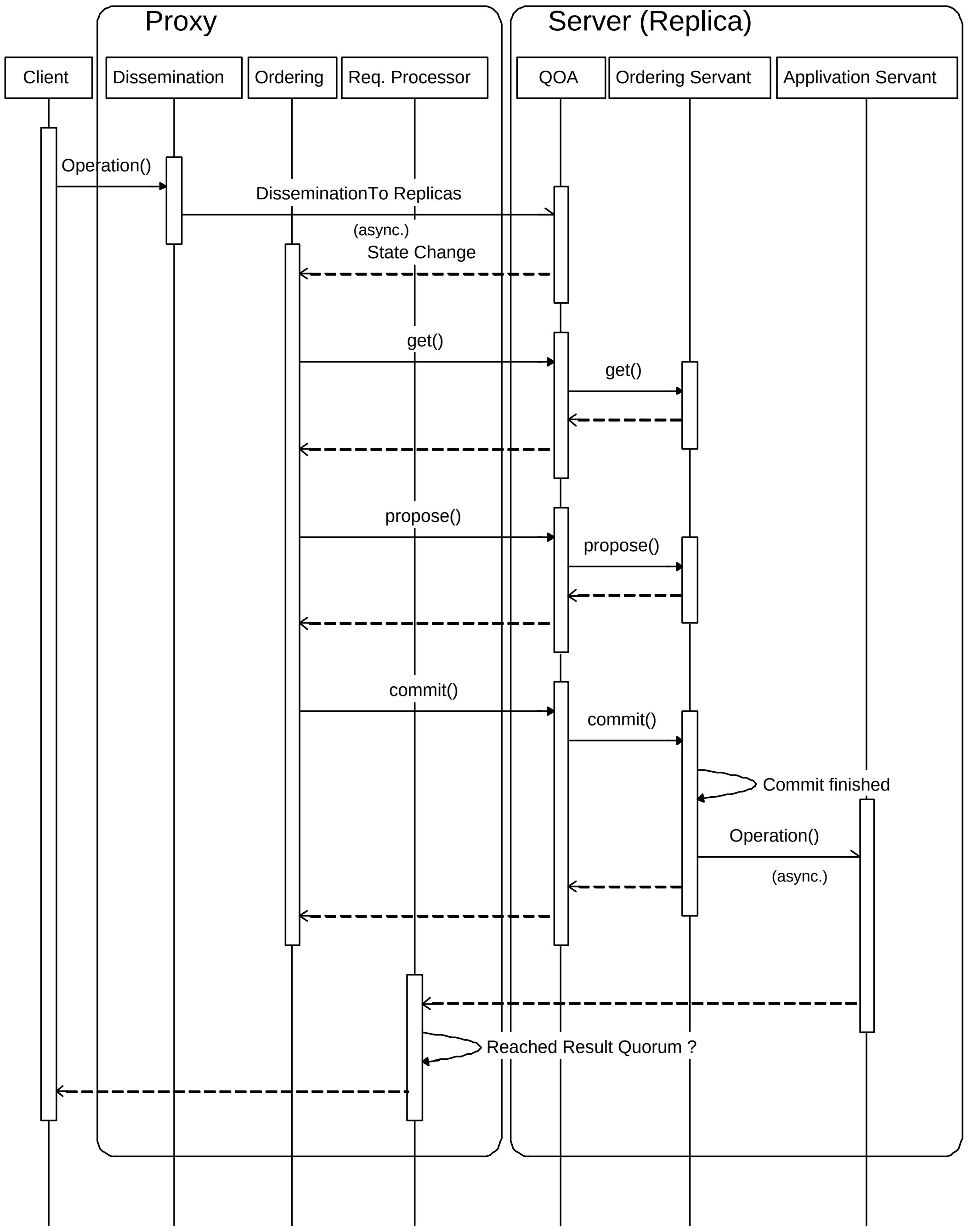

Figure 4: The *Aquarius* Sequence Diagram

**Garbage collection:** Maintaining and transmitting the state of the ordering protocol over time requires a growing amount of resources. *Aquarius* minimizes this by adding the notion of a 'stable line'. This in the most recent command that has been executed by *all* servers. The state before this request can be discarded. This saves considerable amounts of information that must be sent between the proxy and the QOAs.

**Message batching:** The proxy can batch the ordering of multiple messages, sent by multiple clients in one ordering message, thus saving the bandwidth and round trip time required for the remote calls.

**Notifications:** The notifications described above are a practical solution required for the proxy architecture, which also increases the efficiency of the system.

**Threading model:** The specially designed threading model described above requires a constant amount of memory allowing for greater scalability, while ensuring that the proxy never blocks.

## 5 Performance

This section outlines measurements for the *Aquarius* system in a test environment. The system was implemented using ORBacus 4.1.0 [26] and the Java language, using JDK 1.3.1. The experiments were performed on Pentium III PCs over a 100Mbps local area network. Each PC is equipped with a 500Mhz CPU, 256MB of RAM, and runs the Debian Linux (kernel 2.4.18) operating system.

A simple client/server was developed for the experiments. The server contains a single remote operation that receives a buffer of varying size as a parameter. This allows us to measure the round trip time of a request as affected by the size of the request. Two sets of tests were run: the first tests the performance of the system with one client and proxy, and an increasing number of application servers (equivalently, the replication degree). The second test increases the number of concurrent clients while working with a single proxy and five application servers. The results of both tests are depicted in Figure 5 and Figure 6, respectively.

## 6 A database application

Recent years witnessed a great interest in replicated systems, databases in particular. The need to maintain the availability of commercial data led to the development of several database replication techniques. All major database vendors support some sort of replication for their products. Other companies offer middleware that enables replication.

Our approach offers an easy-to-construct replication middleware. The combination of such a middleware with a standard, non-replicated, database is a cheap alternative to commercial products.

We have built a prototype replicated database using our methods over the HSQL [12] database, an open source relational database system written in java. HSQL is a JDBC-compliant SQL database, but has no replication support. By combining HSQL with the Aquarius system, replication is achieved with very little additional code. A simple server object was written in java, that supports a single remote operation. This operation receives an SQL query, and returns its result. The server itself requires only two hundred lines of code,

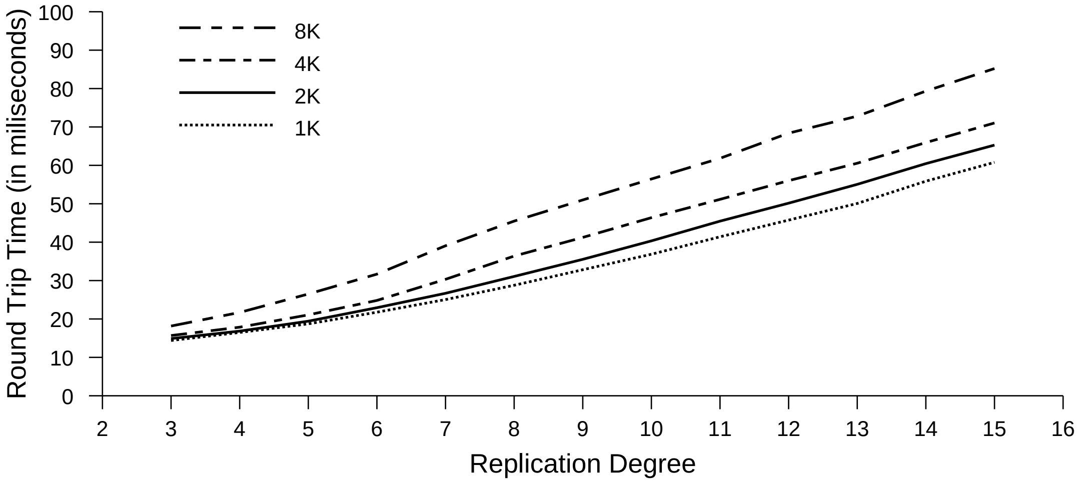

Figure 5: Round trip time according to replication degree and request size

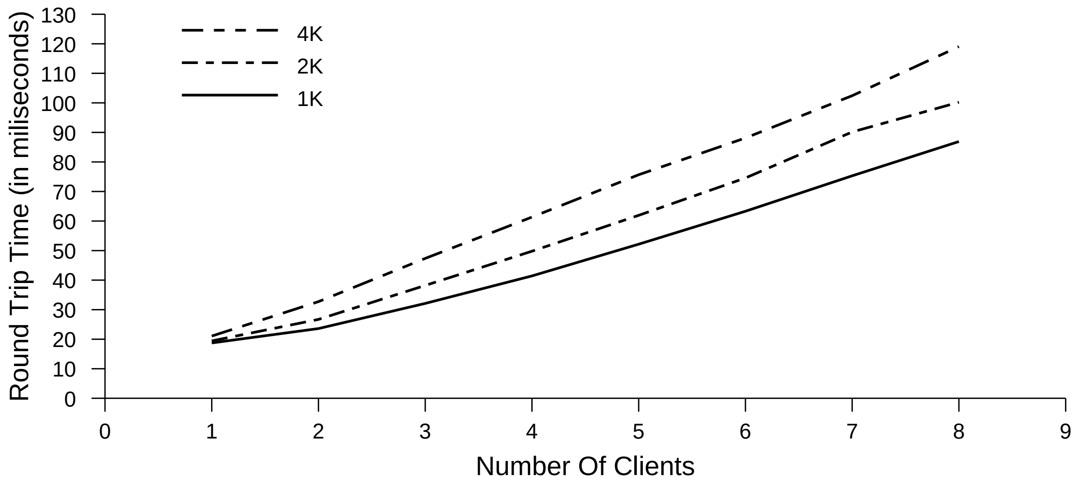

Figure 6: Round trip time according to number of clients and request size (Using one proxy and 5 application servers)

including server initialization and configuration. The resulting system shows good scalability, and supports 50 operations per second for 5 replicas on the test environment described above.

## 7 Future Directions

Future work on the *Aquarius* system includes several possible directions.

**Quorum definitions:** The pluggable quorum management module allows new quorum systems to be tested for efficiency. In addition, the module responsible for communicating with quorums may be extended beyond strict quorums access. Two possible examples is allowing asynchronous backup of a slower secondary site (without slowing the primary site) or supporting dirty reads (read operations that do not require a quorum of replies, risking reading old information).

**Recovery:** Transferring the state of an object replica-group to a new replica, or to a faulty one that recovers.

**Monitoring and Security:** Proxies are an excellent location for handling system monitoring and maintaining access control lists. Portable Interceptors can be used to add these features transparently to the system.

## References

[1] L. Alvisi, J.P. Martin and M. Dahlin. Minimal Byzantine Storage. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC 2002)*, Toulouse, France, October 2002, pp. 311-326.

[2] L. Alvisi, J.P. Martin and M. Dahlin. Small Byzantine Quorum Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002 and FTCS 32)*, DCC Symposium, Washington, DC, June 2002, pp. 374-383.

[3] R. Bazzi. Synchronous Byzantine Quorum Systems. *Distributed Computing* 13(1), pages 45–52, 2000.

[4] G. V. Chockler, I. Keidar and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys* 33(4):1–43, December 2001.

[5] G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC'02)*, July 20-24, 2002, Monterey, California, USA.

[6] G. Chokler, D. Malkhi and D. Dolev. A data-centric approach for scalable state machine replication. In *Future Directions in Distributed Computing, Lecture Notes in Computer Science Volume 2584*, Springer-Verlag, 2003.

[7] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. Proceedings of the *21st International Conference on Distributed Computing Systems*, pages 11-20, April 2001.

[8] P. Felber. The CORBA Object Group Service. A service approach to object groups in CORBA. *PhD Thesis, Ecole Polytechnique Federale de Lausanne*, 1998.

[9] P. Felber and R. Guerraoui and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93-105, 1998.

[10] R. Friedman and E. Hadad. FTS: A High-Performance CORBA Fault-Tolerance Service. In *The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*

[11] E. Gafni and L. Lamport. Disk Paxos. Proceedings of *14th International Symposium on Distributed Computing (DISC'2000)*, pages 330–344, October 2000.

[12] HSQL Database
`http://www.hsqldb.org`

[13] IONA and Isis. An Introduction to Orbix+ISIS. *IONA Technologies Ltd. and Isis Distributed Systems, Inc.*, 1994.

[14] S. Lakshmanan, M. Ahamad, H. Venkateswaran. A secure and highly available distributed store for meeting diverse data storage needs. In *proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*, July 2001, Goteborg, Sweden.

[15] S. Landis and S. Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1), 1997.

[16] S. Lakshmanan, M. Ahamad and H. Venkateswaran. Responsive security for stored data, In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2003).*

[17] D. Malkhi. From Byzantine Agreement to Practical Survivability; A position paper. In *Proceedings of the International Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS 2002)*, October 2002, Osaka, Japan.

[18] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing* 11(4):203-213, 1998.

[19] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering* 12(2):187–202, March/April 2000.

[20] D. Malkhi, M. K. Reiter, D. Tulone and E. Ziskind. Persistent Objects in the Fleet System. In*DARPA's second DARPA Information Survivability Conference and Exposition (DISCEX II)*, California, June 2001.

[21] D. Malkhi, M. Reiter and A. Wool. The load and availability of Byzantine quorum systems. *SIAM Journal of Computing* 29(6):1889–1906, 2000.

[22] C. Marchetti, A. Virgillito, R. Baldoni. Design of an Interoperable FT-CORBA Compliant Infrastructure. In *proceedings of the 4th European Research Seminar on Advances in Distributed SystemsDependable Systems (ERSADS'01).*

[23] L. E. Moser and P. M. Meliar-Smith and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81-92, 1998.

[24] Object Management Group. The Common Object Request Broker: Architecture and Specification, 2.3 edition, June 1999.

[25] Object Management Group. Fault Tolerant CORBA Specification, *OMG Document ptc/2000-04-04*, April 2000.

[26] IONA's ORBacus
`http://www.iona.com/products/orbacus_home.htm`

[27] D. Powell, editor. Group communication. *Communications of the ACM* 39(4), April 1996.

## A Pseudo Code Of The Ordering Protocol

```
Boolean $finish = false$;
Dissemination thread:
    When an operation $op$ is submitted for ordering:
        Assign $op$ a unique id;
        Invoke SUBMIT($id, op$) on all servers;
        Wait until some server responds with $\langle id, res \rangle$;
        $finish \leftarrow true$;
        Return $res$;

Ordering thread:
    do
        Wait until ($isLeader \lor finish$);
        While ($isLeader \land \neg finish$) do
            Pick a unique, monotonically increasing rank $r$;
            Invoke GET($r$) on $n$ servers;
            Wait for more than $n/2$ servers $s_i$ to respond with $\langle r_i, \mathsf{prefix}_i, \mathsf{pending}_i \rangle$;
            Let $Pending = \bigcup_i \mathsf{pending}_i$;
            Let $\mathsf{prefix} = \mathsf{prefix}_j$ such that $r_j = max_i r_i$;
            For each $id \in Pending$ which is not included in $\mathsf{prefix}$
                $\mathsf{prefix} \leftarrow append(\mathsf{prefix}, id)$;
            Invoke PROPOSE($r, \mathsf{prefix}$) at all servers;
            Wait for more than $n/2$ servers $s_i$ to respond with $ack/abort$;
            If more than $n/2$ servers respond with $ack$
                Invoke COMMIT($r, \mathsf{prefix}$) on all servers;
        od
    Until ($finish$)
```

Figure 7: Data-centric operation ordering: The client side

```
Sets $\mathsf{pending}$, $\mathsf{Ops}$, initially empty;
Sequences $\mathsf{prefix}^p$, $\mathsf{prefix}^c$, initially empty;
Ranks $\mathsf{getRank}$, $\mathsf{propRank}$, initialized to a predefined initial value;

SUBMIT($id, op$):
    $pending \leftarrow \mathsf{pending} \cup \{id\}$;
    $\mathsf{Ops} \leftarrow \mathsf{Ops} \cup \{\langle id, op \rangle\}$;
    Execute WAITANDAPPLY($id$)
    in a separate thread;

WAITANDAPPLY($id$):
    Wait until:
        (1) $id$ appears on $\mathsf{prefix}^c$;
        (2) all operations preceding $id$
              in $\mathsf{prefix}^c$ were applied;
        (3) $\langle id, op \rangle \in \mathsf{Ops}$ for some operation $op$;
    Apply $op$ to the application object
    and return the result to client;

GET($r$):
    if ($r > \mathsf{getRank}$)
        $\mathsf{getRank} \leftarrow r$;
    return $\langle \mathsf{propRank}, \mathsf{prefix}^p, \mathsf{pending} \rangle$;

PROPOSE($r, \mathsf{prefix}$):
    if ($getRank \leq r \vee \mathsf{propRank} < r$)
        $\mathsf{propRank} \leftarrow r$;
        $prefix^p \leftarrow \mathsf{prefix}$;
        return $ack$;
    return $nack$;

COMMIT($r, \mathsf{prefix}$):
    if ($\mathsf{propRank} \leq r$)
        $prefix^c \leftarrow \mathsf{prefix}$;
    return $ack$;
```

Figure 8: Data-centric operation ordering: The server side