

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

A HIGH AVAILABLE REPLICATED COMPONENT-BASED DISTRIBUTED ARCHITECTURE

Marcia Pasin, Taisy Siva Weber, Michel Riveill

Projet RAINBOW

Rapport de recherche
ISRN I3S/RR-2003-14-FR

Juin 2003

RÉSUMÉ :

Les architectures à composants sont de plus en plus utilisées car elles permettent un cycle de développement rapide des logiciels par assemblage de composants existants. Certaines des applications visées ont besoin d'avoir des composants hautement disponible, accessible avec des temps de reprises courts en cas de pannes. Ceux-ci peuvent être facilement obtenus en utilisant des données répliquées. Dans cet article, nous présentons une expérimentation utilisant les communications de groupes pour synchroniser des réplicats dans une architecture à composants pour permettre la haute-disponibilité des composants. Cette expérimentation nous a permis de valider notre modèle et de mettre en évidence que la baisse des performances liées à la gestion des données répliquées n'est pas aussi importante qu'elle pouvait l'être initialement prévue.

MOTS CLÉS :

tolérances aux pannes, réplication, composant, EJB

ABSTRACT:

Component - based architectures became a popular abstraction due to the fast software development cycle they provide. High-
available systems require fast recovery time, which can be easily achieved using replicas. However enough experimentation with replication are still required so that replication problems in component-based architectures, as performance bottlenecks, could be well understood. In this paper we show an experiment using group communication abstractions to synchronize replicas in a component-based distributed architecture. The performance penalty of replication in these architectures is not as high as it could be initially expected.

KEY WORDS :

fault tolerance, réplication, component, EJB

A High Available Replicated Component-Based Distributed Architecture

Marcia Pasin^{1±}

Taisy Silva Weber[±]

Michel Riveill^{2*}

[±]Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15064 – CEP 91501–970 Porto Alegre – Brazil
{pasin,taisy}@inf.ufrgs.br

^{*}Université de Nice – Sophia Antipolis
Ecole Supérieure en Sciences Informatiques
930 route des Colles – BP 145 – 06903 Sophia Antipolis Cedex – France
riveill@essi.fr

Abstract

Component-based architectures became a popular abstraction due to the fast software development cycle they provide. High-available systems require fast recovery time, which can be easily achieved using replicas. However enough experimentation with replication are still required so that replication problems in component-based architectures, as performance bottlenecks, could be well understood. In this paper we show an experiment using group communication abstractions to synchronize replicas in a component-based distributed architecture. The performance penalty of replication in these architectures is not as high as it could be initially expected.

Résumé

Les architectures à composants sont de plus en plus utilisées car elles permettent un cycle de développement rapide des logiciels par assemblage de composants existants. Certaines des applications visées ont besoin d'avoir des composants hautement disponible, accessible avec des

¹ Ph.D. student sponsored by CNPq/Brazil under grant 200594/00-1

temps de reprises courts en cas de pannes. Ceux-ci peuvent être facilement obtenus en utilisant des données répliquées. Dans cet article, nous présentons une expérimentation utilisant les communications de groupes pour synchroniser des réplicats dans une architecture à composants pour permettre la haute-disponibilité des composants. Cette expérimentation nous a permis de valider notre modèle et de mettre en évidence que la baisse des performances liées à la gestion des données répliquées n'est pas aussi importante qu'elle pouvait l'être initialement prévue.

1 Introduction

Replication has been proposed to guarantee continuous service in distributed environments [GUE97, WIE2000]. Software designers have traditionally worried about productivity improvement, which led to the development of component-based abstractions. Techniques and tools to allow easy implementation of replication protocols aiming high-available environments are highly desirable. Replication allows automatic and transparent failover, an important fault tolerance function for systems that rely on high availability. If the primary system fails, automatic switchover to a backup database or server that guarantees the service can occur.

Despite of much research focusing on replication, both enough experimentation and its application to a wide variety of practical systems are still required. Replication problems in practical systems, as the low performance associated to replication protocols, are not properly understood. Experimental environments are required to evaluate the performance penalties using replicas can be measure.

Group communication has been proved to be a suitable abstraction to synchronize replicas even in presence of server failures and concurrent client access [AMI94, PED99]. We combine group communication with a component-based architecture to achieve high availability using replicas.

² work partially sponsored by the French Ministry of Research (RNTL project ARCAD)

The architecture is implemented using a popular environment, Enterprise Java Beans (EJB) [KAS2000]. The specification of EJB does not describe any high-available service.

The paper is organized as follows. In the section 2, we present the system model with replicas. In section 3, we describe a multilayer architecture including a hybrid protocol that provides replica consistency. In section 4, we look inside the replication layer. We build a multilayer prototype and in section 5 we show the measurements we obtain using different replica numbers. The paper ends with concluding remarks.

2 System model

We assume an asynchronous distributed system composed of clients and application servers. Clients and servers are implemented as components, i.e., independent binary software units. The component abstraction transparently provides non-functional services to the application. Examples of non-functional services are database management and persistency guarantee.

The servers can use transactions to assure database consistent states. A transaction is a sequence of methods encapsulated by a *begin* and a *commit* operation. A method can be *update* or *read*. If a transaction cannot be completed, due to a system fault, all update methods will be undone.

There are three different kinds of objects: *stateless*, *stateful* and *persistent objects*. Both stateless and stateful objects are typically non-transaction-aware and usually exist just for the duration of a client-server session. Stateful objects have a conversational state, which is stored in memory. This state is only kept while the session remains operational. Persistent objects otherwise are typically transaction-aware and are associated with a database to store their state through persistent data items. Stateless and stateful objects are not shareable by different clients, while persistent objects are shareable and require mechanisms to assure consistency in case of concurrent accesses.

Stateless objects retain no data or conversational state for a specific client. They only require switchover to another server to be high-available. In contrast, stateful and persistent objects require both state replication and switchover to be high available.

Unlike stateless and stateful objects, persistent objects can survive even without state replication if the application server they reside crashes, because their state remains saved on a database. However, persistency and transaction ACID properties are not sufficient to provide a high-available service. If the server goes down, the service will become unavailable. ACID properties and persistency assure a *safe behavior for a system*, but high-available systems require *liveness* [GAR99] that can be achieved using replication. A replication protocol guided by group communication abstractions can synchronize replicas [GUE97] and allow for transparent and automatic failover.

To provide liveness despite of server failures, we extend the previous described system model. The server is replicated and has n identical replicas. Each database has a copy of all data items. Each server represents a *group member* in a *replication group*. An underneath group communication system provides the necessary group communication primitives, group membership and fault detection services. We use the system model with some restrictions. When transactions are cascaded, with replicas calling other replicas, consistency can not be assured. We also do not consider Byzantine failures and network partition. We treat only crash failures.

3 A multilayer architecture

We built a service (fig. 1) extending the non-functional services of a popular non-replicated component-based system, the EJB, to reach high-availability (HA). The architecture that implements the HA service comprises the following layers: the object, the application server, the replication and the group communication layers. Each layer is implemented by a service and is executed by a server component.

Replication corresponds to a new non-functional service. The replication is transparent to the programmer. The new service provides an interface between the application server and a group communication service. Group communication is used only between servers.

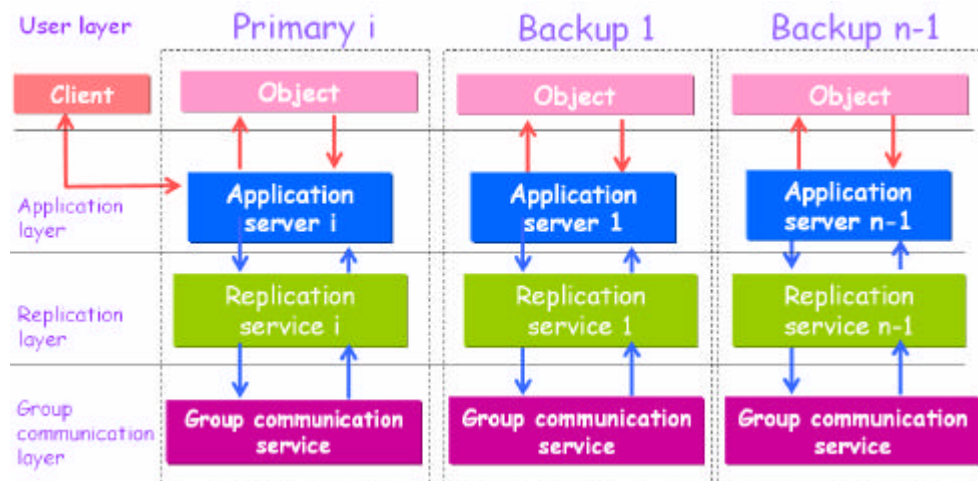


Figure 1 – The multilayer architecture for the HA service

The server the client first contacts is the primary for that client. The other servers in a replication group are the backup servers for that client. The same server can be simultaneously both primary for a client and backup for another one, even when both clients are updating the same object.

The replication layer propagates object updates to backup servers from primary servers. To minimize communication costs, only new states are forward to backups. Read requests are processed locally. The replication layer transparently distinguishes updates from reads using code previously created examining the *object code* at compiling time. When a client wants to use the HA service, it first contacts a *name service* to receive a unique primary server identification. The other servers in the replication group act as backups. Whenever a primary server receives an update request from a client, it executes the update and requests an operation to the replication layer. The replication layer generates a message containing the new *object state* and forwards it to the group communication layer, which propagates this message to all group members using a *multicast primitive*. Each group member receives the message in its local group communication layer and

delivers it to its replication layer, which uses it to implement a *distributed object state*. The distributed state is the set of *local states* stored in each group member.

We use JavaGroups [BAN99] to implement the group communication layer. JavaGroups provides group communication primitives to group members, group membership service to dynamic groups and failure detection service, which removes fault-suspected group members from the group.

We implement the distributed state selecting, at compiling time, updates that will be forward during runtime. An update is assumed to be a method without results (it returns a null value). Optionally, the user could specify which methods would be replicated using a deployment tool. Considering the client side, we modify the client stub to automatically switchover faulty requests to another server.

We expect that our replication approach does not disturb considerably the application response time allowing requests to be handled by multiple sites rather than just one and avoiding any single point-of-failure, a drawback of centralized protocols which are typically use to guarantee *one-copy serializability* for replicated systems. In the sequence, the application server, the group communication and the replication layers will be described properly.

3.1 Application server layer

In the HA service, the Enterprise Java Beans (EJB) architecture [KAS2000] implements the application server layer. The EJB specification describes a server side technology for developing and deploying objects containing the business logic of an enterprise distributed application. The objects of the EJB architecture are called *beans*. *Beans* are scalable, multi-user secure and could be transaction-aware. The EJB runtime environment comprises the EJB application server, which provides access to *beans* offering transactional service, distribution mechanisms, persistence management and security service. The environment allows users to create their EJB applications, freeing them from low-level system these details.

A client accesses a *bean* using point-to-point protocols as RMI, HTTP and RMI-IIOP. The EJB object, the interfaces, the bean class and the bean instance mainly compose a *bean*. A container manages and keeps all *beans* interfacing the *beans* with the application server.

The EJB specification describes three kinds of *beans*: *session beans*, *entity beans* and *message-driven beans*. *Session beans* usually exist only for the duration of a single client-server session and are called transitory *beans*. They implement the stateful and stateless objects previously defined. A *session bean* executes operations such as calculations and FTP downloads. *Entity beans* implement persistent objects and provide methods to locate, create, and manipulate rows of tables. *Message-driven beans* were introduced in the EJB 2.0 specification to accommodate the growing popularity of asynchronous messaging. *Message-driven beans* are a combination of *session beans* and Java Message Service (JMS) clients and they implement stateless objects. *Message-driven beans* and stateless *session beans* retain no data or conversational state for a specific client.

EJB applications use transactions to ensure consistent states despite of concurrent access and failures. Transaction control and other non-functional services are not coded in the EJB application, but they are configured at deployment time as part of the deployment descriptor of the *bean*.

3.2 Replication layer

To enable processing reads locally, the replication layer must implement *strong consistency*. It can be achieved using classical replication protocols, as *active replication* or *primary-backup*, and group communication abstractions.

In active replication [GUE97], each group member executes a local method and sends back its result to the client that typically waits for the first answer. To assure determinism, a client uses a total order multicast primitive, TOCAST, to send a request to all group members. Originally, the TOCAST primitive [GUE97] was designed to groups, that do not change their membership along the time. However, the replication protocol we use allows multiples primaries simultaneously,

which can crash and leave dynamically the group. So we need a total order multicast primitive over a group that dynamically changes their membership.

In the classical primary–backup replication [GUE97], all clients send messages to a primary. All other group members are backups. Determinism is assured because the primary orders all requests. However, if the primary crashes, backups must select a new primary and install a new membership view, removing the old primary from this new view. An inconsistent state can occur if at least one update is not applied to all backups. The view synchronous multicast primitive or VSCAST [GUE97] allows group members to agree on a unique sequence of group views and assures atomicity to all group members despite of crashes.

Nowadays many group communication systems are available and they differ significantly with respect to the assumptions of multicast primitives [CHO2001]. These systems often implement the *closed group model*, which makes the primary–backup replication more attractive than active replication. Active replication requires each client requesting a service first joining the group and then leaving the group, when the service is not more necessary. Every join or leave installs a new view. Successive joins and leaves degrade the distributed system performance. But, if all clients are maintained in the group, it can become difficult to manage and the overhead is unavoidable. On the other hand, primary–backup replication can provide unacceptable response times. This drawback can be avoided using multiple primary servers simultaneously.

3.3 The hybrid protocol

To overcome the drawbacks of the classical replication protocols, we combine them in a single *hybrid protocol*. In this protocol, any group member is able to work as a primary server. When multiple clients request update services simultaneously, the first server each client contacts will be the primary for that client. By delivering the same set of messages using a proper multicast primitive, the hybrid protocol guarantees that all group members will keep the same *state* for all object replicas in a replication group.

Remember that stateless and stateful objects are not shareable by different clients. Two primaries cannot share the same stateless or stateful object. Persistent objects, otherwise, are shareable. Mechanisms to assure consistency for persistent objects despite of concurrent access must be provided.

The hybrid protocol has two variations: immediate update and deferred update. Using the hybrid protocol with immediate update, clients could contact a group member and issue requests to it using point-to-point communication. The contacted member will work as primary to those clients while the other group members will be backups. The primary server immediately executes the service and sends a VSCAST message to the backups that apply the changes. The agreement coordination is required to assure that all backups will receive the message (like the backups in the primary-backup protocol). Then the primary returns client response. Note that this variation of the hybrid protocol uses the VSCAST primitive. Total order is not required because it will be used only to implement high available stateful objects, which are not shareable among clients. Persistent objects use another variation of the hybrid protocol that we describe as following.

Using the hybrid protocol with deferred update, once again the primary server is the first server a client contacts. The primary does not execute the service and forward all client requests to all group members (including itself) using the TOCAST primitive. When all group members receive the message sent through TOCAST, they execute the service and update their local states. Only the primary server sends back the answer to the client. This approach is called *deferred update* because the primary server defers updates until all backups receive the service request which is immediately propagated. In addition, the *deferred update* approach is suitable to multithread database servers and persistent objects which typically lead to non-determinism. Observe that a component-based system could support different off-the-shelf databases connected to the application server. Even if all databases maintain identical replicas from the same data items, the database implementation could be different. This premise implies that any update must be independently processed by any

group member. However the final result must be exactly the same for all group members, i.e., determinism is required to be assured despite of the non-deterministic environment.

It should be noted that group communication primitives are originally proposed for distributed systems, not for database systems. So the mapping from classical replication protocols to database systems using group communication abstraction can not work properly.

Using the hybrid protocol, any client just knows the address of the primary server provided by the naming service. If the current primary fails before it forwards a client request to the backups, the client must reissue the request to a new primary. The crashed primary will be excluded of the next view. If the primary server fails after forwarding a client request, the termination property of the TOCAST primitive assures the system progress.

Detecting if a primary had crashed before or after it forwards a request to its backups is straightforward if the client orders all requests using sequence numbers. The use of sequence numbers helps to maintain the *exactly-once semantic* even in the presence of failures. A client detects a fault-suspected primary server using *timeouts*. Crashed backups are totally transparent to clients.

Finally, if a client crashes after sending a request, it could potentially generate orphan object instances in the server. The server could be able to periodically remove all orphan instances.

4 Implementing the replication layer

The high-available application server extends the actual behavior of the non-replicated application server adding new interposition classes. Achieving the proposed availability requires treating client-primary interaction as well primary-backups interaction. We handle client-primary interaction switching over the client requests to an alternative server, every time the current service is interrupted. We handle primary-backup interaction implementing a distributed state for all

stateful objects and persistent objects. Stateful objects are made high available retaining state behalf of an individual client; and persistent objects using stable storage.

4.1 The client side

Algorithm 1 shows the client side of the replication protocol. Before a new object instance is created (line 5), the client executes a lookup operation through a naming server³ to receive the application server name (line 3). Then the client contacts the application server to create an object instance. The client can access the objects methods or execute transactions with one or more object methods (line 6 and 12). To finish, the client destroys the object instance in the application server.

```
1. client {
2.     object obj;
3.     server := lookup (server_name);
4.     int candidate := number (server);
5.     create (obj);
6.     [ begin a transaction ]
7.     for any client_method do {
8.         execute_a_client_method (obj);
9.         if exception {
10.            switchover (candidate + 1);
11.            execute_a_client_method (obj);
12.        }
13.    }
14. }
15. [ commit a transaction ]
```

Algorithm 1 – The client code

If there is an exception (line 8), generated due *a primary server crash*, during the execution of a method, the client will switchover (see algorithm 2) to another candidate to primary server. Note that the method invocation *could* be inside a transaction, if persistent objects are used.

Algorithm 2 receives as parameter the number id of a candidate to be primary server. The possible candidates are previously ranking in a file edited by a system administrator. Roughly speaking this algorithm looks for an active backup server described in the file using as information the received parameter. If the *current candidate* is unavailable, a next candidate server is tried. If none

³ that could be the JNDI (Java Naming and Directory Interface) server.

server is available, the algorithm finally signals “service unavailable”. The system is high available while at least one server is up.

```
1. switchover (int candidate) {
    object obj;
2.     int total = get_number_of_candidates();
3.     if candidate <= total {
        do {
4.             server = get_candidate_address (candidate);
5.             lookup (server);
6.             create (obj) ;
7.         } if exception {
            candidate := candidate + 1;
8.             switchover (candidate);
9.         }
    } else {
        print "service unavailable"
    }
10.}
```

Algorithm 2 – The switchover code at the client side

In the EJB architecture, the naming service is usually provided through the JNDI server, which makes the previous described approach suitable because the conventional JNDI server does not allow switchover.

4.2 The server side

Algorithms 3-6 show the server side of the replication protocol. The server side consists of two different threads: primary (algorithms 3 and 5) for the primary server and backup (4 and 6) for the backup servers. These threads execute concurrently on each group member because a member could act as a backup for a client and as primary for another client. The backup thread maintains the *distributed state* while the primary thread generates and forwards the distributed state.

The *distributed state* retains in-memory information about replicated objects in a replication group. This information consists of updates for active stateful objects and current transaction information for persistent objects. Stateless objects do not maintain distributed states because they do not retain information about its current execution. Recovering an application server with stateless objects is straightforward: it just requires to switchover the client requests.

Once the service is done and the client disconnects the server, a particular *in-memory object instance* is destroyed in the primary server and a particular distributed state is removed in the replication group. As described previously, there are two possibilities to use the hybrid protocol relating to the kind of object we are using, immediate update for stateful objects and deferred update for transactional persistent objects. The HA service comprises all these different techniques.

4.2.1 Immediate update

The algorithm 3 shows the primary server behavior with immediate update. Whenever the primary server (line 3 in algorithm 3) receives a method invocation from a client to a stateful object, it executes the service (line 4), updates its own state and multicasts the result to the backups using the VSCAST primitive (line 5). The primary answers the client without waiting for any backup answer. Note that the TOCAST primitive is not used because total order is not required. Remember that clients do not share stateful objects.

```
1. primary_server {
2.     operation op;
   while (true) do {
3.         wait until (receive (op)) from client;
4.         if op = write
5.             distributed state ds = execute (op);
6.         VSCAST (op, ds, replication_group);
   }
7. }
```

Algorithm 3 – The immediate update approach at the primary server

The algorithm 4 shows the immediate update approach at the backup servers. Backups process two kinds of synchronization messages: *new view* and *operation*. If the message is a *new view* (line 4), all backups install a new view which is shared between both threads (backup and primary). The group communication layer sends a new view message automatically whenever the membership changes.

```

1. backup_server {
2.     message msg;
3.     while (true) do {
4.         receive (msg);
5.         if msg = new_view
6.             install_new_view();
7.         if msg = op {
8.             if op = create {
9.                 // create a state
10.            }
11.            else if op = write {
12.                // update a state
13.            }
14.            else if op = remove {
15.                // remove a state
16.            }
17.        }
18.    }
19. }

```

Algorithm 4 – The immediate update approach at backup servers

If a backup receive an *operation* message (line 5), it verifies if the operation is a *create*, a *write* or a *remove* one. If the operation is *create*, the backup creates a local state for the object. If the operation is a *write*, the backup updates its local state associated with the object. When the client–server session finishes, the backups receive a *remove operation* and then remove the local state associated to that object.

In the immediate update approach, the distributed state comprises the client identification, the primary server identification and the new object state. The object state is forwarded to the backups as a stream, which is a serialized representation of the object state. The distributed state enables to recovery the object state in the backups if failover occurs.

4.2.2 Deferred update

The deferred update approach for primaries and backups is based on Pedone et al. [PED99]. The deferred update approach at the primary server side is shown in algorithm 5. Whenever the primary s_i receives a commit operation request (line 5) from a client to a persistent object, it immediately forwards a message to all group members using the TOCAST primitive (line 6). In fact, it is not the transaction itself that is sent to the replication group but information about the transaction executing in the primary server. This information comprises: the readset, the writeset and the updates generated by this transaction.

```

1.  primary_server {
2.      operation op;
3.      while (true) do {
4.          wait until (receive (op, transaction)) from client;
5.          // send a message to the replication group
6.          TOCAST ((op, transaction), replication_group);
7.      }

```

Algorithm 5 – The deferred update approach to the primary server

All group members, including the primary server s_i , execute the backup server approach of algorithm 6. The group members can receive three kinds of operation messages: *begin*, *commit* or *abort*. If the operation is *begin* (line 4), it means that a transaction was started in the primary server, so all servers must set a distributed state for this transaction.

If the message is *commit* (line 6), it means that the transaction updates must be applied to all group members, so all group members executes a *certification test* [PED99] (line 7). The *certification test* aims to ensure *one-copy serializability*, i.e., it decides to abort a transaction if its commit would lead the replication group to an inconsistent state. Remember that persistent objects could be shared among clients, so concurrent transactions t_a and t_b which are executing at different database sites and updating the same data item could violate the *one-copy serializability* criteria.

If the *certification test* approves the transaction, i.e., if the transaction does not conflict with another one, all group members commit this transaction. In addition, another approach is required to abort local transactions that conflicts with the recently-committed transaction.

Otherwise, if the operation is *abort* (line 11), the distributed state for the current transaction (previously set by the *begin* operation) is removed from all group members. To conclude, this approach periodically removes instances of the distributed state if they are not more required. In the deferred update, the distributed state is used to detect conflicts among transactions in the certification test as well to allow failover.

It should be noted that sending TOCAST with a *commit* operation represents a blocking message for the sender, while TOCAST with *begin* and *abort* operations is non-blocking. The blocking

message assures that the primary (i.e., the sender) does not continue its service before TOCAST was correctly received by all group members. It is assumed that all group members have the same *state* for their replicated objects.

```
1. backup_server {
2.     message msg;
3.     while (true) do {
4.         receive (msg);
5.         if msg = new_vision
6.             install_new_vision;
7.         if msg = op {
8.             if op = begin
9.                 add. distributed_state (transaction);
10.            else if op = commit {
11.                boolean result := certification_test (transaction);
12.                if result = true {
13.                    commit (transaction);
14.                    // committing
15.                } else { abort (transaction);
16.            } else {
17.                // op is abort
18.                remove. distributed_state (transaction);
19.                if_possible_remove_local_state_from_distributed_state ();
20.            }
21.        }
22.    }
23. }
```

Algorithm 6 – The deferred update approach to backups

5 Performance evaluation

Our experimental evaluation aims to analyze the overhead of both replication protocol and group communication system. We measured the response time of the HA service implementing a prototype with both deferred and immediate techniques in a heterogeneous network of ULTRA 1 and ULTRA Sparc 5 machines connected by a 10Mbps Ethernet. Our experiments take into account different replica number. The HA service was implemented overloading classes of a non-replicated application server (JOnAS [JON2003]).

A prototype allows the best means to estimate the availability and performance constraints of a system. Measuring the prototype allows considering all the support layers of an application and to reach results next possible to the real operational conditions of a system. There is no better way to understand dependability and performance constraints of a complex computer system than by direct

measurement and analyses. The prototype running an appropriate workload emulates real condition and make possible to consider performance bottlenecks before we actually build the system, so that constraints can be factored into the further design phases. We expected to show that replication leads to the desired availability but does not considerably disturb applications response time, when compared with non-replicated application servers.

Figure 3 shows the response time (*msec*) for an application with immediate update and *eager* and *lazy techniques* for different group size. The eager technique means that the algorithm does not continue while all replicas apply an update. The lazy technique means that the algorithm continues immediately after the primary server sends a VSCAST message to group members. The application updates a stateful object. The order property is not required for this kind of object.

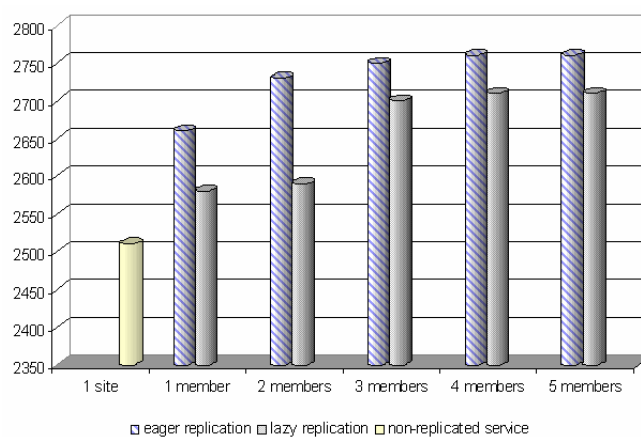


Figure 3 – Response time for eager and lazy replication for different group size

Figure 4 shows the response time (*msec*) to execute an application with a persistent transaction-aware object. Here the total order property is required to avoid concurrency conflicts due to multiple client updates over multiple database replicas.

Each measure in fig 3 and 4 is obtained running the environment 12 times with a client sending a set of read, write and update requests to a synthetic transactional system used as workload. In both experiments the first measure, named *one site*, means no replication and no group communication.

This measure in each figure is taken as basis for comparison. The other measures are obtained varying the number of members in the replication group.

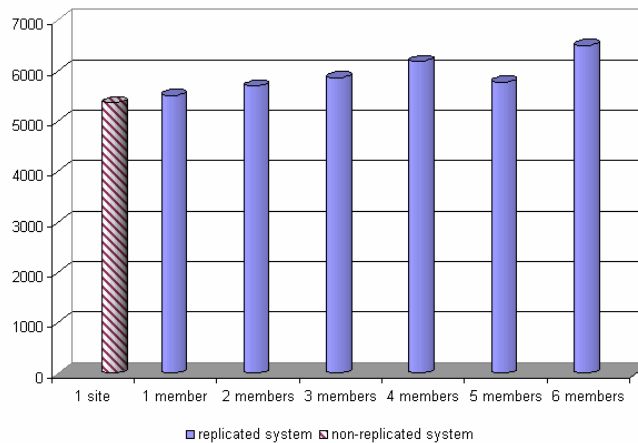


Figure 4 – Response time execute a transaction-aware application with a persistent object

The measure assigned as *one member* in both figures shows only the performance penalty due to group communication. This case also lacks replication as the first measure. The overhead is due the primary sending messages to himself in the group. Replication appears with two or more members, which one holding a replica. The measures show a ascending curve in both figures, but tending quickly to saturate for eager and lazy replication (fig 3).

As should be noted, the performance penalty to achieve high availability for stateful objects is really small, ranging from 3% to 10% overhead in response time (figure 3). For persistent objects the penalty is higher, from 10% to 25% (figure 4). But even in the worst case analyzed, 6 replicas, the overhead is acceptable for high available applications.

6 Concluding remarks

Much research has been done combining group communication and transactions [AMI94, PED99, and SCH96]. Pedone presents some experimentation with different replica configurations while other works remain within conceptual models. However replication applied to experimental practical systems are not offered so that replication problems, as the assumed low performance for

replication protocols, are waiting to be better understood. So this present work contributed in this sense.

Despite providing transactional management that guarantees object persistency, the EJB specification does not describe a high available service. In our experimental environment we implement this service using replication. Group communication demonstrates to be a convenient abstraction to synchronize replicas supporting a non-restrictive failure model and avoiding performance bottlenecks.

Applying group communication to achieve availability is not straightforward. Strategies to minimize communication costs as well to efficiently add new services are still required. Performance bottlenecks must be identified and solved. As we show, excluding the clients from the replication group avoids frequent membership changing. Propagating only updates to the backups reduces the message passing traffic. Enabling a client to contact any group member provides multiple primary servers working simultaneously and avoids additional bottlenecks, a typical drawback of the primary site approach. The immediate propagation with deferred update approach requires more processing than the deferred propagation with immediate update approach but allows employing off-the-shelf databases.

Contrary to what could be expected, our experimentation shows that the performance penalties using replication over a group communication layer is not too high, even in the case of multiple server crashes. Group communication is a feasible abstraction to develop high available data base system in a component-based architecture.

References

- [AMI94] Amir, Y.; Dolev, D.; Melliar-Smith, P. M.; Moser, L. E. *Robust and efficient replication using group communication*. Technical Report CS94-20, Institute of Computer Science, the Hebrew University of Jerusalem, Nov. 1994.

- [BAN99] Ban, B. *JavaGroups user's guide*. Department of Computer Science, Cornell University. August 1999. 73p. <http://JavaGroups.sourceforge.net/>
- [CHO2001] Chockler, G. V.; Keidar, I.; Vitenberg, R. *Group communication specifications: a comprehensive study*. ACM Computing Surveys, v.33, n.4, Dec. 2001, p. 427–469.
- [GAR99] Gärtner, Felix C. Fundamentals of fault–tolerant distributed computing in asynchronous environments. ACM *Computing Surveys*, v.31, n.1, March 1999.
- [GUE97] Guerraoui, R. and Schiper, A. *Software–based replication for fault tolerance*. Computer, IEEE. April 1997. pp. 68–74.
- [JON2003] JOnAS – Java Open Application Server. <http://www.objectweb.org/jonas/>
- [JOH96] Johnson, Barry W. An introduction to the design and analysis of fault–tolerant systems. In: Pradhan, D. F. (Ed.). *Fault–tolerant computer system design*. Prentice Hall, 1996. pp.1–87.
- [KAS2000] Kassem, N. and Enterprise Team. *Designing enterprise applications with the Java™ 2 Platform, Enterprise Edition*. Version 1.0.1, Oct. 2000. 362p.
- [PED99] Pedone, F.; Guerraoui, R. Schiper, *The database state machine approach*. Technical Report SSC/ 1999/008. École Polytechnique Fédérale de Lausanne. Switzerland, March 1999.
- [SCH96] Schiper, A.; Raynal, M. From group communication to transactions in distributed systems. *Communications of the ACM*. v.39, n.4. April 1996.
- [WIE2000] Wiesmann, M.; Pedone, F.; Schiper, A.; Kemme, B. and Alonso, G. *Understanding replication in databases and distributed systems*. Proc. ICDCS 2000, pp.264–274, Taipei, Taiwan, R.O.C., April 2000.