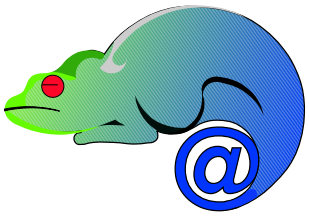
ADAPT

IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

Replication Tools



Deliverable Identifier:	D3
Delivery Date:	8/19/2004
Classification:	Public Circulation
Authors:	B. Kemme, V. Maverick, A. Bartoli, R. Jiménez-Peris, M. Patiño-Martínez, S. Patarin, H. Wu, J. Vuckovic, M. Prica, E. Antoniutti di Muro, S. Wu, J. Milan-Franco
Document version:	Final, 08/16/2004
Contract Start Date:	1 September 2002
Duration:	36 months
Project Coordinator:	Universidad Politécnica de Madrid (Spain)
Partners:	Università di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Università di Trieste (Italy), University of Newcastle (UK), Arjuna Technologies Ltd.(UK)

Project funded by the European Commission under the Information Society Technologies Programme of the 5^{th} Framework (1998-2002)

Contents

1 Dependencies with other Deliverables

This deliverable is a description of several replication tools that have been developed. Some of them will build the building blocks for the integrated BS Middleware (D13 due on month 29). The aspects of the evaluation plan that pertain to replication tools are covered by this deliverable in form of performance measurements. Additional performance measurements can be found in the cited technical reports. The deliverable builds upon deliverable D1 (due on month 6) which outlines the tasks that have to be performed and the challenges associated to it. These tasks can be provided by quite independent modules for the different tiers of a basic service architecture. The tasks are as follows:

- Replication at the application server. Sections 2 to 4 discuss our contributions in this context.
- Replication at the database server. Sections 5 to 7 discuss replication tools on the database level.

Clearly, if both application server and database server are replicated, integration tasks might be necessary to provide correct interaction. Deliverable D13 will focus on this aspect. We will discuss throughout this deliverable, where such integration tasks will take place. Several papers (accepted for publication) and technical reports have been written summarizing parts of developments that have been made within the ADAPT project and which fall under deliverable D13. As such, this deliverable, will only provide a short overview of each of the contributions and refer the interested reader to the detailed papers and technical reports.

Basically all developed replication solutions both at the level of application servers as for database replication, use group communication systems (GCS) for communication among replicas. The group communication system is used for group maintenance and for multicast. In most cases, uniform reliable total order multicast is used. For more information about GCS, see deliverable D2.

2 The ADAPT Framework for Application Server Replication

2.1 Introduction

In recent years, the J2EE architecture has become popular for web-based applications and web services, providing services to manage transactions, persistence, security, and object life-cycles. In J2EE based application server systems, replication is an essential strategy for reliability and efficiency. Typically, replication means that several instances of a server are started. If one instance crashes, the others can continue to work. Furthermore, depending on the replication configuration, the load can be distributed across the different replica, and new replicas can be added in order to serve more client requests.

There exist many different replication strategies that are determined by various parameters. (i) Different *types of components* might have to be replicated. In J2EE, the main components containing business logic and state are servlets and EJBs. (ii) Replication can be active, i.e., each request is executed at all replicas. The system is responsive as long as one replica is running and returns a result. Active schemes require deterministic execution, and can only be used for fault-tolerance but not for scalability. In a passive scheme, for a given client session, one server replica is the primary for this client while the others are backups. Only the primary executes the requests and multicasts any state changes to the backups. If the primary fails, failover takes place, and one of the backups becomes the new primary, and continues the execution. Maintaining and applying state changes at the backup is usually much faster than executing the request itself and requires less resources. Hence, the backups can use the unused resources to be primaries for other clients. The disadvantage of the passive scheme is the complex state transfer from the

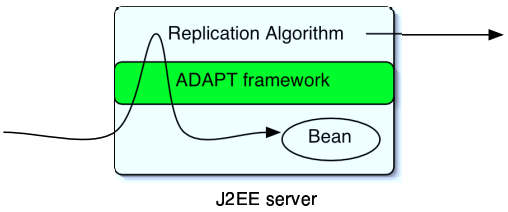


Figure 1: A replication algorithm is layered on top of the ADAPT framework

primary to the backups. (iii) *State propagation* defines how the state is propagated to backups in passive schemes. Using *cold propagation*, the primary stores the state information on an error-free persistent storage which can be accessed at failover by the new primary. In this case, the new primary can actually be initiated only when needed after a crash. Using *warm* propagation, the primary sends the state to the backups directly via messages or distributed shared memory. Backup instances must exist but we can assume that in-memory propagation is faster than writing to disk. (iv) The *propagation time* defines when state propagation takes place. Using *eager* propagation the state is propagated some time before the response is returned to the user, in *lazy* propagation, only some time after. Eager replication leads to slower responses but can guarantee consistency. Lazy replication provides fast response but consistency might be lost if the primary crashes after a response is returned but before state propagation.

Many commercial J2EE application servers provide some form of replication for J2EE. We will discuss them more closely in Section 3.4. Understanding the implications of the different alternatives is difficult, and the choice of the best one, depending on the expected application profile, remains an open research question. Our work attempts to be a further step in this direction.

The component model of J2EE (providing components like servlets, session and entity beans, etc.) has benefits for the development of replication support. It requires the developer to keep application state within clearly-declared objects; the invocations of these objects are intercepted by the server; and transactions are handled by a central transaction manager with a well-known API. Together, these constraints mean that in a J2EE system, all the application events of particular interest for replication are already "exposed" to the application server so that, in principle, a replication algorithm can observe and intervene in them.

Still, the design and evaluation of a replication algorithm for J2EE (or any practical component architecture) requires a substantial investment in development. An application server is a complex piece of code, and modifying it is not easy. Further, much of the work of modifying the server is common between different replication algorithms: most algorithms, for example, will need to intercept component invocations from outside the server. Finally, the modifications will need to be redone or re-examined for every new version of the underlying application server.

For all these reasons, we have chosen to develop J2EE replication strategies in two layers (Figure 1). The lower layer is our ADAPT framework, which handles all the detailed interactions with the underlying server code. The specific replication algorithm is plugged into the framework and runs on top of it. The framework is implemented once and for all for a given application servers, and can be used by different replication algorithms. The interaction between the layers is defined by an API. Through the API, the replication algorithm sees a simplified view of the components in the system and the invocations between them. It does not see the underlying J2EE implementation. When a component is invoked, control passes to the replication algorithm before reaching the component. The algorithm may perform other actions,

such as communicating with other replicas, before or after the invocation. Building replication algorithms on top of a framework has several advantages:

- The framework provides important building blocks that are necessary for many replication algorithms, e.g. to get and set the state of components, and to intercept calls to components, the transaction manager, or other services. Implementing such general functionality once for all replication algorithms, easens development and allows for reusability.
- The framework hides the specifics of the given application server. Only the framework developer must be very familiar with the underlying application server to be wrapped. The developer of the replication algorithm, in contrast, is provided with a high-level interface that is designed specifically with replication in mind.
- In the context of our ADAPT project, the chosen approach allowed us to split development and expertise needed for D3 among the different research partners in a modular way. Bologna was the main developer of the framework, with extensive knowledge of JBoss. Trieste and McGill developed replication algorithms focusing on their correctness and efficiency. Of course, throughout the development, extensive communication among the research partners took place. This was in particular needed, since the development of the replication algorithms started before the final version of the framework was in place. Hence, useful feedback could be given at all times.

2.2 Design

In this section, we only present the major features of the framework. For more details, we refer to [BBM+04].

2.2.1 Uniform model of components

For a detailed description of J2EE components, we refer to D1. Here, we only want to recall the following abbreviations. In regard to *Enterprise JavaBeans (EJB)*, *session beans*(SB) can be *stateless session beans* (SLSB) and do not contain internal state across client calls, or *stateful session beans* (SFSB) and maintain internal state for the lifetime of a caller session. *Entity beans*(EB) are objects that represent data in persistent storage (mostly database system). *Message beans* are another kind of EJBs. They are currently outside the scope of our research. Servlets are components of the web-container. We are interested in both supporting client-server applications as well as web services. Client applications invoke beans through the RMI remote-procedure-call protocol. For web-services, stateless session beans can be used as web-service endpoints (see the EJB specification [DeM03], section 5.5). An alternative is to use the Axis SOAP engine [Apa03]. In Axis, a web-service endpoint is implemented by a special type of component, somewhat simpler than an EJB, but likewise deployed with a declarative descriptor. For our purposes, an Axis service implementation object is simply another variety of component. Web-service clients invoke Axis service objects or stateless session beans through SOAP messages sent by HTTP.

All kinds of components (currently stateful session beans, stateless session beans, entity beans, and Axis web-service objects) are presented in the same way to the replication algorithms. Components are created by application code, whether directly from the client or indirectly through other components. The framework notifies the local replication interface after a component has been created, passing it a *component handle* which refers to the specific component instance. The client may also look up persistent

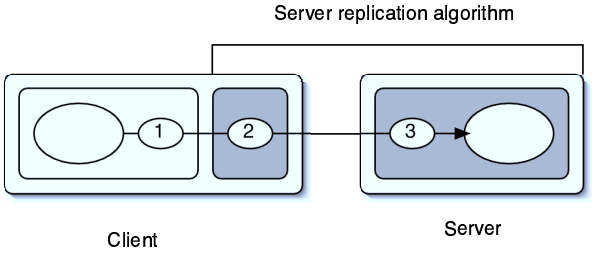


Figure 2: Interception points

components (entity beans) by their primary keys. When they are found, the server instantiates them, and then the framework notifies the replication algorithm that they have been instantiated, again providing a corresponding *component handle*. The handle provides information about the kind of component, its name (each deployed bean class has a name), and the instance identifier. It provide methods to create the component (e.g., at another replica), or test whether two components are equal. The framework provides methods to test whether a component has state, and to get and set the state. In the API, the state of a component is an opaque serializable object, which can be sent between replicas. The framework uses EJB mechanims to access state (e.g., passivation for session beans, and reading/writing attributes defined by container-managed persistence for entity beans). For Axis web-service objects, the class must implement the Java `Serializable` interface.

When the client deletes a component, the framework notifies the replication algorithm before the deletion takes place. The algorithm may not prevent the deletion, but it can perform any related processing before the component disappears.

While an entity bean is in memory, the J2EE server treats it as a cached copy of the corresponding persistent data. If its state is consistent with the database, the server may choose to flush it from memory. The ADAPT framework allows the replication algorithm to block a component from being flushed, forcing it to remain in local memory.

2.2.2 Interception of invocations

When one component makes a call to another component, the replication framework intercepts the call at three key points (see Fig. 2). At each point, the replication algorithm may intervene, performing any computation or communication before or after continuing. In fact, it does not have to continue execution along this path; it may throw an exception, or return a response computed elsewhere. The interception points are (1) Just before control leaves the caller. If the caller is replicated, the replication algorithm may synchronize with other replicas before proceeding. (2) If the caller is an RMI client, we intercept at the stub, i.e., client-side logic belonging to the server replication algorithm. It may resend requests, fail over to another server host, etc. (3) Just before control is transferred to the target component. The component is ready for the invocation, but also for other operations such as reading and writing state.

If the caller is replicated as well as the server, then there are two replication algorithms in this scenario. At point 1, control is still in the domain of the caller's replication algorithm. At point 2, even though execution is still on the calling host, logical control belongs to the callee's algorithm. In particular, if the stub detects a server failure, it can fail over to another server host. The failover mechanism depends entirely on the server's replication algorithm, not the caller's. (We indicate the domains of the two

algorithms with shading.)

The replication algorithm expresses the logic at each of these points by implementing an interface. The framework API defines a "local" interface, for logic in the local server, and a "stub" interface, for logic referring to a remote server. When the framework intercepts execution at points 1 and 3, it passes control to the local interface, implemented respectively by the replication algorithms of the client and the server. At point 2, it passes control to the stub interface, which runs on the caller but provides invocation logic for the server's replication algorithm. For instance, at point 3, when a component is about to be invoked, the ADAPT framework calls a special method `call` in the local replication interface taking the request and the component handle as input. The replication algorithm, if it wants to execute the request on the component, it calls the corresponding method of the component handle with the request as input parameter. The component handle performs the call on the corresponding component and returns the response to the replication algorithm. The replication algorithm can do additional work (e.g., replicate state), and then return to the framework with the response.

In an EJB invocation, the stub interface is actually downloaded from the remote server during EJB lookup. In a web service invocation, though, the client cannot download code from the server (and would not trust it if it did). In a real web-service application, server-specific logic on the client side would be spelled out in the service contract, and implemented by the client. For convenience in prototyping, though, we maintain the distinction between the two interfaces even with web services.

For EJBs, we distinguish two interception points on the server side, that is, 3 is split into 3a and 3b. 3a comes before the component reference has been resolved. Breaking here allows the replication algorithm to instantiate the component itself, if necessary. The second point (3b) comes after all the EJB properties, such as security and transactions, have been set up. At this point, the replication algorithm can get and set the component's state, attach a listener to the transaction, etc. For web services, there is no useful distinction to be made between these two points, because the invocation model for Axis objects is much simpler than for EJBs. Thus, we provide interception only at 3b.

2.2.3 Requests and responses

In the ADAPT invocation API, a *request* is passed to a component, yielding a *response*. Generally, request and response are opaque to the replication algorithm. However, in a request, we allow the replication algorithm to read the name of the method that is being invoked (or, for a web-service component, the operation). This permits the algorithm to classify the methods of a component, and treat them differently. For example, if it has access to more information about the application (perhaps an extra descriptor provided by the developer), it might distinguish between read-only and read-write methods.

When the invocation completes normally, the response encapsulates the return value (or, for a web component, the SOAP response). In this case, the replication algorithm cannot examine the content. When the invocation throws an exception, though, this is wrapped in a special response which provides the details of the exception and identifies its source. If the exception was thrown by the component itself (developer-written code), the replication algorithm should simply pass the response back, where it will be handled by the calling component. If the exception is thrown by the system or framework, for example when the server crashes, the client-side replication code can catch this and "fail over" to another server before returning to the caller. If the exception is thrown by the replication algorithm, the replication algorithm is free to examine the exception details and handle them as it chooses.

Both requests and responses can be tagged with "headers". These are arbitrary key-value pairs, which are transmitted along with the content of the message, but they are visible only to the replication algorithm. The key must be a string; the value may be of any class that can be serialized in the invocation.

A common use for headers is to tag each request with a unique ID in order to guarantee that each request will be executed exactly once, despite retransmissions and communication failures.

2.2.4 Transactions

J2EE models transactions with a standard API ([Sha03], chapter 4). This defines a transaction manager which is called by clients, application components, and the server itself, to begin and end transactions and to register participants (in case of 2-phase-commit). And it defines a transaction, which may be associated (one-to-one) with a thread. If so, transactional operations in that thread, such as database access and EJB invocation, are logically contained within the transaction.

To let the replication algorithm follow the association of component invocations with transactions, we provide two framework methods. If the bean uses container-managed transactions, then at interception point 3b, the transaction will already have been associated with the thread. The algorithm can look it up through the transaction manager. If the component manages transactions itself, through direct calls to the coordinator, the framework notifies the algorithm through a callback.

To track the later commit and rollback of a transaction, the replication algorithm may attach a listener in case of 2-phase-commit, using one of two interfaces defined by J2EE. One interface is notified after the transaction has committed or rolled back; another actually participates in the two-phase commit.

If the replication algorithm wants to intervene more actively in local transaction processing, the framework allows it to "wrap" the entire transaction manager, intercepting every transactional event. It may choose to pass the event on to the underlying transaction manager, or to perform its own distributed logic, or both.

Another approach might have been to open the internals of the transaction manager to the replication algorithm. However, J2EE does not define these internals, so this approach would have meant choosing a particular transaction manager implementation.

2.2.5 EJB lookup mechanisms

We also allow the replication algorithm to intercept the lookup and instantiation mechanisms of EJB even before the component itself is created. First, J2EE defines a naming service, JNDI. Each component is registered with the JNDI service of its local server. To find the component, the client connects to the service and looks up the component's name. We allow the client stub to redirect the JNDI lookup allowing the replication algorithm to provide a custom implementation of the interface. Second, the JNDI lookup yields the home interface for the component, which provides methods to create new instances and to find existing ones. The invocations of these methods are intercepted on both the client and server sides, at points 2 and 3. On the client side, the replication algorithm may redirect the calls to another server; on the server side, it may perform related actions (such as deployment) before allowing the server to proceed.

2.2.6 Deployment

A component cannot be instantiated on a server unless it is deployed there, that is, its code, configuration, etc., are available. Before replicating components across a cluster, an algorithm must ensure that they are deployed on all servers.

In J2EE ([Sha03], chapter 8), components are deployed in archive files, with a specified filename extension and internal structure. Typically, this file is "delivered" to a server by being copied to a specified directory. The server checks the directory at startup, and at regular intervals afterward.

Our API provides a simple model for deployment information. Each component archive is represented by an identifying handle and a content object, which can be transmitted together or separately. At startup, the replication algorithm can query the framework for all the units that are currently deployed. During runtime, the framework notifies the replication algorithm whenever a new unit is deployed. The algorithm can transmit the handle to its peers, which can test whether the handle is deployed locally. If not, they can deploy it through the framework, by providing the handle and the content object.

2.2.7 Further issues

The framework provides a server address class that encapsulates the IP address and ports for a server. The replication algorithm can share addesses, and send a set of addresses to the client using response headers. The framework also provides suspend and resume methods to block and unblock requests on the local server. They can be used if, e.g., state transfer needs to block request processing for the time of the transfer.

2.3 Implementation

We have implemented the replication framework by building on the open-source J2EE server JBoss [JBo03]. For web services, we used the SOAP engine Axis [Apa03], which is integrated into JBoss. In each case, the existing architecture provides hooks for intercepting and restart invocation. JBoss's EJB implementation is structured around "interceptors", a pattern which is built up into invocation "stacks" described by a configuration file. Axis supports the handler model defined by JAX-RPC [Chi03]. A configuration file defines the sequence of handlers to be executed before a service request is finally delegated to the service object. We did not modify any of the Java source files in the JBoss or Axis distribution; instead, we modified the configuration files, inserting our own interceptors and handlers into the existing invocation paths. This will make it easier to port the framework to future versions of JBoss and Axis.

The performance of the framework was evaluated individually and performance results are presented in [BBM+04]. Section 3 is based on the framework. It includes a performance analysis which also evaluates the overhead of the framework.

3 EJB Replication

So far, we have developed two independent replication algorithms. One algorithm considers EJB replication. Clients access EJBs through the client RMI interface (which is similar to the servlet RMI interface). The second considers web-service object replication, in particular, how to replicate the Axis web-service objects. So far, however, these objects may not call other objects or components. D13 will discuss the integration of these two replication algorithms into one algorithm where EJBs can be called from web-services. In the current deliverable, we will discuss the two algorithms separately. This section focus on EJB replication, Section 4 will present web-service replication. A detailed description and performance analysis for EJB replication is provided in [WKM04]. Here, we only highlight the main features.

3.1 Model and Assumptions

Since methods on EJBs are usually called within the context of transactions, we have to keep transactions in mind when we do application server replication. In particular, we want our replication algorithm to have two important properties. (i) The replicated system should provide the same degree of *state*

consistency as the non-replicated J2EE in the non-failure case. In a non-replicated system, if a transaction commits, both the database and the EJBs within the application server should reflect the changes of this transaction. If the transaction aborts, if full state-consistency is required, none of the changes should remain. Database systems provide efficient abort mechanisms. Within J2EE, application programmers can specify a *compensation* method for each business method that will be called by the server in case of an abort, undoing the state changes performed within the application server. If state consistency is relaxed, only the database changes are cancelled, the changes in the EJBs remain. In a replicated environment, we have to extend this definition. Changes of EJBs performed by a committed transaction should be reflected on all server replicas and the database, changes of aborted transactions should not remain on any server with full state consistency, and should be reflected in all replicas with relaxed state consistency. (ii) The system should provide *exactly-once* transactions, i.e., as long as the client does not crash, a transaction is executed exactly once (if the client crashes the semantics should be *at-most-once*). That is, even if a replica crashes during the execution of a transacton, the others will continue and terminate the transaction as if no crash had happened.

The implemented algorithm makes a couple of assumptions. We assume that the nodes running application servers can fail by crashing (no byzantine failures). We assume reliable, asynchronous communication and no network partitions. The algorithm in the next section is able to handle network partitions, and we believe that we can use a similar mechanism for the same purpose. Furthermore, we assume both clients and database do not crash and the connection to the database is reliable. Of course, our final goal is to provide web-service access to the EJBs, and web-service clients can easily fail. Such failure can easily be handled by simply not sending a response to the client. Sections 6 and 7 discuss how database replication approaches can make databases reliable. The integration of a replicated database with the replicated application server is topic of D13. The replication of clients is currently not considered but we might considers this in future work.

In regard to the application server, the current implementation makes a couple of assumptions, some of which will be relaxed in the near future. (i) We assume both client and EJBs (except of SLSB) follow the typical *request/reply* protocol. The caller of an EJB submits a request to the EJB, and is blocked until it receives a reply. This is a typical programming model, and hence, we believe it is not a severe restriction. Consequently, SFSBs do not need any concurrency control since they are associated with a single client with at most one outstanding request. (ii) We assume CMT where all methods have the transaction attribute set to *required*. Our current efforts eliminate this restriction. (iii) In the description of this deliverable, we assume that a regular J2EE server (without replication) correctly handles state consistency and at-most-once execution in the failure-free case. The case for relaxed consistency can be handled with an easy extension of the algorithm but is omitted for space reasons. (iv) If the J2EE server crashes, the J2EE server's state and its connections to the database will be lost, and the database aborts all active transactions. That is, the database contains the changes of all committed transactions but no changes of aborted transactions or transactions active at the time of the crash. We tested with DB2 that this database behavior is, in fact, true. (v) We assume that J2EE will eventually give a client a response (either the application dependent response or an exception if the server crashes). This is standard behavior of J2EE. (vi) Finally, we do not discuss the case where a transaction accesses more than one database. In principle, our algorithm can handle this. The current implementation, however, does not provide the functionality since we have encountered some problems with the XA interfaces of the database systems.

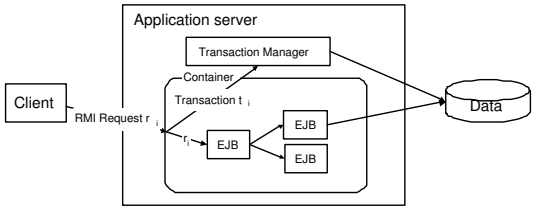


Figure 3: Typical Execution Flow of a J2EE server

3.2 Replication Algorithm

Our replication scheme uses passive replication to allow non-deterministic execution, and to avoid redundant computation. For EBs, we choose cold replication since changes are always written to the database at commit time by default. SFSBs use warm replication to achieve faster failover, and alleviate the load on the database. Group communication (GCS) is used for this purpose (see D2). Since our main goal is state-consistency, we use eager replication. Any lazy approach makes non-determinism hard to handle and typically does not allow for a generally applicable consistent solution. We split the protocol into five different parts. The *client protocol* executed at the client side is responsible for resubmitting requests in case of server crash. At the server site, a *primary protocol* runs at the primary and a *backup protocol* runs at the backups during normal processing. Furthermore, a *failover protocol* runs at the new primary when the old primary crashes, and a *recovery protocol* is used when new replica joins the replica group. Currently, there is only one primary executing all request. In principle, requests of different clients could be executed at different primaries to allow for load balancing. We are currently implementing this extension. Due to space limitations, we keep the protocol description rather high-level. The interested reader is referred to [WKM04].

3.2.1 Client protocol

The basic task of the client protocol is to forward a client request to the current primary. The client protocol maintains a list of server replicas, and a pointer to the replica it believes is the current primary. New server lists are piggybacked by response messages of the primary. The client protocol also attaches unique request ids to requests. If the response to a request is an exception indicating a server crash, the client protocol asks each server in the server list whether it is the new primary and resubmits the failed request with the same id to the next new primary.

3.2.2 Primary Protocol

Let's first recall the execution logic of a container managed transaction in a regular J2EE model as depicted in Figure 3. When the container intercepts a client request, it first calls the transaction manager (TM) to begin a new transaction, then forwards the request to the EJB and waits for the response, then calls the TM to commit the transaction, and finally returns the response to the client. Each call to an EJB carries the identifier $txid$ of the transaction that is associated with this call.

The primary protocol extends now this basic scheme. For each request, it keeps track of its id, the associated $txid$, the response to the client when existing, and a list of updated EJBs. A *replication manager RM* intercepts the following actions. (i) It intercepts the *begin* call to the TM to keep track of the $txid$ associated with the request. (ii) It intercepts requests to EJBs. When a client request comes

in, it first checks whether already a response for this request exists. If yes, the response is returned. Otherwise, the corresponding action on the EJB is executed, and if the EJB is updated, the RM keeps track of this (state changes are detected by comparing pre- and post state). The response created by the EJB is kept together with the request in a hashtable. This is only done if it is the response returned to the client (note that subrequests create sub-responses that need not be kept). (iii) The RM also intercepts *commit* requests. A *committing* message is FIFO multicast containing, among other, request, response, state of all changed SFSBs, identifiers of all changed EBs, and some other things. A marker is inserted in the database containing the $txid$. When it is guaranteed that all backups have received the message (uniform reliable delivery), the database transaction is committed, and response is returned to the user. A *committed* message is multicast to the backups to speed up failover (not needed for correctness). We will discuss transaction abort later.

3.2.3 Backup protocol

The backup protocol used during normal processing is designed to put as little load as possible on the backups so that they can be used for other purposes, too. In particular, it does not immediately apply all state changes. An EJB might be changed by subsequent transactions – applying all these changes would waste resources because at the time of failover, only the last state is relevant.

When the backup receives a *committing* message it temporarily stores it in a queue. Upon receiving the corresponding *committed* message, the *committing* message is parsed, and the request/response pair stored in a hashtable. For all EJBs listed in the message, if they have not yet been initialized, this will be done. Additionally, for each existing EJB, the last state is kept as presented in the message (not applied to the bean).

3.2.4 Failover protocol

When a backup receives a view change message from the GCS, it checks whether the current primary is still member of the view. If not, it determines whether it is the new primary (using any deterministic mechanism). If it is the new primary, the failover protocol starts. During failover, client requests asking whether this is the new primary, are blocked. Once failover is completed, the replica will confirm that it is now primary.

A simple failover protocol does the following. For each *committing* message for which no corresponding *committed* message was received, we check whether the corresponding transaction has committed at the database or not. For that, we check whether we find the corresponding $txid$ marker in the database. If we find the marker, the database transaction has committed, and hence, the changes on the EJBs indicated in the message have to be considered. If there is no marker, then the primary crashed after sending the *committing* message and before committing the database transaction. Hence, we discard the content of the message. Then, for all initialized SFSBs, we set the state to last recorded state. For all initialized EBs, we load the state from the database (can be done upon first request on this EB). Then, the replica switches to the primary protocol.

At this time, we briefly want to discuss why the combination of primary, backup and failover protocols guarantees state consistency and exactly-once execution in case of primary crash. Recall that we assume that if the primary crashes all database transactions active at the primary will be automatically aborted at the database. The primary might crash at several places during the execution of a transaction. (i) If the primary crashes before sending the $committing$ message, the backups do not have the SFSB state changes, and the database transaction will be aborted. This provides state consistency. When the

client resubmits the request to the new primary, it will simply be handled as a new request. (ii) If the primary crashes after sending the $committing$ but before committing the transaction, the backups have not received the $committed$ message. Hence, at failover, the new primary checks at the database and will detect that the corresponding $txid$ cannot be found in the database. Hence, it knows that the transaction has not committed, and will disregard the $committing$ message. Again, the system behaves as if the request had never been processed. (iii) If the primary fails after committing the database transaction but before sending the $committed$ message, the backup will again look for the marker in the database. This time, it will find the corresponding $txid$. Hence, it will consider the state of the EJBs in the $committing$ message. Again, application server and database have a consistent state. When the client resubmits the request to the new primary, it will not be re-executed but the response will be immediately returned. (iv) When the primary fails after sending the $committed$ message, the backup does not need to check for the marker to know that the transaction has committed. Our algorithm provides exactly-once execution and state consistency by using the marker mechanism to determine whether the database transaction has committed, and by resubmitting client requests if the primary crashes before the client receives a result.

Our failover implementation does not exactly follow the approach above but only lazily applies state. That is, during failover, we do not set the state of all beans. Instead, we switch to normal processing once all *committing* messages are handled. Then, the state of SFSBs is lazily set whenever a bean is accessed for the first time by a new request. This procedure slows down request execution shortly after failover but makes the failover itself much faster.

During normal processing, a transaction can abort at several places, e.g., when the TM executes the commit request or in the middle of the execution of an EJB method. Depending on the degree of state consistency this will require to send according information to the backups in one or both cases. The important thing is that if the transaction aborted because of regular application semantics, if full state consistency is requested, none of the replicas may install the EJB state changes triggered by this transaction. For space reasons, we do not discuss the various scenarios in detail but refer to [WKM04].

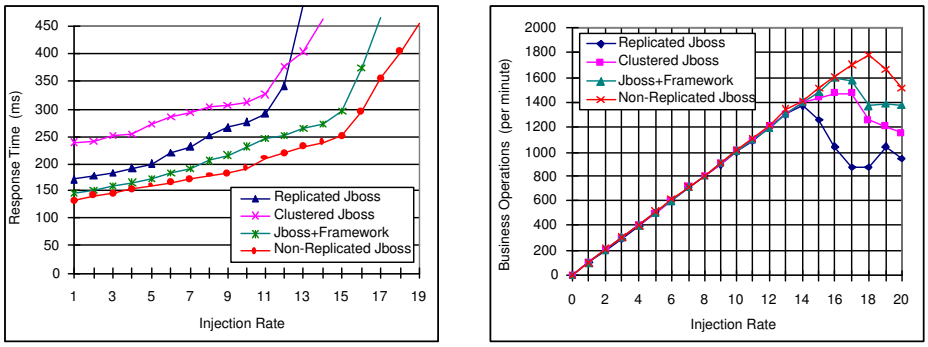
3.2.5 Recovery protocol

When a failed replica recovers or a new replica joins it has to first receive the current state, and then will become a backup. Currently, our implementation requires that at least one backup already exists. Thus, when a new or failed replica joins, one of the existing backups, referred to as the *peer replica* will send its current state to the joining replica. We use an agreement protocol among the backups to agree on the peer. The chosen peer generates a *recovery* message containing all *committing* messages, all request/reply pairs, and the necessary information about all changed EJB using point-to-point communication. While waiting for the *recovery* message, the joining replica might have already received messages from the primary (it starts receiving messages when the GCS delivers the view change). A special mechanism is used to determine whether the content of some of these messages is already contained in the *recovery* message sent by the peer replica.

3.3 Performance Evaluation

We implemented the algorithm based on the ADAPT framework taking advantage of the interceptor points, and the set/get state methods on beans. As group communication system, we used JBora on top of Spread (see D2 for details).

We evaluated our system running three different suites of experiments. First, we use the ECperf benchmark [Sun03] to evaluate the performance on a “real” application and compare it with JBoss’s



(a) Response Time (b) Throughput

Figure 4: ECperf Comparison

existing clustering technique. For this case, we used JBoss 3.2.3. A second test suite presents a series of micro benchmarks that show the performance for different components (SFSB vs. EB), and database access patterns. The third experiment suite evaluates failover. All tests were run on a cluster of PCs (3.0 GHz Pentium 4 with 1 GB of RAM) running RedHat Linux. The configuration consists of one machine emulating clients, one machine running the web server (if needed), two machines running JBoss application server instances, and one machine running DB2 as our database system.

3.3.1 Evaluation based on ECperf benchmark

ECperf [Sun03] emulates businesses involved in manufacturing, supply chain and order/inventory management. The application is split into four domains: customer, manufacturing, supplier domain, and corporate. The main configuration variable is the *transaction injection rate* (IR) which refers to the rate at which a specified subset of business transaction requests are injected into the system.

In this experiment, we evaluate the following architectures. (1) A regular, non-replicated JBoss server as baseline for comparison. (2) The JBoss server including the framework without the replication protocol to evaluate the overhead of an abstraction layer useful for reusability and platform independence. (3) Two application server replicas using our eager replication protocol. (4) Two application server replicas using JBoss's own clustering solution [1]. For both (3) and (4) we did not take advantage of load balancing, and submitted all requests to one server.

Figure 4 shows the results of the experiment measured over the steady state phase of the run (the ramp-up and ramp-down phases are ignored). Figure 4(a) shows the average response time for order entry transactions of the customer domain. At low throughput, the framework adds around 10 ms to the non-replicated JBoss, our protocol adds 25 ms while JBoss's clustering method adds around 100 ms. This gives an overhead of around 25% for our protocol plus the framework, and 70% for the JBoss clustering. The latter performs so badly because it sends state after each method invocation while our solution only communicates at the end of the transaction. As a comparison, in [MMSN+99] the authors

[1] JBoss's clustering solution [JBo] uses passive, warm, and eager replication. Each replica can act as a primary for a client session. If a client request triggers execution on several stateful components, state transfer takes place individually once execution on the component has terminated. Although eager, problems occur if state propagation for some of the components was successful but the primary fails before committing the transaction at the database. In this case the backups have a partially replicated state while the database transaction aborted. Hence, neither state consistency nor exactly-once semantics are provided.

indicate around 15% overhead for FT-CORBA (primary-backup) compared to non-replicated CORBA. With increasing IR, the response time in all systems increases steadily until saturation point. The gap between non-replicated and replicated JBoss increases slightly but steadily, while it remains nearly the same for the clustering approach until around 11 IR beyond which it becomes significantly worse. More information about the saturation point can be found in Figure 4(b). This figure uses the average *business operations per minute* to represent the maximum achievable throughput when the IR increases. The maximum in each curve shows the system shortly before saturation. The replicated server is saturated at an IR of 15 (due to CPU overhead). JBoss's clustering saturates at 17 IR (also due to CPU) while the non-replicated JBoss saturates at 19 IR (in this case, DB2 is the bottleneck). The reason for earlier saturation of our approach compared to the clustering approach is higher CPU overhead (keeping old responses to guarantee exactly-once, keeping information during transaction execution in order to send all state in a single message).

In summary, we believe that our approach provides acceptable performance considering the strong consistency guarantees that it provides. It compares favorable with JBoss's clustering mechanism. Nevertheless, the overhead is not negligible. We believe, however, that more "engineering" work in optimizing our in-memory data structures could lead to further improvement.

3.3.2 Component Analysis

In order to better understand where to start such optimizations, our second experiment suite looks at the overhead of replication for different components and component combinations. We evaluate cases (i) where no database access takes place, (ii) where the database is accessed but no conflicts occur, and (iii) where database conflicts occur. We tested with only SFSB, and with a combination of SFSB and EB. Detailed results are given in [WKM04]. In case (i) execution is very fast both for the replicated and non-replicated system, and less than 10 ms. However, the overhead of replication is nearly 100% and a replicated system has only half the maximum achievable throughput than a non-replicated system. Since no database access takes place, the system is completely CPU bound. Since the the non-replicated system takes half the time to execute one request compared to the replicated system, it can execute double as many requests before saturation. In (ii) the replication overhead is around 20%. Response times are generally higher. The saturation point of the replicated system is at around 75% of the saturation point of a non-replicated system. In (iii), response times and max. achievable throughput are generally much worse than in (ii) due to blocking behavior at the database. The relative behavior of the replicated system compared to the non-replicated system is, however, similar to case (ii). In all cases, using EBs slows down both replicated and non-replicated systems in similar ways compared to a system with only SFSBs.

3.3.3 Failover

We evaluated the failover costs after different running times of ECPerf. Since failover time only depends on the number of *committing* messages for which no *committed* message was received, it is independent of the running time before the crash but depends on the load shortly before the crash. If the load was high, many *committed* messages are missing. At medium loads, the failover time was around 100 ms, at high loads it rose up to 160 ms. This can be considered as very fast failover.

3.4 Related Work

Most J2EE products provide some form of EJB replication. WebLogic [BEA02] uses passive, warm, and lazy replication. A single primary server processes requests, and propagates the state soon after returning the response to keep replicas as consistent as possible. JBoss's clustering solution [JBo] uses passive, warm, and eager replication. Each replica can act as a primary for a client session. If a client request triggers execution on several stateful components, state transfer takes place individually once execution on the component has terminated. Although eager, problems occur if state propagation for some of the components was successful but the primary fails before committing the transaction at the database. In this case the backups have a partially replicated state while the database transaction aborted. Pramati [Pra02] uses passive, cold, and eager replication. Each replica can be the primary. State changes are immediately written into the database. If the primary crashes in the middle of execution, the database transaction aborts and with it the state changes of the application server. None of the above solutions provides exactly-once semantics, and only Pramati guarantees state consistency.

As an example of replication in CORBA, the Eternal system [NMMS01] is based on the FT CORBA architecture [OMG00]. Eternal supports active replication and both warm and cold passive replication. Determinism is required even for passive replication since it uses lazy propagation. The primary replicates the state to backups periodically in form of checkpoints. Between two checkpoints, all messages from clients and the database are logged. At failover, the new primary first restores the state of the last checkpoint, and then replays logged requests. During replay, outgoing messages are suppressed. To guarantee exactly-once semantics at the database level, Wei Zhao et al. [ZMMS02, ZMMS03] extend the Eternal system to use a distributed out-bound gateway to replicate transactional context. Phoenix/COM+ [BLW02] is based on .NET using passive, cold, and lazy replication. It has similarities to the Eternal system. States are replicated periodically, and requests between two checkpoints are logged. However, it distinguishes nondeterministic events from deterministic events. For nondeterministic events, it uses eager replication to avoid the problems that exists in Eternal. However, it is unclear how to determine whether an event is non-deterministic or not. Neither Eternal nor Phoenix/COM+ explicitly discuss state consistency.

There are also some general solutions that are not developed within the context of a specific application server architecture. e-Transactions [FG00a] offer exactly-once semantics for stateless application servers. When a replica of an application server executes a request, it inserts the response and a marker into the database. If the server crashes before sending the response and the client resubmits the request to the new primary, the new primary checks whether a marker for this request exists in the database. If yes, the response will be retrieved and returned without re-executing the request. X-ability [FG00b] provides a general replication solution for stateful servers, however, it is presented on a very abstract level.

Eternal, Phoenix/COM+, e-Transactions, and X-ability are all implemented in different environments with different conditions. In our approach, we have taken advantage of some of J2EE's properties leading to a different solution.

3.5 Current Work

Our current work attempts to enhance the current system in several ways. (i) We are extending the system to handle network partitions. This is quite simple if the network partition occurs between client and server cluster. However, when it occurs between the servers, things are more complicated because then a client might resubmit a request to a backup or a backup becomes a new primary although the old primary has not failed. (ii) We are implementing the algorithm for relaxed state consistency. (iii) We

want to implement the version which allows multiple database instances and a 2-phase-commit protocol. (iv) We are evaluating the impact of different failover strategies and the costs of recovery. (v) We are looking into more advanced transaction models. For instance, a transaction can span more than one request, or a request can span more than one transaction. In the first case, failover must be able to handle transactions for which a client has received some but not all responses. In the second case, failover must be able to handle requests for which some but not all transactions have committed. These transactions can be BMT or CMT. We are also planning to support the ONT model described in D4 (or rather, the successor of D4). (vi) The system has to be extended to handle web-service interfaces. Currently, only RMI clients are supported. (vii) We will provide a module to connect the replicated application server with one of the replicated database systems we have developed.

4 Web-service Object replication

The second algorithm developed within the ADAPT framework focuses on the replication of Axis-based web-service objects. While it is similar in many regards to the EJB replication algorithm described in the previous section, the focus of the work has been quite different. A detailed description of the work presented in this section can be found in [BPA04]. We only summarize the results here.

4.1 Model and Assumptions

We will outline the model and assumptions by pointing out the commonalities and differences to the EJB replication algorithms. In the web-service replication algorithm, the components to be replicated are Axis web-service objects. These objects can be session oriented and maintain state of the lifetime of a session. In this case, each object is always associated with a single client. The web-service algorithm currently completely focuses on the replication of this object state. That is, execution of methods may only change the state of the object and must not have any side effect outside of this object. In particular, calls of this object to EJBs or a database are not considered.

While this model is somehow simpler than for EJB replication (no database access, no subrequests, no transactions), the properties to be supported are quite similar. (i) The replicated system must provide the same consistency as the non-replicated system in a non-failure case. In particular, the current implementation provides sequential consistency (given any set of operations, the execution of these operations produces the same results as if these operations were executed in some sequential order; and, the operations for each client appear in this sequence in the same order in which they were issued by the client). (ii) Exactly once execution is provided. If the replicated environment receives multiple copies of a given update request, the operations should only be executed once. Moreover, the system responds to each copy of the request with the result of that only execution.

Compared to the EJB algorithm, the web-service algorithm provides correctness not only in the case of server crashes, but also in case of network failures. Network might partition between the client and the server replicas, or between the individual server replicas. Handling network partitions of the first case is especially important when clients access the service through the Internet which is the typical scenario for web-services. Networks partitions in the second case can occur if the servers reside on different subnets within a company. A crashed process may recover as a new process, and partitions are eventually repaired. The system is asynchronous in that no bounds are assumed on communication delays or relative speed of processes. Message corruption and byzantine failures are excluded.

Objects export read and write operations. Write operations are neither assumed to be idempotent nor commutative. The application programmer is assumed to use a special naming mechanism to indicate

whether an operation is a write or a read. The replication overhead takes only place for write operations.

4.2 Replication Algorithm

4.2.1 Client Protocol

The basic task of the client protocol is to forward a client request to any server replica. The addresses of server replica can be obtained from external naming services. Furthermore, for each object belonging to the client, the client keeps a counter indicating the number of updates performed on the object. This counter is piggybacked with each request sent. This allows the server replica to check whether its copy of the object is up-to-date. The client protocol attaches a unique request ids to requests. The response to a request can be OK, and then is returned to the client. It can also be UNABLE indicating that the contacted server replica does not have an up-to-date version of the object or is not allowed to currently perform operations. Furthermore, it can be UNKNOWN indicating that another server replica might have executed the response but this replica does not have the response. In the last two cases the client protocol might either try another server replica or give up.

4.2.2 Server Protocol

Each server replica keeps for each object the number of updates that have been performed on the object (as far as the replica is aware of), and the last response to the client.

Group Membership Servers communicate with each other using group communication. See D2 for details. We assume that the group communication system provides primary-partition membership, i.e., one in which all group members have the same perception of the group membership. For example, in case the set of replicas splits in two or more disjoint sets because of a network failure, then the GC-layer automatically selects one of these sets to be the next view and forcibly expels from the group the replicas that happen to be on the wrong side of the partition (by delivering to these replicas a special ”shutdown” view change). All replicas receive view changes in the same order. Our prototype selects the primary partition as the one containing a majority of statically known number of replicas. Replicas that enter the primary partition (i.e., those that recover after a failure or that were isolated because of a network failure that has repaired) can execute client requests only after acquiring the up-to-date version of all relevant information through a procedure called state transfer.

Normal Processing Each client is connected with one single replica at a time, say R. R can execute a request submitted by the client only if the request identifies an object for which R holds a version of the object that is more recent than the latest version accessed by the client. It first checks whether the replica is “sufficiently up-to-date” for the specified object. If not, it responds to the client by declaring that the replica is UNABLE to execute the request. Otherwise, if the request is a read, it executes the request completely locally and responds. If the request is an update, the replica first inspects whether the replica is in the primary partition and the state transfer to that replica has completed. If this is not the case, the replica responds to the client by declaring that the replica is UNABLE to execute the request. Otherwise, the replica inspects whether the update request is a duplicate or is fresh. If it is a duplicate, it sends immediately the result to the client, without carrying out the update again. If it is fresh, it (i) executes the update on a copy of the object; (ii) propagates the new version and associated result to the other replicas using total order uniform reliable multicast, waits to receive the message locally, and then

sends the result to the client. A client that does not receive the response to an update or that observes the connection breaks before receiving such a response, may submit the very same request immediately, to either the same replica or another one. There is no hypothesis whatsoever on the retransmission policy used by clients.

The reason why each update is performed on a copy of the object is because the propagation at step (ii) might fail. In particular, the replica might issue a multicast and then receive a“shutdown” view change before receiving the corresponding message. In this case the replica is forcibly expelled from the primary partition and it cannot tell whether the replicas in the primary partition have received the message, and thus updated the object accordingly, or not (both outcomes are possible, depending on the protocols within the GC layer and on ”when” the failure occurred) . It follows that the replica must drop the updated copy of the object and must respond an UNKNOWN status to the client.

Recovery Finally, upon receiving a view change, a replica checks whether any replica (including itself) has just entered the primary partition. In this case, all replicas in the primary partition execute a function which triggers the state transfer from a replica with an up-to-date version to all new replicas. It might be impossible to determine whether there exists a replica with an up-to-date version. This is the case, e.g., when the group reforms after the primary partition ceased to exist and at least one server replica has crashed since the last existence of the primary partition. Such solutions require that each server replica saves (part of) its state on stable storage, and are currently not implemented. State transfer takes place without suspending the execution of the service.

4.3 Implementation

The implementation described in [BPA04] and summarized in the following was based on Apache Tomcat. The algorithm has been implemented based on the ADAPT replication framework developed for JBoss. The implementation described in [BPA04] was based on Apache Tomcat without using the framework. Each server runs an instance of Tomcat/JBoss and a Spread daemon. Each server is also equipped with two Java packages developed by us: JBora, that implements our group communication interface (see Deliverable D2); JMiramare, that implements the replication algorithm on top of JBora. JMiramare is for its most part Tomcat-independent, i.e., it may be executed unchanged in other servlet containers (this is the portion that has been totally rewritten for use with the replication framework). Each client is equipped with a small stub that includes within the requests the information required by the replication algorithm and that takes care of retransmissions and fail-over as appropriate. For the web-service implementation, SOAP messages include a replication header for storing all necessary information. It is possible that the WSDL description of web service contains the information whether a method is a read or a write.

4.4 Performance Evaluation

A performance analysis of the Tomcat-based prototype is given in [BPA04]. The results refer to a service consisting of 3 replicas, each placed on a Dell Optiplex GX260 (PIII 800 MHz, 512 MB) running Windows 2000 Professional. Communication occurs through a 100 Mb switched Ethernet. We used Sun Microsystems’ JVM 1.4.0 and Tomcat 4.0.6. We configured the JVM executing Tomcat with -Xms128m (initial heap space 128 MB) and -Xmx384m (maximum heap space 384 MB). All experiments began with a warm-up phase of a couple of minutes. Data collected during warm-up were discarded. We simulated an increasing number of clients by running a publicly available tool on another machine on

the same Ethernet (http://grinder.sourceforge.net). Each simulated client constructs a request, waits the response and sends the next request. Construction of a request involves parsing the response to the previous request. All the results below have been obtained by filtering out data collected during warm-up. We have verified that the client machine was not the bottleneck in any experiment.

Salient results of this analysis include what follows. First, usage of the standard Java serialization mechanism is likely to deliver poor performance. This mechanism leads to a rather verbose and redundant description of objects. Since replication makes heavy use of multicast communication, the resulting overhead turns out to be excessive. Usage of customized serialization procedures is therefore highly advisable, and implemented in our prototype. Another interesting finding is that, with 100%-write workload, the replicated implementation delivers a better throughput than the non-replicated one (single replica). This positive result was unexpected. In the replicated implementation each replica has to participate in the execution of all requests (recall that we are considering 100%-write workload). Thus, we expected that the throughput of the non-replicated implementation (single replica) would be an upper bound for the throughput of the replicated implementation. In fact, since receiving a multicast involves a significant cost at the group communication layer, we thought the upper bound would not even be achieved. This expectation turned out to be wrong. The reason is because we underestimated the cost of receiving and parsing HTTP requests, constructing and sending HTTP responses. In the replicated implementation this cost is fairly distributed amongst replicas, each replica being responsible for one third of the total number of requests. As it turns out from our experiments, distributing this cost may partly compensate the overhead intrinsic to replication. Finally, we have verified that when the workload includes "read" operations the throughput of the replicated implementation indeed grows. We expect that when the workload is close to 100% "read", the replicated implementation should exhibit a close-to-linear scalability, i.e, a throughput close to k times the throughput of the replicated implementation, k being the number of replicas. However, we did not address this scenario in our experiments. Latency results showed an increase of around 12 ms for the replicated system compared to the non-replicated system. Most of this time is spent for the mulitcast. In our setting this was in increase of around 30%. However, since in a real system the client is probably connected through a slow wide-area link, the 12 ms additional overhead will barely be recognizable by the client.

We want to point out that by making explicitly visible the nature of each operation ("read" vs. "write"), this information can be exploited by the replication infrastructure in order to improve performance. When this information is not available, a replication infrastructure can only handle each operation as if it were a "write", or perform expensive state comparisons, thereby introducing unnecessary overhead.

4.5 Related Work

Our algorithm is such that the service stops being available when there is no majority of replicas that are active and mutually connected. That is, when an excessive number of failures occur, the service stops responding to requests because it is no longer able to guarantee state consistency and exactly-once execution (in practice, the service will have to reboot as a new incarnation). While this feature is not peculiar of our approach (e.g., [FG01]), it is worth to recall that many replicated services take quite a different approach, in which certain failures may cause the system to silently stop guaranteeing the above properties — the system continues to respond but its behavior no longer satisfies one or both of the properties, without any explicit notification of this fact. As an example, consider the session failover support in replicated servers based on IBM WebSphere [UAB+00]: the conversational state for a client (i.e., an HTTP session) is kept in a database shared by all replicas, so that it remains available

in case the replica connected with client fails; since session information is accessed very frequently, a caching mechanism is used to decrease the overhead related to database access; this mechanism is such that, should a failure occur within a certain vulnerability window, some updates already seen by the client could be (silently) lost. As another example, consider some recent implementations of distributed data structures (DDS) [GBHC00]. A DDS is an object (e.g., a hash table) partitioned and replicated across several replicas. The service implementation, that ensures excellent performance and scalability, is based on the assumption that the network between replicas never partitions. Should such an event occur, the service could silently stop satisfying its consistency criterion. While the above design choices are sensible in many environments, in particular, where state-of-the-art performance and scalability are essential, we are interested in exploring other design trade-offs, more suitable for application domains where it may be preferrable to eliminate the potential for inconsistencies even at some cost in terms of performance and scalability. Indeed, many environments do not need state-of-the-art performance and scalability [VvRB98] and leading research groups believe that "it is time to broaden our performance-dominated research agenda"[PBB+02].

5 Database Replication: An overview

For a long time, database replication has been considered an excellent solution to increase throughput (more replicas can serve more requests), decrease response times (distribute the load), and provide fault-tolerance. Replication, however, has the challenge of replica control. When one replica is updated, the changes have to be propagated and applied at the other replicas, and the different copies of the database must remain consistent despite concurrent updates. Standard correctness criteria is 1-copy-serializability, i.e., the concurrent execution of a set of transactions on the different replicas should have the same effect as a serial execution on a centralized database. As such, replica control has to be combined with or at least must be aware of the concurrency control mechanisms used to determine the execution order of operations at each individual replica. Early research solutions focused on fault-tolerance [BHG87], and were seldomly implemented in commercial systems, which mostly offered ad-hoc solutions violating traditional transactional properties in order to achieve acceptable performance [Gol94]. A thorough analysis by Gray et.al. in 1996 [GHOS96] claimed that existing approaches scale badly, and are not suitable for modern applications. Their analysis revived research in database replication leading to many new solutions that attempt to eliminate the limitations pointed out by [GHOS96], while still providing global serializability, e.g., [ABKW98, PGS98, BKR+99, HAA99, BGRS00, KA00a, KA00b, PMS99, JPPMKA02, ACZ03b, ACZ03a, CMZ03].

Database replication can be implemented at two levels, each level with its own advantages and disadvantages. The data replication tool can be implemented as a middleware on top of off-the-shelf database systems, or it can be integrated into the kernel of a database system. In a middleware based approach, the database system itself, often needs no changes. Instead, all transactions have to be submitted to a middleware layer which coordinates the execution on the database replicas. This has the advantage that replica control is an additional module and separated from the complex database kernel. Additionally, it might have the potential of being able to work in a hetereogenous environment, although few solutions support this. On the negative side, the middleware layer has to reimplement concurrency control since it has no control over the concurrency control module within the database systems. Since the middleware typically only has access to the SQL statements submitted by the transactions and does not know the concrete tuples to be accessed by these statements, concurrency control is typically on a rather course level, e.g., a table. If the middleware has a centralized component, e.g., a centralized schedule, this sin-

gle point of failure is an additional disadvantage. Also, the middleware is yet another indirection in the execution flow leading to an even more complex architecture.

In contrast, when the replication tool is integrated into the database system, more options for optimization exist. The replication tool can take full advantage of its access to other internal components and the homogeneity of the system, and hence, is hopefully more efficient. For instance, it can directly interact with the often tuple-based concurrency control of the database system, and does not need to implement its own concurrency control mechanism. Furthermore, direct access to the tuples and/or logs is given allowing for an efficient propagation of changed tuples. Another advantage is that the replica tool comes within the same software package as the database system making installation and usage easier. Commercial systems are able to sell their replication modules at high price for exactly these reasons. As its disadvantage, an integrated solution only works in a homogeneous environment. Furthermore, it has the challenge, that the replication tool should ideally be an *addition* to the system without major changes to the existing component. In particular, in case the extended system runs in non-replicated mode, replication related code should not be executed and the semantics and principle protocol of the centralized system should remain unchanged.

We have developed database replication tools both at the middleware layer and as integrated solution. In the following, we will present both of the approaches and current activities.

6 Postgres-R(SI): An integrated database replication solution

6.1 Introduction and Overview

6.1.1 Postgres-R: the predecessor

A first version of Postgres-R was already developed in 2000. It integrated replication into the kernel of PostgreSQL, version 6.4. At this time, PostgreSQL's concurrency control was based on strict two-phase locking. Postgres-R integrated a replication algorithm based on group communication (GCS). The execution is as follows. A transaction T_i, consisting of a sequence of read r(X) and write w(X) operations on tuples X can be submitted to any replica. This replica is T_i's *local* replica and T_i is local at this replica. All other replicas are *remote* replicas for T_i and T_i is *remote* at these replicas. T_i is first completely executed at the local replica, and write operations are collected within a *writeset*. At commit time, the writeset is multicast to all replicas using the total order multicast. All replicas now use the total order delivery to determine the serialization order. Whenever two operations conflict they will be executed in the order the writesets were delivered. Since this is the same at all replicas, all replicas serialize in the same way. No complex agreement protocol or distributed concurrency control is necessary. In the locking based approach of Postgres-R, when a writeset of a transaction T_i was delivered and there was a local transaction T_j whose writeset was not yet delivered and it had a read or write operation that conflicted with one of the write operations of T_i, T_j was aborted. If T_j had already sent its writeset, the other sites have to be informed about the abort. This guaranteed 1-copy-serializability. Furthermore, uniform-reliable delivery is used to avoid lost transactions. When the sender receives a writeset itself it knows that everybody else will or has received it. Only one total order message is sent within the transaction boundaries. 2-phase-commit is avoided, and a transaction can commit locally without waiting that other replicas have executed the writeset. When replicas fail, the GCS informs the remaining replicas. They simply can continue as a smaller group. This approach avoids many of the limiations pointed out by [GHOS96], at least for local area networks.

While failures are handled transparently by the GCS, recovery requires to transfer the current database

state to the recovering replica. [KBB01] proposes a suite of recovery mechanisms for this purpose. In [Cho03], one of these approaches is implemented into a master/slave version of Postgres-R.

The PostgreSQL community showed great interest in Postgres-R. The project became an open-source effort hosted at the GBorg website (http://gborg.postgresql.org/project/pgreplication/projdisplay.php) in a plan to move the prototype to a production mode system.

6.1.2 Snapshot Isolation

However, since the development of Postgres-R, PostgreSQL has moved to a new version 7 with a completely new concurrency control module. Now, PostgreSQL does not provide anymore classical serializability, but provides the isolation level *Snapshot Isolation*. This is a very common isolation level, implemented in similar way in Borland [Bor04], Oracle [Ora01], and also offered by Microsoft SQL Server. This isolation level can be implemented using multi-version concurrency control. Snapshot isolation allows some non-serializable executions, but it provides much more concurrency for read-only transactions, and hence is very useful for read intensive applications. We can expect that even more database systems will provide this isolation level in the future. The basic idea is to keep several versions of a data object. Read operations read from a committed "snapshot" of the database, and work completely independent from writes. Conflicts are only detected between write operations. We denote an object version to be committed at the time the transaction that created the version commits. Furthermore, we define two transactions to be concurrent if neither terminated (commit/abort) before the other started. A concurrency control system providing SI must obey the following rules. (i) Each first write operation $w(x)$ of a transaction T on object X creates a new version of X, (ii) subsequent read $r(X)$ and $w(X)$ of T on X access the newly created version, and (iii) a $r(X)$ of T on X not preceded by a $w(X)$, reads the last version that committed before T started. Finally, if two concurrent transactions write object X, at least one of them must abort.

6.1.3 Postgres-R(SI)

As a result, we had to redesign Postgres-R's replication algorithm to work with the new concurrency control method of PostgreSQL, v.7. Posgres-R(SI) has a similar architecture as the original Postgres-R using a total order multicast. Also the execution model remains the same (first local execution, then sending the writeset determining the global serialization order). What is new is the integration of replica control with the multi-version concurrency control algorithm of PostgreSQL, version 7. While the locking based concurrency control of version 6 was encapsulated in a lock manager, the current algorithm heavily depends on PostgreSQL's multi-version storage system to detect write/write conflicts and determine the snapshot of read operations. This required us to obtain a detailed understanding of PosgreSQL's multi-version system, and made it more challenging to extend PostgreSQL in a modular way without really changing existing components. In the following, we shortly outline our algorithm, its implementation, and a performance evaluation we conducted. For more detailed information, we refer to [WK04].

6.2 Replica and Concurrency Control

6.2.1 Concurrency control in PostgreSQL

In PostgreSQL, each tuple X is assigned a unique identifier which is common to all versions of X. Each update creates a new version of X. Concurrency control is a mixture of reading different object versions, performing checks, and acquiring exclusive locks for write operations. Each transaction T_i has

two phases. In the execution phase it executes read and write operations. When T_i performs a write operation $w_i(X)$ on tuple X for the first time, it first performs a *version check*. It checks whether there is any concurrent transaction T_j that updated X and already committed. If this is the case T_i aborts. Otherwise, T_i requests an exclusive lock on X. If the lock is granted, T_i creates a new version of X and performs the update on the new version. When T_i performs a successive write $w_i(X)$, it simply uses the previously created version. If there is already a lock, T_i's request is appended to a waiting queue for X. Upon being woken up by the transaction releasing the lock on X, T_i starts all over again with the version check. When T_i performs a read operation $r_i(X)$, it either reads its own version if existing, or it reads the version created by transaction T_j such that T_j committed before T_i started and there is no other transaction T_k that updated X and committed after T_j committed and before T_i started. The second phase is the termination phase and very fast. Upon the commit request or abort for T_i, the necessary logging is performed, T_i releases all locks, and wakes up all transactions waiting for one of these locks.

Note that the version check for write operations happens before requesting the lock and will be repeated if the lock can not be immediately granted. The locking procedure serializes conflicting write operations. When a transaction T_i holding the lock commits and wakes up a waiting transaction T_j, T_j performs again the version check which will fail since T_i is concurrent and committed. If T_i aborts, T_j's second check will succeed, and it will again request the lock.

6.2.2 Replica Control for PostgreSQL

In a replicated environment, we have to distinguish between local and remote transactions. As in the centralized concurrency protocol of PostgreSQL, local transactions perform conflict checks and read and write operations step by step whenever a statement is submitted. Remote transactions, however, only have write operations that are all known at the time of writeset delivery. Furthemore, we must guarantee that conflicting operations of both local and remote transactions are executed in the order of writeset delivery. In order to achieve this without adding to much complexity, we decided to execute all remote transactions serially. Or more precise, whenever a writeset is delivered for either local or remote transaction, the transaction has to completely terminate before the next writeset is delivered. We denote this as atomic in the outline of the algorithm given below. The description below is not exactly the algorithm implemented in PostgreSQL. In order to shorten the text we had to simplify some issues changing the semantics slightly at some points. See [WK04] for the exact algorithm.

Transaction identifiers In order to perform checks and retrieve tuple versions, PostgreSQL labels each version with the local transaction identifier (TID) of the transaction who created the version (and other identifiers which we do not discuss for lack of space). TIDs, however, are individual assigned by different database instances. Hence, the TIDs for the same transaction might differ at different replicas. Many components of PostgreSQL use $TIDs$ in different ways, and hence, we do not want to change their generation. Therefore, each transaction keeps locally its TID. At the same time, each update transaction will receive a global identifier GID, which will be the same at all replicas. The GID of a transaction will be used to match the different local TIDs created on different replicas. We generate GIDs without extra coordination overhead by using the total order in which writesets are delivered. We keep a GID counter at each replica. Whenever a writeset is delivered, the counter is increased and its current value assigned as GID to the corresponding transaction. Furthermore, each replica keeps an internal table that allows for a fast matching between TID and corresponding GID. Note that for a local transaction, TID and GID are generated at different timepoints (start and later writeset delivery), for a remote transaction both are generated at the same time (writeset delivery).

Execution of local transactions For local transactions, the execution phase is the same as before. Additionally, the updated tuples are collected in a writeset. When T_i submits the commit request, and T_i is read-only, it commits immediately. Otherwise the writeset WS_i is multicast to all replicas using total order multicast. The writeset also contains the set G which lists the GIDs of those transactions that are not concurrent to T_i, i.e., that terminated before T_i started. The atomic commit phase for T_i starts upon delivery of the writeset. If T_i has not been aborted so far, the GID_i is generated, and recorded. The rest is the same as the commit in the centralized case. If T_i is aborted sometime during execution and before receiving its own writeset, it releases all locks. If the first waiting transaction for a lock is a remote transaction, only the remote transaction is woken up. Otherwise, all waiting transactions are woken up. (Explanation see below).

Execution of remote transactions For remote transactions the entire execution is atomic. Upon delivery of a remote writeset WS_i with all updated tuples and set G, a transaction T_i is started, its GID_i generated, and recorded together with TID_i. For each tuple X in the writeset, T_i checks whether there exists a committed version labeled with a transaction whose GID is not in G. If this is the case T_i aborts. If no such version exists and there is currently no lock on X, then T_i gets the lock and performs the update. If there are locks, execution on X is delayed until all tuples in the writeset have been checked. If all checks have been successful, and there is a delayed update on a tuple X, T_i sends an abort request to the transaction holding the lock. This transaction must be local (no concurrent remote transactions), and its writeset is not yet delivered (because this would have lead to immediate commit). The local transaction, upon completion of the abort, will grant the lock to T_i. Once all updates have been performed, T_i commits, and releases all locks. The next writeset can be processed.

Discussion Note that we only delay the execution for tuples for which a local transaction holds the lock. All other updates are performed at the same time as the version check, avoiding to access a tuple twice whenever possible. Note also that there is no version check upon writeset delivery of a local transaction. If there is any conflict between a local transaction T_i and a remote transaction T_j whose writeset is delivered before T_i's writeset and which committed, T_i would have been aborted by T_j. Hence, upon delivering the writeset for T_i, if it is still alive, it has already passed the version check implicitly. Note also that a local transaction can commit when its writeset is locally delivered. It does not wait until the corresponding remote transactions have executed. The GCS guarantees that the writeset will be delivered, and hence, executed at all replicas.

Although a local transaction on replica R does not need to wait until the corresponding remote transactions on the other replicas have executed, it might be indirectly delayed by the serial execution of remote transactions executing locally on R. We suggest to perform what we call a *pre-lock phase* that does not need to access tuple versions but which needs a separate lock table LT completely independent from the locking mechanism in PostreSQL. For space reasons, we do not discuss this approach here but refer the interested reader to [WK04].

6.3 Implementation

Writeset The writeset contains for each SQL data manipulation statement (i.e., update, delete, insert) for each changed tuple all modified attribute values and the corresponding attribute identifiers, and the primary key values of the tuple. For SQL data definition statements (e.g., create table, create function,

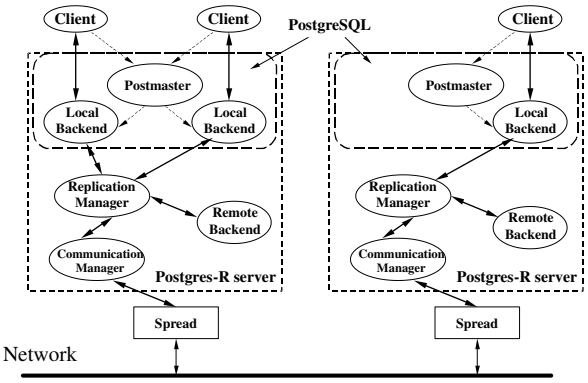


Figure 5: Architecture of Postgres-R

etc.) the writeset simply contains the query text [2]. Remote replicas process the statements in the order they appear in the writeset. For DDL statements, the execution path is the same as it is for a local transaction (parser, planner, etc.). For DML statements, for each tuple to be changed, the remote backend retrieves directly the valid version of the tuple using the index on the primary key skipping most of the normal planning and execution steps.

Architecture Figure 5 depicts the architecture of Postgres-R(SI). In PostgreSQL, the postmaster process listens for a connection request from a client, and then creates a dedicated backend process which will connect to the client and execute its transactions. Postgres-R(SI) extends PostgreSQL with three new processes: *remote backend*, *replication manager* and *communication manager*. The original backends are now called *local backends*, and execute local transactions. A remote backend processes the writesets of remote transactions. The communication manager's only purpose is to hide the details of the GCS (currently Spread). We will not mention it any further. The replication manager (RM) is the coordinator of the replica control algorithm. The local backends create writesets which are forwarded through the replication manager and the GCS to the other replicas. When the replication manager receives a writeset of a local transaction it forwards a confirmation message to the corresponding backend. A remote writeset is forwarded to the remote backend. Since only one writeset may be processed at a time, the replication manager does not accept any writeset from the GCS until the local or remote backend have confirmed that they have processed the writeset. Additional tasks are necessary at the backends and the replication manager to handle aborts correctly.

Implementation Details We *added* a system table at each replica matching local transaction identifiers with GIDs in order to perform the version checks appropriately and efficiently. Furthermore, we had to *enhance* the abort mechansim in order to allow a remote transaction to abort a local transaction. This required a quite deep understanding of the signaling mechanism whithin PostgreSQL. The only place where we really had to *change* the code was in the lock module. When a transaction joins the waiting queue to acquire a lock, it is normally appended to the end of the queue. We could simply adjust this procedure and put the lock request of the remote transaction at the head of the waiting queue. However,

[2] Not all DDL statements will be replicated. [Cho03] discusses which statements to replicate, which to only executed at the replica they are submitted to, or which to disallow in a replicated system.

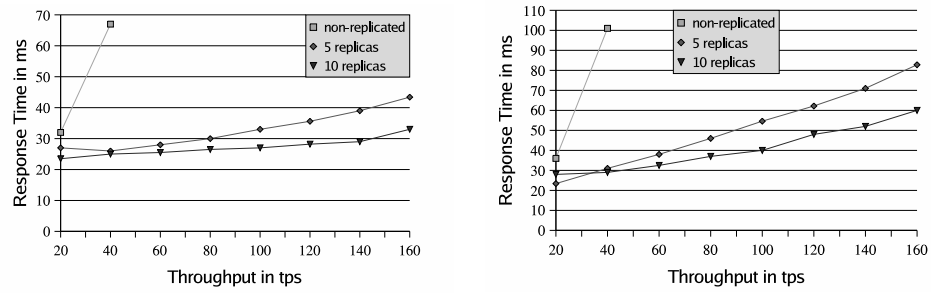


Figure 6: TPC-W: (a) Browsing (read-only) and (b) Ordering (update)

in PostgreSQL, upon releasing a lock, all waiting processes are woken up. Although they are woken up in the order in which they are waiting, this does not guarantee that the lock is actually granted to the first one in the queue due to possible race conditions of UNIX process scheduling. Hence, we had to change the lock release procedure slightly. If a local transaction T_i holds a lock and a remote transaction T_j requests the lock, we put the lock request of T_j at the head of the waiting queue, and send an abort signal to T_i. When T_i aborts and releases the lock, and it sees that the first waiting transaction T_j is a remote transaction (by checking whether it has already a GID), it only wakes up T_j. The rest of the waiting queue is passed to T_j. When the remote transaction receives the lock, it wakes up the rest of the processes in the waiting queue (in order to continue with the standard PostgreSQL procedure). Note also that remote transactions can only abort when failing a version check. Hence, remote transactions should never invoke the deadlock detection routine. We achieve this by not setting the timer for the deadlock detection.

6.4 Evaluation and Discussion

We evaluated the performance using two different applications. The first test suite uses a TPC-W benchmark variant to simulate a real-world application. The second test suite uses a 100% update workload. All experiments are performed on a cluster of PCs (2.66 GHz Pentium 4 with 512 M RAM) running RedHat Linux. For each experiment, we run at least 20000 transactions to achieve stable results. In here, we only present results on the TPC-W benchmark. Further results can be found in [WK04].

We performed our tests using the OSDL-DBT-1 benchmark [Ope02]. It is a simplified version of the TPC-W benchmark [Tra00] simulating an online bookstore. There are three different workload types by varying the ratio of browsing to buying transactions: primarily shopping, browsing and ordering. In our experiment, we choose the browsing workload, which contains 80% browsing transactions and 20% ordering transactions. We have set up a two-tier testbed where the OSDL-DBT-1 driver is the front-tier which directly connects to the database. There are 8 tables in the schema. The database size is determined by the items and clients in the system. We use a very small configuration with only 1000 items and 40 clients. Larger sizes will only decrease conflict rates and increase disk I/O which will favor the replicated approach. We performed the experiment with a fixed number of 40 client connections. The number of clients on each server and the load on each client is evenly distributed. The throughput in transactions per second (tps) is controlled by the think-time parameter, i.e., the time a client waits between two consecutive requests.

We run the experiment with a centralized, non-replicated server, and then with 5 and 10 replicas. Figure 6 shows the client response time for browsing and ordering transactions when we increase the

overall load to the system. For all graphs, the response time increases with increasing load since more transactions concurrently compete for resources. The response time of the centralized system is much worse than our replicated configuration, and can achieve a much lower maximum throughput. The reason is that the server is overloaded very fast while in the replicated systems read-only transactions are distributed among the replicas. Additionally, the centralized server has problems handling many clients. The 10-replica system has smaller response times than the 5-replica system for a given throughput because read-only transactions are distributed over even more replicas. The only exception are update transactions at 20 tps where the 5-replica system is better than 10 replicas. The reason might be that with 10 replicas, more update transactions are remote, and hence, it is more likely that a local update transaction has to wait for a remote transaction whose writeset is received earlier. At higher throughputs this disadvantage does not show because the 10-replica system is much less loaded. In these experiments, abort rates were always well below 1%, which shows that our system can handle real world conflict rates even for very small database sizes.

However, scalability is not unlimited. Updates have to be performed at all replicas. If the update load increases, each replica has less resources to execute queries. Hence, the performance gain from 5 to 10 replicas is not as big as from the non-replicated system to 5 replicas. More about this phenomena can be found in [JPPMAK03].

6.5 Conclusion and Future Work

In summary, this experiment proves that the performance of our system is excellent for a real world situation where most of the transactions are read-only. Our replication solution performs better than a centralized approach by distributing the load and clients throughout the replicas in the system. Hence, our approach implementing eager update everywhere replication within the database system is feasible and performs well for read-world applications.

Our current efforts lie in the integration of an online recovery mechanism into Postgres-R(SI). Our approach will be based on approaches in [KBB01] and use the implementation provided in [Cho03] as a basis.

7 Middle-R: Database Replication at the Middleware Level

Our middleware based replication tool Middle-R also uses GCS as underlying communication mechansims. Before the start of ADAPT, a prototype version of Middle-R existed to enable its evaluation [JPPMKA02]. It provided access to a single evaluation database and did not provide access to arbitrary databases nor had any recovery facility implemented. Since the project start, Middle-R has evolved to a product in which arbitrary databases can be used. It has been additionally extended with dynamic adaptability features – the goal of Adapt – such as online recovery, dynamic load balancing and adaptive load control. It has also been enhanced with replication-aware JDBC connectivity. We will outline these advanced features in the following sections. For more detailed information, we will refer to the according publications.

7.1 Overview

We first want to give an overview of the general architecture of Middle-R. [JPPMKA02] describes the system in more detail. Middle-R is a cluster based database replication tool. The system consists of N nodes (machines), each node hosts a database system and a Middle-R server. Each database system

stores a full copy of the database (replica). The database application programs are written in the usual way, using one of the standard database interfaces to interact with the database. Given a transaction in form of an application program, the application programmer has to identify which data objects are going to be accessed by the transaction. Object granularity can be a table, or any application specific granularity level. The application programs are then deployed within Middle-R, and can be called via the Middle-R client interface (the newly developed JDBC interface is discussed in Section 7.4).

Middle-R is responsible for the execution of the application programs on the database replicas, and performs concurrency and replica control. It uses the group communication system (GCS) Ensemble [Hay98] to disseminate information among the replicas. We use the total order multicast to determine the execution order for conflicting transactions that want to access the same objects. In order to achieve this, Middle-R performs its own lock-based concurrency control.

The system applies asymmetric transaction processing. Each update transaction is only executed at one replica. The other replicas do not re-execute the transaction (neither read nor write operations) but simply change the affected records which is usually faster than reexecuting the statement. This spare capacity can be used to process additional transactions. Asymmetric processing can outperform symmetric processing [JPPMAK03], and might be the only feasible approach for database systems with triggers or non-deterministic behavior. Hence, most commercial systems use asymmetric replication approaches [Gol94]. In order to use asymmetric processing at the middleware layer, the underlying database system has to provide a function to get the changes performed by a transaction (the *write set*), and a second that takes the write set as input and applies it without re-executing the entire SQL statements. We implemented this functionality for PostgreSQL and it is currently being implemented for MySQL and Microsoft SQL Server. Oracle uses such mechanism for its own replication protocol [Ora01].

In order to share the load among all the replicas, we follow a primary copy approach. Each set of data objects that can be accessed within a single transaction is assigned a primary replica that will be in charge of executing programs that access this specific set. For instance, replica $N1$ might be primary of object set $\{O1\}$ and replica $N2$ of object sets $\{O2\}, \{O1, O2\}$. We allow transactions to access arbitrary object sets, and overlapping sets might be assigned to different replicas. This requires a global concurrency control. In contrast, disallowing overlapping object sets to be assigned to different replicas would mean that we partition the data among the replicas, each replica being responsible for transactions accessing object sets within its partition. In this case, each replica could use its own local concurrency control strategy. However, we would disallow transactions to access arbitrary object sets (spanning two or more partitions) limiting transaction types.

The conflict-aware scheduling algorithm and its correctness can be found at [PJKA00]. We will only outline it here. Let us first have a look at *update transactions* that perform at least one update operation. The client can submit an *update request*, i.e., a request to execute an update transaction, to any Middle-R server. This server multicasts it to all middleware servers using uniform reliable, total order multicast. Upon receiving a request to execute transaction T delivered in total order, all servers append the lock requests for objects accessed by T into the corresponding queues of the lock table (a form of conservative 2PL locking). The primary executes T when T's locks are the first in all queues. It starts a database transaction, executes T's code, and retrieves the write set from the database. Then it commits T locally and reliably multicasts (without uniformity or ordering requirement) the write set to the other Middle-R servers which apply it at their databases. The Middle-R server which originally received the client request returns the commit confirmation once it receives the write set (or after local commit if it was the primary of the transaction). Since determining the total order can take a long time, the system uses optimistic execution to increase performance. A detailed description of this feature can be found at [PJKA00].

For queries (read-only transactions), there exist several alternatives. Firstly, they could always be executed locally at the server they are submitted avoiding communication. However, this disallows any form of load balancing, and if all requests are submitted to one server, this server will quickly become a bottleneck. Since communication in a local area network is usually not the bottleneck, an alternative is to also execute queries at the primary. Apart of load balancing issues this might lead to a better use of the main memory of the database system since each replica is primary only of a subset of the data. Hence, we can expect higher cache hit ratios at each replica [CAZ02] than if each replica executes any type of query. In the primary approach, a query request is forwarded to all replicas, the primary then executes the query, returns the result to the submitting Middle-R server and notifies the end of the query to all replicas. Independently of whether a local or primary approach is used, the executing Middle-R server might not need to acquire locks for queries but immediately submit them for execution if the database uses snapshots for queries (as is done by PostgreSQL or Oracle).

The approach provides 1-copy-serializability because all replica decide on the same execution order of conflicting transactions due to the total order multicast. Even if the primary fails after committing but before sending the changes, a new primary will take over and re-execute the transaction in the same order due to the total order multicast.

7.2 Online Recovery

Replication aims to attain high availability by tolerating failures of some replicas. In order to maintain certain level of availability new replicas (crashed or new) should join the system. But what happens during recovery? When a new replica joins the system its state must be updated according from the state of running replicas. This state transfer or recovery is usually performed offline. However, if transaction processing is stopped availability is lost. On the other hand, if transaction processing is not stopped (i.e. online recovery), the recovery takes place whilst transactions are being processed in the system. In this case, data consistency becomes an issue.

Middle-R has been enriched with online recovery ([KBB01]) to guarantee a high level of availability [JPPMA02]. In this approach, replicas can play four roles: master of an object sets (the one that fully executes update transactions modifying this object set, recovering replica (the replica that has joined the system and needs to recover an up-to-date state of the database), recoverer replica of an object set (a working replica that will transfer an up-to-date snapshot of the objects), and replicas not participating in the recovery.

In Middle-R, objects have been assimilated to tables and therefore, recovery is based on the notion of tables. One of the features of the online recovery of Middle-R is that *each table can be recovered independently*.

The online recovery works as follows. When a new replica joins the system, a short state transfer takes place. In this state transfer the recovering replica indicates the last transaction it processed on each table. Then, a set of replicas is selected as recoverers. Each of these replicas will recover a set of tables. For the sake of simplicity, assume there is a single recoverer and that tables are recovered one by one. The recoverer will send the relevant log records (those corresponding to the table being recovered that were missed by the new replica) to the recovering replica. The recovering replica applies the corresponding updates (the received log records).

Since the recovery is performed online, transactions can update a table whilst it is being recovered. The recoverer will be able to process the updates of those transactions and send the corresponding log entries to the recovering replica later. The recovering replica will discard the updates from those transactions, since queueing them might result in running out of memory during long recoveries. The recovering

replica will instead receive these updates from the recoverer.

When the recoverer reaches the end of its log for the table being recovered, it will send an end recovery message to complete the recovery of the table. Upon receiving this message, the master of that table will send a message indicating the last transaction that will be considered part of the recovery. This message will be used by the recoverer to determine which it is the last log entry (updates) that it will forward to the recovering replica. The recovering replica will use this message to discriminate when it will stop discarding update messages on this table and start to queue them for their processing.

Once all tables have been recovered the recovering replica will become a regular working replica. It has to be noted that a recovering replica can process transactions that access tables already recovered. So, it can help to process the current load as soon as one table is recovered.

Another feature of the online recovery is that it can deal with *simultaneous and cascading recoveries* in an efficient way. Multiple replicas starting recovery at the same time will be recovered at the same time exploiting the underlying broadcast network therefore, minimizing the consumed resources (CPU and disk bandwidth from recoverers and network bandwidth). Cascading recoveries are also managed efficiently. If a new replica joins the system when there is an ongoing recovery, the set of tables yet to be recovered in the ongoing recovery will be recovered simultaneously by the new recovering replica and the recovering replica that was already engaged in the recovery process. This idea is extended to an arbitrary number of cascading recoveries.

7.3 Load Balancing

7.3.1 Overview

A replicated database system can only then be used for scalability, if the load can be equally distributed among all replicas, and none of the replicas becomes overloaded. In order to evaluate whether a database system performs well, one of the target performance metrics is the throughput (rate of executed transactions per time unit). This metrics depends on the workload (mix of transaction types), load (rate of submitted transactions), cache hit ratio, load distribution among replicas, etc. Another important aspect of the database system to work well under a given workload is the *multiprogramming level* (MPL), i.e., the number of transactions that are allowed to run concurrently within the database system. Initially, when resources are freely available, then a high MPL boosts throughput. Also, if some transactions are I/O bound, concurrent transactions might keep the CPU busy while I/O takes place. However, when resources are highly utilized, or a single resource becomes the bottleneck (e.g., the log), increasing the MPL will only increase context switches, and hence, put even more restraint on the resources. Performance is then lower than in a system with less concurrent transactions. Also, if conflict rates are high, additional transactions will only lead to higher abort rates, and hence, wasted execution time.

In dynamic environments, workload and/or the load can change over time. As a result, the system configuration has to be adapted dynamically, i.e., the MPL and the distribution of transactions across replicas must be adjusted. An additional dynamic behavior is the crash of individual components. If a node fails, the other nodes must take over the load of the failed node.

The contribution of our work lies in providing a hierarchical approach with two levels of adaptation for Middle-R. At the local level, the focus is on maximizing the performance of each individual replica by adjusting the MPL to changes in the load and workload. Middle-R maintains a pool of connections to the database which is shared among transactions. The size of the pool determines how many transactions are concurrently submitted to the database, and hence determines the MPL. At the global level, the system tries to maximize the performance of the system as a whole by deciding how to share the load

among the different replicas. Assigning object sets to primary nodes determines which transactions are executed at which nodes. Given a static workload, an optimal distribution of object sets can easily be found. However, when the workload characteristics change over time, and a node becomes overloaded, a reassignment is necessary. The challenge of performing these kinds of adaptation at the middleware level is the reduced information that is available about the changes in behavior and internals of the database making it hard to detect bottlenecks. At the local level, we use a feedback driven approach that adjusts the MPL according to the observed throughput in the recent past. At the global level, we take the number of transactions waiting at each node for execution, to evaluate the load in the system. In order to keep the report reasonably short, we only present the general ideas behind the algorithms, and their performance. A detailed description of the algorithms can be found in [MFJPK04].

7.3.2 Local Level Adaptation

At the local level, each middleware server is configured to maximize the performance of its local database replica. Measurements have shown that Middle-R servers are light-weight while the database servers are the first to be the bottleneck [JPPMKA02]. Hence, controlling the MPL is an important step in dynamic performance optimization, and is done by limiting the connection pool to the database replica.

Our solution to control the MPL is based on the feedback control approach proposed in [HW91]. Since it does not require database internal information like conflict rate, memory and other resource consumption, etc., it is suitable for a middleware-based system. In a feedback system, one uses the output of the system as an indicator whether the input of the system should be changed. [HW91] proposes to take the transaction throughput as output parameter. In a system without control on the number of concurrent transactions, the throughput of the database system usually rises with increasing the number of transactions until the system saturates at a throughput peak. If the number of concurrent transactions increases further, the database enters the thrashing region in which the throughput falls very fast until it stabilizes at some low residual value. Figure 7, adjusted from [HW91], illustrates this behavior. The x-axis depicts the MPL, the y-axis depicts the throughput achieved by the system with it, and the z-axis shows how the curve changes over time assuming the workload changes over time. Our measurements on a real database have shown that if the workload contains many complex read operations, throughput is generally low but many many transactions should run concurrently, if the workload contains many simple write operations, a high throughput can be achieved but only at small MPLs (basically serializing writes). When now the workload moves from a read intensive workload to a write intensive workload, so does the dependency between MPL and throughput.

Hence, we have two goals. Firstly, at any given time with a given workload, we have to determine the optimal MPL, i.e., to deny newly submitted transactions to execute whenever this would lead to a load that cannot be handled anymore by the database. That is, we should set a MPL such that the database system is never in the thrashing region. The *optimal MPL* is now defined as the MPL allowing for the maximum achievable throughput. The second goal is to provide dynamic adaptability, that is, to adjust the MPL when the workload changes such that it is never higher than the optimal MPL. [HW91] approximates the relationship between concurrent transactions and throughput at each time point with a parabola. In order to estimate the coefficients, the system periodically measures the number of concurrent transactions and the throughput. In order to capture the time dependency of the parabola, more recent measurements are given a higher weight than older measurements. After each measurement period, the optimal MPL is set to the number of concurrent transactions achieving the highest throughput. The approach also addresses some stability problems. We have implemented this approach within Middle-R using a incremental adjustment of the MPL. See [MFJPK04] for the algorithm and details.

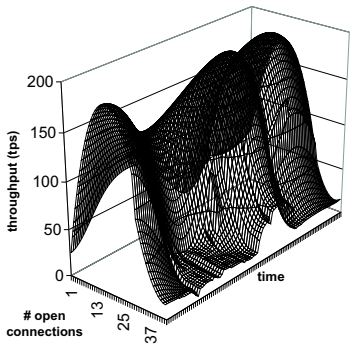


Figure 7: Throughput as a function of MPL over time

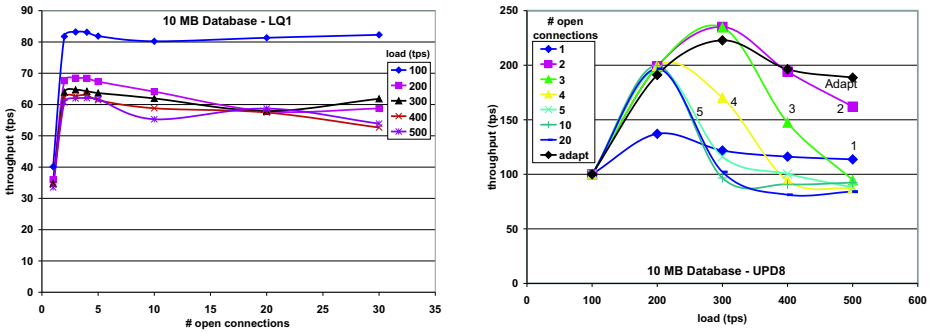
7.3.3 Global Level Adaptation

A replicated database might potentially improve its throughput as more replicas are added to the system [JPPMAK03]. However, this potential throughput is only reached under an even load in which all replicas receive the same amount of work (assuming a homogeneous setting), which in practice might never happen. If the load is concentrated at one replica, the throughput will be the throughput of a single replica or even worse due to the overhead of the replication protocol to ensure consistency among replicas. Load balancing is aimed to correct situations in which some replicas are overloaded, while others have still execution capacity. This is done by redistributing the load as evenly as possible among the replicas. Therefore, any load balancing algorithm requires a means to estimate the current load at each replica.

Each Middle-R server knows the total number of concurrent active transactions in the system, since requests are sent to all servers. All servers acquire locks for the objects accessed by transactions that are kept until the transaction terminates locally. Hence, looking at its lock table, each server has a good estimate of the total number of active transactions. For some of them the server is the primary copy. We call these the local transactions of the server. Local transactions might either be executing or waiting for locks or waiting for a free connection. For others, the server is not the primary. We call them remote transactions. If it is an update transaction the server is waiting for the write set (it is still active at the primary), or currently applying the write set or waiting for a free connection (the transaction is committed at the primary). If it is a query, the write set message is empty and used as an indication of the end of the query. Looking at this lock table, each server can estimate the number of local active transactions of any other server S. This calculation can be done without any additional communication overhead.

The number of local active transactions at a server is a good estimate of the load at this server. If it is known that different transaction types have different execution times, then this load metrics can be made more precise by weighting the number of local active transactions with the observed average execution time [ACZ03b]. The degree of balance is captured by the variance among the number of local active transactions at each node. A variance of zero represents a evenly distributed load. On the other extreme, the maximum variance is achieved when the entire load is concentrated on a single replica.

The load balancing algorithm is in charge of finding a primary assignment of object sets to servers that minimizes the variance. We have developed two algorithms. Our first algorithm uses a branch-and-bound mechanism to assign object sets to primaries. At each step the algorithm selects an object set that is accessed by many transactions and assigns it to all servers yielding a set of partial solutions. The



a b

Figure 8: Adaptation under constant load

algorithm then traverses all partial solutions and prunes those that will not yield in a better solution than the current one. The pruning is based on an estimation function which provides a lower bound of the actual possible variance. Although the algorithm provides an optimal assignment, its use is limited to small number of object sets and replicas since its computation time grows exponentially. Our alternative is an inexpensive greedy algorithm which assigns at each step the unassigned object set with the highest load to the replica with the smallest current load. Our comparison experiments show that in 80% of the runs the greedy allocation is optimal and in the remaining runs it was very close to be optimal. The execution time is highly reduced compared to the branch-and-bound algorithm.

A Middle-R server S responsible for load balancing periodically calculates the load variance and runs the greedy algorithm if it exceed a given threshold. The new assignment will only be applied if it leads to a significant improvement. S multicasts a load balancing message m_l to all servers to inform about the new assignment using the total order multicast. Transactions received before m_l are still executed at the old primaries, transactions received after m_l at the new primary.

7.3.4 Experimental Results

A detailed performance analysis is presented in [MFJPK04]. In here, we only summarize the results. Experiments have been performed on 10 machines, each with two processors AMD, 512 MB, and 60 GB disk, interconnected with a 100-MBit switch.

Local Adapatability We first performed a series of preliminary results. In each test run, we used a different workload (update intensive vs. query intensive, small database vs. large database), and then checked what it the maximum achievable throughput when we vary MPL and the load submitted to the system. We determined that depending on the workload, size of the database, and the load submitted to the system, a different MPL provides the maximum achievable throughput. The goal of our local adaptation mechanism is that this optimal MPL is automatically selected without an external administrator indicating the type of workload, the size of the database or the load. As an example of its performance, Fig. 8 presents for update intensive (a) and query intensive (b) workloads at a small database size of 10MB, what is the maximum achievable throughput, when MPL is set to fixed values, and we vary the load submitted to the system. Additionally, the figures contain a curve of what is the throughput achieved by our local adaptation algorithm. We can see, that for all loads (x-axis) and workload types ((a) and (b)),

it achieves a nearly optimal throughput by choosing the optimal MPL automatically. In an experiment where we chose an MPL far of being optimal at system startup, we evaluated the time the system needed to calculate the optimal MPL. It took around 5 seconds to adjust. This means that our algorithm works well if workload changes occur at most in minute intervals which we believe is quite realistic.

Global Adaptability To test global adaptability, we first tested the maximum achievable throughput in a system where the load is evenly distributed among all replicas, and the throughput achievable when all load goes to one replica (uneven distributed) and there is no adaptation. Then we tested a system where originally all load goes to one replica but the global adaptability algorithm is in place. The result shows that after an initialization time, the system with global adaptability moves from the low maximum throughput of the unbalanced system to the high maximum throughput of a perfectly balanced system. The transition needed around 4 seconds. We performed these tests for various loads, workloads, database sizes, and number of replicas, all with similar results.

Tests combining both local adaptation and global adaptation showed that both algorithms worked smoothly with each other leading to an optimal MPL and load distribution for various kinds of workloads, and number of database replicas.

7.4 JDBC Connectivity

In order to provide connectivity with Java applications, such as the application server, a JDBC driver has been developed for Middle-R. JDBC drivers were designed to contact a single database and not a replicated database. In order to support the access to Middle-R, a set of mechanisms have been built into the Middle-R JDBC driver:

- A *replica discovery mechanism* has been incorporated into the JDBC driver. The database IP address is an IP multicast address that is used by the driver to multicast a discovery message. Replicas (Middle-R instances) answer to this message with their IP addresses. From the set of collected addresses the JDBC driver tries to establish a TPC connection with one of them (traversing the set if necessary). Once the connection is established, all the client requests are submitted to this replica through the TPC connection. The replica multicasts the request to all replicas.

- A *fail-over mechanism*. In case the replica with which the driver is connected fails (or the connection fails), the driver tries to connect with one of the other replicas using the IP addresses collected through the discovery process. If none of the addresses work, it retries again the discovery process. Since the last request made to the failed replica might be or might not be multicast to the rest of the replicas, this last request is in doubt (i.e., it could have been submitted and the reply was not received). For this reason, a duplicate removal mechanism is included as well. Connections are uniquely identified as well as requests submitted through them. If a request is submitted twice or more due to the fail-over, it is removed by Middle-R. This can be done thanks to the unique identifiers of the connection and the requests. Replicas only need to recall the last request submitted and its reply (if it was performed). Upon the reception of a request that was previously submitted, there are two possibilities: 1) There is no reply recorded. In that case the request is processed (since it has not been processed by the working replicas) and the reply returned; 2) There is a reply recorded. Then, the request has already been processed. The reply is returned and the request discarded. Note that this mechanism is very similar to the mechanisms presented in Sections 3 and 4 for exactly-once execution.

- A *object set discriminator*. This discriminator parses the SQL statement and determines whether it is a query or an update transaction. Additionally, it determines which tables are being accessed and in which mode. It also finds out whether the statement is an update of single tuples identified by their primary key. All this information is packed with the request to help Middle-R to discriminate the object set of the transaction.

The driver is currently being enriched with an additional mechanism to enable its use by replicated clients such as the replicated application server:

- A *connection serialization interface*. This interface enables to serialize the state of a connection that includes the unique connection identifier and the identifier of the last request submitted. That way the connection can be checkpointed to another replica and resume the connection. In case of duplication of request, the duplication removal mechanism guarantees exactly once semantics. This mechanism will support as well active replication. The connection is established at one of the active client replicas and then propagated to the remainder client replicas. From then on, the duplicate removal enforces the exactly once semantics.

References

[ABKW98] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *ACM SIGMOD Conf.*, 1998.

[ACZ03a] C. Amza, A. L. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *USENIX Symp. on Internet Technologies and Systems*, 2003.

[ACZ03b] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, 2003.

[Apa03] Apache Web Services Project. Axis SOAP library, version 1.1, June 2003. http://ws.apache.org/axis.

[BBM+04] O. Babaoglu, A. Bartoli, V. Maverick, S. Patarin, and H. Wu. A Framework for Prototyping J2EE Replication Algorithms. In *Proc. of the Int. Symposium on Distributed Objects and Applications (DOA)*, 2004. accepted.

[BEA02] BEA Systems Inc. *BEA WebLogic Server Programming WebLogic Enterprise JavaBeans*, release 7.0 edition, September 2002.

[BGRS00] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the coordination overhead of synchronous replica maintenance in a cluster of databases. In *Euro-Par*, 2000.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BKR+99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *ACM SIGMOD Conf.*, 1999.

[BLW02] R. Barga, D. Lomet, and G. Weikum. Recovery guarantees for general multi-tier aplications. In *Proc. of the Int. Conf. on Data Engineering*, San Jose, California,USA, 2002.

[Bor04] Borland. Interbase Documentation, 2004.

[BPA04] A. Bartoli, M. Prica, and E. Antoniutti. A replication framework for program-to-program interaction across unreliable networks and its implementation in a servlet container. Technical report, DEEI University of Trieste, 2004. Accepted for publication in Concurrency and Computation: Practice and Experience (subject to minor revisions).

[CAZ02] A. L. Cox C. Amza and W. Zwaenepoel. Scaling and Availability for Dymaic Content Web Sites. Technical Report TR-02-395, Rice University, 2002.

[Chi03] R. Chinnici. *Java™ API for XML-based RPC: JAX-RPC 1.1*, 2003. http://java.sun.com/xml/jaxrpc/index.jsp.

[Cho03] M. Chouk. Master–slave replication, failover and distributed recovery in PostgreSQL database. Master's thesis, McGill University, June 2003.

[CMZ03] E. Cecchet, J. Marguerite, and W. Zwaenepoel. RAIDb: Redundant array of inexpensive databases. Technical Report 4921, INRIA, 2003.

[DeM03] L. G. DeMichiel. *Enterprise JavaBeans™ Specification, Version 2.1*, November 2003. http://java.sun.com/products/ejb/docs.html.

[FG00a] S. Frølund and R. Guerraoui. A pragmatic implementation of e-transactions. In *Proc. of Symp. on Reliable Distributed Systems (SRDS)*, Nürnberg, Germany, 2000.

[FG00b] S. Frølund and R. Guerraoui. X-ability: a theory of replication. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, Portland, Oregon, USA, 2000.

[FG01] S. Frølund and R. Guerraoui. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):50–97, 2001.

[GBHC00] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD Conf.*, 1996.

[Gol94] R. Goldring. A discussion of relational database replication technology. *InfoDB*, 8(1), 1994.

[HAA99] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group communication. In *Int. Symp. on Fault-tolerant Computing*, 1999.

[Hay98] M. Hayden. The Ensemble System. Technical Report TR-98-1662, Department of Computer Science. Cornell University, January 1998.

[HW91] H. Heiss and R. Wagner. Adaptive Load Control in Transaction Processing Systems. In *Proc. of 17th VLDB*, 1991.

[JBo] JBoss Group. JBoss. http://www.jboss.org/.

[JBo03] JBoss Group. JBoss 3.2.3, November 2003. http://www.jboss.org/.

[JPPMA02] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. In *IEEE Symp. on Reliable Distributed Systems (SRDS)*, 2002.

[JPPMAK03] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication. *ACM Transactions on Database Systems*, 28(3), 2003.

[JPPMKA02] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proc. of Int. Conf. on Distributed Computing Systems*, 2002.

[KA00a] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Int. Conf. on Very Large Databases*, 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3), 2000.

[KBB01] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2001.

[MFJPK04] J. M. Milan-Franco, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Adaptive distributed middleware for data replication. In *Middleware*, 2004. accepted.

[MMSN+99] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. The Eternal System: An Architecture for Enterprise Applications. In *Int. Enterprise Distributed Object Computing Conference*, September 1999.

[NMMS01] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks(DSN)*. IEEE Computer Society Press, 2001.

[OMG00] OMG. *Fault Tolerant CORBA*. Object Management Group, 2000.

[Ope02] Open Source Development Lab. Descriptions and Documentation of OSDL-DBT-1, 2002.

[Ora01] Oracle. Oracle 9i Replication, June 2001.

[PBB+02] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Technical Report UCB CSD-02-1175, UC Berkeley, Computer Science, 2002.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In D. J. Pritchard and J. Reeve, editors, *Euro-Par*, 1998.

[PJKA00] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*, October 2000.

[PMS99] E. Pacitti, P. Mine, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Int. Conf. on Very Large Data Bases*, 1999.

[Pra02] Pramati Technologies Private Limited. *Pramati Server 3.0 Administration Guide*, 2002. http://www.pramati.com.

[Sha03] Bill Shannon. *Java™ 2 Platform Enterprise Edition Specification, v1.4*. Sun Microsystems, Inc., November 2003. http://java.sun.com/j2ee/1.4/docs/.

[Sun03] Sun Microsystems, Inc. ECperf™ specification: 1.1 final release, November 2003. http://java.sun.com/j2ee/ecperf/.

[Tra00] Transaction Processing Performance Council. TPC Benchmark W, 2000.

[UAB+00] K. Ueno, T. Alcott, J. Blight, J. Dekelver, D. Julin, C. Pfannkuch, and T. Shieh. *WebSphere Scalability: WLM and Clustering*. IBM RedBooks SG246153, 2000.

[VvRB98] W. Vogels, R. van Renesse, and K. Birman. Six misconceptions about reliable distributed computing. In *Proceedings of the 8th ACM SIGOPS European Workshop*, 1998.

[WK04] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. Technical report, School of Computer Science, McGill University, 2004. submitted for publication.

[WKM04] H. Wu, B. Kemme, and V. Maverick. Eager Replication for Stateful J2EE Servers. In *Proc. of the Int. Symposium on Distributed Objects and Applications (DOA)*, 2004. accepted.

[ZMMS02] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[ZMMS03] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Design and implementation of a pluggable fault tolerant CORBA infrastructure. In *Proc. of the Int. Parallel and Distributed Processing Symp.*, Fort Lauderdale, California, 2003.