Customizable Service State Durability for Service Oriented Architectures

Xianan Zhang
Keith Marzullo
University of California, San Diego
La Jolla, CA 92037, USA
{xzhang,marzullo}@cs.ucsd.edu

Matti A. Hiltunen
Richard D. Schlichting
AT&T Labs-Research
Florham Park, NJ 07932, USA
{hiltunen, rick}@research.att.com

Abstract

Service Oriented Architectures (SOAs) are gaining increased use and interest with the growing importance of web and grid services, and the dependability of services in such architectures is critical. While the techniques for implementing highly dependable services are well known, each technique has its tradeoffs relative to runtime performance overhead, resource cost, and level of assurance provided. The ability to choose the dependability based on the service requirements and characteristics allows the optimization of these tradeoffs. The goal of our work is to provide mechanisms that allow different durability mechanisms (e.g., in-memory replication or database storage) to be implemented as separate modules and then applied as needed to different variables and data structures constituting the service state. We describe an overall architecture that allows such durability customization for each web service and present preliminary performance results.
Keywords: availability, performance, web services, SOA, durability
Category: Regular paper

1 Introduction

Service Oriented Architectures (SOAs) are increasingly important as an approach for restructuring enterprise software architectures and for automating B2B interactions. In an SOA, the enterprise core functions are implemented as services that interact using well established web services standards [8] such as XML, SOAP, HTTP, and WSDL. Since SOAs are collections of interacting services implemented on multiple interconnected machines, they inherit all the normal challenges associated with building distributed systems. While there are many such challenges, certainly one of the most critical is dealing with the potential impact of machine failures and server process crashes. A failure resulting in the unavailability of one service S_i in such an SOA can result in the unavailability of a large number of other services that rely on S_i.

A key requirement for implementing highly available services is the ability to maintain service state across machine failures and server process crashes. This problem has been extensively studied and many techniques exist for protecting service state, including storing the state in a database or maintaining it in replicated processes. Each such technique can be characterized based on its level of protection and its cost in terms of hardware resources required and performance overhead. While many such techniques exist,

service developers typically have a very small set of options (e.g., a database or nothing) and the chosen option must be coded in the service implementation (e.g., by denoting objects as session or entity beans in J2EE [7]). Specifically, there is no integrated and transparent way to use different techniques to protect service state. However, different types of service state differ in the degree of protection required; some types such as billing information needs to survive very severe failures, while others might be reconstructed relatively easily should a failure occur. Furthermore, the business or cost requirements associated with the service may change over time, resulting in large code rewrites.

This paper introduces an architecture that allows service developers to use different techniques with different tradeoffs to protect service state flexibly and transparently. The service state is stored in one or more *state objects*. We propose to treat *state durability*—that is, the likelihood that the state can survive failures—as an explicit design parameter that is associated with each state objects used by a service. Based on this durability attribute, different techniques with different tradeoffs can then be used to ensure the durability of different state objects. For example, the value of one state object can be stored in a database, while another is replicated in-memory on two or more computers. Note that one can view the different techniques as implementations of the stable storage abstraction [16], but with an explicit recognition of and control over the tradeoff between the fidelity of the abstraction and the cost of implementing it.

We have built a prototype based on Globus 4.0 [1] using Java. In this prototype, service designers can choose for each state object between distinct points on the durability-performance continuum, including implementing durability using primary-backup replication or implementing it using a database. Different choices can be made for different service state objects and the choice has no impact on the way in which the state object is used by services. To illustrate the way in which the architecture can be used, we describe a prototype implementation of a highly available Matchmaker service [22].

2 Concepts and Background

In this section we discuss the role of state in SOAs and define the concept of state durability.

2.1 Service Oriented Architectures

Service Oriented Architectures (SOAs) structure software functionality in a distributed system as collections of interacting services as illustrated in figure 1. The services include both *infrastructure services*, such as directory services (UDDI), monitoring, and resource allocation services, as well as *application services* that implement some application specific functions. The infrastructure services, denoted as shaded ovals in the figure, provide the foundation for building, running, and accessing the application services. The web services architecture defines a service-oriented distributed computing model in which services interact by

exchanging XML documents.

A service invocation, from a client's desktop application for example, often results in a sequence or chain of interleaved service invocations between different services potentially in different administrative domains. This leads to a situation where, paraphrasing Leslie Lamport, *a service-oriented architecture is one in which the failure of a service you didn't even know existed can render your application unusable*.

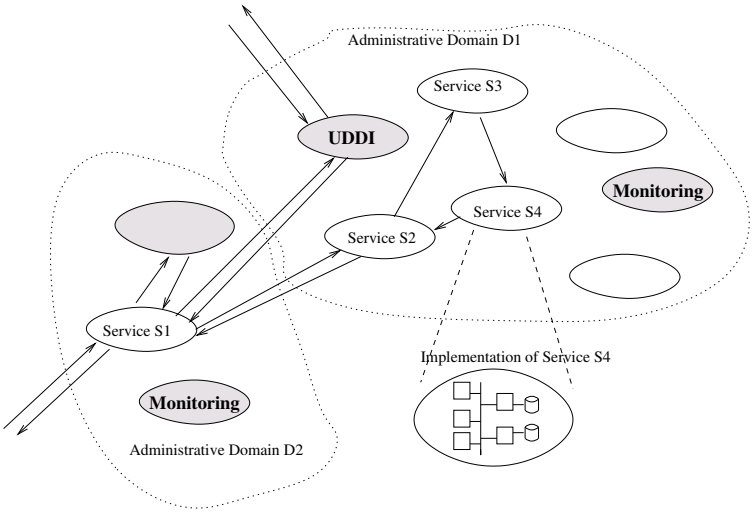

Figure 1: Example Service Oriented Architecture

2.2 State in Service Oriented Architectures

There are many aspects to providing reliable and highly available services. Specifically, the service architecture must be able to detect failures of services, restart failed services, and reconnect clients (which may be end-applications or other services) with the recovered or relocated service instances. A key factor in providing such services is maintaining the service state through failures. If the service state can survive the failure of the server processes that provide the service, the service can be restarted by simply starting a new server process with the preserved service state.

The type of service state depends on the specifics of the service as well as the middleware platform used to implement the services and service interactions. Services may be completely stateless, that is, all the state required to process a service request is included in a request (e.g., an encryption service that receives the data and key in the request). Otherwise, the service may have

- **Internal state.** State that the service reads or updates when processing a request, e.g., an inventory database.

- **Session state.** State associated with the interaction between the service and (typically) one client,

e.g., a shopping cart.

Distributed object platforms such as CORBA and DCOM support both internal and session state maintained in the object instance accessed by the client. Traditional web services do not explicitly support any state and are therefore often considered stateless for this reason. However, many web service applications require state that is maintained across client requests and most web service platforms support storing important state in a database (either explicitly like in ASP.NET or implicitly like in J2EE entity beans).

The first generation of grid services as defined by the OGSI (Open Grid Services Infrastructure) [25] standard supported both session and internal state. Conceptually, these grid services were very similar to distributed object systems with the concept of creating grid service instances that maintain state. The new WSRF (Web Service Resource Framework) [12] replaces OGSI and allows stateful grid services to be built directly on the web services foundation. WSRF defines a framework for managing state in distributed systems using web services. Such state is modeled as a WS-Resource that provides a web service interface for manipulating a stateful resource (e.g., a database table, file, J2EE entity bean).

While WSRF provides a clear separation between the (stateless) web service and the state it operates on, such a fundamental separation is often present even in web services implemented without WSRF. For example, web services can be implemented using J2EE (Java 2 Platform, Enterprise Edition) [7] by exposing business services written using Enterprise JavaBeans (EJBs) as web services. When a service is implemented using EJBs, the designer chooses which service state objects need to be persistent by implementing them as entity beans and which state objects do not need to be persistent by implementing them as session beans. Thus, the service state is separated from the service functions.

2.3 State Durability

We use the term *durability* to describe the attribute of service state that describes its resilience against hardware and software failures, that is, durability is the probability that the state will persist for a specified length of time despite failures. While other dependability concepts such as reliability and availability [5] are attributes of measuring the overall service, durability has its focus on service state, which is often the key factor to provide the dependability of the overall service. Analogously to reliability or availability, the state durability depends on the techniques used and the numbers, types, and frequency of failures that occur in the underlying resources during the lifetime of the service state. Similar to these two other metrics, it is impossible to provide 100% durability but it can be be increased to be arbitrarily close to 1.0 by using the appropriate techniques.

There are a number of techniques that can be used to increase the durability of a state object. For example, the state object may be replicated on multiple processors, stored in a file, or stored in a database. Note that not all file systems and databases provide the same degree of durability. For example, a file system

that uses redundant disks (e.g., RAID) provides higher durability than a normal file system that relies on a single hardware disk, and a replicated database typically provides a higher durability than a non-replicated database. The cost of providing durability depends on the technique and implementation platform chosen. This cost includes both of the runtime performance overhead and the cost of the hardware and software. Furthermore, the different durability techniques may have different recovery time (MTTR - mean time to repair) and thus, affect the overall availability of the service.

The state in different services, and different types of state within a service, have different durability requirements. For example, an e-commerce service maintains inventory information, information about the regular customers (e.g., address, credit card number, preferences), and state about on-going customer interactions ("shopping carts"). The inventory information is most valuable because its loss would prevent the service from operating, while the loss of the information about on-going customer interactions would be a nuisance for the users, but would not stop the service. Note, however, that even such issue may be enough to cause users to switch to a competing service, resulting in loss of revenue.

Note that the durability concept is more general than the typical classification of state as soft or hard state. *Soft state* is often defined as state that can be reconstructed automatically given enough time and/or work (computation), while *hard state* is typically defined as state that cannot be reconstructed automatically if lost due to a failure. Note, however, that whether the data can be automatically reconstructed does not necessarily reflect the true value of a state. For example, a shopping cart is hard state but not extremely high value, while a soft state such as a simulation result may have a high value if its reconstruction takes days of CPU time on hundreds of computers. Therefore, the concept of state durability allows the durability techniques to be tailored based purely on the value of the state. This realization that all the state is not the same can result in performance improvements, cost reduction, as well as an increase in service availability.

3 Customizable Transparent Durability

The goal of our work is to provide web services with customizable availability, durability, performance, and resource cost tradeoffs. Specifically, our approach allows the web service state to be made durable by applying different durability techniques transparently and automatically to different state objects, as independently as possible from the semantics and representation of the state. By doing so, we provide *durability transparency* to the service: the desired mechanisms can be instantiated when the service is built or configured. Furthermore, given such durable state, we show how highly available web services can be constructed using standard web services.

3.1 Assumptions

We assume that a web service is implemented as a Java class and each state object used by the web service is implemented as a separate Java class. Thus, all state access operations by the web service are method calls to the Java objects implementing the state. Each method exported by the web service may read and update one or more of the state objects, one or more times.

As an example, consider an on-line bagel shop web service that allows clients to order bagels by issuing web service requests. Examples of state objects in this example could be the store inventory (e.g., how many bagels of each kind are available) and customer information (e.g., delivery address and credit card information). The web service can be implemented as a stateless Java class that takes the request, checks the inventory and reserves bagels for the customer, issues a credit card transaction to the customer's bank (using a web service request), and if everything succeeds, schedules delivery to the customer's delivery address. Standard transactional techniques can be used to ensure bagels are delivered if and only if the credit card transaction succeeds. Other web service operations are available for registering a new customer, querying price of bagels, etc.

In this paper, we consider crash failures of the machines on which the web service is running, as well as crash failures of the web container process that hosts the web service. Specifically, we assume that all state maintained in the memory will be lost if a failure occurs. Furthermore, we assume a failure can be detected reliably (i.e., the availability of perfect failure detector).

3.2 Durability mechanisms

In this paper, we consider three mechanisms with different performance-durability tradeoffs:

- **Database.** State object is backed up in a database.
- **Replication.** State object is backed up in a backup replica(s) of the state object on other machine(s).
- **Reconstruction.** State object is stored in memory only without any backup, and then reconstructed in case of failure.

We chose these three different mechanisms because their implementations are quite different: accommodating them in a way that is transparent to the web service is a challenge. Also, they provide different performance, durability, availability, and resource cost tradeoffs. Note, however, that a specific mechanism can provide a range of possible tradeoff points depending on factors such as number of replicas used (for replication) or the type of database (and the underlying file system, OS, and hardware) used. For example, an in-memory database implementation uses the database interface, but does not provide any additional increase in durability.

Many other mechanisms with different performance-durability tradeoffs, such as client-side caching, files, and state machine replication, resemble in some ways resources constructed using one of these approaches.

3.3 Challenges

There are many challenges in providing such customizable transparent durability.

- **Internal transparency.** How to add state durability mechanisms to an existing web service without requiring manual modification of the web service code (including the state object code).
- **External transparency.** How to ensure that the different durability mechanisms used within a web service do not change the external behavior of the web service.
- **State update and restoration.** The different durability mechanisms use very different representations of the state and thus, the operations for updating the state are very different. The database solution requires the state to be converted into database tables and state update operations into updates of the tables. A trivial solution is to use Java serialization to serialize the state object and store this serialized form in the database. However, this solution can be very expensive if the state is large. Since the replication approach replicates the whole state object, it may be possible to execute the same operations on the backup (in the same order) as are executed on the original object (primary). However, if the update operations are nondeterministic, this simple solution cannot be used.
- **Atomicity w.r.t. failures.** Since a web service request may update multiple state objects (or a single object more than once), we may want to ensure all-or-nothing semantics in the case of the web service failure in the middle of processing a request. That is, if the web service fails in the middle of the request, the impact on the state objects is as if the request was never executed. Given such semantics, the client can simply reissue to request with no undesirable side effects. While database transactions provide such atomicity for the the database-based solution, similar guarantees must be provided for the replication approach.

4 Solution

This section describes our approach in detail and explains how the challenges above can be addressed.

4.1 Overview

Our solution is based on the following key concepts:

- **Durability proxies** that implement a durability mechanism in a generic, state independent, manner.
- **Durability mapping** that specifies which durability mechanism is to be used for each state object, as well as any needed object-specific instructions.
- **Durability compiler** that takes the web service and state object code, the durability mapping, and the necessary durability proxies and generates a web service and the associated state objects where the desired durability mechanisms are used for each state object.

An example of a web service transformed by the durability compiler is shown in Figure 2. In this example, a web service consists of two state objects, one of which uses the database (via the database proxy) and the other uses replication (via replication proxies). Note that the backup state replica is maintained inside a backup web service, but this backup web service only maintains the state object and does not serve clients' web service requests, unless the primary web service fails.

If the (primary) web service fails, a backup web service takes over the processing of client requests. Specifically, the recovery operation in the database proxy retrieves the object state from the database. Since the replication proxy at the backup already maintains the current object state, it simply changes its role to be the primary replication proxy and a new backup web service is created, if necessary. If no backup web service exists when the web service fails, a new web service instance can be created and the chosen proxies restores the service state before the web service starts serving client requests.

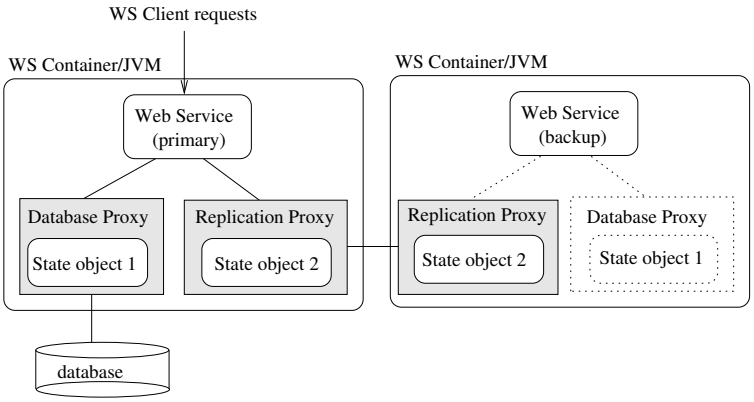

Figure 2: Example Service

The web service client may use some form of web service transactions—defined by the WS-Transactions specifications including WS-Coordination [10], WS-AtomicTransaction [11], and WS-BusinessActivity [9]—to access the web service and the web service may participate in WS-Transactions spanning multiple other web services. When transactions are used, the usual transactional guarantees are provided by our customized web service with regard to atomicity and serializability. When WS-Transactions are not used,

our web service ensures semantic durability and consistency of any state changes. Specifically, this means that if a client executes a web service request that updates the state of the service and the client receives a response indicating success, this change remains in effect even if the original web service instance subsequently fails. Thus, if the client issues a subsequent web service request that is served by a backup web service, the state of this service will be consistent with the client expectation, that is, as if no failure had occurred. If a web service crashes in the middle of a web service request, our solution ensures that any partial changes made as a part of this request are rolled back.

4.2 Assumptions

We assume that the web service client detects the failure of the web service and reissues the failed request. The failure of the web service is detected when the request times out and the underlying HTTP layer returns an error. The fact that the web service has moved to another machine can be hidden using standard load balancing techniques that virtualize the physical address of the machine providing the service. Alternatively, the client may need to relocate the service using facilities based on the proposed WS-RenewableReference specification.

This paper does not address the specific details of how the failure(s) of the web service instance(s) are detected or which component in the system is responsible for starting a new instance of the web service or notifying the backup to change its role. There are no established web services standards for such services yet, although the work on web services and grid computing standards at OASIS (Organization for the Advancement of Structured Information Standards), DMTF (Distributed Management Task Force), and GGF (Global Grid Forum) may provide appropriate standards in the near future. Without such standards, each SOA environment provides its own proprietary facilities. Typical monitoring systems for distributed systems (e.g., HP Openview, BigBrother [21]) allow scripts to be specified to be executed when a specific event (e.g., failure of the primary web service) occurs and such scripts can be used to start or activate a web service. Alternatively, systems specifically geared for automatic recovery through restart can be used [14]. Finally, the backup web service can be programmed to be activated if it receives a client request directly, assuming that the backup service only receives requests when the original web service has failed.

4.3 Durability Proxies

The durability proxies "wrap" state objects to make their state durable. The wrapping is implemented using the Java reflection API—Java Dynamic Proxy classes. Java Dynamic Proxies allow a method call to the object proxied to be intercepted and customized behaviors to be added before and after the actual object method call is completed.

Figure 3 illustrates the interface provided by a durability proxy. Specifically, each durability proxy pro-

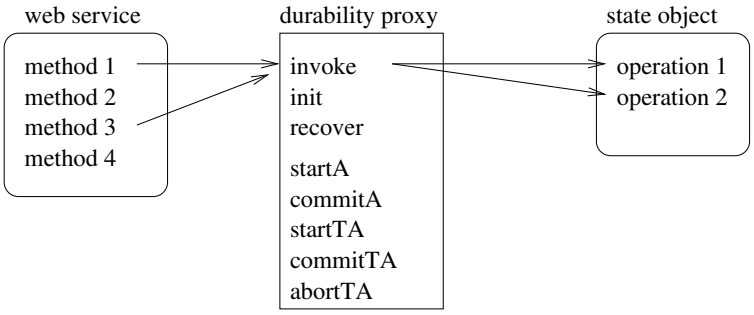

Figure 3: Durability proxy interface

vides an *invoke* operation that is invoked every time an operation on the proxied state object is invoked. If this operation updates the state object, the durability proxy saves the new state using its durability mechanism (e.g., database). Furthermore, each durability proxy provides *init* and *recovery* methods that are used to initialize the object state at startup and to restore the object state in case the original web service fails, respectively. For example, at initialization, a database proxy creates the database tables used for storing the object state and at recovery, the proxy at the backup or newly created web service instance loads the object from the database.

Finally, a durability proxy also implements five *action control* methods that the (modified) web service calls directly to ensure the desired atomicity properties for the web service request. These operations are used to ensure the desired atomicity for the web service method execution with and without WS Transactions:

- *startA:* start of a new web service request (atomic action),
- *commitA:* complete (commit) the state changes made during the execution of this web service request (complete atomic action),
- *startTA:* start web service transaction,
- *commitTA:* commit state changes made during the current web service transaction,
- *abortTA:* abort (undo) any state changes made during the current web service transaction.

The durability compiler modifies the web service code so that it issues calls to these proxy methods when appropriate, see section 4.5. Specifically *startA* is called at the beginning of the web service request and *commitA* at the end of the web service request for all the state objects for which the update operations are called during the processing of the web service request. The *startTA*, *commitTA*, and *abortTA* operations are called by the web service when it participates in the WS transaction coordination protocols. Note that when

a web service method is called inside a WS transaction, the *startA* and *commitA* operations have no impact since the state changes cannot be completed until the WS Transaction commits (or aborts). The durability proxy simply ignores these operations when it knows that the state object is involved in a WS Transaction.

Since it is impossible to implement efficient generic durability proxies, some of the state object-specific instructions used by the durability proxies are specified in the durability mapping, see section 4.4. The durability compiler combines the generic durability proxy code with the information specified in the durability mapping to generate state object specific durability proxies.

Replication proxy

As illustrated in figure 2, the replication proxy replicates the state object by maintaining backup copies of the object in backup copies of the web service. The replication proxy at the web service acts in the role of the primary proxy and the replication proxies at the backup web services act in the role of backup proxies. The replication proxy implements the basic proxy API as follows:

- *init*: create a TCP connection between the primary and backup proxies.
- *recover*: (backup) change role to primary and create TCP connections to new backup proxies.
- *startA*: (primary) create an empty update buffer at the primary.
- *invoke*: (primary) if the operation issued is an update operation, forward the request to the local state object and store the corresponding state update in the update buffer.
- *commitA*: (primary) send update buffer to backup(s), wait for acknowledgment. When a backup proxy receives this state update, it sends back an acknowledgment to the primary proxy. Note that the backup can issue the state updates after sending this acknowledgment.

Implementing transactional semantics involves deciding whether the failure of a web service instance (primary or any of the backups) should be masked from the WS transaction or whether the WS transaction should be aborted if one of the web service instance fails during the transaction. While doing the first would provide failure masking and slightly better service reliability, it would require changes to the WS-Transaction specifications. As WS-Transaction is currently defined, a service automatically joins a transaction when it receives a service request containing a new transaction object. If this service does not respond during the two-phase commit protocol, then the decision will eventually be to abort since the service will not be around to vote *yes* to commit the transaction. Hence, one would need to have a way to de-register the failed service with the transaction (which would be a change to WS-Coordination) and to allow the backup (or newly started service instance) to join the transaction. Another alternative would be to migrate the identity of the registered service from the failed service to the backup. Doing the latter would require some support for migration of identities.

The second approach can easily be implemented with the current WS-Transactions standards and is thus used by our current replication proxy. In this case, if a web service instance fails during the transaction, the transaction coordinators will eventually abort the transaction. After the web service has recovered, the client can reissue a transaction request similar to the case without transactions. When a transaction starts, the web service invokes the *startTA* method on the replication proxy and when the transaction commits or aborts, the web service invokes the *commitTA* or *abortTA* operations, respectively. The transaction context is passed as an argument to the *startTA* operation. The implementations of these operations are as follows:

- *startTA*: (primary) checkpoint current state (see durability mapping below); create update buffer; send transaction context to the backup(s), wait for acknowledgment(s); (backup) register as a participant in the transaction.
- *commitTA*: (primary) send update buffer to backup(s); wait for acknowledgement(s); discard checkpoint; (backup) apply state updates sent by primary; send an acknowledgement.
- *abortTA*: roll back state, discard update buffer.

Database proxy

As illustrated in figure 2, the database proxy backs up the object state in a database. When an update operation is invoked on the state object, the database proxy updates the object state in the database by issuing appropriate database operations. The *init* method creates the database tables necessary for storing the object state and the *recover* method at the database proxy at a backup (or newly created) web service issues the appropriate database query operations to load the current object state.

The action control operations in this case simply rely on the transactional support provided by the underlying database system. Both the atomic actions (defined by *startA* and *commitA* methods) and WS Transactions (defined by *startTA*, *commitTA*, and *abortTA*) are implemented by calling the methods for starting, committing, and aborting a transaction provided by the database. For example, if the Java Transaction API (JTA) is used, the operations could be `TransactionManager.begin`, `TransactionManager.commit`, and `TransactionManager.rollback`. Note that the *startA* and *commitA* operations check to see if the state object is already involved in a WS Transaction and if it is, they simply return without invoking any database operations.

When the state object is involved in a WS Transaction, it must be able to undo any changes made in case the transaction is aborted. Similar to the replication proxy case, the object state can be checkpointed at the beginning of the transaction (*startTA* operation); however, in this case the state can also be restored by simply using the *recover* operation that pulls the state from the database (after the transaction has been aborted). These operations can be specified in the durability mapping.

Reconstruction proxy

The reconstruction proxy is appropriate in situations where semantically durable state can be implemented by reconstructing the object state after failure. The *init* operation simply creates an "empty" state object (the definition of "empty" is object specific), while the *recover* operation invokes the *init* operation. The *invoke* operation forwards operations to the state object and *startA* and *commitA* operations do nothing (i.e., return).

To implement transactional semantics—particularly, to be able to abort a transaction—even this proxy has to implement the *startTA*, *commitTA*, and *abortTA* operations. A checkpointing strategy similar to the replication proxy can be used, or alternatively, if aborts are unlikely, it would be semantically valid to abort to an "empty" state (i.e., *abortTA* simply calls *init*).

4.4 Durability mapping

The durability mapping describes for each state object the durability mechanism chosen for this object and the state object specific instructions. These specific instructions may be required for the following operations:

- Initialization and recovery,
- for each update method of this object, the state update instructions,
- checkpointing and restoring instructions.

While Java object serialization can be used to implement both the state update and checkpointing, more efficient methods are often available depending on the state object semantics. For example, if the state object operations are deterministic, the state update in the case of a replication proxy can be achieved by simply executing the same operation on the object replica(s). For example, for a state object that uses the database proxy, the durability mapping could include the following:

- Initialization: Create database table(s) for the state.
- Recovery: Database instructions to query database for the object state and instructions to assign the state.
- State update(s): Database instructions to update a table based on operation parameters.
- Checkpointing: mapping specifies a no op.
- Restoring: Call the *recover* operation.

4.5 Durability compiler

The durability compiler takes the web service code (including the code for the state objects) and the durability mapping, and generates modified web service code and customized durability proxies for the state objects. The durability proxies are generated by taking the generic proxy code and inserting the required customization code from the durability mapping; see section 5 for an example. The output of the durability compiler is normal Java code that can be compiled and executed as a normal Java program.

The web service code is modified as follows. The initialization code of the web service is extended to create the durability proxies for each state object. All method calls to the state objects are replaced with calls to the corresponding durability proxy. For each method of the web service, the durability compiler inserts calls to the state objects' *startA* methods at the beginning and *commitA* methods at the end for all state objects that may be updated by the web service request. Note that the code for the backup web service is the same as for the web service except that it creates any replication proxies in the backup role and it does not call the *init* methods of the database proxies. It also provides a method that can be invoked to force the backup to assume the role of the primary web service. Some additional code to interact with the system monitoring and registry system may also be necessary at the web service and in the backups depending on the specifics of the system.

5 Examples

We use two services to illustrate our approach and evaluate the performance overhead imposed by the different durability mechanisms. The first service is the CounterService that is included as an example in the Globus Toolkit software and is built using WSRF. We use this simple service for micro-benchmarking throughput and latency. The second service is a simple resource allocation service, MatchMaker, modeled after existing (but more complex) services, such as the resource management tools in Globus[13], the non-grid service Condor match maker[22], and Java Market[2]. We designed our own Matchmaker service so we could experiment with the performance tradeoffs of services using more than one state objects.

Our performance evaluation is based on Globus Toolkit 4.0. Our configuration consisted of three dual-CPU Pentium III 2x2785 MHz workstations connected to a 1 Gbit Ethernet and running RedHat 9. One machine ran the clients, a second machine ran the server, and the third machine hosted the database server and was used to run the web service backup. We used MySQL 4.0 as our database servers. The durability compiler has not yet been implemented; so, the corresponding transformations were performed manually. Furthermore, since the Globus Toolkit does not yet provide support for WS Transactions, we did not implement transaction support.

5.1 The CounterService

The CounterService uses WSRF to maintain stateful information. This web service has one operation "add", which increments the value of the counter state by one and returns the incremented value. The state object, Counter, provides two operations: "setValue" and "getValue", and only "setValue" updates its state. We compared the performance of three service configurations: the original CounterService, the service with the Counter object protected using the replication proxy (PB service), and the service with the Counter protected using the database proxy (DB service). Since the reconstruction proxy does not perform any operations except when failures occur or when the state object is involved in a transaction (neither of which were present in the experiments), its performance would be identical to the original CounterService. Our test client sends a specified number of "add" requests sequentially to the service. For the latency tests, we used one client that sends 40,000 requests, while for the throughput tests we used c concurrent clients, $c \in \{1, 2, 4, 8, 12, 16\}$, where each client sends $40000/c$ requests.

The average request round-trip time (RTT) of the original service was 139 ms (with the 95% confidence interval ± 0.3 ms). The RTT of the PB service is essentially the same, while the average RTT of the DB service is 171 ms (with the 95% confidence interval ± 0.3 ms), which is an increase of 23%. Figure 4 shows the throughput of the three services. The throughput of the PB service is close to the throughput of the original counter service, especially when the number of clients is small. The throughput of the DB service is lower than the other two. For all three services, the throughput increases when the number of clients increases from 1 to 8, and does not change much when the number of clients increases from 8 to 16. The 95% confidence intervals of all the throughput values are no wider than ± 0.11. Thus, the confidence intervals would not be readable in the figure.

5.2 The MatchmakerService

The MatchmakerService keeps track of machines available in a grid and allocates them to clients requesting computing resources. The service provides two methods: "machineAdvertise" and "jobSubmit". Each machine periodically advertises its availability by invoking "machineAdvertise" with information about the resources it has available (e.g., CPU speed, available memory, available disk, etc). A client that requires a machine invokes "jobSubmit" with requirements for the desired machine. If suitable machines are available, Matchmaker choose ones and returns it to the client. Otherwise, it returns an error message indicating that machines of the requested type are currently not available.

The MatchmakerService contains two state objects: MachineQueue and AccountSet. MachineQueue maintains a list of available machines and AccountSet maintains usage records for each client used to charge the user for the resources. If the MachineQueue is lost, it can be reconstructed based on the periodic "machineAdvertise" calls but there is no way to reconstruct AccountSet if it is lost. Thus, these two state

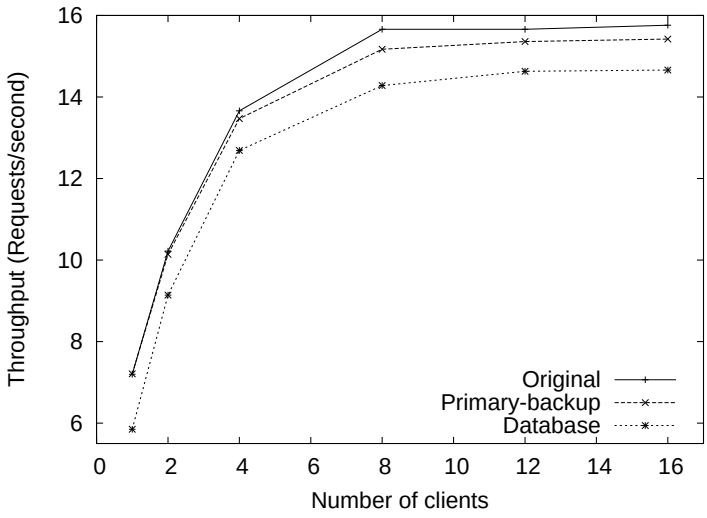

Figure 4: Counter Service Throughput

objects have different durability requirements. Note that "machineAdvertise" updates the MachineQueue, while "jobSubmit" may update both state objects.

Tables 5 and 6 illustrate the durability mappings for MachineQueue and AccountSet. For MachineQueue, the mapping specifies that a replication proxy should be used. The state update method is to execute the same method (with the same arguments) on the backup state object (keyword **repeat**). No special initialization or recovery code is needed in this case and we omit checkpointing and restoration instructions. For AccountSet, this mapping specifies that a database proxy should be used. The initialization section specifies the instructions for creating a database table for the object state. The recovery section specifies the intructions for retrieving the object state from the database. The state update methods are based on the arguments of the state object operations and map to simple database update operations.

<durability proxy> <type> Replication </type>
  <update methods>
    <method><name = insertNode><state update> **repeat** </state update></method>
    <method><name = deleteNode><state update> **repeat** </state update></method>
  </update methods>
</durability proxy>

Figure 5: Durability mapping for MachineQueue

We compared the performance of four service configurations: the original MatchmakerService, one using replication proxies for both states objects(PB), one using a replication proxy for MachineQueue and a database proxy for AccountSet (PB+DB), and one using database proxies for both state objects (DB). Our

```
<durability proxy> <type> Database </type>
  <initialization> CREATE TABLE bills (clientID INT, balance INT); ENGINE = INNODB; </initialization>
  <recovery> SELECT * FROM bills; for (each line) insertBill(clientID, balance); </recovery>
  <update methods>
    <method><name = insertBill>
    <state update> INSERT INTO bills VALUES (arg[0], arg[1]); </state update></method>
    <method><name = setBill>
    <state update> UPDATE bills SET balance = arg[0] WHERE clientID = arg[1]; </state update></method>
  </update methods>
</durability proxy>
```

Figure 6: Durability mapping for AccountSet

experiment has one client first advertise a large enough number of machines that all subsequent "jobSubmit" calls can be satisfied. Then, four clients generate load for the MatchmakerService. Each client executes the following sequence of actions 30 times: (1) pick a random number n between 1 and 5; (2) send MatchMaker n requests, with each request asking for one machine; (3) pick a random number C from a specified "computation interval"; (4) sleep for C seconds. The sleep period simulates the time during which the client is using the allocated machines.

The results are shown in figure 7. As expected, the average latency of PB is the closest to the original service: it is about 60 ms less than DB. The performance of PB+DB is between PB and DB. These results indicate that by using a cheaper durability mechanism for the less critical state object, the overall performance of the service can be improved compared to using a database for both. When the computation time interval increases, the average latency decreases, since the chance that two clients invoke the MatchmakerService at the same time is reduced.

6 Related work

The fault tolerance of services in distributed systems has been an issue since the inception of distributed computing. The basic techniques used to achieve such fault tolerance are fundamentally the same: the state of the service is secured against failure using some form of redundancy. We classify the related work based on the view taken on the service state.

One traditional approach is to view the service state as a part of the server, and provide state durability by making the service fault tolerant via server replication. This approach was formalized by the *replicated state machine approach* [24]. Numerous distributed computing platforms based on group communication take this approach for replicating processes to provide fault-tolerant services. This approach was also standardized as the method for providing fault tolerance in CORBA [19] and it was also used in our earlier work

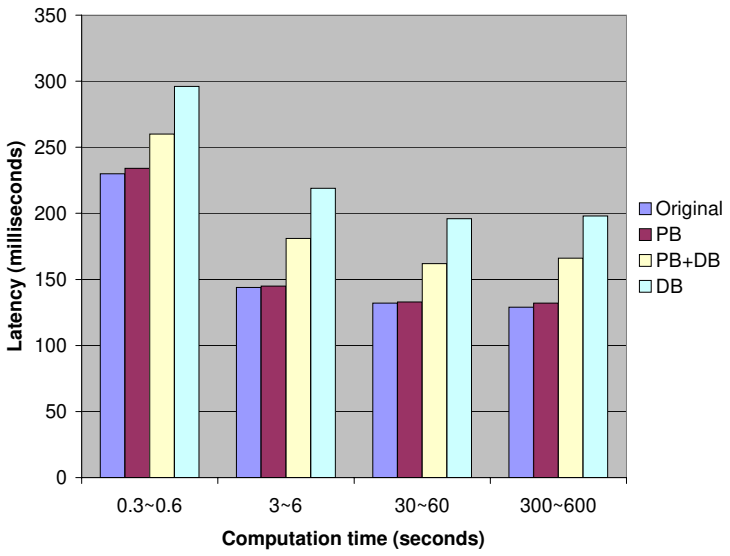

Figure 7: MatchmakerService Performance

on providing fault-tolerance for grid services based on the OGSI model [27]. Such an approach has also been proposed for web services in the form of FT-SOAP [17].

Another approach taken to securing service state is to use stable storage provided by a file system or a database. In an SOA, each state change is typically stored to a database immediately before a reply is returned to the client. The Java J2EE [7] architecture for 3-tier e-commerce services provides a durability mechanism transparent to the programmer in the form of *entity beans*. The J2EE runtime environment (application server) provides database based persistence for entity beans transparently. Complete transparency has performance implications, and therefore, systems such as Hibernate have been designed to provide object persistence at a lower cost [6]. Hibernate is a framework that allows the mapping of a data representation from a Java object model to a relational data model with an SQL-based schema. Hibernate maps Java classes to database tables at runtime. Finally, [26] proposes a replication algorithm that provides both state consistency and exactly-once semantics for stateful J2EE application servers.

The availability of stable storage implemented as a database can be increased using replication. Database replication at the middleware level has recently received considerable attention [18, 3, 4, 20, 23] since it can support a heterogeneous environment without the need to change the underlying database system.

Finally, [15] presents models for evaluating the dependability of data storage system, including both individual data protection techniques and their compositions. These models estimate storage system recovery time, data loss, normal mode system utilization, and operational costs under a variety of failure scenarios.

To our knowledge, no prior work provided the flexibility of allowing different techniques to be used transparently to improve the service state durability.

7 Conclusions

This paper addresses the increasingly important issue of how to make web services, or services in an SOA in general, highly available and fault tolerant. Our work is based on the observation that if the state of the service can survive failures (i.e., is durable), it is relatively easy to construct highly available services. However, durability may have a considerable performance overhead depending on the specific techniques used. Therefore, our approach allows the transparent customization of the durability techniques used for different parts of the service state, that is, different objects that store the service's state. The durability requirements and chosen techniques can be determined after the service and its state objects have been implemented. Our approach is based on using a durability compiler that takes the service implementation, a service-specific durability mapping specification, and reusable durabability proxies and generates code where each state object in the web service is protected by the chosen durability technique. To our knowledge, our system is the first to offer such a level of customization of the durability/performance tradeoff. While we have not implemented the durability compiler, we anticipate that its complexity is comparable to a stub compiler.

Our performance measurements show that being careful about choosing the appropriate durability techniques can significantly boost performance, with the gain increasing as the number of state objects accessed in a single client invocation increases. We also anticipate that the relative performance gain will improve further as more efficient implementations of web services platforms are developed.

References

[1] The globus alliance. http://www.globus.org.

[2] Y. Amir, B. Awerbuch, and R. S. Borgstrom. Managing checkpoints for parallel programs. In *Proceedings of the 1st International Conference on Information and Computation Economies (ICE-98)*, 1998.

[3] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, July 2002.

[4] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proceedings of the fourth ACM/IFIP/USENIX International Conference on Middleware*, June 2003.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, Jan 2004.

[6] C. Bauer and G. King. *Hibernate in Action*. Manning Publishing Company, Aug 2004.

[7] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, and B. Stearns. *The J2EE Tutorial*. Addison-Wesley, Mar 2002.

[8] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web service architecture, Feb 2004. http://www.w3.org/TR/ws-arch.

[9] L. F. Cabrera, G. Copeland, W. Cox, T. Freund, J. Klein, D. Langworthy, I. Robinson, T. Storey, and S. Thatte. Web Service BusinessActivity (WS-BusinessActivity), Nov 2004. http://www6.software.ibm.com/software/-developer/library/ws-busact200401.pdf.

[10] L. F. Cabrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, D. Langworthy, A. Nadalin, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey. Web Service Coordination (WS-Coordination), Nov 2004. http://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[11] L. F. Cabrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, D. Langworthy, A. Nadalin, D. Orchard, I. Robinson, T. Storey, and S. Thatte. Web Service Atomictransaction (WS-AtomicTransaction), Nov 2004. http://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.

[12] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource Framework, Jan 2004. http://www.globus.org/wsrf/specs/ws-wsrf.pdf.

[13] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in computational grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, 1999.

[14] K. Joshi, M. Hiltunen, W. Sanders, and R. Schlichting. Automatic model-driven recovery in distributed systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*, pages 25–36, Oct 2005.

[15] K. Keeton and A. Merchant. A framework for evaluating storage system dependability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004)*, 2004.

[16] B. Lampson. Atomic transactions. In *Distributed System-Architecture and Implementation*, pages 246–265. Springer-Verlag, 1981.

[17] D. Liang, C.-H. Fang, C. Chen, and F. Lin. Fault tolerant web service. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 310–319, Dec 2003.

[18] J. M. Milan-Franco, R. Jiménez-Peris, M. Patino-Martínez, and B. Kemme. Adaptive middleware for data replication. In *Proceedings of the fifth ACM/IFIP/USENIX International Conference on Middleware*, Oct 2004.

[19] Object Managemnt Group. Fault tolerant CORBA. In *Common Object Request Broker Architecture: Core Specification*, chapter 23, pages 955–1059. Object Management Group, Dec 2002.

[20] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the fifth ACM/IFIP/USENIX International Conference on Middleware*, Oct 2004.

[21] Quest Software. Big brother professional edition, 2005. http://www.quest.com/bigbrother/.

[22] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, 1998.

[23] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the globdata middleware. In *Proceedings of the Workshop on Dependable Middleware-Based Systems*, June 2002.

[24] F. Schneider. Implementing fault-Tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec 1990.

[25] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open Grid Services Infrastructure (OGSI), June 2003. http://xml.coverpages.org/OGSI-SpecificationV110.pdf.

[26] H. Wu, B. Kemme, and V. Maverick. Eager replication for stateful J2EE servers. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Oct 2004.

[27] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. Schlichting. Fault-tolerant grid services using Primary-Backup: Feasibility and performance. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing (Cluster 2004)*, Sep 2004.