Fault–tolerant Grid Services Using Primary–Backup: Feasibility and Performance

Xianan Zhang* xzhang@cs.ucsd.edu
Dmitrii Zagorodnov* dzagorod@cs.ucsd.edu
Matti Hiltunen† hiltunen@research.att.com
Keith Marzullo* marzullo@cs.ucsd.edu
Richard D. Schlichting† rick@research.att.com

Abstract

The combination of Grid technology and web services has produced an attractive platform for deploying distributed applications: Grid services, *as represented by the Open Grid Services Infrastructure (OGSI) and its Globus toolkit implementation. As the use of Grid services grows in popularity, tolerating failures becomes increasingly important. This paper addresses the problem of building a reliable and highly-available Grid service by replicating the service on two or more hosts using the primary–backup approach. The primary goal is to evaluate the ease and efficiency with which this can be done, by first designing a primary–backup protocol using OGSI, and then implementing it using Globus to evaluate performance implications and tradeoffs. We compared three implementations: one that makes heavy use of the notification interface defined in OGSI, one that uses standard Grid service requests instead of notification, and one that uses low-level socket primitives. The overall conclusion is that, while the performance penalty of using Globus primitives—especially notification—for replica coordination can be significant, the OGSI model is suitable for building highly-available services and it makes the task of engineering such services easier.*

1 Introduction

A Grid infrastructure, being a collection of resources, is prone to many kinds of failures: application crashes, hardware faults, network partitions, and unplanned resource downtime. Most Grid platforms have mechanisms for tolerating at least some kinds of these failures. These mechanisms typically retry failed executions, perhaps starting at a recent checkpoint [13, 4]. Furthermore, the need for fault tolerance in Grid infrastructures is well known; an overview of these techniques and a unifying failure handling framework, called *Grid Workflow*, is given in [12].

Recently, however, the emphasis in the Grid standardization efforts has moved from a focus on supporting job execution to service-oriented architectures that can be used not only for the traditional resource-intensive scientific computation tasks, but also as a general distributed computing platform. Specifically, the recent GGF (Global Grid Forum) standards define Grid computing platforms as collections of *Grid services* [9][1] that include services that provide the Grid computing infrastructure, e.g., scheduling, and monitoring, as well as services that comprise the Grid application itself. While the fault-tolerance issues have been extensively explored in the traditional job execution scenarios, the issue of handling failures in the Grid services model as represented by the Open Grid Services Infrastructure (OGSI) [10] and its Globus [8] toolkit 3 (GT3) implementation, remains largely unexplored. Given that Grid services are expected to become the basis for commercial as well as scientific applications, such support is critical for wide-scale acceptance.

This paper addresses the problem of building highly available Grid services by replicating a service on two or more hosts. Making services highly available is not a new research area: it has been a research topic for decades, and there are commercial products for making non-Grid services highly available. So, one

*Department of Computer Science and Engineering, University of California, San Diego
†AT&T Research Lab

[1] a new standard, Web Service Resource Framework (WS-RF) works on the convergence of web services and grid services.

might assume that some standard approach could be used for Grid services. Which approach to use, however, requires some study:

1. While there are currently only a few examples of Grid services, a common theme is that such services are expected to be *stateful*. This is in contrast to the closely related technology of Web services, which are *stateless* in that any changes to the service's state that must persist across failures is recorded in a database. If the machine executing a Web service fails, then the client can rebind to another machine (perhaps via a load balancer) that can reference the persistent state in the database[2]. Commercial products, like the Veritas cluster manager [16], also assume that a failed server can be recovered by restarting it on another machine: any persistent state is kept in files or a database. A service that is stateful can have a lower latency (by avoiding writing state to a database) and have a simpler client-server protocol.

2. Extrapolating from existing services that are part of Grid systems, we can expect some of the services to have nondeterministic behaviors. A nondeterministic service by itself offers no problems to a client. Maintaining the consistency among replicas of a nondeterministic service is a problem, though: two replicas may become inconsistent even though they execute the same sequence of commands.

   Nondeterminism can arise from many sources. At one end of the spectrum, a service's methods may be explicitly nondeterministic. For example, it is often better to randomly choose a "good" resource than to deterministically choose the "best" resource, either because it is computationally easier to do or because doing so spreads the load among several resources [3]. At the other end of the spectrum, nondeterminism may arise from timing issues, such as when exactly an external event (like an interrupt) occurs. For example, a resource manager service may use a lease-like mechanism to reclaim resources: if, after an interval of time, the use of a resource is not reconfirmed, then the manager automatically reclaims the resource. Consider a resource management service with two replicas where a client requests a resource of some class C. Let there be two resources c_1 and c_2 of class C, and c_1 has been previously allocated. One replica may get the client's request before the lease for c_1 has expired, and so allocate c_2 to the client. The second replica may instead get the request after the lease has expired and allocate c_1 to the client.

3. To make service integration easier, Grid services are designed using very high-level protocols and services. All else being equal, one would not expect that a service built on top of, say, SOAP will have the same latency as a service built directly on top of TCP. Using the OGSI functions to provide for high service availability is appealing: one could provide it as a feature portable across any OGSI service. But, if the performance is much worse than the same feature implemented at a low level, then performance may outweigh the engineering appeal of a high-level implementation.

[2]Some state, like a "shopping basket", can be bound to an individual server. Losing such state due to a server failure isn't often seen as being critical.

The first two points suggest using a *primary–backup approach* for service availability. Point one requires replicas to be consistent with each other. For example, a client could interact with replica r_1, and then start interacting with another replica r_2 if r_1 fails. Anything that the client knows from r_1 about the state of the service should also be known by r_2. Point two is addressed by having only one replica, the primary, respond to requests. The primary will keep one or more backup services consistent and ready to take over should the primary fail [6].

The primary goal of this paper is to evaluate the tradeoffs associated with using primary–backup as a fundamental technique for building highly available Grid services in the context of OGSI and Globus. Much of this focuses on the third point above— the tradeoff of performance versus use of facilities provided by the OGSI standard. We first designed a primary–backup protocol using OGSI to determine whether it supplies the necessary features, such as state update and client rebinding, and to see what changes might be needed to support such an approach. As described in Section 2, we found that it is not hard to accommodate primary–backup, and that the solution is simple and requires only small changes to the service to handle non-determinism. The use of the OGSI notification interface to handle replica updates is perhaps the key distinguishing feature of this approach.

We then implemented this approach using GT3 to better understand the performance implications and tradeoffs of doing primary–backup at such a high level. In particular, using a simple example Grid service, we compared the performance of this notification-based approach to variants in which replica update is done using standard Grid service method calls and TCP, respectively. Our example Grid service implements a sim-

plified version of the Condor Matchmaker service [14]. Nondeterminism arises in this service both from the way resources are selected and from priorities.

Section 3 gives the performance results. We found the performance penalty was, in fact, quite high. While some of this may result from the lack of performance tuning in GT3, we believe that our findings also have larger implications related to how and where replication should be used to provide fault tolerance in Grid service architectures.

We do not consider client failure in this paper. One of the attractions of the primary–backup approach is that it defines a very simple client–server protocol that does not depend on clients being reliable. In other words, the correctness of the server, in terms of how it responds to requests, does not depend on help from the clients, which means that client failures can be dealt with using orthogonal approaches such as timeouts and leases [15]. We also do not consider software bugs that can lead to completely correlated failures. In this case, the primary and all backups could simultaneously crash. Again, there are separate techniques that are used in practice for tolerating such failures.

2 Architecture

We first describe the primary–backup approach to replication. We then cover the concepts behind Grid services, and then give a design of a primary–backup service on top of OGSI.

2.1 Primary–Backup replication

Primary–backup is a well-known technique for making services highly available [1, 5, 6]. A client sends a request to the primary, which receives and executes the request. The primary then sends a state update message to the backups and replies to the client. Typically, the primary does not reply to the client until it knows that all backups have received the state update. This is done to ensure that the backups are always consistent with the client: it is impossible for the client to know that the primary executed the request without the backups also knowing this. Figure 1 shows a space-time diagram of the execution of a simple primary–backup protocol.

Primary–backup requires that a client be notified that the primary has failed and allow the client to rebind to the newly-appointed primary. Ideally, this ability should be available below the level of the service request: doing so allows a client designed to interact with a single non-replicated server to be transparently ported to interact with a primary–backup service.

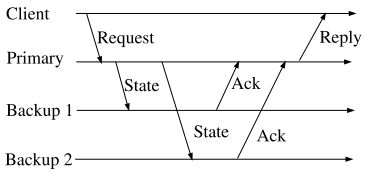

Figure 1. Primary-Backup Protocol

Primary–backup has a few drawbacks. First, it is best suited for tolerating benign failures such as crashes and message loss, rather than arbitrary or malicious failures. Unless malicious failures are a concern in a specific Grid environment, we consider masking only benign failures a worthwhile tradeoff for the ability to run non-deterministic applications. Another drawback is that primary–backup requires that the environment is synchronous enough to support the use of heartbeats to detect failures. In practice, this means that the primary and the backups need to run on a cluster that is managed by some failure detector and recovery manager. Commercial products, such as the VERITAS Cluster Server [16], can be used for this purpose.

2.2 Grid Services

Grid services in OGSI combine Grid technologies with Web services to provide a platform for building distributed applications. Unlike Web services, Grid services are stateful and may be short-lived. The OGSI model allows each client to choose among several available instances of a service or create its own instance. The instances may have a limited lifetime since resources can be created to serve certain clients and are removed after they are no longer needed.

Interaction between a Grid service and a client happens in a request-reply fashion using strictly-defined interfaces and a certain encoding of data (interfaces are described by WSDL and the messages are encoded using SOAP, both of which are XML-based). In addition to regular requests and replies, Grid instances may subscribe using an OGSI-specified interface to *notifications*, which asynchronously alert subscribers (called *sinks* in OGSI terminology) of state changes. Using notifications avoids wasteful polling.

Notifications can be of two types: a *push* notification sends information along with the notification, whereas a *pull* notification is used to indicate that something has changed: it is up to the notification subscriber to request (or pull) the information using a regular request. Pull notification gives the subscriber the freedom to decide whether and when to get information associ-

ated with the notification, while push notification avoids the overhead of that additional call in situations where the information is needed immediately.

2.3 Primary–backup for Grid Services

There are three general problems that any implementation of a primary–backup mechanism needs to solve:

1. *Transfer of application state.* Before replying to the client, the primary needs to send the change in its state to the backups. A reply can be sent to the client only when it is known that the backups will eventually apply the state change.

2. *Detecting failures.* Crashes and lost messages need to be detected. This is normally done by setting a timeout for every message. If no messages are sent for a long time, then a *heartbeat* message can be sent to check on a machine.

3. *Switching to a new primary.* Originally, one of the service instances is designated as a primary and others as backups. After a failure of the primary, the backups agree on a new primary and ensure that all future requests are directed to it.

Grid service notifications are a natural mechanism for solving all three of these problems because state updates and failures are inherently asynchronous events. Also, notifications provide a simple mechanism for disseminating information to a number of interested parties—several backups may be interested in the same state update, and several clients may be interested in the same failure notification. Consequently, in our system, backups register with the primary as sinks for state update notifications and heartbeat notifications, and each client registers with each backup as a sink for the failover notification that tells it to switch to a different primary and to resend the last request if it was expecting a reply. We use a push notification for the state transfer because the backup needs every state update. For heartbeat and failover, pull notifications are used because there is no data associated with those events.

The normal execution proceeds as follows. A client makes a Grid service request to the primary, which executes the request. When execution ends, the change in the state of the service is extracted and sent to the backups via a notification. When the primary collects acknowledgments from all backups it replies to the client. The state extraction and injection are application-specific: the Grid service needs to support methods that allow this to be done. In addition, the service can be designed to have the primary send checkpoints to the backups if its computation is long-running.

Failure of the primary is detected by backups when they do not receive a heartbeat message after a certain period of time. This method allows detection of host and task crashes, as well as network partitions. At that point, the backups need to cooperate in election of the new primary. The newly elected primary then sends a failover notification to the client so it can obtain a new server instance handle. If the client was expecting a reply from the service when a failover notification arrives, then the client resubmits the request to the new service instance. If the old primary had already sent a state update to the new primary, then the new primary can reply with the result computed by the old primary. Otherwise, it can compute the result itself (perhaps starting from a checkpoint if the primary had sent checkpoints to the backups).

Failures of the backups do not interfere with the operation of the surviving system components, so the only new issues are the detection of backup failures and the integration of new backups into the system. Neither is conceptually hard to implement, although integration of a new backup may require a large amount of state to be transferred. The details of how to best do this are outside the scope of this paper.

2.4 Implementation details

In this section we give a more detailed overview of our system using pseudocode to illustrate key actions performed by each of the three participants: a client, a primary, and a backup. Each one is enclosed in an object with private variables and methods. Note that we use C language convention for pointers: $\&x$ is a reference to variable x.

The client code, shown in Figure 2, is interposed between the client application and the original SOAP stub in such a way that client code is not changed. The original stub supports the INIT method, which is called when the client binds to a Grid service, and a number of operations, shown here collectively as OP. We intercept INIT with the INIT_CLIENT method to register for receipt of failover notifications from each backup. The OP method spawns a separate thread, implemented by INVOKE_OP, to invoke the operation via the original stub.

If the primary crashes during this invocation, two things will happen in arbitrary order: the call to $stub.op()$ will return an error message, and a failure notification will arrive from the backup, causing FAILURE_HANDLER method to execute. To reduce the

```
var target // points to the current primary
var ops // contains records of on-going operations

INIT_CLIENT ( ) // called when client binds to a Grid service
    (replica_1, replica_2, ...replica_n) ← find_replicas();
    target ← replica_1;
    ops ← ∅;
    for each host ∈ {replica_2...replica_n} do
        register_notification(&FAILURE_HANDLER, host,
        FAILURE);
    stub.init();

OP (params)
    var op // object for holding operation parameters
    var done // a semaphore for waiting until success
    var result // object for holding results of executions
    op ← { params, &done, &result };
    ops ← ops ∪ &op; // add op to the list of on-going operations
    create_thread(&INVOKE_OP, &op);
    wait_on_semaphore(&done); // wait for someone to succeed in op
    ops ← ops \ &op; // remove op from the list
    return result;

INVOKE_OP (op)
    var result // object for holding results of executions
    result ← stub.op(op.params); // make the SOAP call
    if result ≠ failure then
        op.result ← result; // pass result back to OP()
        signal(op.done); // wake it up

FAILURE_HANDLER (new_primary)
    target ← new_primary;
    for each op ∈ ops do
        create_thread(&INVOKE_OP, op);
```

Figure 2. Pseudocode for the fault-tolerant stub on the client.

failover duration, we don't wait for the $stub.op()$ to return (it may take a while for the TCP socket to time out), so we re-submit the request to the new primary in FAILURE_HANDLER by spawning another INVOKE_OP thread. The parameters for this invocation are kept in the list ops. Eventually, some invocation should succeed, allowing OP to wake up and return the result.

The OGSI model specifies a method for clients to deal with failures of Grid service instances. Specifically, each Grid service has a persistent handle called a GSH (Grid Service Handle) and this handle can be resolved into a handle, called a GSR (Grid Service Reference), for an instance of this Grid service. The GSR may become invalid over time and the client can reacquire a valid GSR by re-resolving its GSH. The handle resolution is performed by a Grid service called the Handle Resolution Service. Although our design could incorporate this approach, in this paper we use a design that by-passes the Handle Resolution Service for two reasons:

- The Handle Resolution Service, if not fault-tolerant itself, would provide a single point of failure that could make all Grid services that rely on it unavailable.

```
var rate_sending // time interval for sending heartbeats
INIT_PRIMARY ( )
    claim_notification_source(HEARTBEAT); // register as source
    claim_notification_source(STATE_UPDATE);
    schedule(&HEARTBEAT_GENERATOR, rate_sending); // run
    regularly

HEARTBEAT_GENERATOR ( )
    notify_change(HEARTBEAT); // send a notification

EXECUTE (request)
    var result // object for holding results of executions
    result ← check_previous_requests(request);
    // if request is completed result is not NULL, but for new requests it
    is
    if result = NULL then
        var state // encoding of application state
        result ← service.op(request.params); // execute the request
        state ← service.extract_state(); // obtain state of the
        application
        notify_change_with_ack(STATE_UPDATE, state); // waits for
        acks
    return result;
```

Figure 3. Pseudocode for the primary service.

- In the handle resolution approach, the client only detects the failure of the primary when it attempts to use its GSR. In our approach, the client is notified immediately.

The code on the replicas is interposed between the Grid infrastructure and the service implementation. For each client OP there is an implementation of that operation on the server. To make stateful primary–backup replication possible, the service must implement two additional methods for state transfer: $extract\_state()$ and $inject\_state()$. Ideally, state transfer can be done by a small set of values describing all the relevant application state, but in the extreme it could be a full application checkpoint.

On the primary, as shown in Figure 3, we intercept each one of the operations with the EXECUTE method. It first checks whether this request has already been processed—this can happen when a server crashes after sending the state to the backups, but before replying to the client. In that case the old result is returned without executing the request. Otherwise, the request is executed, followed by the extraction of state, which is sent to backups via notifications. Note that $notify\_change\_with\_ack$ blocks until it gets an acknowledgment from every backup. In the initialization routine, the primary advertises itself as a source of two types of notifications (HEARTBEAT and STATE_UPDATE) and schedules a heartbeat routine to run regularly.

Figure 4 shows the pseudocode for backups. During normal operation they receive two kinds of no-

```
var rate_checking // time interval for checking for notifications
var last_notification // timestamp of the last notification
var primary_is_up // boolean flag
var senior // this is the senior backup

INIT_BACKUP ( )
    (replica_1, replica_2, ...replica_n) ← find_replicas();
    primary_is_up ← TRUE;
    if my_url() = replica_2 then
        senior ← TRUE;
    register_notification(&HB_HANDLER, replica_1, HEARTBEAT); // register sinks with primary
    register_notification(&STATE_HANDLER, replica_1, STATE_UPDATE);
    claim_notification_source(FAILURE); // register as a source for clients
    schedule(&FAILURE_DETECTOR, rate_checking);
    SETUP_SENIOR(replica_2);

SETUP_SENIOR (senior_url)
    if senior = TRUE then
        claim_notification_source(HEARTBEAT);
        claim_notification_source(STATE_UPDATE);
    else
        register_notification(&HB_HANDLER, senior_url, HEARTBEAT);
        register_notification(&STATE_HANDLER, senior_url, STATE_UPDATE);

FAILURE_DETECTOR ( )
    if ( current_time() − last_notification ) > rate_checking then
        if senior = TRUE then
            switch_to_primary();
            notify_change(FAILURE); // notify client
        else
            if primary_is_up = TRUE then
                primary_is_up ← FALSE; // wait for the next timeout
            else
                INIT_BACKUP(); // backups re-initialize, electing a new primary
    else
        if primary_is_up = FALSE then
            primary_is_up ← TRUE;
            (replica_1, replica_2, ...replica_n) ← find_replicas(); // elect new senior
            SETUP_SENIOR(replica_2);

STATE_HANDLER (state)
    service.inject_state(state);
    last_notification ← current_time();

HB_HANDLER ( )
    last_notification ← current_time();
```

Figure 4. Pseudocode for the backup service

tifications: their STATE_HANDLER receives state updates and injects the state into the service application and their HB_HANDLER receives heartbeats. Both store the current timestamp in the global variable $last\_notification$. FAILURE_DETECTOR checks this variable to make sure it is not stale. If it is then the primary is assumed to have failed.

Switching to a new primary can take a long time because it needs to register as a source of notifications and all backups must re-bind to the new primary. If we delayed client-bound failover notification until re-binding is complete, the failover time of our system would be extremely large (binding can take seconds!). We avoid this performance penalty by binding all backups to one special backup, which we call the *senior* backup, at the time of service initialization.

If a failure is detected, the senior backup becomes the primary and notifies the client immediately, since it already has all backups registered with it to receive state updates and heartbeats. The remaining backups then chose a new senior and bind to it "off-line", without delaying processing of client requests. This binding is implemented in the SETUP_SENIOR method, which is called during initialization and during recovery. In the rare situation that the senior backup fails together with the primary, all surviving backups will assume the failure of the senior after the second missing heartbeat and they will go through full re-initialization by calling INIT_BACKUP.

For simplicity, the pseudocode shows that the state is applied immediately by calling $inject\_state$ in STATE_HANDLER. In a real implementation it it would be better to queue up the state update, send back an acknowledgment and apply the state later, so as to impose as small of a penalty on the response time as possible. As implemented, the protocol queues state updates and applies them later in this way. Doing so can slow down failover because the backup may have to apply queued

state messages before processing new requests.

3 Performance

While it appears that OGSI is a suitable platform for building primary–backup fault tolerance, the overhead of replication may ultimately determine whether the technique is useful in practice. In this section, we describe the performance of our prototype implementation using GT3.

Our example highly available Grid service is a simplified version of the well-known Condor Matchmaker service. We measured the transfer overhead, the request response time, and the failure notification overhead of a prototype service structured according to the primary–backup approach described above. We performed experiments on a pair of dual-CPU Pentium II 300MHz workstations with 400Mb of memory, running Linux 2.4. We only considered a system with a primary and one backup, since this is by far the most common way primary–backup is used.

3.1 Matchmaker Grid Service

We designed the Grid Matchmaker service based on existing (but more complex) non-Grid services, such as Condor Matchmaker [14], Java Market [2], and the resource management tools in Globus [7]. We chose to use this service because it is an example of an important class of Grid service, and because is inherently nondeterministic.

Our Matchmaker service keeps track of machines available in the Grid, accepts requests for allocating machines, and maps each request to a suitable machine. There are two kinds of requests: one is a *resourceAdvertise* request, and the other is a *jobSubmit* request. A resourceAdvertise request provides information about a machine that is available for allocation. The input of this request is: the resource ID, the available CPU speed, the available memory size, the available disk size, the machine's IP address, and an identification string used to implement a simple capability for using the machine. A jobSubmit request sends a specification for a desired machine. If there are suitable machines available, then the Matchmaker service will choose one and send the address of this machine and the identification string back to the client. The input of this request is: the job ID, the required CPU speed, the required memory size, the required disk size, the priority of the job. The response of this request is the address and identity of the chosen machine, if there is one available; otherwise, the request returns a null string.

This Matchmaker service is non-deterministic for two reasons. First, if there are several machines that satisfy a jobSubmit request, then the machine that is allocated can be nondeterministically chosen. Second, the Matchmaker service is implemented by two threads: one enqueues requests and one executes enqueued requests. Requests are enqueued in priority order, and is FIFO within each priority. Two servers S_1 and S_2 could behave differently because of these rules on priority. Consider two jobSubmit requests: r_h is a high priority request and r_ℓ is a low priority request. Let r_ℓ arrive at the servers before r_h. If server S_1 is slower than S_2, then r_ℓ may arrive at S_1 when it is busy and arrive at S_2 when it is not busy. If r_h arrives shortly thereafter, then S_1 will execute r_h before r_ℓ and S_2 will execute r_ℓ before r_h.

3.2 State transfer

To fully understand the sources of overhead in state transfer, we compared the implementation using OGSI notifications for state updates (labeled *Notification*) to two alternative implementations, one that uses direct Grid service method calls (labeled *Call*) and one that uses TCP connections (labeled *Socket*). In the following tables we present the median, the mean, and the standard deviation for a set of 20 round-trip measurements. To better understand the overhead of state updates, our service can be configured to send an arbitrary amount of data in each state update.

First, Table 1 shows the round-trip time of a single state update, for a number of different state sizes (from 10 bytes to 100 kilobytes), as measured on the primary. Not surprisingly, *Socket* always has the smallest round-trip time, but this advantage goes from around 200 times faster for 10B updates to only 1.4 times faster when the state size is 100 kB. *Call* has intermediate round-trip times: at 10B, it is about 20 times slower than *Socket*, while at 10 kB it is only 2.5 times slower.

Note that the round-trip times for *Notification* are mostly insensitive to the size of the state update. Essentially, the cost of sending 10 bytes and 100 kilobytes with a notification is roughly the same. We think the cause of this lies in the format of GT3 notification messages; this is something that might be worth examining for later versions of the toolkit.

We also observed very high variance in samples: the standard deviation is sometimes higher than 50% and in one case is larger than the mean. This last case is due to a single outlier in the *Call* experiment for 100 kB of state update, which took 1.7 seconds. We think that much of this large variance is an artifact of Java garbage collection or other background processing in the Java

Table 1. State transfer round-trip time (milliseconds)

		10 B	100 B	1 kB	10 kB	100 kB
Notification	Median	192.5	189.0	193.0	195.5	190.0
	Mean	201.3	191.8	196.7	199.8	192.7
	St. Dev.	29.8	13.0	15.5	23.6	14.8
Call	Median	19.5	19.0	26.5	30.0	209.0
	Mean	26.4	24.2	33.1	32.6	299.0
	St. Dev.	12.0	9.2	17.9	6.3	333.0
Socket	Median	1.0	2.0	2.5	12.0	133.0
	Mean	1.5	1.7	2.5	14.8	144.5
	St. Dev.	0.8	0.5	0.5	11.9	32.4

Table 2. Client request round-trip time (milliseconds)

		10 B	100 B	1 kB	10 kB	100 kB
Notification	Median	242.0	241.0	240.0	238.5	232.0
	Mean	252.3	247.4	251.2	301.1	241.2
	St. Dev.	41.9	36.5	36.6	257.9	34.2
Call	Median	65.0	72.0	76.5	74.0	261.0
	Mean	71.4	78.0	89.7	82.5	350.8
	St. Dev.	21.9	28.9	40.9	21.9	333.5
Socket	Median	45.5	47.0	48.5	55.5	182.0
	Mean	52.8	56.5	55.5	62.6	195.9
	St. Dev.	21.6	26.6	21.8	22.9	50.4

virtual machine or the Grid container.

Table 2 shows round-trip times of client requests during normal, failure-free operation, as measured by the client. We would expect these numbers to be, approximately, the sum of the request round-trip time without primary–backup replication plus the state update overhead shown in Table 1 above. That is, indeed, the case since the request round-trip time of a normal Grid service, without replication, is 54.3 ms on average with the median being 44 ms. The data from the previous two tables is summarized graphically in Figure 5, where client request round-trip time is broken down into interaction between the client and the primary (white) and the interaction between the primary and the backup (solid, upward diagonal, and downward diagonal).

In Table 3, we normalized the data of Table 2 by dividing the median and mean numbers by the median and mean of the normal Grid service round-trip. So, each number shows the magnitude of overhead imposed by replication. The table shows that with *Socket* the median overhead of replication is small: for small state sizes (up to 10 kB) is 30% or less. With *Call*, the median overhead is 70% or less for small state sizes. For large state sizes all approaches perform similarly, with overheads of 400% and more.

From these results, we conclude that notifications are considerably less efficient than socket messages and service calls for small state sizes. For larger state sizes all of the three approaches impose a high overhead. Note that in all cases the requests have very low overheads. In this situation a request that used to take 44 ms ends up taking between 4 and 6.5 times as long with replication. For Grid services that have longer-running requests, the overhead of replication will be diminished. For example, for a request that takes 3 seconds to execute and has state size of 100 kB, the overhead of replication is less than 10%. Hence, the drawback of using GT3 to implement primary–backup becomes negligible for long-running requests.

3.3 Failover

Another important metric for the performance of a fault-tolerant system is failover duration. This is a sum of two quantities: the time it takes for the backup to detect the failure, and the time it takes for the backup to notify the clients of a failover. The first quantity depends on the frequency of heartbeat messages and is

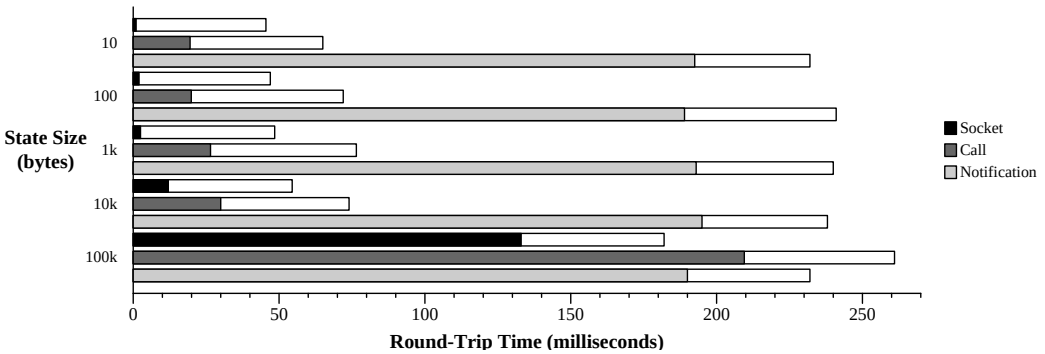

Figure 5. Median Round-trip times for state transfer using *Socket*, *Call* and *Notification* with different state sizes. The white portion is the round-trip time of a client request without replication. Therefore, the overall size of the each bar is the total client round-trip time.

Table 3. Ratio of client request round-trip with primary-backup to median/mean client round-trip without replication

		10 B	100 B	1 kB	10 kB	100 kB
Notification	Median	5.5	5.5	5.5	5.4	5.3
	Mean	4.6	4.6	4.6	5.5	4.4
Call	Median	1.5	1.6	1.7	1.7	5.9
	Mean	1.3	1.4	1.7	1.5	6.5
Socket	Median	1.0	1.1	1.1	1.3	4.1
	Mean	1.0	1.0	1.0	1.2	3.6

Table 4. Failover duration (milliseconds)

	1	2	4	8	16
Median	189	215	524	896	1989
Mean	194	302	547	1018	2269
St. Dev.	21	352	83	454	666

largely independent of the implementation. Therefore, we only measure the second quantity, as shown in Table 4.

With one client, it takes 194 ms on average to notify the client of a failure. As the number of clients increases, the notification overhead increases linearly. In [17] we report that recovering a network connection endpoint in less than 200 ms requires significant investment in equipment for logging of packets that may be lost due to the failure, and so we believe that 194 ms is quite acceptable. If the number of clients that shares the same instance is large, however, then the overhead may become too large. Again, these results were obtained based the current implementation of GT3. A later version may be able to have notifications run faster than linear in the number of sinks.

Note that if the client doesn't have outstanding requests to the primary service when the failure happens, then the overhead of the failover at client is almost zero, since the client only needs to change the address of the service invoked.

4 Conclusion

Fault tolerance of stateful Grid services is becoming increasingly important with the development and use of OGSI. Both the infrastructure services such as monitoring, resource allocation, and scheduling, as well as Grid applications implemented as Grid services, are required to be reliable and highly available. In this paper, we showed that the facilities defined in OGSI and the newly proposed WS-Notification extension to Web services [11] can be used to design a primary–backup service. While not described in this paper, this service can be easily extended to multiple backups and

to dynamically adding backups. In addition, by using slightly modified client stubs, failover can be done transparently to clients. We did need to make strong assumptions on failure detection, but they can be satisfied by existing commercial software.

We found the overhead of using GT3 implementation of the OGSI notification to be quite high. The overhead is particularly large in the cases where the state data is small or the number of clients is large. Much of the overhead seems to come from the cost of notifications, which can most likely be improved in future implementations of GT3. Failing that, one might wish to provide state update below the OGSI level or by using simpler OGSI facilities such as basic Grid service method calls. It might be possible to improve performance of primary–backup by using an alternate protocol binding—something that is specified in OGSI but not available in GT3—but we have not explored this option in any detail.

Our approach for primary–backup is only applicable for replicas located in a cluster, since otherwise failure detection becomes too unreliable for primary–backup. We are currently looking at methods that still accommodate nondeterminism, like primary–backup, but that can work in a wide-area network where asynchronism is more of an issue.

References

[1] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, Oct 1976.

[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] A. Amoroso, K. Marzullo, and A. Ricciardi. Wide-area Nile: a case study of a wide-area data-parallel application. In *Proc. 18th International Conference on Distributed Computing Systems*, pages 506–515, Amsterdam, The Netherlands, May 1998.

[4] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Parallel and Distributed Computing on Workstation Clusters and Networked-based Computing*, Jun 1997.

[5] K. Birman, T. Joseph, T. Raeuchle, and A. Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, Jun 1985.

[6] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Primary–backup protocols: Lower bounds and optimal implementations. In *Proc. 3rd IFIP Conf. on Dependable Computing for Critical Applications*, pages 187–198, Mondello, Italy, September 1992. Springer-Verlag, Wien.

[7] 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.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[10] Global Grid Forum. *Final OGSI Specification V1.0, Proposed Recommendation*, Jul 2003.

[11] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratham, J. Parikh, S. Patil, S. Samdarshi, S. Tuecke, W. Vambenepe, and B. Weihl. Web service notification (ws-notification), Jan 2004. http://www.-ibm.com/developerworks/library/ws-resource/ws-notification.pdf.

[12] S. Hwang and C. Kesselman. Grid workflow: A flexible failure handling framework for the grid. In *Proc. 13th IEEE Intl. Symp. on High Performance Distributed Computing (HPDC-13)*, pages 126–137, Seattle, Washington, USA, June 2003.

[13] M. Livny and J. Pruyne. Managing checkpoints for parallel programs. In *Proceedings of IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing*, 1996.

[14] 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.

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

[16] Veritas company homepage. http://www.veritas.com/index.html.

[17] D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. Bressoud. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 393–402, San Francisco, California, USA, June 2003.