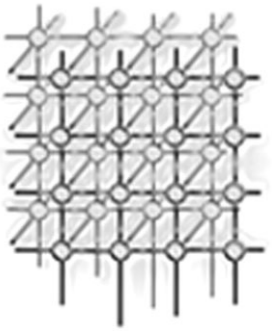
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **18**:701–724
Published online 20 October 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.957



A replication framework for program-to-program interaction across unreliable networks and its implementation in a servlet container

Alberto Bartoli*,†, Milan Prica and
Etienne Antoniutti di Muro

Dipartimento di Elettrotecnica, Elettronica, Informatica, University of Trieste, Via Valerio 10, 34100 Trieste, Italy

SUMMARY

We propose a service replication framework for unreliable networks. The service exhibits the same consistency guarantees about the order of execution of operation requests as its non-replicated implementation. Such guarantees are preserved in spite of server replica failure or network failure (either between server replicas or between a client and a server replica), and irrespective of when the failure occurs. Moreover, the service guarantees that in the case when a client sends an 'update' request multiple times, there is no risk that the request be executed multiple times. No hypotheses about the timing retransmission policy of clients are made, e.g. the very same request might even arrive at different server replicas simultaneously. All of these features make the proposed framework particularly suitable for interaction between *remote programs*, a scenario that is gaining increasing importance. We discuss a prototype implementation of our replication framework based on Tomcat, a very popular Java-based Web server. The prototype comes into two flavors: replication of HTTP client session data and replication of a counter accessed as a Web service. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: replication; axis; group communication; Tomcat; Web service; HTTP reliability

*Correspondence to: Alberto Bartoli, Dipartimento di Elettrotecnica, Elettronica, Informatica, University of Trieste, Via Valerio 10, 34100 Trieste, Italy.
†E-mail: bartolia@univ.trieste.it

Contract/grant sponsor: EEUU Project Adopt: Middleware Technologies for Adaptive and Composable Distributed Components; contract/grant number: IST-2001-37126
Contract/grant sponsor: Microsoft Research Ltd

Copyright © 2005 John Wiley & Sons, Ltd.

Received 9 October 2003
Revised 1 September 2004
Accepted 18 February 2005

1. INTRODUCTION

There has recently been increasing interest toward technologies for enabling business-to-business interactions across the Internet. Given this trend, the importance of such requirements as high availability, high throughput and scalability for Internet-based services can only grow. While such requirements could be met by *scale-up* strategies (increasing the computing power of the platform hosting the service), *scale-out* strategies based on *replication* of commercial off-the-shelf components has become a largely diffused approach, in particular, due to cost reasons. Business-to-business interactions are typically characterized by interactions between remote *programs* [1]. In this context, a crucial requirement for a replicated service is the ability to preserve the consistency guarantees expected by clients of a non-replicated service. For example, if a client issues a sequence of operation requests to a replicated service, the results should be the same as if this sequence was processed by a non-replicated service. Clearly, requirements of this kind are much more important when clients are *programs*, as opposed to when clients are *humans*. A replicated service may or may not guarantee that these requirements be satisfied, depending on the details of the replicated service implementation. For example, in many distributed Web server systems [2], it is possible that different requests from the same client be routed by different server replicas. If these replicas neither synchronize their executions nor have access to shared storage, then the result of a 'read' could not reflect the outcome of a preceding 'write'. Most commonly, the consistency guarantees of a non-replicated service are preserved in a replicated implementation only under certain hypotheses about the operating environment and request processing execution (we elaborate on these issues in the next section).

In this paper we describe a replicated service designed to address the needs of program-to-program interaction across possibly unreliable networks, i.e. networks that could temporarily partition. In particular, the service preserves the strong consistency guarantees offered by its non-replicated counterpart in spite of server and network failures, and irrespective of when such failures occur. The only kind of failures that are not addressed are byzantine failures, i.e. either server replicas or communication channels that behave arbitrarily and possibly maliciously. First, we specify the replicated service and discuss related work in this area. Then, we provide a detailed algorithm for service replication providing the desired properties. Finally, we describe a prototype implementation of this algorithm. The implementation consists of a replicated Web server based on Tomcat. The implementation will demonstrate that our proposal is indeed very simple to deploy. It basically consists of wrappers for the code and state of a non-replicated implementation.

2. OUR FRAMEWORK AND RELATED WORK

The distributed system is modeled as a collection of processes that communicate through a network and that do not share storage. Processes may crash and communication failures may partition the network. A crashed process may recover as a 'new' process and partitions are eventually repaired. The system is asynchronous in that no bounds are assumed on communication delays or relative speeds of processes. Message corruption and byzantine faults are excluded.

Commercially available products for high-availability typically assume *reliable* communication channels and a secondary memory storage that can be accessed by *all* replicas (e.g. [3,4]). In contrast, we consider a system model where messages may be lost and communication between replicas may occur only through message passing. We are interested in this model to investigate replicated services

Copyright © 2005 John Wiley & Sons, Ltd.

where: (i) clients are remote entities that access the service through a possibly unreliable wide-area network; and (ii) the server replicas could span over a building, a campus or even a geographical area, for example.

We consider a service S exporting a set of objects. We denote the object with the (unique) identifier id as obj[id]. Each object exports a read operation and an update operation. Updates are assumed to be neither idempotent nor commutative. The order in which S executes operations is defined by a *consistency criterion*. For example, *sequential consistency* requires that [5]: given any set of operations, the execution of these operations produces the same results as if these operations were executed in some sequential order; and, the operations from each client appear in this sequence in the same order in which they were issued by the client. This criterion is equivalent to the *serializability* consistency criterion used in database systems when every transaction consists of a single operation [6,7]. Another consistency criterion is *linearizability* [8]. With this criterion, the order in which operations appear to have executed must preserve the ordering of non-overlapping operations issued by different clients. Our replication framework uses linearizability as a consistency criterion. Our prototype implementation, on the other hand, uses sequential consistency.

We consider a *replicated* implementation of S, denoted S^R, that consists of a set of processes (server replicas, briefly *replicas*). Replication is used to increase the service availability and throughput. S^R must satisfy the following properties

Property P1. S^R guarantees the same consistency criterion as S.

Property P2. If S^R receives multiple copies of a given update request, S^R executes the operation only once. Moreover, S^R responds to each copy of the request with the result of only that execution.

For example, in an e-commerce setting, Property P1 states that if a customer places an order and at a later time decides to cancel it, the second request must see the pending order in the customer's account. Achieving this property with a replicated service may not be obvious, in particular, when the two requests are routed to different replicas. Property P2 states that if a customer issues a 'buy' order for X shares of a certain stock and, perhaps due to the apparent unresponsiveness of the service, the customer reissues the very same order again, the customer must not end up holding $2X$ shares of the stock. Note that this second scenario is far from being artificial. When a Web site appears unresponsive, users often repeat their last request several times. Indeed, a recent study conducted over several months has shown that when a Web site appears to be unresponsive, simply retrying the same request has a high probability of obtaining a response [9]. This observation encourages users to retry requests even after relatively short delays in obtaining a response.

Property P2 is called 'transparent reinvocation' in the Fault-Tolerant CORBA (FT-CORBA) standard [10–12]. Replicated service implementations complying with the FT-CORBA specification are perhaps those providing the strongest guarantees to date (e.g. http://www.eternal-systems.com). However, the specification does *not* address network partitioning faults. Instead we require that this property be satisfied in an environment where the network, either between replicas or between clients and replicas, may fail. Properties P1 and P2 are similar to the 'exactly-once' requirement for a replicated service considered in [13], except that the cited paper considers a three-tier architecture where the second tier is replicated and all replicas share a back-end database (third tier). In our model, replicas do not share any storage and may only communicate through message passing.

In our approach, server replicas belong to a *group* and have access to a *group communication* (GC) middleware (*GC-layer*) [14]. A detailed description of GC is beyond the scope of this paper and in

Copyright © 2005 John Wiley & Sons, Ltd.

the next section we only provide the necessary background. GC was proposed several years ago as a technique for implementing replicated services with strong reliability requirements [15] and since then it has become a well-established framework [14,16]. An introductory treatment of GC can be found, for example, in [17]. Use of GC as a tool for implementing fault-tolerant distributed objects enforcing linearizability was first proposed in [18]. Numerous distributed object systems based on GC have been implemented since then, initially based on CORBA (e.g. [19–21]) and then on FT-CORBA (e.g. [11,12]). A critical analysis of such research can be found in [22,23]. Our proposal follows all these prior efforts and builds upon them.

Use of GC as a tool for implementing replicated databases ensuring Property P1 (serializability) has been proposed in [24–29]. However, these works are not concerned with service access from remote clients and (implicitly) assume that each client is co-located with one replica. Therefore, such works are not concerned with Property P2.

A work close to ours is that of Karamanolis and Magee who study in detail the interaction of clients with a replicated service based on GC [30]. The system model is the same (in particular, servers do not share any storage and the network is unreliable) and they require guarantees analogous from Property P1 to the service. However, they do *not* require Property P2 either: the server-side of their replication protocol does not detect duplicate requests and defers their handling to an upper-level protocol. We consider instead Property P2 as an integral part of our design, because the upper-level protocol is far from being trivial and because it is important to give clients a simple way to (eventually) learn the outcome of an update operation for which they received no response. Simply retransmitting the *very same* operation request appears to be a promising strategy, in particular, in environments where clients are programs as opposed to when they are humans. Indeed, we do not make any assumption about the 'timing policy' of client retransmissions: we do not exclude that the very same update request arrives more or less simultaneously at multiple server replicas.

The experiments performed by Karamanolis and Magee confirmed the superior performance and scalability of a structuring where clients are not group members [16]; that is the approach we have taken: clients do not run any GC protocol. Karamanolis and Magee considered two algorithms for this structuring: one in which clients are *replication-aware* and one in which they are not. In the former algorithm, whenever the group of server replicas experiences a membership change *all* clients have to be involved in the internal reconfiguration of the service. In our approach clients are not involved in the service reconfiguration, which is clearly desirable in an Internet-based service, for example. In the non-replication-aware algorithm, a client multicasts *each* request to *all* server replicas, whereas in our protocol a client communicates with only one server replica at a time. We believe that our approach is more suitable for integration with existing technology and for use by geographically-dispersed clients. Finally, Karamanolis and Magee do not consider the problem of reintegrating failed server replicas in the service after they recover.

Our algorithm is such that the service stops being available when there is no majority of replicas that are active and mutually connected. That is, when an excessive number of failures occur, the service stops responding to requests because it is no longer able to guarantee Properties P1 and P2 (in practice, the service will have to reboot as a new incarnation). While this feature is not peculiar of our approach (e.g. [13]), it is worth recalling that many replicated services take quite a different approach, in which certain failures may cause the system to *silently* stop guaranteeing Property P1 and/or P2—the system continues to respond but its behavior no longer satisfies either Property P1 or P2, without any explicit notification of this fact. As an example, consider the *session failover* support

Copyright © 2005 John Wiley & Sons, Ltd.

in replicated servers based on IBM WebSphere [31]: the conversational state for a client (i.e. am HTTP 'session') is kept in a database shared by all replicas, so that it remains available in case the replica connected with client fails; since session information is accessed very frequently, a caching mechanism is used to decrease the overhead related to database access; this mechanism is such that, should a failure occur within a certain 'vulnerability window', some updates already seen by the client could be (silently) lost[‡]. As another example, consider some recent implementations of *distributed data structures* (*DDS*) [32]. A DDS is an object (e.g. a hash table) partitioned and replicated across several replicas. The service implementation, which ensures excellent performance and scalability, is based on the assumption that the network between replicas never partitions. Should such an event occur, the service could silently stop satisfying its consistency criterion. While the above design choices are sensible in many environments, in particular where state-of-the-art performance and scalability are essential, we are interested in exploring other design trade-offs, more suitable for application domains where it may be preferable to *eliminate* the potential for 'inconsistencies' even at some cost in terms of performance and scalability. Indeed, many environments do not need state-of-the-art performance and scalability [33] and leading research groups believe that 'it is time to broaden our performance-dominated research agenda' [34].

3. REPLICATED SERVICE IMPLEMENTATION

Our framework is a form of *primary-backup*, *passive* replication scheme. Different clients may be, and usually are, associated with different primaries. Associating different clients with different primaries may be useful for load balancing (see also Section 5.2). For simplicity and without loss of generality we make the following assumptions. First, the set of objects is defined statically (our prototype supports dynamic object creation though, see Section 4). Second, each replica has its own IP address and this address is public, i.e. meaningful to clients (see also Section 3.1.1).

3.1. Client side

Each client has a system-wide unique identifier, denoted clientId, and a serviceHandle table with one element for each object. Initially, all elements of this table are zero. We denote by serviceHandle[objId] the element associated with object objId (note that we use square brackets to indicate elements of a keyed table, not elements in an array). The serviceHandle is opaque to clients. As clarified in the next section, S^R interprets the value of serviceHandle[objId] as the number of updates that this client has applied to objId.

A client C obtains the network address of one or more server replicas from an external naming service and then connects to one server replica, say R. The interaction follows a request–response pattern. After sending a request, C does not send further requests until receiving a response.

[‡]It may be useful to mention that the numerous tools for building distributed Web servers do not usually provide either Property P1 or P2, as most such tools do not provide any session failover capability (a deep survey of such tools can be found in [2]).

Copyright © 2005 John Wiley & Sons, Ltd.

Each request is composed of the following fields: (i) method, an enumerated whose value can be either 'read' or 'update'; (ii) params, the input parameters of the method; (iii) objId, the identifier of the object to which the method is to be applied; (iv) clientId; and (v) serviceHandle[objId].

The response from S^R is composed of the following fields: (i) status, an enumerated whose value can be one of OK, Unable, Unknown, as explained below; (ii) result, the output parameters of the method invoked; (iii) newHandle, the new value that C must assign to serviceHandle[objId]; and (iv) alternatives, the network address of some of the replicas that implement S^R. The only field present in all responses is status, each of the other fields may or may not be present. The meaning of the status field is as follows.

- OK: S^R has executed the request.
- Unable: R is not able to execute the request. This code is returned either when R is not 'sufficiently up-to-date' with respect to the pair clientId, objId specified in the request, or when the request is an update and R is currently not able to execute updates (both cases will be clarified in the next section).
- Unknown: R cannot tell whether S^R has indeed executed the update request (this status value may only be returned for an update request). When C receives an Unknown response, we say that C has a *pending update*. A connection that breaks while C is waiting for the response to an update request is the same as an Unknown response.

When the status of response is not OK, C immediately sends a close request and closes the connection. At this point, C may (try to) open another connection immediately, with either the same replica R or another one. If C has a pending update, the first request that C must submit after opening a new connection is this very same pending request. If the response is OK, then C clears the pending update and may submit other requests. Otherwise, C sends a close request, closes the connection and then may (try to) contact another replica.

If C sends a message m to another client C′ of S^R, then *m* includes a copy of the serviceHandle. When C′ receives *m*, C′ updates its copy of the serviceHandle as follows: ∀objId, serviceHandle[objId] ← max(serviceHandle[objId], m.serviceHandle[objId]). These operations are not required when either of the following holds: different clients do not share objects; or, the consistency criterion enforced by S^R places no constraint on the ordering of operations issued by different clients (the previously mentioned sequential consistency criterion is an example).

3.1.1. On redirection messages

Our assumption that each replica has its own public IP address does not include all possible environments. It is also possible that: (i) a special device is interposed between clients and replicas (e.g. a Network Address Translator (NAT)-enabled router), clients only know the public IP address of this device, or each replica has a *private* IP address; or (ii) all replicas share the same IP address.

This issue has important implications on *redirection messages*, i.e. responses including an alternatives field. In case (i), redirection messages have to be followed by the device, transparently to clients. This functionality is rather commonplace for several protocols [2]. In case (ii), on the other hand, redirection messages are not meaningful. Indeed, as argued in [2], the inability to exploit any form of *content-aware dispatching* is a limitation of architecture (ii). We do not mention these architectural issues any further for brevity.

Copyright © 2005 John Wiley & Sons, Ltd.

3.2. Server side

3.2.1. Group communication

Each replica is equipped with a GC middleware (GC-layer) and the replication algorithm on the server side is built above this GC-layer. We provide only the necessary background about the GC-layer and refer to the literature on this topic for a more rigorous description (e.g. [14]). A process joins the group with the grp.Join() operation (we assume there is only one group). A process leaves the group either explicitly, with the grp.Leave() operation, or implicitly, by failing. The grp.GetEvent() operation returns either a message or a *view change*. A *view* is a set of process identifiers corresponding to the current group membership. A view change $vchg(v)$ notifies the receiving process that the view has changed and has become v. The GC-layer determines a new view not only as a result of explicit join and leave operations, but also as a result of crashes, recoveries, network partitions and mergers. We consider a GC-layer supporting a *primary-partition* membership, i.e. one in which all group members have the same perception of the group membership. For example, in the case when the set of replicas splits into two or more disjoint sets because of a network failure, then the GC-layer *automatically* selects one of these sets to be the next view and forcibly expels from the group the replicas that happen to be on the wrong side of the partition (by delivering to these replicas a special 'shutdown' view change)[§]. The GC-layer ensures that the perception of the group membership evolution at the various replicas is 'consistent': all processes in v receive $vchg(v)$ and view changes are received in the same order at all processes (see [14] for a more rigorous description). Our prototype selects the primary partition as the one containing a majority of the (statically known) number of replicas.

Group members communicate by means of grp.Mcast(m), which sends message m through a *totally-ordered* multicast: if replicas p and q receive messages $m1$ and $m2$, then p and q receive these messages in the same order, i.e. either they both receive $m1$ before $m2$ or they both receive $m2$ before $m1$.

The deliveries of multicasts and view changes guarantee *uniform delivery* and *virtual synchrony*, as follows. We say that p *delivers view* v if p receives the associated view change $vchg(v)$. We say that p *delivers* m *in* v if the last view delivered by p before m is v.

- Let p, q deliver v. If p delivers m in v, then: (i) q also delivers m, unless q crashes or leaves the primary partition; and (ii) if q delivers m, it does so in v.

These properties are very powerful for programming algorithms that have to cope with failures and recoveries. As a key example, consider a replica p that delivers view v. Suppose that p crashes while multicasting m and, as a result, view w is delivered. It is guaranteed that either: (i) *all* replicas that deliver v and w receive m (and do so *before* receiving w); or (ii) *none* of them receives m.

[§]Our prototype, described in Section 4, is actually based on a *partitionable* membership layer: replicas that end up in the 'wrong partition' remains group members but can execute only read operations. In this paper we assume a primary-partition membership layer because it greatly simplifies the presentation of the group communication issues.

Copyright © 2005 John Wiley & Sons, Ltd.

```
// The actual object replicas
obj:            set of objects initial class-specific values;
// Number of updates applied on each object (version number)
versionTable:   table of integer indexed by objId initial 0 for each element;
// For each < clientId, objId >, last version number of objId updated by clientId and corresponding result (see also the text)
clientTable:    table of table of (integer, resultType) indexed by clientId, objId initial empty;
// True only when the server can execute updates
canUpdate:      boolean initial false;
// The last received view
currV:          view initial null;
```

Figure 1. Data structures of a server replica.

3.2.2. *Overview of the algorithm*

The internal state of S^R consists of data structures kept in volatile storage (Figure 1). These data are replicated and kept in sync at all servers in the primary partition. In particular, clientTable[clientId][objId] is a pair n, result: the former is the value of versionTable[objId] corresponding to the last update executed by clientId on objId; the latter is the result that was returned to the client[¶].

Each client C is connected with one single replica at a time, say R. R can execute a request submitted by C only if the request identifies an object for which R is 'sufficiently up-to-date'. Informally, this means that R holds a version of the object that is more recent than the latest version accessed by C (see the next section for full details). If R is not able to execute the request, R redirects C to another replica. R executes read operations locally, without any interaction with the other replicas. R can execute update operations only if it belongs to the primary partition of the group of the replicas. Executing an update involves multicasting the new version of the object and the response to be sent to C. A client that does not receive the response to an update, or that observes the connection breaks before receiving such a response, may submit the very same request immediately, to either the same replica or another one. There is no hypothesis whatsoever on the retransmission policy used by clients.

Replicas that enter the primary partition (i.e. those that recover after a failure or that were isolated because of a network failure that has repaired) can execute updates only after acquiring the up-to-date version of the data in Figure 1, which is done by means of a procedure called *state transfer*.

3.2.3. *Algorithm*

Each replica has to handle multiple, independent flows of events: requests from clients, each client being associated with a dedicated connection; connection requests from new client; and messages and view changes from the GC-layer. In practice, coordinating these flows may be a significant source

[¶]We have not detailed the actions for garbage-collecting entries in the clientTable. In practice, an entry will be garbage-collected when it has not been referenced for a 'sufficiently long' time. This is the only realistic possibility as there is no way of 'proving' that the client associated with an entry declared as garbage will show up later; of course, one could make additional assumptions on the client-service protocol to make this event less likely, but the problem cannot be eliminated.

Copyright © 2005 John Wiley & Sons, Ltd.

of complexity. We paid special attention to this issue, which we solved by devising two novel calls to be exported by the GC-layer and by a carefully designed threading architecture. These novel calls are localCall() and localRespond(), which are used for inter-thread communication as follows: resp ← grp.LocalCall(msg) sends msg to the executing process and blocks the invoking thread T1; msg is inserted in the flow of events from the GC layer and will be received by another thread T2, through grp.getEvent(). Thread T1 unblocks when T2 executes grp.LocalRespond(msg,resp).

The detailed algorithm executed by each replica is as follows. When the replica bootstraps, it spawns a MainT() thread, discussed later, and a thread for listening to connection requests from clients. When a new connection is established, a ClientSessionT() thread is associated with the connection. This thread could have the same lifetime as the connection, or it could be allocated from a pool when the connection is created and returned to the pool when the connection is closed (Section 4).

A ClientSessionT() processes each request by executing the code in Figure 2. It first checks whether the replica is 'sufficiently up-to-date' for the specified object. If not, ClientSessionT() responds to the client by declaring that the replica is Unable to execute the request. Then, if the request is a read, ClientSessionT() executes the request and responds. If the request is an update, ClientSessionT() first inspects the flag canUpdate. This flag is managed by the MainT() thread so that it is set only when the replica is in the primary partition and the state transfer to that replica has completed. If canUpdate is not set, ClientSessionT() responds to the client by declaring that the replica is Unable to execute the request. If canUpdate is set, ClientSessionT() inspects whether the update request is a duplicate or is fresh. If it is a duplicate, ClientSessionT() immediately sends the result to the client, without carrying out the update again. If it is fresh: (i) ClientSessionT() executes the update on a *copy* of the object; (ii) ClientSessionT() passes the new version of the object and the result of the update to MainT(), through a grp.LocalCall(); (iii) MainT() propagates the new version and associated result to the other replicas, through a grp.Multicast(); (iv) when ClientSessionT() completes the grp.LocalCall() (i.e. when MainT() has responded), ClientSessionT() sends the result to the client. Note, a ClientSessionT() thread only interacts with the GC-layer through grp.localCall().

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

The MainT() thread is shown in Figure 3. Understanding of the code is simplified by assuming that all replicas are in the primary partition, no update request has been submitted to the service yet, canUpdate is true at all replicas. Upon receiving a message m from a ClientSessionT(), MainT() saves m in localQ and multicasts m (the localQ acts as a list of threads that are waiting for a response from MainT(), see the handling of view changes below). Upon receiving a multicast m, MainT() at each replica repeats the freshness check (a client might send different copies of the same request to different replicas very quickly; we make no assumption on the timing of the client retransmission policy). If the request is a duplicate, each MainT() ignores the request. If the request is fresh, each MainT() updates the object and all necessary data structures. The MainT() at the replica that sent the multicast also executes grp.localRespond() in order to unblock the ClientSessionT() that will actually respond to the client.

Copyright © 2005 John Wiley & Sons, Ltd.

```
1.  Procedure handleRequest(RequestFromClient req);
2.      hdl = req.serviceHandle;
3.      if (hdl = null) then
4.              Add entry in clientTable for req.clientId;                  // This is a new client;
5.              ∀objId, clientTable[req.clientId][objId].n ← 0;
6.      else-if (hdl[req.objId] > versionTable[req.objId]) then
7.              sendResponse (<Unable, Redirect to another replica >)  // Replica not sufficiently up-to-date
8.  end-if
9.  multi-if
10.             ♦ req is read ⇒ result ← obj[req.objId].Read(req.params);
11.                             sendResponse (<Ok, result >)
12.             ♦ req is close ⇒ no-op;
13.             ♦ req is update ⇒ clientSlot ← clientTable[req.clientId][req.objId];
14.                             multi-if
15.                                 ♦ hdl = clientSlot.n −1 ⇒                  // Duplicate of last request
16.                                     sendResponse (<Ok, clientSlot.result >);
17.                                 ♦ hdl = null or hdl[req.objId] = clientSlot.n ⇒   // Fresh request
18.                                 if not canUpdate then
19.                                     sendResponse (<Unable, Redirect to another replica >);
20.                                     else    objTmp ← obj[objId].Clone();
21.                                             result ← objTmp.Update(req.params);
22.                                             resp ← grp.LocalCall(< req, objTmp, result >);
23.                                             sendResponse(resp);                          (X)
24.                                 end-multi-if
25. end multi-if
```

Figure 2. Request handling logic.

Finally, upon receiving a view change, MainT() checks whether a replica has just entered the primary partition (the case oldV=null is when it is the executing replica that has just entered the primary partition). In this case, the MainT() at all replicas in the primary partition execute the function ViewExpansion(). We do not discuss this function in detail for brevity (full details can be found in [35]). Its specification is as follows.

- ViewExpansion() returned true: it has selected the most recent state among the group members and updated the variables of Figure 1 if necessary. In other words, this function implements state transfer.
- ViewExpansion() returned false: it was not possible to determine the most recent group state and the variables of Figure 1 remained unchanged. It is not possible to provide update functionality. This is the case when the group reforms after the primary partition ceased to exist and at least one server replica has crashed since the last existence of the primary partition. Preventing the occurrence of such executions requires more complex solutions that are beyond the scope of this paper. Such solutions require that each server replica saves (part of) its state on stable storage.

MainT() ignores multicasts received within ViewExpansion(). Thus, upon return, messages queued in localQ are to be handled as appropriate, depending on the returned value.

We do not provide a detailed correctness proof. Very briefly, we argue that the replication algorithm satisfies Property P1 because: updates are totally ordered and system-wide; the use and management of ServiceHandles and of the versionTable ensure that: (i) requests from a given client are executed in the order in which they were issued; and (ii) this order is consistent with the global ordering of requests

Copyright © 2005 John Wiley & Sons, Ltd.

```
1.  Local variables:        localQ: queue of messages initial empty; oldV: view initial null;
2.  Thread MainT()
3.    Init variables in Figure 1;
4.    grp.JoinGroup();
5.    repeat
6.      e := grp.GetEvent();
7.      multi-if
8.        ♦ e is message m from ClientSession thread ⇒
9.            if canUpdate then { localQ.Insert(m); grp.Mcast(m); } else grp.LocalRespond ((<Unable, m>);
10.       ♦ e is multicast m ⇒
11.           if canUpdate then
12.               clientSlot ← clientTable[req.clientId];
13.               hdl ← req.serviceHandle;
14.               multi-if
15.                ♦ hdl = null or hdl[req.objId] = clientSlot[req.objId].n) ⇒      // Fresh update request
16.                     obj[m.objId] ← m.objTmp;
17.                     clientSlot[m.objId].result ← m.result;
18.                     clientSlot[m.objId].n ++;
19.                     versionTable[m.objId] ++;
20.                     hdl[m.objId] ++;
21.                ♦ hdl[objId] < clientSlot[objId].n ⇒ no-op;                      // Duplicate update request
22.               end-multi-if
23.               if m.sender = myId then
24.                     response ← < OK, hdl, m.result >;);
25.                     grp.LocalRespond(m, response);                             // Awaken ClientSession thread
26.                     localQ.Drop(m);
27.               end-if
28.           elseif m.sender = myId then
29.                 { grp.LocalRespond(m, <Unable>;); localQ.Drop(m); }            // Awaken ClientSession thread
30.           end-if
31.       ♦ e is view change vchg(v) ⇒
32.           oldV ← currV; currV ← v;
33.           if oldV = null or (∃p : p ∈ currV and p ∉ oldV) then canUpdate ← ViewExpansion();
34.           if canUpdate then     ∀m ∈ localQ grp. Mcast(m);
35.           else                  ∀m ∈ localQ { grp.LocalRespond(m, <Unknown>); localQ.Drop(m); }
36.           if currV = "shutdown" then
37.                { oldV ← null; grp.JoinGroup();}                                // No longer in the primary partition
38.     end-multi-if
39. forever
```

Figure 3. The Main thread.

from different clients (so as to satisfy linearizability). We argue that the algorithm satisfies Property P2 because updates are totally-ordered and system-wide, and because of the use and management of the clientTable.

3.2.4. On the use of object cloning upon updates

The proposed algorithm is such that a ClientSessionT() thread executes each update on a copy of the object. This is the approach taken in our implementation, but other alternatives are possible that do not require object cloning upon updates.

Copyright © 2005 John Wiley & Sons, Ltd.

- When MainT() returns the Unknown status, it marks the corresponding object objId as 'dirty' (recall that MainT() returns Unknown only when the replica is expelled from the primary partition while multicasting an update). Requests for 'dirty' objects provoke the Unable response. The 'dirty' flag for objId will be cleared when the network failure is repaired and the replica has acquired the correct state of objId through state transfer. We intend to explore this approach in the future as it is very efficient.
- ClientSessionT() does not execute update requests and forwards them to MainT(). MainT() multicasts each update request and will execute the update request only upon receiving this multicast. In this case, if MainT() receives a 'shutdown' view change before receiving the message, the local copy of the object will remain unchanged. Note that with this approach *all* replicas will execute the update request—an *active* replication scheme. Note also that the multicast will not contain the serialized contents of objId. While this approach avoids object cloning, it requires that all replicas be completely deterministic in the processing of each update request—quite a strong assumption that may be difficult to make true in practice [16]. We prefer not to rely on this assumption.

3.2.5. Load balancing

Since each replica executes reads without any interaction with the other replicas, fairly distributing connections from clients to replicas may significantly improve performance, in particular for workloads where reads are much more frequent than updates. The GC-layer we used for implementing our replication algorithm incorporates a novel mechanism that we have developed for propagating a *load index* amongst replicas quickly and efficiently. Based on this mechanism, described below, many different load-balancing policies can be implemented simply. We show one such policy for illustration purposes only, as load balancing is beyond the scope of this paper.

Each group member has access to a grp.WhiteBoard object that it can read and write. The collection of these objects is meant to simulate a 'whiteboard' shared among the group members. Each grp.WhiteBoard instance is a table of *integers*, with one element corresponding to each member. The call grp.WhiteBoard.Write(val) sets the element of the table corresponding to the invoking member to val. The call grp.WhiteBoard.Read(p) returns the value of the table element associated with group member p.

Writes are *not* propagated with virtual synchrony semantics: if, for example, p applies a sequence X of writes to grp.WhiteBoard while in view v, then different members of v could observe different subsequences of X. Causal precedence relationships between writes to the grp.WhiteBoard and multicast transmissions may not be preserved either: if, for example, p issues grp.WhiteBoard.Write(val) and then multicasts m, then different processes could observe the write and the delivery of m in different orders.

The semantics associated with grp.WhiteBoard have been purposefully kept weak in order to admit cheap implementations. In our implementation, the table is replicated at each group member. Reads are performed on the local copy, whereas writes are performed on the local copy and then propagated to other copies by piggybacking the new value to messages that the GC-layer has to exchange anyway (multicasts, view changes). It follows that writes propagate at basically no cost and, typically, within a few hundred milliseconds.

Copyright © 2005 John Wiley & Sons, Ltd.

```
1.  Procedure acceptRequest (RequestFromClient req);
2.      if (grp.Whiteboard.Read(myId) < threshold) then
3.              loadIndex ++;
4.              grp.Whiteboard.Write(loadIndex);
5.              proceed in handling the request req;
6.      else    trySet ← {p: p ∈ currV and grp.Whiteboard.Read(p) < threshold };
7.              do not handle req and respond with a redirection message to any in trySet;
8.      end-if
```

Figure 4. Load-balancing logic.

As a simple example, we consider the number of *in-progress requests* as the load index. Each ClientSessionT() increments the load index when it starts processing a request and decrements the load index when the processing has completed. Changes to the load index occur through grp.WhiteBoard.Write() (the corresponding code is not shown for brevity). Before incrementing the load index, though, ClientSessionT() decides whether to accept or refuse the request based on a threshold-based load-balancing policy [36] (Figure 4). Refused requests are redirected to other replicas, selected based on the load index. We remark again that updates to the whiteboard need *not* trigger additional multicasts: the whiteboard value is automatically piggybacked into messages that the GC-layer has to exchange anyway.

Of course, one could define many different policies based on other load indices—e.g. average CPU load, number of open connections, number of requests processed in the last k seconds, k being a configurable parameter. The nature of the load index is irrelevant to the whiteboard mechanism.

4. PROTOTYPE IMPLEMENTATION

In this section we describe how we put the abstract framework of Section 3 into practice. We developed a prototype based on Apache Tomcat, a very popular HTTP server complying with the Java servlet specifications for generating dynamic Web content. The prototype comes into two flavors: replication of HTTP client session data; and replication of a counter accessed as a Web service, through SOAP over HTTP. The former may use any HTTP browser as a client, while the latter requires a Web service client program.

Each server runs an instance of Tomcat and a Spread daemon (Spread is an open-source group communication toolkit; see http://www.spread.org). Each server is also equipped with two Java packages developed by us: JBora, which implements our group communication interface (Section 3.2.1) on top of Spread; JMiramare, which implements the replication algorithm on top of JBora‖. JMiramare is for the most part Tomcat-independent, i.e. it may be executed unchanged in other

‖Bora and Miramare are terms denoting features that are ‘unique’ of Trieste. The former is the name of a very strong and cold wind that is typical of Trieste, the latter is the name of a famous castle in Trieste.

Copyright © 2005 John Wiley & Sons, Ltd.

servlet containers. There is only a small Tomcat-specific portion, approximately 350 lines, discussed below.

Clearly, if the replica connected to a client C fails, the service may still be available to C only if C knows the IP address of other replicas. This is achieved simply in our Web service client program: it suffices to include the list of all replicas in the client program simply. With browser clients, one can include suitable links in each HTML page returned by the service, so that the user can simply try different buttons until it receives a response.

4.1. A replicated Web server

We assume that the reader has some familiarity with the servlet framework (see http://java.sun.com/products/servlet). In this framework, a HTTP request is processed by a servlet, i.e. a piece of code identified by the URL specified in the request. Session management occurs with a HttpSession object. The servlet container, i.e. Tomcat, associates a HttpSession with each client and ensures that the servlet operates on the HttpSession associated with the client that sent the request. A HttpSession is a sort of keyed table whose most significant methods are: setAttribute(String, Object), which binds the specified object to the HttpSession and associates the object with the specified string; and getAttribute(String), which returns a reference to the object bound with the specified string in the HttpSession. Our prototype does not support the sharing of objects among different clients. That is, an object cannot be bound to two or more HttpSession objects.

We implemented each operation of S^R as a separate servlet. Each service operation is thus identified by a URL. At configuration time, the application designer categorizes each servlet as being either a read operation or as an update operation, as explained later. The replication infrastructure is fully transparent to the programmer, except that information returned by the HttpSession method getLastAccessedTime() is meaningless (this is because read operations of S^R are executed by each replica locally, without interacting with the other replicas).

Categorization of servlets is done as follows. We have categorized HttpSession methods as either *read methods* or *write methods*. A read method is one whose execution can have no influence on later method executions on the same HttpSession. For example, getAttribute() is a read method, whereas setAttribute() is a write method. A servlet is categorized as a read operation if and only if none of the following events can occur: (i) creation of a HttpSession object; (ii) execution of a write method on a HttpSession; (iii) change in the internal state of an object bound with a HttpSession. As for point (iii), the issue is that a servlet could obtain a reference to some object x bound to a HttpSession, by applying a read method to the HttpSession, and then modify the internal state of x. When the servlet logic may allow this event the servlet clearly has to be categorized as a write operation. Of course, if the specification of x does not allow the application designer to assess point (iii) with certainty, then the only safe choice is to categorize the servlet as a write operation.

We note that other replication frameworks for Tomcat consider as 'operations' the individual methods on the HttpSession object (e.g. [37]). Such an approach makes it impossible to guarantee the desired consistency properties to clients. A server replica might fail while executing a write servlet and after performing at least one write method on the HttpSession. Re-executing this servlet at another replica could clearly lead to inconsistencies, because the HttpSession object would reflect the operations performed prior to the failure. Instead, we consider servlets as operations, because this

Copyright © 2005 John Wiley & Sons, Ltd.

is the unit of work for users, as well as for clients of Web services, for example. Therefore, we require atomicity of servlet executions as opposed to atomicity of executions of HttpSession methods.

Finally, we remark that operations issued by different clients access different objects—each client has its own HttpSession object and, as pointed out above, we assume that no object can be bound to multiple HttpSession objects. It follows that there are no ordering constraints between operations issued by different clients. Thus, our prototype enforces sequential consistency, it does not enforce linearizability.

4.2. Implementation

Each client is associated with one object (the HttpSession) private of that client. This fact has several consequences on the data structures maintained at each replica (see Figure 1): (i) the serviceHandle at each client is only the element associated with that client, i.e. it is not an array with one element for each client; (ii) a clientId may act as an objId; (iii) the clientTable is an array with only one dimension, i.e. clientTable[clientId] contains the number of updates executed by clientId and the response returned upon the last update; (iv) the versionTable is not needed (versionTable[objId] is always the same as clientTable[objId].n). Moreover, it follows clearly from (i) that messages exchanged between clients need not include a copy of the serviceHandle (see the end of Section 3.1).

Thread MainT() is started when Tomcat bootstraps. The logic for ClientSessionT() is implemented by means of *filters*, a standard feature of the servlet framework. A filter is a piece of code that intercepts requests and responses. The chain of filters to be associated with a given URL is specified at configuration time. When Tomcat receives a request, it allocates a thread from a pool to process the request. This thread, which plays the role of ClientSessionT(), executes the chain of filters associated with the requested URL, then the corresponding servlet and finally the same chain of filters, in reversed order.

We developed three filters (Figure 5). The *load-balancing filter* implements the load-balancing logic in Figure 4. The *consistency filter* implements the logic of Figure 2 up to line 20. Line 21 corresponds to the servlet. The *update filter* implements the remaining lines. The information multicast across all replicas consists of the HttpSession object and the response computed by the servlet. The clone() operation at line 20 is implemented simply, by deserializing the HttpSession and passing the resulting object to the servlet (we keep HttpSessions stored in serialized form).

We use the session identifier contained in HTTP requests as clientId $\equiv$ objId. The ClientSessionT() that executes the first update request submitted by a client C creates a HttpSession in the consistency filter, then executes the update (servlet) and finally passes the session identifier to the MainT() in the update filter. Upon receiving the corresponding multicast, the MainT() at each replica creates a HttpSession and associates it with that session identifier. If the sending MainT() does not receive the multicast, it drops the HttpSession. This procedure implements dynamic creation of objects, not discussed in Section 3 for simplicity. All replicas of the HttpSession are associated with the same identifier, as necessary. We had to extend the Tomcat session management code, because this code does not allow users of a HttpSession to choose the identifier to be associated with that HttpSession.

The client-access protocol requires that: each client maintains a counter, called serviceHandle, of the updates executed by that client; each request includes the current value of the serviceHandle; and each response includes the new value for the serviceHandle. The only way to make sure that

Copyright © 2005 John Wiley & Sons, Ltd.

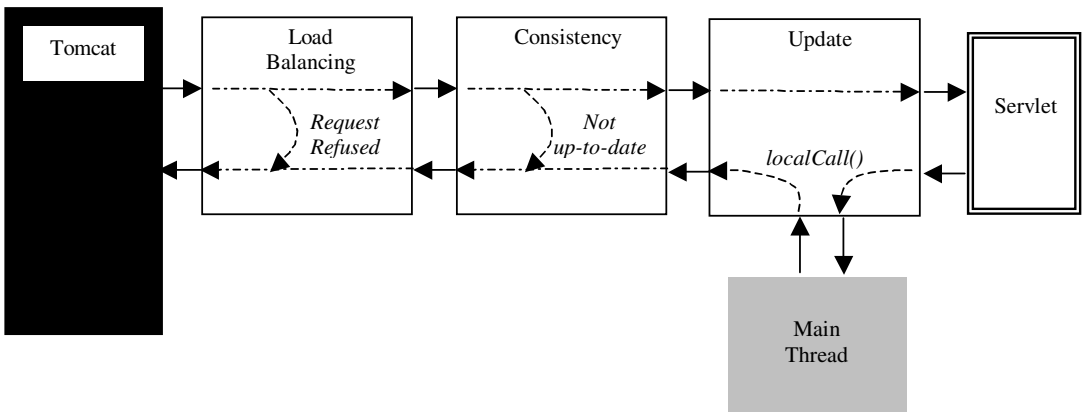


Figure 5. Execution path of a HTTP request. Each servlet is associated with a URL and each URL is associated with a filter chain. The update filter is not present in URLs categorized as read operations. The update filter operates only on responses, the other filters operate only on requests.

an ordinary HTTP browser fulfills these requirements is by storing the serviceHandle in a dedicated *cookie*. This issue has introduced several practical problems as explained below.

The server producing the response must extract the cookie from the request, determine the new value of the counter and insert in the response a cookie with the new value. The new value will be the old value incremented by one, if the request is an update and the update has been successful. Otherwise, the new value will be the same as the old value. Insertion of the cookie in the response must be accomplished transparently to the servlet code, i.e. by a filter.

If the cookie is inserted *before* executing the servlet, then the cookie might have to be replaced—in the case when the update fails, the counter must not be incremented. However, failure of an update may be discovered only by MainT(), i.e. after complete execution of the servlet. Unfortunately, the servlet specifications dictate that when the entire HTTP response has been constructed, operating on cookies for that response is no longer allowed. It follows that this option is not feasible. If the cookie is inserted *after* executing the servlet, on the other hand, the very same problem has to be faced: at this point the servlet has already constructed the entire HTTP response, thus adding a cookie is no longer possible.

We solved this problem as follows. The first filter in the chain receives from Tomcat an object R_1 that represents the response to be sent to the client. This object, created and implemented by Tomcat, usually flows along all filters in the chain up to the servlet. Let F_L denote the last filter in the chain, i.e. the one that actually invokes the servlet. In our case, F_L does *not* pass R_1 to the servlet: the servlet operates instead on an object R_2 created by F_L and implemented by us (of course, R_1 and R_2 offer the same standard interface, called HttpServletResponse). When the servlet execution has completed and MainT() has actually executed the update, the value to be inserted in the cookie can be committed. This is done by our update filter, which: (i) inserts the cookie into the response R_2 constructed by the servlet; and (ii) transfers R_2 into R_1. Implementing our own HttpServletResponse has also permitted us to serialize R_2, which is necessary for multicasting it to the other replicas, but is not possible with R_1.

Copyright © 2005 John Wiley & Sons, Ltd.

Finally, we have implemented two versions of state transfer. We do not discuss this issue in detail for brevity. The first 'naive' version is such that each replica in the primary partition multicasts all of its state whenever there is a new replica in the primary partition (so as to enable the new replica to pick the state of any up-to-date replica). The second, much more efficient version is based on an algorithm proposed in [38] in the context of replicated databases. State transfer occurs through TCP and involves only two replicas: the one that has to be brought up-to-date and one replica that was already in the primary partition. It occurs *on-line*, i.e. state transfer occurs for the most part in the background, perhaps transferring portions of state multiple times when they are updated after having been transferred. The service is suspended only for a very short time (i.e. when the state left is very small). This option is particularly desirable when the state is large (e.g. tens of MB) and the workload is high. Moreover, state transfer is *negotiated*, i.e. only the state that is really necessary to the recovering replica is actually transferred. For example, when a replica recovers after a short downtime or a transient network failure, only a small part of state is (probably) required. Similarly, if the replica that is providing state fails before completing the transfer, the state that has been transferred already will not be transferred again.

4.3. A replicated Web service

We implemented a simple counter object accessed as a *Web service* (we assume the reader is familiar with Web services technology [1]). The object is an integer with two methods, read() and increment(). When a client submits the first request, an object private of that client is created and initialized to zero. The Web service is implemented as a servlet hosted by Tomcat. The counter object associated with a client is stored in the HttpSession of that client. The replication engine is similar to that described in the previous section, except for the following.

While in the previous section each operation is associated with a separate URL, in the Web services framework a URL identifies a Web service as a whole. It follows that individual operations of the Web service are to be specified within the request, by means of fields in the SOAP envelope. Thus, we map all operations of a Web service to the *same* servlet, which analyzes the request and then invokes the requested operation (method). The servlet is tailored to the counter object used in this example. It is possible, however, to integrate our replication infrastructure with a fully generic Web service dispatcher, e.g. Apache Axis (indeed, we have recently done so in the context of a replication framework for J2EE containers [39]).

The construction of requests and responses is much simpler than in the previous section, because in this case we are not constrained by the behavior of standard HTTP browsers. The client can construct requests and parse responses programmatically. We have chosen to include in each request a SOAP field, which we call ReplicationHeader, for storing all the necessary information: MethodName, ClientIdentifier, VersionNumber, ActionType. The MethodName is used by the dispatching servlet. The ClientIdentifier and VersionNumber are used by the consistency filter. The ActionType is used by the update filter, as follows. The ActionType is a Boolean that tells us whether a request has to be treated as a 'read' or as a 'write'. This information is necessary because all requests pass through the same filter chain, unlike the previous case in which 'read' operations skip the update filter. In this case, the update filter parses each request and depending on the ActionType value, decides whether to act as a no-op. Our prototype client knows *a priori* the ActionType to be associated with each MethodName but, in a more

Copyright © 2005 John Wiley & Sons, Ltd.

general setting, this association could be described in the Web Services Description Language (WSDL) description of the Web service (in much the same way as parameters to be associated with each operation).

5. PERFORMANCE ANALYSIS

We have carried out an extensive performance analysis for the replicated Web service. The results below refer to a service consisting of three replicas, each placed on a Dell Optiplex GX260 (PIII 800 MHz, 512 MB) running Windows 2000 Professional. Communication occurs through a 100 Mb switched Ethernet. We used Sun Microsystems' JVM 1.4.0 and Tomcat 4.0.6. We configured the JVM executing Tomcat with −Xms128m (initial heap space 128 MB) and −Xmx384m (maximum heap space 384 MB). All experiments began with a warm-up phase of a couple of minutes. Data collected during warm-up were discarded.

We simulated an increasing number of clients by running a publicly available tool on another machine on the same Ethernet (http://grinder.sourceforge.net). Each simulated client constructs a request, waits for the response and sends the next request. Construction of a request involves parsing the response to the previous request. We have verified that the client machine was not the bottleneck in any experiment.

5.1. Preliminary considerations

We measured the throughput in terms of operations per second delivered by the replicated implementation. We also measured the throughput delivered by a single machine (not equipped with the replication engine) in order to establish a baseline. The early experiments resulted in a very low throughput for the replicated implementation, approximately 10–20% of the throughput delivered by a single non-replicated machine. We analyzed the reasons for such a disappointing performance and it soon became clear that the excessive size of each multicast message (11 160 bytes) was a candidate. In the early prototype, each message was a byte stream obtained by applying the standard Java serialization procedure to the HttpSession and the HttpServletResponse. The resulting size was 11 160 bytes, 282 bytes for the HttpSession and the rest for the HttpServletResponse. This large size was quite unexpected. For example, Figure 6 shows a HTTP response returned to the client and it can be seen that it includes less than 800 characters. The reason why the size of a serialized HttpServletResponse is so large is because a HttpServletResponse is a fairly complex object made up of many objects. As it turns out, applying the standard Java serialization procedure results in a very 'verbose' description of all such objects.

In order to use smaller messages, we wrote a custom serialization procedure for HttpServletResponse objects. Essentially, only the portions in bold (Figure 6) are actually transmitted within a multicast. Each replica reconstructs all the missing parts of the HttpServletResponse locally, by means of a procedure hard-wired within the HttpServletResponse implementation. The resulting size of each message became 810 bytes—approximately 7% of the original size. Performance was much more appealing, as shown in the next section, making it evident that this specific optimization is necessary in practice. All of the following results have been obtained with this optimization.

We remark, though, that our HttpServletResponse implementation is tailored to our specific counter object. Our experiments demonstrate that a fully generic replication infrastructure should be able to

Copyright © 2005 John Wiley & Sons, Ltd.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Date: Thu, 08 May 2003 09:32:15 GMT
Server: Apache Tomcat/4.0.6 (HTTP/1.1 Connector)
Content-Length: 613
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Header>
<ReplicationHeader>
<ClientID>0204FB37F25878454538AB543329219AB2A741D87MANOLETE3</ClientID>
<VersionID>3<VersionID >
</ReplicationHeader>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<ns1:incrResponse xmlns:ns1="urn:counter" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:int">1<return>
</ns1:incrResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 6. An example of HTTP response as returned to the client (increment() method).

make the custom serialization procedure for HttpServletResponse *independent* of the specific Web service.

5.2. Throughput

Figure 7(a) shows throughput for 100% write workload, i.e. all clients only invoke the increment() method. It can be seen that the non-replicated implementation is limited by a throughput of approximately 300 operations per second. This is indeed the typical performance level that can be obtained with Tomcat, e.g. [40,41]. As for the replicated implementation, we have observed that its throughput may benefit from a careful tuning of Tomcat. In particular, the key parameter is MaxProcessors, the maximum number of threads that Tomcat internally uses for processing requests. The figure shows a curve for Tomcat 'out-of-the-box' (MaxProcessors = 75) and one for Tomcat 'tuned' (MaxProcessors = 175). With MaxProcessors in the range 120–200, the behavior remains almost identical to the 'tuned' curve. Beyond 200, throughput of the replicated implementation drops. Tuning the value of MaxProcessors has no effect on the non-replicated implementation.

It can be seen that, beyond 120 clients, the (tuned) replicated implementation delivers a *better* throughput than the non-replicated implementation. This positive result was unexpected. In the replicated implementation each replica has to participate in the execution of *all* requests (recall that we are considering a 100% writes workload). Thus, we expected that the throughput of the non-replicated implementation would be an upper bound for the throughput of the replicated implementation. In fact, since receiving a multicast involves a significant cost at the group communication layer, we thought the upper bound would not even be achieved. This expectation turned out to be wrong. The reason is because we underestimated the cost of receiving and parsing HTTP requests, and constructing

Copyright © 2005 John Wiley & Sons, Ltd.

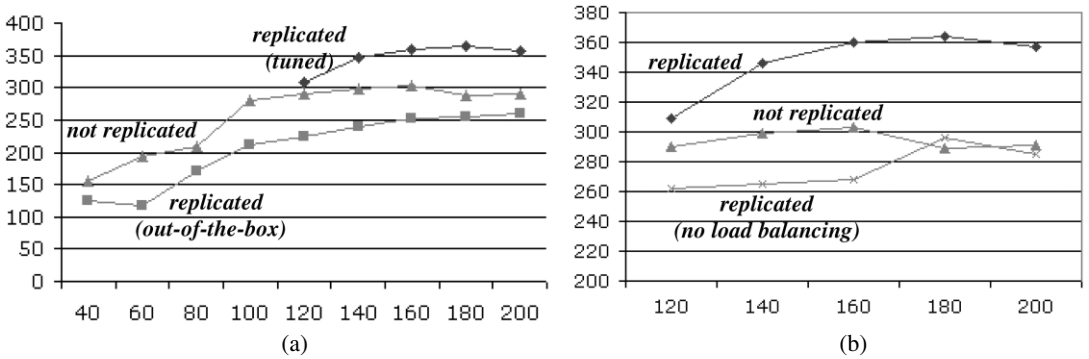


Figure 7. Throughput in operations per second as a function of the number of clients. Both graphs refer to a 100% writes workload. (b) Considers only the 'tuned' replicated implementation (see the text).

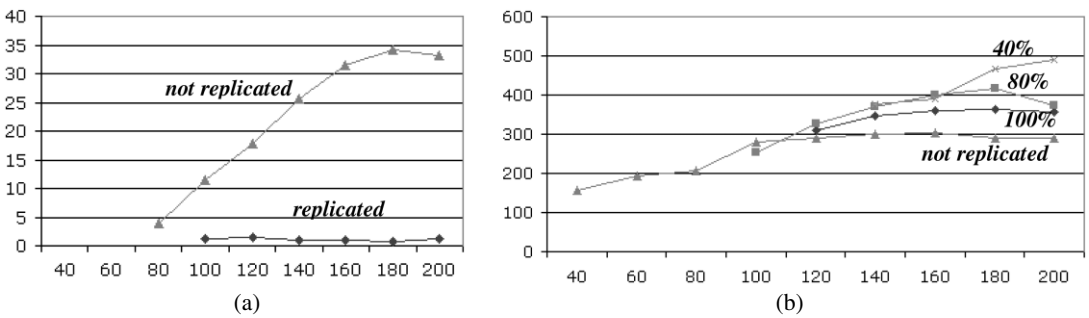


Figure 8. (a) Percentage of connection refused errors as a function of the number of clients, for 100% writes workload. (b) Throughput in operations per second as a function of the number of clients, for varying workloads: 40, 80 and 100% writes. For the non-replicated implementation only one curve is given as the throughput is independent of the workload. In both figures, only the 'tuned' version is given for the replicated implementation.

and sending HTTP responses. In the replicated implementation this cost is fairly distributed amongst replicas, each replica being responsible for one third of the total number of requests. As it turns out from our experiments, distributing this cost may partly compensate for the overhead intrinsic to replication. Figure 7(b) demonstrates that this interpretation is correct: by eliminating the load-balancing module, all of the HTTP processing load is placed on one replica and throughput falls below that of the non-replicated implementation. Another important benefit that follows from distributing the HTTP processing load can be observed in Figure 8(a), which shows the percentage of connection refused errors as a function of the number of clients.

Copyright © 2005 John Wiley & Sons, Ltd.

So far, we have considered a 100% write workload. Figure 8(b) shows the throughput for several mixed read/write workloads. It can be seen that, as expected, when the workload includes 'read' operations the throughput of the replicated implementation grows. We expect that when the workload is close to 0% write, the replicated implementation should exhibit a close-to-linear scalability, i.e. a throughput close to k times the throughput of the replicated implementation, k being the number of replicas. However, we did not address this scenario in our experiments.

We argue that the specification of a Web service should make it possible to specify the nature of each operation ('read' versus 'write'), because this information could be exploited by the replication infrastructure in order to improve performance. As demonstrated by our experiments, the performance gain that could be obtained is significant. When this information is not available, a replication infrastructure can only handle each operation as if it were a 'write', thereby introducing unnecessary overhead.

5.3. Latency

We measured the latency incurred in the execution of a request. We measured time intervals with a publicly available timing library that exploits system-specific hooks and provides a resolution of 1 μs [42]. We did not use the standard timer available in Java through the System class, because its resolution (approximately 16 ms) was not sufficient for our measurements. The results below are the average of 10 requests, after discarding warm-up data.

The latency perceived by the client is 25 ms for the non-replicated implementation. This rather high value is due to the high cost incurred in HTTP and SOAP processing and is in line with the findings obtained in [43]. The cited paper contains an extensive performance analysis of SOAP implementations and demonstrates, according to the authors, that SOAP is orders of magnitude slower than JavaRMI or CORBA. The latency perceived by the client for the replicated implementation is 36 ms. Although the additional 11 ms constitutes a significant relative increase with respect to the non-replicated case, we remark that in many practical cases the baseline latency could be much higher than 25 ms: (i) clients will usually access the service through a wide-area network, in which case the latency of the network alone may be tens of milliseconds; (ii) we are considering a very simple object (an integer), more complex objects will introduce higher latency.

In order to gain insights into the cost of the replication infrastructure, we instrumented the code in order to obtain a detailed breakdown of the latency incurred within the service. The results are as follows: 1.38 ms are spent in the load-balancing filter; 1.55 are spent in the consistency filter; the remaining $11 - 1.38 - 1.55 = 8.07$ ms are spent in the update filter. Most of this time is due to the multicast latency (6 ms). This latency value is in line with the values obtained for Spread alone on hardware similar to ours [44], having considered that we use 'safe delivery', i.e. the strongest delivery guarantee available in Spread.

6. CONCLUDING REMARKS

With the increasing trend towards intra-enterprise and inter-enterprise integration, the need for reliable implementations of replicated services will grow. In particular, because the reliability requirements of a service whose clients are programs must necessarily be more stringent than when clients are humans.

Copyright © 2005 John Wiley & Sons, Ltd.

Meeting such requirements in environments characterized by possibly unreliable networks, which is the case when remote programs have to interact, is particularly challenging.

In this paper we have proposed a detailed algorithm for service replication addressing the above needs. The algorithm preserves the linearizability consistency criterion of the non-replicated implementation in a replicated setting. When a replicated service is no longer able to provide its guarantees, due to an excessive number of failures, the service stops being available, rather than continuing to work while silently giving up such guarantees. Moreover the algorithm enables clients to simply resubmit an unanswered update request multiple times without incurring the risk of having the update executed multiple times, not even if each retransmission is addressed to a different replica. This feature is particularly attractive to simplify the management of network failures by client programs.

The algorithm has been implemented in the context of Apache Tomcat for replicating HTTP session data and for replicating objects accessed as a Web service. The implementation has been described in detail and its performance has been analyzed extensively. As it turns out, the three-way replicated system is capable of sustaining a throughput comparable to that of a single non-replicated machine—more than 300 operations per second with more than 200 clients. The latency increase is moderate and smaller than the latency cost intrinsic into wide-area networks, for example. When the workload includes a significant percentage of 'read' operations, the throughput of the replicated system is higher (we argue that it could be made even higher than that which we measured by tuning the load-balancing policy, but we have not investigated this issue in depth). We believe that the proposed approach could be a realistic solution for those settings where availability and reliability are far more important than state-of-the-art performance.

ACKNOWLEDGEMENTS

This work has been partly supported by the EEUU (project Adapt: Middleware Technologies for Adaptive and Composable Distributed Components, IST-2001-37126), Microsoft Research Ltd. (Cambridge, U.K.). Alessandro Brunettin and Anna Vatta implemented the Web server prototype. Bettina Kemme, Ozalp Babaoglu and Vance Maverick provided very useful comments. All of this support is gratefully acknowledged. The authors are grateful to the anonymous reviewers for their numerous and constructive comments.

REFERENCES

1. Gottschalk K, Graham S, Kreger H, Snell J. Introduction to Web services architecture. *IBM Systems Journal* 2002; **41**(2):170–177.
2. Cardellini V, Casalicchio E, Colajanni M, Yu P. The state of the art in locally distributed Web server systems. *ACM Computing Surveys* 2002; **34**(2):263–311.
3. Vogels W, Dumitriu D, Birman K, Gamache R, Short R, Vert J, Massa M, Barrera J, Gray J. The design and architecture of the Microsoft Cluster Service—a practical approach to high-availability and scalability. *Proceedings of the 28th Symposium on Fault-Tolerant Computing*, June 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 422–432.
4. Kosuge Y, Hirano C, Lascu O, Marcantoni C, Vovk J. Exploiting HACMP 4.4: Enhancing the capabilities of cluster multiprocessing. *IBM RedBooks SG245979*, IBM, 2000.
5. Lamport L. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers* 1979; **28**(9):690–691.
6. Davidson SB, Garcia-Molino H, Skeen D. Consistency in partitioned networks. *ACM Computing Surveys* 2002; **17**(3):341–370.

Copyright © 2005 John Wiley & Sons, Ltd.

7. Raynal M, Thia-Kime G, Ahamad M. From serializable to causal transactions for collaborative applications. *Proceedings of the 23rd IEEE Euromicro Conference*. IEEE Computer Society Press: Los Alamitos, CA, 1997; 314–321.
8. Herlihy M, Wing J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 1990; **12**(3):463–492.
9. Merzbacher M, Patterson D. Measuring end-user availability on the Web: Practical experience. *Proceedings of the International Performance and Dependability Symposium*, June 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 473–477.
10. OMG. *Common Object Request Broker Architecture v2.6*. Object Management Group: Needham, MA, 2001.
11. Moser L, Melliar-Smith P, Narasimhan P. Consistent object replication in the eternal system. *Theory and Practice of Object Systems* 1998; **4**(2):81–92.
12. Mishra S, Fei L, Lin X, Xing G. On group communication support in CORBA. *IEEE Transactions on Parallel and Distributed Systems* 2001; **12**(2):193–208.
13. Frølound S, Guerraoui R. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems* 2001; **12**(2):133–146.
14. Powell D (ed.). Special issue on group communication. *Communications of the ACM* 1996; **39**(4):50–97.
15. Birman KP, Joseph TA. Exploiting virtual synchrony in distributed systems. *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. ACM Press: New York, 1987; 123–138.
16. Birman K. A review of experiences with reliable multicast. *Software—Practice and Experience* 1999; **29**(9):741–774.
17. Bartoli A. Implementing a replicated service with group communication. *Journal of Systems Architecture* 2004; **50**(8):493–519.
18. Birman K. Maintaining consistency in distributed systems. *Technical Report TR 91-1240*, Department of Computer Science, Cornell University, November 1991.
19. IONA/Isis. *An Introduction to Orbix + Isis*. IONA Technologies and Isis Distributed Systems: Dublin, 1994.
20. Maffeis S. Adding group communication and fault-tolerance to CORBA. *Proceedings of the 1995 USENIX Conference on Object-oriented Technologies*, June 1995. USENIX Association: Berkeley, CA, 1995; 135–146.
21. Vaysburd A. Building reliable interoperable distributed objects with the maestro tools. *PhD Thesis*, Department of Computer Science, Cornell University, 1998.
22. Guerraoui R, Eugster P, Felber P, Garbinato B, Mazouni K. Experiences with object group systems. *Software—Practice and Experience* 2000; **30**(12):1375–1404.
23. Felber P, Narasimhan P. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers* 2004; **53**(5):497–511.
24. Amir Y. Replication using group communication over a partitioned network. *PhD Thesis*, The Hebrew University of Jerusalem, 1995.
25. Schiper A, Raynal M. From group communication to transactions in distributed systems. *Communications of the ACM* 1996; **39**(4):84–87.
26. Agrawal D, Alonso G, El Abbadi A, Stanoi I. Exploiting atomic broadcast in replicated databases. *Proceedings of EuroPar 1997* (*Lecture Notes in Computer Science*, vol. 1300), Lengauer C, Griebl M, Gorlatch S (eds.). Springer: Berlin, 1997; 496–503.
27. Kemme B, Alonso G. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems* 2000; **25**(3):335–379.
28. Patiño-Martínez M, Jiménez-Peris R, Kemme B, Alonso G. Scalable replication in database clusters. *Proceedings of the 14th Symposium on Distributed Computing (DISC 2000)* (*Lecture Notes in Computer Science*, vol. 1914), Herlihy M (ed.). Springer: Berlin, 2000; 315–329.
29. Amir Y, Tutu C. From total order to database replication. *Proceedings of the 22nd International Conference on Distributed Computing Systems*, July 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 494–503.
30. Karamanolis C, Magee J. Client-access protocols for replicated services. *IEEE Transactions on Software Engineering* 1999; **25**(1):3–21.
31. Ueno K, Alcott T, Blight J, Dekelver J, Julin D, Pfannkuch C, Shieh T. WebSphere scalability: WLM and clustering. *IBM RedBooks SG246153*, IBM, 2000.
32. Gribble S, Brewer E, Hellerstein J, Culler D. Scalable, distributed data structures for Internet service construction. *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000. USENIX Association: Berkeley, CA, 2000; 319–332.
33. Vogels W, van Renesse R, Birman K. Six misconceptions about reliable distributed computing. *Proceedings of the 8th ACM SIGOPS European Workshop*, September 1998. ACM Press: New York, 1998.
34. Patterson DA *et al.* Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. *Technical Report UCB//CSD-02-1175*, Department of Computer Science, UC Berkeley, 15 March 2002.
35. Bartoli A, Babaoglu O. Service replication with sequential consistency in unreliable networks. *Technical Report*, University of Trieste, January 2002. Available at: http://www.webdeei.univ.trieste/Archivio/Docenti/Bartoli/repProgram.pdf.

Copyright © 2005 John Wiley & Sons, Ltd.

36. Othman O, O'Ryan C, Schmidt DC. Strategies for CORBA middleware-based load balancing. *IEEE Distributed Systems Online* 2001; **2**(3). Available at: http://dsonline.computer.org.
37. Hanik F. In-memory session replication with Tomcat 4, April 2002. http://www.theserverside.com.
38. Kemme B, Bartoli A, Babaoglu Ö. On-line reconfiguration in replicated databases based on group communication. *Proceedings of IEEE/IFIP Dependable Systems and Networks 2001*. IEEE Computer Society Press: Los Alamitos, CA, 2001; 117–126.
39. Babaoglu Ö, Bartoli A, Maverick V, Patarin S, Vučković J, Wu H. A framework for prototyping J2EE replication algorithms. *Proceedings of the International Symposium on Distributed Objects and Applications (DOA) 2004* (*Lecture Notes in Computer Science*, vol. 3291), Meersman R, Tari Z (eds.). Springer: Berlin, 2004; 1413–1426.
40. Web Performance Inc. Servlet performance report: Comparing the performance of J2EE servers, July 2004. http://www.webperformanceinc.com/library/ServletReport/index.html [11 April 2005].
41. KeyLabs. SunONE and Apache/Tomcat performance comparison, August 2003. http://www.keylabs.com/results/sun/sunone.html [11 April 2005].
42. Roubtsov V. My kingdom for a good timer! January 2003. http://www.javaworld.com [11 April 2005].
43. Davis D, Parashar M. Latency performance of SOAP implementations. *Proceedings of the 2nd IEEE/ACM Symposium on Cluster Computing and the Grid, Workshop on Global and Peer-to-Peer on Large Scale Distributed Systems*. IEEE/ACM Press: New York, 2002; 407–412.
44. Amir Y, Danilov C, Miskin-Amir M, Schultz J, Stanton J. The Spread toolkit: Architecture and performance. *Technical Report CNDS-2004-1*, Johns Hopkins University, July 2004.

Copyright © 2005 John Wiley & Sons, Ltd.