A method for combining replication with cacheing

M. C. Little and S. K. Shrivastava
Department of Computing Science, Newcastle University, Newcastle upon Tyne, England, NE1 7RU
(M.C.Little@ncl.ac.uk, Santosh.Shrivastava@ncl.ac.uk)

Appeared in the proceedings of the International Workshop on Reliable Middleware Systems, Lausanne Switzerland,October 19th-22nd.

Abstract

Object replication and cacheing have been used individually in distributed systems for many years. There are benefits from being able to support both: replication for availability, and cacheing for performance. Although both involve handling multiple copies of objects, there are sufficient differences between the two to make the design of an integrated approach a challenging exercise. In this paper we shall describe a design that provides the benefits of both replication and cacheing by allowing both types of protocols to co-exist within the same application. We do not propose a new replication or cacheing protocol; rather a way in which existing implementations can be combined.

1. Introduction

Object replication and cacheing have been used individually in distributed systems for many years. Although both types of protocols allow for the creation and maintenance of multiple copies of an object (a resource), they do so for different purposes: replication for availability, and cacheing for performance. In addition, each protocol type suffers from deficiencies for which the other offers possible solutions, e.g., replicated services typically do not perform as well as non-replicated counterparts.

Although applications which require replication or cacheing use either one or the other, there are benefits from being able to support both: replication for fault-tolerance and availability, and cacheing for performance. However, there are sufficient differences between the two types of protocols that attempting to implement one type using the other may result in inefficient implementations.

In this paper we shall describe a unifying approach to both replication and cacheing, using existing protocols and implementations, which provides the benefits of both availability and cacheing performance. We shall also give an indication of a possible implementation of this model which we are currently undertaking.

2. Replication and cacheing

Before describing the unified replication and cacheing model, it is necessary to give a brief overview of both types of protocols, concentrating on their key features.

2.1 Replication

Replication protocols (*replica consistency protocols*) are used in a distributed system to increase the availability of critical resources (services). As shown in Figure 1, when an object is replicated the replicas are treated as a single abstract entity, the *replica group*. Information about the group membership (the *group view*) is maintained by the replication protocol. It may also be possible to improve the performance of a service by replicating it, although this is typically only for read operations, e.g., a client reading from a replica which is located physically closer.

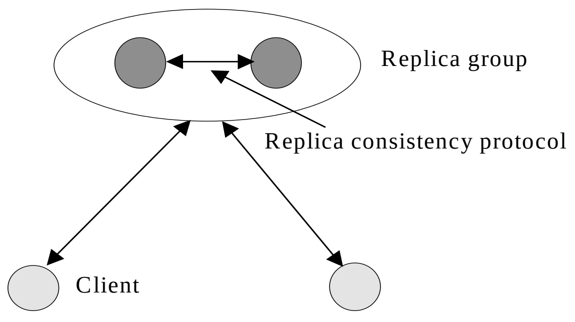

Figure 1: Replica group interactions.

Replication protocols can be categorised by the type of consistency they maintain between the replicas: *strong consistency*, where all replicas have identical states (e.g., [1,2]), and *weak consistency*, where replica states are allowed to diverge (e.g., [3]). Within these categories there are basically two classes of protocols:

- *active replication*: more than one copy of an object is activated on distinct nodes and all copies perform processing [1]. Active replication is often the preferred choice for supporting high availability of services where masking of replica failures with minimum time penalty is considered highly desirable. Since every functioning replica performs processing, active replication requires that they receive identical invocations in an identical order, and that the computation performed by each replica is deterministic.
- *passive replication*: Here only a single copy (the primary) performs processing; the primary regularly checkpoints its state to other 'backup' (secondary) members of the group [4]. Passive replication does not require the restriction of deterministic processing; however, its performance in the presence of primary failures can be poorer than under no failures, because the time taken to switch over to a secondary is non-negligible.

Within a strong consistency protocol replicas must be treated as a single group by the application in a manner which preserves mutual consistency. Suppose the replication policy is active replication. Consider the following scenario (see Figure 2), where client group G_A (replicas A_1, A_2) is invoking a service operation on group G_B (a single object B) and B fails during delivery of the reply to G_A. Suppose that the reply message is received by A_1 but not by A_2, in which case the subsequent action taken by A_1 and A_2 can diverge. The problem is caused by the fact that the failure of B has been 'seen' by A_2 and not by A_1. To avoid these problems, communication between replica groups requires reliable distribution and ordering guarantees not associated with non-replicated systems: reliability ensures that all correctly functioning members of a group receive messages intended for that group and ordering ensures that these messages are received in an identical order at each functioning member.

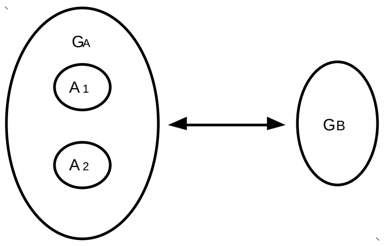

Figure 2: Operation invocation for replicated objects.

One of the drawbacks of using strong consistency replication is the overhead incurred for ensuring that all replicas have the same state. If replicas need not be fully consistent, then more "lightweight" protocols can be used. Weak consistency replication offers a trade-off between consistency and performance. Typical protocols allow the states of replicas to diverge in a controlled manner, maintaining a specific maximum level of inconsistency, e.g., replicas may only be synchronised after a specified time interval, or number of updates.

However, this then leads to a different problem: what types of application can best use inconsistent replicas? Whereas when an application uses a strongly consistent replica group it perceives the group as a highly available (albeit possibly slower) counterpart to the non-replicated object, with weak consistency each client may essentially see a completely different object. Thus, without application specific corrective measures, clients sharing the object may end up take conflicting actions. This may be one of the reasons why such protocols have found limited use.

2.2 Cacheing

Whereas replication protocols are employed in distributed systems to increase the availability of resources, cacheing protocols are used to improve application performance. Despite the fact that the performance of network infrastructures is improving, processor performance is increasing faster, with the result that in a distributed system the network is often the application bottle-neck. Reducing the frequency of network interactions is therefore an obvious way to improve application performance (e.g., [5,6]) .

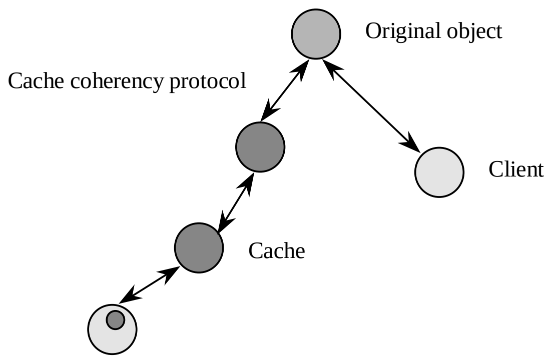

Figure 3: Client and cache interactions.

A cache is essentially a copy of the object which resides with users, or physically closer to them than the original object, e.g., within a separate server in the case of the Web. As illustrated in Figure 3 (which shows multi-level cacheing), to reduce the latency in accessing (possibly many) distributed objects, clients use the cache rather than the original object(s). Obviously certain clients may continue to use the original object. The cache is typically populated on demand by obtaining a copy of the object either from the original location or another cache.

Unlike replication, clients typically only know of a single cache copy; thus its failure may prevent the client from being able to access the object. In addition, since a cache may be associated with only a single client it is often possible to co-locate caches with clients, since they are no longer required once the client terminates; most replica groups, however, are shared resources and exist beyond the lifetime of individual clients, thus co-locating replicas is often not possible.

By reducing the amount of network activity, cacheing of read-only objects offers the potential for a significant performance benefit to an application. However, if the state of cached objects can change, a cache coherency protocol may be required to ensure that all clients (eventually) see such updates. The manner in which updates are managed will be protocol specific, e.g., many cache protocols employ a *write-through* policy, whereby updates which are applied to the cache are sent to the original object first. In the case of the Web, updates can only be made to the original object (document), and caches are not informed of updates; each document has a "time-to-live" associated with it (estimated from how frequently it was updated in the past), and a cache will only fetch a new copy if it believes its current copy is out-of-date.

3. Unifying replication with cacheing

In this section we shall describe a model that provides the benefits of both replication and cacheing by allowing both types of protocols to co-exist within the same application. We are not proposing a new replication or cacheing protocol; only a way in which existing implementations can be combined. Our basic idea is straightforward: we essentially propose replacing an original object and its caches by corresponding replica groups. Each group member is enriched with replica consistency and (where appropriate) cache consistency objects that perform protocol specific operations.

3.1 Consistency domains

The main concept within our model is that of a *consistency domain*. With reference to Figure 4, a consistency domain is a grouping of replicas which have the same state, i.e., it is essentially a strongly consistent replica group. There are no restrictions on the locations of the members of a consistency domain (e.g., they may be co-located with clients), or on the size of a domain; correspondingly, membership of a domain need not be static. To its users, a domain therefore appears as a single (possibly highly available) object, whose members may also be located to optimise client performance. Therefore, all clients which require to observe the same state for replicas will use the same consistency domain.

The model does not impose restrictions on *intra-domain* strong consistency protocols; in principle, one domain could use active replication whereas the other passive, and each domain may tolerate different types of member failures. Externally such protocols are not visible to users, so any implementation can be used. Consistency protocols will normally require support from some form of object group service (e.g., [2]).

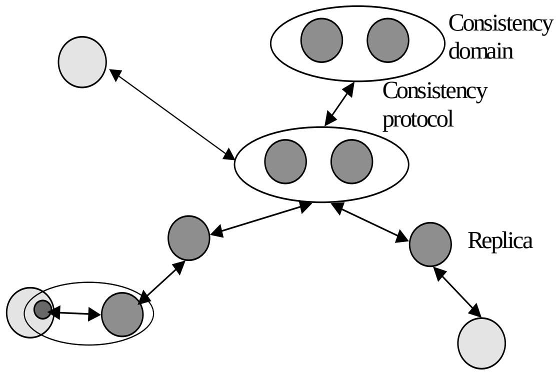

Figure 4: Replication and cacheing model.

To maintain levels of (in-)consistency between the object copies, the domains must co-operate to propagate updates; the protocols used to connect these domains and maintain their levels of inconsistency are essentially the cacheing protocols used in existing distributed systems. For the same reason that we wish to support arbitrary intra-domain consistency protocols, i.e., that each protocol may need to be configured for the domain that uses it, our model does not impose any restrictions on the implementations for the *inter-domain* consistency protocols: each inter-domain connection may therefore be managed by a different protocol.

Structuring an object's replicas into consistency domains will be an application specific task. It is unlikely that the domain configuration for one object will be exactly the same for all objects, or that it will remain unchanged between different incarnations of the application. New consistency domains may be formed by merging or splitting existing domains. Additionally, because we wish to continue to provide the performance benefits of cacheing by reducing their network usage, consistency domains (or their constituent members) may be created and destroyed dynamically during the lifetime of an object. For example, in Figure 4, one of the members of a consistency domain is co-located with a client.

By structuring object replicas into different, co-operating consistency domains, it is possible to guarantee that clients which require consistency can observe the same replica states (use the same consistency domain). At the same time application overheads can be reduced by not requiring all replicas to be consistent, and allowing replicas to be co-located with clients. Although this model is simple, it is powerful and flexible, giving applications control over the configuration of object copies, and allowing them the benefits of performance, consistency and availability.

3.2 Consistency objects

To understand how consistency domains operate and can be used to group replicas into different levels of consistency, we must first define a basic model for application objects. Note, this model represents just one way of incorporating consistency domains, and should not be seen as the only possible model.

We assume that all objects contain persistent state, and that they have operations for obtaining and updating their states to/from an object store (*getState* and *updateState*). We further assume that all operations on an object will require (at least) to invoke *getState* prior to doing any work. Obviously these state accesses may be hidden by an appropriate high-level API, as shown in Figure 5.

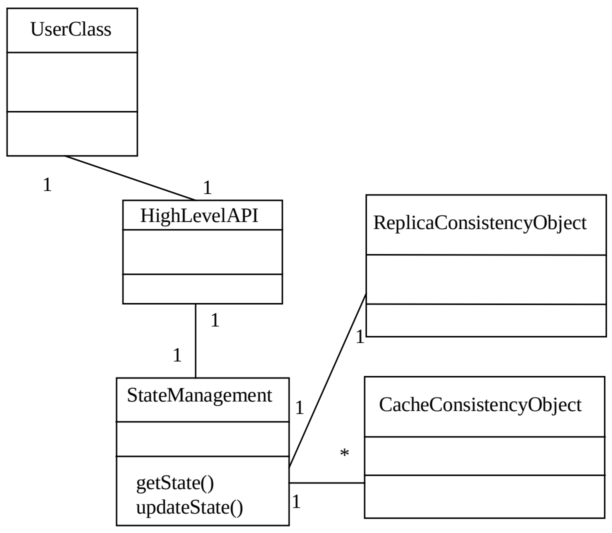

Figure 5: Object model.

To maintain inter-domain consistency, each domain must possess sufficient information about which other domains it is connected to. Likewise, each domain will require membership information to maintain its internal consistency, i.e., keep the replica states identical. In addition, we wish to support arbitrary inter- and intra-consistency domain protocols. Within our model the *consistency object* addresses these issues.

For *every* domain inter-connection, there is a corresponding *cache* consistency object, which is responsible for managing the consistency between the domains *in a protocol specific manner*. It is the consistency object which hides the protocol specific nature of connections. Importantly, the cache consistency object is only maintained by the domain which is *responsible for guaranteeing consistency* (see later for examples). This gives the system the flexibility to configure the responsibility for maintaining consistency. Intra-domain consistency is also managed by a single *replica* consistency object, collocated with each member object.

When an object is initially created in an un-replicated (or single-domain) form, there is only the replica consistency object associated with the domain (actually with each domain member), which is responsible for managing the intra-domain consistency. To manage the inter-domain consistency, a cache consistency object is associated with the domain whenever a new domain connection is established. When connections are removed, the corresponding cache consistency object will also be removed. Therefore, although there will only ever be a single replica consistency object for each domain, there may be as many cache consistency objects as there are domain connections.

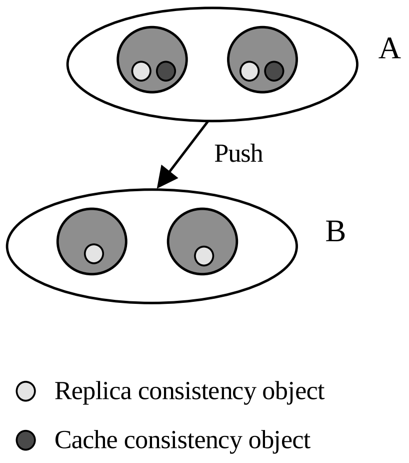

Figure 6: Push protocol.

For example, consider Figure 6, where domain A is the *master consistency domain* (i.e., the one with the most up-to-date copies of the object), and the different consistency objects are represented as small circles within the domain's constituent members; members of A and B always require a replica consistency object. For the sake of illustration, we shall assume a primary copy strategy for maintaining replica consistency. As mentioned previously, A's primary must checkpoint its state to the backup *whenever it changes*. In this first example we give responsibility for inter-domain consistency maintenance to domain A, whose primary periodically (depending upon the protocol agreed between A and B) "pushes" its new state to B. Because A is responsible for maintaining consistency with B, its members have a cache consistency object which manages the connection and protocol to B.

Alternatively, in Figure 7 we have removed the requirement on A for maintaining consistency with other domains; therefore, its members only require their own replica consistency object. Domain B is now responsible for periodically "pulling" its state from A, and therefore its members have a corresponding cache consistency object to do this.

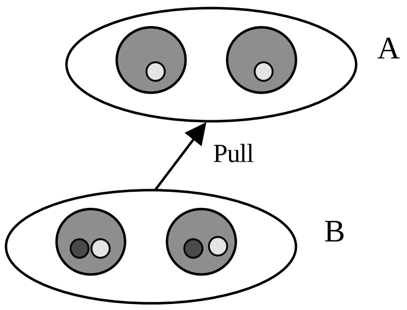

Figure 7: Pull protocol.

Note, it is possible for a given domain to receive concurrent messages from different consistency domains and even different objects within the same domain. We assume that the intra-domain replica consistency protocol/group service at the sending or receiving domain is used to manage such requests, e.g., filtering out duplicates or ordering messages consistently between domain members.

In order to maintain inter-domain consistency, cache consistency objects must intercept all accesses to an object's state. (Replica consistency objects may behave similarly, although some implementations may require interception at client side, at the object invocation stage). Therefore, as shown in Figure 5 the state management layer delegates all *getState* and *updateState* calls to the consistency objects. Whenever an application object requires its "current" state (*getState* is called), the state management layer must ask each cache consistency object for the state. Each consistency object will act in a manner specific to the connection (and hence consistency protocol) it represents, either returning a new state or an indication that the application object's current state is up-to-date.

For example, a cache consistency object may determine that the state the object already has is sufficient to meet its consistency requirements (e.g., it was last updated within a specific time interval). However, if another consistency object determines that a new state must be used, then it will obtain this state from its corresponding consistency domain.

Obviously only the most up-to-date state which satisfies all cache consistency objects should be used. Thus, consistency objects can be ordered, such that obtaining a new state from one can be used to implicitly satisfy the remaining consistency objects

If the state of the object is modified during an operation, each consistency object will be informed by the state management layer when *updateState* is called. It is once again up to the individual consistency objects to determine whether they must propagate such updates to other consistency domains; ordering of consistency objects may mean that not all such objects need be informed. Obviously the act of propagating updates (essentially calling *updateState* on objects in other domains) may cause other domains to propagate further updates, i.e., a ripple of state updates can occur.

4. Implementation

Having presented our model we shall now outline one possible implementation. For this we have chosen to combine a standards compliant transaction system, *OTSArjuna*, with replication. The *OTSArjuna* model for building transactional applications exploits object-oriented techniques to present programmers with a toolkit of classes from which application classes can inherit to obtain desired properties, such as persistence and concurrency control [7,8]. Each class is concerned with a single functionality, and these classes form a hierarchy, part of which is shown in Figure 8.

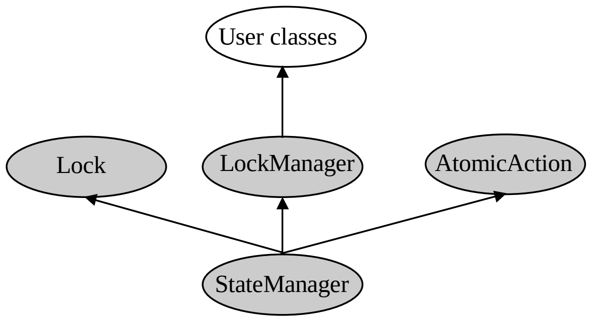

Figure 8: OTSArjuna class hierarchy.

StateManager is the class responsible for naming, persistence and recovery control of persistent objects. When not in use, persistent objects are stored in an object store. To satisfy the durability requirement, StateManager makes use of a persistence service when loading and storing persistent objects.

LockManager is responsible for two-phase locking, and uses instances of the Lock class, in conjunction with a suitable concurrency control service. The programmer uses the `setlock` method to acquire appropriate locks; the transaction system is responsible for releasing these locks.

Prior to using an object's state, each method *must* obtain appropriate locks, as shown below. LockManager uses this lock acquisition as a means of driving StateManager to load the object's current state, and record, from the type of lock, whether or not the state is being modified. Assuming the transaction commits and the object state was modified, the StateManager component of the object is instructed to save the new state to the object store.

```
Boolean Example::foobar ()
{
  AtomicAction A;
  Boolean result = FALSE;

  A.begin();

  if (setlock(new Lock(WRITE) == GRANTED)
  {
    /*
     * Do some work using object's state.
     */

    if (A.commit() == COMMITTED)
    {
      result = TRUE;
    }
  }
  else
    A.rollback();

  return result;
}
```

With *OTSArjuna*'s object model, interception of accesses to an object's state and hence incorporating an implementation of our unified replication and cacheing model is relatively straightforward. All StateManager accesses to the object's state currently go directly to the persistence service; therefore, we replace this interaction with an indirection through a consistency object "layer", as shown in Figure 9.

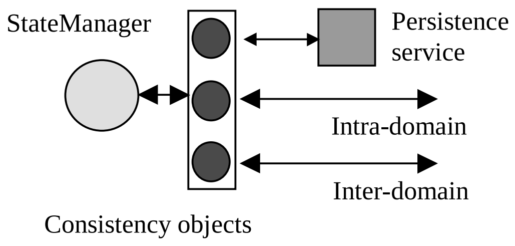

Figure 9: OTSArjuna consistency objects.

Whenever StateManager either requires access to the object's state, or wishes to write the state to the object store, it contacts the consistency objects. Most of these consistency objects will interact with the consistency domains they represent, whereas in the figure, one consistency object may access the state from the local stable storage (if that copy of the state is the most up-to-date), whereas another is responsible for maintaining intra-domain strong consistency. Importantly, all of this is transparent to the object and the application.

5. Conclusions

In this paper we have discussed a simple model for combining object replication and cacheing to enable applications to simultaneously benefit from availability and performance. This model offers the advantage that applications can continue to use existing replication and cacheing protocols by enabling the structuring of replicas into different consistency domains. Domains can be configured to offer specific clients different levels of consistency, performance and availability. Importantly, this is transparent to clients, who use these consistency domains without knowing how the domain maintains its internal and external consistency.

Acknowledgements

The work reported here has been supported in part by a grant from UK Engineering and Physical Sciences Research Council (grant no. GR/L 73708), ESPRIT working group BROADCAST (project No. 22455) and British Telecommunications plc.

6. References

[1] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, 22(4), December 1990, pp. 299-319.

[2] R. Guerraoui, P. Felber, B. Garbinato and K. Mazouni, "System support for object groups", Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications, OOPSLA 98.

[3] H. Garcia-Molina and D. Barbará, "The Case for Controlled Inconsistency in Replicated Data", Proceedings of the IEEE Workshop on Replicated Data, November 1990.

[4] P. A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources", Proceedings of the 2nd International Conference on Software Engineering, 1976.

[5] M. Nelson et al, "Caching in the Sprite File System", ACM Transactions on Computing Systems, February 1988.

[6] J. Archibald et al, "Cache Coherency Protocols: Evaluation using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems, vol. 4, pp. 273-298, November 1986.

[7] M. C. Little and S. K. Shrivastava, "Java Transactions for the Internet", The 4th Conference on Object-Oriented Technologies and Systems (COOTS'98), Santa Fe, New Mexico, USA, April 1998.

[8] M.C. Little and S K Shrivastava, "Implementing high availability CORBA applications with Java",

IEEE Workshop on Internet Applications, WIAPP'99, San Jose, July 1999.