Orcia: Replication for Large-Scale Clusters

Michael M. Swift and Henry M. Levy

University of Washington

{mikesw, levy}@cs.washington.edu

Phone: 206-685-2675  Fax: 206-543-2969

Contact: Michael M. Swift
Orcia: Replication for Large-Scale Clusters

Michael M. Swift and Henry M. Levy

University of Washington
{mikesw, levy}@cs.washington.edu

Abstract

Internet-scale workloads have motivated the use of PC or workstation clusters to provide high availability and scalability. Such workloads depend on data replication, both to tolerate failures and to facilitate load balancing. Unfortunately, traditional replication algorithms are inadequate for large clusters, because they are not targeted for environments with fine-grained load balancing, frequent failures, and frequent configuration changes. This paper presents a replication algorithm targeted specifically at large-scale clusters; the algorithm is an extension to that used in the Porcupine mail server [19], designed specifically to improve scalability and to manage update conflicts. Our evaluation demonstrates that our new algorithm provides many more capabilities with a small increase in overhead over a simple single-master protocol.

1 Introduction

With the growth of the World Wide Web, server applications must scale to handle quickly increasing client populations. As a result, clusters have become a popular structuring technique for implementing a scalable, distributed server [14]. One complexity of the distributed cluster implementation, though, is application data management. While the frequency of failures increases with the cluster size, data must always be available to the application running on the cluster.

Replication is a common technique for providing high-availability services by ensuring that multiple copies of the data exist in the cluster. Replication has also been used to provide scalability by distributing the workload across many machines. While many replication algorithms have been proposed over the past thirty years [7], they are often unsuitable for large-scale clusters for one or more of the following reasons:

- Adding a new replica or removing an existing replica is difficult.
- The protocol is tuned for the wrong environment, such as high-latency networking.
- The consistency guarantees provided are mismatched to those needed for Internet applications.
- Failure of a few machines may prevent access to data, limiting availability.
- Relocating objects to balance load is difficult.

One replication algorithm designed specifically for the clustered environment is that developed by Saito [19] for the Porcupine clustered mail server [18]. This protocol is lightweight and highly available, but it suffers from two weaknesses:

- It does not provide conflict detection and resolves conflicts by over-writing a replicated object with the latest copy of its data.
- It stores the replica set (the list of replica locations) with each object, requiring updates to a large number of objects when the configuration changes due to node failure or addition.

In this paper, we present the Orcia replication protocol (Object Replication for Clustered Internet Applications). While based on Saito’s Porcupine protocol, Orcia adds several key enhancements. First, it detects update conflicts using version vectors, which provide more accurate conflict detection than timestamps. Second, it clusters replicated objects into policy units, which share a common replica set and replication policy. Centralizing the replication information reduces both the amount of storage needed and the number of objects that must be updated when adding a replica. Finally, Orcia doesn’t require any per-object metadata, which reduces the storage costs for small objects.

Orcia was designed for clustered Internet applications and builds on the Porcupine mail system’s...
design, which scales to thousands of nodes. In Porcupine, data for each user is stored on a small number of machines, called the replica set. Machines may be members of many replica sets, so that the failure of a single machine spreads the load increase evenly across the remaining nodes in the system. This is in contrast to systems with a smaller number of replica sets, such as the Echo file system [20], and the Windows 2000 Active Directory [10]. In these systems every node must store a copy of all the objects or the objects must be statically partitioned into groups of replicas. Replicating the complete data set to all machines requires more powerful machines because each machine must store every object. Partitioning machines into separate replication groups, though, decreases efficiency by requiring that the machines in only one group carry the load of a failed machine.

In contrast, the key benefit of Orcia is that it allows objects to be dynamically partitioned into clusters that are replicated to many overlapping replica sets. As a result, the load increase from a failed machine can be spread more evenly over the entire cluster. This increases efficiency of the cluster because less excess capacity must be reserved for tolerating failures.

This paper is organized as follows. Section 2 describes Porcupine’s protocol as background. We then discuss the goals and requirements for the Orcia protocol in Section 3. Section 4 presents the Orcia protocol and some brief examples. We then evaluate the protocol, comparing it to other protocols in Section 5. Finally, Section 6 presents related work and we conclude in Section 7.

2 Porcupine’s Protocol

The Orcia protocol is derived from the replication system used in the Porcupine clustered mail system [18] and formalized in [19]. Porcupine’s protocol allows lightweight updates to an object’s replicas by maintaining a separate replica set for each object. The goals for Porcupine’s protocol are to: (1) allow for frequent failures, (2) allow updates at any machine with a copy of the data, (3) support efficient replica set changes to implement load balancing and addition or removal of replicas, (4) efficiently replicate small objects, and (5) be eventually consistent, so that all nodes eventually have the same values for data. For Porcupine’s protocol, which replicates by transferring data values rather than updates or differences, an object is an atomic unit of data; it is the smallest unit of data understood by the replication protocol. A replica for an object is the copy of the object stored on a machine.

We briefly present Porcupine’s protocol from [19] to illustrate the basis of Orcia. In that protocol, an update changing an object’s data or replica set may be issued to a machine storing a replica. The protocol runs in three phases. In the first phase, the update is applied locally on a machine holding a replica. In the second phase, that machine propagates the change to the other members of the replica set, which then apply the change. In the third phase, the update is retired and each replica set member is notified that the change has been fully propagated. For each object, the only data stored after replication completes is the set of machines storing replicas of the object. While the update is in progress, replication metadata stores the phase of the update, the set of nodes that must be contacted, the set of nodes that have replied, and a timestamp providing total-ordering of updates. However, because this metadata is required only while the update is in progress, the metadata is transient and is stored in a persistent garbage-collected log.

There are four key features of Porcupine’s protocol. First, for every update there are four messages sent: the update and retire messages and their replies. Second, the set of nodes receiving updates is expanded at each replica by including any nodes participating in previous unfinished updates, so that updates aren’t lost during replica set changes. Third, updates are ordered by wall-clock timestamps, and are only rejected when the timestamp for an update is earlier than the timestamp of an existing update of the object. Finally, every update must have a coordinator that ensures that the update is propagated. While the role of coordinator can be assigned in the protocol, the originator of an update typically takes that role. Having multiple coordinators doesn’t affect the correctness of the protocol, however. As a result, if the coordinator crashes, another machine may become the coordinator.

Porcupine’s protocol offers great flexibility for large clusters because every object can be replicated to a different set of machines. As a result, as machines are added, removed, or upgraded, the location of objects can be adjusted dynamically to spread load evenly across a cluster. This allows fine-grained load balancing, in which individual application requests are routed to the replica with the lowest load. However, Porcupine does not detect conflicts from simultaneous updates. In addition, the protocol requires substantial overhead to move a group of objects to a new machine, because every
object must be updated separately with the new replica set.

3 Goals and Requirements

While based on Porcupine’s protocol, the Orcia protocol adds three key new goals:

1. Detect conflicts from simultaneous updates. This is crucial to support applications with stronger consistency requirements than electronic mail. In contrast, Porcupine uses a simple last-writer-wins conflict-resolution mechanism, which is unable to detect conflicts.

2. Make changes to the cluster membership set highly efficient. This improves both handling of failed machines and cluster hardware updates (e.g., addition of machines to improve performance). Our goal is to make dead-machine removal or new-machine addition simple and fast from the point of view of replica changes.

3. Require zero space overhead for objects that are not changing. This is important for situations such as news or email, in which there are objects that are relative small and rarely modified. In these cases, it is beneficial to eliminate metadata storage costs.

To meet these goals, we made several important decisions in the Orcia design. The first is that updates are propagated as values (objects), not as deltas or as operations. As a result, nodes need to receive only the latest update to an object rather than the complete history of updates. The second choice is that conflict resolution is done only once, rather than at every node that detects the conflict. The result is that the resolution mechanism does not need to be deterministic. Third, the protocol requires bi-directional communication because it depends on nodes receiving and then responding to updates. As a result Orcia can avoid unnecessary work by letting the recipient of an update reply that the update has already been overwritten and does not need to be propagated further. Finally, because messages are bi-directional, both sender and receiver cooperate to resolve conflicts. Conflicts between data updates are resolved at the receiver and conflicts between data updates and metadata updates are resolved at the sender.

The Orcia protocol makes several demands of its environment. The protocol depends on permanent unique identifiers for each data item, allowing it to detect object deletions and conflicting updates. The replication system may assign identifiers, such as Globally Unique Identifiers (GUIDs) [9], for all objects. (Applications frequently have their own notion of unique identifiers, such as part numbers for a parts database or ISBNs for a book database, which could be used as well.) Orcia must also be integrated with other cluster services, including a membership service announcing when nodes are added or removed from the cluster, and a location service that determines which nodes store which objects. However, these services are crucial for any clustered application and aren’t unique to Orcia.

4 The Orcia Protocol

The Orcia algorithm extends Porcupine’s protocol with several key changes to support conflict detection, replica set changes, and replica object changes (changes to the objects covered by a replica set). To detect conflicts we replace the timestamp in each update record with a version vector and add an additional round of messages to retire these vectors. The second change, supporting efficient changes to replica sets, adds a new object that combines the replica set from a large number of objects. The final change, allowing objects to be moved between replica sets, extends the concept of an update by allowing it to modify multiple objects rather than a single object. We present each of these extensions, starting with the use of version vectors to enable conflict detection.

4.1 Version Vectors

Orcia uses version vectors to detect simultaneous updates made at different replicas. A version vector, first described in [12], is an array of version numbers as shown in Figure 1. Every node in a replica set has an entry in the vector, and the version for the node represents the last update made at that replica. A version vector compactly represents the update history of an object, so that two versions of an object may be compared; the objective is to discover easily when a past update is present in one version and not in the other. To compare two version vectors, the versions for each node are compared pairwise as shown in Figure 1. If the entries for one vector are all greater than or equal to those of the other vector, then the vectors are compatible. Otherwise, the vectors conflict because each vector incorporates updates missing from the other. In addition, each object has a separate version vector, which ensures
Figure 1: Two examples of version vectors being compared: the first, on the left, shows compatible vectors in which all the versions for vector 1 are greater than or equal to the versions from vector 2, so vector 1 is later. The second example, on the right, shows a case where the vectors conflict: vector 1 has a later version for node B while vector 2 has a later version for node A.

Figure 2: The states of the protocol are active, retiring, finishing, finished, and suspended. Only the coordinator for an update ever enters the suspended and finishing states.

4.2 Conflict Detection Protocol

The basic Orcia protocol consists of four phases. First, when an update is applied locally, a node creates a new update record to hold the replication state for the object. Second, the update record is sent to all replicas, which may accept the change, reject it if it has been overwritten, or ignore it if it conflicts with another active change. Third, the coordinator sends a retire message to notify all replicas that the change has been applied. Finally, the coordinator sends a finish message letting replicas know that they may safely discard the update. A state diagram for this protocol is shown in Figure 2.

The transient state for an update contains an identifier for the update, an identifier for the object, the state of the update, the set of nodes that must be contacted, the set of nodes that have replied, the new replica set, and the latest version vector for the object. If there was no previous update still pending for the object, then the version vector is constructed by setting the version for the node to the current time and the version for all other nodes in the replica set to zero. Otherwise, the existing vector is updated by setting the version for the node to the current time. The timestamps used in version vectors are only compared to timestamps from the same machine, so clock skew is not an issue, as it is with Porcupine’s protocol. This update record is shown in Figure 3.

An update is initialized in the active state, and the set of targets for the update is constructed from the current replica set for the object, the new replica set for the object, and any machines involved in pending updates to the object. The update record is then sent to all the targets. Once an update has been applied by all the targets, the coordinator changes its state to retiring and sends a retire message to all the targets. Upon receiving this message, each node changes the update’s state to retiring and replies. Finally, after receiving retire replies from all the targets, the coordinator changes its state to finishing and sends the finish message to all the targets.
Upon receiving the finish message, the nodes change the update state to \textit{finished} and delete the update. The coordinator does the same once it has received replies to the finish message from all the targets. If at any point during this protocol the update is overwritten, then a node simply acknowledges the subsequent messages without changing the new update.

The first major change from Porcupine’s protocol is the procedure for processing update requests from other nodes. Porcupine’s protocol applies only a simple test for accepting updates: is the timestamp on the new update greater than the timestamp on an existing update. If so, the update is accepted, and otherwise the update is rejected. Orica, however, requires a more complex process. If there is no previous update in the \textit{active} state, then the update is applied. Otherwise, the version vectors of the incoming update and the stored update are compared. If the version vector of the incoming update is greater than or equal to that of the existing update then the update is applied. If the existing update’s vector is greater than the incoming update’s vector, then the new update is rejected because it has been overwritten, and the coordinator changes the update’s state to \textit{suspended}. The update remains in this state until the more recent update overwrites it. Finally, if the two updates were made simultaneously, meaning that neither update incorporates the other’s changes, then their vectors will conflict.

In the conflict case one of the nodes in the replica set, called the \textit{resolver}, creates a new compensating update. The application code using Orica must specify how conflicts are resolved and may supply a resolution procedure. This compensating update has a version vector later than both conflicting updates, so it overwrites both of them. Only one machine should choose to be the resolver for two conflicting updates, but if two nodes choose to be resolver, then their compensating updates can be reconciled again. As a result, the algorithm for picking the coordinator should ensure that the procedure will terminate. A simple option for choosing a single resolver is to pick the node with the lowest node identifier. Other nodes that aren’t the resolver ignore the conflicting update and wait for the compensating update to arrive.

The second major change from Porcupine is the addition of the finish message. This message allows version vectors to be discarded after an update has competed. When a node has received the finish message, it knows that all other nodes that haven’t overwritten the update are in either the \textit{retiring}, \textit{finishing}, or \textit{finished} state. As a result, if the node generates a new update with a fresh version vector, containing zeroes for all nodes but itself, that vector will not conflict with the existing update. Thus, version vectors are only needed while an update is in progress and may be discarded after it has finished. The finish message may be delayed, trading off bandwidth and message processing for the cost of remembering the version vector. This is similar to the technique in [17] for discarding version vectors that have been fully propagated.

Figure 4 demonstrates normal operation of the protocol. The update is propagated in the first exchange, and then retired and then finished in the next two exchanges. In Figure 5 we show the flow of messages when two nodes simultaneously update an object. In this case, the conflict between the updates is resolved at one node, which rejects one of the two updates and creates a new update that resolves the conflict.
labeled with a unique identifier, as is each revision to the replica set membership. Each machine stores a log of replica set identifiers, which allows it to determine if the replica set for an incoming update is the same as its local version of the replica set or is an earlier version.

We extended Porcupine’s protocol to use policy units by tagging all update records with the identifiers of the policy unit containing the object and its current replica set. Any node receiving the update can immediately identify whether or not the update has an old version of the replica set, or, if the replica set identifier is unknown, a newer version. When applying an update from another node, a machine rejects the update if its replica set identifier is out of date. In this case, the node sends back the error code obsolete-replica-version. This error forces the coordinator of the update to mark the update as awaiting-replica and wait for the new replica set to arrive. Once a new replica set arrives, the coordinator creates a new update record with a new version vector and replica set version identifier and sends it to both the old and the new replica sets.

A node may also reject an update if it holds a replica and does not recognize the replica set identifier. In this case, an update to the replica set must not have been fully propagated, so the node sends back the error unknown-replica-version. This error can only occur while an update to the replica set is in progress, so the reply forces the coordinator of the update to resend the update once the current change to the replica set is retired.

The policy unit object is also replicated using the Orcia protocol. When a node receives an update to a policy unit that adds it to the replica set, the node delays replying until it has copied all the objects. The objects only need to be sent to machines hosting new replicas, so when a node receives an update adding it to a policy unit, the update’s state is set to bulk-pending and the node contacts the coordinator to copy the objects. Once the data arrives, the node then makes the update active and replies to the coordinator. The actual protocol used to copy the data is not important as long as it ensures that all the data eventually arrives. A state diagram describing the protocol is shown in Figure 6, and an example of this protocol is shown in Figure 7. In this example, one node updates an object while another adds a replica to the policy unit. The node adding the replica rejects the update to the object because its replica set identifier is out of date. The node updating the object then creates a new update with the new replica set and resends it.

Figure 5: Two simultaneous updates to object X conflict, and node A resolves the conflict by creating a new update that incorporates both changes. This update is then propagated normally. The solid lines are request messages, and the dotted lines are replies. The retire and finish exchanges aren’t shown. The states shown are before messages have been received.

4.3 Replica Set Changes

The Orcia protocol as described supports conflict detection but still requires that every object store its replica set separately. This is inefficient for two reasons: first, if many objects have the same replica set, then storing the replica sets separately is redundant. Second, moving a group of objects to a new machine requires updating the replica sets of all the objects being moved. The approach we developed was to combine this information in a new object, called a policy unit. A policy unit consists of the set of objects the policy unit covers and the set of machines holding the objects; this is similar to a partition in the Windows 2000 Active Directory [10] and in Distributed Data Structures [6]. The set of objects can be specified as a list of object identifiers or as an application-specific predicate describing the objects, such as a file system path. Policy units are
Figure 6: The protocol incorporating policy units adds two new states: bulk-pending, and awaiting-replica, which are used when replicating in bulk data and when waiting for a replica set change to replicate in, respectively.

There is the possibility that a node will never receive an updated replica membership after it has been removed from a policy unit. This could prevent updates made before the removal but not propagated from ever being sent to the new replicas. To avoid this problem, when a node is removed from a replica set it doesn’t retire the policy unit change until it has retired all of its updates in the policy unit. This ensures that the node is notified of all changes to the policy unit while it has updates waiting to replicate, even if it isn’t still a member of the replica set.

The Orcia protocol assumes the existence of a membership or failure-detection service in the cluster. When the coordinator for an update learns that a member of the replica set has failed permanently, it may complete the protocol by assuming that the failed machine has replied affirmatively to all messages. The failed machine can be removed from policy units by updating their replica sets (and again assuming that the failed machine has accepted the change). This technique is also used in Porcupine’s protocol [19], but is simplified in Orcia because failed machines can be removed without modifying all the objects stored on the machine.

The addition of policy units to Orcia allows replication metadata to be aggregated, which makes adding and removing replicas more efficient than in Porcupine’s protocol. In addition, the individual replication messages may be smaller, because for updates that don’t overlap with replica set changes, the replica set may be inferred from the replica set version identifier. However, combining objects into policy units limits Orcia’s flexibility to replicate objects to any subset of machines in a cluster.

4.4 Replica Object Changes

The Orcia protocol can provide complete flexibility in placing replicas by allowing objects to be moved between policy units. Moving an object from one policy unit to another allows the load associated with the object to be redistributed, which enables fine-grained load balancing. Moving objects requires changing the set of objects covered by two policy units simultaneously, so Orcia allows a single update to modify multiple objects. The move must be made on a node that is a member of both replica sets because machines normally only have knowledge of replica sets of which they are members.

Orcia’s propagation of data values rather than operations makes it difficult to change multiple objects in a single update for two reasons: first, any updates that are overwritten must be propagated to the machines involved in both updates, and second, updates must completely overwrite an object. As a result, if either of the policy units moving objects have previous move operations that have not yet finished, then the other policy units from those moves must be incorporated in the update. The update
record for a move includes both the policy units performing the move and any policy units transitively related by pending updates for those units. The move update then sets the values for all these policy units, not just those moving objects.

Similarly, when overlapping move operations conflict, the compensating update must include the policy units from both moves. However, the machine resolving a conflict may not have all the objects covered by the policy units involved. As a result, each update to a policy unit must include a list of the objects being moved and which machines store those objects. Compensating updates for move operations include the lists of object locations from the conflicting updates so that the machines that are storing new replicas can copy the objects. Pseudo-code for this protocol is given in [21].

The ability to move objects between policy units gives Orcia the same flexibility in placing objects that Porcupine offered. For example, if the objects in a policy unit grow too large, or are accessed too frequently, then the policy unit may be split by moving objects out of the original unit and into a new one. The result is that applications using Orcia are able to take advantage of the cluster by allowing data to be redistributed dynamically to avoid load imbalances.

4.5 Summary

In summary, the Orcia protocol meets the goals set out in Section 3: i.e., (1) it detects conflicts from simultaneous updates, (2) it stores no per-object metadata, and (3) most importantly, it supports efficient addition and removal of machines from replica sets by centralizing the replica information into policy units. Adding a machine requires updating only a small number of policy units; removing a machine requires the removal of that machine from the replica sets of the policy units it stores. These abilities make Orcia a good match for clusters, because they accommodate the unique features of clusters, such as the frequent change of machines, and the need to redistribute the load of a failed machine dynamically.

The protocol leaves several parameters up to application control. Most significantly, the application can choose what unit of data to treat as an object. Splitting a data structure into multiple objects allows for fewer update conflicts, but may cause extra overhead and consistency problems if many parts of the structure are modified at once. Large objects, though, may require transmitting unnecessary data when only one part of the object is modified. The application can also choose when to send update messages. For stronger consistency or failure resilience, an application may choose to replicate data before returning from a user request, similar to how contexts are used in [22]. In order to amortize load, the application may instead wait for many updates to occur and then transfer them all in a single message. In addition, the application may choose how to resolve conflicting updates to the same object, such as by merging values, discarding one update, or requiring user intervention.

There are many applications that could take advantage of the consistency model supported by Orcia. Beyond electronic mail or bulletin boards, which have a single data model, additional example applications include personalization databases for customizable web sites, personal information managers, and auction web sites. In these applications there are frequent updates to the data and it is important to detect update conflicts.

The Orcia protocol is not suitable for all applications, however. Unlike many database replication algorithms, such as [3], it does not provide transactional consistency or single-copy serializability. Instead, Orcia treats all objects separately. Also, the protocol was not designed for wide-area networks or intermittent connectivity. It was instead targeted at a well-connected cluster in which the machines are connected with a high-speed network (100 Mb/s to 1Gb/s) and partitions are rare, so that propagating updates through a second party is not necessary. Finally, Orcia was not designed to support a large number of replicas for a single data item. Instead, the protocol was designed for a few replicas, on the order of two to five. While the protocol will work for a larger number of replicas, the choice of requiring all machines to cooperate to commit a change leads to more delay in propagating updates when there are many replicas for an object. Nonetheless, Orcia fits the cluster and application environment used for many Internet applications.

5 Evaluation

To evaluate the Orcia protocol, we compare its performance and capabilities to a simple single-master primary-copy protocol as described in [4]. In that protocol, all updates are written into a log at a single machine, the master, and later sent to replica set members. The single-master protocol has the advantage of not requiring conflict detection; it can also support both replica set changes and replica object changes by performing the operations at the
master for the replica set.

In terms of absolute performance, the single-master system is superior to Orcia. In a single-master log protocol, any update, whether to a single object or to a replica set, requires one message and one reply per replica: the message to transmit the update and the reply to receive an acknowledgment. Orcia instead requires three messages and replies per update, which triples the network load. In addition, Orcia must update the log for each of these messages, causing the disk workload to increase similarly. Aggregating many retire and finish messages and only transmitting them only periodically, however, can reduce this penalty. Delaying the retire and finish messages does not impact the correctness of the protocol, but it improves performance significantly because these messages are account for most of the overhead of Orcia.

Despite its higher overhead, Orcia offers several benefits to applications. The greatest advantage is the ability to load balance writes across multiple machines, which may not be possible with a single-master log system. This capability is important if policy units have widely varying workloads. In addition, Orcia provides more flexibility in moving objects between machines, because changes to policy units do not need to be made on the master for both replica sets. Finally, in the case of a failure of the master, Orcia does not need to detect the failure to progress, because the data may be written at any replica. There is also no need to run a consensus protocol, as was used in Echo [20], to elect a new master after a crash. Therefore, while single-master protocols naturally support stronger consistency models, Orcia’s increased flexibility may be beneficial for cluster-based applications not requiring such strong semantics.

6 Related Work

Replication has a long history and has been applied to databases, file systems, web servers, and special-purpose applications. However, in most situations replication has been used to make services available, rather than making them scale to larger workloads. For write-intensive workloads, replicating a data set may not improve scalability at all if all data writes must be performed on all replicas [5].

Several systems have used replication to provide scalability for read-intensive workloads. The Windows 2000 Active Directory [10], for example, allows the directory service to be run on multiple machines, each of which contains a full copy of the data. The data may be updated at any machine and is then propagated to other replicas. The service, however, was designed for workloads in which reads are much more common than writes, and doesn’t support moving objects between partitions.

Distributed Data Structures [6] provides scalability by partitioning data. It uses a hash table to place data into partitions, which may then be replicated by several machines. Each machine may also host several partitions. The result is very similar to Orcia, except that the system does not support arbitrary data topologies. Partitions may be split into pieces but objects may not be arbitrarily moved between partitions.

Harp [2] scales replication up to large numbers of nodes using weak consistency and hierarchically structured replica sets. Harp also supports lightweight addition and removal of replicas, but differs from Orcia in its focus on large numbers of replicas rather than large numbers of replica sets.

Replication for web servers, such as such as [16], allow objects to be moved between replicas. However, the focus of these systems is to place objects close to clients rather than distribute load within a cluster, and they generally allow updates only at the primary copy of an object.

Replicated databases, such as those from Oracle [11] and Microsoft [1], support similar features such selectively replicating portions of a table to different replicas. Microsoft SQL Server supports changing the set of objects replicated to a machine. However, in these databases there must be a machine that contains the entire table, so the only changes that can be made are to the set of replicas and the subsets of the table they store.

In the field of optimistic replication, Orcia draws heavily on many previous algorithms. The use of version vectors for conflict detection was pioneered by Locus [12], and used in Ficus [15] and Coda [8]. Bayou [13] offers lightweight replica addition similar to Orcia, but was designed to provide availability in a network with mobile hosts. Our work complements these protocols by providing high availability with low overhead in a cluster environment.

7 Conclusions

This paper describes a new replication protocol, Orcia, that is designed for applications running on a large-scale computer cluster. In particular, Orcia is intended for a system in which data is distributed across a large number of processors and requires both high availability and scalability to millions of
clients. The key features of the protocol are:

- Version vectors are used to detect conflicts but
do not need to be stored indefinitely.
- Objects are clustered into policy units for com-
  bining replication state.
- Objects may be efficiently moved between repli-
cas for load balancing.
- Adding and removing replicas is a lightweight
  operation.
- No replication metadata must be stored on ob-
  jects persistently.
- The properties of the protocol in [19], which
  include eventual consistency, update at any
  replica at any time and minimal overhead, are
  preserved.

These features combine to make Orcia a compelling
protocol for large applications by allowing fine-
gained load balancing across a large number of ma-
machines and by allowing write-intensive traffic to scale
with the size of the cluster. In particular, Orcia en-
ables sets of objects to be replicated to overlapping
groups of machines, spreading the data workload
evenly across the cluster.

Acknowledgments

We thank Yasushi Saito for his work on the Porcu-
pine protocol and his help with this paper. Suzanne
Swift, Robert Grimm, Joshua Redstone and David
Ely provided valuable comments on earlier versions
of this paper.

References

    microsoft.com/ml.
    algorithms in replicated databases. In Proceedings of the 16th
    ACM Symposium on Principles of Database Systems, pages
[4] P. A. Bernstein and E. Newcomer. Principles of Trans-
    ACM SIGMOD International Conference on Management of Data, pages
    173–182, Montreal, Canada, June 1996.
    D. Culler. Scalable, distributed data structures. In Pro-
    ceedings of the 4th USENIX Symposium on Operating Systems
    cation Techniques in Distributed Systems. Kluwer
    operation in the Coda file system. ACM Transactions on
[9] P. J. Leach and R. Salz. UUIDs and GUIDs. Internet
    Draft draft-leach-uuids-guids-01.txt, Internet Engineering
    paper, Oracle Corporation, Redwood Shores, Califor-
    documents/adv_replication_twp.pdf.
[12] D. S. Parker et al. Detection of mutual inconsistency on
    distributed systems. IEEE Transactions on Software
    Theimer, and A. J. Demers. Flexible update propag-
    ation for weakly consistent replication. In Proceedings of the 16th
    ACM Symposium on Operating Systems Principles, pages
    1995.
    Heidemann. Replication in Ficus distributed file sys-
    tems. In Proceedings of the Workshop on Management of
[16] M. Rabinoch, I. Rabinoch, and R. Rajaraman. Dy-
    as ATT Labs Technical Report HAN177000-980005-01-
    TM.
    sion-vector maintenance. Technical Report CSD-97-0022,
    UCLA, June 1997.
[18] Y. Saito, B. N. Benhad, and H. M. Levy. Manageable-
    ness, availability and performance in Porcupine: A highly
    scalable Internet mail service. In Proceedings of the 17th
    ACM Symposium on Operating Systems Principles, Ki-
    awah Island Resort, South Carolina, Dec. 1999.
    internet data services. In Proceedings of the 4th Interna-
    tional Symposium on Distributed Computing, Toledo,
    homes/yasushi/disc.ps.gz.
[20] G. Swart, A. Birrell, A. Hiegen, and T. Mann. Avail-
    ability in the Echo file system. Technical Report 112,
    Digital Equipment Systems Research Center, Palo Alto,
    2000.
[22] H. Yu and A. Vahdat. Design and evaluation of a con-
    tinuous consistency model for replicated services. In Pro-
    ceedings of the 4th USENIX Symposium on Operating
    Systems Design and Implementation, San Diego, CA,