OSCAR: An Architecture for Weak-Consistency Replication

Alan R. Downing, Ira B. Greenberg, and Jon M. Peha
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025

Abstract

This paper presents an architecture for providing weak-consistency replication for databases in an internetwork. It is designed to make the databases highly available and to operate reliably under difficult conditions such as unreliable communication, low-bandwidth communication, network partitions, and host failures. Updates are stored in logs until they have been propagated to all database sites and properly delivered to the databases. A new approach called *mediation* is used to provide integrated support for reliable replication and log purging. Other interesting features include requiring minimal support from database management systems, support of multiple weak-consistency methods, and easy tuning of the architecture's basic algorithms to particular environments.

1 Introduction

Database replication is an important technique for building distributed systems in an internetwork. It can be used to improve system performance, the availability of data, and the survivability of data. A replication system must ensure that all updates are propagated to all database copies, and that all copies are consistent.

Both update propagation and consistency are difficult to achieve in an internetwork, because internetworks often contain high-latency, low-bandwidth links, and complicated fault modes including transmission failures, node failures, and partitions. Update propagation mechanisms must be efficient and reliable while tolerating these fault modes. Consistency mechanisms must balance the availability and consistency requirements of applications, despite low transaction throughput caused by internetwork links and network partitions. In an internetwork, weak-consistency methods are often preferable to strong-consistency methods. For example, highly-available databases in airline reservation systems and military command and control systems cannot use strong-consistency methods, such as primary copy [1] or quorum consensus [2], because the availability of data would be too limited. Weak-consistency methods provide higher availability for the databases at the expense of temporary inconsistencies among the database replicas.

There are numerous weak-consistency methods. Many of them are based on the semantics of applications, such as the knowledge that certain data items will only be incremented or decremented. When weak consistency is used, multiple copies of a database can be independently updated. These updates are then propagated to the other replicas. When they arrive, the application semantics are used to uniquely reconcile any inconsistencies that may have occurred. Mutual consistency will eventually be achieved when all updates have propagated throughout the system. Because all data items do not have the same semantics, the use of any particular semantic weak-consistency method will force tradeoffs to be made among 1) the availability of the database for writing, 2) the availability of new updates for reading, and 3) the accuracy of the contents of the database.

This paper describes our Open System for Consistency and Replication (OSCAR), an architecture for a weak-consistency replication system. OSCAR's objectives are to efficiently provide reliable replication in systems that are subject to the failure modes mentioned above, to efficiently provide appropriate weak consistency for a variety of common application semantics, and to be appropriate for a wide range of application semantics, database management systems (DBMSs), and system configurations.

Our approach to reliable replication is to use two complementary mechanisms. When an update initially enters the replication system, it is unreliably pushed to all other replicas. A novel technique, called mediation, ensures reliable propagation of the updates throughout the system. OSCAR provides appropriate weak consistency for a variety of applications by simultaneously supporting several weak-consistency methods. Thus, an appropriate method can be assigned to different data items. Three sample methods that can be used are described in Section 5. OSCAR is flexible and adaptable in a variety of ways. Its modular design permits its components to be configured for different networks, and allows its algorithms, such as the consistency methods, to be selected and tuned for different environments and applications. In addition, OSCAR can be used with many database management systems because it requires no DBMS modification, and makes minimal assumptions about internal DBMS

CH2728-4/90/0000/0350$01.00 © 1990 IEEE

structure and mechanisms.

The paper is organized as follows. First, we present the assumptions, fault modes, and objectives on which OSCAR is based. Then, we describe the major components of OSCAR and their functions and organization, followed by a brief discussion of policies for managing updates. We then describe the current status and future work. We conclude with a discussion of OSCAR's features. This paper concentrates on the functionality provided by the architecture, not the design options that can be used to provide this functionality or a specific system based on this architecture.

2 Assumptions and Failure Modes

OSCAR is based on the following assumptions. We assume that no byzantine failures occur [3], and that, in particular, all host failures are fail-stop.[4] This means that there are no malicious agents deliberately sending out erroneous messages, and that system components do not break in such a way that they emit random messages. Our only assumption about communication among hosts is that it is commutative and transitive. We assume that the schemas for replicated databases are static. Some consistency methods use timestamps, and the degree to which clocks are synchronized limits the correctness these methods can achieve. We assume that clock synchronization is provided by an external mechanism. Finally, we assume that updates will enter OSCAR in their logical order. This means that a site that originates an update with timestamp t_1 cannot later originate an update with timestamp t_2, where $t_2 < t_1$.

The following assumptions were also made for this paper. We assume that the configuration of the database servers and OSCAR's components is static. In addition, we assume that a relational data model is used, and that a database is fully replicated at each database site. Finally, we assume that compensation [5] is not required. This means that an application does not perform any actions, due to an incorrect database state, that later requires special actions beyond correction of state.

Note that a network partition does not necessarily divide the users into isolated groups; they may still be able to communicate using an outside system. In addition, there is no restriction on when a partition can occur. Finally, data is replicated and transactions are allowed to run in all groups of hosts, even if they are isolated by a partition.

OSCAR must be able to tolerate transmission failures, node failures, and temporary network partitions. It also must be able to handle transmission faults that result in lost messages or out-of-order receipt of messages. All temporary and some permanent node failures can be tolerated. Although it is possible to lose some updates if permanent failures occur, the remaining replicas will maintain mutual consistency. OSCAR can be configured to tolerate an arbitrary number of permanent failures, without inconsistencies. OSCAR must continue to function as long as at least one node is operational. It can tolerate temporary network partitions, even if there is no time at which the network is not partitioned at some location. Although it is not possible to support replication and consistency between permanent network partitions, OSCAR must support them within each partition. Partitions that last beyond some user-defined period must also be reported to the user. Because timely detection of partitions is difficult and expensive, no algorithm should depend on this function for operation.

3 Objectives

OSCAR's architecture was designed to meet three primary objectives: fault tolerance, flexibility, and efficiency. In particular, OSCAR was designed for environments in which fault tolerance is essential for operation, and must tolerate the system failure modes discussed in the previous section.

In addition, OSCAR's architecture is designed for flexibility so that it can operate efficiently with a wide variety of network systems and applications. Different applications require different performance constraints. For example, they may stress availability, data integrity, or speed of information flow. These goals must be balanced with the constraints of the underlying system, such as bandwidth or memory. Also, many applications do not require completely general database semantics. For example, a database of physical observations will generally have transactions that read from or write to the database, but not both. Performance can be improved significantly by offering different services and consistency methods for different databases or portions thereof.

OSCAR is also intended to work with many existing commercial or custom DBMSs. Therefore, OSCAR exists outside of the DBMS, requires no modifications to the DBMSs, and makes few assumptions about their internal organization or mechanisms.

Not only must OSCAR work with different DBMSs and applications, it must be adaptable to different networks: that is, it must be possible to configure an OSCAR implementation so that it operates efficiently over the intended network. Network topology can dramatically effect performance. For example, if two sets of networks are connected by a single low-bandwidth link, the replication system must be configured to minimize traffic over that link. If processing is typically the system bottleneck, the replication system should use the information from any existing monitoring mechanisms to divert more of its processing to machines with lighter loads.

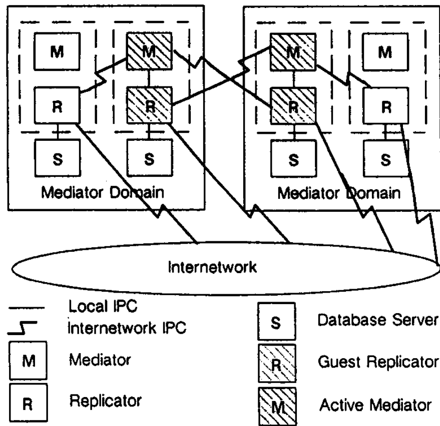

Figure 1: Architecture of OSCAR

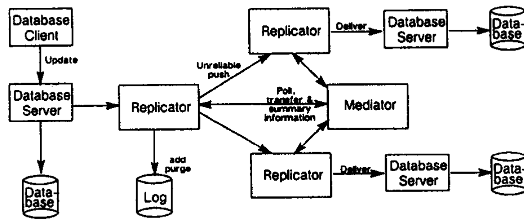

Figure 2: Information Flow in OSCAR

4 Architecture

OSCAR is based on two cooperating agents called *replicators* and *mediators*, which work together to provide replication and consistency for a set of database servers. Each mediator is paired with a replicator, and collections of these pairs are organized into domains. The domains are then associated with each other as neighbors. At least one mediator must be active in each domain; the others can be dormant. The basic architecture, with two domains, is illustrated in Figure 1.

Information flows through the system as shown in Figure 2. A database client application enters an update into the database server. The update is sent to the local replicator through any mechanism supported by the DBMS, such as a trigger or an audit file and translator. The replicator sends the update to all other replicators responsible for copies of the database using an unreliable multicast. Any replicators that do not receive the originating replicator's initial push of the update will eventually be updated by another replicator, as determined by the mediator. Each receiving replicator delivers the update to its associated database according to the appropriate consistency method. Finally, the replicator purges unneeded updates from its log according to information collected, summarized, and distributed by the mediator.

The remainder of this section discusses the high-level functionality of replicators and mediators. A more detailed description of the associated policies will be presented in the next section.

4.1 Replicators

Replicators provide replication and ensure consistency for a database server. To simplify the discussion, we will assume that each replicator is assigned only a single database server. The replicator maintains a log of updates for its server until they have been properly replicated and delivered. It is assisted by the mediators in reliably propagating updates and knowing when entries can be purged from its log. A group of replicators manage the replicas of a given database. A replicator:

- attempts to push updates that originate at its server to other replicators.
- maintains a log of database updates.
- enforces weak consistency by appropriately delivering replicated updates to its associated database.
- attempts to forward old updates to other replicators, as instructed by a mediator.

These functions will be discussed in detail in the sections that follow.

Initial Push When a replicator receives an update that originated at its associated database server, it makes an unreliable "best effort" to push the update to the other replicators responsible for copies of the same database. Because OSCAR makes no assumption about the mechanism that pushes the update to the other replicators, any available multicast facility can be used. Each receiving replicator places the update in its update logs and then delivers it to its associated database according to the appropriate weak-consistency method. A reliable push is not used because it would require a possibly unlimited number of retransmissions for each update, which would not be practical in networks where long partitions can occur. Furthermore, a reliable push cannot tolerate failure at the replicator that is pushing the update. Instead, the reliability of update propagation is ensured by the mediators.

Maintaining the Update Log Each replicator maintains a log of update information that may be needed to complete replication or provide consistency. This is a central concept in the design of OSCAR. The logs contain database updates and other descriptive information that can aid in replication or consistency decisions. Examples of this additional information include version information and timestamps. When updates are replicated, their descriptive information will be replicated as well.

A replicator receives update information about initial database updates from its associated database server, and update information about replicated updates from other replicators. This update information can be obtained by any appropriate mechanism that is supported by the DBMS, such as triggers or an update log. The replicator's log-management duties include storing new update information in the log, maintaining the delivery and purge status of log entries, and purging obsolete log entries. The replicator cannot purge a log entry until 1) it has been delivered to the replicator's associated database, 2) it can no longer affect the consistency of the replicator's associated database, and 3) it is no longer needed for replication. Each replicator decides independently when these criteria have been met.

Version information consists of a replicator identifier and a sequence number. A replicator assigns a sequence number to an update when the update first enters the replication system; that is, when the update is received from the replicator's associated database server. The version information is used to uniquely identify updates, and to construct version vectors.[6] Version information can be used to filter out duplicate log entries that may result from replication or unreliable communication. Because the replicator assigns a timestamp to an update when it enters the replication system, and thus out-of-order timestamps cannot be created at a site, the timestamp order for two updates from the same originating site will be identical to the updates' sequence number order. Timestamps are needed only in consistency methods that require the comparison of entry times for updates that originated at different sites.

Version vectors are an important concept in OSCAR, because they are used to summarize collections of updates. For example, they can be used to efficiently identify updates that have not been received by a replicator, and the entries that can be purged from a log.

A variety of purge algorithms can be used to remove obsolete entries as described in Section 5.2. These algorithms try to keep the logs highly purged and work well with the sample weak-consistency methods discussed in Section 5.1.

Consistency The replicators are also responsible for ensuring that the replicated databases remain consistent, according to the appropriate weak-consistency methods. They do this by independently delivering updates, in the right order, from their update logs to their associated database. These actions, which are completely distributed, will eventually result in mutually consistent databases.

Because a replicator can simultaneously support multiple weak-consistency methods, an appropriate method can be assigned to different parts of the database. For example, a different method could be assigned to each table that will be replicated. Each weak-consistency method is implemented by a delivery algorithm, which indicates 1) which updates need to be delivered from the update log to the database, 2) the order in which they should be delivered, and 3) when they should be delivered. An extensive survey of consistency methods has been published by Davidson.[7]

Forwarding Old Updates The fact that an update has not reached all replicas of a database will eventually be discovered by one or more mediators. Depending on the information available to them and the policies implemented in their transfer algorithms, the mediators will ask one or more replicators with an update to forward it to replicators that have not received it. If a replicator receives a request to send an update to multiple sites it should try to do so with an efficient multicast. The update can be sent unreliably, because mediators will eventually detect any updates that are still missing.

4.2 Mediators

Mediators are used to efficiently collect and exploit global information about the state of replication. Their main purpose is to efficiently provide reliable replication by acting as a backup to the initial unreliable push of new updates. A mediator is powerful because it is a centralized component in a *domain*, where a domain is a predefined collection of replicators and databases. A system contains one or more domains. Fault tolerance for an active mediator is handled by the activation of dormant mediators, by a stateless design, and by the use of algorithms that are not affected by transient system states and lost messages. Although mediators are stateless and all information needed for their decisions can be obtained from each poll they make, mediators can store some information as hints to improve efficiency. Mediators are responsible for:

- collecting, summarizing, and distributing the global state of their domain.
- determining which replicators are missing updates, and how update information should be transferred to them.

These functions will be further discussed in the sections that follow.

Mediator Domain State An active mediator periodically polls the replicators in its domain and a configurable

number of replicators from each of its neighboring domains for information, such as version vectors, to obtain a snapshot of their state. The version vectors are used to indicate the set of updates that have been received or delivered by the replicator. This information can be used to determine which replicators are missing update information, and to guide transfer and purge decisions. If a replicator does not respond to a poll, a mediator can use the replicator's previous response. If a replicator does not respond for some specified length of time, it is presumed that the replicator has permanently failed or been partitioned away, and an administrator is notified. Once a round of information has been collected, a mediator summarizes the information and sends the summaries back to the replicators that were polled in its domain.

Update Mediation Mediators perform the important task of making replication reliable. A mediator acts as a third party causing missing updates to be transferred between replicators. Using the information that it has gathered, a mediator can efficiently decide which updates each of its replicators is missing, and where those updates can be transferred. There will usually be several sources for each update. Thus, the transfer algorithm can be designed to implement a variety of policies, such as improving the transfer rate, network traffic, processor utilization, or reliability. Because mediators are centralized, these policies can provide a form of load balancing, which is most beneficial when a network partition merges and many updates may need to be transferred. Finally, the transfer instructions are sent to the appropriate replicators. The transfers can be unreliable because the mediator will detect any updates that remain missing in a future poll. A specific transfer algorithm is discussed in Section 5.3.

Mediator Fault Tolerance It is important to provide fault tolerance for the mediators, because they are a centralized component. The host on which an active mediator is running may fail or a domain may become partitioned. In the steady state, there will be one mediator active in every domain or partition of a domain.

A mediator runs on every host where a replicator is located. For efficiency, only one mediator is active in each domain, although no harm will occur if several are active. This contrasts with many other election algorithms that require global agreement before proceeding.[8] Each mediator has a unique priority that reflects the extent to which it is the best choice to be the active mediator. In the steady state, the mediator with the highest priority will be active and the rest dormant. If a replicator is not polled by a mediator with a higher priority within a priority-dependent time period, its associated mediator will become active. Otherwise, the mediator will become or remain dormant. This approach is feasible, because mediators are stateless.

4.3 Mediation Domains

OSCAR can be configured so that its mediator/replicator pairs are grouped into multiple domains. These domains are then connected as neighbors. To perform its functions, each mediator contacts the replicators in its domain plus n replicators in each neighboring domain, where n is chosen to meet fault tolerance requirements. Note that the specific replicators contacted in a neighboring domain can be different for each mediation round. This is important for fault tolerance, in case one of the replicators fails or is partitioned away.

The use of domains improves the scalability of the system by removing global responsibilities. It enables communication to be localized into natural neighborhoods in the internetwork, and enhances concurrency. It also makes it possible to mark a log entry for purging as soon as it has been replicated throughout a domain and partially into all neighboring domains, instead of throughout the entire system. This helps reduce the size of the logs.

A mediator is responsible for ensuring that updates are propagated to the replicators that it contacts, whether inside or outside its domain. Sometimes an update can miss an entire domain during the initial push phase. These updates are obtained from the replicators contacted in a neighboring domain. Some updates that were not successfully pushed may have to pass through several domains until they are completely replicated.

Only mediators need to know about other domains; the replicators do not. A mediator collects status information from all of the replicators that it contacts. However, it only returns summary information indicating which updates can be purged to the replicators in its domain.

5 Update Management Policies

So far, we have described the components of OSCAR and the basic structural framework used to provide reliable replication. This section describes the three ways a system designer can select policies to manage updates. These policies control the delivery of updates to databases, the purging of updates from update logs, and the transfer of updates to sites that missed them. Based on these choices, the system designer can tailor the system for a specific application or network.

OSCAR provides a framework that will perform the basic chores required for reliable replication and system fault tolerance. These include assigning sequence numbers and timestamps to updates, pushing initial updates throughout the system, storing updates in logs, detecting missing updates, and providing fault tolerance for system components. Some important adjustments can be made during system configuration, such as determining, the placement of system components, the membership of domains, the neighbor relation among domains, and the number of repli-

cas to contact in a neighboring domain.

This framework supports the three types of update management policies mentioned above. After analyzing the needs of the application and the requirements of the environment in which it will operate, appropriate modules can be selected and filled in for update delivery, purging, and transfer. These policies will now be discussed in more detail.

5.1 Update Delivery and Weak-Consistency Methods

A weak-consistency method represents a tradeoff between database consistency and availability. The choice of a method depends on the semantics and requirements of the application. A method specifies the rules governing when updates can be delivered to a database, and thus how to resolve conflicts. In OSCAR, weak-consistency methods are implemented by delivery algorithms. This means that a specific form of weak consistency is achieved by controlling the order and timing of update delivery to a database. Three weak-consistency methods called Commutative and Associative, Overwrite, and Site-Sequential will now be described. These methods would be assigned to different parts of the database according to the update semantics of the data.

Commutative and Associative For some applications, all updates are commutative and associative. This means that the final state of the database will be the same regardless of the order in which they are delivered, and that intermediate values do not have to correspond to correct values. For example, operations on certain tables may be restricted to updates that perform only multiplication and division, or only addition and subtraction.

A replicator can deliver a commutative and associative update as soon as it is received, and mark it for purging with respect to consistency. Although the original update may not be a commutative and associative update, it is often possible to derive an equivalent set of commutative and associative updates from information in the DBMS audit logs.

Overwrite The overwrite consistency method is intended for transactions that only use absolute updates, and do not contain relative or conditional terms. For example, "LET X = 3" is an overwrite transaction, but "IF Y = 10 LET X = 3" and "LET P = P + 3" are not. Overwrite is more appropriate for "image" or physically observable data, such as the current location of a plane. as described by Garcia-Molina.[9] It is similar to weak-consistency mechanisms discussed in [10, 11, 12].

A replicator can deliver an overwrite transaction as soon as it arrives, but if the newly-arrived transaction has been superseded by a previously-delivered transaction then it can be ignored. If the newly-arrived transaction has an older timestamp than other previously-delivered, conflicting transactions, then transactions must be redelivered, in timestamp order, from the log. This ensures that updates are eventually delivered in the correct order. If the key, database name, and table name of the record being updated can be obtained, then it is possible to check whether a new transaction has been superseded or whether previously delivered (but newer) transactions conflict. Otherwise, it must be assumed that the new transaction must be delivered and that all newer, previously delivered transactions conflict. An overwrite transaction can be considered purgeable with respect to consistency, when all older transactions have been accounted for and delivered, and it has been delivered. Once again, although the original update may not be an overwrite update, it is usually possible to derive an equivalent set of overwrite transactions from information in the DBMS audit logs.

Site-Sequential
The site-sequential consistency method requires all transactions initiated at a specific replicator to be delivered in sequence number order. Transactions initiated at different replicators, however, are independent, and can be delivered in any order. This method is useful in a system where each site writes only local information but must be able to read all information. A replicator must not deliver a site-sequential transaction from a site until all previous transactions from that site have been delivered. One advantage of site-sequential consistency is that a site can issue conditional transactions as long as data referenced in the transaction is controlled by that site.

5.2 Update Purging

Obsolete entries should be removed from update logs as soon as possible, so that processing time and memory are not wasted. A replicator can remove a log entry when it is not needed for replication, has been delivered, and is no longer needed for consistency. The choice of an effective purge algorithm depends on the consistency methods being used. It is possible to detect obsolete updates, because updates receive their timestamps as they enter the system, and therefore it is possible to determine when all relevant updates with smaller timestamps have been delivered. This is in contrast with SHARD [13, 14], which allowed the creation of out-of-order timestamps. As a result, SHARD was forced to use complex protocols to purge update logs. This caused an increase in memory and communication-bandwidth requirements. We believe that such out-of-order updates should be handled by the applications, not by the replication and consistency system.

This section describes two efficient purge algorithms that work well with the previously-described consistency

methods. Both of these algorithms rely on a mediator to summarize the state of the replicators in their domain, but the actual purge decisions are made in a distributed fashion by each of the replicators. It is not important if a replicator misses a summary, because the next summary will contain a superset of the information in the previous one. Both of these algorithms require a version vector, called a summary version vector, from the mediator that indicates the set of updates that have reached all replicators in the domain.

Version Vector Purge Version Vector Purge can be used when the sequence order between updates that originated at different sites is not important, as in the Site-Sequential and Commutative and Associative consistency methods. When a replicator receives the summary version vector from a mediator, it can flag as purgeable due to replication any log entry whose sequence number is less than or equal to the corresponding component in the summary version vector, because these entries have been received by all replicators in the domain and the correct number of replicators in each neighboring domain. A log entry can be deleted after the replicator has also delivered it to its associated database and marked it as purgeable due to consistency.

Timestamp Purge Timestamp Purge can be used when the sequence order between updates that originated at different sites is important, as in the Overwrite consistency method. Each replicator calculates the earliest time t before which it has received an update that originated at every replicator. When a replicator receives the summary version vector from the mediator, it can flag as purgeable due to replication any log entry 1) whose sequence number is less than or equal to the corresponding component in the summary version vector, and 2) whose timestamp is less than t. Entries with a newer timestamp cannot be marked, even if they have been received by all replicators in the domain and the correct number of replicators in each neighboring domain, because a conflicting update with an earlier timestamp could still appear. A log entry can be deleted after the replicator has also delivered it to its associated database and marked it as purgeable due to consistency.

Timestamp Purge cannot distinguish between a network partition and a replicator that has not sent out an update recently. To overcome this problem, a replicator that has been silent for a specified time-out period must send a "null" message that contains a current timestamp.

5.3 Update Transfer

A transfer algorithm is used as part of the mediation process to decide which replicators will be asked to forward updates to other replicators. Because there often are several sources for each update that needs to be transferred, it can be used to implement some system policy such as minimizing network traffic or forwarding delay, and can be tuned to a particular environment. It can be used to achieve load balancing according to a variety of metrics, and can take advantage of any system status information that is available.

For example, consider a transfer algorithm that tries to minimize message traffic. The optimal solution of this problem maps to the Minimum Cover problem, which is NP-complete [15], so the following approximation could be used. For each replicator that is missing updates, the replicator that can forward the greatest number of those updates is selected. If several replicators are eligible, one is selected randomly. This process is repeated until transfer instructions are defined for all missing updates. The transfer instructions are then sent to the selected replicators, who forward the updates directly to the target replicators.

6 Current Status and Future Work

We have completed the basic architecture of OSCAR and are developing a prototype system. To help us understand the implications of the many design alternatives available, we are developing a simulation to enable us to analyze the system's performance with various parameters and algorithms. In the future, we plan to study ways to extend the basic architecture, such as permitting dynamic configuration and increasing scalability. In addition, we plan to study alternative transfer algorithms, examine other weak-consistency methods, and investigate the use of OSCAR as the basis for a reliable internetwork multicast system.

OSCAR is being developed for the Army Distributed Command and Control Program (ADCCP), whose purpose is to demonstrate a survivable, distributed command and control system. ADCCP's basic environment consists of Ethernets with Sun workstations, connected by packet radios, satellite links, the ARPAnet, or phone lines. The system must tolerate links with low throughput, high error rate, or high delay. In addition, it must survive multiple failures and network partitions. ADCCP provides resource management [16], a network operating system [17], location transparency, replicated databases, and a replicated name server. Several distributed database applications have also been developed. ADCCP has been operational for five years.

7 Discussion

This paper has presented an architecture for providing weak-consistency replication for databases in an internetwork. The architecture is designed to provide fault toler-

ance in spite of frequent host or communication failures, to operate efficiently when providing both replication and consistency, and to be very flexible so that it can be effectively adapted to a wide variety of applications, networks, and databases. New ideas in OSCAR include the use of mediation to provide reliable replication, and the ability to simultaneously support multiple weak-consistency methods.

OSCAR can tolerate multiple host failures, communication failures, and network partitions. If an active mediator is lost or partitioned away, a dormant mediator will automatically take over. Replication and availability within a domain are not affected if a replicator is lost in the domain, and if a replicator fails in a neighboring domain, any other replicator in that domain can be used in its place. Lost updates are propagated by mediation, and lost status information is re-sent during the next mediation round. Partitions can occur at any time, and need never be detected for system operation. Updates will propagate through the system even if the system is never completely up. As long as an update has not been lost by permanent failure, mediation will propagate it throughout the system.

Efficient approaches are used in OSCAR to provide replication, consistency, and purging. Updates are pushed to multiple destinations with an unreliable multicast. $O(n)$ messages are used to exchange status information, so that missing updates can be detected and purge decisions can be made, where n is the number of replicators polled by a mediator, in contrast to other systems that require $O(n^2)$ messages.[5] Missing updates and their locations are quickly detected by the use of version vectors. The transfer algorithm enables the mediator to take advantage of system status information to increase the performance of the system by performing load balancing based on critical resources, such as bandwidth or processing capabilities. The transfer algorithm can also alter the distribution of this load, to avoid bottlenecks or favor underutilized resources. Multiple mediators can operate in parallel to propagate missing updates. The delivery algorithms that implement weak-consistency methods contain simple rules indicating how updates should be delivered to a database. Finally, update logs are purged incrementally and kept relatively empty, to save processing time and memory.

OSCAR provides flexibility in a variety of ways. By simultaneously supporting multiple weak-consistency methods, it allows an appropriate method to be selected for each portion of the database. This makes it possible to match the consistency method to the application semantics. Multiple purge algorithms can also be supported to complement the consistency methods. The components of the system can be grouped into domains and the domains can be assigned as neighbors to match the characteristics of the internetwork. In addition, other parameters can be adjusted, such as mediator priorities and the number of replicators to contact in a neighboring domain. The transfer algorithm makes it possible to adapt to information about the status of the system, and to implement a variety of system policies that can focus on improving a wide variety of metrics. Finally, OSCAR can be used with many DBMSs because it requires no DBMS modification, and makes minimal assumptions concerning available DBMS features.

8 Acknowledgments

The authors would like to thank Dr. Michael Davis and Mr. Craig Seidel for their help during the preparation of this paper. We would also like to acknowledge the U.S. Army Communications and Electronics Command, which funded this work under Contract DAAB07-86-D-A035, Delivery Order 0077.

References

[1] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES," *IEEE Transactions on Software Engineering*, Vol. SE-5, No.3, May 1979.

[2] Herlihy, M. P., "General Quorum Consensus: A Replication Method for Abstract Data Types," Technical Report *CMU-CS-84-164*, December 1984.

[3] Lamport, L., Shostak, R., and Pease, M., "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382-401, July 1982.

[4] Schlichting, R. D. and Schneider, F. B., "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *IEEE Transactions on Computer Systems*, Vol. 1, No. 3, April 1983, pp. 222-238.

[5] Sarin, S. K., Blaustein B., and Kaufman, C., "System Architecture for Partition-Tolerant Distributed Databases," *IEEE Transactions on Computers*, Vol. C-34, No. 122, December 1985, pp. 1158-1162.

[6] Parker et al., "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 9, No. 3, May 1983.

[7] Davidson, S. B., Garcia-Molina, H., and Skeen, D., "Consistency in Partitioned Networks," *ACM Computing Surveys*, Vol. 17, No. 3, September 1985.

[8] Garcia-Molina, H., "Elections in Distributed Computer Systems," *IEEE Transactions on Computers*, C-31, 1982, pp. 48-59.

[9] H. Garcia-Molina et al., "Data-Patch: Integrating Inconsistent Copies of a Database After a Partition," *Proceedings of 3rd IEEE Symposium on Reliability in*

Distributed Software and Database Systems, 1983, pp. 38-48.

[10] Allchin, J. E., "A Suite of Robust Algorithms for Maintaining Replicated Data using Weak Consistency Conditions," *Proceedings of the Third Symposium in Distributed Software and Database Systems*, October 1983.

[11] Fischer, M. J. and Michael, A., "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," *Proceedings of the ACM SIGACT-SIGMOD Symposium Principles of Database Systems*, 1982, pp. 70-75.

[12] Wuu, G. T. J. and Bernstein A., "Efficient Solutions to the Replicated Log and Dictionary Problems," *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, August 1984.

[13] Sarin, S. K., Kaufman, C., and Somers, J. E., "Using History Information To Process Delayed Database Updates," *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, pp. 71-78.

[14] Sarin, S. K. and Lynch, N. A., "Discarding Obsolete Information in a Replicated Database System." *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, January 1987.

[15] Karp, R. M., "Reducibility among Combinatorial Problems." *Complexity of Computer Communications*, R. E. Miller and J. W. Thatcher editors, Plenum Press, New York, 1972, pp. 85-103.

[16] Skinner, G., Wrabetz, J., and Schreier L., "Resource Management in a Distributed Internetwork Environment," *ACM SIGCOMM 87, Computer Communication Review,* Vol. 17, No. 5, August 1987.

[17] Davis, M., Schreier, L., and Wrabetz, J., "A Processing Architecture for Survivable Command, Control, and Communications," *Conference Record of 1987 IEEE Military Communication Conference*, October 1987.