The Bengal Database Replication System

T. Ekenstam, C. Matheny, P. Reiher, G.J. Popek

Abstract

Mobile computers often suffer from limited connectivity, or even complete lack of network access. Moreover, in wireless networks some machines are often more accessible to mobile computers than others are. Ideally, nomadic users want the same quality and speed of database access as when they are well connected to the network. Replicated databases meet this need by allowing each mobile machine to carry its own copy of key data. But existing replicated systems are not well suited for all nomadic scenarios. Certain important mobile computing situations require optimistic peer-oriented database replication. The Bengal Database Replication System has these characteristics and was designed to operate in difficult nomadic conditions. This paper presents the design assumptions of the system, describes its architecture, presents performance data on its operation, and discusses future enhancements for the system. The paper also contrasts the system to other replicated databases, concentrating on their suitability for nomadic computing. The Bengal technology can play an important role in the development of highly scalable, highly available, fault-tolerant database systems.

Keywords: distributed, optimistic, database, replication, peer-to-peer

1. Introduction

The central function of a distributed database system is to provide access to data while maintaining the integrity and consistency of that data. The system must also have the ability to support large numbers of users without sacrificing performance. Higher reliability and availability in the presence of equipment and network failures are also requirements for mission-critical enterprise data systems. These requirements are often at odds with each other, leading to solutions that compromise between availability, consistency, scalability and performance.

Nomadic computing exacerbates these problems. In a nomadic scenario, some users of the database travel from place to place, experiencing varying connectivity to other participants. In some cases, they are completely disconnected from all other users. In other cases, cliques of nomadic users have good connectivity to each other, but no connectivity to any other machine. Sometimes bandwidth to connected machines is cheap and plentiful, sometimes expensive and scarce. Yet the nomadic user should, ideally, experience high quality database access regardless of where he goes and what network connectivity is available there.

The central problem is supporting shared database access for multiple nomadic users when some of them are poorly connected or disconnected. In this situation, requiring all users to connect to a single central server will not meet the requirements. The system must allow the nomadic users to access at least parts of the database locally. Several solutions exist for providing such access.

One solution uses a client/server architecture in which a central server coordinates the data updates initiated by the clients. The server machine is typically a large, well-connected, non-nomadic machine, while some or all of the clients are less well-connected nomadic machines. Clients do not store any of the data locally except for a cache of the data currently being manipulated. For the client to access any data it must connect to the server and submit a request. A single server can only handle a finite number of clients before performance is adversely affected. If the server is down or the connectivity between the client and the server is lost, then the client cannot perform any operations on the database. This solution clearly poses problems for nomadic users.

Scalability, availability, and performance can be improved by replicating the database across multiple physical servers that work cooperatively to provide access to the same logical database. Traditional database replication techniques rely on a constant network connection between the servers to coordinate access to the data replicas and enforce consistency. This approach has been termed *update-propagation based* or *conservative replication* [7]. Other methods involve locking a subset of the data and giving "write ownership" of it to a particular server or, in some cases, client. Any client that wishes to update that subset must funnel a request to the site currently responsible for that data.

Replication also improves availability of data. If a replica server becomes unavailable, the clients that normally access that server can be rerouted to another server that is available. However, some of the techniques used to implement replication actually reduce the availability of data, particularly for writes. If a client "replicates" data by locking it on the server, then all the other clients do not have write access to that data. Lock recovery in these types of systems is also problematic.

These traditional database replication techniques are not well suited for the challenges of the nomadic environment. They rely on a constant network connection, so they do not function well in a mobile computing environment where wireless connectivity may be intermittent or degraded due to environmental interference. In a hostile military environment, intentional jamming may also inhibit communications. In a commercial setting, the high cost of wireless or remote network access may make a constant mobile connection prohibitively expensive to maintain. In either case, current wireless technology does not offer the same amount of bandwidth as wired networks. Therefore, database techniques that assume wide bandwidth and work in a wired network environment will not work well, or at all, in a constrained bandwidth wireless environment. The common requirement driven by these issues is the ability of mobile computers to access and update database replicas while they are disconnected from the other replicas.

In addition, client/server replication does not match the communication patterns really needed by mobile users. Often a mobile user making updates to shared data needs to share those updates with other mobile users. Client/server replication requires the updates to be sent back to the server from where the second mobile user must then download them. This double communication may not be a problem in a typical office environment where connectivity is fast, cheap and (hopefully) constant. It becomes a big problem, however, when the mobile clients are both in the Sahara desert and the server is in Los Angeles. If the two mobile machines are merely inches apart, requiring them to send their data thousands of miles over difficult connections is unreasonable. Nomadic users should be able to form ad hoc workgroups of arbitrary mobile users while disconnected from central computing resources. This peer-to-peer relationship is not supported by current replication techniques.

The Bengal technology uses an approach that avoids the limitations of replication based purely on update propagation and the inflexibilities of client/server architecture. Bengal uses a peer-to-peer reconciliation-based (optimistic) database replication system that allows transparent, disconnected operation by mobile peers on a distributed database. Updates are reconciled between replicas when connectivity is available, with a mechanism to detect and resolve update conflicts. We have developed a prototype implementation of this technology called the *Bengal Database Replication System* that provides replication services between a heterogeneous mix of commercial database engines.

This paper describes our approach to replication and how it can be used in a mobile computing environment. Section 2 describes optimistic replication, the central mechanism used in Bengal. Section 3 presents the architecture and implementation of Bengal. Section 4 presents performance data for Bengal. Section 5 discusses future enhancements to Bengal. Section 6 briefly contrasts Bengal to related systems. Section 7 offers lessons learned and conclusions.

2. Optimistic Replication

One fundamental problem in replicated data systems is maintaining consistency. This problem is addressed either conservatively or optimistically.

Conservative replication (also called *update propagation-based replication*) requires consistency to be maintained whenever data records are updated. The copies of the database must remain consistent, so all the database replica servers must coordinate their reads and writes. Coordination is achieved by locking or restricting access to records so updates can be applied to all the database replicas reliably. Various methods are used to achieve this effect, including primary copies [7], voting [25] and weighted voting [6], and tokens [13]. This conservative approach preserves data consistency, but relies on constant connectivity between all the database replica servers. When the replicas are partitioned from each other, access to the data is severely limited or not available at all. In a nomadic environment, disconnected replicas would be unwriteable, even if the update is very important and would actually cause no consistency problems. Alternately, the mobile computer can hold the writeable copy, but only at the cost of making the data unwriteable by anyone else.

Optimistic replication allows replica servers to access and update data records independent from one another by delaying consistency checks [26]. Instead of preventing inconsistency from occurring, mechanisms for detecting and resolving inconsistency are used. Periodically, replicas exchange updates with one another in a process called *reconciliation*. During the reconciliation process, two replicas exchange all updates that occurred since the last reconciliation. Optimistic replication systems must detect inconsistencies during reconciliation and address them with a process called *conflict resolution* [18]. The replicas need not be connected at other times, thus allowing disconnected operation, a requirement for most distributed database applications for mobile computing.

Another form of optimistic replication allows updates to be made to disconnected replicas and, where no consistency problems exist, a designated server accepts the updates. If, however, the update conflicts with a previously accepted update, the update generated on the disconnected client is simply discarded. While this simplified approach to conflict resolution allows some form of disconnected operation, is has the obvious problem of potential loss of data. For most real-world situations, a more capable mechanism for conflict detection and resolution is needed.

Optimistic replication can use a client/server model [20] or a peer model [26]. In the client/server model, one replica of the data is designated as the special server replica. All updates created at other replicas must be registered with the server before they can be propagated further. The client/server model tremendously simplifies the design of a replicated system, at the cost of limiting its flexibility. Since all updates must go through the server, the server acts as a physical synchronization point. Conflicts can still occur, but they will always be detected only at the server, and only the server needs to handle them.

However, if the single server machine fails or is unavailable, no updates can be propagated to other replicas. Individual machines can accept their local updates, but they cannot learn of the updates applied at other machines. In a mobile environment where connectivity is limited and changing, the server may be difficult or impossible to contact, while other client replicas are simple and cheap to contact. Servers can be replicated [10], but the replicated servers must typically be in close communications at all time. If a mobile computer cannot reach one of the servers easily, almost by definition it cannot reach the others.

The peer model of optimistic replication can work better in these conditions. In the peer model, any replica can communicate its updates directly to any other replica. While the system may have a preferred topology for how updates are propagated (and that preferred topology can even include a central site that receives all updates directly), the peer model also allows opportunistic use of whatever replicas happen to be available at any particular moment. Simulation studies in the file system arena have demonstrated that the peer model increases the speed of update propagation among a set of replicas, decreasing the frequency of using an outdated version of the data [27].

As a concrete example, consider a case in which two users of a set of replicated database records are traveling together to work with a client at a remote site across the country. As they work, they each update some of the replicated records. Obviously, since they're working together, they would like to share each other's updates. In a client/server system, each user would have to propagate her updates and receive other replicas' updates by contacting the server machine on the other side of the country, perhaps requiring several long-distance telephone calls from the mobile computers to the server. Since the machines that need to share data are perhaps within inches of each other at a convenient moment, requiring this long-distance, indirect update propagation path is inefficient. If no telephone lines or network connections are available, the client/server paradigm makes transfer of updates between two collocated portable computers impossible, even though they can otherwise transfer data between the machines. In a peer system, the collocated peers could simply trade their updates directly using a local network.

One major cost of peer replication is implementation complexity. In the client/server model, the necessary record keeping that determines which version of the data is stored at each replica is simplified, since clients apply new updates directly to the server. The server always knows where the authoritative version of the data is. In the peer model, new versions of the data can be gossiped between replicas by arbitrary paths. Determining which version of the data is stored at each replica, and which versions supersede (or conflict with) which other versions requires more complex data structures and algorithms.

Fortunately, previous research on peer file system replication has defined precisely the required data structures and algorithms [7, 15]. Bengal was able to adapt these data structures and algorithms from file systems to databases with relatively little difficulty.

There are three main activities in optimistic replication systems:

1. identifying updates
2. propagating updates between replicas
3. detecting and resolving data inconsistency between updates

A data structure called a *version vector* is central to the optimistic replication implementation of Bengal [15]. The primary function of version vectors is to determine the pedigree of each record in order to detect collisions between updates made on partitioned systems. Version vectors are used in many distributed systems and have been well defined in the literature [17].

3. The Bengal Architecture

The Bengal Database Replication system allows nomadic computer users to replicate multiple copies of a database. Bengal allows each replica to be stored at a different machine. Some or all of those machines can be portable computers. No Bengal replica is any more important than any other replica, from the system's point of view. All replicas have full capabilities to perform any database operation.

Bengal is designed to operate as an add-on service to an existing database system. Bengal has been added to several popular commercial databases, including Sybase® SQL Anywhere™, Oracle®, and Microsoft® SQL Server™. The design of Bengal does not require any alterations to the source code of these systems. Instead, Bengal runs as a separate process, augmented by the use of extensibility mechanisms found in all popular commercial databases.

Bengal replicates at the record level. Each record is regarded as a separate entity for purposes of determining whether updates have been made to it. Bengal does not maintain information to enforce inter-record consistency. For example, Bengal is not aware of transactions. In normal operation, Bengal's methods of update propagation will produce the same results as if it did enforce transactional semantics, but under unusual circumstances involving normal updates occurring concurrently with reconciliation, non-transactional results can occur. These undesirable effects can be easily avoided by halting other forms of database update during reconciliation.

Figure 1 shows how Bengal enhances existing single-site databases to provide peer replication. Bengal uses existing database management system event triggers to help the Bengal engine trap updates and keep necessary records. The Bengal User Interface allows users to control when Bengal reconciles updates and various other configuration issues. The actual data is stored in a database management system, as is information Bengal requires to maintain database consistency.

Insert Figure 1 Here

Section 3.1 discusses how Bengal identifies which records have been updated. Section 3.2 discusses the mechanisms Bengal uses to propagate updates between replicas. Section 3.3 discusses how Bengal detects and handles concurrent updates.

3.1 Identifying Updates

Optimistic replication requires a reliable mechanism for detecting updates to the replicated objects to ensure that all updates are eventually propagated to all other replicas. Mechanically, updates must be identified to ensure that version vectors are updated properly. The version vectors can then be used to compare local and remote versions of the data to decide which versions are newer. Updates are typically identified in replicated systems in one of two ways: update trapping or passive scanning.

Update trapping uses system hooks to capture updates as they occur. These system hooks take on a variety of guises, but are defined loosely as code inserted into the execution stream that detects updates as they occur and takes whatever action is necessary to support replication. For example, Ficus uses file system extensions to trap all operations performed on the replicated file system, allowing an update log to be generated [7]. Trapping updates allows updated records to be identified with certainty and avoids the need to examine unchanged records. However, because trapping code is executed during user processing, care must be taken to avoid adversely affecting normal system performance.

Passive scanning identifies updated objects by comparison of the current state of replicated data and a known earlier state. For example, Rumor uses scanning, identifying updates by comparing update timestamps on replicated files [19]. All files that have a newer timestamp than when they were last examined have their version vector incremented. This approach has the advantage of little or no reduction in user performance since most processing can be postponed until just before reconciliation or during periods of inactivity. However, this approach generally requires that all replicated objects be examined to determine the few that have been changed. If there is a large number of replicated objects, or if the examination of those objects requires significant processing, then the replication process itself becomes slow. Also, depending on system time as a reliable timestamp can be problematic.

Bengal traps updates as they occur for several reasons. First, passive scanning would have required artificial changes to the database to accommodate this strategy. Every replicated table would need a timestamp field and each record's timestamp would need to be examined to identify changed records. Second, most modern databases provide triggers that offer an easy and efficient mechanism to execute code when performing an update. By creating insert, update, and delete triggers on each replicated table, changed records can be easily detected and update log entries created. Other systems have used this approach to implement distributed or replicated databases. However, care is still required to ensure that the code executed when a trigger fires is relatively efficient.

Another approach is to deduce updates from examination of the database transaction log. However, this method would require intimate knowledge of the format of the transaction log, which could vary significantly from one database vendor to another. This approach was ruled out in the initial design of Bengal for reasons of heterogeneity and speed of development. An additional obstacle is that the structure of the transaction logs of most commercial databases is considered proprietary and confidential. If this information were available, Bengal's modular design would allow a transaction log examination mechanism to be implemented later in place of the existing trigger mechanism.

When an update causes a trigger to fire, Bengal must take two actions. First, the version vector for the record must be incremented to reflect that a new version has been made. Second, the updated records must be placed at the end of the sequence of version vectors. This sequence identifies which records have been modified since the last reconciliation with various other replicas, and is used at reconciliation time to identify updated records that might be of interest to the remote replica. Instead of having the trigger perform these functions itself, an entry is made to a single *update log*. The update log contains a list of records that have been updated with an indication of the update type (insert, update, or delete). The update log is periodically flushed by the Bengal engine, which runs as a separate process from the database engine or database application. This use of the update log minimizes the performance impact of the trigger.

When the Bengal engine flushes the update log, as each update log entry is read, the version vector for its record is updated and placed at the end of the sequence. If these operations successfully complete, the update log entry is removed. If the processing of an update log entry is interrupted for some reason before it completes, that entry is processed again, incrementing the version vector more than is needed. Double incrementation of the version vector is harmless, having no impact on correctness.

3.2 Distributing Updates

Bengal distributes updates purely through reconciliation. Reconciliation is a one-way process between a single pair of replicas. One replica requests relevant updates from the other replica. Bengal supports multiple reconciliation threads, allowing simultaneous reconciliation in both directions.

When reconciliation is initiated, the replication engine on the replica receiving the request retrieves a stored version vector sequence pointer corresponding to the remote replica. If there is no stored sequence pointer, one including version vectors for all records is generated.

For each record in the version vector sequence, the version vector and ID for that record are sent to the remote replica. Before the version vector can be sent, the record must be locked and all entries in the update log for this record flushed. Locking and flushing insures that the version vector sent accurately reflects the data stored in the current version of the record. While a particular record is being processed, it cannot be externally modified. If modification were permitted during this process, race conditions could lead to failure to propagate some updates or failure to notice some concurrent updates.

The receiving replica then compares the version vector from the remote replica to its local version vector to determine if the remote version contains new updates and whether a conflict exists between the remote version and local version. The version vector contains an update count for each node. New updates are indicated when any update count in the version vector is greater than the corresponding update count in the other version vector. One version vector *dominates* a second version vector if all the update counts in the first version vector are greater than or equal to the corresponding update counts in the second version vector. A conflict exists when both version vectors indicate new updates but neither is dominant. See Section 3.3 for a complete discussion of conflicts.

If the record being processed contains new updates, a request for the record data is sent back. The sending replica transmits the record data and the receiving replica either installs the new version of the record or, if the comparison of the version vector indicated a conflict, stores it separately as an *other version* and signals a conflict. The database record is then unlocked. This process continues until all entries in the version vector sequence have been processed.

Sending the version vector first and the actual record data only when necessary conserves network traffic on a low bandwidth link. But sending the version vector and data in two steps is inefficient on high bandwidth, high latency links. For such situations, Bengal includes a slight variant of the reconcile protocol that sends the version vector and record data together in the first transmission. The receiving replica can ignore the record data if it is not needed. High latency can also be addressed by batching several version vectors and their data. The user can choose which protocol to use at reconciliation time based on what type of link is being used. For increased network utilization, Bengal packs as many records as possible into a single network packet.

Eventually all updates must be propagated to all replicas. However, this requirement does not imply that all replicas must directly reconcile with each other, as shown in Figure 2. *Gossiping* [5] allows replica A to send changes to replica B, which sends its own and A's changes to replica C. Replica C might never communicate directly with replica A, but nonetheless C receives all updates applied to A's version of the data through replica B.

Insert Figure 2a here

Insert Figure 2b here

Insert Figure 2c here

When data is replicated at more than two sites, the users should define a reconciliation topology that describes which replicas normally reconcile. This topology should form a connected graph. If the graph is not connected, sets of replicas isolated from each other will see their data diverge from the data in other such sets. Each replica should be connected to the rest of the graph in both directions, so its updates flow out and other replicas' updates flow in. Some typical replication topologies are a star (all replicas trade updates with one designated replica, Figure 3a); a ring (all replicas propagate data in a circular topology, Figure 3b); and a tree (with bi-directional links, Figure 3c). Many other topologies are possible, including meshes and completely connected.

Insert Figure 3a here

Insert Figure 3b here

Insert Figure 3c here

Several factors drive the design of reconciliation topology. If any replica has too many links, that replica may be heavily overloaded, since it will participate in many reconciliations. Also, such replicas are likely to become critical to the progress of overall update propagation, so their failure or unavailability may significantly degrade the speed of update propagation. If the shortest path between two replicas contains many links, updates will take a relatively long time to propagate between those replicas, as they will have to be gossiped through several intermediate nodes. Also, the topology should be adaptive. If a node's preferred reconciliation partner fails, the topology should automatically suggest another partner. For example, in a ring topology, if the next replica in the ring is unavailable when a node wants to reconcile, the replica that comes after the unavailable replica in the ring can be used. Simulation results on optimistic file replication topologies suggest that star topologies work well at low scale, but poorly at high scale. Ring topologies are acceptable at low scale, but delay the spread of updates at high scale. They do, however, fairly spread the load of reconciliation. Tree topologies are a good compromise, if more than a few replicas are involved [27].

Note that in a system with many mobile computers, the reconciliation topology may represent an ideal that never actually exists. At any given moment many of the links in this topology might be unusable, as one or both of the nodes they connect might be absent. Bengal users should design reconciliation topologies with this effect in mind. Topologies that rely heavily on ubiquitously present links are more likely to provide good service than topologies that rely heavily on links that are frequently unavailable.

The reconciliation topology does not constrain which replicas can reconcile with each other. At any time, any replica can trade updates with any other replica. Performing the reconciliations suggested by the topology merely guarantees that updates reach all replicas. Performing more reconciliations than the topology suggests speeds up the process of quiescence and allows the system to be opportunistic about unanticipated reconciliation possibilities.

3.3 Conflict Detection & Resolution

Conflicts occur when an update is made to a stale record: the updated replica of the record did not contain the most recent data. Conflicts can involve more than a single pair of replicas. Multiple updates at multiple replicas of the same record can lead to conflicts involving several different versions of the data. The conflicting versions may each contain important data that the other does not have, so the system cannot arbitrarily choose one version in preference to the others. In multiple replica systems that use gossiping to propagate updates, the conflict may be detected at nodes that did not initiate any of the conflicting updates.

Bengal detects conflicts by examining version vectors. A version vector is a compact encoding of the update history of a replica of the data. If two replicas of a record have completely identical version vectors, then each replica has seen all updates that the other replica has seen. No updates need to be propagated. If one replica's version vector contains one or more elements that are larger than the matching elements in the other replica's, the replica with the larger version vector elements has seen more updates, and its data should be propagated to the other replica. If, however, each version vector has at least one element larger than the matching element in the other version vector, then each replica has seen at least one update unknown to the other replica. In this case, two divergent versions of the data exist, and the conflict must be resolved. (Figure 4 shows examples of matching, dominating, and conflicting version vectors.)

Insert Figure 4a here

Insert Figure 4b here

Insert Figure 4c here

When Bengal detects a conflict with a local replica's version of a record, the record data and version vector for the conflicting version is stored as an *other version* in a separate table. The differing versions are kept until the conflict is resolved.

A conflict can be resolved by either merging the data from the two (or more) conflicting versions into a single new version or by selecting one of the versions to dominate the other. Once this choice is made, the result is a new version that incorporates all updates seen in all of the conflicting versions. Therefore, the record's version vector is set to dominate both the local version vector and any version vector stored in the *other versions* table. The entry for the conflict in the other versions table can then be removed.

Conflict resolution can be postponed. Since the other versions table is replicated, conflicts can be resolved on nodes other than the one that originally detected the conflict. Normally, conflicts should be resolved as soon as possible to reduce inconsistency and avoid divergent data versions.

When the comparison of two version vectors indicates a conflict, Bengal calls a series of configured *conflict resolvers* to attempt automatic conflict resolution. A conflict resolver is a software component that contains application-specific rules for determining which of the conflicting versions should be accepted or how a new version that merges the two should be created. Bengal maintains a set of configurable rules that are used to determine which of the available conflict resolvers should be called to handle a particular conflict. If a record matches the rule, then the specified resolver is called. If the first matching resolver does not successfully resolve the conflict, Bengal examines all remaining rules to find other resolvers that might do better. Each cascading conflict resolver is called in turn until the conflict is resolved or until all the configured resolvers have been called.

Conflict resolvers can also be run using the *conflict resolver launcher*, a program that allows the user to examine the conflicts known at his replica and apply any installed resolver on them. These manually initiated conflict resolvers can accept user input to assist in the resolution process. One such manual conflict resolver included in Bengal allows the user to examine the record data of the conflicting versions and select which version to accept.

Users can expand Bengal's conflict resolver suite. Users who understand the semantics of the records in their replicated databases are better equipped to resolve conflicts properly than system designers who are ignorant of the semantics of the data actually being stored. Furthermore, experience with resolving conflicts in the file system environment has shown that it is hard to predict all kinds of conflicts that will occur and are resolvable [18]. Users can configure Bengal's conflict interface to match their new resolvers to conflicts on particular tables or data records.

3.4 Bengal in Operation

The previous sections describe the various components of Bengal. This section describes how the components fit together and the data and control communications paths between the components. Figure 5 illustrates the relationships between the various components of a Bengal replica.

Insert Figure 5 here

Circles represent active components of the system. Cylinders represent data structures. The hashed box represents an arbitrary data transport mechanism, which is simply some method of moving data from one replica to another. Everything above this box in the diagram is part of a single Bengal replica. The dotted circles below the box are corresponding components at another Bengal replica. Solid lines show flows of data, while dotted lines indicate control flow.

At the center of the diagram, a commercial DBMS engine stores application data. Whenever an update is performed to a record of this data, built-in trigger mechanisms supplied by the DBMS notify the Record Event Trigger mechanism. This mechanism makes an entry in the local Update Log for future use. Periodically, the Update Daemon examines the Update Log for unprocessed entries. For each entry found in this log, the Update Daemon updates the version vector for the corresponding entry in the Version Vector Table.

Reconciliations are started by the Replication Manager, either automatically on some pre-defined schedule, because of some external event, or directly at the request of a user. At reconciliation time, the Distribution Daemon on the site initiating the reconciliation contacts the Reconcile Server on the remote site. The Distribution Daemon sends the list of updates to the initiating site's Reconcile Server.

The Reconcile Server consults its local version vector table to determine whether it has seen the updates listed by the remote site. For those it has not seen, the Reconcile Server responds to the remote Distribution Daemon, requesting the data. The remote Distribution Daemon consults the DBMS to extract the data and ships it, along with its current version vector. (Due to potentially concurrent operations, that version vector could have been updated since the list of updates was sent to the initiating replica's Reconcile Server, and some care is necessary to ensure that no updates are lost or conflicts missed due to such concurrency.) The Reconcile Server checks the version vector of the incoming data and, assuming all is well, installs the data in the DBMS and updates the local Version Vector table.

In some cases, conflicting updates occur. These are detected by the Reconcile Server when the remote Distribution Daemon sends its list of updated records. When the data for the conflicting record is shipped to the initiating Reconcile Server, that server invokes the Conflict Resolver Agent to fix the problem. If the Conflict Resolver Agent can resolve the conflict automatically, it does so, usually resulting in changes in the DBMS, and always resulting in changes in the Version Vector Table. If the Conflict Resolver Agent cannot automatically resolve the conflict, the remote version of the conflicting record is stored temporarily in the Other Versions data structure, and the user is notified.

3.5 Implementation Details

Bengal was designed using Microsoft's Component Object Model (COM). Seven components were initially implemented. These components were the Replication Manager, the Update Daemon, the Distribution Daemon, the Reconcile Server, version vector manipulation, database access utilities (DBAccess), and Action Log manipulation. Components are also used to implement generic transport methods and conflict resolvers.

In its first implementation, Bengal used *Action Logs*, a separate list of updates for each configured remote replica. Action Logs were updated by the Update Daemon while it processed updates. If no pre-configured Action Log existed when reconciliation was started, a temporary Action Log was created by including all replicated records in the database.

Bengal's initial implementation simply sent a single record update to the Reconcile Server at a time. While functional, this was an inefficient use of the network bandwidth. To address this problem, multiple updates were packed into a single server request. The actual number of updates sent in a single operation is determined by the size of the updates, so the network packets are always filled to capacity.

Action Logs had the advantage that the list of updates needed for a specific replica was immediately known and accessible. Action Logs eliminated the need for the reconciliation process to scan for potential updates. The disadvantage of Action Logs is that they required significant additional storage. Additionally, if an Action Log was not configured, a complete database scan was required to construct a temporary log. Unmodified records would be sent unnecessarily during the reconciliation process, resulting in a significant performance penalty when reconciling with a replica without an Action Log.

By adding a sequence number to the version vector table (with an associated index), most of the functionality of Action Logs was obtained without the overhead of administering or storing them. This architecture allows any peer to replicate with any other peer without requiring a full scan of the database to find updates the remote peer should be sent.

One disadvantage to integrated action logs is that the version vector removal process now requires a reverse flow when the version vector is finally removed. Previously, the Action Logs would contain entries for the deleted version vectors to send to remote replicas. Since the sequence is now part of the version vector table, this is no longer possible.

An additional data structure was added to the version vector table that identifies which replicas have seen an update. This allows ad hoc reconciliations to skip updates that the remote replica is known to have already seen. These "SeenBy bits," as they are called, are maintained when an update is applied. They can also be updated on the sending side as well, although this impacts performance since an additional database write must be performed.

The sequence number and SeenBy bits replaced the functionality of the Action Log component.

Code profiling and experimentation revealed that significant performance improvement could be obtained by improving the way database recordsets were handled internally by Bengal. These improvements fell into two main categories: Recordset Caching and Recordset Parameterization.

The component that is responsible for database manipulation is DBAccess. Originally, recordsets containing record data were created by DBAccess as needed and then discarded, at high cost because the recordset member columns must be dynamically determined at runtime. A recordset cache was added to save recordsets for particular tables for later use.

Having a recordset cache allows further optimizations of record retrieval from the database. Instead of performing a new database query, the recordset's filter can be parameterized so that only the query parameters need to be changed. This optimization allows the database engine to reuse the prepared statements and perform other internal optimizations.

While the recordset caching did provide a noticeable performance improvement, the recordset parameterization change had the most dramatic impact.

Performance measurements indicated that 99% of the processing time was in the DBAccess component. DBAccess was designed to handle all database operations needed by the replication engine. It performed these operations using a simplistic read record/write record methodology. Databases access table data multiple rows at a time. A complete redesign of the DBAccess interfaces was needed to more closely match the database data access model.

Microsoft offers a COM-based database access model called OLE DB. OLE DB is a set of COM interfaces whose goal is to enable uniform access to data stored in DBMS and non-DBMS information containers [12]. Since this is exactly the functionality required for DBAccess, the DBAccess component was replaced by OLE DB.

In the initial Bengal implementation, DBAccess handled all database accesses. The Update Daemon called DBAccess to read the next available Update log entry (opening a rowset, retrieving the first row, and then closing the rowset), incremented the version vector, then deleted the update log entry. This process required three rowsets to be created for each update log entry processed (two on the Update Log table, one on the Version Vector table).

With OLE DB, the Update Daemon opens a rowset on the Update Log table, accesses the next row and increments the version vectors. When the entire rowset is processed, it deletes all rows that have been processed. Each Update Log entry processed now only requires one rowset creation (Version Vector table), and one rowset of the Update Log table for the entire process. By switching to OLE DB and a multiple row approach the total number of database operations performed was reduced. It also simplified the implementation of record locking and allowed transactions to be used to group and protect updates.

4. Bengal Performance

Fully characterizing the performance of replicated data systems is difficult. Some characteristics, such as the per-replicated-item storage overhead are well-defined and easy to measure. Others, like the amount of time required to perform reconciliation, are somewhat trickier to measure in a meaningful way. Some, like the degree to which the system conceals its distributed and replicated nature, are open areas for research [11].

A further difficulty is that the performance of a replicated data system depends very highly on the workload applied and on details of its configuration, such as number of replicas, reconciliation frequency, and reconciliation topology [27]. Overheads and quality of service metrics can vary dramatically as these details change.

Another factor that greatly affects the performance of a database replication system is the underlying database engine and related drivers. The performance characteristics of commercial databases vary widely and are impacted by the processor speed and amount of memory of the machines they are run on. In addition, the performance of many databases can be greatly impacted by "tuning" and employing sophisticated hardware such as RAID disk drives. To the extent possible, all experiments reported here were conducted on similarly equipped machines and no attempt was made to "tune" the underlying databases.

In this section we characterize the performance of Bengal by presenting the following measurements:

- disk storage overheads required to replicate particular data items
- extra overhead data added to messages that move the real data between replicas
- records per second processed by major Bengal components
- reconciliation speed over LAN network and 56K-modem connection
- analysis of the network traffic generated during reconciliation
- elapsed times seen when increasing percentages of a database are updated

Except where noted, the experiments described here were run on two replicas hosted on a 233 MHz Windows® NT desktop with 128 MB RAM and a 233 MHz Windows NT laptop with 64 MB RAM. The database used was Sybase SQL Anywhere version 5.5. Except where mentioned, all experiments were conducted using Bengal's DCOM data transport over a 10 Mbit Ethernet. For each experiment, the average of at least three trials is presented.

4.1 Storage and Communication Overhead

The storage overhead can be calculated by direct examination of data stored under Bengal. For a six-replica system, the on-disk data overhead per record is 54 bytes. Part of this overhead is used to store the version vector, and the overhead will shrink or grow depending on how many replicas must be represented in the version vector. The replicated table names and the number of primary keys defined by those tables also affect this figure. This information is used to generate a key that associates each version vector with a particular data record. This overhead is deterministic, and does not vary between runs.

The communications overhead to transport a data record is also deterministic. Bengal uses DCOM as its primary data transport, which requires extra messages to set up and tear down the DCOM connection. Four packets totaling 1228 bytes (dependent on length of hostname) are used to set up the DCOM data transport connection, and four packets totaling 452 bytes are used to tear it down. This overhead is deterministic, and does not vary between runs. Keep-alive packets with an average size of 110 bytes are used periodically to maintain the connection. The sizes of other packets vary with the size of data being sent.

For each record sent over the data transport, additional data must be sent that allows the record to be properly reconciled on the receiving side. Part of this overhead is related to the encoding of the record data which, for the sample database used in these experiments, amounted to 57 bytes per record. The size of the record encoding data is highly dependent on the number of columns in the replicated tables. In addition, the corresponding version vector for each record must also be sent which, as discussed above, amounts to 54 bytes. Therefore the total overhead per record is 111 bytes. This overhead is deterministic but will vary for different replicated tables.

4.2 Measurements and Analysis

Four broad metrics characterize Bengal's performance. The first is the time taken by individual Bengal components to process a given number of record updates. The second is the time required by Bengal to reconcile a given number of updates to a remote machine over both 10 Mbit Ethernet and 56K modem connections. The third is the number of network packets generated during a reconciliation of a given number of records. The fourth metric shows how Bengal scales with larger numbers of updates.

4.2.1 Performance of Bengal Components

The operation of Bengal can be divided into three broad functional areas as described in Section 3: Record Version Maintenance, Update Distribution, and Update Reconciliation. These functional areas can be individually measured to determine where most of the time is being spent.

4.2.1.1 Update Tracking

As records are updated, their version vectors must be incremented to reflect the new update. This is implemented in Bengal with the "Update Daemon." The Update Daemon is a background task that does not directly impact application performance or replication speeds; nevertheless, its performance is important since it determines how quickly an update is "noticed" by the system, which would impact the overall responsiveness of the replication system. Bengal's current version of the Update Daemon can handle 40 updates per second, measured by making a batch update of all records in a 500 record database and measuring the time taken by the Update Daemon to update the version vector for all 500 records. Achieving this performance required making better use of rowsets.

4.2.1.2 Update Distribution

After the version vectors have been incremented, the updated records are ready for distribution to the remote replicas. During update distribution, the version vector and record data for each changed record must be read and sent through the transport to the remote replica. Transport is a synchronous operation when using DCOM. An asynchronous transport was needed to separate the transmission and reconciliation time from these measurements. This was accomplished by using the file transport that writes the updates to a file that can be read in later by the receiving replica.

After collecting the measurements for the Update Daemon as described above, 500 record updates were ready for distribution. These record updates were processed by Bengal and written to a file. By this method, Bengal was able to process 155 updates per second. Intelligent recordset caching, query parameterization, and intelligent use of rowsets proved key to achieving this performance.

4.2.1.3 Update Reconciliation

The last functional area is the reconciliation of these updates on the receiving replica. The file containing the record updates generated as described above was transferred to the receiving replica. The time required to process and apply the record updates was measured. Bengal is able to reconcile slightly less than 6 updates per second.

Reconciliation is clearly the performance bottleneck of the system, primarily because records are reconciled one at a time, each in a single recordset. The Update Daemon and the Distribution Daemon, on the other hand, can read many records into one recordset. Also, the reconciliation process involves a larger number of database writes, which are significantly slower than reads.

Theoretically, the time required to completely reconcile one replica with another would be the sum of the Replicate and Reconcile processes and the time spent transmitting the data over the transport. Experimental results described below confirm this intuition.

4.2.2 Reconciliation Speeds

The most basic time-based performance metric that describes Bengal is how long the system takes to reconcile two replicas after a given number of record updates have been applied. To measure this, 500 updates were generated on one replica and reconciled with a second replica. Reconciling 500 updates over a 10 Mbit Ethernet took 1:38 minutes. Reconciling the same number of updates over a 56 Kbit modem took slightly less than 3 minutes. While speed of the communications link clearly has an effect on reconciliation speed, a drop in bandwidth of two orders of magnitude caused reconciliation to take less than twice as long. Clearly, much of the overhead must be related to computation, rather than network delays.

4.2.3 Network Overhead

In another experiment, the number of network packets needed to reconcile 500 records over DCOM was measured by generating 500 updates on one replica and reconciling those updates with a second replica. A network analyzer was used to capture the network traffic generated during the reconciliation. 264 packets were required to perform this reconciliation. Earlier versions of Bengal had used nearly eight times as many packets for the same task. The reduction was largely due to bundling multiple record updates into a single packet, with lesser improvements from better bundling and record encoding algorithms.

4.2.4 Bengal Update Times

Figure 6 shows the time Bengal required to reconcile different numbers of updates. In this experiment, increasing numbers of updates were applied to a database containing 10,000 records and then reconciled with a remote replica over an Ethernet link. After 4000 updates, the increase in time required to reconcile more updates is slightly sublinear.

Insert Figure 6 here

5. Future Work

During the course of this research areas of further development related to optimistic, peer-based database replication were identified.

Transactional Replication – The current system replicates individual records. Maintaining transactional integrity of updates that involve multiple records over multiple tables is difficult using this method. By replicating transactions instead of records, or by replicating associated record updates in a transactional manner, the integrity of those updates could be maintained. Transactional replication could be accomplished by determining the relationship between tables by identifying foreign keys or by scouring the database transaction log and merging it with a system-wide transaction log that can be replicated to all the nodes.

Selective Database Replication – Replicated file systems, such as Rumor, implement selective replication using directory structure to define which files are replicated as well as individual file inclusion/exclusion [16]. New files can inherit the replication properties of their parent directory. Selective database replication is a much tougher problem because the criteria for determining which data should be replicated at each replica could vary widely, and could change over time. If the criteria for selecting records changes, then new records that are needed must be replicated and old records that are no longer needed must be pruned.

Further problems arise at reconciliation time. When two replicas using selective replication reconcile, the system must determine which records are involved in the reconciliation, since each partner may store different records and may change the set of records it stores at any time. The interactions between selective database replication and the action log optimizations mentioned above would further complicate the problem.

Automatic Periodic Reconciliation – To maintain consistency between the replicas and to decrease the possibility of conflicting updates, replicas must reconcile with each other on a frequent, regular basis. The reconciliation process could be automatically initiated when either a specific amount of time has passed or a high watermark of updates has occurred. The system would reconcile with an available replica (see Dynamic Choice of Reconcile Partners) or inform the user that connectivity must be established for replication. Other important events could also be used to trigger automatic reconciliation. For example, two portable computers using wireless networks could choose to reconcile when they first come within radio range of each other. Recent modeling work done at UCLA has developed methods of trading off the decreasing value of possibly stale data against the costs of using scarce bandwidth to fetch newer versions of the data. This modeling work has been used to drive file system reconciliation, and might also be usable in the database arena.

Dynamic Choice of Reconcile Partners – Current replication topologies manually select a specific replica to reconcile with, making no attempt to improve the choice with dynamic information. Instead, the system could choose reconciliation partners intelligently to optimize the spread of updates in the system or ensure that high-priority updates are sent to the most important replicas as quickly as possible. One method to achieve these goals is to advertise for prospective reconciliation partners. Replicas that match the criteria for the advertisement would respond, offering to reconcile. The advertiser could decide which ones it wants to reconcile with. Another method would be to examine the results of past reconciliations to deduce which other replicas are likely to be good partners. Dynamic Choice coupled with Automatic Periodic Reconciliation could offer a near maintenance-free replication solution for a highly dynamic, mobile replica set.

6. Related Work

Much research has been done on replicated data management. Relatively little of it relates directly to mobile computing, however, particularly replication for database systems. We will briefly survey relevant work in replication for databases and general data replication for mobile computers.

Transaction processing in distributed database systems (some of which use replication) requires consistency control. One form of consistency control for such systems is an optimistic form [2]. However, this form of optimism is different than the optimistic replication discussed here. Optimistic concurrency control is related to when the system determines that a transaction can be properly committed. Optimistic concurrency control systems permit transaction processing to proceed without obtaining locks, but check at the end of the transaction to determine if the optimism led to improper results. If so, the transaction is rolled back and re-executed. This type of optimism is insufficient for mobile environments, since it requires communications among all sites storing database components or replicas at the end of every transaction.

Much of the research on replication for database systems has concentrated on conservative solutions [7, 25, 6, 13]. These systems are largely inappropriate for general mobile computing requirements. The reference titled "Consistency in a Partitioned Network: A Survey", contains a good survey on these systems, as well as coverage of several forms of optimistic replication [3].

The growth of mobile computers has led many of the major database manufacturers to supply some form of replicated database solution that would allow some functionality to users working from portable machines. By and large, these systems support client/server optimistic replication, with varying degrees of support for handling concurrent updates and the resulting conflicts. Because many of these systems are commercial products, in many cases detailed information about their architecture and implementation are unavailable, or can only be deduced through use of the system.

Oracle® 7 and Oracle 8 contain advanced database replication capabilities. Oracle allows various styles of replication, from a master/slave situation where all updates must be applied to the master, to client/server replication where updates to clients are eventually propagated to the server, to a peer-style replication where updates performed at any replica are propagated to all other replicas. Oracle provides selective replication, a wide range of transport options (including broadcast update propagation), conflict detection and resolution, choice of either data or procedural replication (where propagation of an update causes deterministic running of a procedure, rather than transfer of data), and strong user and administration tools [4].

Oracle supports disconnected mobile computers through a client/server method. Oracle only offers a rather limited form of peer replication. Oracle's peer replication requires updates applied at any replica to be propagated to all other replicas by the originating replica. Gossiping is not permitted. This choice simplifies many problems of concurrency detection and update propagation, at the cost of slowing the progress of updates through the system. It is suitable for an environment where all replicas are well connected, but less so when some replicas must communicate indirectly with others due to constraints of mobility. Finally, Oracle's solution is specific to Oracle's own products, while Bengal can be ported easily to any SQL database [28].

Sybase's SQL Remote™ product supports optimistic client/server replication for mobile computers. It permits a mobile computer to selectively replicate a portion of a database. The mobile client maintains a log of updates, which can be shipped to the server when convenient, using email. Sybase offers other replication solutions for handling replication between different offices and multiple replication servers within an office, but these solutions are not relevant to the mobile environment [21]. SQL Anywhere™ offers limited support for mobile users, typically allowing single mobile replicas where only one replica (either the server or mobile replica) receives updates. Like Oracle, Sybase's SQL Remote and SQL Anywhere products are specific to its own database offering.

Novell® Replication Services™ permit replication among office servers [14]. Novell Replication Services provides hierarchical client/server replication, with support for selective replication. This replication service is not designed to support frequently disconnected machines, so it is not suitable for mobile computing. Like Oracle and Sybase, Novell's replication system is specific to its own product.

Lotus Notes® provides optimistic replication [23]. Like Coda, Notes provides peer replication among server machines, and client/server replication for workstation machines. Portable computers are typically configured as workstation machines, rather than servers. Thus, mobile users of Lotus Notes typically cannot directly synchronize with each other. Lotus Notes provides various data transport mechanisms, selective replication, prioritized synchronization, and many other useful replication services. Lotus Notes only replicates its own internal forms of data, though a number of products have integrated Lotus Notes replication with commercial databases, typically in a simple client/server mode.

Lotus Notes uses timestamps and a single version number to detect potential conflicts. In some complex circumstances, Lotus Notes might improperly identify non-conflicting updates as conflicting. Lotus Notes has an unusual method for dealing with conflicts. Like most other systems, Notes provides an interface for automatically resolving conflicts. However, if the conflict cannot be resolved, Notes simply treats all conflicting replicas as equally good versions of the data, saving all of them. While, in practice, many other systems save conflicting replicas until the conflict is resolved, Notes is unusual in that it does not treat those conflicting versions as being in any sense incorrect or requiring action to clean up.

CA-OpenIngres®/Replicator from Computer Associates supports various forms of optimistic replication. CA-OpenIngres/Replicator replicates transactions, building them from a journal whenever it needs to communicate them to another replica. It includes an interface for specifying conflict resolvers. CA-OpenIngres/Replicator supports peer-to-peer replication, but because a replication queue must be maintained for each replica it does not appear that the system would scale well to large numbers of peers. The documentation suggests that conflicts are detected by use of a replication key that consists of a database number, a transaction ID, and a single sequence number [29]. If so, complex patterns of updates and propagations may cause misidentification of conflicts where none exists. Similarly, the documentation states that the system is suitable for mobile use, but few details are provided. It does not appear that the system supports indirect distribution of updates

SYWARE® builds a database synchronization product called Datasync that provides replication services for any ODBC-compliant database [22]. Datasync provides client/server replication, with tools for conflict detection and resolution, support for selective replication, and the ability to use multiple data transports. Datasync transmits difference-logs of updates. The system provides an API for integration with mobile database applications. Datasync seems to detect updates by maintaining a shadow copy of the original state of the data when checked out from the server on each client. Comparing this shadow copy to either the real local copy or the current server copy allows detection of all relevant changes, at the cost of twice the data storage on the client computer. Datasync does not support peer replication. Datasync is representative of a wide class of similar products that will not be discussed here in detail. Broadly, they provide the same kinds of services as Datasync, though each product has its own strengths and weaknesses. A common theme among all of them is lack of support for peer replication.

A number of optimistically replicated file systems have been built, many specifically designed to support mobility. They include Ficus [7], Coda [10], Reconcile [9], and Rumor [8]. These systems have provided many useful lessons for the development of Bengal, including insights into the costs and benefits of different methods of detecting updates, methods of handling conflicting data, and data structures for maintaining state information about the update history of replicated data items. All are intended for use with file systems rather than databases, however, and some choices made in these systems are not suitable for the database environment.

Bayou [24] is described as a replicated storage system. It is capable of supporting both file-oriented and database-oriented operations. Bayou takes an application-aware approach to optimistic peer replication.

Applications are written to include code and information that assists the system in providing the best possible results in the face of mobile replicas communicating over links of varying connectivity. Bayou does not support commercial database systems. Bengal seeks to provide transparent replication, while Bayou requires application writers to be aware of the replication of their data.

7. Conclusions

Bengal has demonstrated that peer-oriented optimistic database replication is a practical technique for maintaining shared, updateable information at multiple mobile nodes in a mobile environment. Bengal is able to provide efficient access to the stored data and correct update propagation of that data to all other replicas, using gossiping techniques.

The client/server model and most current replication techniques depend on constant network connectivity, a requirement that is not realistic in a mobile, wireless environment. Client/server architectures enforce a rigid communication structure and prevent mobile users from directly communicating with each other. Other master-slave replication schemes are unacceptable because they enforce consistency by restricting update access.

Bengal uses a peer-to-peer, reconciliation-based replication approach that addresses the problems that mobile computing has with client/server and other forms of replication. This new replication system allows disconnected operation on a shared database with synchronization between any set of replica peers.

The Bengal Database Replication System is a prototype implementation of this scheme that can be used for further research and to study the problems mobile computing presents to distributed databases. While the Bengal system is not sufficiently developed for immediate use in critical environments, we feel Bengal offers the features and flexibility needed for mobile computing. Further development of the prototype system would improve its performance and reliability, and decrease the resources required to use Bengal. We have focused here on the problems of mobile computing, but the replication technology described may benefit other environments as well.

8. Additional Information

This research was conducted jointly by Computer Associates International, Inc., and the University of California, Los Angeles. The authors can be contacted at the following address:

Todd Ekenstam, Charles Matheny
Computer Associates International, Inc.
9800 La Cienega Blvd.
Inglewood, CA 90301
310-337-5202
800-526-9096
firstname.lastname@ca.com
(e.g. todd.ekenstam@ca.com)

Peter Reiher, Ph.D., Gerald Popek, Ph.D.
University of California, Los Angeles
Dept. of Computer Science
Los Angeles, CA 90095
310-825-8332
reiher@cs.ucla.edu
popek@cs.ucla.edu

9. Acknowledgements

The authors would like to thank Laura Smith, Joel Lilienkamp, and Steve Levy who have been supportive in their management role by providing the authors the time and resources to conduct this research. We especially thank Shirley Wei for her work on the initial implementation of Bengal. She also contributed significantly to the performance measurement and analysis section of this paper.

This research was supported by the Defense Advanced Research Projects Agency (DARPA) under contract DABT63-95-C-0063 as part of the DARPA GloMo program.

The names of actual companies and products mentioned in this paper may be the trademarks of their respective owners.

10. References

[1] P.A. Alsberg and J.D. Day, "A principle for resilient sharing of distributed resources," in IEEE 2nd International Conference on Software Engineering, San Francisco, CA, USA, 1976, pp.562-70.

[2] M.J. Carey and M. Livny, "Conflict detection tradeoffs for replicated data," ACM Transactions on Database Systems, vol.16, (no.4), p.703-46, Dec. 1991.

[3] S.B. Davidson; H. Garcia-Molina; D. Skeen, "Consistency in partitioned networks," Computing Surveys, vol.17, (no.3), p.341-70, Sept 1986.

[4] A.R. Downing. "Conflict Resolution in Symmetric Replication," *Proceedings of the European Oracle User Group Conference*, April 1995.

[5] M.J. Fischer and A. Michael. "Sacrificing Serializability To Attain High Availability of Data In an Unreliable Network," *Proceedings of the 4th Symposium on Principles of Database Systems*, 1982.

[6] D.K. Gifford, "Weighted voting for replicated data," Proceedings of the Seventh Symposium on Operating Systems Principles, Pacific Grove, CA, USA, ACM, 1979. pp.150-62.

[7] R.G. Guy; J.S. Heidemann; W. Mak; T.W.Page; G.J. Popek; D. Rothmeier, "Implementation of the Ficus replicated file system," In USENIX Conference Proceedings, pp. 63-71, June 1990.

[8] R. Guy; P. Reiher; D. Ratner; M. Gunter; W. Ma; G. Popek, "Rumor: mobile data access through optimistic peer-to-peer replication," Proceedings, Advances in Database Technologies, 1999. pp.254-65.

[9] J.H. Howard, "Using Reconciliation to Share Files Between Occasionally Connected Computers," *Proceedings of the Fourth Workshop on Workstation Operating Systems*, October 1993, pp. 56-60.

[10] J.J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda File System," ACM Transactions on Computer Systems, vol.10, (no.1), pp.3-25, Feb. 1992.

[11] G.H. Kuenning; R. Bagrodia; R.G. Guy; G.J. Popek; P. Reiher; A. Wang "Measuring the quality of service of optimistic replication," Proceedings,Object-Oriented Technology. ECOOP'98 Workshop Reader, Brussels, Belgium, 1998. pp.319-20.

[12] Microsoft, Inc. "OLE DB/ADO: Making Universal Data Access a Reality," White Paper, www.microsoft.com/data/ado/sigmod98.htm, May 1998.

[13] T. Minoura and G. Wiederhold, "Resilient extended true-copy token scheme for a distributed database system," IEEE Transactions on Software Engineering, vol.SE-8, (no.3),.pp.173-89, May 1982.

[14] Novell, Inc. "Novell Replication Services (NRS) Product White Paper," available via the Web at http://www.novell.com/whitepapers/nrs, 1997.

[15] D.S. Parker; G.J. Popek; G. Rudisin; A. Stoughton; B.J. Walker; E. Walton; J.M. Chow; D. Edwards; S. Kiser; C. Kline, "Detection of mutual inconsistency in distributed systems," IEEE Transactions on Software Engineering, vol.SE-9, (no.3), p.240-7, May 1983.

[16] D. Ratner; G. Popek; P. Reiher, "Peer Replication With Selective Control," UCLA Technical Report CSD-960031, 1996.

[17] D. Ratner; P. Reiher; G. Popek, "Dynamic Version Vector Maintenance." Computer Science Department, University of California, Los Angeles. 1997.

[18] P. Reiher; J. Heidemann; D. Ratner; G. Skinner; G. Popek, "Resolving file conflicts in the Ficus file system," Proceedings of the Summer 1994 USENIX Conference, Boston, MA, USA, pp.183-95, 1994.

[19] P. Reiher; G. Popek; M. Gunter; J. Salomone; D. Ratner. "Peer-to-Peer Reconciliation Based Replication for Mobile Computers," Proceedings of the ECOOP Workshop on Mobility and Replication, July 1996.

[20] M. Satyanarayanan; J.J. Kistler; P. Kumar; M.E. Okasaki; E.H. Siegel; D.C. Steere, "Coda: a highly available file system for a distributed workstation environment," IEEE Transactions on Computers, vol.39, (no.4),pp.447-59, April 1990.

[21] Sybase, Inc. "Sybase SQL Anywhere and Replication Server: The Enterprise Wide Replication Solution," www.sybase.com/products/system11/repsrvr.html, 1998.

[22] Syware Inc. "Datasync Architecture and Operation," www.syware.com/datasync/newdsync/ds_ao.htm, 1999.

[23] R.A. Tamura (editor). *Lotus Notes and Domino Server 4.6*, Chapter 40, “Replication and Its Administration,” SAMS Publishing, 1997.

[24] D.B. Terry; M.M. Theimer; K. Petersen; A.J. Demers; M.J. Spreitzer; C.H. Hauser, “Managing update conflicts in Bayou, a weakly connected replicated storage system,” Operating Systems Review, vol.29, (no.5),pp.172-83, Dec. 1995.

[25] R.H. Thomas, “A solution to the concurrency control problem for multiple copy data bases,” IEEE Proceedings of Spring Compcon 78, San Francisco, CA, USA, 1978, pp.56-62.

[26] B. Walker; G. Popek; R. English; C. Kline; G. Thiel, “The LOCUS distributed operating system,” Operating Systems Review, vol.17, (no.5, spec. issue.), pp.49-70, 1983.

[27] A. Wang; P.L. Reiher; R. Bagrodia, “A simulation evaluation of optimistic replicated filing in mobile environments,” Proceedings of International Performance, Computing and Communications Scottsdale, AZ, USA, 1999, pp.43-51.

[28] K. Watterson, “Database Replication Explained,” *Datamation*, pp. 62-68, 1996.

[29] Computer Associates, Inc. “CA-OpenIngres/Replicator,” Ingres-World ’96, wint.decsy.ru/du/COMPU_2H/OPENI_CY/INDEX.HTM, 1996.

FIGURES

Figure 1

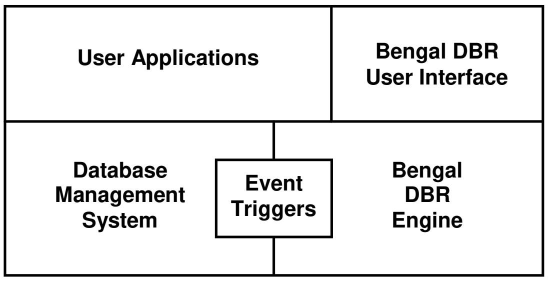


Figure 2a

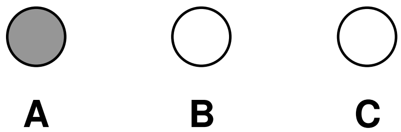


Figure 2b

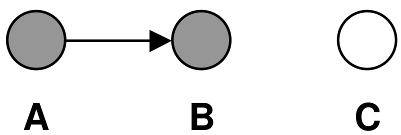


Figure 2c

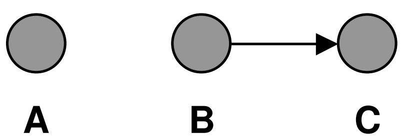


Figure 3a

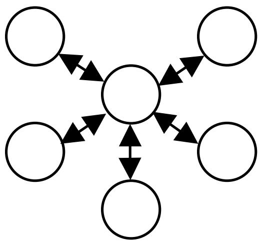


Figure 3b

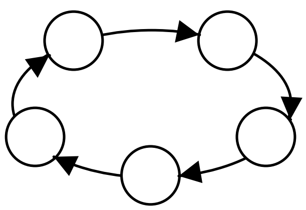


Figure 3c

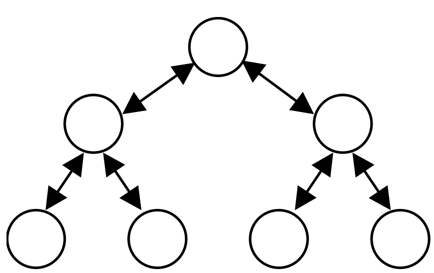


Figure 4a

8	17
1	2

Replica 1's version vector

8	17
1	2

Replica 2's version vector

Figure 4b

9	17
1	2

Replica 1's version vector

8	17
1	2

Replica 2's version vector

Figure 4c

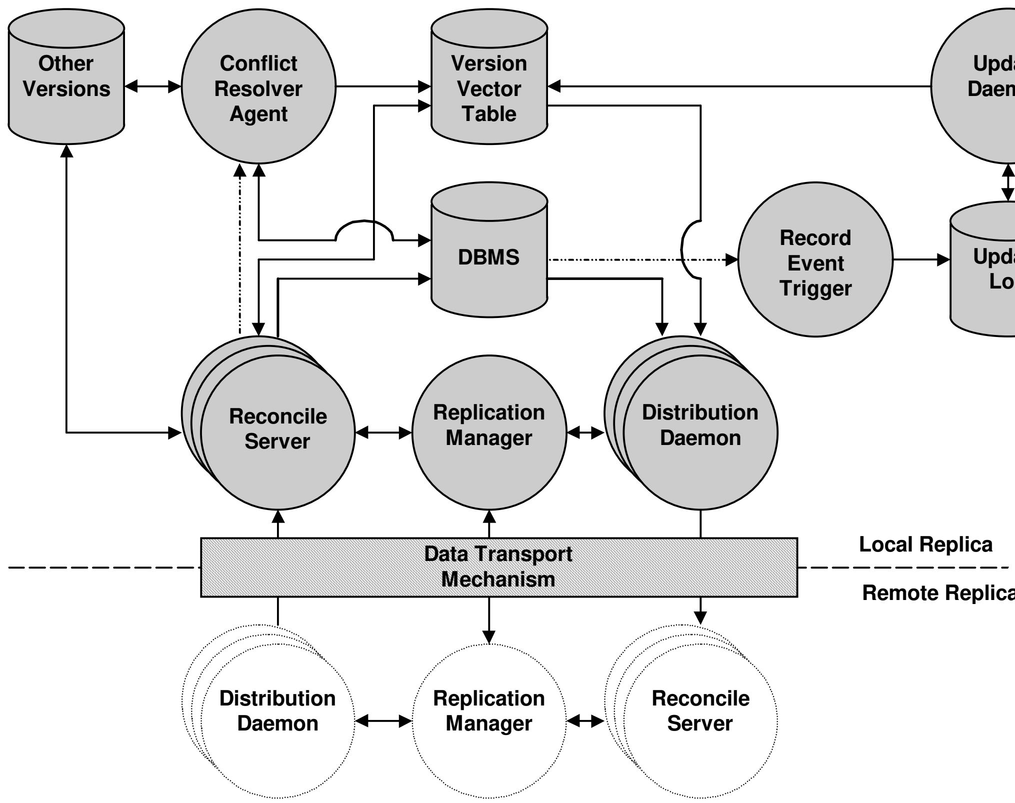
9	17
1	2

Replica 1's version vector

8	18
1	2

Replica 2's version vector

Figure 5



Other Versions
Conflict Resolver Agent
Version Vector Table
Update Daemon
DBMS
Record Event Trigger
Update Log
Reconcile Server
Replication Manager
Distribution Daemon
Data Transport Mechanism
Local Replica
Remote Replica
Distribution Daemon
Replication Manager
Reconcile Server

Figure 6

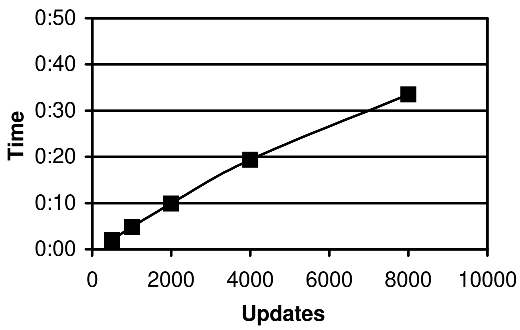

0:50
0:40
0:30
0:20
0:10
0:00
Time
0
2000
4000
6000
8000
10000
Updates

FIGURE CAPTIONS

Figure 1 - Bengal's Place in the System Architecture

Figure 2a - Updates applied at A

Figure 2b - Updates propagated from A to B

Figure 2c - Updates propagated from B to C

Figure 3a - A Star Topology

Figure 3b - A Ring Topology

Figure 3c - A Tree Topology

Figure 4a - Matching version vectors

Figure 4b - Replica 1 dominates

Figure 4c - Conflicting version vectors

Figure 5 - Relationships Among Bengal Components

Figure 6 - Update Scalability