Log-Structured Storage for Efficient Weakly-Connected Replication

Felix Hupfeld
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany,
hupfeld@zib.de, http://www.zib.de/hupfeld/

Abstract

Optimistic replication accepts changes to replicated data sets without immediate coordination, with the assumption that conflicts can later be resolved by a separate protocol. This protocol will subsequently reconcile changes between replicas, and detect and resolve any conflicts. In this paper we present a log-based reconciliation architecture that is designed to record and reconcile changes to data efficiently in terms of communication and storage overhead. Redundancy is eliminated through the use of a log-based storage mechanism. A general data model accommodates a large variety of data types. Because of its storage efficiency, the architecture is especially suited to small data such as database records.

1. Introduction

Optimistic replication is a strategy for maintaining multiple copies of data when global access protocols are too expensive or not feasible. Updates to the replicated data are accepted optimistically with the assumption that any conflicts between changes can be resolved later. These updates are disseminated by a reconciliation process to other replicas, which spreads them epidemically throughout the entire system, leading to an eventual convergence [3].

This strategy is especially useful in mobile environments, where connectivity to other replicas may be sporadic. Because a central replica might not be reachable much of the time, a decentralized approach that uses all available connections to other replicas to exchange updates is better suited to the environment.

While storage and communication resources are relatively scarce in mobile environments, today's devices are already capable of holding considerable amounts of data, making scalability become an issue. Our work concentrates on mobile applications that operate on significant amounts of small database records.

The reconciliation architecture we describe in this paper has been integrated into a personal information management system named StorageBox. A StorageBox is a distributed personal store that holds all information belonging to an individual user. Because changes are induced by actions of a single user, the change rate is relatively low and conflicts are rare.

The contribution of this paper is to show how updates to a wide range of data types can be stored and reconciled efficiently while detecting and resolving any conflicts. Using a log-based reconciliation protocol, the amount of data transferred during reconciliation grows linearly with the number of observed changes. A carefully designed log-structured store records changes to the data with an overhead that is dependent on the number and pattern of reconciliation cycles. No metadata per data item is required, so the architecture can handle small data items without penalty.

The paper is organized as follows: in Section 2 we survey research and illustrate some of the drawbacks of current approaches in this context. Section 3 relates our approach to efficiently reconciling a range of data types, using a generalized data model. Furthermore, we show how updates to the data set can be organized in a log structure, and how to use this structure to reconcile updates. Finally, we describe the processes of physically persisting and managing the log storage in Section 4.

2. Related Work

Replication systems handle data that is stored redundantly at multiple distributed *replicas*. Changes to one of the copies at one replica must be coordinated with other replicas to keep the data in a consistent state. While some system design have replicas coordinating before actually accepting changes [2], *optimistic replication* systems accept all changes and coordinate later. In these systems, the *state* of data at the replica nodes inevitably diverges until changes are *reconciled* to bring the data to a consistent state.

The connection topology between nodes is random; regardless of the order the replicas communicate, they will all eventually converge to a consistent state. Changes received from a peer are applied to the local data set, and conflicts are then detected and resolved.

Replicas use metadata to track the update history of the local data set, in order to decide which data should be reconciled, and for detecting conflicts. Two classes of reconciliation architectures can be differentiated according to the nature of this tracking data: those that keep metadata per data item and those logging changes to the data.

2.1. Architectures Using Per-Item Metadata

Architectures such as Bengal [4] or Ficus [13] use version vectors, version stamps [1], or hash histories [8] to keep the update history of their data. Separate metadata is recorded for each data item, which allows data items to be reconciled independently. This approach facilitates conflict resolution as eventual conflicts can be detected directly by comparing metadata of data items. As both conflicting versions of the data are available, a replica can decide directly how to resolve the conflict.

The disadvantage of this approach is that it adds a non-constant overhead to all data items. This overhead depends directly on the number of replicas, but is independent from the update frequency. While such overhead might not be an issue for systems in which data items are large, it can significantly enlarge the storage footprint of replicated databases. Various methods have been presented that reduce the amount of metadata in this class of systems [1, 7, 8].

To reconcile changes, the client replica requests the server's complete metadata, compares it with its own to infer data item changes, creations and deletions, and requests the needed changes from the server. This complete transfer of the metadata is inefficient when there are only a small number of changes to a large data set, or when the actual data is relatively small when compared to its accompanying metadata.

2.2. Log-based Architectures

Instead of storing the causal histories for each data item separately, the second class of architectures uses a log of changes as a common structure to represent reconciliation metadata [5, 11, 12, 16]. This log records changes and implies the history of update events in its structure. The metadata overhead of log-based architectures depends on the change rate relative to the number of data items, and on the reconciliation frequency and topology, and is independent from the size of single data items, the number of data items, and the number of replicas. Thus these architectures are particularly suited for systems that have a larger number of small data items.

For reconciliation of changes, the client replica sends a small representation of its own state to the server, which then infers the unknown parts of its log, and sends them to the client. Thus the communication demand of log-based approaches grows linearly with the number of changes to the data set.

Efficient log-based reconciliation was first described by Wuu et al. [16] where they are used to reconcile changes to a replicated dictionary.

Bayou [11] applies this technique in a database management system with a client interface that reflects that changes are only accepted provisionally. For client-controlled conflict detection and resolution, each write operation includes the expected state and a conflict resolution procedure. This allows for very flexible and content-adaptive conflict detection and resolution, but imposes a considerable communication and storage overhead.

The refdbms system [5] uses a log-based architecture for distributing a database of bibliography records. These records, however, are write-once; a mechanism ensures that record identifiers of newly create records are globally unique. Therefore, updates to the data set are never updates to data items, so conflicts cannot arise.

The more general approach of Rabinovich et al. [12] uses a log-based reconciliation protocol but keeps a log list per data item. It is functionally close to our approach but has a larger overhead.

We chose a log-based approach to benefit from its efficient reconciliation protocol and the possible savings in storage overhead. Our general data model accommodates a wide range of data types and is able to keep multiple versions of data. Thus it can provide the same conflict detection and resolution functionality that version vector and similar approaches provide.

The key to efficiency in our approach is to use a log-only storage, which previously has been applied to databases [9] and file systems where it improves write performance by taking advantage of the physical characteristics of hard disks. We exploit the log structure to embed reconciliation metadata in the data storage. This way we are able to store optimistically replicated data with a metadata overhead that is dependent on the number of reconciled but unacknowledged changes. In the optimal case, when all changes are reconciled and acknowledged, our storage architecture has virtually no metadata overhead.

3. Efficient Log-Based Reconciliation

3.1. Generalized Data Model and Operators

In our model, a *data set* has a locally known current *state*, which consists of multiple tuples of the form

(key, values).

The *key* is a unique handle to the data; one (*single-valued*) or more (*multi-valued*) distinct *values* can be associated with each key.

This generalized data model subsumes a wide range of data types (Table 1), from files to database records. It is equivalent to the model that is used in version vector based systems, except that our model is also able to keep multiple versions of a data item.

The state of data in the model can be changed by three operations:

- *add*(*key, value*), which associates a *value* with a *key* (reasonable only for multi-valued data),
- *remove*(*key, value*), which removes the given *value* from the *key*, and
- *set*(*key, value*), which removes all existing values of the given *key* and replaces them with the given *value*.

Because a distinct value can only be associated once with a certain key, these operations are idempotent: adding, setting or removing the same value more than once has no additional effect.

When mapping this data model to a replicated record database management system, for example, the database itself would be the data set, and the records would be the tuples (Figure 1). Changing a record is conceptually done with the *set* operation, which removes the old version of the record and adds the new one. *Add* and *remove* are used for creating and deleting records, respectively.

Table 1: Types of handled data

Data type	Key	Value
Files	filename	contents
Attribute-value pairs	(object-id, attribute-name)	value
Tabular data	(table-id, row-id, column-id)	value
Records	record identifier	record data
Tuples	(tuple-id, element-id)	value

3.2. Logging of updates

When operations are applied to the current data set, their effect is recorded in parallel to a *log*, which is an ordered set of *log entries* of the form

('+'/'-', key, value),

with which the system indicates that a certain *value* has been added ('+') or removed ('-') from a *key*.

These log entries act as a 'diff' to the previous state of the data set. Only those operations that have an actual effect on the current state are recorded. Because of their idempotency, operations such as an *add* with a value that is already present, or a *remove* on a non-existent value, do not change the data set and therefore do not result in a log entry. Because the log is only recording changes to the data set, *set* operations are recorded as a number of '–' (remove) log entries to remove previous values of this key followed by an '+' (add) log entry for the new value.

Log entries are organized into regions, similar to the simultaneous regions in [15]. A region contains all updates that happened at a certain replica node between two reconciliation requests. All entries of a region share a node identifier and a logical timestamp. The start of a region is marked with a *timestamp marker* in the log,

(node-name, logical time),

where *node-name* is the replica name, and *logical time* is the replica's current local logical time when the region was created.

Regions are created on a local node by appending local changes to the end of the log. To be able to name the entries of the region consistently, the current local region is closed before sending it to other replicas. Afterwards, a new region with an increased logical timestamp is started. With reconciliation, some of the regions are communicated to other replicas where they are added to the log there. Thus the log is a mixture of *locally* created and *remotely* created regions.

The complete log of a replica node is a 'diff' to the initially empty data set. In sum, it contains the data set's current state at the respective replica node. This state can be represented by a logical *state vector* that contains the newest logical timestamp for the locally known regions of each replica node (Figure 1). While this vector is cached normally, it can be rebuilt by scanning the complete log in chronological order, and noting for each found node-name the newest logical timestamp in the state vector.

Because each log entry, and thus each region, is a 'diff' to the respective previous state of the data set, it depends on its predecessors in the log. This means for reconciliation, that the regions have to be sent to other replicas completely and in log order.

This log structure does not keep metadata per data item. Combined with the log-structured storage, which we describe in Section 4, it imposes no fixed metadata overhead to stored data.

3.3. Reconciliation of Changes

When using a log to record updates, extracts of this log contain all information to bring other replicas up to date. This information is used by the reconciliation protocol, which lets an initiating client replica request changes from a responding server replica.

To be able to request a minimal amount of update information, a reconciliation client has to send the *state vector* as a description for its currently known state to the reconciliation server (Figure 1, Step 1).

Using this state vector, the reconciliation server can infer the regions that it needs to send to bring the client replica to the current state. To this end, it compares each entry of its *state vector* with the client's. Having the greater state entry for a replica node name means knowing the newer state for this replica node. If for one replica the server's entry is greater than the client's, the difference between these states defines the interval for the log entries that would bring the client up to date for this one replica (Figure 1, Step 2). The intervals of all state vector entries together define the log extract that has to be sent to the client.

The order of the sent log regions is important, because their content depends by definition on the current state of the data at the time they have been added. Thus the server has to scan the log in chronological order and send all regions that match one of the intervals to the client (Figure 1, Step 3). The client appends these region to its own log in order to be able to distribute them further to other replicas.

In this step, the log semantics apply, which means that operations are only appended to the log, when they have an actual effect on the state of the data. If this would be ignored, a '-' (remove) entry which is added twice and has no corresponding '+' (add) entry somewhere before would act as a 'pending remove'. When it gets reconciled to a replica that already has a key with the value, the value gets removed accidentally. A similar example can be constructed for '+' (add) entries that have no effect on the current state.

The log extract sent by the server may contain gaps, depending on the reconciliation topology. This occurs when some newer regions are already known by the client and thus not requested from the server, like for example, when a region stems from a third replica and has reached the client directly from that replica. This situation is illustrated in Figure 2, where both replicas received a region from a third party replica T independently. At reconciliation, this region is already contained in the client's log, and thus is excluded from the set of regions the server sends (A, B), causing a gap in the middle of the sent log extract.

This results in a difference in ordering of the regions between the replicas, which, however, does not cause inconsistencies. To support this statement, it is sufficient to look at one value of one key of the data set, because other values of the same key or of other keys do not influence the decision of whether to add or to drop a log entry.

When the third party region T is appended to the server's log, a log entry might get dropped because it is already contained in a previous region A. In Figure 2, we have marked this dropped entry with parentheses. Later, these regions are transferred to the client. As the client already knows the third party region, only regions A and B of the server will be transferred and appended to the client's log. The value that is contained in region A in the server, is already contained in region T on the client. Thus it has to be dropped from region A when it is appended. While the log ordering and region contents differ on both sides,

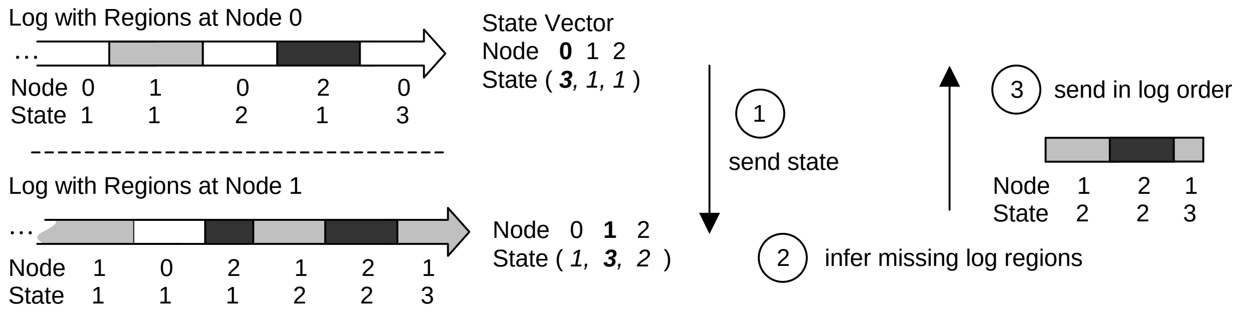

Figure 1. Log-based reconciliation protocol. The node's state vector summarizes the latest locally known state. For reconciliation, the client sends its state vector (step 1), the server infers the known but missing regions (step 2) and sends them in log order (step 3).

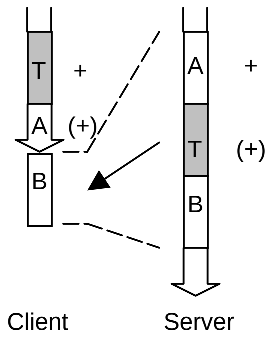

Figure 2. Different ordering does not cause inconsistencies

the data sets are consistent. Because other replicas always request complete state updates, they will also get a consistent data set whose ordering of regions differs depending on the path the regions came to them.

The amount of transferred information depends directly on the number of changes the client needs to be brought up to date. Because this is independent from the size of the data set, the reconciliation protocol can be used with large data sets.

3.4. Conflict Detection and Resolution

While the client is appending the regions to its log, it also has to apply these 'diffs' to its data set and detect any conflicts. Changes to data items are logged as deletions of their old ('-') and creation ('+') of a new state. Thus there are no conflicting updates for a value. For a key, however, there may be an update conflict, resulting in multiple values for one key.

When the system tries to apply an '+' (add) entry to a key, and finds that the key already has one or more associated values in the current state, a conflicting update might have occurred, depending on the data model. For single-valued data (such as a file system), a key that has two or more values (versions) conflicts with the model. In a multi-valued data schema, the newly-added value can be checked against existing values to detect any application-dependent conflicts. Conflicts are resolved by locally removing the conflicting values and adding the deconflicted version to the key.

4. Storage Architecture

4.1. Log-structured Storage

Conceptually, we keep the data set's current state and a log noting changes to the data set. As the complete log already contains the data set's state, a separate storage of the current state would contain redundant information. By using a log-structured storage that fuses the current state and the log, we remove this redundancy in the implementation.

This log-structured storage records the log regions in sequential order, as described in the previous section. Additionally, the store keeps a number of indices that indicate the current state of the data, so that client applications can perform queries on the data set.

These indices are also used during reconciliation to enforce the log semantics. Before an operation is appended to the log, the system checks whether the *add* or *remove* operation would actually change the state of the data set. Using an index that maps a key to its values, the current values of the changed key are retrieved and compared to the new value.

Physically, for each key there is a reference in the index to the newest log *add* entry (or entries, for a multi-valued data model). Together, the values of these referenced entries represent the current state of the values associated with the key. An index can be built by executing the operations of the log in chronological order on a normal index data structure.

4.2. Log Maintenance

In order to bring other replicas up to date, the log of a replica node accumulates all updates that it requested as a client. When the receipt of older regions is acknowledged by all replicas in the system, regions won't be requested further by other nodes, and can thus be removed from the log. The distribution state of log regions can be communicated with one of the known schemes [5, 14, 16], which can safely detect when a region is known to all replicas.

If the data set's current state is stored separately from the log, the acknowledged regions can in fact be erased from the log, because the data set does not refer to them. In a log-structured data store, however, where the log itself contains the data set, parts of regions still contribute to the local data set's current state even if they were already distributed and acknowledged, and thus these parts cannot be removed.

Those log entries that do not contribute to the data set's current state can only be erased under certain circumstances. In general, entries that do not contribute to the data set's current state are removed *add* entries, along with their corresponding *remove* entry. Whether these *add/remove* pairs can be erased depends on distribution status of their containing regions.

There are several possible distribution states an *add/remove* pair. A pair can be contained

1. in two different regions that are both not known to all replicas yet, or
2. in two different regions only one of which is known to all replicas. In both cases, the removed *add* will never contribute to the current state and can be removed from the log. The corresponding *remove*, however, has to be kept to remove the *add* at replicas to which the *add* already has been distributed separately and where it is already part of the log.
3. in different two regions that are both known to all replicas. These pairs can be removed, because they won't be requested anymore in the future, and do not contribute to the data set's current state by definition.
4. in the same region. As they do not contribute in sum to a data set's state and are never distributed independently, they can be removed at any time.

Case 1 and 2 only apply when *remove* entries are added to the local log, either in the current local region or in newly arrived log regions. When the corresponding *add* is removed from the indices during this operation, it can also be removed from the log in the same step.

As the obsolete pairs of case 3 can only be located in regions that are known to all replicas, it is sufficient to scan those regions for *remove* entries that have recently been acknowledged by all replicas.

5. Conclusions

In this paper we have presented a log-based architecture for reconciling changes to optimistically replicated data. The architecture is founded on a data model that subsumes a wide range of data types, and is scaled to significantly-large collections of small data items. We maintain a low storage footprint by keeping an efficient log format in a log-structured storage architecture, thus eliminating the redundancy of an additional store for the current state of data. To store only information that is necessary for data set's current state and for further reconciliation, we identify obsolete log entries and erase them as early as possible.

6. References

[1] P. S. Almeida. C. Baquero V. Fonte. "Version Stamps – Decentralized Version Vectors", *Proc. of the 22nd International Conference on Distributed Computing Systems*, 2002.

[2] P. A. Bernstein, V. Hadzilacos, N. Goodman. „Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987.

[3] Demers et al., "Epidemic Algorithms for Replicated Database Maintenance", *Proceedings of the Sixth Symposium on Principles of Distributed Computing*, 1987, pp. 1-12.

[4] T. Ekenstam. C. Matheny. P. Reiher. and G. Popek, "The Bengal Database Replication System", *Distributed and Parallel Databases*, vol. 9, no. 3, 2001.

[5] R. Golding, "Weak-consistency group communication and membership", *PhD thesis, University of California, Santa Cruz, 1992.*

[6] R. G. Guy. G. J. Popek. T. W. Page, Jr., "Consistency algorithms for optimistic replication", *Proc. of the 1st International Conference on Network Protocols,* 1993.

[7] Y.-W. Huang, P. S. Yu. « Lightweight Version Vectors for Pervasive Computing Devices", *Proc. of the International Workshop on Parallel Processing,* 2000.

[8] B. K. Kang. R. Wilensky. J. Kubiatowicz, "The Hash History Approach for Reconciling Mutual Inconsistency", *Proc. of the 23rd International Conference on Distributed Computing Systems*, 2003.

[9] K. Nørvåg, K. Bratbergsengen, "Log-Only Temporal Object Storage", *Proceedings of the 8th International Workshop on Database and Expert Systems Applications*, DEXA'97, Toulouse, France, pp. 728-733

[10] Parker et al., „Detection of mutual inconsistency in distributed systems", *IEEE Transactions on Software Engineering*, volume 9 (3), 1983, pp. 240-247.

[11] Petersen et al., "Flexible Update Propagation for weakly consistent replication", *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[12] M. Rabinovich. N. Gehani. A. Kononov, "Scalable Update Propagation in Epidemic Replicated Databases", *Advances in Database Technology - EDBT'96*, Lecture Notes in Computer Science Vol. 1057, Springer, pp. 207-222.

[13] Reiher et al., "Resolving File Conflicts in the Ficus File System", *USENIX Conference Proceedings*, Boston, MA, June 1994, pp. 183-195.

[14] S. K. Sarin, Nancy Lynch, "Discarding obsolete information in a replicated database system", *IEEE Transactions on Software Engineering*, SE-13(1), January 1987, pp. 39-47.

[15] M. Spezialetti, J. P. Kearns, "Simultaneous Regions: A Framework for the Consistent Monitoring of Distributed Systems", *Proc. of the 9th International Conference on Distributed Computing Systems*, 1989.

[16] G. Wuu, A Bernstein. „Efficient Solutions to the Replicated Log and Dictionary Problems", *Proceedings of the third ACM Symposium on Principles of Distributed Computing*, August 1984, pp. 233-242.