Replication by Diffusion in Large Networks*

Dahlia Malkhi† Yaron Sella‡

Abstract

We present a design for using diffusion as a vehicle for sharing data in large networks. The approach is keen to primary-backup replication in that a client informs representative replica-servers of updates, and replicas further diffuse the update among themselves. It is distinguished from primary-backup in a number of important ways. First, it tackles Byzantine failures, which are inevitable in highly decentralized systems. Second, it provides for multiple client-access points, rather than a single primary, allowing for greater flexibility, load balancing and fault tolerance. Third, it optimizes the diffusion of updates among the replicas.

1 Introduction

The rapid growth of the Internet in every domain - number of computers, available bandwidth, level of connectivity - made the concept of a global, decentralized information sharing services very realistic. Undoubtedly, such services will have to employ replication techniques in order to increase data availability and system performance. This paper outlines the challenges that arise from replication in very large networks, and presents a comprehensive design that answers these challenges.

Informally, the service is based on a client-server model, where a set of *replica servers* (in short, replicas), provide a data storage and replication service to a set of *clients*, by keeping a set of *data objects* that the clients created. The service allows clients to create new objects, read the content of objects, and write to (update) objects.

The fundamental novel principle of our approach is to provide replication based on diffusion of updates from a client's entry point to the entire system. Thus, the idea is that clients access a small group of replicas, while the replicas propagate updates between themselves. The main advantage of this design is minimizing the entry point through which the client accesses the system, and shifting the load of update diffusion onto the servers. The first result of this is a better use of communication resources, as multiple updates are batched in the communication among servers. Second, it allows localized (and hence better optimized) communication of the client with the system. Last, it shifts work load from clients to servers, which are often more capable of sustaining that load.

The scale at which we target our design mandates several important considerations:

1. Resistance to Byzantine failures - In a very large system, scenarios in which a server becomes faulty, or even maliciously faulty (Byzantine) cannot be ignored. A good design must take this scenario into consideration from its very first stages.

2. Non-blocking primitives - The common, frequent operations (read/write) must be non-blocking, otherwise the service might be locked during significant periods of time.

3. Load balancing - The scale at which we target our design implies that thousands of clients could be generating millions of updates. Unbalanced access could result in server crash. Good load balancing is therefore essential for a large system to cope with load.

*This paper appears in the European Research Seminar on Advances in Distributed Systems (Ersads 2001), Bologna, Italy, May 2001.

†School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel, dalia@cs.huji.ac.il

‡School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel, ysella@cs.huji.ac.il

4. Tunable level of service - It is unreasonable to expect that all the clients will require (and be ready to pay for) the same level of service. Even the same client may want different degrees of replication or different levels of resistance to failures for different types of data. Therefore, the infrastructure must offer flexible service levels.

The requirements listed above guided us in designing a new paradigm for a replication system, which relies on update propagation. It works as follows. Suppose for simplicity that the system emulates shared storage of a single read/write variable. We have two separate protocols, one for *read* and one for *write*. For a client to *write* a new value, it needs to successfully contact $2b+1$ servers with the written value, where b is some presumed threshold on the number of faulty servers. Servers then engage in a *diffusion* protocol, whereby the new value propagates to all the non-faulty servers, and acknowledgments are gathered and returned to the client to complete the operation. For a client to *read* the most up-to-date value stored in the system, it needs to contact $b+1$ correct servers that were written to. Left out of this description are the specification of multi-object multi-client support, of the sets of servers contacted in a read and a write, and the diffusion scheme. Determining efficient and correct mechanisms for these is the topic of the rest of this paper.

1.1 Related Work

Our work is related to the vast body of knowledge on replication and fault tolerant distributed systems. Generally, replication algorithms can be categorized by the semantical level of data emulation they achieve. The strongest guarantee in replication provides transactional semantics, whereby a group of operations (a transaction) on multiple objects, possibly nested, is done indivisibly. Transactional semantics are traditionally sought in distributed and replicated database systems. Our design differs from most of the work in this area in its attention to scale, and to arbitrary failures, whereas database replication methods traditionally focused on smaller systems with benign failures only. An example of a persistent object replication system that provide transactional guarantees in the face of Byzantine failures is Thor [LCSA99]. Due to its strong guarantee, however, the methods in Thor do not scale well.

A weaker guarantee used for replicated data is linearizability [HW90]. Linearizability is a condition that realizes atomic (indivisible) operation semantics on replicated objects as defined by Lamport [Lam86]. It is strictly weaker than atomic transactions, as it does not bind multiple or nested operations. Most prior systems implementing linearizable, persistent objects with tolerance to Byzantine failures are based on *state machine replication* [Sch90]. A recent realization of state machine replication that uses quorum systems for scalability and load balancing is provided in [CMR01]. Unfortunately, due to the need to a-priori order method invocations on different replicas in order to provide linearizable semantics, all the above methods for realizing state machine replication bear a significant cost in delay and in communication. Our techniques focus on read/write operations with weaker guarantees (safe read/write variable emulation), resulting in significantly faster protocols.

There are several recent works on large scale information sharing services providing weak emulation semantics similar to ours (e.g., for scalable data storage [CEG+99]). Our work differs from these in its use of diffusion as a fundamental vehicle for information dissemination.

Compared with all of the methods above, our approach stresses minimal client entry-point and the efficient batching of communication among servers to disperse updates. In our techniques, the load on clients scales well with the system, as does the total communication cost.

Update diffusion was proposed as a secondary support mechanism for Byzantine quorum systems in [MR98]. The performance of several diffusion algorithms was analyzed in [MMR99], and further extended in [MRRS01, MPS01]. These papers presented general strategies for achieving correct update diffusion in a Byzantine setting, and serve as the motivating technology for our scheme. However, our diffusion scheme significantly differs from these due to the need to efficiently gather acknowledgments for clients.

2 Preliminaries

Our system model is laid out in the realm of a single object replication universe. However, it should be understood that it is designed for multiple objects, with possibly distinct universes, and with varying resilience parameters. This is discussed further below, in Section 6.

The replication system of an object comprises of a universe S of n replicas, to which updates are submitted by a distinct set of clients. It is assumed that each client has a unique client ID. Up to some known threshold b of the replicas could be arbitrarily (Byzantine) *faulty*, and the rest are *correct*. Replicas can communicate via a completely connected point-to-point network. Clients can also communicate with all the replicas. The communication channels used are reliable and authenticated, i.e., A receives a message from B if and only if B actually sent it.

Clients and replicas cannot apply digital signatures to their messages, even though signatures can be quite useful in a Byzantine setting. The reason is that digital signatures impose limitations on the service, for example, preventing data post-processing by the replicas. In addition to that, digital signatures consume a lot of resources, and must be accompanied by an appropriate public-key infrastructure. Hence our model is the full Byzantine model without signatures.

Our design is concerned first and foremost with good performance, as the need to scale prohibits costly solutions. In order to reason about system performance, we will conceive of our replication schemes as operating in synchronous *rounds*. In practice, it does not matter whether these rounds are in fact synchronized. We assume that in each round, each replica may send a message to one other replica ("fan-out" is 1). A replica receives and processes all the messages sent to it in a round before the next round starts.

The performance measures that concern us are: (1) **Delay**, the number of rounds it takes for an update to reach the entire system, (2) **Communication**, measured by the total number of messages sent, and (3) **Fan-in**, the load (in terms of communication) inflicted on the servers.

3 Replication by Diffusion

The fundamental framework of our replication technique is the use of diffusion of information among replicas to disseminate client updates. At a high level, our write protocol requires a client to access a *write entry point*, such that the entry point is small, but yet it guarantees that $b + 1$ correct replicas obtain the client's update. Replicas then engage in a diffusion method whereby a *write set* obtains a copy of the update and diffuses back acknowledgments to the client. Our read protocol requires a client to access a client *read entry point* in order to obtain the variable's value from a *read set*. The requisite on the sets is that read and write sets intersect in $b + 1$ correct replicas. A timestamp on written values then assists the client in determining the correct and most recent value of the variable, simply as the highest timestamp-value returned by $b + 1$ replicas.

It is left to specify the client read and write entry sets, the read and write sets, and the diffusion method by which client write entry sets reach write sets (and by which a client's read entry set assists the client in obtaining responses from a read set). There is much room for variation in these. In fact, the use of read and write sets intersecting in $b + 1$ correct elements is, in itself, not novel: With read and write sets alone, the generic framework above fits the safe variable replication using Byzantine quorum systems of Malkhi and Reiter [MR98]. However, entry sets allow us flexibility in shifting some of the load from clients to servers, and for utilizing off-line diffusion by the servers. Additional flexibility comes from the design of the diffusion among servers. In the rest of this paper, we focus on a specific design realizing the above framework, though others are certainly viable alternatives.

Generally, the novelty in the approach above is the separation of client read and write entry points from the read and write sets, allowing clients to access minimal sets only. The advantage is in allowing localized, minimal communication load on clients, and by allowing servers to optimize the diffusion among them, e.g., by packing multiple updates in a single message. Additional novelty stems from the specific server-to-server diffusion method we propose.

4 Read and Write Protocols

This section presents specific read and write protocols, which guarantee safety and liveness properties for the data objects maintained by them. In both protocols the client addresses (directly) a small group of replicas (the entry sets). The read protocol is much cheaper than the write protocol because it does not involve propagation.

For our read/write protocols, we arrange the replicas in a tree of degree d. Each node in the tree contains $4b+1$ [1] replicas. A client's read and write entry sets consist each of a single node (any node). Efficient propagation of updates and acknowledgments is achieved along the tree, in that each node communicates only with its neighboring nodes in the tree.

Definitions and notation (for a specific object v, update u and replica r):

$Replicas(v)$ - the set of replicas on which v is replicated

$Tree(v)$ - an arrangement of $Replicas(v)$ in a (balanced) tree structure, of degree d, with $4b+1$ distinct replicas in each node

$Origin(u)$ - the node in $Tree(v)$ to which u was submitted (by a client)

$TS(u)$ - the time-stamp of an update u

$CID(u)$ - the client ID of the client who submitted u

$IN_r(v)$ ($IN_N(v)$) - The immediate neighbors of a replica r (node N) in $Tree(v)$

$TS_r(v)$ - $TS(u)$, where u is the current update stored for the object v in the replica r

$Src_r(u)$ ($Src_N(u)$) - the source node from which an update u arrives to a replica r (node N)

$u1$ *is newer than* $u2 := TS(u1) > TS(u2) \,||\, (TS(u1) = TS(u2) \,\&\&\, CID(u1) > CID(u2))$

Sending/Reading M to/from a node := Sending/Reading M to/from all replicas in that node

Read protocol, $u \leftarrow Read(v)$:

1. When a client wants to read the value of an object v, it picks any node $N \in Tree(v)$, and does the following: (a) Request every replica $r \in N$ to report its current update for v (including the time-stamp). (b) Wait until $3b+1$ responses are received.
2. All the responses whose time-stamp is lower than $b+1$ other time-stamps are discarded.
3. All the responses that were not supported by at least $b+1$ replicas are discarded.
4. There can be up to 2 distinct candidates among the $3b+1$ responses for the right value of v. If there are no candidates - an error-code indicating that there is currently no consistent value for v is returned. Otherwise, the newest of the candidates is returned.

Write protocol, $Write(v, u)$:

1. When a client wants to write an update u to an object v, it picks any node $N \in Tree(v)$, and does the following: (a) Read $TS_r(v)$ from $3b+1$ replicas N and sort in an array $TS[0 \ldots 3b]$ (minimum in $TS[0]$). (b) Send the pair $\{u, \max_{i=0 \ldots 2b}\{TS[i]\}+1\}$ to N as an update for v.
2. A replica $r \in N$ that receives an update u directly from the client, marks that $Src_r(u) = \phi$, and becomes *u-active*. A *u-active* replica r stores u as the new value of v if u is newer than v. Then the *u-active* replica r sends u further to $IN_r(v) - Src_r(u)$.
3. A replica r that receives $b+1$ copies (or more) of u from $b+1$ different replicas all belonging to the same node N, marks N as its $Src_r(u)$, and also becomes *u-active*.
4. A replica r that becomes *u-active* and cannot propagate u further (because $IN_r(v) - Src_r(u) = \phi$) becomes *u-acknowledged*. A *u-acknowledged* replica r sends an acknowledgment for u to $Src_r(u)$.

[1] Node size can be reduced from 4b+1 to 3b+1 replicas, if communication channels between correct replicas are assumed to impose bounded latency on message transmission; i.e., communication channels are assumed to be *synchronous*.

5. A *u-active* replica r that receives $3b+1$ acknowledgments for u from each node in $IN_r(v) - Src_r(u)$ also becomes *u-acknowledged*.

6. A *u-acknowledged* replica r that cannot propagate acknowledgments for u further (because $Src_r(u) = \phi$), sends an acknowledgment for u to the client.

7. Writing u finishes when the client receives $3b+1$ acknowledgments for u from $3b+1$ (or more) different replicas in $Origin(u)$.

A note on Time-stamps. We use logical time-stamps (i.e., counters) which do **not** reflect the actual date and time in which an operation occurs. The main issue regarding such time-stamps is to use enough bits so that overflow will never happen in the lifetime of the system. Since the b highest timestamp values obtained in a write operation are discarded, faulty replicas cannot purposely cause such an overflow. A reasonable choice would be to use 64 bits for the counter.
A note on old and new updates. Note that according to the read/write protocol described above, replicas store an update only if it is newer than the value currently stored, but they propagate an update (and its acknowledgments) even if it is not newer than the value currently stored.
A note on error return value from read. The likelihood of this event depends on the probabilty that a read overlaps some write operation.

Correctness proofs for the safety and liveness of our protocols are included in the full paper.

5 Performance

This section discusses the performance of our read/write protocol, which is asymmetric in the sense that read is much cheaper than write. In read, the client simply sends/receives messages to/from a single node of $4b+1$ replicas, so the delay is $O(b)$, the communication is $O(b)$, and the fan-in is 1. Note that the cost of a read operation is not dependent on n. Write involves diffusion and requires a more careful analysis. The delay of a write operation includes traversing twice the height of the tree of nodes, once for sending the update and once for collecting acknowledgments. Each node-to-node communication takes $4b+1$ rounds. Finally, at any given communication round, a node can only communicate with one of its neighbors, which adds a factor of $d+1$. The worst case bound for the delay is therefore:

$$delay = (4b+1) + 2(4b+1)(d+1)(1+2\log_d(n/(4b+1))) = O(bd\log_d(n/b))$$

The fan-in incurred by the diffusion (with a smart communication schedule) is 1, but the load inflicted on the replicas by client updates varies, of course, with the system load. The communication cost is $2(4b+1)n$, but the m-amortized communication cost (for m simultaneous client updates) is only $[(4b+1)m + 2(4b+1)n]/m$ (assuming m updates fit in a single message).

6 Implementation issues

Our infrastructure is designed to support a dynamic world of replicas maintaining multiple objects, possibly replicated on distinct universes and with varying levels of fault tolerance. This necessitates mapping objects to their universes. It also requires consideration of the possible universe overlap in the diffusion of updates among servers. Here we highlight several points in our current effort of building a prototype based on our design principles.

In our design, we give replicas logical names for easy manipulation and management. This implies that we must employ a name-translation service, e.g., the DNS. We conveniety use a sequential name space, (e.g., PSYS_001, PSYS_002). "Holes" in the name sequence can be temporarily tolerated, due to the system resistance to failures. For any given resilience level b (and tree degree d), the sequential naming induces a natural mapping of replicas to a d-ary tree with node size $4b+1$. The individual replication trees of all data objects with parameter b are therefore

chosen as subtrees of a single skeleton tree for resilience level b. This global skeleton tree enables efficient batching of multiple objects' updates, and furthermore, optimizes the communication among servers.

The mapping of objects onto their replication sub-trees is determined at object creation time. We would like to allow the client to set the replication level per object, and let the system designate the replicas realising it. To this end, a *world-view* system state is maintained as a special data object, which is made available for all clients to read. This state loosely specifies which servers participate in the service. Under sequential naming, this state can be compactly represented.

When a data object v is created, the creating client should specify parameters that determine its level of service: (1) $b(v)$ is v's level of resistance to Byzantine failures; (2) the replication-level, i.e., the desired number of replicas in $Tree(v)$; (3) an $Origin(v)$ which must be included in $Tree(v)$. The client must also specify (4) $Uname(v)$, a unique name for the data object, and (5) the client's world-view. The system determines everything else (e.g., $Tree(v)$), and returns a system name, $Sname(v)$, which contains (1)-(5), for future reference.

The rules for determining the mapping are a topic of ongoing investigation. For example, a good rule should prefer subtrees containing newer nodes over subtrees with older nodes, in order to balance the replication load between old and new replicas.

References

[CEG+99] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti and P. Yianilos. "A prototype implementation of archival intermemory". In *Proceedings of the 4th ACM Conference on Digital Libraries*, August 1999.

[CMR01] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, April 2001. To appear.

[HW90] M. P. Herlihy and J. M. Wing. "Linearizability: A correctness condition for concurrent objects". *ACM Transactions on Programming Languages and Systems* 12(3):463–492, 1990.

[Lam86] L. Lamport. "On interprocess communication (Part II: algorithms)". *Distributed Computing* 1:86-101, 1986.

[LCSA99] B. Liskov, M. Castro, L. Shrira and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, June 1999.

[MMR99] D. Malkhi, Y. Mansour and M. K. Reiter. "On diffusing updates in a Byzantine environment". In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 134-143, October 1999.

[MPS01] D. Malkhi, E. Pavlov and Y. Sella. "Optimal unconditional information diffusion". Submitted for publication.

[MR98] D. Malkhi and M. Reiter. "Byzantine quorum systems". *Distributed Computing* 11(4):203-213, 1998.

[MRRS01] D. Malkhi, M. Reiter, O. Rodeh and Y. Sella. Efficient update diffusion in Byzantine environments. Submitted for publication.

[Sch90] F. B. Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial". *ACM Computing Surveys* 22(4):299–319, December 1990.