PacificA: Replication in Log-Based Distributed Storage Systems

Wei Lin, Mao Yang
Microsoft Research Asia
{weilin, maoyang}@microsoft.com

Lintao Zhang, Lidong Zhou
Microsoft Research Silicon Valley
{lintaoz, lidongz}@microsoft.com

ABSTRACT

Large-scale distributed storage systems have gained popularity for storing and processing ever increasing amount of data. Replication mechanisms are often key to achieving high availability and high throughput in such systems. Research on fundamental problems such as consensus has laid out a solid foundation for replication protocols. Yet, both the architectural design and engineering issues of practical replication mechanisms remain an art.

This paper describes our experience in designing and implementing replication for commonly used log-based storage systems. We advocate a general replication framework that is simple, practical, and strongly consistent. We show that the framework is flexible enough to accommodate a variety of different design choices that we explore. Using a prototype system called PacificA, we implemented three different replication strategies, all using the same replication framework. The paper reports detailed performance evaluation results, especially on system behavior during failure, reconciliation, and recovery.

1. INTRODUCTION

There is a growing need for large-scale storage systems to hold an ever increasing amount of digitized information. To be cost-effective, such systems are often built with cheap commodity components. Failures therefore become a norm and fault tolerant mechanisms such as replication become the key to the reliability and availability of such systems.

Well-known and provably correct replication protocols (e.g., Paxos [18]) exist. However, building practical distributed storage systems is more than applying a known replication protocol. There is a significant gap between "theoretical" replication protocols and practical replication mechanisms. For example, a theoretical design of replication protocols is often guided with over-simplified performance metrics, such as the number of message rounds, whereas a practical replication mechanism aims to optimize end-to-end client-perceived performance metrics, such as system throughput, request processing latency, and recovery time after failure. A theoretical design focuses on a single instance of the replication protocol, whereas a practical replication mechanism must optimize overall performance of a system with multiple instances that co-exist.

In this paper, we describe our experience in designing, implementing, and evaluating the replication mechanism for large-scale log-based storage system in a local-area-network (LAN) cluster environment. Such log-based designs are commonly used in storage systems (e.g., [12, 13, 15, 24]) to improve performance by transforming random writes to sequential writes and by allowing batching, as well as to support transactional semantics. Many recently proposed storage systems (e.g., Petal [19], Frangipani [28], Boxwood [21], and Bigtable [7]) for LAN-based clusters use logs.

Log-based storage systems expose a rich structure that offers interesting design choices to explore. For example, replication can be done at different semantic levels and to different types of targets (e.g., to logical state, the log, or the on-disk physical data.)

Rather than implementing different replication mechanisms to explore the design choices, we have chosen to design and implement a simple, practical, and provably correct common replication framework. Different choices then translate into different instantiations of the same replication framework. This significantly reduces the overhead for implementing, debugging, and maintaining the system, and facilitates fair comparisons of different schemes.

The design of the replication framework reflects our strong belief that provably correct replication protocols are important foundations for practical replication mechanisms. A well-defined system model clarifies the assumptions under which the protocols work and allows us to understand and assess the risks. In contrast, ad hoc replication protocols often fail in some unknown corner cases.

Choosing the appropriate replication protocols, as well as choosing the right architectural designs, is the key to a good practical replication mechanism. We believe that simplicity and modularity are key ingredients to building practical systems. Our replication framework embraces these principles through the following features: (i) the separation of replica-group configuration management from data replication, with Paxos for managing configurations and primary/backup [2, 22] for data replication, (ii) decentralized monitoring for detecting failures and triggering reconfiguration, with monitoring traffic follows the communication patterns of data repli-

cation, and (iii) a general and abstract model that clarifies correctness and allows different practical instantiations.

In the context of log-based distributed systems, the replication framework also enables us to explore new replication schemes that differ from those in previous systems such as Boxwood and Bigtable. In particular, our schemes offer appealing advantages in terms of simplicity by not relying on a separate lock service and in terms of reduced network traffic due to better data co-location.

To evaluate the proposed replication schemes, we have implemented PacificA, a prototype of a distributed log-based system for storing structured and semi-structured web data. We have performed extensive evaluations to understand the performance and cost of various replication schemes, both for a single replication instance and for an entire system with multiple instances. System behaviors during failure and recovery are important in practice. We provide detailed analysis of system behavior in those cases.

The paper is organized as follows. Section 2 describes the replication framework we propose. Section 3 presents different approaches that the framework can be applied to a general log-based storage system. The implementation and the evaluation of the replication schemes in the context of PacificA are the focus of Section 4. Related work is surveyed in Section 5. We conclude in Section 6.

2. PACIFICA REPLICATION FRAMEWORK

In this paper, we focus our attention on a local-area network based cluster environment, where a system typically consists of a large number of *servers*. Each server has one or more CPUs, local memory, and one or more locally attached disks. All servers are connected to a high-speed local area network. We assume a single administrative domain, where servers are all trusted. No security mechanisms are considered.

Servers might fail. We assume fail-stop failures. We do not assume a bound on the message delay between servers, although they tend to be small in normal cases. Messages could be dropped or re-ordered, but not injected or modified. Network partitions could also occur. Clocks on servers are not necessarily synchronized or even loosely synchronized, but clock drifts are assumed to be bounded.

The system maintains a (large) set of data. Each piece of data is replicated on a set of servers, referred to as the *replica group*. Each server in a replica group serves as a *replica*. A server can serve as replicas in multiple groups. The data replication protocol follows the primary/backup paradigm. One replica in a replica group is designated as the *primary*, while others are called the *secondaries*. The composition of a replica group, which indicates who the primary is and who the secondaries are, is referred to as the *configuration* of the replica group. The configuration for a replica group changes due to replica failures or additions; versions are used to track such changes.

We focus on replication protocols that achieve *strong consistency*, because it represents the most natural consistency model. Informally, strong consistency ensures that the replicated system behaves the same as its non-replicated counterpart that achieves semantics such as linearizability [14]. Coping with a weaker consistency model often adds complexity to applications that are built on top of the replication layer. In Subsection 2.7, we discuss one relaxation of strong consistency and its implications.

2.1 Primary/Backup Data Replication

We adopt the primary/backup paradigm for data replication. We distinguish two types of client requests: *queries* that do not modify the data and *updates* that do. In the primary/backup paradigm, all requests are sent to the primary. The primary processes all queries locally and involves all secondaries in processing updates. The primary/backup paradigm is attractive in practice for several reasons. It is simple and resembles closely the non-replicated case with the primary as the sole access point. Queries, which tend to dominate in many workloads, are processed directly on a single server.

Strong consistency can be achieved if all servers in a replica group process the same set of requests in the same order, assuming that updates are deterministic. A primary therefore assigns continuous and monotonically increasing serial numbers to updates and instructs all secondaries to process requests continuously in this order. For clarity, we model a replica as maintaining a *prepared list* of requests and a *committed point* to the list. The list is ordered based on the serial numbers assigned to the requests and is continuous (i.e., request with serial number sn is inserted only after $sn - 1$ has been inserted.) The prefix of the prepared list up to the committed point is referred to as the *committed list*. We describe the practical implementations of the abstract model in Section 2.6.

The application state on the primary is the result of applying all requests in the committed list in the increasing order of sequence numbers on the initial state. Requests on the committed list are guaranteed not to be lost, as long as the system does not experience failures that it cannot tolerate (e.g., permanent failure of all replicas in the current configuration.) The uncommitted suffices of the prepared lists are used to ensure that requests on a committed list are preserved during reconfiguration. We use $\mathsf{committed}_p$ to denote the set of updates on server p's committed list and $\mathsf{prepared}_p$ for p's prepared list.

Normal-Case Query and Update Protocol. In the normal case (i.e., without reconfigurations), the data replication protocol is straightforward. When a primary receives a query, it processes the query against the state represented by the current committed list and returns the response immediately.

For an update, the primary p assigns the next available serial number to the request. It then sends the request with its configuration version and the serial number in a prepare

message to all replicas. Upon receiving a prepare message, a replica r inserts the request to its prepared list in the serial-number order. The request is considered *prepared* on the replica. r then sends an acknowledgment in a prepared message to the primary. The request is *committed* when the primary receives acknowledgments from all replicas. At this point, the primary moves its committed point forward to the highest point such that all requests up to this point are committed. p then sends the acknowledgment to the client indicating a successful completion. With each prepare message, the primary further piggybacks the serial number at the committed point to inform the secondaries about all requests that have been committed. A secondary can then move its committed point forward accordingly.

Because a primary adds a request into its committed list (when moving the committed point forward) only after all replicas have inserted it into their prepared list, the committed list on the primary is always a prefix of the prepared list on any replica. Also, because a secondary adds a request into its committed list only after the primary has done so, the committed list on a secondary is guaranteed to be a prefix of that on the primary. Therefore, this simple data replication protocol upholds the following Commit Invariant.

Commit Invariant: Let p be the primary and q be any replica in the current configuration, $\mathsf{committed}_q \subseteq \mathsf{committed}_p \subseteq \mathsf{prepared}_q$ holds.

The basic primary/backup protocol works only when there is no change in the configuration of the replica group. Our replication framework separates configuration management from data management. We describe configuration management and its interaction with data replication in the next subsections.

2.2 Configuration Management

In our design, a *global configuration manager* is in charge of maintaining the configurations for all replica groups in the system. For each replica group, the configuration manager maintains the current configuration and its version.

A server could initiate reconfiguration when it *suspects* (via failure detection) that a replica is faulty; the reconfiguration would remove the faulty replica from the configuration. A server could also propose to add new replicas to the configuration; for example, to restore the desirable level of redundancies. In either cases, a server sends the proposed new configuration together with its current configuration version to the configuration manager. The request will be honored if and only if the version matches that of the current configuration on the configuration manager. In this case, the proposed new configuration is installed with the next higher version. Otherwise, the request is rejected.

In the case of network partition that disconnects the primary from the secondaries, conflicting reconfiguration requests might emerge: the primary might want to remove some secondaries, whereas some secondaries try to remove the primary. Because all those requests are based on the current configuration, the first request that is accepted by the configuration manager wins. All other conflicting requests are rejected because the configuration manager has already advanced to a new configuration with a higher version.

Any failure detection mechanism that ensures the following Primary Invariant can be employed to trigger removal of a current replica. In Section 2.3, we describe in details one implementation of such a failure detection mechanism.

Primary Invariant: At any time, a server p considers itself a primary only if the configuration manager has p as the primary in the current configuration it maintains. Hence, at any time, there exists at most one server that considers itself a primary for the replica group.

2.3 Leases and Failure Detection

Even with the configuration manager maintaining the truth of the current configurations, Primary Invariant does not necessarily hold. This is because the local views of the configuration on different servers are not necessarily synchronized. In particular, we must avoid the case where an old primary and a new primary both process queries at the same time—the old primary might not be aware that a reconfiguration has created a new primary and has removed it from the configuration. Because the new primary could have processed new updates, the old primary might be processing queries on outdated states, thus violating strong consistency.

Our solution is to use *leases* [11]. The primary takes a lease from each of the secondaries by periodically sending beacons to the secondaries and waiting for acknowledgments. The primary considers its lease expired if a specified *lease period* has passed since the sending time of the last acknowledged beacon. When any lease from a secondary expires, the primary no longer considers itself a primary and stops processing queries or updates. In this case, the primary will contact the configuration manager to remove the secondary from the current configuration.

A secondary acknowledges the beacons as long as the sender remains the primary in the current configuration. If a *grace period* has passed since the last received beacon from the primary, then the secondary considers the lease to the primary expired and will contact the configuration manager to remove the current primary and become the new primary.

Assuming zero clock drift, as long as the grace period is the same as or longer than the lease period, the lease is guaranteed to expire on the primary before it does on the secondary. A secondary proposes configuration changes to try to assume the role of primary if and only if its lease to the old primary expires. Therefore, it is guaranteed that the old primary has resigned before the new primary is established; hence Primary Invariant holds.

We use the lease mechanism as the failure detection mechanism. A similar failure detection mechanism is used in other systems, such as GFS [10], Boxwood, and Bigtable/Chubby

[5]. A key difference in these systems is that leases are obtained from a centralized entity. In our case, the monitoring traffic for failure detection is always between two servers that already need to communicate with each other for data processing: the primary communicates with the secondaries when processing updates; the beacons and acknowledgments are also between the primary and the secondaries. This way, failure detection accurately captures the status of the communication channels needed for replication. Also, the data-processing messages can themselves serve as beacons and acknowledgments when the communication channel is busy. Actual beacons and acknowledgments are sent only when the communication channel is idle, thereby minimizing the overhead of failure detection. Furthermore, such a decentralized implementation eliminates loads and dependencies on a centralized entity. The load could be significant because beacons and acknowledgements are exchanged periodically between the centralized entity and every server in the system; the interval has to be reasonably small to allow fast failure detection. In the centralized solution, the unavailability of the centralized entity (e.g., due to network partition) could render the entire system unavailable because all primaries have to resign when losing leases from the central entity.

2.4 Reconfiguration, Reconciliation, and Recovery

The complexity of a replication protocol lies in how it handles reconfigurations. We divide reconfigurations into three types: removal of secondaries, removal of the primary, and addition of secondaries.

Removal of Secondaries. Removal of one or more secondaries is triggered when the primary suspects failures on a subset of the secondaries. The primary proposes a new configuration that excludes those suspected secondaries and continues without those secondaries after the configuration manager approves the proposal.

Change of Primary. Removal of the primary is triggered when a secondary suspects that the primary has failed using the lease mechanism described previously. The secondary proposes a new configuration with itself being the new primary and with the old primary excluded. When approved, the secondary p considers itself the new primary and starts to process new requests only after it completes a *reconciliation* process. During reconciliation, p sends prepare messages for uncommitted requests in its prepared list and have them committed on the new configuration. Let sn be the serial number corresponding to p's committed point after reconciliation, p then instructs all replicas to truncate their prepared lists and keep only all requests up to serial number sn. Another reconfiguration can be triggered even during reconciliation, if any replica in the new configuration fails.

Because a prepared list on a secondary always subsumes all committed requests, by having all requests in a prepared list committed, reconciliation ensures the following Reconfiguration Invariant.

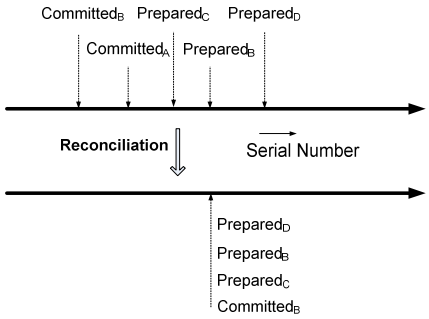

Figure 1: Reconciliation: An Example. A was the primary in the old configuration. In the new configuration, an old secondary B is promoted to be the new primary with A removed from the configuration. The first line shows the state of the prepared lists and the committed lists of the replicas based on the highest serial numbers on the lists. The second line shows the corresponding state after reconciliation.

Reconfiguration Invariant: If a new primary p completes reconciliation at time t, any committed list that is maintained by any replica in the replica group at any time before t is guaranteed to be a prefix of $\mathsf{committed}_p$ at time t. Commit Invariant holds in the new configuration.

Reconfiguration Invariant extends Commit Invariant across configurations. It also ensures that serial numbers assigned to any committed requests will be preserved and never reassigned even with primary changes. In contrast, a request in the uncommitted portion of a prepared list is not necessarily preserved. For example, when the primary assigning that serial number fails, a new primary could assign the same serial number to a different request as long as the serial number has not been assigned to a committed request. The secondaries always choose the assignment that comes from the primary with the highest configuration version.

Figure 1 illustrates how the prepared lists and the committed points change when the primary changes. Initially, A was the primary with B, C, and D as secondaries. Note that $\mathsf{committed}_B$ is a prefix of $\mathsf{committed}_A$, which is a prefix of the prepared list on any replica. Consider a reconfiguration that has B replacing the failed A as the primary. After B completes the reconciliation, the new $\mathsf{committed}_B$ is the same as the old $\mathsf{prepared}_B$, which subsumes $\mathsf{committed}_A$. The $\mathsf{prepared}_C$ is updated to preserve Commit Invariant. The extra requests on $\mathsf{prepared}_D$ are removed (or undone.)[1]

Addition of New Secondaries. New replicas can be added to a replica group to restore the level of redundancies after failures of replicas in the group. Commit Invariant must be preserved when a new server is added to the configuration.

[1]In some cases, the truncation of $\mathsf{prepared}_D$ can be done lazily when D receives from the new primary B new requests with the same serial numbers. Those requests will replace the existing ones.

This requires that the new replicas have the proper prepared lists before joining the replica group.[2] The process for a new replica to acquire the right state is referred to as *recovery*.

A simple approach is to have the primary delay the processing of any new updates until the new replicas copy the prepared list from any existing replica. This tends to be disruptive in practice. Alternatively, a new replica joins as a *candidate* secondary. The primary continues to process updates and sends prepare messages to the candidate secondaries as well. The primary terminates the candidacy of a new replica if the primary fails to get an acknowledgement from that replica.

Besides recording the new updates in the prepare messages from the primary, a candidate secondary c also fetches the portion of the prepared list it does not yet have from any existing replica. When c finally catches up, it asks the primary to add c to the configuration. As long as the candidacy of c is not yet terminated, the primary will contact the configuration manager to add c into the configuration as a secondary. Upon approval, the primary considers c as a secondary.

State transfer to a new replica during recovery could be costly. A full state transfer is unavoidable for a replica that has never been in the replica group. There are cases where a replica was removed from a replica group due to false suspicion, network partition, or other transient failures. When such an outdated replica rejoins the group, it is ideal to have the new replica perform *catch-up* recovery [26], which only fetches the missing updates that have been processed when the replica was out of the group. Due to Reconfiguration Invariant, an old replica's committed list is guaranteed to be a prefix of the current committed list and the current prepared list. However, the prepared list on the old replica is not guaranteed to be a prefix of the current prepared list if any primary change has occurred after the old replica departed the group. Therefore, an old replica can only preserve its prepared list up to the committed point in catch-up recovery and must fetch the rest.

2.5 Correctness

The protocol we have described in this section achieves strong consistency if at any time there exists at least one non-faulty replica in the current configuration defined by the configuration manager. We leave the full description of the replication protocol and a formal proof of its correctness to a separate technical report. Here we provide an informal argument of correctness for Linearizability, Durability, and Progress.

Linearizability: The system execution is equivalent to a linearized execution of all processed requests.
Durability: If a client receives an acknowledgment from the system for an update, the update has been included in the equivalent linearized execution.
Progress: Assuming that communication links between non-faulty servers are reliable, that the configuration manager eventually responds to reconfiguration requests, and that eventually there are no failures and no reconfigurations in the system, the system will return a response to each client request.

For Linearizability, observe that the committed list on the primary defines a linearized execution of all updates. Primary Invariant defines a series of primaries for a replica group, each is in action in a non-overlapping period of time. Even with primary changes, due to Reconfiguration Invariant, the committed list on any previous primary is guaranteed to be a prefix of the committed list on the new primary. To add queries to the linearized execution, observe that a primary processes each query on the state represented by a committed list; the query is naturally added after all committed updates in that committed list and after all previously completed queries. Again, because of Primary Invariant, at any time, a single primary is processing all queries. The addition of queries preserves a linearized execution. It is easy to see that the linearized execution satisfies the real-time ordering required by linearizability.

For Durability, if a client receives an acknowledgement for an update, the primary sending the acknowledgement must have inserted the request to its committed list and hence in the equivalent linearized execution.

For Progress, assuming that the configuration manager eventually returns responses for each reconfiguration request (Paxos does so under reasonable timing assumptions), the replica group eventually reconfigures to a configuration with only the non-faulty replicas. At this point, all requests will be processed and acknowledged according to the data replication protocol.

2.6 Implementations

In practice, replicas do not maintain the prepared lists and committed points as described in the abstract model. How they are best implemented depends on the semantics of the data (or the application state) being replicated. Based on the replication protocol, the implementation must support three types of actions efficiently.

- Query processing: Process queries on the state corresponding to the committed list.
- State transfer during reconciliation: Make the prepared list on the replicas identical to the new primary. Some uncommitted entries in the prepared list on a secondary might be removed as a result of the state transfer.
- State transfer during recovery: Transfer the needed portion of the prepared list from a replica to a joining server. Some uncommitted entries in the prepared list on the joining server might be removed as a result of the state transfer.

[2] In practice, the joining server copies the current committed state, as well as the part of the prepared list that has not been committed.

A natural way to allow efficient processing of queries is to apply the committed requests in the serial-number order to the application state. Prepared requests that are not yet committed might need to be maintained separately: these requests are not applied directly on the application state because they might need to be undone during state transfer: in general, undoing an operation is tricky and might require a separate undo log.

For reconciliation, a new primary simply sends all prepared and uncommitted requests; they are easy to identify because they are maintained separately from the application state. To expedite catch-up recovery, the replicas can also maintain in memory or on disk the recently committed updates. Alternatively, they can mark the segments of the state modified by those recently committed updates via dirty-region logging.

In this design, to have an update committed, a secondary has to record the update in the prepared list first and then applies the update to the application state when it learns that the request is committed. This normally involves two disk writes for each committed request.

One optimization is to batch the updates to the application state by applying them first to an in-memory image of the state and writing the new image to the disk periodically. The separate prepared list keeps not only prepared requests that are not yet committed, but also the committed requests that are not yet reflected on the on-disk application state. The latter is needed to recover the committed updates when the in-memory state is lost due to a power outage.

Design for an Append-Only Application State. In the case where the system maintains an append-only application state, where a later update appends to the old state, rather than overwriting it, the prepared list can be maintained directly on the append-only application state. This is because undoing an update on the append-only state is straightforward.

In this design, when receiving a request from the primary, a secondary applies requests directly to the application state in the serial order, effectively using the application state for maintaining its prepared list. The primary still waits for acknowledgments from all secondaries before committing the request, effectively using the application state as its committed list . A secondary maintains the committed points lazily when getting the piggybacked information on the committed point from the primary.

For reconciliation, when a new primary p starts, each secondary s ensures that the portion of the state after the committed point is the same as the portion of the state on p. For recovery, a returning replica simply copies everything starting from its committed point from the current primary.

2.7 Discussions

In this subsection, we discuss some design details of the framework and explore alternative design choices.

Availability and Performance of the configuration manager. Having the centralized configuration manager simplifies system management because it is a central point that maintains the current truth about the configuration of all replica groups in the system. More significantly, the separation also simplifies the data replication protocol and allows the tolerance of $n - 1$ failures in a replica group of size n.

For high availability, the configuration manager itself is implemented using the replicated state-machine approach with the standard Paxos protocol. The service is usually deployed on a small number (say 5 or 7) of fixed servers and can tolerate the unavailability of less than half of the servers. Unlike a centralized lock service, the unavailability of the configuration manager does not affect normal-case operations of other servers.

The configuration manager is unlikely to become a performance bottleneck. The configuration manager is not involved in normal-case request processing, but only when replica groups reconfigure. Normally, Paxos processes a command within a single round of communication, with each member server commits the changes to the local disks. Further measures can be taken to reduce the load on the configuration manager. Normally, every request to the configuration manager (even for reads) requires the execution of the consensus protocol. In our case, however, any member of the configuration manager is free to reject outdated configuration change request based on its locally stored configuration version because the up-to-date configuration has at least the same version or higher.

Durability during Transient Replica-Group Failure. In the previous sections, we show that the replication protocol is able to tolerate $n - 1$ failures in a replica group of size n. In practice, there are situations, such as during power failure, when all servers in the replica group fail. In such cases, the replica group will not be able to make progress until some of the replicas in the group come back online. If all replicas in the most recent configuration fail permanently, then data loss ensues. If at least one of the replica in the most recent configuration comes back online, our replication protocol allows the system to resume with no data loss, as long as the configuration manager also recovers from power failure. For the configuration manager implemented with the Paxos protocol, the manager can recover if a majority of the servers of the configuration manager recover and if all Paxos states are maintained on persistent storage.

To achieve durability against transient failure of an entire replica group, all replicas maintain their prepared lists and committed points on persistent storage. When a replica recovers from failure, it contacts the configuration manager to see whether it remains a replica in the current configuration. If so, it will assume its current role: if it is a primary, it will try to re-obtain leases from the secondaries; if it is a secondary, it will respond to beacons from the current primary. The server initiates reconfiguration when detecting failures.

Primary/Backup vs. Paxos. An alternative to the use of the primary/backup paradigm is to adopt the Paxos paradigm.

In Paxos, a request is committed only after it is prepared on a quorum (usually a majority) of replicas. This is true even for query requests, although the same leasing mechanism in our protocol can be used to allow query on a single replica that holds a lease from a quorum.

In the primary/backup paradigm, an update request is committed only after it is prepared on all replicas, rather than on a majority as in Paxos. The different choices of quorums in the two protocols have several implications.

First, Paxos is not sensitive to transient performance problems related to a minority of replicas: it is able to commit requests on the fastest majority; others can catch up in the background. Similarly, the failure of a minority of replicas has minimal performance impact.

In contrast, in our primary/backup protocol, the slowness of any replica causes the protocol to slow down. The failure of any replica stalls the progress until the reconfiguration removes that replica from the replica group. The timeout value for leases must be set appropriately: a high value translates into higher disruption time during replica failure, whereas a low value leads to possible false suspicion that removes a non-faulty slow replica from the replica group. False suspicion reduces the resilience of the system; our protocol facilitates fast catch-up recovery for a replica to rejoin quickly, thereby reducing the window of vulnerability.

Second, Paxos is unable to process any request when a majority of the replicas becomes unavailable. In our protocol, however, the replica group can reconfigure, with the help of the configuration manager, and continues to operate as long as at least one replica is available .

Third, reconfiguration in our protocol is simple with the help of the separate configuration manager. With Paxos, a replica group can reconfigure itself by treating a reconfiguration as a consensus decision in the current configuration. A new replica must contact a majority in the current configuration to transfer the state properly.

It is debatable which choice offers the better tradeoffs in practice. We choose the primary/backup paradigm mainly for its simplicity. Most of the experience and results we report remain valid even if the Paxos paradigm were adopted.

Weakening Strong Consistency. Strong consistency essentially imposes two requirements. First, all replicas must be executing the same set of updates in the same order. Second, a query returns the up-to-date state. There might be benefits in relaxing strong consistency. Relaxing the first requirement leads to state diversions among replicas, which often require the system to detect and recover to a consistent state and also require clients to cope with inconsistencies. Examples of such systems include Coda [16], Sprite [3], Bayou [27], CONIT [30], and PRACTI [4] and GFS.

We instead discuss the relaxation of the second; that is, we allow a query to return a somewhat outdated state. This has two interesting implications. First, the replication protocol achieves this weakened semantics without relying on the leasing mechanism: note that the Primary Invariant is the only invariant that relies on the timing assumption in the leasing mechanism. Without this invariant, an old primary could mistakenly think it is still the primary and process queries on the outdated state. However, the existence of two concurrent primaries would not lead to conflicting states on different replicas. This is because our reconfiguration protocol dictates that a new primary be a replica in the previous configuration. Given that an update is committed only after the consent from all replicas in a configuration, as long as the new primary ceases to act as a secondary in the old configuration, updates can no longer be committed after the new primary gets approved by the configuration manager. Therefore, no divergence in states can happen even during primary changes.

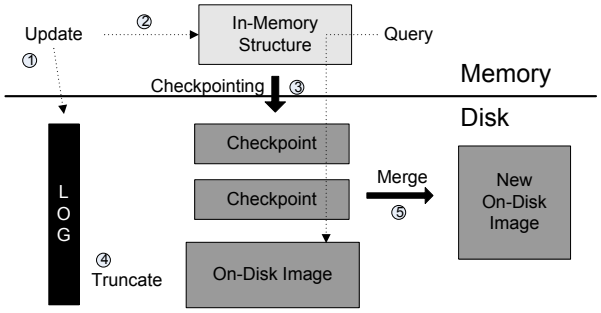

Figure 2: A Log-Based Storage System Architecture.

The second implication is that secondaries can also process queries. Due to Commit Invariant and Reconfiguration Invariant, the committed list on a secondary represents a valid, but outdated, state.

3. REPLICATION FOR DISTRIBUTED LOG-BASED STORAGE SYSTEMS

We explore the practical aspects of the proposed replication framework through log-based distributed storage systems.

Figure 2 shows a general architecture for a (non-replicated) log-based storage system. The system maintains the on-disk image for application data and also an in-memory data structure that captures the updates to the on-disk image. For persistency, the system also keeps an *application log* for the updates that are maintained only in memory.

When the system receives an update, it writes the request to the log for persistency (step 1) and applies the request to the in-memory data structure (step 2). The log is replayed to reconstruct the in-memory state when the server reboots. Periodically (before the in-memory data structure uses up the memory), a checkpoint on the disk is created (step 3) for the in-memory data structure. After the checkpoint is created, all log entries for updates that have been reflected in the on-disk checkpoint can be truncated from the log (step 4). The system can choose to store one or more checkpoints. Periodically, updates reflected in the checkpoints are merged with the on-disk image to create the new

on-disk image (step 5). A system can choose to merge without checkpointing: the updates reflected in the in-memory data structure are merged directly with the on-disk image. We choose the more-general form because it covers the special case nicely, because it accurately reflects the working of some practical real systems such as Bigtable, and because the maintenance of checkpoints can speed up reconciliation in some cases. Queries are served using the in-memory data structure, the on-disk checkpoints, and the on-disk image. It is worth pointing out that such a log-based design eliminates random disk writes and transforms them into more-efficient sequential disk writes.

3.1 Logical Replication

In the first replication strategy, replication is simply applied to the entire system state: each replica supports the same interface as the original non-replicated system and is capable of processing the same type of updates and queries. It is *logical* because the states on the replicas are logically the same. Replicas might maintain different physical states; a replica can decide unilaterally when to checkpoint and when to merge.

For logical replication, because the application state is not append-only, we choose to maintain the prepared list separately from the application state. Importantly, we can eliminate the cost of writing an update to the prepared list by maintaining a log that serves both as the application log and as the store for the prepared requests. To do so, we augment an application log entry containing the client request with three fields: the configuration version, the serial number, and the last committed serial number. A log entry subsumes any other entry that has the same serial number and a lower configuration version; this happens when a prepared request is undone during primary change. The application log therefore consists of all non-subsumed log entries with a committed serial number, while the log entries that are uncommitted belong to the prepared list.

In the first phase, when receiving a prepare message containing the request, the configuration version, the serial number, and the last committed serial number, a replica appends the message to the log. In the second phase, when a request is committed, it is applied to the in-memory data structure, but there is no need for the application to write the log any more: the request is already in the log.[3]

The application truncates its log after checkpointing. But only log entries corresponding to committed and checkpointed updates are truncated. Our replication protocol needs only the committed suffix of the prepared list. All those entries are indeed preserved during truncation.

Each checkpoint reflects updates in a range of serial numbers; the range is associated with the checkpoint. We also associate with the on-disk image the last serial number it reflects. Those associated serial numbers can help decide which piece of state is needed during catch-up recovery.

A Variation of Logical Replication. Processing updates on the in-memory data structure, generating checkpoints, and merging all consume resources, such as CPU cycles and memory, and could potentially be expensive. In logical replication, these operations are done both on the primary and redundantly on the secondaries.

It might be preferable to have only the primary perform those operations. The results of those operations can then be replicated on all replicas. We therefore introduce a variation of the logical replication scheme, referred to as *logical-V*. In logical-V, Only the primary maintains the in-memory state. A secondary, when receiving an update from the primary, simply appends the request to the log, as in logical replication, without applying the request to an in-memory state. A secondary does fetch the checkpoints from the primary as they are generated. A log entry on a secondary can be discarded only when the update corresponding to the entry is either discarded or committed and reflected in a local checkpoint.

Logical-V represents an interesting tradeoff from logical replication. In logical-V, secondaries consume much less memory (since they no longer maintain the in-memory structure) and less CPU (since they no longer generate the checkpoints, which involves compression.) However, logical-V does incur higher network load because checkpoints are now transferred among replicas. More importantly, primary failover in logical-V is more time-consuming: to become a new primary, a secondary must replay the log to reconstruct the in-memory state and to generate the needed checkpoints.

3.2 Layered Replication

The persistent state in a log-based system consists of checkpoints, on-disk images, and logs. They can all be implemented as files. A log-based storage system can then be thought as consisting of two layers, where the lower layer provides persistent file storage, and the upper layer translates the application logic into operations on files.

The view of layering leads to an alternative design that applies replication to the two layers separately. At the lower layer, a simple append-only file abstraction suffices to support operations such as appending a new log entry and creating a new checkpoint. Therefore, we can use the design for append-only state outlined in Section 2.6 for file replication.

Having a lower-layer file replication is attractive also because the abstraction is reusable and simplifies the logic at upper layers. RAID is a classic example for such a design. More recent distributed storage systems, such as Frangipani on Petal, Boxwood on RLDev, and Bigtable on GFS, all use a layered approach and have the main replication logic at the lower layer.

At the upper layer, for each piece of data, some *owner* server is responsible for accepting client requests on the data,

[3] On a primary, although a committed request is immediately applied to the in-memory data structure, the last committed serial number is updated lazily on the log. This does not affect correctness because our protocol ensures that any request in the prepared lists of all replicas will not be lost.

maintains the in-memory structure, and writes (appends) to the files at the lower layer for persistency and reliability. For example, to write a log entry, the upper layer simply request the lower layer to append the entry to a log file. The upper layer is not concerned about where the log is maintained, whether it is replicated, or how it is replicated. The lower layer does not know or care about the semantics of the files being operated on, whether the file is a log or a checkpoint.

Lower-layer replication does not completely eliminate the need for fault tolerance at the upper layer: when an owner server fails, a new owner server must take over its responsibility. In previous systems, a separate distributed lock service is used to ensure that a unique owner server is elected at the upper layer for each piece of data. An important observation we make is that, instead of inventing and deploying a new mechanism, the same replication logic can be reused at the upper layer for electing a unique owner to accept and process client requests: the problem of electing a unique owner is the same as the configuration management part of our replication framework that achieves Primary Invariant.

A natural design is to choose as the owner server the primary of the replica group that manages the data at the lower layer. There are two main advantages. First, we get a "free" fail-over mechanism at the upper layer from the replication protocol at the lower layer, without a distributed lock service and without incurring any extra overhead. When the owner server fails, a new primary will be elected at the lower layer and will be the new owner server. Second, this arrangement ensures that a copy of the data always resides on the owner server, thereby reducing network traffic.

3.3 Log Merging

The system usually maintains multiple replica groups. As advocated by GFS and Chain Replication [29], a server participates in a set of different replica groups in order to spread the recovery load when the server fails. Given that there is logically a log associated with each replica group, each server would have to maintain multiple such logs. The benefit of using logging could diminish because writing multiple logs cause the disk head to seek regularly.

One solution is to merge multiple logs to a single physical log, as done in Bigtable. The choice of how replication is done influences how logs are merged. For logical and logical-V, the log entries are explicitly sent to secondaries. Each server can then place log entries in a single local physical log. The resulting local physical log contains log entries for all replica groups that the server belongs to, whether the server is a primary or a secondary for the group. In this scheme, there is exactly one local physical log for each server in the system.

For layered replication, a server writes the log entries for all data it owns as a primary to a single replicated log at the lower layer. This effectively merges the logs for all replica groups in which the server is a primary, but not the logs for replica groups in which the server is a secondary.

With n servers in the system, each is a primary for some replica group, in the layered scheme there will be n replicated logs. If each log is replicated k times, there will be kn individual local logs in the system. This is in contrast to the n local logs in the logical and logical-V cases. In the layered scheme, there is no guarantee that the log entries needed for a new primary during fail-over are local. This is because log entries for different replica groups are merged into one log. Those different replica groups usually have their secondaries on different servers to enable parallel recovery. Therefore, the secondary copies of the log cannot be co-located with all those secondaries. On the contrary, in the logical-V case, during fail-over all log entries needed are guaranteed to be local, but each server must parse the merged log file to find the necessary log entries that pertain to the new primary replicas. For logical replication, a replica replays its local log to reconstruct the in-memory state when the replica restarts from a power loss, but a new primary does not need to replay its local log during fail-over because it maintains the in-memory state.

4. EVALUATIONS

To evaluate the replication framework and assess how well it works for a practical log-based distributed storage system, we have implemented PacificA, a prototype log-based storage system. Only features that are relevant to replication are described here. PacificA implements a distributed storage system for storing a large number of *records*. Each record has a unique key; a total order is defined for all keys. The records are split into *slices* with each slice consisting of all records in a continuous key range. PacificA provides an interface for clients to scan the records in a given key range, look up the record for a particular key, add a new record with a particular key, or modify/delete an existing record.

The PacificA design follows the architecture depicted in Figure 2 and adopts various features in Bigtable. In the on-disk image of a slice, records in the slice are stored continuously in the sorted order of their keys. An index structure and a Bloom filter are maintained for fast lookups. All modifications are logged and reflected in the in-memory data structure. The in-memory data structure is also sorted based on the keys. When the in-memory data structure grows to a certain size, the information in the data structure is compressed and written to the persistent storage in the same sorted order. The sorted order allows a simple merge sort when merging the checkpoints with the on-disk image. To reduce the amount of data written to the disks, ZLib (compression level 1) is used to compress the data in the checkpoints and the on-disk images.

We have fully implemented three different strategies for replication: logical replication, logical-V replication, and layered replication. Due to the use of the same replication framework, a significant portion of the code base is common for different strategies.

The evaluation is done on a cluster of 28 machines con-

nected through a 48-port 1GB Ethernet switch. Sixteen of those machines are used as PacificA servers; others are used as client machines. The 16 server machines each has a 2.0 GHz Intel Xeon Dual-Core CPU, 4096 MB of memory, a 1Gb Ethernet card, and a 232 GB SATA hard drive. In our experiments, we have the configuration manager run non-replicated on a separate machine. We set the lease period to be 10 seconds and the grace period to be 15 seconds.

Because queries are served by the primary only, the evaluation of the replication schemes focus mostly on performance with updates. To generate a workload with updates, a client randomly generates a 26-byte key and a record of 32 KB with a compression ratio of around 65%. The size and the compression ratio mimics a set of 100 GB crawled web pages we have. We have experimented with the the workload that inserts the real web-page data with the URLs as keys and found comparable results. The synthetic workload allows clients to generate the load efficiently (without loading the web pages from the disk and uncompressing the data first) in order to saturate the servers. For all experiments that reports update throughput, we always have enough clients with enough threads sending randomly-generated updates to saturate the servers. Group commit is implemented in all schemes and turned on by default. The default number of replicas for a replica group is 3.

4.1 Normal-Case Performance

In the first experiment, a client sends updates on a single slice stored on a replica group of three servers. No merge operations are performed in this experiment and group commit is disabled. Table 1 shows the client-observed throughput, along with the internal costs of replication on the replicas in terms of CPU utilization, disk throughput, and network utilization, for all three replication schemes. For comparison, the client throughput for the non-replicated setting is 23.7 MB/s, which is about 10% higher than that of logical-V due to lack of replication cost. We show resource consumption on both the primary and a secondary. The reported numbers are averages over a period of 10 minutes.

The resource consumptions on two secondaries actually differ. This is mainly because our implementation uses a chained structure for a primary to send requests to secondaries, as done in GFS: the primary sends the data to the first secondary, which will then pipeline the data further to the next secondary. This structure reduces the overhead on the primary for sending data and utilize the network bandwidth in a more balanced way. In such an implementation, the first secondary in the chain will have to forward the data to the next secondary, incurring more overhead than that of the next secondary. The differences are not significant. The numbers reported are for the first secondary.

For all three schemes, both disk and CPU are close to saturation. Logical-V has slightly higher throughput, which leads to higher CPU utilization, network throughput, and disk throughput. Layered seems to suffer slightly because

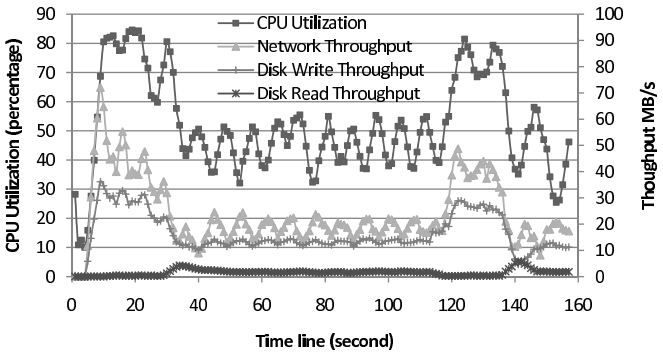

Figure 3: Impact of Checkpointing and Merging, logical replication.

it has to pass the log entries to a lower layer for transmission, disrupting the request processing pipelining. The client-observed throughput is lower than the disk throughput on the replicas because each record is not only written to the log, but also compressed and written to the checkpoint. Logical-V and layered replication have lower CPU utilizations on the secondaries because the secondaries do not maintain the in-memory structure or perform checkpointing. The network throughput is higher for these two schemes because they need to ship checkpoints to secondaries.

A Temporal View. In the second experiment, we investigate the temporal fluctuation that is inherent in a log-based system due to checkpointing and merging. Figure 3 shows the CPU utilization, network throughput, and disk read/write throughput, all on a primary, over time for logical replication.

The two elevations in disk read throughput (seconds 30 and 135) mark the start of merge operations that read checkpoints and the on-disk image. Merge also introduces competing disk writes to log writes. The result is much slower log writes and client-request processing. This translates to lower CPU utilization (due to fewer requests to process), lower network throughput, and lower disk write throughput (even with an extra source of write requests.) The first merge ends at second 115, at which point the system is able to process more client requests, reflected by a large spike in CPU utilization, network throughput, the disk write throughput, all due to client-request processing.

Checkpointing occurs regularly, starting from second 10, even during the merge operations. When performing checkpointing, CPU utilization goes up due to compression. Checkpointing also writes to the disk, competing with log writes. This slows down log writes and client-request processing. The result is lower disk throughput and network throughput. Each small fluctuation therefore corresponds to a checkpointing action.

The graphs for logical-V and layered replication are similar and thus omitted. The only noticeable difference is that the network throughput is higher in these two schemes during checkpointing because checkpoints are shipped to the

Replication Scheme	Total Throughput (MB/s)	Primary Replica			Secondary Replica		
		CPU (%)	Network (MB/s)	Disk (MB/s)	CPU (%)	Network (MB/s)	Disk (MB/s)
Logical	20.3	81.6	41.5	27.5	75.2	41.0	27.2
Logical-V	21.4	94.4	60.4	28.9	31.4	51.7	28.9
Layered	19.2	83.0	53.2	25.6	22.3	46.2	25.6

Table 1:

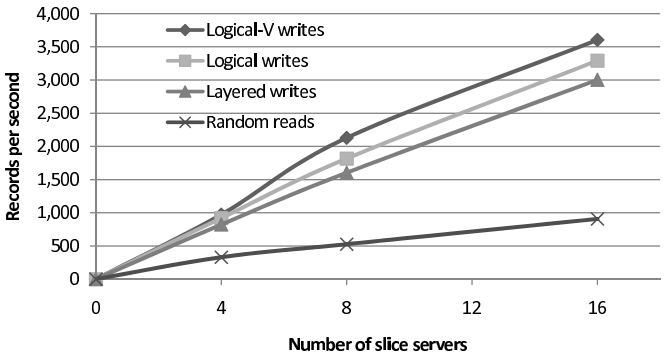

Figure 4: Scalability.

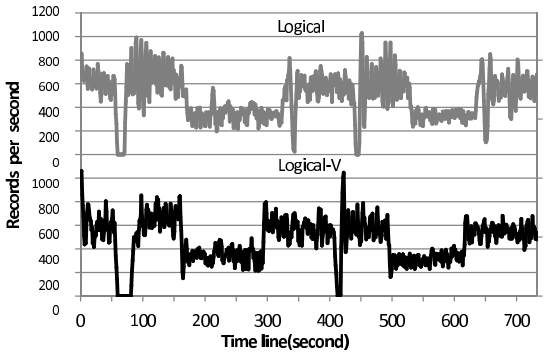

Figure 5: Performance During Failure, Reconciliation, and Recovery.

secondaries.

Scalability. In the final experiment for the normal cases, we investigate the scalability of the system. We fix the number of slices assigned to each server to 40 and use a random replica placement strategy that provides a balanced allocation of slices and primary/secondary roles to the servers. We then measure the aggregated client throughput, in terms of number of (32KB) records per second, with different numbers of servers in the system. Figure 4 shows the near-perfect scalability for all those schemes. Because logical replication utilizes extra resources on the secondaries, its overall throughput becomes lower in this end-to-end experiments with multiple replica groups. Layered replication is lower because there are three logs on each server in that scheme and because it is observed to be more sensitive to load imbalance: an overloaded server seems to impact more servers in layered replication.

We further measure the scalability of queries on random keys. Each server is a primary for one slice with about 3GB data, organized in around 56 checkpoints. We turn off our own cache, but not the file system cache or the disk cache. We report query throughput at a steady state. Because queries are processed by the primaries only, with the same data placement, all three replication schemes yield the same result. We therefore report only one series of numbers. Due to data co-location, each query request is processed locally, which leads to near-perfect scaling.

4.2 Performance during Failure, Reconciliation, and Recovery

In this subsection, we report the evaluation results during failure, reconciliation, and recovery. The first experiment uses a single replica group on three servers that stores a single slice. Figure 5 shows the changes to the client-perceived throughput (in terms of number of records per second) in face of the following series of events. At second 60, the primary is killed; the server recovers and rejoins at second 160; the same server, now a secondary, is again killed at second 410; and it recovers and rejoins again at second 490.

Figure 5 shows the results for logical replication and logical-V replication over time. Layered replication behaves almost identically with logical-V despite differences in log handling, so only logical-V curve is presented. (The next experiment distinguishes logical-V and layered replication.)

At second 60, the primary is killed. The lease expires within 15 seconds, at which time a secondary proposes a new configuration to remove the faulty primary. Reconciliation in logical replication is much faster since the secondaries already maintain the in-memory data structure. This translates into shorter downtime. For logical-V replication, the new secondary must load the log and play the log in order to reconstruct the in-memory data structure before it can accept new requests.

At second 160, the failed server recovers and rejoins. The resource consumption due to the catch-up recovery for the new replica leads to lower client-perceived throughput. When the recovery is completed at second 300, the replica group is restored to 3 servers. When the server, as a secondary, is killed at second 410, it takes 10 seconds before the primary reconfigures to remove the faulty secondary. But no reconciliation is needed. We therefore see smaller disruptions. A similar drop in the throughput is observed due to catch-up recovery when the server recovers and rejoins.

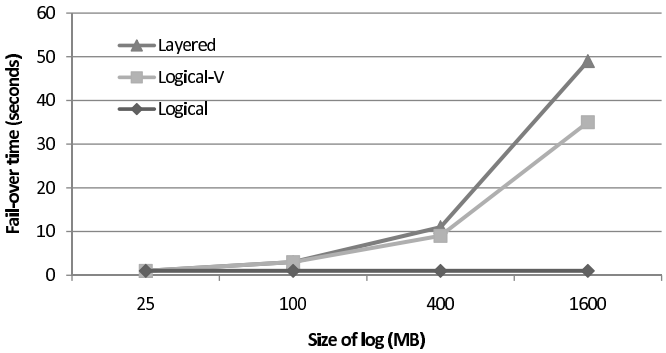

Figure 6: Fail-over Time vs. Log Size

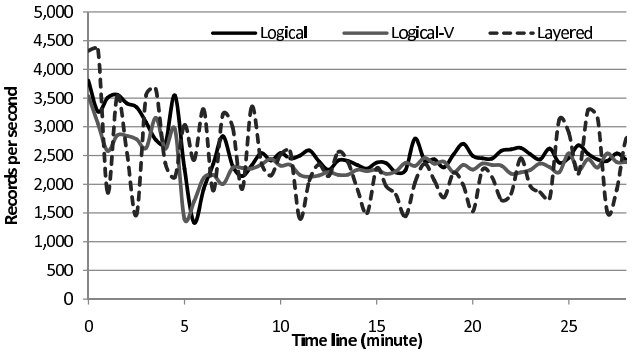

Figure 7: Global Impact of Failures and Recovery.

There are fluctuations at the points of reconfigurations (e.g., when adding a new secondary at seconds 350 and 650), especially for logical replication. This is because requests are buffered when the primary contacts the configuration manager for reconfiguration. Those buffered requests are processed in batch when reconfiguration is approved. Due to lack of precise control of the timing of the events, the two graphs are not perfectly aligned.

Fail-Over Time. In the next experiment, we look at fail-over time for different schemes. In the layered replication, fail-over happens at two layers: in the lower layer, prepared, but not yet committed, requests are re-sent. This is relatively fast because those requests are in memory. Fail-over at the upper layer involves fetching and playing the log to reconstruct the in-memory data structure. The latter is more expensive and depends on the size of the log that needs to be replayed.

In Logical-V, fail-over also involves log processing on the new primary, but the processing logic differs from that of the layered replication. The difference is more evident when there are multiple slices on each server. We quantify the fail-over time with respect to the size of the log using a three-server three-slice configuration, where each server is a primary for a slice and the other two as secondaries. We then kill one server; another server then becomes the primary for the slice originally assigned to the failed server. We measure the fail-over time with different log sizes.

Figure 6 shows the results for three schemes. Logical replication performs the best because the new primary does not have to replay the log in order to reconstruct the in-memory data structure. Logical-V replays the log locally: its local log is about three times as long as the log for the slice to be recovered due to log merging. For layered replication, a designated server loads the log entries and sends them to the new primary. This leads to remote disk reads, which are in general unavoidable: the failed server is usually a primary for multiple slices; a designated server needs to process the log and send the relevant log entries to different new primaries. But because the log contains only the log entries for the slice to be recovered, the total number of bytes read in layered replication is only 1/3 of that in logical-V. Yet, layered is still inferior in this experiment due to remote read.

The comparison between logical-V and layered replication is not as straightforward as the results indicate. For logical-V, if each server is a primary for n slices, the log on each server contains entries of $3n$ slices on average. A new primary has to read the entire log in order to fetch a small portion of useful log entries. We maintain an in-memory map to locate log entries for each slice, thereby avoiding reading the entire log. When a server fails, there could be up to n servers replaying their local log. For layered replication, only a single log is parsed and the relevant log entries sent to the appropriate new primaries. It reads less data from the disk overall, but could incur higher delay because one server has to process the entire log.

Global Impact of Failure and Recovery. To quantify the impact of failures in a real setting, we also investigate the system performance in a 16-machine cluster when two servers are killed. Figure 7 shows the changes in throughput (in terms of number of records per second) over time. In this experiment, two servers are killed at around minute five. For layered replication, the reconciliation lasts 5 minutes and ends at minute 10. We wait until all reconciliation completes before instructing replica groups that lose replicas to add random servers in order to restore the level of redundancies. This process leads to even lower throughput and finishes at minute 24. The system then returns to a normal state with throughput that is slightly lower than that before the failure, because only 14 servers are left. Logical replication finishes reconciliation almost instantly and finishes recovery at around minute 8: the recovery is faster because fewer data have been inserted into the system at this point. The curve for logical-V is similar to that of logical replication. This was somewhat surprising. The in-memory map for locating log entries for a slice on the merged log significantly reduces the amount of data the new primaries read in logical-V. Also, the global reconciliation is executed in parallel on all servers and completes promptly. This is in contrast to the case for layered replication, where two 2 GB logs need to be parsed.

Summary: Overall, we have shown that the replication

framework does yield practical replication schemes that offer good performance both during normal operations and during failure and recovery, as well as good scalability. Among the three schemes we evaluate, the logical replication is clearly superior during fail-over. This is at the expense of extra CPU and memory resources, which leads to inferior normal-case performance, especially in a workload that exercises CPU and memory fully. Layered replication is more complicated due to its special treatment of the log and the extra mechanism to process the log during fail-over. The complexity also seems to translate into slightly worse performance. Logical-V is simpler and offers reasonable overall performance.

5. RELATED WORK

Replication protocols have been extensively studied in the form of consensus protocols and the replicated state-machine approach. Representative protocols include Paxos [18] and view-stamped replication [22].

There is also a long history of distributed systems that implement replication for high reliability and availability. It is hard to enumerate them all. We instead focus on a small subset of the solutions that are closest to ours.

HARP [20] was among the first to build a highly-available file system on commodity hardware. It adopts primary/backup for replication with a write-ahead log. HARP's log is maintained entirely in-memory and is replicated on all secondaries before the requests are committed. HARP does not need to maintain an on-disk log because it assumes that each server is equipped with UPS. HARP uses the view-stamped replication protocol [22]. Each replica group consists of $2n + 1$ servers with $n + 1$ servers storing the data. Other n servers are witnesses and participate during reconfigurations (i.e., view changes) only.

Like HARP, Echo [26] also uses primary/backup with the help of witnesses. Echo looks at the different sematic levels for replication, although all the choices are at a level of file system and lower. A log is used for journaling and the solution is similar to logical-V replication in PacificA, although Echo does not consider log merging. Echo's design takes into account reconciliation and recovery. Echo also uses the lease mechanism both for clients and for replicas.

Petal [19] implements distributed virtual disks that are highly available. Petal was the first to use Paxos for implementing a global state manager. Data replication employs mirroring. The configuration of each replica group is maintained in the global state manager. For reconciliation and recovery, the scheme uses busy bits to track in-flight operations and uses stale bits to track data regions that have been updated by only one replica. Boxwood [21] uses a similar technique for implementing a low-level replicated logical device. Unlike Petal, which implements the global state manager on all servers, Boxwood uses a small set of servers. Both systems support mirroring only.

Frangipani [28] is a distributed file system layered on top of Petal. Frangipani uses metadata logging for clean failure recovery and for better performance. The logs are maintained in Petal, as in our layered replication, although no log merging is considered. A distributed lock service is used to coordinate accesses to the Petal storage. The distributed lock service itself uses the global state manager implementing Paxos. A similar lock-server based design is used in Boxwood to layer a high-level B-Link tree abstraction on top of lower-layer reliable storage. In both cases, different layers are not coordinated for co-location. The workload is expected to exhibit locality with very little conflicting requests to make caching effective.

Google File System (GFS) [10] is a distributed file system designed specifically for the workload at Google, where files are large and mainly append-only. GFS uses a log-based approach for maintaining meta data on a master; the master is replicated in a variation of primary/backup scheme. For data replication (at the chunkservers), GFS adopts a relaxed consistency model, where replicas might have inconsistent states when failures happen. The relaxed consistency model allows the replication protocol not to deal with reconfiguration and reconciliations explicitly. This choice simplifies the replication protocol, but at the cost of increased complexity for clients to deal with inconsistencies.

Bigtable [7] is a distributed structured store layered on top of GFS. PacificA design is significantly influenced by Bigtable, but with significant differences as well. The persistent state of Bigtable is stored in GFS; this includes both the logs and the SSTables (which correspond to the checkpoints and the on-disk image in Figure 2). At the upper layer, Bigtable uses the Chubby lock service [5] to keep track of the managers (or tablet servers), where a manager must obtain an exclusive lock in order to operate on the relevant state on GFS. Co-location between Bigtable's tablet masters and their tablet data can be done only with best effort. Yet the layered design was a sensible engineering choice given that GFS was already available when Bigtable was built.

Chain replication [29] organizes the replicas in a replica group into a chain. Updates are sent to the head of the chain and processed along the chain until they are committed at the tail. The tail responds to both queries and updates. Chain replication also uses Paxos to implement a configuration manager. Chain replication can be regarded as a variation of our replication framework. Consider the tail as the primary in our framework, the chain replication presents another way of achieving Reconfiguration Invariant and Commit Invariant. Chain replication assumes no network partition, but can use the same lease mechanism to achieve Primary Invariant even during network partition. Each replica can simply maintain a lease from its predecessor on the chain.

At the cost of increased latencies, chain replication benefits by splitting the query and update load between the head and the tail. However, the benefit diminishes when servers participate in multiple replica groups with different roles. For layered replication, because the owner server does the

extra work of maintaining the in-memory data structure and performing checkpointing/merging, using chain replication at a lower layer does not help offload the overhead from the owner server. Although a direct and fair comparison between our experimental results and Chain Replication's simulation results is hard, based on our experience, we believe that chain replication offers comparable throughput to our primary/backup counterpart, if not worse.

Unlike previous proposals that use primary/backup, Federated Array of Bricks (FAB) [9] exemplifies an alternative quorum-based approach. FAB implements a reliable and scalable distributed disk array using new majority voting based schemes that allow reconfiguration. The reconfiguration is done through a dynamic voting scheme that resembles that of Paxos and requires state synchronization.

Many peer-to-peer storage systems, such as CFS [8] and PAST [25], also employ replication for reliability. Most of such systems store immutable data, which greatly simplifies the replication logic. Systems such as Oceanstore [17] and FARSITE [1] do allow mutations to data. Due to possible malicious attacks in an untrusted P2P environment, both systems use Byzantine fault tolerance protocols (e.g., [6].)

6. CONCLUDING REMARKS

Replication schemes are an important building block for distributed systems and play a significant role in the performance and reliability of those systems. This paper offers a simple and clean way of thinking about replication and its correctness in the context of a general replication framework. At the same time, we show how to cross the bridge from provably correct schemes to practical replication schemes. Our experiences bring new insights on replication for log-based storage systems and on how the way replication handles failures influences system behavior.

7. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI 2002*, pages 1–14, December 2002.

[2] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proc. of the 2nd International Conference on Software Engineering*, pages 627–644, October 1976.

[3] M. Baker and J. K. Ousterhout. Availability in the Sprite distributed file system. *Operating System Review*, 25(2):95–98, April 1991.

[4] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of 3rd NSDI*, May 2006.

[5] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proc. of the Seventh Symposium on Operating System Design and Implementation (OSDI 2006)*, November 2006.

[6] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. of the 3rd OSDI*, February 1999.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of the 7th OSDI*, November 2006.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th SOSP*, pages 202–215, 2001.

[9] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Building reliable enterprise storage systems on the cheap. In *Proc. 11th ASPLOS*, October 2004.

[10] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proc. of the 19th SOSP*, October, 2003.

[11] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th Symp. Operating Systems Principles (SOSP)*, pages 202–210, December 1989.

[12] J. Gray. Notes on data base operating systems. *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, 60:393–481, 1978.

[13] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. of 11th SOSP*, pages 155–162, 1987.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[15] M. Kazar, B. Leverett, O. Anderson, V. Apostolides, B. Buttos, S. Chutani, C. Everhart, W. Mason, S. Tu, and E. Zayas. Decorum file system architectural overview. In *Proc. of the USENIX Summer'90 Conference*, pages 151–163, 1990.

[16] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proc. of the 13th SOSP*, pages 213–225, 1991.

[17] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. of 9th ASPLOS*, pages 190–201, November 2000.

[18] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.

[19] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th ASPLOS*, pages 84–92, October 1996.

[20] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. of 13th SOSP*, pages 226–238, 1991.

[21] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of the 6th OSDI*, pages 105–120, December 2004.

[22] B. M. Oki and B. H. Liskov. Viewstamped replication: a new primary copy method to support highly-available distributed systems. In *Proc. of 7th PODC*, pages 8–17, 1988.

[23] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[24] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. on Computer Systems*, 10(1):26–52, 1992.

[25] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the 18th SOSP*, pages 188–201, 2001.

[26] G. Swart, A. Birrell, A. Hisgen, and T. Mann. Availability in the Echo file system. Research Report 112, System Research Center, Digital Equipment Corporation, September 1993.

[27] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the 15th SOSP*, pages 172–183, December 1995.

[28] C. A. Thekkath, T. P. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th Symp. Operating Systems Principles (SOSP)*, pages 224–237, October 1997.

[29] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of the 6th OSDI*, pages 91–104, 2004.

[30] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–182, 2002.