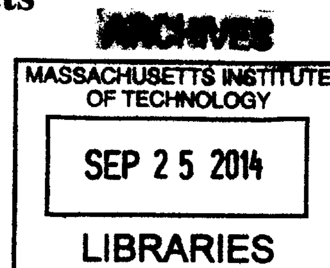
Fast In-Memory Storage Systems : Two Aspects


ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
SEP 25 2014
LIBRARIES

by

Yandong Mao

B.E., Software Engineering, Fudan University (2006)
M.S., Computer Science, Fudan University (2009)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
August 29, 2014

Signature redacted

Certified by ...
Robert T. Morris
Professor
Thesis Supervisor

Signature redacted

Accepted by.......
Leslie A. Kolodziejski
Professor
Chairman, Department Committee on Graduate Students

Fast In-Memory Storage Systems : Two Aspects
by
Yandong Mao

Submitted to the Department of Electrical Engineering and Computer Science
on Aug 29, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This dissertation addresses two challenges relating to in-memory storage systems.

The first challenge is storing and retrieving data at a rate close to the capabilities of the underlying memory system, particularly in the face of parallel accesses from multiple cores. We present Masstree, a high performance in-memory key-value store that runs on a single multi-core server. Masstree is derived from a concurrent B+tree. It provides lock-free reads for good multi-core performance, which requires special care to avoid writes interfering with concurrent reads. To reduce time spent waiting for memory for workloads with long common key prefixes, Masstree arranges a set of B+trees into a Trie. Masstree uses software prefetch to further hide DRAM latency. Several optimizations improve concurrency. Masstree achieves millions of queries per second on a 16-core server, which is more than 30× as fast as MongoDB [6] or VoltDB [17].

The second challenge is replicating storage for fault-tolerance without being limited by slow writes to stable disk storage. Lazy VSR is a quorum-based replication protocol that is fast and can recover from simultaneous crashes of all the replicas as long as a majority revive with intact disks. The main idea is to acknowledge requests after recording them in memory, and to write updates to disk in the background, allowing large batched writes and thus good performance. A simultaneous crash of all replicas may leave the replicas with significantly different on-disk states; much of the design of Lazy VSR is concerned with reconciling these states efficiently during recovery. Lazy VSR's client-visible semantics are unusual in that the service may discard recent acknowledged updates if a majority of replicas crash. To demonstrate that clients can nevertheless make good use of Lazy VSR, we built a file system backend on it. Evaluation shows that Lazy VSR achieves much better performance than a version of itself with traditional group commit. Lazy VSR achieves 1.7× the performance of ZooKeeper [42] and 3.6× the performance of MongoDB [6].

Thesis Supervisor: Robert Morris
Title: Professor

Acknowledgments

I would like to thank Robert Morris and Frans Kaashoek for their guidance, encouraging me to think critically, and being my friends. I also thank Nickolai Zeldovich and Eddie Kohler for very fruitful research interactions. I have enjoyed working with all of them. Their broad knowledge, critical thinking and solid engineering skills inspired me.

I thank the following collaborators: Alex, Austin, Cody, Haogang, Neha, Bryan, Michael, Silas, and Xi. Working with them enriched my knowledge base, and left me with a lot of great memories.

Thanks to all my friends and colleagues at PDOS, CSAIL and MIT: Albert, Alex, Amy, Alvin, Austin, Chris, Charles, Cody, Emily, Eugene, Fan, Frank, Haogang, Jelle, Jonas, Kirth, Meelap, Neha, Priya, Raluca, Ramesh, Rasha, Silas, Taesoo, Xi, Yang, Yun, and Zhilei. They helped me generously in both life and work. They made my stay here an enjoyable experience.

Thanks to Frans for inviting me to visit PDOS in 2008. Frans, Robert and Nickolai helped me complete the visit. It was a wonderful time. Thank you. Thanks to Silas for accommodating me at the beginning of my visit.

Thanks to Frans again for encouraging me to apply to MIT's Ph.D. program. It was a great change to my life. I have met many great people here, and learned a lot from them. Thanks to Frans, Robert and Nickolai for supporting my application. Thanks to Silas, who also inspired me and encouraged me to apply. Thanks to Binyu Zang and Zhongding Jiang, my advisors at Fudan University. They encouraged me and supported my application.

Thanks to Quanta and NSF for supporting this work.

Many thanks to my wife, my parents and my sister. They are always there to back me up.

Contents

1 Introduction **13**

1.1 Masstree 13
1.1.1 Failure Model 14
1.1.2 Workload 14
1.1.3 Data and Query Model 14
1.1.4 Semantic 14
1.1.5 Challenges 15
1.1.6 Approach 15
1.1.7 Results 15
1.2 Lazy VSR 15
1.2.1 Background 16
1.2.2 Failure Model 16
1.2.3 Workload 17
1.2.4 Challenges 17
1.2.5 Approach 18
1.2.6 Semantics and Applications 18
1.2.7 Results 19
1.3 Contributions 19
1.4 Organization 19

2 Related Work **21**

3 Fast Multi-Core Key-Value Storage **25**

3.1 Tree Design 25
3.1.1 Overview 26
3.1.2 Layout 28
3.1.3 Nonconcurrent modification 28
3.1.4 Concurrency overview 29
3.1.5 Writer–writer coordination 30
3.1.6 Writer–reader coordination 30
3.1.7 Values 37
3.1.8 Discussion 37
3.2 Networking and persistence 37
3.3 Evaluation 38
3.3.1 Setup 39

3.3.2 Factor analysis . . . 40
3.3.3 System relevance of tree design . . . 41
3.3.4 Flexibility . . . 41
3.3.5 Scalability . . . 42
3.3.6 Partitioning and skew . . . 43
3.4 Discussion . . . 44

4 Fast Fault-Tolerant Replication with Lazy VSR 45
4.1 Background: View-Stamped Replication . . . 45
4.1.1 VSR . . . 45
4.1.2 Simultaneous Crashes and VSR Performance . . . 46
4.2 Design . . . 47
4.2.1 Properties . . . 48
4.2.2 State . . . 49
4.2.3 Operation Within a View . . . 51
4.2.4 Checkpoint and crash recovery . . . 51
4.2.5 View Change . . . 52
4.2.6 Re-Synchronization . . . 54
4.3 Proof . . . 54
4.3.1 Durable Point . . . 55
4.3.2 Proof Sketch of Other Properties . . . 57
4.4 Lazen . . . 58
4.4.1 Sharding . . . 58
4.4.2 Clients . . . 59
4.4.3 Lazen Implementation . . . 60
4.5 Use Cases . . . 60
4.5.1 File System Backend . . . 60
4.5.2 Blizzard . . . 62
4.5.3 MongoDB . . . 62
4.6 Evaluation . . . 63
4.6.1 Throughput and Latency . . . 64
4.6.2 Performance with Synchronous Put . . . 66
4.6.3 Failure Recovery . . . 66
4.6.4 Scalability and Shard Transfer . . . 67
4.6.5 System Comparison . . . 67
4.6.6 File System Backend . . . 68
4.7 Discussion . . . 69

5 Conclusion 71

A TLA Specification and Partial Proof for Lazy VSR 73

List of Figures

3-1 Masstree structure: layers of B^+trees form a trie. 26
3-2 Masstree node structures. 27
3-3 Version number layout. The *locked* bit is claimed by update or insert. *inserting* and *splitting* are "dirty" bits set during inserts and splits, respectively. *vinsert* and *vsplit* are counters incremented after each insert or split. *isroot* tells whether the node is the root of some B^+tree. *isborder* tells whether the node is interior or border. *unused* allows more efficient operations on the version number. 30
3-4 Helper functions. 31
3-5 Split a border node and insert a key. 32
3-6 Find the border node containing a key. 34
3-7 Find the value for a key. 35
3-8 Contributions of design features to Masstree's performance (§3.3.2). Design features are cumulative. Measurements use 16 cores and each server thread generates its own load (no clients or network traffic). Bar numbers give throughput relative to the binary tree running the get workload. 39
3-9 Performance effect of varying key length on Masstree and "+Permuter." For each key length, keys differ only in the last 8 bytes. 16-core get workload. 42
3-10 Masstree scalability. 43
3-11 Throughput of Masstree and hard-partitioned Masstree with various skewness (16-core get workload). 43

4-1 A challenging Lazy VSR recovery scenario. In order to proceed from Time 5, C must roll back its last two operations, and apply the put(x,0). 48
4-2 Per-replica state. 50
4-3 Example of the content of a replica's log at the time of a view change, along with the replica's v and *opMax* values after receipt of each message. The old view number is 21; the new one is 32. The ELECTACCEPT has viewstamp $\langle 32, 309 \rangle$. 53
4-4 Machine configuration. "Disk Thru." is measured file system throughput with sequential writes. "Disk Latency" is the time to overwrite and *fdatasync()* 4096 bytes. Network RTT is user space to user space latency measured with packets with 20-byte payloads. 63
4-5 Lazy VSR has higher throughput than VSR-Sync with 1024-byte put requests. 64
4-6 Lazy VSR has lower latency than VSR-Sync with 1024-byte put requests. . 65

4-7 Lazy VSR has higher throughput than VSR-Sync with 8-byte put requests. . 65
4-8 Lazy VSR has lower latency than VSR-Sync with 8-byte put requests. . . . 65
4-9 Lazy VSR improves throughput over VSR-Sync significantly when the ratio of synchronous puts is relatively small. 66
4-10 Throughput of Lazen with three replica groups. Each request puts an 8-byte value. A shard transfer takes place between time 38 and time 44. 67
4-11 Throughput (req/sec) of different systems for put requests with 1-byte and 1024-byte values. 68
4-12 Performance of our file system backend on Lazy VSR and VSR-Sync . . . 69

List of Tables

Chapter 1

Introduction

This thesis addresses two problems that arise in the design of in-memory storage systems.

On a single multi-core server, performance is typically limited by the latency of DRAM, encountered while looking up or manipulating data items. In order to perform well, the in-memory data structures of the storage system must keep cache miss rates low and hide memory fetch latencies.

The second problem is that, for replicated in-memory storage systems, existing replication protocols are slow, vulnerable to failures, or rely on battery backups. The reason is that they require writing stable storage in the critical path. Most existing systems, such as Raft [60], Zab [44] and Multi-Paxos [47], write to disk synchronously. However, disk latency is high, which limits throughput and increases request latency. Viewstamped Replication (VSR) as described in the Harp File System [55] is fast because each VSR replica writes only to memory in the critical path. To tolerate power failure, each VSR replica uses an uninterruptible power supply (UPS) to allow it to copy its in-memory log to disk during a power failure. However, UPS cannot guard against simultaneous crashes caused by software bugs, CPU overheating, or human errors. These failures, plus power failure and network partition, are referred as "clean" failures in this dissertation because they are non-Byzantine and leave the disk intact.

This dissertation presents a solution for each of these problems: Masstree and Lazy VSR. The two solutions are separate, though they both address different aspects of the overall high-performance storage problem.

1.1 Masstree

Masstree is a high performance in-memory key-value store that runs on a single multi-core server. The key to the design is a concurrent in-memory tree and a set of optimization techniques that reduce per-operation DRAM latencies. Masstree achieves millions of queries per second, which is more than 30× as fast as MongoDB [6] or VoltDB [17]. Masstree is described here as a stand-alone service, though one could use it as a component of a larger multi-server storage system.

1.1.1 Failure Model

Masstree relies on the following assumption about failures. The server may crash, but Masstree assumes that it does so in a way that leaves durable (on-disk) storage intact. That is, from the point of view of the disk, a crash must behave as if the server simply stopped issuing disk writes at some point. In particular, Masstree assumes that the software never issues an incorrect disk write. Masstree assumes that, after repair and/or reboot, the server's disk is intact.

1.1.2 Workload

We want to provide good performance on various workloads. These include write-heavy workloads (which are often more challenging than read-heavy ones); workloads with arbitrary, variable-length keys; workloads with many keys that share long prefixes; workloads with short keys and values (for which low overhead is important); and workloads with large values. The combination of these properties could free performance-sensitive users to use richer data models than is common for stores like memcached today.

1.1.3 Data and Query Model

Masstree supports a simple key-value data model, like many recent systems [6, 11, 5, 46, 68, 27, 32, 35]. We want to support key range queries, so an ordered store (e.g. a tree) is an important feature (and is generally harder for performance than a hash table). While some key-value stores place all keys unordered in an in-memory hash table [11, 5], many production systems order keys and offer range queries [6, 4, 46, 68].

This work doesn't focus on the value representation or query language. Support for value representation and query languages are quite different among different systems. MongoDB [6] represents each value as a document, which can have nested key-value pairs. The Bigtable family of storage systems [27, 46] provides multi-version and (two-level) hierarchical columns per key [27]. While the value representation and the query language are important practical concerns, there is no consensus about the "right" value representation and query language. However, we do want to support multiple columns per key efficiently because it is a common feature provided in most key-value stores.

1.1.4 Semantic

Traditional relational databases provide strong semantics, i.e. atomicity, consistency, isolation and durability (ACID) [1]. However, strong semantics can be costly. For example, durability requires the system to write an update to the disk before replying to the client. Many storage systems [6, 11, 57] offer weakened semantics for higher performance. The downside is that the system is often more difficult to program [34].

Masstree provides weaker semantics. It provides atomicity and isolation for get and put operations, but not for range queries. The rationale behind this choice is to process gets and puts efficiently, which are most common operations in most workloads [33]. Masstree may lose recent acknowledged operations during a failure, but can recover a prefix of completed

operations before the crash. This choice is justified by the following observation: many applications (e.g. file systems [29], POSIX applications [57]) can cope with loss of most recent acknowledged writes, and can recover from a crash as long as a prefix of operations is preserved after the crash. Others have made the same observation [29, 57, 6].

1.1.5 Challenges

The main challenge in achieving high performance on a single machine is reducing the impact of DRAM latency. Modern machines are equipped with multi-core processor(s) and multi-channel DRAM. The single-core performance of an in-memory key-value store may be limited by DRAM latency, which is $2\times$ to $100\times$ higher than the latency of on-chip cache. With multiple cores, a naive implementation may introduce contention for cache lines when shared data is modified, as well as contention for locks, both of which further limit performance.

1.1.6 Approach

Masstree uses a combination of old and new techniques to address DRAM latency and achieve high performance [25, 50, 22, 59, 38, 28, 56]. It achieves fast concurrent operation using a scheme inspired by OLFIT [25], Bronson *et al.* [24], and read-copy update [56]. Lookups use no locks or interlocked instructions, and thus operate without invalidating shared cache lines and in parallel with most inserts and updates. Updates acquire only local locks on the tree nodes involved, allowing modifications to different parts of the tree to proceed in parallel. Masstree shares a single tree among all cores to avoid load imbalances that can occur in partitioned designs. The tree is a trie-like concatenation of B^+trees, and provides high performance even for long common key prefixes, an area in which other tree designs have trouble. Query time is dominated by the total DRAM fetch time of successive nodes during tree descent; to reduce this cost, Masstree uses a wide-fanout tree to reduce the tree depth, prefetches nodes from DRAM to overlap fetch latencies, and carefully lays out data in cache lines to reduce the amount of data needed per node. Operations are logged in batches for crash recovery and the tree is periodically checkpointed.

1.1.7 Results

We evaluate Masstree on a 16-core machine with simple benchmarks and a version of the Yahoo! Cloud Serving Benchmark (YCSB) [33] modified to use small keys and values. Masstree achieves six to ten million operations per second on parts A–C of the benchmark, more than $30\times$ as fast as VoltDB [17] or MongoDB [6].

1.2 Lazy VSR

Lazy VSR is a quorum-based replication protocol that is fast and can recover from simultaneous crashes of all the replicas as long as a majority revive with intact disks. The main idea is to take disk writes off the critical path so that performance is not limited by disk latency.

The trade-off is that Lazy VSR may lose acknowledged writes if a majority of replicas fail simultaneously. However, after such a failure, Lazy VSR can resume operation as long as a majority of the replicas revive with disks intact. To demonstrate the use of Lazy VSR, we built a file system backend on it. Evaluation shows that Lazy VSR achieves much better performance than traditional group commit. It also shows that Lazy VSR achieves $1.7\times$ the performance of ZooKeeper [42] and $3.6\times$ the performance of MongoDB [6].

The intent of replication is to provide better fault-tolerance than a single-machine service. We consider replication within in a single datacenter only, where network latency is relatively low.

1.2.1 Background

Many replication protocols have performance limited by the need to write to durable storage before replying to the client; examples include Multi-Paxos [47], Zab [44], Raft [60], Pacifica [53] and some eventually consistent systems [35, 32]. The durable writes are required to ensure that the replication protocol can eventually reconstruct its state correctly if a majority of replicas should simultaneously fail.

VSR and Harp [54, 55] achieve high performance by writing only to memory in the critical path, and writing the disk in the background. To guard against power failure, each VSR replica has an uninterruptible power supply (UPS) which can power the replica long enough to save its in-memory log to disk during a power failure. However, UPSs can be awkward to install for a large numbers of servers [21]. Furthermore, the UPS only helps with power failures, but does not protect against other kinds of simultaneous failure.

Our goal is to develop a replication protocol that is fast during normal operations and crash recovery, tolerates all clean failures, and doesn't require battery-backups.

Lazy VSR is derived from VSR, and more distantly from Paxos; both are state-machine replication protocols that require a quorum of replicas. A quorum based replication protocol involves $2f+1$ replicas, $f+1$ of which are required to be alive in order for progress to be made. This requirement helps avoid the "split brain" problem that could otherwise arise during network partitions, in which multiple replicas process requests independently. Examples of quorum based replication protocols include Multi-Paxos [47], Zab [44] Viewstamped Replication (VSR) [55, 54] and Raft [60]. They are applied in production systems such as Petal [49], Spanner [34] and ZooKeeper [42].

1.2.2 Failure Model

Lazy VSR remains available as long as at least $f+1$ of $2f+1$ replicas remain alive and in communication. If more than f replicas fail, Lazy VSR cannot immediately continue. It will be able to continue as soon as $f+1$ servers revive with disks intact, though in this case Lazy VSR may discard some acknowledged operations that were submitted immediately before the crashes.

The "with disks intact" requirement means that the replica failed "cleanly:" in a way that did not invalidate the disk contents, and did not write incorrect data to the disk. The most common example of a clean failure is a power failure. However, operating system, hardware, and human errors can also result in clean failures.

If more than f replicas fail and do not revive, or more than f disks fail permanently, Lazy VSR cannot continue.

Compared to replication schemes that write the disk synchronously, Lazy VSR is faster but may lose recent operations if more than f replicas simultaneously fail. Compared to existing replication schemes that use UPS for speed (such as Harp), Lazy VSR can recover from a wider range of simultaneous failures (i.e. more than just power failures).

1.2.3 Workload

Lazy VSR, as a state-machine replication protocol, is largely indifferent to the specifics of the operations it is processing. However, it provides the most performance advantage in particular situations. First, the operations must be short and small enough that improvements in the cost of replication are significant. Second, the number of outstanding requests must be small enough that traditional synchronous group commit does not already saturate the disk. On our test-bed with SSD and Infiniband, tens of thousands of outstanding small requests are required to saturate traditional group commit, yielding a latency of 100s of milliseconds. With fewer outstanding requests, or with a lower tolerance for request latency, Lazy VSR is attractive.

1.2.4 Challenges

Many replicated services need to be able to recover from simultaneous failure of all replicas (e.g., site-wide power failure); a common defense [48, 42] is for each replica to keep its state on a disk. This approach can be unacceptably slow [21]: a disk write may take 10 milliseconds, limiting a straight-forward implementation to 100 operations per second. Group commit [23, 42] helps; however, to achieve high throughput, it can require thousands of concurrent requests and lead to high latency.

Asynchronous disk write is an attractive way to achieve both goals: apply requests to in-memory state initially, and write the disk in the background. This takes the disk write off the critical path for request latency and can increase throughput by delaying and batching multiple writes. Such a system may forget recent operations during a failure despite having acknowledged them; this property requires tolerant clients, but many file systems and even databases behave like this [6, 29, 57].

A more troubling consequence of asynchronous writes is that a multi-server crash may leave replicas with on-disk states that reflect divergent histories, particularly when coupled with network partitions. This makes reconstructing identical application state in all replicas difficult. Loss of recent state changes also makes it hard to resume operation of the replication protocol. Paxos [48], for example, requires promises made during agreement to persist across crashes; failure to do so can result in "agreement" on multiple values. View-Stamped Replication [55] avoids this problem for power failures by equipping replicas with uninterruptible power supplies.

1.2.5 Approach

Lazy VSR acknowledges each client operation after logging it in the volatile RAM of a majority of replicas; this avoids putting a disk write latency on the critical path. Each replica writes its in-memory log to disk in the background; this allows efficient large batched disk writes without increasing latency or requiring many concurrent operations.

A Lazy VSR operation is "tentative" until it is known to have reached disk on a majority of replicas, at which point it becomes durable. Lazy VSR will preserve a tentative operation's effect on state as long as no more than f of $2f+1$ replicas crash. It will preserve a durable operation even if all the replicas crash, as long as a majority eventually reboot with disks intact. These guarantees allow tentative operations to be lost, but also promise to resume the replicated service despite that loss. Because Lazy VSR removes disk writes from the critical path, it decreases latency and increases throughput.

The main challenge Lazy VSR faces is post-crash reconciliation of replicas with conflicting states. For example, suppose replica r_1 becomes partitioned due to a network failure, and r_2 and r_3 crash and lose their memory of recent operations. r_2 and r_3 restart and resume service, with replicated state that omits those recent operations. When r_1's network connection heals, it must recognize that its state reflects operations that are no longer part of the replica group's history; r_1 must discard the corresponding part of its log and roll back the operations' modifications to its state. However, there must be a bound to how much can be rolled back, since otherwise no operation could ever become durable. A key part of Lazy VSR's design is that it ensures that if a majority of replicas have received and logged the same operation to disk, that operation is durable.

1.2.6 Semantics and Applications

The fact that Lazy VSR may discard acknowledged updates limits its use to applications that can cope with this data loss.

One reason to believe that these semantics do not pose an insurmountable problem is that they are similar to those of eventually consistent storage systems, such as Cassandra [46] and Dynamo [35]. Eventual consistency is sufficient for many services and applications that run in data centers [67]. These applications have to be able to work with responses which may not include some previously executed requests [67]. For example, customers of Amazon may see some items they deleted reappear in their shopping carts, in which case the delete operation may have to be redone [67]. Thus, these applications would tolerate Lazy VSR failure in the same way. Use of Lazy VSR would likely decrease programmer burden, since (unlike eventual consistency) Lazy VSR does not suffer from split brain: there is never more than one version of the data.

We have built one application for Lazy VSR: a virtual block store for a file system, or set of independent file systems. The key requirement is that clients (file systems) do not share data. Thus each file system can maintain a log of recent updates it has sent to Lazy VSR, and replay that log if Lazy VSR indicates that it has lost operations. If the client file system fails, the file system would then perform a recovery with a restarted client as if there was a power failure. We describe our implementation of a file system backend and how the client handles failure in this way in section 4.5.1.

1.2.7 Results

We evaluate Lazy VSR with Lazen, an unordered in-memory key-value store replicated with Lazy VSR. It shows that Lazy VSR's advantage over synchronous group commit is particularly high with modest numbers of concurrent client operations. For 1024-byte items, twenty concurrent operations, and SSD storage, Lazy VSR increases Lazen's throughput by a factor of four, and decreases latency by a factor of three. Lazen achieves $1.7\times$ the throughput of ZooKeeper and $3.6\times$ the throughput of MongoDB.

1.3 Contributions

The contributions of Masstree are:

- An in-memory concurrent tree that supports keys with shared prefixes efficiently.
- A set of techniques for laying out the data of each tree node, and accessing it, that reduces the time spent waiting for DRAM while descending the tree.
- A demonstration that a single tree shared among multiple cores can provide higher performance than a partitioned design for some workloads.
- A complete design that addresses all bottlenecks in the way of million-query-per-second performance.

The contribution of Lazy VSR are:

- The demonstration that it is possible to combine state-machine replication with volatile state for increased performance.
- A novel replication protocol to maintain fault-tolerant replication with volatile state.
- A proof shows that operations before the durable point endure, i.e. they are preserved across failures.
- A file system storage backend that offers both high performance and high availability.
- The design of an atomic shard transfer protocol for a storage system built on Lazy VSR.

1.4 Organization

The rest of this dissertation is organized as follows. The next chapter presents related work. We describe the design, implementation and evaluation of Masstree in Chapter 3. Chapter 4 presents the details of Lazy VSR and Lazen. Finally, we briefly discuss the work.

Chapter 2

Related Work

There is much work in the area of high performance key-value storage.

Concurrent In Memory Data Structures. OLFIT [25] is a B^{link}-tree [50] with optimistic concurrency control. Each update to a node changes the node's version number. Lookups check a node's version number before and after observing its contents, and retry if the version number changes (which indicates that the lookup may have observed an inconsistent state). Masstree uses this idea, but, like Bronson *et al.* [24], it splits the version number into two parts; this, and other improvements, lead to less frequent retries during lookup.

PALM [66] is a lock-free concurrent B^+tree with twice the throughput of OLFIT. PALM uses SIMD instructions to take advantage of parallelism *within* each core. Lookups for an entire batch of queries are sorted, partitioned across cores, and processed simultaneously, a clever way to optimize cache usage. PALM requires fixed-length keys and its query batching results in higher query latency than OLFIT and Masstree. Many of its techniques are complementary to our work.

Cache Aware Optimizing Techniques. Bohannon *et al.* [22] store parts of keys directly in tree nodes, resulting in fewer DRAM fetches than storing keys indirectly. AlphaSort [59] explores several ideas to minimize cache misses by storing partial keys. Masstree uses a trie [38] like data structure to achieve the same goal.

Rao *et al.* [61] propose storing each node's children in contiguous memory to make better use of cache. Fewer node pointers are required, and prefetching is simplified, but some memory is wasted on nonexistent nodes. Cha *et al.* report that a fast B^+tree outperforms a CSB^+tree [26]; Masstree improves cache efficiency using more local techniques.

Data-cache stalls are a major bottleneck for database systems, and many techniques have been used to improve caching [40, 30, 31, 63]. Chen *et al.* [28] prefetch tree nodes; Masstree adopts this idea.

In-Memory Storage Systems Deployed on Single Machine. H-Store [68, 45] and VoltDB, its commercial version, are in-memory relational databases designed to be orders of magnitude faster than previous systems. To take advantage of multi-core processors on a single machine, VoltDB runs multiple single threaded instances and partitions data among instances. This avoids concurrency, and thus avoids data structure locking costs. In contrast, Masstree shares data among all cores to avoid load imbalances that can occur with partitioned data, and achieves good scaling with lock-free lookups and locally locked

inserts.

Shore-MT [43] identifies lock contention as a major bottleneck for multicore databases, and improves performance by removing locks incrementally. Masstree provides high concurrency from the start.

Recent key-value stores [6, 27, 11, 35, 46] provide high performance partially by offering a simpler query and data model than relational databases, and partially by partitioning data over a cluster of servers. Masstree adopts the first idea. Its design focuses on multicore performance rather than clustering, though in principle one could operate a cluster of Masstree servers.

Replicated Storage Systems using Battery-Backup Hardware. Harp [55] uses VSR. It writes its log to disk asynchronously, and uses uninterruptible power supplies (UPS) to tolerate simultaneous power failure. VSR Revisited [54] does not use disk at all, but does require its memory to be non-volatile. Neither Harp nor VSR Revisited can recover from simultaneous crashes of more than f replicas that cause memory to be lost. Lazy VSR has more relaxed configuration requirements (just that replicas have disks), can recover from a wider range of failures (e.g., if $f+1$ replicas crash and lose their memory), and can be expected to have similar performance (since Lazy VSR keeps the disk off the critical path). The cost is that Lazy VSR may forget about recent acknowledged operations if more than f replicas lose their memory.

RAMCloud [36] initially logs updates only in memory, and periodically writes log segments to disk. The design requires that the in-memory log not be lost during crashes, suggesting use of a UPS or non-volatile RAM.

Replicated Storage Systems with Synchronous Disk Writes. ZooKeeper [42] uses a replicated state machine protocol [44] similar to VSR. Lazen uses a number of detailed techniques similar to those of ZooKeeper: data that fits in memory, periodic checkpoints, and crash recovery from checkpoints and log. A ZooKeeper replica waits for an operation to be logged to disk before responding to the primary, but uses group commit to improve throughput. Operation latency is roughly 40 milliseconds [44] due to the synchronous on-disk log write; it is this latency that Lazy VSR primarily addresses.

Spinnaker [62] is a replicated key-value store built on Multi-Paxos. Multi-Paxos designates a leader for many future Paxos agreements, which allows it to commit each operation using just two network round trips in most cases. However, replicas in Spinnaker write to disk synchronously and suffer from disk latency.

Spanner [34] uses Paxos [48] to build a fault-tolerant database with sharding. While the paper does not explain how shard transfer works, the use of Paxos suggests that it may work in a way similar to that of Lazen.

Scatter [39] is a peer-to-peer key/value store that supports sharding. It uses two phase commit and replication to atomically alter the assignment of shards to replica groups. However, unlike Lazy VSR, Scatter doesn't deal with simultaneous network or power failure.

Gaios [23] is a storage service replicated with Paxos. Gaios uses pipelining and group commit to improve performance, and the authors report it can execute 3,200 small operations per second using hard drives. Gaios is likely limited by its use of synchronous disk writes, which perhaps explains why we observe much higher throughput for Lazy VSR.

CORFU [18] is a replicated append-only log service using SSDs. It is a simpler service than Lazy VSR. The reported replication performance is 70,000 ops/second on a 32-SSD

cluster. Lazy VSR achieves close to this performance with a tenth as many SSDs.

Replicated Storage Systems with Asynchronous Replication. A MongoDB [6] replica logs to memory, then replies, then writes the on-disk log asynchronously. A single crashed replica recovers by copying the tail of another replica's log. Simultaneous replica crashes are likely to cause the replica group to lose recent client operations. While MongoDB can recover from many such situations, it lacks a strong mechanism for sequencing primaries and operations; this causes recovery to result in inconsistent replicas [16], or to lose data after a network partition [2]. Another problem is that MongoDB cannot recover from failures during certain operations [7]. Lazy VSR combines the speed of asynchronous logging with better recovery properties.

Redis also support asynchronous replication [12]. The master processes a client request and replies to the client, without waiting for the background thread to propagate the client request to slaves. As a result, Redis may also lose data during a majority failure. However, Redis may perform full resynchronization more often than Lazy VSR. For example, when the master failed (either partitioned or crashed), all slaves will discard their local copy of the database and copy the whole database from the new master. Lazy VSR only needs to perform full copy if disk fails.

VoltDB [17] supports a rich set of replication and logging configurations. The command log can be logged synchronously or asynchronously, and replicated synchronously or asynchronously among replicas. Synchronous command logging is likely to be slow. Thus, VoltDB recommend to use a dedicated disk with BBWC controller. Lazy VSR's performance doesn't rely on the responsiveness of the disk. With asynchronous logging or replication, VoltDB may lose data, but it will recover by copying the whole database from the new master. Lazy VSR performs full copy less frequently.

Byzantine Fault Tolerant Protocol Zeno [67] is a BFT protocol that provides high availability and eventual consistency. For high availability, the Zeno replica can provide service as long as there is a quorum of $f+1$ replicas, while traditional BFT requires a quorum of $2f+1$ replicas. The trade off is that there can be multiple primaries/partitions at one time, leading to conflicts. Zeno provides eventual consistency to deal with conflicts. When conflicts are detected, Zeno merges conflicted operations. Not all operations are merged. Operations are either weak or strong. The order of strong operations never changes and are never discarded. The order of weak operations may change during a merge, and some conflicting operations may be discarded. Lazy VSR doesn't handle BFT failure. However, Lazy VSR offers stronger consistency and the clients observers simpler behavior after a failure. In Lazy VSR, there is at most one active primary at a time. Thus, after a failure, a client won't see its operations being merged with others, but loss of a suffix of its operations.

Chapter 3

Fast Multi-Core Key-Value Storage

This chapter presents Masstree, an in-memory key/value storage system for a single multi-core machine. Masstree is a complete system which addresses all the bottlenecks in the way of achieving millions of queries per second across various workloads. Masstree stores all key/value pairs in a single in-memory tree. Masstree optimizes its use of processor caches to achieve high single-core performance and good scalability, and uses a set of techniques to allow high multi-core parallelism. Masstree provides durability via write-ahead logging, and can recover a prefix of operations completed before the crash.

Masstree is implemented as a network key-value storage server. Its requests query and change the mapping of keys to values. Values can be further divided into columns, each of which is an uninterpreted byte string.

Masstree supports four operations: $get_c(k)$, $put_c(k,v)$, $remove(k)$, and $getrange_c(k,n)$. The c parameter is an optional list of column numbers that allows clients to get or set subsets of a key's full value. The *getrange* operation, also called "scan," implements a form of range query. It returns up to n key-value pairs, starting with the next key at or after k and proceeding in lexicographic order by key. *Getrange* is not atomic with respect to inserts and updates. A single client message can include many queries.

3.1 Tree Design

Our key data structure is Masstree, a shared-memory, concurrent-access data structure combining aspects of B$^+$trees [19] and tries [38]. Masstree offers fast random access and stores keys in sorted order to support range queries. The design was shaped by three challenges. First, Masstree must efficiently support many key distributions, including variable-length binary keys where many keys might have long common prefixes. Second, for high performance and scalability, Masstree must allow fine-grained concurrent access, and its get operations must never dirty shared cache lines by writing shared data structures. Third, Masstree's layout must support prefetching and collocate important information on small numbers of cache lines. The second and third properties together constitute cache craftiness.

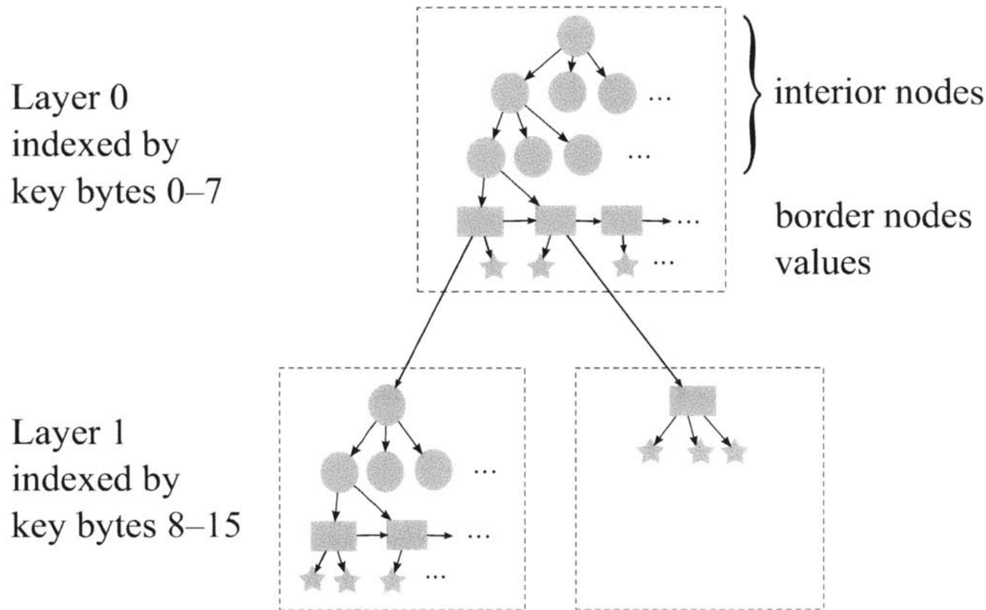


Figure 3-1: Masstree structure: layers of B^+trees form a trie.

3.1.1 Overview

A Masstree is a trie with fanout 2^{64} where each trie node is a B^+tree. The trie structure efficiently supports long keys with shared prefixes; the B^+tree structures efficiently support short keys and fine-grained concurrency, and their medium fanout uses cache lines effectively.

Put another way, a Masstree comprises one or more *layers* of B^+trees, where each layer is indexed by a different 8-byte *slice* of key. Figure 3-1 shows an example. The trie's single root tree, layer 0, is indexed by the slice comprising key bytes 0–7, and holds all keys up to 8 bytes long. Trees in layer 1, the next deeper layer, are indexed by bytes 8–15; trees in layer 2 by bytes 16–23; and so forth.

Each tree contains at least one *border* node and zero or more *interior* nodes. Border nodes resemble leaf nodes in conventional B^+trees, but where leaf nodes store only keys and values, Masstree border nodes can also store pointers to deeper trie layers.

Keys are generally stored as close to the root as possible, subject to three invariants. (1) Keys shorter than $8h+8$ bytes are stored at layer $\leq h$. (2) Any keys stored in the same layer-h tree have the same $8h$-byte prefix. (3) When two keys share a prefix, they are stored at least as deep as the shared prefix. That is, if two keys longer than $8h$ bytes have the same $8h$-byte prefix, then they are stored at layer $\geq h$.

Masstree creates layers as needed (as is usual for tries). Key insertion prefers to use existing trees; new trees are created only when insertion would otherwise violate an invariant. Key removal deletes completely empty trees but does not otherwise rearrange keys. For example, if t begins as an empty Masstree:

1. *t.put*("01234567AB") stores key "01234567AB" in the root layer. The relevant key slice, "01234567", is stored separately from the 2-byte suffix "AB". A *get* for this key first searches for the slice, then compares the suffix.

```
struct interior_node:
    uint32_t version;
    uint8_t nkeys;
    uint64_t keyslice[15];
    node* child[16];
    interior_node* parent;

union link_or_value:
    node* next_layer;
    [opaque] value;

struct border_node:
    uint32_t version;
    uint8_t nremoved;
    uint8_t keylen[15];
    uint64_t permutation;
    uint64_t keyslice[15];
    link_or_value lv[15];
    border_node* next;
    border_node* prev;
    interior_node* parent;
    keysuffix_t keysuffixes;
```

Figure 3-2: Masstree node structures.

2. *t.put*("01234567XY"): Since this key shares an 8-byte prefix with an existing key, Masstree must create a new layer. The values for "01234567AB" and "01234567XY" are stored, under slices "AB" and "XY", in a freshly allocated B^+tree border node. This node then replaces the "01234567AB" entry in the root layer. Concurrent gets observe either the old state (with "01234567AB") or the new layer, so the "01234567AB" key remains visible throughout the operation.

3. *t.remove*("01234567XY") traverses through the root layer to the layer-1 B^+tree, where it deletes key "XY". The "AB" key remains in the layer-1 B^+tree.

Balance A Masstree's shape depends on its key distribution. For example, 1000 keys that share a 64-byte prefix generate at least 8 layers; without the prefix they would fit comfortably in one layer. Despite this, Masstrees have the same query complexity as B-trees. Given n keys of maximum length ℓ, query operations on a B-tree examine $O(\log n)$ nodes and make $O(\log n)$ key comparisons; but since each key has length $O(\ell)$, the total comparison cost is $O(\ell \log n)$. A Masstree will make $O(\log n)$ comparisons in each of $O(\ell)$ layers, but each comparison considers *fixed-size* key slices, for the same total cost of $O(\ell \log n)$. When keys have long common prefixes, Masstree outperforms conventional balanced trees, performing $O(\ell + \log n)$ comparisons per query (ℓ for the prefix plus $\log n$ for the suffix). However, Masstree's range queries have higher worst-case complexity than in a B^+tree, since they must traverse multiple layers of tree.

Partial-key B-trees [22] can avoid some key comparisons while preserving true balance. However, unlike these trees, Masstree bounds the number of non-node memory references required to find a key to at most one per lookup. Masstree lookups, which focus on 8-byte key slice comparisons, are also easy to code efficiently. Though Masstree can use more memory on some key distributions, since its nodes are relatively wide, it outperformed our pkB-tree implementation on several benchmarks by 20% or more.

3.1.2 Layout

Figure 3-2 defines Masstree's node structures. At heart, Masstree's interior and border nodes are internal and leaf nodes of a B^+tree with width 15. Border nodes are linked to facilitate *remove* and *getrange*. The *version*, *nremoved*, and *permutation* fields are used during concurrent updates and described below; we now briefly mention other features.

The *keyslice* variables store 8-byte key slices as 64-bit integers, byte-swapped if necessary so that native less-than comparisons provide the same results as lexicographic string comparison. This was the most valuable of our coding tricks, improving performance by 13–19%. Short key slices are padded with 0 bytes.

Border nodes store key slices, lengths, and suffixes. Lengths, which distinguish different keys with the same slice, are a consequence of our decision to allow binary strings as keys. Since null characters are valid within key strings, Masstree must for example distinguish the 8-byte key "ABCDEFG\0" from the 7-byte key "ABCDEFG", which have the same slice representation.

A single tree can store at most 10 keys with the same slice, namely keys with lengths 0 through 8 plus either one key with length > 8 or a link to a deeper trie layer.[1] We ensure that all keys with the same slice are stored in the same border node. This simplifies and slims down interior nodes, which need not contain key lengths, and simplifies the maintenance of other invariants important for concurrent operation, at the cost of some checking when nodes are split. (Masstree is in this sense a restricted type of prefix B-tree [20].)

Border nodes store the suffixes of their keys in *keysuffixes* data structures. These are located either inline or in separate memory blocks; Masstree adaptively decides how much per-node memory to allocate for suffixes and whether to place that memory inline or externally. Compared to a simpler technique (namely, allocating fixed space for up to 15 suffixes per node), this approach reduces memory usage by up to 16% for workloads with short keys and improves performance by 3%.

Values are stored in link_or_value unions, which contain either values or pointers to next-layer trees. These cases are distinguished by the *keylen* field. Users have full control over the bits stored in *value* slots.

Masstree's performance is dominated by the latency of fetching tree nodes from DRAM. Many such fetches are required for a single *put* or *get*. Masstree prefetches all of a tree node's cache lines in parallel before using the node, so the entire node can be used after a single DRAM latency. Up to a point, this allows larger tree nodes to be fetched in the same amount of time as smaller ones; larger nodes have wider fanout and thus reduce tree height. On our hardware, tree nodes of four cache lines (256 bytes, which allows a fanout of 15) provide the highest total performance.

3.1.3 Nonconcurrent modification

Masstree's tree modification algorithms are based on sequential algorithms for B^+tree modification. We describe them as a starting point.

[1] At most one key can have length > 8 because of the invariants above: the second such key will create the deeper trie layer. Not all key slices can support 10 keys—any slice whose byte 7 is not null occurs at most twice.

Inserting a key into a full border node causes a *split*. A new border node is allocated, and the old keys (plus the inserted key) are distributed among the old and new nodes. The new node is then inserted into the old node's parent interior node; if full, this interior node must itself be split (updating its children's *parent* pointers). The split process terminates either at a node with insertion room or at the root, where a new interior node is created and installed. Removing a key simply deletes it from the relevant border node. Empty border nodes are then freed and deleted from their parent interior nodes. This process, like split, continues up the tree as necessary. Though remove in classical B^+trees can redistribute keys among nodes to preserve balance, removal without rebalancing has theoretical and practical advantages [65].

Insert and remove maintain a per-tree doubly linked list among border nodes. This list speeds up range queries in either direction. If only forward range queries were required, a singly linked list could suffice, but the backlinks are required anyway for our implementation of concurrent remove.

We apply common case optimizations. For example, sequential insertions are easy to detect (the item is inserted at the end of a node with no *next* sibling). If a sequential insert requires a split, the old node's keys remain in place and Masstree inserts the new item into an empty node. This improves memory utilization and performance for sequential workloads. (Berkeley DB and others also implement this optimization.)

3.1.4 Concurrency overview

Masstree achieves high performance on multicore hardware using fine-grained locking and optimistic concurrency control. Fine-grained locking means writer operations in different parts of the tree can execute in parallel: an update requires only local locks.[2] Optimistic concurrency control means reader operations, such as *get*, acquire no locks whatsoever, and in fact *never write to globally-accessible shared memory*. Writes to shared memory can limit performance by causing contention—for example, contention among readers for a node's read lock—or by wasting DRAM bandwidth on writebacks. But since readers don't lock out concurrent writers, readers might observe intermediate states created by writers, such as partially-inserted keys. Masstree readers and writers must cooperate to avoid confusion. The key communication channel between them is a per-node *version* counter that writers mark as "dirty" before creating intermediate states, and then increment when done. Readers snapshot a node's *version* before accessing the node, then compare this snapshot to the *version* afterwards. If the versions differ or are dirty, the reader may have observed an inconsistent intermediate state and must retry.

Our optimistic concurrency control design was inspired by read-copy update [56], and borrows from OLFIT [25] and Bronson *et al.*'s concurrent AVL trees [24].

Masstree's correctness condition can be summarized as *no lost keys*: A $get(k)$ operation must return a correct value for k, regardless of concurrent writers. (When $get(k)$ and $put(k,v)$ run concurrently, the *get* can return either the old or the new value.) The biggest challenge in preserving correctness is concurrent splits and removes, which can shift re-

[2]These data structure locks are often called "latches," with the word "lock" reserved for transaction locks. We do not discuss transactions or their locks.

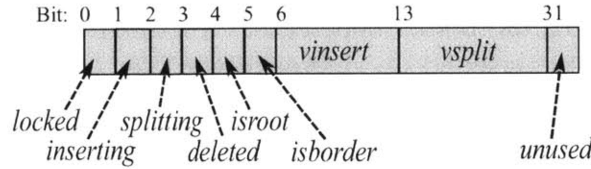


Figure 3-3: Version number layout. The *locked* bit is claimed by update or insert. *inserting* and *splitting* are "dirty" bits set during inserts and splits, respectively. *vinsert* and *vsplit* are counters incremented after each insert or split. *isroot* tells whether the node is the root of some B^+tree. *isborder* tells whether the node is interior or border. *unused* allows more efficient operations on the version number.

sponsibility for a key away from a subtree even as a reader traverses that subtree.

3.1.5 Writer–writer coordination

Masstree writers coordinate using per-node spinlocks. A node's lock is stored in a single bit in its *version* counter. (Figure 3-3 shows the version counter's layout.)

Any modification to a node's keys or values requires holding the node's lock. Some data is protected by other nodes' locks, however. A node's *parent* pointer is protected by its parent's lock, and a border node's *prev* pointer is protected by its previous sibling's lock. This minimizes the simultaneous locks required by split operations; when an interior node splits, for example, it can assign its children's *parent* pointers without obtaining their locks.

Splits and node deletions require a writer to hold several locks simultaneously. When node n splits, for example, the writer must simultaneously hold n's lock, n's new sibling's lock, and n's parent's lock. (The simultaneous locking prevents a concurrent split from moving n, and therefore its sibling, to a different parent before the new sibling is inserted.) As with B^{link}-trees [50], lock ordering prevents deadlock: locks are always acquired up the tree.

We evaluated several writer–writer coordination protocols on different tree variants, including lock-free algorithms relying on compare-and-swap operations. The current locking protocol performs as well or better. On current cache-coherent shared-memory multicore machines, the major cost of locking, namely the cache coherence protocol, is also incurred by lock-free operations like compare-and-swap, and Masstree never holds a lock for very long.

3.1.6 Writer–reader coordination

We now turn to writer–reader coordination, which uses optimistic concurrency control. Note that even an all-put workload involves some writer–reader coordination, since the initial *put* phase that reaches the node responsible for a key is logically a reader and takes no locks.

It's simple to design a correct, though inefficient, optimistic writer–reader coordination algorithm using *version* fields.

```
stableversion(node n):
        v ← n.version
        while v.inserting or v.splitting:
            v ← n.version
        return v

lock(node n):
        while n ≠ NIL and swap(n.version.locked, 1) = 1:
            // retry

unlock(node n):                                   // implemented with one memory write
        if n.version.inserting:
            ++n.version.vinsert
        else if n.version.splitting:
            ++n.version.vsplit
        n.version.{locked, inserting, splitting} ← 0

lockedparent(node n):
retry:  p ← n.parent; lock(p)
        if p ≠ n.parent:                          // parent changed underneath us
            unlock(p); goto retry
        return p
```

Figure 3-4: Helper functions.

1. Before making any change to a node n, a writer operation must mark *n.version* as "dirty." After making its change, it clears this mark and increments the *n.version* counter.

2. Every reader operation first snapshots *every* node's *version*. It then computes, keeping track of the nodes it examines. After finishing its computation (but before returning the result), it checks whether any examined node's *version* was dirty or has changed from the snapshot; if so, the reader must retry with a fresh snapshot.

Universal before-and-after version checking would clearly ensure that readers detect any concurrent split (assuming version numbers didn't wrap mid-computation[3]). It would equally clearly perform terribly. Efficiency is recovered by eliminating unnecessary version changes, by restricting the version snapshots readers must track, and by limiting the scope over which readers must retry. The rest of this section describes different aspects of coordination by increasing complexity.

Updates

Update operations, which change values associated with existing keys, must prevent concurrent readers from observing intermediate results. This is achieved by atomically updat-

[3]Our current counter could wrap if a reader blocked mid-computation for 2^{22} inserts. A 64-bit version counter would never overflow in practice.

```
split(node n, key k):                                   // precondition: n locked
        n′ ← new border node
        n.version.splitting ← 1
        n′.version ← n.version                          // n′ is initially locked
        split keys among n and n′, inserting k
ascend: p ← lockedparent(n)                             // hand-over-hand locking
        if p = NIL:                                     // n was old root
            create a new interior node p with children n, n′
            unlock(n); unlock(n′); return
        else if p is not full:
            p.version.inserting ← 1
            insert n′ into p
            unlock(n); unlock(n′); unlock(p); return
        else:
            p.version.splitting ← 1
            unlock(n)
            p′ ← new interior node
            p′.version ← p.version
            split keys among p and p′, inserting n′
            unlock(n′); n ← p; n′ ← p′; goto ascend
```

Figure 3-5: Split a border node and insert a key.

ing values using aligned write instructions. On modern machines, such writes have atomic effect: any concurrent reader will see either the old value or the new value, not some unholy mixture. Updates therefore don't need to increment the border node's version number, and don't force readers to retry.

However, writers must not delete old values until all concurrent readers are done examining them. We solve this garbage collection problem with read-copy update techniques, namely a form of epoch-based reclamation [37]. All data accessible to readers is freed using similar techniques.

Border inserts

Insertion in a conventional B-tree leaf rearranges keys into sorted order, which creates invalid intermediate states. One solution is forcing readers to retry, but Masstree's border-node *permutation* field makes each insert visible in one atomic step instead. This solves the problem by eliminating invalid intermediate states. The *permutation* field compactly represents the correct key order plus the current number of keys, so writers expose a new sort order *and* a new key with a single aligned write. Readers see either the old order, without the new key, or the new order, with the new key in its proper place. No key rearrangement, and therefore no version increment, is required.

The 64-bit *permutation* is divided into 16 four-bit subfields. The lowest 4 bits, *nkeys*, holds the number of keys in the node (0–15). The remaining bits constitute a fifteen-element array, *keyindex*[15], containing a permutation of the numbers 0 through 15. El-

ements *keyindex*[0] through *keyindex*[*nkeys* − 1] store the indexes of the border node's live keys, in increasing order by key. The other elements list currently-unused slots. To insert a key, a writer locks the node; loads the permutation; rearranges the permutation to shift an unused slot to the correct insertion position and increment *nkeys*; writes the new key and value to the previously-unused slot; and finally writes back the new permutation and unlocks the node. The new key becomes visible to readers only at this last step.

A compiler fence, and on some architectures a machine fence instruction, is required between the writes of the key and value and the write of the permutation. Our implementation includes fences whenever required, such as in *version* checks.

New layers

Masstree creates a new layer when inserting a key k_1 into a border node that contains a conflicting key k_2. It allocates a new empty border node n', inserts k_2's current value into it under the appropriate key slice, and then replaces k_2's value in n with the *next_layer* pointer n'. Finally, it unlocks n and continues the attempt to insert k_1, now using the newly created layer n'.

Since this process only affects a single key, there is no need to update n's *version* or *permutation*. However, readers must reliably distinguish true values from *next_layer* pointers. Since the pointer and the layer marker are stored separately, this requires a sequence of writes. First, the writer marks the key as UNSTABLE; readers seeing this marker will retry. It then writes the *next_layer* pointer, and finally marks the key as a LAYER.

Splits

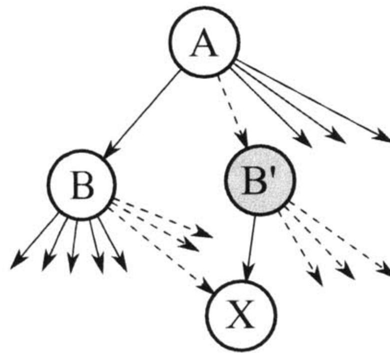
Splits, unlike non-split inserts, remove active keys from a visible node and insert them in another. Without care, a *get* concurrent with the split might mistakenly report these shifting keys as lost. Writers must therefore update *version* fields to signal splits to readers. The challenge is to update these fields in writers, and check them in readers, in such a way that no change is lost.

Figures 3-5 and 3-6 present pseudocode for splitting a border node and for traversing down a B$^+$tree to the border node responsible for a key. (Figure 3-4 presents some helper functions.) The split code uses hand-over-hand locking and marking [24]: lower levels of the tree are locked and marked as "splitting" (a type of dirty marking) before higher levels. Conversely, the traversal code checks versions hand-over-hand in the opposite direction: higher levels' versions are verified before the traversal shifts to lower levels.

To see why this is correct, consider an interior node B that splits to create a new node B′:

```
findborder(node root, key k):
retry:    n ← root; v ← stableversion(n)
          if v.isroot is false:
              root ← root.parent; goto retry
descend:  if n is a border node:
              return ⟨n, v⟩
          n′ ← child of n containing k
          v′ ← stableversion(n′)
          if n.version ⊕ v ≤ "locked":                 // hand-over-hand validation
              n ← n′; v ← v′; goto descend
          v″ ← stableversion(n)
          if v″.vsplit ≠ v.vsplit:
              goto retry                                // if split, retry from root
          v ← v″; goto descend                          // otherwise, retry from n
```

Figure 3-6: Find the border node containing a key.



(Dashed lines from B indicate child pointers that were shifted to B′.) The split procedure changes versions and shifts keys in the following steps.

1. B and B′ are marked *splitting*.
2. Children, including X, are shifted from B to B′.
3. A (B's parent) is locked and marked *inserting*.
4. The new node, B′, is inserted into A.
5. A, B, and B′ are unlocked, which increments the A *vinsert* counter and the B and B′ *vsplit* counters.

Now consider a concurrent findborder(X) operation that starts at node A. We show that this operation either finds X or eventually retries. First, if findborder(X) traverses to node B′, then it will find X, which moved to B′ (in step 2) before the pointer to B′ was published (in step 4). Instead, assume findborder(X) traverses to B. Since the findborder operation retries on any version difference, and since findborder loads the child's version

```
get(node root, key k):
retry:   ⟨n, v⟩ ← findborder(root, k)
forward: if v.deleted:
             goto retry
         ⟨t, lv⟩ ← extract link_or_value for k in n
         if n.version ⊕ v > "locked":
             v ← stableversion(n); next ← n.next
             while !v.deleted and next ≠ NIL and k ≥ lowkey(next):
                 n ← next; v ← stableversion(n); next ← n.next
             goto forward
         else if t = NOTFOUND:
             return NOTFOUND
         else if t = VALUE:
             return lv.value
         else if t = LAYER:
             root ← lv.next_layer; advance k to next slice
             goto retry
         else: // t = UNSTABLE
             goto forward
```

Figure 3-7: Find the value for a key.

before double-checking the parent's ("hand-over-hand validation" in Figure 3-6), we know that findborder loaded B's version before A was marked as *inserting* (step 3). This in turn means that the load of B's version happened before step 1. (That step marks B as *splitting*, which would have caused stableversion to retry.) Then there are two possibilities. If findborder completes before the split operation's step 1, it will clearly locate node X. On the other hand, if findborder is delayed past step 1, it will always detect a split and retry from the root. The $\mathrm{B}.version \oplus v$ check will fail because of B's *splitting* flag; the following stableversion(B) will delay until that flag is cleared, which happens when the split executes step 5; and at that point, B's *vsplit* counter has changed.

Masstree readers treat splits and inserts differently. Inserts retry locally, while splits require retrying from the root. Wide B-tree fanout and fast code mean concurrent splits are rarely observed: in an insert test with 8 threads, less than 1 insert in 10^6 had to retry from the root due to a concurrent split. Other algorithms, such as backing up the tree step by step, were more complex to code but performed no better. However, concurrent inserts are (as one might expect) observed $15\times$ more frequently than splits. It is simple to handle them locally, so Masstree maintains separate split and insert counters to distinguish the cases.

Figure 3-7 shows full code for Masstree's *get* operation. (Puts are similar, but since they obtain locks, the retry logic is simpler.) Again, the node's contents are extracted between checks of its *version*, and *version* changes cause retries.

Border nodes, unlike interior nodes, can handle splits using their links.[4] The key invari-

[4] $\mathrm{B}^{\mathrm{link}}$-trees [50] and OLFIT [25] also link interior nodes, but our "B^{-} tree" implementation of remove [65] breaks the invariants that make this possible.

ant is that nodes split "to the right": when a border node n splits, its *higher* keys are shifted to its new sibling. Specifically, Masstree maintains the following invariants:

- The initial node in a B^+tree is a border node. This node is not deleted until the B^+tree itself is completely empty, and always remains the leftmost node in the tree.
- Every border node n is responsible for a range of keys $[\text{lowkey}(n), \text{highkey}(n))$. (The leftmost and rightmost nodes have $\text{lowkey}(n) = -\infty$ and $\text{highkey}(n) = \infty$, respectively.) Splits and deletes can modify $\text{highkey}(n)$, but $\text{lowkey}(n)$ remains constant over n's lifetime.

Thus, *get* can reliably find the relevant border node by comparing the current key and the next border node's lowkey.

The first lines of findborder (Figure 3-6) handle stale roots caused by concurrent splits, which can occur at any layer. When the layer-0 global root splits, we update it immediately, but other roots, which are stored in border nodes' *next_layer* pointers, are updated lazily during later operations.

Removes

Masstree, unlike some prior work [50, 25], includes a full implementation of concurrent remove. Space constraints preclude a full discussion, but we mention several interesting features.

First, *remove* operations, when combined with inserts, must sometimes cause readers to retry! Consider the following threads running in parallel on a one-node tree:

get(n, k_1): locate k_1 at n position i	

⋮	remove(n, k_1):
	  remove k_1 from n position i
	put(n, k_2, v_2):
	  insert k_2, v_2 at n position j

$lv \leftarrow n.lv[i]$; check $n.version$; return $lv.value$	

The *get* operation may return k_1's (removed) value, since the operations overlapped. *Remove* thus must not clear the memory corresponding to the key or its value: it just changes the *permutation*. But then if the *put* operation happened to pick $j = i$, the *get* operation might return v_2, which isn't a valid value for k_1. Masstree must therefore update the *version* counter's *vinsert* field when removed slots are reused.

When a border node becomes empty, Masstree removes it and any resulting empty ancestors. This requires the border-node list be doubly-, not singly-, linked. A naive implementation could break the list under concurrent splits and removes; compare-and-swap operations (some including flag bits) are required for both split and remove, which slightly slows down split. As with any state observable by concurrent readers, removed nodes must not be freed immediately. Instead, we mark them as *deleted* and reclaim them later. Any

operation that encounters a *deleted* node retries from the root. *Remove*'s code for manipulating interior nodes resembles that for split; hand-over-hand locking is used to find the right key to remove. Once that key is found, the *deleted* node becomes completely unreferenced and future readers will not encounter it.

Removes can delete entire layer-h trees for $h \geq 1$. These are not cleaned up right away: normal operations lock at most one layer at a time, and removing a full tree requires locking both the empty layer-h tree and the layer-$(h-1)$ border node that points to it. Epoch-based reclamation tasks are scheduled as needed to clean up empty and pathologically-shaped layer-h trees.

3.1.7 Values

The Masstree system stores values consisting of a version number and an array of variable-length strings called *columns*. Gets can retrieve multiple columns (identified by integer indexes) and puts can modify multiple columns. Multi-column puts are atomic: a concurrent *get* will see either all or none of a *put*'s column modifications.

Masstree includes several value implementations; we evaluate one most appropriate for small values. Each value is allocated as a single memory block. Modifications don't act in place, since this could expose intermediate states to concurrent readers. Instead, put creates a new value object, copying unmodified columns from the old value object as appropriate. This design uses cache effectively for small values, but would cause excessive data copying for large values; for those, Masstree offers a design that stores each column in a separately-allocated block.

3.1.8 Discussion

More than 30% of the cost of a Masstree lookup is in computation (as opposed to DRAM waits), mostly due to key search within tree nodes. Linear search has higher complexity than binary search, but exhibits better locality. For Masstree, the performance difference of the two search schemes is architecture dependent. On an Intel processor, linear search can be up to 5% faster than binary search. On an AMD processor, both perform the same.

One important PALM optimization is parallel lookup [66]. This effectively overlaps the DRAM fetches for many operations by looking up the keys for a batch of requests in parallel. Our implementation of this technique did not improve performance on our 48-core AMD machine, but on a 24-core Intel machine, throughput rose by up to 34%. We plan to change Masstree's network stack to apply this technique.

3.2 Networking and persistence

Masstree uses network interfaces that support per-core receive and transmit queues, which reduce contention when short query packets arrive from many clients. To support short connections efficiently, Masstree can configure per-core UDP ports that are each associated with a single core's receive queue. Our benchmarks, however, use long-lived TCP query

connections from few clients (or client aggregators), a common operating mode that is equally effective at avoiding network overhead.

Masstree logs updates to persistent storage to achieve persistence and crash recovery. Each server query thread (core) maintains its own log file and in-memory log buffer. A corresponding logging thread, running on the same core as the query thread, writes out the log buffer in the background. Logging thus proceeds in parallel on each core.

A put operation appends to the query thread's log buffer and responds to the client without forcing that buffer to storage. Logging threads batch updates to take advantage of higher bulk sequential throughput, but force logs to storage at least every 200 ms for safety. Different logs may be on different disks or SSDs for higher total log throughput.

Value version numbers and log record timestamps aid the process of log recovery. Sequential updates to a value obtain distinct, and increasing, version numbers. Update version numbers are written into the log along with the operation, and each log record is timestamped. When restoring a database from logs, Masstree sorts logs by timestamp. It first calculates the recovery cutoff point, which is the minimum of the logs' last timestamps, $\tau = \min_{\ell \in L} \max_{u \in \ell} u.timestamp$, where L is the set of available logs and u denotes a single logged update. Masstree plays back the logged updates in parallel, taking care to apply a value's updates in increasing order by version, except that updates with $u.timestamp \geq \tau$ are dropped.

Masstree periodically writes out a checkpoint containing all keys and values. This speeds recovery and allows log space to be reclaimed. Recovery loads the latest valid checkpoint that completed before τ, the log recovery time, and then replays logs starting from the timestamp at which the checkpoint began.

Our checkpoint facility is independent of the Masstree design; we include it to show that persistence need not limit system performance, but do not evaluate it in depth. It takes Masstree 58 seconds to create a checkpoint of 140 million key-value pairs (9.1 GB of data in total), and 38 seconds to recover from that checkpoint. The main bottleneck for both is imbalance in the parallelization among cores. Checkpoints run in parallel with request processing. When run concurrently with a checkpoint, a put-only workload achieves 72% of its ordinary throughput due to disk contention.

3.3 Evaluation

We evaluate Masstree in two parts. In this section, we focus on Masstree's central data structure, the trie of B$^+$trees. We show the cumulative impact on performance of various tree design choices and optimizations. We show that Masstree scales effectively and that its single shared tree can outperform separate per-core trees when the workload is skewed. We also quantify the costs of Masstree's flexibility. While variable-length key support comes for free, range query support does not: a near-best-case hash table (which lacks range query support) can provide 2.5× the throughput of Masstree.

The next section evaluates Masstree as a system. There, we describe the performance impact of checkpoint and recovery, and compare the whole Masstree system against other high performance storage systems: MongoDB, VoltDB, Redis, and memcached. Masstree performs very well, achieving 26–1000× the throughput of the other tree-based (range-

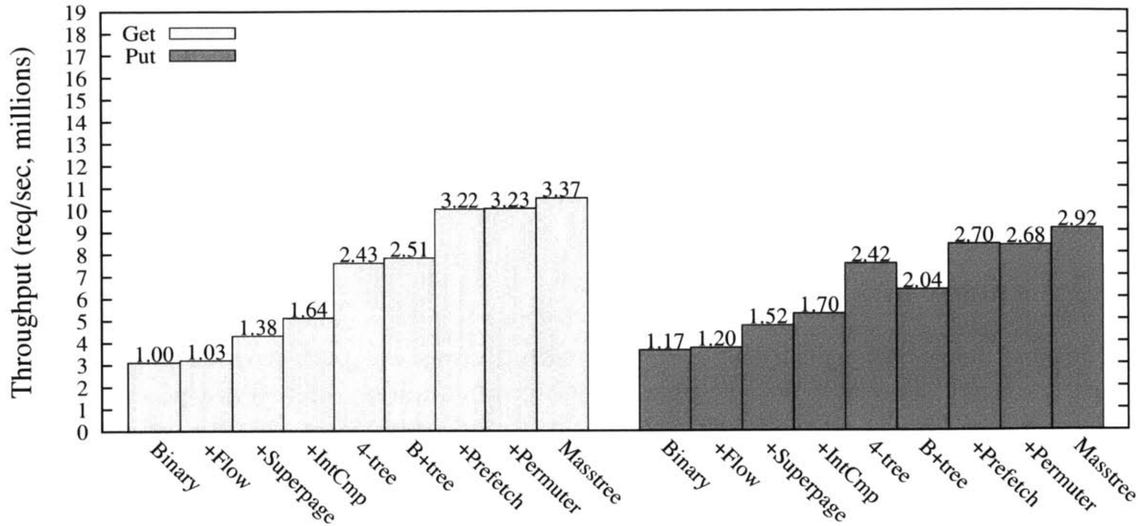


Figure 3-8: Contributions of design features to Masstree's performance (§3.3.2). Design features are cumulative. Measurements use 16 cores and each server thread generates its own load (no clients or network traffic). Bar numbers give throughput relative to the binary tree running the get workload.

query-supporting) stores. Redis and memcached are based on hash tables; this gives them $O(1)$ average-case lookup in exchange for not supporting range queries. memcached can exceed Masstree's throughput on uniform workloads; on other workloads, Masstree provides up to 3.7$\times$ the throughput of these systems.

3.3.1 Setup

The experiments use a 48-core server (eight 2.4 GHz six-core AMD Opteron 8431 chips) running Linux 3.1.5. Each core has private 64 KB instruction and data caches and a 512 KB private L2 cache. The six cores in each chip share a 6 MB L3 cache. Cache lines are 64 bytes. Each of the chips has 8 GB of DRAM attached to it. The tests use up to 16 cores on up to three chips, and use DRAM attached to only those three chips; the extra cores are disabled. The goal is to mimic the configuration of a machine more like those easily purchasable today. The machine has four SSDs, each with a measured sequential write speed of 90 to 160 MB/sec. Masstree uses all four SSDs to store logs and checkpoints. The server has a 10 Gb Ethernet card (NIC) connected to a switch. Also on that switch are 25 client machines that send requests over TCP. The server's NIC distributes interrupts over all cores. Results are averaged over three runs.

All experiments in this section use small keys and values. Most keys are no more than 10 bytes long; values are always 1–10 bytes long. Keys are distributed uniformly at random over some range (the range changes by experiment). The key space is not partitioned: a border node generally contains keys created by different clients, and sometimes one client will overwrite a key originally inserted by another. One common key distribution is "1-to-10-byte decimal," which comprises the decimal string representations of random numbers

between 0 and 2^{31}. This exercises Masstree's variable-length key support, and 80% of the keys are 9 or 10 bytes long, causing Masstree to create layer-1 trees.

We run separate experiments for gets and puts. Get experiments start with a full store (80–140 million keys) and run for 20 seconds. Put experiments start with an empty store and run for 140 million total puts. Most puts are inserts, but about 10% are updates since multiple clients occasionally put the same key. Puts generally run 30% slower than gets.

3.3.2 Factor analysis

We analyze Masstree's performance by breaking down the performance gap between a binary tree and Masstree. We evaluate several configurations on 140M-key 1-to-10-byte-decimal get and put workloads with 16 cores. Each server thread generates its own workload: these numbers do not include the overhead of network and logging. Figure 3-8 shows the results.

Binary We first evaluate a fast, concurrent, lock-free binary tree. Each 40-byte tree node here contains a full key, a value pointer, and two child pointers. The fast jemalloc memory allocator is used.

+Flow, +Superpage, +IntCmp Memory allocation often bottlenecks multicore performance. We switch to Flow, our implementation of the Streamflow [64] allocator ("+Flow"). Flow supports 2 MB x86 superpages, which, when introduced ("+Superpage"), improve throughput by 27–34% due to fewer TLB misses and lower kernel overhead for allocation. Integer key comparison (§3.1.2, "+IntCmp") further improves throughput by 11–19%.

4-tree A balanced binary tree has $\log_2 n$ depth, imposing an average of $\log_2 n - 1$ serial DRAM latencies per lookup. We aim to reduce and overlap those latencies and to pack more useful information into cache lines that must be fetched. "4-tree," a tree with fanout 4, uses both these techniques. Its wider fanout nearly halves average depth relative to the binary tree. Each 4-tree node comprises two cache lines, but usually only the first must be fetched from DRAM. This line contains all data important for traversal—the node's four child pointers and the first 8 bytes of each of its keys. (The binary tree also fetches only one cache line per node, but most of it is not useful for traversal.) All internal nodes are full. Reads are lockless and need never retry; inserts are lock-free but use compare-and-swap. "4-tree" improves throughput by 43–48% over "+IntCmp".

B-tree, +Prefetch, +Permuter 4-tree yields good performance, but would be difficult to balance. B-trees have even wider fanout and stay balanced, at the cost of somewhat less efficient memory usage (nodes average 75% full). "B-tree" is a concurrent B^+tree with fanout 15 that implements our concurrency control scheme from §3.1. Each node has space for up to the first 16 bytes of each key. Unfortunately this tree *reduces* put throughput by 16% over 4-tree, and does not improve get throughput much. Conventional B-tree inserts must rearrange a node's keys—4-tree never rearranges keys—and B-tree nodes spend 5 cache lines to achieve average fanout 11, a worse cache-line-to-fanout ratio than 4-tree's.

However, wide B-tree nodes are easily prefetched to overlap these DRAM latencies. When prefetching is added, B-tree improves throughput by 11–32% over 4-tree ("+Prefetch"). Leaf-node permutations (§3.1.6, "+Permuter") further improve put throughput by -1%.

Masstree Finally, Masstree itself improves throughput by 5–9% over "+Permuter" in these experiments. This surprised us. 1-to-10-byte decimal keys can share an 8-byte prefix, forcing Masstree to create layer-1 trie-nodes, but in these experiments such nodes are quite empty. A 140M-key put workload, for example, creates a tree with 33% of its keys in layer-1 trie-nodes, but the average number of keys per layer-1 trie-node is just 2.3. One might expect this to perform worse than a true B-tree, which has better node utilization. Masstree's design, thanks to features such as storing 8 bytes per key per interior node rather than 16, appears efficient enough to overcome this effect.

3.3.3 System relevance of tree design

Cache-crafty design matters not just in isolation, but also in the context of a full system. We turn on logging, generate load using network clients, and compare "+IntCmp," the fastest binary tree from the previous section, with Masstree. On 140M-key 1-to-10-byte-decimal workloads with 16 cores, Masstree provides 1.90× and 1.53× the throughput of the binary tree for gets and puts, respectively.[5] Thus, if logging and networking infrastructure are reasonably well implemented, tree design can improve system performance.

3.3.4 Flexibility

Masstree supports several features that not all key-value applications require, including range queries, variable-length keys, and concurrency. We now evaluate how much these features cost by evaluating tree variants that do not support them. We include network and logging.

Variable-length keys We compare Masstree with a concurrent B-tree supporting only fixed-size 8-byte keys (a version of "+Permuter"). When run on a 16-core get workload with 80M 8-byte decimal keys, Masstree supports 9.84 Mreq/sec and the fixed-size B-tree 9.93 Mreq/sec, just 0.8% more. The difference is so small likely because the trie-of-trees design effectively has fixed-size keys in most tree nodes.

Keys with common prefixes Masstree is intended to preserve good cache performance when keys share common prefixes. However, unlike some designs, such as partial-key B-trees, Masstree can become superficially unbalanced. Figure 3-9 provides support for Masstree's choice. The workloads use 16 cores and 80M decimal keys. The X axis gives each test's key length in bytes, but only the final 8 bytes vary uniformly. A 0-to-40-byte prefix is the same for every key. Despite the resulting imbalance, Masstree has 3.4× the throughput of "+Permuter" for relatively long keys. This is because "+Permuter" incurs a

[5] Absolute Masstree throughput is 8.03 Mreq/sec for gets (77% of the Figure 3-8 value) and 5.78 Mreq/sec for puts (63% of the Figure 3-8 value).

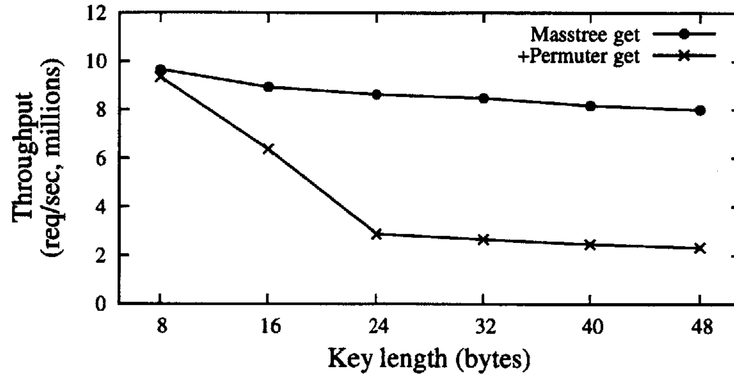


Figure 3-9: Performance effect of varying key length on Masstree and "+Permuter." For each key length, keys differ only in the last 8 bytes. 16-core get workload.

cache miss for the suffix of every key it compares. However, Masstree has 1.4× the throughput of "+Permuter" even for 16-byte keys, which "+Permuter" stores entirely inline. Here Masstree's performance comes from avoiding repeated comparisons: it examines the key's first 8 bytes once, rather than $O(\log_2 n)$ times.

Concurrency Masstree uses interlocked instructions, such as compare-and-swap, that would be unnecessary for a single-core store. We implemented a single-core version of Masstree by removing locking, node versions, and interlocked instructions. When evaluated on one core using a 140M-key, 1-to-10-byte-decimal put workload, single-core Masstree beats concurrent Masstree by just 13%.

Range queries Masstree uses a tree to support range queries. If they were not needed, a hash table might be preferable, since hash tables have $O(1)$ lookup cost while a tree has $O(\log n)$. To measure this factor, we implemented a concurrent hash table in the Masstree framework and measured a 16-core, 80M-key workload with 8-byte random alphabetical keys.[6] Our hash table has 2.5× higher total throughput than Masstree. Thus, of these features, only range queries appear inherently expensive.

3.3.5 Scalability

This section investigates how Masstree's performance scales with the number of cores. Figure 3-10 shows the results for 16-core get and put workloads using 140M 1-to-10-byte decimal keys. The Y axis shows per-core throughput; ideal scalability would appear as a horizontal line. At 16 cores, Masstree scales to 12.7× and 12.5× its one-core performance for gets and puts respectively.

The limiting factor for the get workload is high and increasing DRAM fetch cost. Each operation consumes about 1000 cycles of CPU time in computation independent of the

[6] Digit-only keys caused collisions and we wanted the test to favor the hash table. The hash table is open-coded and allocated using superpages, and has 30% occupancy. Each hash lookup inspects 1.1 entries on average.

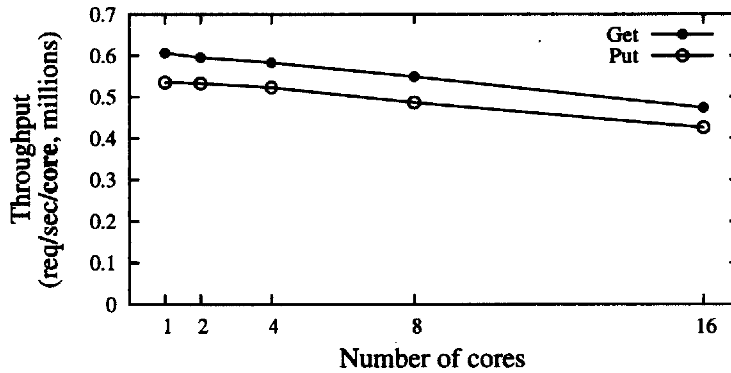


Figure 3-10: Masstree scalability.

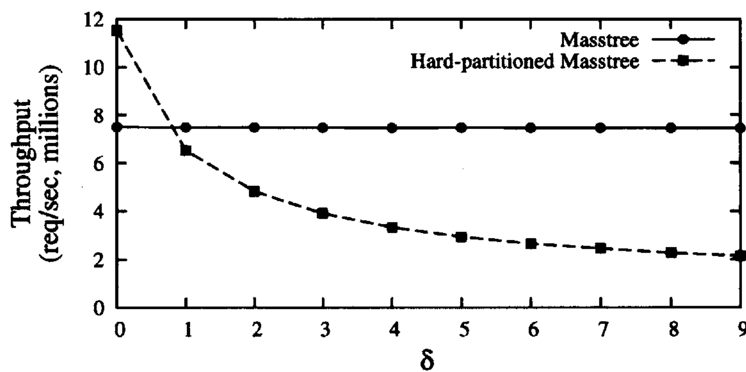


Figure 3-11: Throughput of Masstree and hard-partitioned Masstree with various skewness (16-core get workload).

number of cores, but average per-operation DRAM stall time varies from 2050 cycles with one core to 2800 cycles with 16 cores. This increase roughly matches the decrease in performance from one to 16 cores in Figure 3-10, and is consistent with the cores contending for some limited resource having to do with memory fetches, such as DRAM or interconnect bandwidth.

3.3.6 Partitioning and skew

Some key-value stores partition data among cores in order to avoid contention. We show here that, while partitioning works well for some workloads, sharing data among all cores works better for others. We compare Masstree with 16 separate instances of the single-core Masstree variant described above, each serving a partition of the overall data. The partitioning is static, and each instance holds the same number of keys. Each instance allocates memory from its local DRAM node. Clients send each query to the instance appropriate for the query's key. We refer this configuration as "hard-partitioned" Masstree.

Tests use 140M-key, 1-to-10-byte decimal get workloads with various partition skewness. Following Hua *et al.* [41], we model skewness with a single parameter δ. For skewness δ, 15 partitions receive the same number of requests, while the last one receives $\delta\times$ more

than the others. For example, at $\delta = 9$, one partition handles 40% of the requests and each other partition handles 4%.

Figure 3-11 shows that the throughput of hard-partitioned Masstree decreases with skewness. The core serving the hot partition is saturated for $\delta \geq 1$. This throttles the entire system, since other partitions' clients must wait for the slow partition in order to preserve skewness, leaving the other cores partially idle. At $\delta = 9$, 80% of total CPU time is idle. Masstree throughput is constant; at $\delta = 9$ it provides 3.5× the throughput of hard-partitioned. However, for a uniform workload ($\delta = 0$), hard-partitioned Masstree has 1.5× the throughput of Masstree, mostly because it avoids remote DRAM access (and interlocked instructions). Thus Masstree's shared data is an advantage with skewed workloads, but can be slower than hard-partitioning for uniform ones. This problem may diminish on single-chip machines, where all DRAM is local.

3.4 Discussion

Masstree has the following limitations:

- All the Masstree B+tree nodes have the same fanout. ART [51] shows how to dynamically resize tree nodes. It reports much higher performance numbers (more than 7 million lookups per core per second on 256 million keys) than Masstree. While the numbers are not directly comparable because ART doesn't support concurrent updates and the benchmark doesn't allocate a piece of memory to store the per-key value (which allows it to avoid an additional DRAM dereference for each lookup), it might still be profitable for Masstree to support multiple tree node sizes.

- Masstree supports only Ethernet and relies on the clients to send requests in large batches to achieve high throughput. MICA [52] shows that one can get high packet rates with Intel DPDK, which is a user-level library that performs direct I/O with the network card. Adding support for such a network stack may give Masstree more performance. Infiniband might also yield similar levels of performance since Infiniband comes with a user-level library that bypasses the kernel.

Chapter 4

Fast Fault-Tolerant Replication with Lazy VSR

This chapter presents a fast fault-tolerant replication protocol called Lazy VSR. Applying traditional replication protocols to in-memory key/value storage will limit storage performance. Such protocols typically require synchronous disk writes in order to be able to recover from crashes of a majority of replicas [48, 42]. A disk write could take 10 milliseconds, which would limit a straightforward implementation to 100 requests per second.

Lazy VSR addresses the slowness of traditional replication protocols by avoiding disk writes on the critical path for request latency. It writes requests to disk in batches in the background. The trade-off is that Lazy VSR may forget recent acknowledged operations if a majority of replicas crash simultaneously; however, many applications can tolerate such data loss. The challenge of Lazy VSR is post-crash recovery reconciliation, which is the focus of the protocol design. We build an in-memory key/value store called Lazen on Lazy VSR, and show that it can achieve nearly one million requests per second.

4.1 Background: View-Stamped Replication

Lazy VSR is derived from View-Stamped Replication (VSR) [55]. This section outlines the operation of VSR, and then discusses how VSR implementations obtain good disk performance, and how they cope with simultaneous failure of more than f servers.

4.1.1 VSR

VSR is a replicated state machine protocol to build fault-tolerant services. A VSR-based service has $2f+1$ replicas. If fewer than $f+1$ replicas are available, VSR will stop processing client requests. Once $f+1$ replicas with intact state have recovered and can communicate, VSR will continue operating. VSR guarantees that the service as a whole appears to execute requests sequentially: that any acknowledged client request will be reflected in the service's state.

During normal operation one replica is the primary and the rest are backups. The primary ensures that a majority of replicas sees each request to ensure durability, and that a

majority sees the same sequence of requests to ensure that those replicas remain identical.

Clients send requests to the primary. The primary assigns a sequence number to each request and then forwards it in a "prepare" message to all replicas (including itself). Each replica checks that it knows of no newer primary and then appends the request to its log. When a request is in a majority of the logs, the request is "committed": it will endure despite up to f replica failures. When the primary receives replies from a majority of replicas, it executes the request, sends a reply to the client, and sends a "commit" message to the replicas. When a replica receives the commit it executes the request.

The protocol never requires the primary to wait for more than a majority of replicas so that the group can tolerate up to f unavailable members. VSR copes with primary failures by progressing through a sequence of "views." A view consists of a view number and the identity of the primary for that view. A replica that sees that the primary is unreachable initiates a "view change."

The view change protocol ensures that each view is followed by exactly one successor view in order to avoid multiple primaries and "split brain". VSR does this with Paxos-like [48] agreement.

The view change protocol also ensures that replicas in the new view start with identical state, and that the state includes all committed requests in the previous view. Any committed request was known by a majority of replicas; that majority must overlap with the new primary's majority, so the new primary can discover all committed requests by consulting its majority.

4.1.2 Simultaneous Crashes and VSR Performance

VSR as described above can tolerate f failures. However, this is not enough. Power failure, human error, operating system bugs, etc. can cause all $2f+1$ servers to simultaneously crash. After enough servers reboot, it's important that the service be able to recover and continue processing requests.

One strategy VSR can use is for replicas to write all state changes to disk before replying; call this "VSR-Sync." Many forms of crash leave the disk contents intact, so that a VSR-Sync replica can recover up-to-date replication state from disk after reboot. A VSR-Sync system can tolerate simultaneous crashes of all servers as long as $f+1$ of them restart with disks intact. However, VSR-Sync is likely to be slow; a hard disk write takes roughly 10 milliseconds, so a straight-forward implementation might be able to process just 100 operations per second. Using an SSD may yield 2000 per second. A service can use group commit for higher throughput, though this also increases latency. ZooKeeper [42] uses a strategy similar to VSR-Sync with group commit.

At the opposite extreme, one could imagine a "VSR-Async" system that kept all state in volatile memory, never writing the disk. Performance would be $100\times$ to $1000\times$ better than VSR-Sync. However, VSR-Async cannot recover after more than f simultaneous failures.

Harp [55] occupies a middle ground between VSR-Sync and VSR-Async. A Harp replica writes changes only to memory before replying to the client, for high performance; it writes an on-disk log asynchronously. To guard against simultaneous power failure, every Harp replica has a UPS. Upon a power failure, the UPS gives the replica enough time to write its in-memory state to disk before shutting down. When the power is restored, each

replica verifies that it finished writing its state to disk; once $f+1$ such replicas recover, Harp can continue operation.

How does Harp behave if $f+1$ servers crash simultaneously, but do not manage to save their in-memory state to disk? Such a situation could arise from a hardware failure, software bug, or human error. In this situation, fewer than $f+1$ servers would see disk state reflecting a clean shutdown; Harp would then block until repaired by a human. Automatically continuing with stale state from some server's disk would often be illegal due to Harp's guarantee never to forget an acknowledged operation. Even without that guarantee, Harp's recovery algorithm would need to cope with replicas whose on-disk states reflect different histories of committed operations. This is the problem that Lazy VSR solves.

4.2 Design

Lazy VSR provides state machine replication. A Lazy VSR replica replies to the primary after appending the operation to a volatile in-memory log, and writes batches of operations asynchronously to an on-disk log. As a result Lazy VSR largely eliminates the disk as a performance bottleneck (like Harp). Lazy VSR can recover from crashes that cause memory content to be lost on more than f of $2f+1$ replicas (unlike Harp), though it may recover with recent operations missing.

Lazy VSR is intended to be used as part of a replicated service application. On each replica, the application keeps a copy of its own state in memory (e.g., a key/value database). Lazy VSR manages communication, both client/primary and primary/replicas. Lazy VSR gives each client operation to the application to execute (apply to the local copy of its state). The application must generate "undo" information as it executes each operation. The application must checkpoint its complete state to disk when Lazy VSR asks it to.

The hardest part of the Lazy VSR design is recovery after the crash and restart of more than f replicas. In such a failure, different replicas may forget different numbers of recent operations; some may forget little, but miss the first view change after recovery. This can lead to replicas with states that reflect not different prefixes of a single shared history, but divergent histories. Lazy VSR must be able to identify such situations, and roll back the effect of some operations in order to regain true replication.

For example, a majority may crash and restart, forgetting recent operations, while a minority is partitioned and retains the effect of those operations; when each replica in the minority rejoins, it will need to roll back the effects of the forgotten operations on its state. Figure 4-1 illustrates such a scenario. At Time 1, the group's three replicas have identical states and logs. At Time 2, C's network fails, so that C is alive but partitioned. At Time 3, A and B crash, losing the ends of their logs. A and B restart, discover that they have identical on-disk state, and form a new view starting with that state. At time 4, A and B process a put(x,0) operation from a client, and write the operation to their on-disk logs. At Time 5, C's partition heals. The three replicas must now decide which of their states to use for the next view. Even though C has the longest log, in this case the states of A and B must prevail, since they have operations that are more recent than those of C. C must roll back the last two operations to its state, and apply the put(x,0).

Lazy VSR is only appropriate for clients that can cope with a service that may forget

Time 1: The replicas are identical.

replica	state	log
A	x=3, y=2	put(x,1) put(y,2) put(x,3)
B	x=3, y=2	put(x,1) put(y,2) put(x,3)
C	x=3, y=2	put(x,1) put(y,2) put(x,3)

Time 2: C becomes partitioned from A and B.

replica	state	log
A	x=3, y=2	put(x,1) put(y,2) put(x,3)
B	x=3, y=2	put(x,1) put(y,2) put(x,3)
C	x=3, y=2	put(x,1) put(y,2) put(x,3)

Time 3: A and B lose their memory in a crash, reboot, and form a new view with their on-disk state, which is missing the last two operations.

replica	state	log
A	x=1	put(x,1)
B	x=1	put(x,1)
C	x=3, y=2	put(x,1) put(y,2) put(x,3)

Time 4: A and B process a client's put(x,0) operation.

replica	state	log
A	x=0	put(x,1) put(x,0)
B	x=0	put(x,1) put(x,0)
C	x=3, y=2	put(x,1) put(y,2) put(x,3)

Time 5: C's network connection heals.

replica	state	log
A	x=0	put(x,1) put(x,0)
B	x=0	put(x,1) put(x,0)
C	x=3, y=2	put(x,1) put(y,2) put(x,3)

Figure 4-1: A challenging Lazy VSR recovery scenario. In order to proceed from Time 5, C must roll back its last two operations, and apply the put(x,0).

about acknowledged operations. A client can ask a Lazy VSR service for information about what operations have become durable (see §4.2.1). A client may also set the "synchronous operation" flag in a request, which causes the primary to delay the reply until the operation is guaranteed to be durable.

4.2.1 Properties

Lazy VSR provides the following properties.

1. Lazy VSR remains available as long as $f+1$ out of $2f+1$ replicas stay alive and can communicate with each other.

2. If more than f replicas crash, Lazy VSR will recover and regain availability as soon as $f+1$ replicas with intact disks revive and can communicate.

3. If a client receives an acknowledgment for an operation, the operation will survive as long as $f+1$ replicas stay alive and in contact. That is, the operation may be lost if more than f replicas lose their memories. Such an operation is called "tentative."

4. If a majority of replicas receive an operation from the primary and finish logging it to disk before a view change, the operation becomes "durable." "Durable" means that the service will remember its effects on application state unless $f+1$ or more replicas lose their disk contents simultaneously (before Lazy VSR completes recovery). A tentative operation usually becomes durable within 10s of milliseconds after the client receives an acknowledgment.

5. A Lazy VSR primary can determine whether a given operation is durable. This is most interesting for past operations that occurred just before failures. This property allows a primary to tell clients which operations are durable.

6. A replica can free and re-use the log space for a durable operation once that operation is in the on-disk log of all replicas and in this replica's on-disk application checkpoint.

Property 2 envisions failures that cause replica memory to be lost, but preserve disk content. Power failures often have this character. We also expect some kernel panics and perhaps hardware failures to be similarly "clean" with respect to on-disk state. On the other hand, Lazy VSR's properties may not hold in the face of bugs that cause incorrect data to be written to disk. In general, Lazy VSR cannot cope with Byzantine failure or malice.

4.2.2 State

A Lazy VSR replica group progresses through a sequence of views. Each view has a number, with the primary's ID in the low bits for uniqueness; successive views have monotonically increasing numbers, though usually not sequential. v_i and v_{i+1} denote the numbers of successive views. Lazy VSR numbers operations sequentially; there are no gaps in the sequence. An operation is uniquely identified by a view number / operation number pair, called a viewstamp. Viewstamps are ordered, first by view number, then by operation number.

Figure 4-2b shows the in-memory state of each replica:

- *memLog* is a log of messages received from the primary; it includes just the messages important for recovery (PREPARE and ELECTACCEPT). A replica discards a log entry from *memLog* after it has been processed and appended to *diskLog*. As a result, the full log of a replica consists of the union of *diskLog* and *memLog*; we use *log* to refer to this union.

- *n* is used only by the primary, to hold the next operation number to be issued.

- *dp* holds the highest viewstamp known by the primary to be in the *diskLog*s of a majority of replicas. The primary computes *dp* from reports from the replicas; replicas learn *dp* from the primary.

ViewNum:	integer – primary's ID in the low bits
OpNum:	integer – operation number
Viewstamp:	$\langle v : \text{ViewNum}, n : \text{OpNum} \rangle$
LogEntry:	a PREPARE or ELECTACCEPT message
UndoInfo:	application specific information to roll back a client operation

(a) Types

memLog:	array of LogEntry
n:	OpNum – primary's next operation number
dp:	Viewstamp – durable point
undoInfo:	array of UndoInfo
knownDp:	array of Viewstamp; primary's knowledge of each replica's *dp*
dpMin:	Viewstamp – minimal viewstamp in *knownDp*
opMax:	Viewstamp – highest viewstamp in *memLog*
diskOpMax:	Viewstamp – highest viewstamp in *diskLog*

(b) In-memory state

v:	ViewNum – maximum the replica has ever seen
diskLog:	array of LogEntry
checkpoint:	checkpoint of application state and *undoInfo*

(c) On-disk state

Figure 4-2: Per-replica state.

- *undoInfo* holds application-specific information needed to roll back each client operation in the log.

- *knownDp* is used only by the primary. It is an array of viewstamp. The primary uses it to remember each replica's *dp* and compute *dpMin*.

- *dpMin* is the minimal viewstamp in *knownDp*. It is less than or equal to the lowest of all replicas' *dp*s. A replica can discard log entries up through *dpMin* as soon as they are reflected in the replica's *checkpoint*. The primary computes *dpMin* and sends it to replicas.

- *diskOpMax* holds the viewstamp of the last entry in *diskLog*.

- *opMax* holds the viewstamp of the last entry in *memLog*. *opMax* is always greater than or equal to *diskOpMax*.

Figure 4-2c summarizes the state maintained by each replica on disk; this is the state required for post-crash recovery:

- *v* is the highest view number the replica knows of, used to reject messages from stale primaries.

- *diskLog* holds log entries; Lazy VSR copies log entries from *memLog* to the end of *diskLog* in the background.

- *checkpoint* is created by the application. It holds application state and *undoInfo* as of a log entry specified in the checkpoint. Checkpoints allow Lazy VSR to recycle on-disk log space. The *diskLog* is a write-ahead log with respect to checkpoint.

4.2.3 Operation Within a View

Lazy VSR operates as follows during a view:

1. The primary receives a message $\langle op, ev \rangle$ from a client. *op* is the operation. *ev* is the expected current view number; the client library queries Lazy VSR for the current view number during initialization and caches it. If the primary's view number is not *ev*, it rejects the operation and sends the client a "wrong view" error. This prevents Lazy VSR from executing a client operation when some of the client's operations are missing, ensuring that simultaneous replica failure will forget a strict suffix of the client's operations. When the client receives "wrong view," it asks the new primary for the last viewstamp in *ev* and re-issues operations (if any) that were forgotten due to a crash at the end of *ev*. Thus the client must keep a cache of recent operations.

2. The primary sends $\langle$PREPARE *vs, op, dp, dpMin*$\rangle$ to each replica (including itself). *vs* is $\langle v, n \rangle$. *op* is the operation. The primary then increments *n*.

3. On receiving $\langle$PREPARE *vs, op, pDp, pDpMin*$\rangle$, if $vs \neq \langle v, opMax.n + 1 \rangle$ (the receiving replica missed some views or operations), the replica re-synchronizes (see §4.2.6). Otherwise, the replica accepts the message: it executes the operation (i.e. instructs the application to apply the operation to the application's in-memory state), sets its *dp* and *dpMin* to *pDp* and *pDpMin*, appends the PREPARE message to its in-memory log, and sends $\langle$PREPAREOK *diskOpMax*$\rangle$ to the primary. The replica can now discard undo information for operations with viewstamps $\leq pDp$.

4. The primary waits until it receives PREPAREOK from $f+1$ replicas (say *M*). Suppose replica $i \in M$ sends $\langle$PREPAREOK $diskOpMax_i\rangle$. Then the primary updates its *dp* to $min(\{\forall i \in M : diskOpMax_i\})$. It also sets *knownDp*[*i*] to the *dp* included in the PREPARE message sent in step 2. It then replies to the client with the result of the execution, the viewstamp of this operation, and *dp*; the client can discard cached operations with viewstamps $\leq dp$ since they are durable.

4.2.4 Checkpoint and crash recovery

In order to be able to re-use the disk storage holding old parts of its log, a Lazy VSR replica periodically asks the associated application replica to checkpoint its state to disk. The checkpoint must correspond to some point in the replica's log, so that during crash recovery a combination of the checkpoint and the log after that point are sufficient to reconstruct a correct state. A checkpoint may contain tentative operations; Lazy VSR will roll back their effect on application state if required.

A Lazy VSR replica manages checkpoint creation as follows. It stops listening for new messages from the primary, and asks the application to create an on-disk checkpoint of its

state; included in the checkpoint are v, *dp*, *undoInfo*, and the viewstamp of the most recent operation the application has executed (called *ap*).

After creating a checkpoint, the replica can discard log entries with viewstamps < *dpMin*, since they are reflected in the replica's checkpoint, and will not be needed by any other replica for recovery.

After a replica crashes and restarts, it reads its latest complete checkpoint file, restores the application state, *undoInfo* and *dp*, replays entries from *diskLog* whose viewstamps are greater than the *ap* in the checkpoint, sets *opMax* to *diskOpMax*, and then starts listening for Lazy VSR messages. Typically the first message it receives from the current primary will cause it to re-synchronize (§4.2.6).

4.2.5 View Change

View changes recover from the failure or unavailability of a primary. The view change protocol requires a majority in order to prevent multiple primaries and to ensure (by majority intersection) that the new view honors Lazy VSR's guarantees about tentative (Property 3) and durable operations (Property 4). The new primary must make sure that the replicas in the new view agree on the final sequence of operations in the previous view.

A replica starts a view change if it has not heard an acceptable PREPARE message from a primary for three seconds plus a short random interval. Such a "candidate" proceeds as follows:

1. The candidate chooses a view number v' higher than any it has yet seen. It makes the view number unique by placing its own identifier in the low bits.

2. The candidate sends $\langle$ELECTPREPARE $v', opMax\rangle$ to each replica. The candidate processes this ELECTPREPARE message locally before sending it to others. This ensures that the candidate's v is updated to v' so that the candidate won't reuse v' after a crash.

3. A replica ignores $\langle$ELECTPREPARE $v', cOpMax\rangle$ if $v' < v$ or $opMax > cOpMax$. If the replica accepts the message, it replies with an ELECTPREPAREOK. In either case, the replica sets its v to $\max(v', v)$.

4. A candidate is elected if it receives ELECTPREPAREOK from a majority of replicas. This ensures that the new leader preserves all acknowledged writes from the last view in the absence of simultaneous failures of more than f replicas. The candidate can only proceed if it is elected; otherwise it gives up, though it may try again later.

5. The candidate sends $\langle$ELECTACCEPT $v', opMax\rangle$ to each replica. The viewstamp of this message is $\langle v', opMax.n + 1\rangle$ (*opMax* is a viewstamp).

6. A replica rejects $\langle$ELECTACCEPT $v', cOpMax\rangle$ if $v' < v$. Otherwise, the replica update its v to v'. If the replica doesn't have *cOpMax* in its *log*, it initiates a re-synchronization (see §4.2.6). The replica must be up to date before replying to the ELECTACCEPT to ensure that a majority in the new view has the final operations from the previous view in their *log*s. Once the replica is up to date, it appends the ELECTACCEPT message to *memLog*, **appends *memLog* to *diskLog***, and replies with ELECTACCEPTOK. A

Log Entry	v	*opMax*
...		
⟨PREPARE ⟨21,307⟩,*op*307,⟨21,305⟩,...⟩	21	⟨21,307⟩
⟨PREPARE ⟨21,308⟩,*op*308,⟨21,305⟩,...⟩	21	⟨21,308⟩
⟨ELECTACCEPT 32,⟨21,308⟩⟩	32	⟨32,309⟩
⟨PREPARE ⟨32,310⟩,*op*310,⟨32,309⟩,...⟩	32	⟨32,310⟩

Figure 4-3: Example of the content of a replica's log at the time of a view change, along with the replica's v and *opMax* values after receipt of each message. The old view number is 21; the new one is 32. The ELECTACCEPT has viewstamp $\langle 32, 309 \rangle$.

replica will reject the ELECTACCEPT if v is updated to any view other than v' during the process. The replica's *opMax* is now the viewstamp of the ELECTACCEPT: $\langle v', cOpMax.n + 1 \rangle$.

7. The candidate waits until it has ELECTACCEPTOK responses from a majority of replicas. If a timeout period expires, the candidate gives up and may rerun the view change protocol again. Otherwise, the candidate becomes the primary for the new view. It sets n to $opMax.n + 1$, and proceeds to handle client operations as described in §4.2.3.

Figure 4-3 shows an example of the content of a replica's log after a view change has occurred followed by one new client operation. Note that the replica logs ELECTACCEPT (to disk), that the ELECTACCEPT consumes a viewstamp, and that acceptance of the ELECTACCEPT increases *opMax*.

A critical outcome of the view change protocol is permanent agreement on the sequence of operations with which the previous view ended. Different replicas may initially disagree: the old primary may have crashed after sending only a subset of PREPARE messages, some replicas may have missed recent messages, and some replicas may have lost the tails of their in-memory logs in crashes. Freezing an agreed-on tail of the old view's operations, and ensuring that the tail includes all operations on the disks of a majority of replicas, is important to guaranteeing Properties 4 and 5.

When a majority has written the same ELECTACCEPT message to their on-disk logs, the replica group has committed to the new view and primary. Lazy VSR distinguishes an elected candidate from a primary because the re-synchronization protocol (§4.2.6) occurs after candidate election but before committing to a new primary.

When a primary is committed, the agreement on the final operation sequence of the previous view is durable. Suppose the old view number was v_0. Once a majority of replicas has accepted a particular ELECTACCEPT, none of them will accept another ELECTPREPARE for a view change from v_0. The reason is that the *opMax* in such a message mentions v_0, while the majority in question all have *opMax* mentioning v', which is greater than v_0. A new candidate from among the majority will be able to get its ELECTPREPARE accepted and elected.

The ELECTPREPARE exchange ensures that a candidate can only be elected if it knows of at least as many acknowledged operations from the previous view as any replica it can

contact; this simplifies re-synchronization (§4.2.6) since a replica need only contact the elected candidate to learn of any missing operations from previous views.

Lazy VSR's view change is similar to that of View-Stamped Replication in that both must agree on how the previous view ended. They differ mainly in that there is more scope for replicas to initially disagree in Lazy VSR: some replicas may have lost acknowledged operations, and some replicas may have executed operations that the group later decides did not exist. The implications for view change are explained in the next sub-section.

4.2.6 Re-Synchronization

A replica must "re-synchronize" when it realizes that it is missing some operations, or that it has applied operations to its state that the current replica group has forgotten. A replica discovers it has missed view changes or operations when it receives $\langle \text{PREPARE}\ vs, \ldots \rangle$ and $vs.v > v$, or $vs.v = v$ but $vs.n > opMax.n + 1$. The replica must also re-synchronize as part of a view change when it receives $\langle \text{ELECTACCEPT}\ v', cOpMax \rangle$ with $v = v'$ but $opMax \neq cOpMax$.

A replica re-synchronizes by copying log entries from the current leader (either an elected candidate or a primary), since view change ensures that the leader has the most recent log entries. The checkpointing mechanism ensures that the leader will not have discarded any needed log entries. While not described here, a replica that has lost its on-disk checkpoint can copy a checkpoint from the leader.

A replica re-synchronizes as follows.

1. The replica sends a $\langle \text{CATCHUP}\ v,\ tailView \rangle$ message to the current leader. *tailView* holds the last viewstamps of views known to the replica. *tailView* only includes views subsequent to (inclusive) the replica's $dp.v$ because durable operations in the replica's log must be known to the leader.

2. A replica rejects $\langle \text{CATCHUP}\ rV,\ tailView \rangle$ if it is not the leader for view rV. Otherwise, it replies with $\langle \text{CATCHUPOK}\ v,\ vstMin,\ ops \rangle$. *vstMin* is the largest viewstamp that is common to operations represented by *tailView* and the leader's *log*. *ops* are operations in the leader's *log* whose viewstamps are larger than *vstMin*.

3. A replica discards $\langle \text{CATCHUPOK}\ pV,\ vstMin,\ ops \rangle$ if its v is no longer pV. Otherwise, it truncates its *memLog* and *diskLog* to *vstMin*, appends *ops* to *memLog*, and append *memLog* to *diskLog*. The replica then updates its *dp* and *dpMin* to those in the last PREPARE message in *ops*.

4.3 Proof

This section presents proofs of the properties described in §4.2.1. We have formalized the protocol in TLA [14], and used TLAPS [15] to prove type safety and to help check parts of our proofs (i.e. Lemma 4.3.4). Since TLA is not type-safe, proving type safety increases our confidence in the correctness of the protocol. Developing these proofs also caused us to revise the protocol to make it more amenable to proof. This section focuses on Property 4

(an operation is durable when the primary learns it is on the disks of a majority), because it is the most difficult to prove and the most important. We sketch the proof of other properties in section 4.3.2.

We first introduce two lemmas which simplifies the following discussion.

Lemma 4.3.1. *Each viewstamp identifies a unique operation.*

Proof. Any given viewstamp *vs* could only have been generated by the replica identified in *vs.v*'s low-order bits. That replica never re-uses a viewstamp while it stays alive. The replica records each new view number it learns of on disk, so that it can also avoid re-using a view-number (and thus avoid re-issuing a viewstamp) across re-starts.

□

Lemma 4.3.2. *Each viewstamp has a unique preceding viewstamp.*

Proof. The preceding viewstamp is $\langle vs.v, vs.n-1\rangle$ for $\langle$PREPARE $vs, \ldots\rangle$, and *cOpMax* for $\langle$ELECTACCEPT $v', cOpMax\rangle$. The uniqueness of *cOpMax* (the last viewstamp in the previous view) is ensured by the durability of the ELECTACCEPT, which contains *cOpMax*. □

With Lemma 4.3.1 and 4.3.2, a *dp* identifies a unique sequence of operations whose viewstamps are less than or equal to *dp*, which includes *dp*, the operation precedes *dp*, the operation precedes the operation preceding *dp*, and etc. These operations are "covered" by *dp*, and *dp* "succeeds" any of these operations. A viewstamp that is less than *dp* may not be covered by *dp*.

4.3.1 Durable Point

The most critical part of the proof is to prove the following Theorem.

Theorem 4.3.3. *Operations covered by any dp appearing in the protocol are durable, i.e. no primary would ask a replica to discard such operations.*

To prove the theorem, it suffices to show that each dp' computed by the primary is a durable point (because other replicas receive *dp* from the primary). Without loss of generality, let's assume that the primary computes dp' from PREPAREOKs from a set of $f+1$ replicas (say *M*) accepting the same message *m* $\langle$PREPARE $vs, \ldots\rangle$. Without loss of generality, suppose these replicas $r_0, r_1, ..., r_{f+1}$ accept *m* at time $t_0, t_1, ..., t_{f+1}$ $(t_0 < t_1 < ... < t_{f+1})$ respectively. At t_i, r_i sends $\langle$PREPAREOK $diskOpMax_i\rangle$ to the primary. Our proof induces over time, starts from t_0. The proof builds on three key definitions as follows.

Definition 1. *Succeed(a,b) means a succeeds b.*

Definition 2. *For a view v starts with message* $\langle$ELECTPREPARE $v', opMax\rangle$, *let* $v.v = v'$ *and* $v.prev{=}opMax$. *v is bad if the primary of* v' *may ask a replica to discard operations covered by* dp', *or formally:*

$$
\begin{aligned}
&BadView(v) \triangleq \\
&\land\ v.v > dp'.v \\
&\land \lor\ v.prev < dp' \\
&\quad \lor (v.prev > dp' \land \neg Succeed(v.prev, dp'))
\end{aligned}
$$

Definition 3 (*HasNotVoted(t)*). *No replica in M has voted for (i.e. sent* ELECTPREPAREOK *to) any bad view before time t.*

The definitions follows the fact that a replica will ignore messages from stale primaries; that a primary will ask a replica to discard any operations missing in the primary's *log*.

It turns out that, if *HasNotVoted(t)* is true for any t, dp' is durable. Below we first prove this as Theorem 4.3.7, and then prove that *HasNotVoted(t)* is true at any time. We first introduce several lemmas that is important to the subsequent proof.

Lemma 4.3.4. *The v of each replica is monotonically increasing, as proved by TLAPS (see Appendix A).*

Lemma 4.3.5. *HasNotVoted(t) implies that no primary has asked a replica to discard dp' before time t.*

Proof. Suppose an arbitrary replica i has appended dp' to *log* at time tt and $tt < t$. Then $v[i] \geq dp'.v$ since tt (by Lemma 4.3.4). After tt, only a bad view will ask i to discard dp'. Since no bad view was formed before t, i was not asked to discard dp'. □

Lemma 4.3.6. *$dp'.v$ equals $vs.v$. dp' is the durable point the primary computed from the $f+1$* PREPAREOK*s it has collected for the* PREPARE *message identified by vs.*

Proof. Since dp' is computed from $diskOpMax_i$s, it suffices to show that, for any $i \in M$, $diskOpMax_i.v$ equals $vs.v$.

Since r_i accepts m, $v[r_i]$ must still be $vs.v$ at t_i, which suggests r_i has not received any message from views $> vs.v$. So $diskOpMax_i.v \leq vs.v$.

Since r_i accepts m, right before r_i sends PREPAREOK, r_i must have the ELECTACCEPT message of $vs.v$ on its on-disk log. So $diskOpMax_i.v \geq vs.v$. □

Theorem 4.3.7. *If HasNotVoted(t) is true for any t, dp' is durable.*

Proof. By Lemma 4.3.5, dp' is durable. □

We now prove that *HasNotVoted(t)* is true for any time via induction.

Lemma 4.3.8. *HasNotVoted(t_0) is true.*

Proof. Suppose $r_i \in M$ and r_i has voted for a bad view before t_0. By Lemma 4.3.4, $v[r_i] > dp'.v$ at t_0. So r_i won't accept m at t_i. Contradictory. □

Lemma 4.3.9. *Suppose HasNotVoted(t) is true. Then HasNotVoted($t+1$) is also true.*

Proof. Suppose $r_i \in M$, and the action at $t+1$ is for r_i to process the ELECTPREPARE message for a bad view bv. We prove case by case according to the definition of *BadView(bv)* that r_i won't vote for bv.

1. Case: $bv.prev < dp'$.

   (a) Case $t < t_i$. Assume r_i votes for bv at t. Then $v[r_i]$ will be set to $bv.v$, which is $> dp'.v$ (by the definition of *BadView*). By Lemma 4.3.4 and 4.3.6, r_i won't accept m at t_i. Contradictory.

(b) Case $t > t_i$. By the way dp' is computed, r_i must have dp' on its disk at t_i. So $opMax[r_i] \geq diskOpMax[r_i] \geq dp' > bv.prev$ at t because dp' was preserved before time t by Lemma 4.3.5. Then r_i won't vote for bv at t according to step 6 in §4.2.5.

2. Case: $bv.prev > dp'$ and $\neg Succeed(bv.prev, dp')$. Since $\neg Succeed(bv.prev, dp')$, at some time before t, the candidate proposing bv must have two consecutive log entries *prev* and *next* such that $prev < dp'$ and $next > dp'$.

   (a) Case: $next.v = dp'.v$. Then $prev \geq dp'$ because a replica can't accept *next* without accepting dp'. Contradicts with $prev < dp'$.

   (b) Case: $next.v > dp'.v$. Since $prev < dp'$, $prev.v \leq dp'.v$. This means *next* is an ELECTACCEPT message for view $v_2 = \langle next.v, prev \rangle$, which means v_2 was elected before t. Since $next.v > dp'.v$ and $prev < dp'$, v_2 is bad, suggesting that some replica in M has voted for v_2 before t, i.e. *HasNotVoted(t)* is false. Contradictory.

□

4.3.2 Proof Sketch of Other Properties

This sub-section outlines proof sketches of all other properties.

Property 1 (*Availability*) As long as the candidate or the primary can communicate with a majority, Lazy VSR can continue the protocol for view change, re-synchronization and normal operation.

Property 2 (*Recoverability*) As long as the disk is intact, a server can recover from crash. As long as the candidate or the primary can communicate with a majority of live servers, Lazy VSR can continue the protocol for view change, re-synchronization and normal operation.

Property 3 (*Tentative Commitment*) According to the view change protocol, the new primary must have the longest log among a majority. Thus, one of the replicas which has accepted the last acknowledged operation of the previous view will become the primary and preserve the operation.

Property 5 (*Durability Discovery*) The primary wait until $vs \leq dp$ before it can determines the durability of an operation with viewstamp vs. Once the condition holds, the operation is durable if vs is covered by dp and otherwise discarded.

Property 6 (*Log Space Reuse*) No primary would ask a replica to discard a durable operation. So once all replicas has the operation on the disks, no replica would need that operation anymore. Since the operation is also reflected in the on-disk application checkpoint, the application would not need either and Lazy VSR can safely re-use the log space for the operation.

4.4 Lazen

Lazen is a new key/value store, intended to demonstrate Lazy VSR and to explore atomic shard movement techniques. Data must fit in memory, although Lazen logs and checkpoints to disk for crash recovery.

Lazen processes client reads as well as writes through Lazy VSR replication, which ensures that reads see the latest updates. Lazen is not sequentially consistent because replica groups may forget recent operations. This is not a problem for many applications (§4.5).

4.4.1 Sharding

Lazen shards its key/value database, by key ranges, over multiple replica groups in order to increase total throughput. Each group typically serves many shards. Lazen allows transfer of shards between groups in order to adjust load-balance and to move load onto new replica groups.

Shard transfer must be atomic. The steps (the old group stops serving requests for the shard, the old group sends the data, the old group deletes the data, the new group receives the data, the new group starts serving) must appear to happen at a single instant, between two client requests. A failure cannot be allowed to result in neither group, or both groups, believing they are responsible for the shard.

A Lazen deployment includes a replicated configuration manager. The manager's replicated state includes the current mapping of shard (key range) to replica group, and a "transfer log" of initiations and completions of shard transfers. Manager replicas keep their state on disk, so that committed operations are durable. The manager initiates a shard transfer when the human administrator tells it to.

The shard transfer protocol is simple because it can treat the manager and the replica groups as if they cannot fail. Each party uses its replicated state-machine log to durably record action requests and action completions. After a view change, each party inspects its log to see if it has promised an action for which there is no logged completion; if so, it performs the action (perhaps repeating it). This structure allows the manager to ask a replica group to perform its part of a shard transfer, and to be sure, once the replica group acknowledges that it has placed the request in its log, that it will eventually carry out the action even in the face of failures.

The shard transfer protocol "commits" a message by replicating it through Lazy VSR using synchronous disk writes. Shard transfer messages are stored in the same Lazy VSR log as client operations.

Shard transfer proceeds as follows:

1. The administrator asks the configuration manager to transfer shard s from replica group g_1 to group g_2.

2. The manager commits $\langle$BEGIN $s, g_1, g_2\rangle$.

3. The manager periodically looks in its log for BEGIN records that are not matched by END records. For each unmatched $\langle$BEGIN $s, g_1, g_2\rangle$, the manager sends a $\langle$BEGINSEND $s, g_2\rangle$ message to the primary of g_1, and a $\langle$BEGINRECV $s, g_1\rangle$ to the primary of g_2. If the manager receives replies for both, it commits $\langle$END $s, g_1, g_2\rangle$, and updates the shard mapping so that shard s maps to group g_2.

4. The primary of g_1 eventually receives the $\langle$BEGINSEND $s, g_2\rangle$, commits the message, and replies to the manager.

5. If a primary has a $\langle$BEGINSEND $s, g_2\rangle$ in its log, it rejects client operations for shard s.

6. If a primary has a $\langle$BEGINSEND $s, g_2\rangle$, but no corresponding ENDSEND, it sends all the key/value pairs in shard s to the primary of g_2. If the primary of g_2 acknowledges, the primary of g_1 commits $\langle$ENDSEND $s, g_2\rangle$, and then deletes its copy of shard s.

7. The primary of g_2 eventually receives the $\langle$BEGINRECV $s, g_1\rangle$ from the manager, commits it, and replies.

8. When the primary of g_2 receives a transfer of shard s from group g_1, and the primary finds an unmatched $\langle$BEGINRECV $s, g_1\rangle$ in its log, the primary commits $\langle$ENDRECV $s, g_1, db\rangle$, where *db* is the content of the shard. The primary then acknowledges the transfer, and starts serving client requests for shard s.

Group g_2 cannot start serving a shard before g_1 stops: g_2 can't get a copy until g_1 sends it; g_1 won't send it until it has entered the BEGINSEND durably in its log; and g_1 won't serve client requests if it sees the BEGINSEND.

The shard content cannot be lost as long as a majority of each replica group retains its disk content. g_1 discards its copy of the shard after it receives an acknowledgment from g_2's primary; and g_2's primary only sends the acknowledgment after the group has committed the ENDRECV to its log, including the shard content.

4.4.2 Clients

Each client knows the identities of all the replicas in a given replica group. The client remembers which replica it believes to be the primary. If that replica does not respond, or if it responds by directing the client to a different primary, the client re-sends its request to a different replica.

The fact that clients may re-send requests, particularly to different replicas, can give rise to duplicate requests. Lazen detects duplicated requests and avoids re-execution with at-most-once RPC. Each client request includes a unique client identifier and a client sequence number. Both of these are included in the Lazy VSR log. This allows all replicas to maintain

a table, indexed by client identifier, of the most recent request sequence number seen from each client, and of the most recent reply value. When a replica executes a request it hears from the primary, the replica first checks the table to see if the request's sequence number is old; if so, the replica ignores the operation. If the primary sees a duplicate of a client's most recent request, the primary replies with the remembered reply value; if the primary sees a repeated older request, it replies with an error. A replica skip executing old requests. If the old request is a duplication of the most recent request, the primary replies with the remembered reply value.

If a client sends more than one outstanding request, it cannot count on the primary resending a saved reply for all duplicate requests. The client can, however, rely on the primary to return an error indicating that the replica group has already processed the request.

The client table is also recorded in the checkpoint file to filter duplicated requests across restarts. Duplicate detection does not currently work across shard transfers. We leave it as a future work. We plan eventually to transfer the duplicate table along with the shard data.

4.4.3 Lazen Implementation

Lazen is written in C++. It stores all key/value pairs in a single in-memory hash table. We wrote a Protocol Buffers [9] plugin to reduce string copies when serializing and deserializing messages. For example, when parsing a log entry, Masstree reuses the string buffer of the entry rather than make copies.

The primary batches client requests into PREPAREs, and pipelines the PREPAREs. A replica writes to its on-disk log using double-buffering and group commit. Thus, under load, the replica is always writing the on-disk log file. A replica replies to the primary without waiting for the primary's message to reach the on-disk log. However, if one buffer fills before the other has finished being written to disk, the replica stalls request processing. Each buffer holds 100 MB.

4.5 Use Cases

Lazy VSR presents unusual semantics, since it may acknowledge an operation but then forget about it. This section sketches some example applications which fit well with Lazy VSR's semantics.

4.5.1 File System Backend

Lazy VSR's semantics make sense for file systems. File systems are aware of the lazy durability of the storage device; most POSIX applications (e.g. database, compiler, or email server) are aware of lazy durability of the file system [57]. Both file systems and POSIX applications use disk flushes to enforce consistency [57] and they can recover from the loss of acknowledged but un-flushed writes. For these applications, using Lazy VSR as the file system backend is feasible because its support for flushes (as synchronous writes) is sufficient to achieve application consistency.

Compared with existing storage solutions, the advantage of a Lazy VSR based file system backend is that such a back end achieves both high availability and high performance on ordinary hardware.

To demonstrate such usage, we have built a Linux network block device (NBD) [8] server based on Lazen. NBD is a kernel feature that allows Linux to use a TCP server as a block device. The server exports a virtualized block device to the kernel through a TCP connection (called nbd connection). The kernel sends block reads, block writes and flush commands to the server for processing. The kernel maintains multiple outstanding requests and pipelines requests for performance.

Our NBD server (called "lazy-disk") is a process running on the same kernel that mounts lazy-disk. For simplicity and performance, lazy-disk doesn't have any persistent state, which means that it will lose all state once restarted. This is OK because lazy-disk fails only if the kernel reboots, in which case the kernel would perform a recovery with a fresh lazy-disk just as a kernel using disks recovers from a power failure.

lazy-disk maps each block to a unique key/value pair, where the key is the block number and the value is a blob of the block size. For each read/write request, lazy-disk converts it to Lazen get/put requests. For a flush request, lazy-disk converts it to a synchronous writes of a special key. Since a read/write request may access multiple blocks and Lazen doesn't support range queries, lazy-disk splits it into multiple get/put requests. This shouldn't have much overhead because lazy-disk sends requests to Lazen in batch. For performance, NBD requests are processed in pipeline. One lazy-disk thread receives read/write requests from Linux kernel and sends put/get requests to Lazen; another lazy-disk thread receives replies from Lazen and acknowledges Linux kernel.

The challenge of applying Lazen to lazy-disk is handling Lazen replica failure. lazy-disk must hide Lazen failure from the Linux kernel. It cannot report I/O or device failure for outstanding requests to the kernel because file systems don't handle device failure well. For example, ext4 would remount the file system as read only when an IO request fails, which renders the file system almost unusable. As a result, Lazy VSR client must recover from server failures transparently to retain the availability of the file system.

One way for lazy-disk to hide Lazen failure is to implement exact-once RPC. After a Lazen connection failure, lazy-disk re-sends all tentatively committed and outstanding read/write requests to Lazen using the same RPC sequence number. Since the Lazen replicas would remember replies for all unacknowledged requests, the replayed requests would see the same response as in the failure-free case. However, this approach is impractical because Lazen must persist RPC replies. The following example demonstrates that, if the server keeps the replies in volatile-memory only, replayed operations may observe results inconsistent with the results before the failure. Suppose the initial value of key x is 0 and lazy-disk has two outstanding requests: get(x) and put(x,1). Without any failure, the get would observe value 0. If the server have executed both requests but crashed right before sending any replies, the get would observe value 1 rather than value 0.

Fortunately, file systems don't require Lazen to recover faithfully from Lazen failure. The block device is allowed to execute all outstanding requests in arbitrary order as long as a flush is not completed until all acknowledged writes are persist [10]. For read/write requests, this means that it is OK (though maybe not necessary) for lazy-disk to send replies to Linux kernel as long as the replies received by the Linux kernel and the state of Lazen are

consistent with some total order of all requests. lazy-disk recovers from Lazy VSR replica failure in the following way to satisfy this requirement. It caches all tentatively committed and outstanding read/write requests. Whenever the Lazy VSR connection (to the primary) fails, it just re-sends all tentatively committed and outstanding requests to Lazy VSR as new requests, i.e. tagging each RPC request with new sequence numbers. To avoid sending multiple replies for the same request to the Linux kernel, lazy-disk keeps track of which read/write requests are acknowledged and discards duplicated (and potentially different) replies.

With the above recovery scheme, we can always construct a total order that is consistent with NBD replies and the state of Lazen. In the total order, writes and flushes are ordered in the order they were issued by the kernel. Such order satisfies the requirement of the implementation of the flush command because lazy-disk re-sends writes in issuing order during recovery. Thus, when a flush is completed, all previous acknowledged writes must have been durable. It is clear that such order is consistent with the state of Lazen, where each block reflects the value of the most recent write in the order. The order of reads are trivial. A read is ordered anywhere between the observed write and the next write to the same block. Reads of different blocks can be ordered arbitrarily.

Our initial evaluation of lazy-disk shows that lazy-disk with Lazy VSR provides three times the throughput of VSR-Sync on SSD and Infiniband. We will evaluate it in more details.

4.5.2 Blizzard

Blizzard [57] is a good candidate for Lazy VSR. It is a block storage system that offers lazy durability for performance. Most applications using Blizzard can cope with data loss. Blizzard uses FDS [58] as the underlying storage, which relies on clients (here the Blizzard server) to restore replicas to consistency. One problem Blizzard faces is that, since the Blizzard service is not replicated, a Blizzard failure might leave the system unavailable or inconsistent. With Lazy VSR, we can combine the Blizzard service and the storage service so that both services are replicated. Such scheme preserves Blizzard's client semantics and high performance while making Blizzard fault-tolerant and consistent.

4.5.3 MongoDB

MongoDB could profitably use Lazy VSR because MongoDB requires its clients to be able to cope with the loss of recent updates. Use of Lazy VSR would improve MongoDB's recovery correctness and reduce data loss [2, 16] while preserving high performance.

MongoDB has trouble recovering from memory-losing crashes during certain lengthy operations: it needs to copy the entire index in order to undo *dropIndex*, and wait for the administrator to recover manually to undo *dropDatabase* [7]. With Lazy VSR, MongoDB could avoid undoing these operations by applying them *iff* they are durable. VSR-Sync takes the same approach, but sacrifices performance for normal operations.

CPU	1× Intel E5-1410	
DRAM Size	64 GB	
File System	ext4 w/ barrier	
Kernel	Linux 3.11.0-12	
Disk	Intel 520 SSD	
Disk Thru.	260 MB/sec	
Disk Latency	0.5 ms	
Network	Gigabit Ethernet	40Gbps Infiniband
Network RTT	0.5 ms	0.013 ms

Figure 4-4: Machine configuration. "Disk Thru." is measured file system throughput with sequential writes. "Disk Latency" is the time to overwrite and *fdatasync()* 4096 bytes. Network RTT is user space to user space latency measured with packets with 20-byte payloads.

4.6 Evaluation

This section evaluates Lazy VSR in the context of Lazen. Most of the comparisons are against "VSR-Sync," which differs only in that it uses synchronous log writes. The evaluation shows that Lazy VSR provides higher throughput and lower latency than VSR-Sync. For workloads with some synchronous writes, Lazy VSR improves the performance significantly. Lazy VSR can cope with simultaneous failures of all replicas, and the data loss window is small. We also measure scalability and performance during shard transfers, and show that Lazen is significantly faster than ZooKeeper and MongoDB. Finally, we show that Lazy VSR improves the performance of the file system backend significantly.

The experiments use a test-bed with 12 machines (Figure 4-4). The machines can use either TCP over Gigabit Ethernet or Reliable Connection on Infiniband. We use libibverbs 1.1.6 for Infiniband DMA direct to user space. Up to nine machines are servers; up to three are load generators. Each replica group consists of three machines. Disk write caches are enabled, although this doesn't affect the results. Intel specifies the Intel 520 write latency as 80 us; this seems to include just the time to write the SSD's write cache. Overwriting a 4096-byte file block through the file system and forcing it to the flash medium takes 0.5 ms. Each log file's disk space is pre-allocated.

The workload in all experiments consists of "put" requests that update existing key/value pairs. Keys are no more than 5 bytes, randomly chosen from 100,000 keys. Lazen stores all key/value pairs in a hash table with 10 million buckets to avoid hash conflicts. Depending on the experiment, the values are 1, 8 or 1024 bytes. Performance of "get" requests is similar to that of "puts", since Lazen sends them through its replicated state machine.

One or more single-threaded client processes send requests, each sending one at a time. The experiments vary the number of client processes in order to present performance with varying load levels, and because VSR-Sync's group commit efficiency depends on the number of outstanding requests. Enough client machines are used so that client hardware does not limit performance.

For fair comparison, the VSR-Sync implementation uses the techniques described in §4.4.3; unlike Lazy VSR, a VSR-Sync replica waits for an operation to be on disk before responding, to ensure immediate durability.

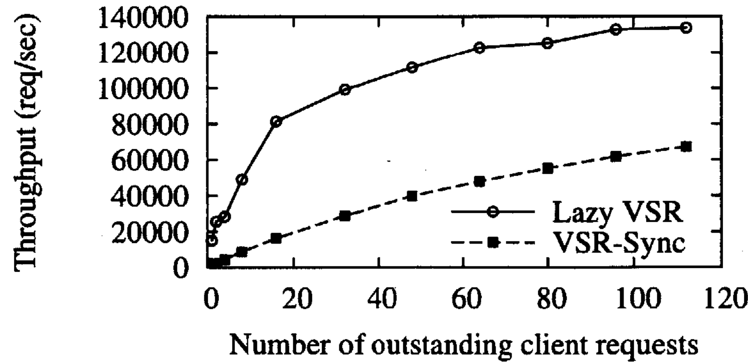


Figure 4-5: Lazy VSR has higher throughput than VSR-Sync with 1024-byte put requests.

We disabled checkpointing, since its cost has more to do with the size of the database than it does with the replication protocol.

4.6.1 Throughput and Latency

This section compares the performance of Lazy VSR with that of VSR-Sync. The SSD/Infiniband test-bed is intended to represent a modern high-performance configuration. The SSDs may also favor VSR-Sync, since SSDs' fast writes reduce the value of Lazy VSR's deferred and batched log writes.

Figures 4-5 and 4-6 show the throughput and latency for put requests with 1024-byte values. Lazy VSR's throughput with small numbers of outstanding requests is determined by network delay and processing time at the primary. For example, with one outstanding request, these times total 58 us, which limits throughput to 17,000 requests/second; this is close to the measured throughput. With 100 or more outstanding requests, the throughput is limited by Lazy VSR's single primary thread executing all requests in serial order. Each request takes 7 us to process, limiting throughput to 140,000, which is close to the measured maximum. The SSDs do not limit Lazy VSR's throughput because it writes large asynchronous batches. With 40 clients, each Lazy VSR replica writes its SSD every 47 requests on average.

VSR-Sync's throughput is limited by its synchronous log writes. An SSD can complete about 2000 small synchronous writes per second. Thus, with 40 outstanding requests, each SSD write holds on average 20 requests, yielding a throughput of $2000 \times 20 = 40,000$ requests/second. VSR-Sync's throughput rises with the number of outstanding requests because it logs more requests per SSD write. The increase is sub-linear due to increased latency; we expect VSR-Sync to saturate the executing thread at about 8,000 outstanding requests.

Figure 4-6 shows that Lazy VSR's latency is one to two SSD write latencies less than that of VSR-Sync; VSR-Sync has an SSD write in the critical path, and Lazy VSR does not. The Lazy VSR request latency is largely determined by the time the primary takes to drain the queue of outstanding requests waiting for service; it takes about 7 us of the primary's CPU time to process each request.

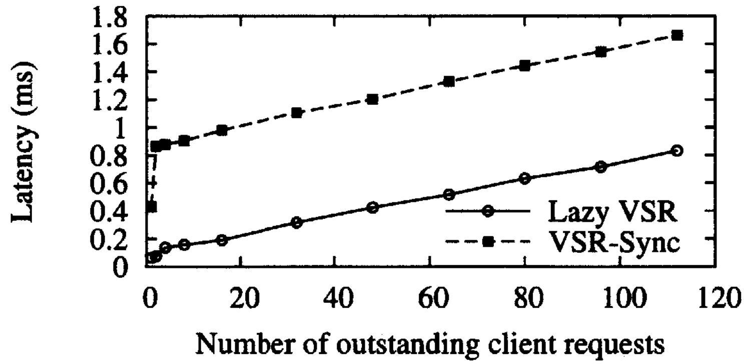


Figure 4-6: Lazy VSR has lower latency than VSR-Sync with 1024-byte put requests.

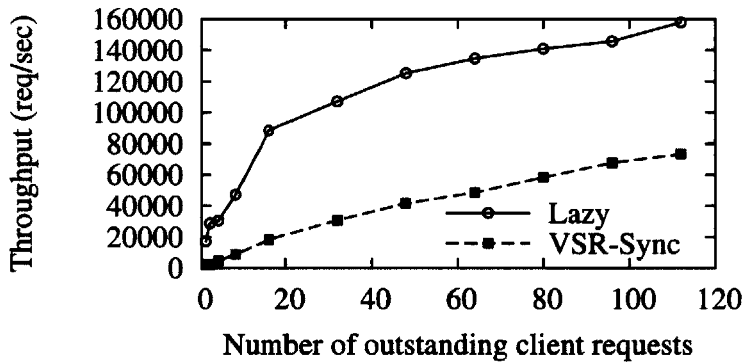


Figure 4-7: Lazy VSR has higher throughput than VSR-Sync with 8-byte put requests.

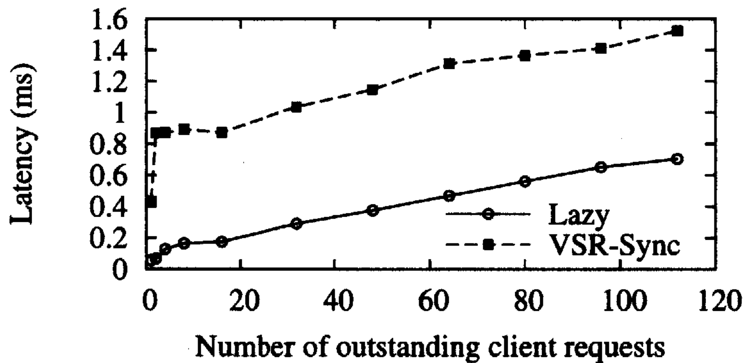


Figure 4-8: Lazy VSR has lower latency than VSR-Sync with 8-byte put requests.

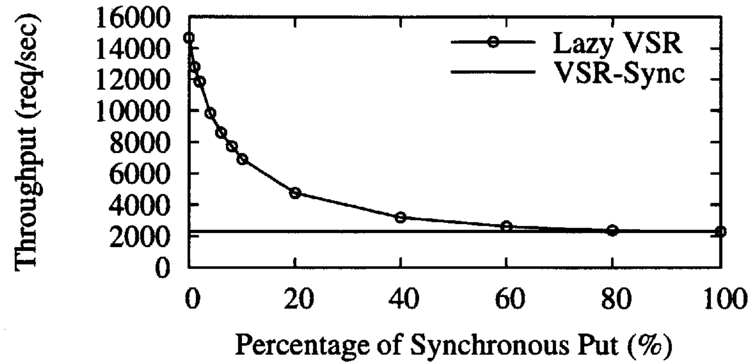


Figure 4-9: Lazy VSR improves throughput over VSR-Sync significantly when the ratio of synchronous puts is relatively small.

Figure 4-7 and 4-8 shows the results with 8-byte values. The results are similar to those of 1024-byte values, except that throughputs are 10% to 15% higher. With hard drives, Lazy VSR has about ten times the throughput of VSR-Sync, because VSR-Sync is much more affected by hard drives' high write latency.

The main conclusion is that by avoiding SSD writes in the critical path, Lazy VSR substantially improves throughput and latency compared to a traditional replication scheme with group commit.

4.6.2 Performance with Synchronous Put

Lazy VSR clients may perform synchronous puts to ensure immediate durability (see §4.5.1). Figure 4-9 shows the throughput of Lazen with 1024-byte puts as the fraction of puts that are synchronous changes. The experiment uses SSDs, Infiniband, a single client process, and one replica group. The throughput at 100% is the throughput of VSR-Sync on the same workload. Lazy VSR improves throughput significantly over VSR-Sync when synchronous puts are relatively rare. For example, at 10%, Lazy VSR offers 3× the throughput of VSR-Sync. As the percentage increases, the throughput of Lazy VSR converges to the throughput of VSR-Sync. Lazy VSR offers the most improvement to applications that require immediate durability infrequently.

4.6.3 Failure Recovery

The number of acknowledged but tentative operations that a Lazy VSR system will lose if a majority of nodes fail depends on how far the replicas are lagging in their asynchronous log writes to disk. To get a feel for how many operations might be involved, we ran an experiment in which 96 client processes continuously issue puts with 8-byte values to a Lazen replica group. The clients average a total of 150,000 requests/second. We terminated all the replica processes with "kill" at roughly the same time (leaving too little time for a view change), so that they all lost their memory. Over 20 runs, a maximum of 56 acknowledged puts were lost.

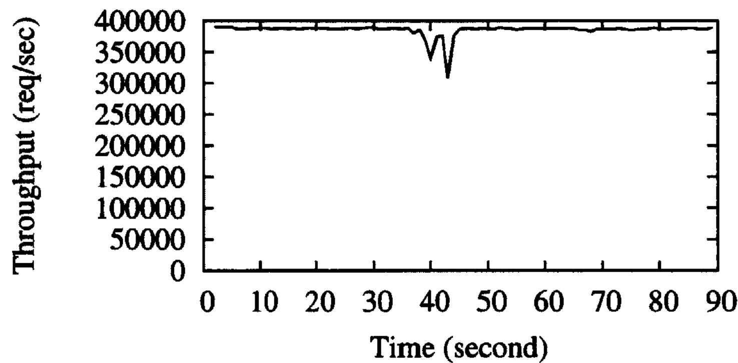


Figure 4-10: Throughput of Lazen with three replica groups. Each request puts an 8-byte value. A shard transfer takes place between time 38 and time 44.

More generally, since each replica accumulates request log entries while waiting for the previous SSD write to complete, and an SSD write takes about 0.5 ms, we'd expect about 0.5 ms of operations to be lost, during a simultaneous crash of a majority of replicas.

4.6.4 Scalability and Shard Transfer

Figure 4-10 shows the performance of Lazen with three replica groups. This experiment uses hard drives and Ethernet because we did not have enough SSDs or Infiniband cards. We start 240 client processes to generate puts of 8-byte values. For most of the graph, the three groups provide about 2.6× the throughput of a single group.

At time 38, the configuration manager starts the transfer of a shard of 100,000 keys. The transfer takes about six seconds, ending at time 44.

At time 40 the total throughput drops to 340,000, because the source replica group locks its hash table buckets one by one while deciding which keys need to be moved. Once that is done, the source replica group transfers the data, ending at time 42. Throughput drops further to 310,000 at time 43 when the source group deletes keys it is no longer serving from its database. This drop is worse than when the source group locks its buckets because it is executed by the same thread that executes client requests. Lazen restores ordinary throughput after the transfer is complete.

4.6.5 System Comparison

This section compares the throughput of Lazen with those of ZooKeeper and MongoDB, both of which are replicated storage systems. The experiments use SSDs and Ethernet (ZooKeeper and MongoDB would need modification to use Infiniband). The comparison is unfair because ZooKeeper and MongoDB have many more capabilities than Lazen; on the other hand, the SSDs' fast writes favor ZooKeeper by reducing the value of Lazen's deferred log writes. The workload consists of puts to random keys. Each experiment lasts 30 seconds. Each system has a single replica group consisting of three replicas.

	1	1024
ZooKeeper	31K	29K
MongoDB	17K	14K
Lazen	150K	50K
Lazen-batch	861K	53K

Figure 4-11: Throughput (req/sec) of different systems for put requests with 1-byte and 1024-byte values.

We configured ZooKeeper 3.4.6 in the same way as the ZooKeeper paper [42]: we disable snapshots, set the global outstanding limit to 2,000, and generate load with 100 single-threaded client processes, each of which keeps 100 requests outstanding. Clients send requests in batches using ZooKeeper's asynchronous API. This configuration maximizes the throughput.

For MongoDB 2.4.9, a single client uses the C++ driver to issue requests without waiting for replies. After 30 seconds the client waits for the server to indicate that all requests have been propagated to a majority. The wait is necessary because MongoDB acknowledges the client before the request is executed. For the same reason, a single client can saturate MongoDB.

Figure 4-11 shows the results. Our results for ZooKeeper are 1.5× the published numbers [44, 42] due to our faster hardware. MongoDB may be limited by the single read/write lock on the database storing its oplog [13]. Lazen yields the highest throughput, and is limited by the per-packet CPU costs at the primary caused by each packet containing only one request or reply. When clients batch requests in the same way as ZooKeeper, Lazen achieves much higher throughput as indicated by "Lazen-batch". The throughput of Lazen and Lazen-batch on 1024-byte values is limited by the throughput of Gigabit Ethernet, which is why it is much lower than the Infiniband numbers in Figure 4-5.

4.6.6 File System Backend

This section compares the performance of Lazy VSR and VSR-Sync on our file system backend. We compared the performance on workloads from filebench [3]. We ran the experiments on three replicas with SSD and Infiniband. Figure 4-12 shows the result. "Lazy-VSR" shows the performance of lazy-disk. "VSR-Sync" stands for a lazy-disk variant that uses VSR-Sync as the replication protocol.

"Random-Read" keeps reading 8KB of data at a random offset into a large file using direct I/O. "Random-Write" does writes in a similar way. Each operation of the "Seqwrite" benchmark appends 1MB of data to a file and calls fdatasync to write the data to disk. "Varmail" performs a sequence of create-append-sync, read-append-sync, reads, and deletes in the same directory.

On all workloads, Lazy VSR outperforms VSR-Sync. On "Random-Read" and "Random-Write", Lazy VSR is more than 4× as fast as VSR-Sync. The reason is that Lazy-VSR avoids disk latency in the critical path for normal operations. The improvement on on "Seqwrite" and "Varmail" is less because these workloads call "fsync" frequently, in which case Lazy-VSR must also wait for disk latencies.

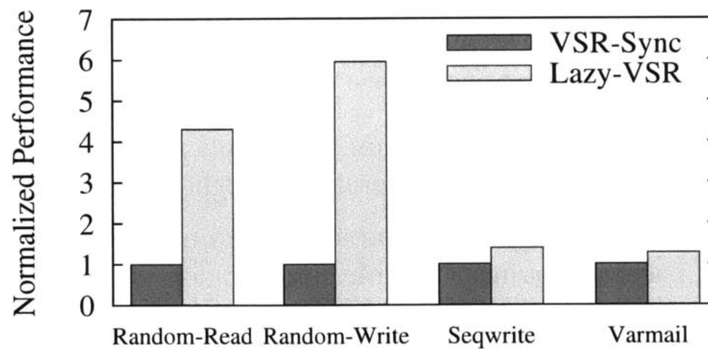


Figure 4-12: Performance of our file system backend on Lazy VSR and VSR-Sync

The conclusion is that, for replicated file system backends, Lazy VSR provides better performance than VSR-Sync by avoiding disk latency in the critical path.

4.7 Discussion

Several things regarding Lazy VSR are worth discussion:

- Current Lazen doesn't support range queries. It stores all key-value pairs in a large preallocated in-memory hash table. One challenge to support ordering is to choose the right data structure. One might use Masstree's tree, PALM, or ART. However, these trees have different performance implications. Masstree might provide better latency but worse throughput compared to PALM. ART might provide the best single-thread throughput, but cannot scale by adding multiple processing thread at each Lazen replica because it doesn't support concurrent updates. Figuring out the requirements of some real-world workloads can help us make a better decision.

- The peak throughput of the current implementation of Lazen, which is 150K requests per second, is limited by the single processing thread at the primary. However, adding more processing threads results in slightly lower throughput due to the coordination overhead. Multiple processing threads process all requests in one PREPARE message (which contains multiple client requests because the primary sends requests in batches to reduce network overhead) at a time. To ensure that all requests are processed in order, Lazen processes requests for the same key in order and acknowledges requests from the same client in order. Experiment shows that such coordination overhead offsets the performance gain of having more processing threads.

  To scale with the number of processing threads, clients can keep more requests outstanding so that each PREPARE message would contain more requests and the coordination overhead would be amortized. One approach is for clients to send requests in large batches. However, batching is not always possible [52] and the result depends on the batch size. For example, sending 100 outstanding requests from 100 clients

with a batch size of one provides lower latency than sending from a single client with a batch size of 100. The result will be more interesting if clients use user-level network stack [52] and send one request per packet.

- While Lazy VSR is a variant of quorum-based replication protocols, building a variant of non quorum-based replication protocols might require similar solution.

  One non-quorum based replication protocol is primary/backup replication such as Pacifica [53]. Pacifica requires a reliable configuration service (such as ZooKeeper), which is replicated with quorum-based replication protocol (such as Zab), to decide the current view. Each view consists of one primary and at most f backups. For each request, the primary broadcasts it to all backups, and commits the request *iff* **all** backups accept the request (thus they are not quorum-based).

  For primary/backup replication, we believe that taking writes to stable storage off the critical path proposes the same challenge and requires similar solution. Lazy durability is a design choice we must make regardless of the protocol we inherit from. To handle loss of acknowledged writes, the Pacifica variant also needs the durable point to optimize the protocol. For example, since the replicated service needs to keep undo information to rollback operations, the protocol has to be able to truncate the undo information. The durable point allows such optimizations.

- After we developed Lazy VSR, we realized that the durable point is very similar to the commit point in traditional quorum-based replication protocols. After all, their commit points are deduced from the on-disk logs of replicas, from which Lazy VSR also computes the durable point. The unique contribution of Lazy VSR is showing that one can separate the durable point from the "in-memory commit point" such that normal operation is fast with the in-memory commit point, and the rest of the protocol can be made efficient with optimizations enabled by the durable point.

Chapter 5

Conclusion

This dissertation presents two techniques for high-performance storage. Masstree is a high performance in-memory key-value storage system that runs on a single multi-core server. Its key idea is an in-memory trie of concurrent B+trees, which hides DRAM latencies to achieve high performance. Lazy VSR is a replication protocol that is fast (i.e. it has no disk write in the critical path), provides high availability (i.e. it tolerates all clean failures), and requires only ordinary disks (i.e. it doesn't require battery backups). Both Masstree and Lazy VSR have significant performance advantages over existing approaches.

Appendix A

TLA Specification and Partial Proof for Lazy VSR

This section presents the TLA specification of Lazy VSR protocol, a proof for some properties of the protocol, and several unit tests which verifies the protocol using TLA's model checker. The source code of the specification is `git@g.csail.mit.edu:w-doc/w.tla`. Theorem Safety asserts that the protocol is type safe and that the view number each server is using is monotonically increasing. The proof of the theorem is verified by TLAPS. This proof corresponds to Lemma 4.3.4, which is the basis of the manual proof presented in Section 4.3.

MODULE w

```
EXTENDS FiniteSets, Sequences, TLC, Naturals, TLAPS, FunctionTheorems

  The set of values go into the log
CONSTANTS ClientRequests, N

ASSUME NumberAssumption ≜ N ∈ Nat ∧ N > 0

  on-disk log containing the operations of all replicas.
VARIABLES diskLog
VARIABLES rvn  maximum the replica has ever seen

  in memory state
VARIABLES memLog
VARIABLES next  primary's next operation number
VARIABLES state
VARIABLES proposing  propsing[i]: the LogEntry replica i is currently proposing.
                     Used only if i is a leader
VARIABLES accepts  accpets[i]: set of replica numbers that accepts proposing[i].
                   Used only if i is a leader

  All outstanding messages
VARIABLES messages

  all state except for messages
allstate ≜ ⟨diskLog, memLog, rvn, next, state, accepts, proposing⟩
  all variables
allvars ≜ ⟨diskLog, memLog, rvn, next, state, accepts, proposing, messages⟩
  variables that will only be accessed by replica in Leader_Decided state
var_leader ≜ ⟨accepts, proposing, next⟩
  variables that will be modified due to leader election
var_elect ≜ ⟨state, rvn⟩
  variables that will only be modified by follower
var_log ≜ ⟨diskLog, memLog⟩

  Debug helper, for Model Check purpose only
Debug(c, msg) ≜ IF c THEN Print(msg, TRUE) ELSE TRUE

  Type definitions
  Zenon or Isabelle doesn't support .. operator!
Proc ≜ 1 .. N
  Proc ≜ {x ∈ Nat : x ≥ 1 ∧ x ≤ N}
  the sequence number, the id of the replica
ViewNum ≜ Nat × Proc
  the ballot the leader composes the LogEntry, the operation number
Viewstamp ≜ ViewNum × Nat
NonRequest ≜ CHOOSE v : v ∉ ClientRequests
```

LogID of this log entry, LogID of previous log entry, requests

$LogEntry \triangleq Viewstamp \times Viewstamp \times (ClientRequests \cup \{NonRequest\})$

$LeaderState \triangleq \{\text{"Leader_Elected"}, \text{"Leader_Decided"}\}$

$State \triangleq LeaderState \cup \{\text{"Backup"}, \text{"Candidate"}\}$

$NonViewNum \triangleq \text{CHOOSE } v : v \notin ViewNum$

$NonViewstamp \triangleq \text{CHOOSE } v : v \notin Viewstamp$

$NonLogEntry \triangleq \text{CHOOSE } v : v \notin LogEntry$

@sent_at and @lastop are copied from the request's @bexpected and @mylastop

$CatchUpReply \triangleq$

$[sent_at: ViewNum,\ lastop: Viewstamp,\ \text{lastcomm}: Viewstamp,\ log: LogEntry]$

$CatchUpReply \triangleq$

$\quad ViewNum \times (Viewstamp \cup \{NonViewstamp\}) \times$

$\quad (Viewstamp \cup \{NonViewstamp\}) \times (LogEntry \cup \{NonLogEntry\})$

$GeneralMessage \triangleq ViewNum \times (Viewstamp \cup \{NonViewstamp\})$

$\forall m \in Message: m.type \triangleq m[1] \wedge m.src \triangleq m[2] \wedge m.dst \triangleq m[3]$

$Message \triangleq$

$\quad (\{\text{"Prepare"}\} \times Proc \times Proc \times LogEntry)$

$\quad \cup (\{\text{"PrepareOK"}\} \times Proc \times Proc \times$

$\qquad (Viewstamp \times (Viewstamp \cup \{NonViewstamp\})))$

$\quad \cup (\{\text{"ElectSyncOK"}\} \times Proc \times Proc \times ViewNum)$

$\quad \cup (\{\text{"ElectPrepare"}, \text{"ElectSync"}, \text{"CatchUp"}, \text{"ElectPrepareOK"}\} \times$

$\qquad Proc \times Proc \times GeneralMessage)$

$\quad \cup (\{\text{"CatchUpReply"}\} \times Proc \times Proc \times CatchUpReply)$

Helpers

$WithMessage(m, msgs) \triangleq \text{IF } m \in \text{DOMAIN } msgs \text{ THEN } [msgs \text{ EXCEPT } ![m] = msgs[m] + 1]$
$\qquad \text{ELSE } msgs \mathbin{@@} (m :> 1)$

$WithoutMessage(m, msgs) \triangleq [msgs \text{ EXCEPT } ![m] = msgs[m] - 1]$

$Discard(m) \triangleq messages[m] > 0 \wedge messages' = WithoutMessage(m, messages)$

$CompleteRPC(reply, req) \triangleq messages' = WithMessage(reply, WithoutMessage(req, messages))$

$LastViewstamp(log) \triangleq \text{IF } Len(log) > 0 \text{ THEN } log[Len(log)][1] \text{ ELSE } NonViewstamp$

$LastLogEntryVNum(log) \triangleq \text{IF } Len(log) > 0 \text{ THEN } log[Len(log)][1][1] \text{ ELSE } NonViewNum$

$a \prec b \triangleq$

$\quad \text{IF } a = NonViewNum \text{ THEN } b \neq NonViewNum$

$\quad \text{ELSE } (\text{IF } b = NonViewNum \text{ THEN FALSE}$

$\qquad \text{ELSE } (a[1] < b[1] \vee (a[1] = b[1] \wedge a[2] < b[2])))$

$NextViewNum(a, i) \triangleq \text{IF } a = NonViewNum \text{ THEN } \langle 1, i\rangle \text{ ELSE } \langle a[1] + 1, i\rangle$

$SetRVN(i, n) \triangleq rvn' = [rvn \text{ EXCEPT } ![i] = n]$

$SetState(i, s) \triangleq state' = [state \text{ EXCEPT } ![i] = s]$

$SetNextOpnum(i, n) \triangleq next' = [next \text{ EXCEPT } ![i] = n]$

$SetAcceptSet(i, s) \triangleq accepts' = [accepts \text{ EXCEPT } ![i] = s]$

$AddAccept(i, s) \triangleq accepts' = [accepts \text{ EXCEPT } ![i] = accepts[i] \cup \{s\}]$

$RawSend(i, j, t, v) \triangleq messages' = WithMessage(\langle t, i, j, v \rangle, messages)$
$Majority(n) \triangleq n * 2 > N$

$ViewstampLess(a, b) \triangleq$
IF $a = NonViewstamp$ THEN $b \neq NonViewstamp$
ELSE (IF $b = NonViewstamp$ THEN FALSE
ELSE $a[1] \prec b[1] \lor (a[1] = b[1] \land a[2] < b[2])$)

$ViewstampLessEqual(a, b) \triangleq ViewstampLess(a, b) \lor a = b$

$OpMax(i) \triangleq LastViewstamp(memLog[i])$

$SetProposing(i, opnum, prev, req) \triangleq$
$proposing' = [proposing$ EXCEPT $![i] = \langle\langle rvn[i], opnum\rangle, prev, req\rangle]$

$LastOpnum(i) \triangleq$ IF $OpMax(i) = NonViewstamp$ THEN 0 ELSE $OpMax(i)[2]$
$StopProposing(i) \triangleq proposing' = [proposing$ EXCEPT $![i] = NonLogEntry]$

The maximum *LogEntry* in @*log* that is $\leq$ @*lastop_oln*
This is where the replica must first synchronized to before it can
roll forward

$LowerBound(log, lastop_vst) \triangleq$
IF $lastop_vst = NonViewstamp \lor Len(log) = 0 \lor$
$ViewstampLess(lastop_vst, log[1][1])$ THEN
$NonViewstamp$
ELSE
LET $index \triangleq$ CHOOSE $i \in$ DOMAIN log :
$\land ViewstampLessEqual(log[i][1], lastop_vst)$
$\land \neg(\exists j \in$ DOMAIN $log : \land ViewstampLessEqual(log[j][1], lastop_vst)$
$\land ViewstampLess(log[i][1], log[j][1]))$
IN $log[index][1]$

$FlushLog(i) \triangleq diskLog' = [diskLog$ EXCEPT $![i] = memLog[i]]$

The *LogEntry* entry follows @*LogID*

$NextEntry(log, vst) \triangleq$
IF $Len(log) = 0 \lor ViewstampLessEqual(log[Len(log)][1], vst)$ THEN
$NonLogEntry$
ELSE IF $vst = NonViewstamp$ THEN
$log[1]$
ELSE
LET $index \triangleq$ CHOOSE $i \in$ DOMAIN $log : log[i][1][2] = vst[2] + 1$
IN $log[index]$

$BuildCatchUpReply(d, lastop) \triangleq$
LET $vst \triangleq LowerBound(memLog[d], lastop)$
IN IF $vst = NonViewstamp \lor vst \neq lastop$ THEN
Ask the replica to rollback

$\langle rvn[d],\ lastop,\ vst,\ NonLogEntry \rangle$

ELSE

Roll forward by one *log* entry

$\langle rvn[d],\ lastop,\ vst,\ NextEntry(memLog[d],\ vst) \rangle$

$TruncateLog(log,\ vst) \triangleq$

$\text{IF } vst = NonViewstamp \lor Len(log) = 0 \lor$

$\quad ViewstampLess(vst,\ log[1][1]) \text{ THEN}$

$\quad \langle \rangle$

$\text{ELSE LET } index \triangleq \text{CHOOSE } i \in \text{DOMAIN } log :$

$\qquad \land \neg ViewstampLess(vst,\ log[i][1])$

$\qquad \land (\lor i = Len(log)$

$\qquad\quad \lor ViewstampLess(vst,\ log[i+1][1]))$

$\quad \text{IN } SubSeq(log,\ 1,\ index)$

$TruncateBothLogsTo(d,\ vst) \triangleq$

$\land memLog' = [memLog \text{ EXCEPT } ![d] = TruncateLog(memLog[d],\ vst)]$

$\land diskLog' = [diskLog \text{ EXCEPT } ![d] = TruncateLog(diskLog[d],\ vst)]$

Initial state

$Init1 \triangleq$ **Follower state**

$\land rvn = [p \in Proc \mapsto NonViewNum]$

$\land memLog = [p \in Proc \mapsto \langle \rangle]$

$\land diskLog = [p \in Proc \mapsto \langle \rangle]$

Leader state

$\land next = [p \in Proc \mapsto 0]$

$\land state = [p \in Proc \mapsto \text{"Backup"}]$

$\land accepts = [p \in Proc \mapsto \{\}]$

$\land proposing = [p \in Proc \mapsto NonLogEntry]$

messages

$\land messages = [m \in Message \mapsto 0]$

Note: Use $\rightarrow$, not $\mapsto$ to assert the types of functions

$TypeInv \triangleq \land (leader_t{::}\ rvn \in [Proc \rightarrow ViewNum \cup \{NonViewNum\}])$

$\qquad \land (mem_log_t{::}\ memLog \in [Proc \rightarrow Seq(LogEntry)])$

$\qquad \land (disk_log_t{::}\ diskLog \in [Proc \rightarrow Seq(LogEntry)])$

$\qquad \land (next_opnum_t{::}\ next \in [Proc \rightarrow Nat])$

$\qquad \land (state_t{::}\ state \in [Proc \rightarrow State])$

$\qquad \land (accepts_t{::}\ accepts \in [Proc \rightarrow \text{SUBSET } Message])$

$\qquad \land (proposing_t{::}\ proposing \in [Proc \rightarrow LogEntry \cup \{NonLogEntry\}])$

$\qquad \land (msg_t{::}\ messages \in [Message \rightarrow Nat])$

Actions below

$DuplicateMessage \triangleq$

$\land \exists\, m \in \text{DOMAIN } messages : messages' = WithMessage(m,\ messages)$

$\land \text{UNCHANGED } allstate$

$DropMessage \triangleq$
$\quad \land \exists m \in \text{DOMAIN } messages : messages[m] > 0 \land Discard(m)$
$\quad \land \text{UNCHANGED } allstate$

Restart. Lose everything except for on disk state, *e.g.* $f_log[i]$ and $f_voted[i]$

$Restart(i) \triangleq$
$\quad \land SetState(i, \text{"Backup"})$
$\quad \land memLog' = [memLog \text{ EXCEPT } ![i] = diskLog[i]]$
$\quad \land SetNextOpnum(i, 0)$
$\quad \land SetAcceptSet(i, \{\})$
$\quad \land StopProposing(i)$
$\quad$ Discard messages targeted @i
$\quad \land messages' = [m \in \text{DOMAIN } messages \mapsto \text{IF } m[3] = i \text{ THEN } 0 \text{ ELSE } messages[m]]$
$\quad \land \text{UNCHANGED } \langle rvn, diskLog \rangle$

Timeout

$Timeout(i) \triangleq$
$\quad \text{LET } nb \triangleq NextViewNum(rvn[i], i) \text{IN}$
$\quad$ The only case that we don't trigger *Timeout* is on the idling primary
$\quad \land \neg(state[i] = \text{"Leader_Decided"} \land proposing = NonLogEntry)$
$\quad \land SetState(i, \text{"Candidate"})$
$\quad$ clear accept set for each new *ViewNum*
$\quad \land SetAcceptSet(i, \{\langle \text{"ElectPrepareOK"}, i, i, \langle nb, OpMax(i) \rangle \rangle\})$
$\quad \land StopProposing(i)$
$\quad \land SetRVN(i, nb)$
$\quad \land FlushLog(i)$
$\quad \land \text{UNCHANGED } \langle memLog, next, messages \rangle$

Send *ElectPrepare* iff in *Candidate* state to ensure that $l_cur_ballot[i]$ is valid.

$SendElectPrepare(i, j) \triangleq$
$\quad \land state[i] = \text{"Candidate"} \land rvn[i] \neq NonViewNum \land i \neq j$
$\quad \land RawSend(i, j, \text{"ElectPrepare"}, \langle rvn[i], OpMax(i) \rangle)$
$\quad \land \text{UNCHANGED } allstate$

$ProcessElectPrepare(m) \triangleq$
$\quad \text{LET } s \triangleq m[2] d \triangleq m[3]$
$\qquad enabled \triangleq rvn[d] \prec m[4][1] \land s \neq d \land$
$\qquad\qquad ViewstampLessEqual(m[4][2], OpMax(d)) \text{IN}$
$\quad \lor (\land enabled$
$\qquad \land SetRVN(d, m[4][1])$
$\qquad \land SetState(d, \text{"Backup"})$ Step down from leader
$\qquad \land StopProposing(d)$
$\qquad \land FlushLog(d)$
$\qquad \land CompleteRPC(\langle \text{"ElectPrepareOK"}, d, s, \langle m[4][1], OpMax(d) \rangle \rangle, m)$
$\qquad \land \text{UNCHANGED } memLog \land \text{UNCHANGED } \langle next, accepts \rangle)$
$\quad \lor (\neg enabled \land \text{UNCHANGED } allstate \land Discard(m))$

$NotAccepted(s, d) \triangleq \forall m \in accepts[d] : m[2] \neq s$

$ProcessElectPrepareOK(m) \triangleq$

$\quad \text{LET } s \triangleq m[2] d \triangleq m[3]$

$\quad\quad enabled \triangleq \wedge state[d] = \text{"Candidate"} \wedge rvn[d] = m[4][1] \wedge NotAccepted(s, d)$

$\quad \text{IN}$

$\quad \vee (\wedge enabled$

$\quad\quad \wedge \text{ IF } Majority(Cardinality(accepts[d]) + 1) \text{ THEN}$

$\quad\quad\quad SetState(d, \text{"Leader_Elected"}) \wedge SetAcceptSet(d, \{\})$

$\quad\quad\quad \text{ELSE}$

$\quad\quad\quad \text{UNCHANGED } state \wedge AddAccept(d, m)$

$\quad\quad \wedge \text{ UNCHANGED } var_log \wedge \text{ UNCHANGED } \langle rvn, next, proposing \rangle)$

$\quad \vee (\neg enabled \wedge \text{ UNCHANGED } allstate)$ discard stale/duplicated reply

$SendElectSync(i, j) \triangleq$

$\quad \wedge state[i] = \text{"Leader_Elected"} \wedge rvn[i] \neq NonViewNum$

$\quad \wedge RawSend(i, j, \text{"ElectSync"}, \langle rvn[i], OpMax(i) \rangle)$

$\quad \wedge \text{ UNCHANGED } allstate$

$ProcessElectSync(m) \triangleq$

$\quad \text{LET } s \triangleq m[2] d \triangleq m[3]$

$\quad\quad enabled \triangleq rvn[d] = m[4][1] \wedge OpMax(d) = m[4][2]$

$\quad \text{IN}$

$\quad \vee (\wedge enabled$

$\quad\quad \wedge FlushLog(d)$ flush the *log* to disk. For simplicity

$\quad\quad \wedge CompleteRPC(\langle \text{"ElectSyncOK"}, d, s, m[4][1] \rangle, m)$

$\quad\quad \wedge \text{ UNCHANGED } var_leader \wedge \text{ UNCHANGED } \langle memLog, rvn, state \rangle)$

$\quad \vee (\neg enabled \wedge \text{ UNCHANGED } allstate \wedge Discard(m))$

$ProcessElectSyncOK(m) \triangleq$

$\quad \text{LET } s \triangleq m[2] d \triangleq m[3]$

$\quad\quad enabled \triangleq rvn[d] = m[4] \wedge state[d] = \text{"Leader_Elected"} \wedge NotAccepted(s, d)$

$\quad \text{IN}$

$\quad \vee (\wedge enabled$

$\quad\quad \wedge \text{ IF } Majority(Cardinality(accepts[d]) + 1) \text{ THEN}$

$\quad\quad\quad \wedge SetState(d, \text{"Leader_Decided"})$

$\quad\quad\quad \wedge SetAcceptSet(d, \{\})$

$\quad\quad\quad \wedge SetProposing(d, LastOpnum(d) + 1, OpMax(d), NonRequest)$

$\quad\quad\quad \wedge SetNextOpnum(d, LastOpnum(d) + 2)$

$\quad\quad\quad \wedge \text{ UNCHANGED } var_log \wedge \text{ UNCHANGED } rvn$

$\quad\quad\quad \text{ELSE}$

$\quad\quad\quad AddAccept(d, m) \wedge \text{ UNCHANGED } var_log \wedge \text{ UNCHANGED } var_elect \wedge$

$\quad\quad\quad \text{UNCHANGED } \langle proposing, next \rangle)$

$\quad \vee (\neg enabled \wedge \text{ UNCHANGED } allstate)$ discard stale/duplicated reply

$HandleClientRequest(i, req) \triangleq$

```
    ∧ state[i] = "Leader_Decided" ∧ proposing[i] = NonLogEntry
    ∧ SetProposing(i, next[i], OpMax(i), req)
    ∧ SetNextOpnum(i, next[i] + 1)
    ∧ SetAcceptSet(i, {})
    ∧ UNCHANGED var_log ∧ UNCHANGED var_elect ∧ UNCHANGED messages

SendPrepare(i, j) ≜
    ∧   state[i] = "Leader_Decided" ∧ proposing[i] ≠ NonLogEntry
    ∧   RawSend(i, j, "Prepare", proposing[i])
    ∧   UNCHANGED allstate

ProcessPrepare(m) ≜
    LET s ≜ m[2]d ≜ m[3]log ≜ m[4]
        accept ≜ rvn[d] = log[1][1] ∧ OpMax(d) = log[2]  continuous condition
        learn ≜ rvn[d] ≺ log[1][1]
    IN
    ∨ ( ∧ accept
        ∧ memLog' = [memLog EXCEPT ![d] = Append(memLog[d], log)]
        ∧ IF log[3] = NonRequest THEN
              FlushLog(d)  flush to disk on first message from a new leader
            ELSE
              UNCHANGED diskLog
        ∧ CompleteRPC(⟨"PrepareOK", d, s, ⟨log[1], LastViewstamp(diskLog[d])⟩⟩, m)
        ∧ UNCHANGED var_leader ∧ UNCHANGED var_elect)
    ∨ ( ∧ ¬accept ∧ learn
        ∧ SetRVN(d, log[1][1])  learn new leader to enable catch up thread
        ∧ SetState(d, "Backup")
        ∧ UNCHANGED var_log ∧ UNCHANGED var_leader
        ∧ Discard(m))
    ∨ (¬accept ∧ ¬learn ∧ UNCHANGED allstate ∧ Discard(m))

ProcessPrepareOK(m) ≜
    LET s ≜ m[2]d ≜ m[3]
        enabled ≜ ∧ state[d] = "Leader_Decided" ∧ proposing[d] ≠ NonLogEntry
                  ∧ proposing[d][1] = m[4][1] ∧ NotAccepted(s, d)
    IN
    ∨ ( ∧ enabled
        ∧ AddAccept(d, m)
        ∧ IF Majority(Cardinality(accepts[d]) + 1) THEN
              StopProposing(d)
            ELSE
              UNCHANGED proposing)
        ∧ UNCHANGED var_log ∧ UNCHANGED var_elect ∧ UNCHANGED next
    ∨ (¬enabled ∧ UNCHANGED allstate)

SendCatchUp(i, j) ≜
```

```
      ∧   i ≠ j ∧ state[i] ∉ LeaderState ∧ rvn[i] ≠ NonViewNum ∧ rvn[i][1] = j
      ∧   RawSend(i, j, "CatchUp", ⟨rvn[i], OpMax(i)⟩)
      ∧   UNCHANGED allstate

ProcessCatchUp(m) ≜
      LET s ≜ m[2]d ≜ m[3]
          enabled ≜ state[d] ∈ LeaderState ∧ rvn[d] ≠ NonViewNum ∧ rvn[d] = m[4][1]
      IN
      ∨ ( ∧ enabled
          The format of catchup reply is: ⟨rvn[d], lastop, lowerbound, NonLogEntry⟩
          ∧ CompleteRPC(⟨"CatchUpReply", d, s, BuildCatchUpReply(d, m[4][2])⟩, m)
          ∧ UNCHANGED allstate)
      ∨ (¬enabled ∧ UNCHANGED allstate ∧ Discard(m))

ProcessCatchUpReply(m) ≜
      LET s ≜ m[2]d ≜ m[3]
          enabled ≜ ∧ state[d] ∉ LeaderState
                    ∧ rvn[d] = m[4][1]  m.v.sent_at
                    ∧ m[4][2] = OpMax(d)
      IN
      ∨ ( ∧ enabled
          ∧ IF m[4][4] = NonLogEntry THEN
                roll back to ≤ m.v.lowerbound
                TruncateBothLogsTo(d, m[4][3])
            ELSE
                roll forward by appending m.v.log to both logs, for simplicity
                ∧ memLog′ = [memLog EXCEPT ![d] = Append(memLog[d], m[4][4])]
                ∧ diskLog′ = [diskLog EXCEPT ![d] = Append(diskLog[d], m[4][4])]
          ∧ UNCHANGED var_leader ∧ UNCHANGED var_elect)
      ∨ (¬enabled ∧ UNCHANGED allstate)

Receive ≜ ∃ m ∈ DOMAIN messages :
            ∧ messages[m] > 0
            ∧ ( ∨ (m[1] = "ElectPrepare" ∧ ProcessElectPrepare(m))
                ∨ (m[1] = "ElectPrepareOK" ∧ ProcessElectPrepareOK(m) ∧ Discard(m))
                ∨ (m[1] = "ElectSync" ∧ ProcessElectSync(m))
                ∨ (m[1] = "ElectSyncOK" ∧ ProcessElectSyncOK(m) ∧ Discard(m))
                ∨ (m[1] = "Prepare" ∧ ProcessPrepare(m))
                ∨ (m[1] = "PrepareOK" ∧ ProcessPrepareOK(m) ∧ Discard(m))
                ∨ (m[1] = "CatchUp" ∧ ProcessCatchUp(m))
                ∨ (m[1] = "CatchUpReply" ∧ ProcessCatchUpReply(m) ∧ Discard(m)))

FlushToDisk(i) ≜
      ∧   memLog[i] ≠ diskLog[i]  optimization
      ∧   FlushLog(i)
      ∧   UNCHANGED var_leader ∧ UNCHANGED var_elect ∧ UNCHANGED ⟨memLog, messages⟩
```

```
OneStep ≜
    ∨ Receive
    ∨ ∃ i, j ∈ Proc : SendPrepare(i, j)
    ∨ ∃ i, j ∈ Proc : SendElectPrepare(i, j)
    ∨ ∃ i, j ∈ Proc : SendElectSync(i, j)
    ∨ ∃ i, j ∈ Proc : SendCatchUp(i, j)
    ∨ ∃ i ∈ Proc, req ∈ ClientRequests : HandleClientRequest(i, req)
    ∨ ∃ i ∈ Proc : Timeout(i)
    ∨ ∃ i ∈ Proc : Restart(i)
    ∨ ∃ i ∈ Proc : FlushToDisk(i)
    ∨ DropMessage
    ∨ DuplicateMessage

Spec ≜ Init1 ∧ □[OneStep]_allstate

Sender(m) ≜
        IF m[1] = "Prepare" THEN
            m[4][1][1]
         ELSE IF m[1] = "ElectPrepare" THEN
           m[4][1]
         ELSE IF m[1] = "ElectSync" THEN
           m[4][1]
         ELSE
           NonViewNum

OtherSafety ≜ ∀ i ∈ Proc : UNCHANGED rvn[i] ∨ rvn[i] ≺ rvn'[i]

THEOREM NextBallotSafety ≜
   ASSUME NEW a ∈ ViewNum ∪ {NonViewNum}, NEW i ∈ Proc
   PROVE a ≺ NextViewNum(a, i)
 OMITTED

THEOREM TruncateLogTypeSafety ≜
   ASSUME NEW log ∈ Seq(LogEntry), NEW vst ∈ Viewstamp ∪ {NonViewstamp}
   PROVE TruncateLog(log, vst) ∈ Seq(LogEntry)
 OMITTED

THEOREM LastLogIDTypeSafety ≜
   ASSUME NEW i ∈ Proc, TypeInv!disk_log_t
   PROVE LastViewstamp(diskLog[i]) ∈ Viewstamp ∪ {NonViewstamp}
 BY DEF LastViewstamp, LogEntry, ViewNum, NonViewNum, Viewstamp, NonViewstamp

THEOREM NextBallotTypeSafety ≜
   ASSUME NEW a ∈ ViewNum ∪ {NonViewNum}, NEW i ∈ Proc
   PROVE NextViewNum(a, i) ∈ ViewNum
BY DEF Proc, NextViewNum, ViewNum
```

THEOREM $SeenThruSafety \triangleq$
  ASSUME $TypeInv!mem_log_t$, NEW $d \in Proc$
  PROVE $OpMax(d) \in Viewstamp \cup \{NonViewstamp\}$
BY DEF $Proc$, $LogEntry$, $NonViewstamp$, $TypeInv$, $OpMax$

THEOREM $MessageTypeSafety1 \triangleq$
  ASSUME NEW MT, NEW $m \in MT$, NEW $msgs \in [MT \rightarrow Nat]$
  PROVE $WithMessage(m, msgs) \in [MT \rightarrow Nat]$
BY DEF $WithMessage$

THEOREM $MessageTypeSafety2 \triangleq$
  ASSUME NEW MT, NEW $m \in MT$, NEW $msgs \in [MT \rightarrow Nat]$, $msgs[m] > 0$
  PROVE $WithoutMessage(m, msgs) \in [MT \rightarrow Nat]$
BY DEF $WithoutMessage$

THEOREM $MessageTypeSafety3 \triangleq$
  ASSUME NEW MT, NEW $reply \in MT$, NEW $req \in MT$, NEW $msgs \in [MT \rightarrow Nat]$, $msgs[req] > 0$
  PROVE $WithMessage(reply, WithoutMessage(req, msgs)) \in [MT \rightarrow Nat]$
$\langle 1 \rangle$ USE DEF $WithoutMessage$, $WithMessage$
$\langle 1 \rangle 1.$ $WithoutMessage(req, msgs) \in [MT \rightarrow Nat]$
  BY $MessageTypeSafety2$
$\langle 1 \rangle 2.$ $WithMessage(reply, WithoutMessage(req, msgs)) \in [MT \rightarrow Nat]$
  BY $\langle 1 \rangle 1$, $MessageTypeSafety1$
$\langle 1 \rangle$ QED
  BY $\langle 1 \rangle 2$

THEOREM $ActionSafety1 \triangleq$
  ASSUME $TypeInv$, $DuplicateMessage$
  PROVE $TypeInv' \land OtherSafety$
$\langle 1 \rangle$ USE DEF $Proc$, $ViewNum$, $NonViewNum$,
        $Viewstamp$, $NonViewstamp$,
        $LogEntry$, $NonLogEntry$,
        $NonRequest$,
        $LeaderState$, $State$,
        $CatchUpReply$,
        $GeneralMessage$,
        $Message$,
        helper
        $TypeInv$
$\langle 1 \rangle 0.$ UNCHANGED $allstate$
  BY DEF $DuplicateMessage$
$\langle 1 \rangle 1.$ PICK $m \in$ DOMAIN $messages : messages' = WithMessage(m, messages)$
  BY $DuplicateMessage$ DEF $DuplicateMessage$
$\langle 1 \rangle a.$ $TypeInv!msg_t'$
  BY $\langle 1 \rangle 1$, $MessageTypeSafety1$
$\langle 1 \rangle b.$ $OtherSafety$

BY $\langle 1 \rangle 0$ DEF *OtherSafety*, *DuplicateMessage*, *allstate*
$\langle 1 \rangle$ QED
BY $\langle 1 \rangle 0$, $\langle 1 \rangle$a, $\langle 1 \rangle$b DEF *allstate*

THEOREM *ActionSafety*2 $\triangleq$
ASSUME *TypeInv*, *DropMessage*
PROVE $TypeInv' \wedge OtherSafety$
$\langle 1 \rangle$ USE DEF *allstate*, *Proc*,
ViewNum, *NonViewNum*,
Viewstamp, *NonViewstamp*,
LogEntry, *NonLogEntry*,
NonRequest,
LeaderState, *State*,
CatchUpReply,
GeneralMessage,
Message,
helper
TypeInv
$\langle 1 \rangle 2$. PICK $m \in$ DOMAIN $messages : Discard(m)$
BY DEF *DropMessage*
$\langle 1 \rangle 3$. $messages[m] > 0 \wedge messages' = WithoutMessage(m, messages)$
BY $\langle 1 \rangle 2$ DEF *Discard*
$\langle 1 \rangle$b. $WithoutMessage(m, messages) \in [Message \rightarrow Nat]$
BY ONLY $TypeInv!msg_t$, $\langle 1 \rangle 3$, *MessageTypeSafety*2
$\langle 1 \rangle 4$. UNCHANGED *allstate*
BY $\langle 1 \rangle 2$ DEF *DropMessage*
$\langle 1 \rangle$other. *OtherSafety*
BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$ DEF *DropMessage*, *OtherSafety*
$\langle 1 \rangle$ QED
BY $\langle 1 \rangle 3$, $\langle 1 \rangle$b, $\langle 1 \rangle 4$, $\langle 1 \rangle$other

THEOREM *ActionSafety*3 $\triangleq$
ASSUME *TypeInv*, $\exists i \in Proc : Timeout(i)$
PROVE $TypeInv' \wedge OtherSafety$
$\langle 1 \rangle$ USE DEF *allstate*, *Proc*,
ViewNum, *NonViewNum*,
Viewstamp, *NonViewstamp*,
LogEntry, *NonLogEntry*,
NonRequest,
helper
TypeInv, *Timeout*
$\langle 1 \rangle 2$. PICK $i \in Proc : Timeout(i)$
OBVIOUS
$\langle 1 \rangle 3$. UNCHANGED $\langle memLog, next, messages \rangle$
BY DEF *GeneralMessage*, *Message*, *CatchUpReply*

$\langle 1 \rangle 4.$ *TypeInv*!*state_t*$'$

BY *TypeInv*, $\langle 1 \rangle 2$ DEF *SetState*, *LeaderState*, *State*

$\langle 1 \rangle$ DEFINE $nb \triangleq NextViewNum(rvn[i], i)$

$m0 \triangleq \langle$ "ElectPrepareOK", $i, i, \langle nb, OpMax(i) \rangle \rangle$

$\langle 1 \rangle a.$ $nb \in ViewNum$

BY *TypeInv*, *NextBallotTypeSafety*

$\langle 1 \rangle b.$ $SetAcceptSet(i, \{m0\})$

BY $\langle 1 \rangle 2$

$\langle 1 \rangle c.$ $m0 \in Message$

BY $\langle 1 \rangle 2$, $\langle 1 \rangle a$, *SeenThruSafety* DEF *Message*, *GeneralMessage*

$\langle 1 \rangle 5.$ *TypeInv*!*accepts_t*$'$

BY *TypeInv*, $\langle 1 \rangle b$, $\langle 1 \rangle c$, $\langle 1 \rangle 2$ DEF *SetAcceptSet*

$\langle 1 \rangle 6.$ *TypeInv*!*proposing_t*$'$

BY *TypeInv*, $\langle 1 \rangle 2$ DEF *StopProposing*

$\langle 1 \rangle 7.$ *TypeInv*!*disk_log_t*$'$

BY *TypeInv*, $\langle 1 \rangle 2$ DEF *FlushLog*

$\langle 1 \rangle 8a.$ $SetRVN(i, nb)$

BY $\langle 1 \rangle 2$

$\langle 1 \rangle 8.$ *TypeInv*!*leader_t*$'$

BY $\langle 1 \rangle 2$, $\langle 1 \rangle a$, $\langle 1 \rangle 8a$ DEF *SetRVN*

$\langle 1 \rangle$incv. *OtherSafety*

$\langle 2 \rangle a.$ $rvn[i] \prec nb$

BY ONLY *NextBallotSafety*, *TypeInv*!*leader_t*, $\langle 1 \rangle 2$

$\langle 2 \rangle$ QED

BY $\langle 1 \rangle 2$, $\langle 2 \rangle a$, $\langle 1 \rangle 8a$ DEF *SetRVN*, *OtherSafety*

$\langle 1 \rangle$ QED

BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$, $\langle 1 \rangle 5$, $\langle 1 \rangle 6$, $\langle 1 \rangle 7$, $\langle 1 \rangle 8$, $\langle 1 \rangle$incv

THEOREM $ActionSafety4 \triangleq$

ASSUME *TypeInv*, $\exists\, i \in Proc : Restart(i)$

PROVE $TypeInv' \land OtherSafety$

$\langle 1 \rangle$ USE DEF *allstate*, *Proc*,

ViewNum, *NonViewNum*,

Viewstamp, *NonViewstamp*,

LogEntry, *NonLogEntry*,

NonRequest,

helper

TypeInv, *Restart*

$\langle 1 \rangle 2.$ PICK $i \in Proc : Restart(i)$

OBVIOUS

$\langle 1 \rangle 3.$ UNCHANGED *diskLog*

BY DEF *Restart*

$\langle 1 \rangle 4.$ *TypeInv*!*state_t*$'$

BY *TypeInv*, $\langle 1 \rangle 2$ DEF *SetState*, *LeaderState*, *State*

$\langle 1 \rangle 5.$ *TypeInv*!*accepts_t*$'$

BY *TypeInv*, ⟨1⟩2 DEF *SetAcceptSet*
⟨1⟩6. *TypeInv*!*proposing_t′*
BY *TypeInv*, ⟨1⟩2 DEF *StopProposing*
⟨1⟩8. *TypeInv*!*leader_t′*
BY *TypeInv*, ⟨1⟩2 DEF *SetRVN*, *LastLogEntryVNum*
⟨1⟩9. *TypeInv*!*mem_log_t′*
BY *TypeInv*, ⟨1⟩2
⟨1⟩10. *TypeInv*!*next_opnum_t′*
BY *TypeInv*, ⟨1⟩2 DEF *SetNextOpnum*
⟨1⟩11. *TypeInv*!*msg_t′*
BY *TypeInv*, ⟨1⟩2
⟨1⟩other. *OtherSafety*
BY ⟨1⟩3, ⟨1⟩2 DEF *OtherSafety*, *SetRVN*
⟨1⟩ QED
BY ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6, ⟨1⟩8, ⟨1⟩9, ⟨1⟩10, ⟨1⟩11, ⟨1⟩other

THEOREM *ActionSafety5* ≜
ASSUME *TypeInv*, ∃ *i* ∈ *Proc* : *FlushToDisk*(*i*)
PROVE *TypeInv′* ∧ *OtherSafety*
⟨1⟩ USE DEF *Proc*,
ViewNum, *NonViewNum*,
Viewstamp, *NonViewstamp*,
LogEntry, *NonLogEntry*,
NonRequest,
helper
TypeInv
⟨1⟩1. PICK *i* ∈ *Proc* : *FlushToDisk*(*i*)
OBVIOUS
⟨1⟩3. UNCHANGED *var_leader* ∧ UNCHANGED *var_elect* ∧ UNCHANGED ⟨*memLog*, *messages*⟩
BY DEF *FlushToDisk*
⟨1⟩4. *diskLog′* ∈ [*Proc* → *Seq*(*LogEntry*)]
BY DEF *FlushToDisk*, *FlushLog*
⟨1⟩ QED
BY ⟨1⟩3, ⟨1⟩4 DEF *var_leader*, *var_elect*, *OtherSafety*

THEOREM *ActionSafety6* ≜
ASSUME *TypeInv*, ∃ *i*, *j* ∈ *Proc* : *SendPrepare*(*i*, *j*)
PROVE *TypeInv′* ∧ *OtherSafety*
⟨1⟩ USE DEF *Proc*,
ViewNum, *NonViewNum*,
Viewstamp, *NonViewstamp*,
LogEntry, *NonLogEntry*,
NonRequest,
helper
TypeInv, *SendPrepare*

⟨1⟩2. PICK $i, j \in Proc : SendPrepare(i, j)$
OBVIOUS
⟨1⟩3. UNCHANGED *allstate*
BY ⟨1⟩2
⟨1⟩ DEFINE $m \triangleq \langle \text{"Prepare"}, i, j, proposing[i] \rangle$
⟨1⟩4. $TypeInv!msg_t'$
⟨2⟩1a. $messages' = WithMessage(m, messages)$
BY ⟨1⟩2 DEF *RawSend*
⟨2⟩2. $proposing[i] \in LogEntry \cup \{NonLogEntry\}$
OBVIOUS
⟨2⟩ USE DEF *GeneralMessage*, *CatchUpReply*, *Message*
⟨2⟩3. $m \in Message$
BY ⟨1⟩2, ⟨2⟩2
⟨2⟩ QED
BY ⟨2⟩1a, ⟨2⟩3, *MessageTypeSafety1*
⟨1⟩ QED
BY ⟨1⟩3, ⟨1⟩4 DEF *allstate*, *OtherSafety*

THEOREM $ActionSafety7 \triangleq$
ASSUME $TypeInv, \exists\, i, j \in Proc : SendElectPrepare(i, j)$
PROVE $TypeInv' \land OtherSafety$
⟨1⟩ USE DEF *Proc*,
ViewNum, *NonViewNum*,
Viewstamp, *NonViewstamp*,
LogEntry, *NonLogEntry*,
NonRequest,
helper
TypeInv, *SendElectPrepare*
⟨1⟩2. PICK $i, j \in Proc : SendElectPrepare(i, j)$
OBVIOUS
⟨1⟩3. UNCHANGED *allstate*
BY ⟨1⟩2
⟨1⟩4. $TypeInv!msg_t'$
⟨2⟩ DEFINE $m \triangleq \langle \text{"ElectPrepare"}, i, j, \langle rvn[i], OpMax(i) \rangle \rangle$
⟨2⟩1a. $messages' = WithMessage(m, messages)$
BY ⟨1⟩2 DEF *RawSend*
⟨2⟩2. $rvn[i] \in ViewNum \land OpMax(i) \in Viewstamp \cup \{NonViewstamp\}$
BY ⟨1⟩2, *SeenThruSafety*
⟨2⟩ USE DEF *GeneralMessage*, *CatchUpReply*, *Message*
⟨2⟩3. $m \in Message$
BY ⟨1⟩2, ⟨2⟩2
⟨2⟩ QED
BY ⟨1⟩2, ⟨2⟩1a, ⟨2⟩3, *MessageTypeSafety1*
⟨1⟩ QED
BY ⟨1⟩2, ⟨1⟩3, ⟨1⟩4 DEF *allstate*, *OtherSafety*

THEOREM $ActionSafety8 \triangleq$
    ASSUME $TypeInv, \exists i, j \in Proc : SendElectSync(i, j)$
    PROVE $TypeInv' \land OtherSafety$
$\langle 1 \rangle$ USE DEF $Proc,$
            $ViewNum, NonViewNum,$
            $Viewstamp, NonViewstamp,$
            $LogEntry, NonLogEntry,$
            $NonRequest,$
            helper
            $TypeInv, SendElectSync$
$\langle 1 \rangle 2.$ PICK $i, j \in Proc : SendElectSync(i, j)$
  OBVIOUS
$\langle 1 \rangle 3.$ UNCHANGED $allstate$
  BY $\langle 1 \rangle 2$
$\langle 1 \rangle 4.$ $TypeInv ! msg_t'$
  $\langle 2 \rangle$ DEFINE $m \triangleq \langle$"ElectSync"$, i, j, \langle rvn[i], OpMax(i) \rangle \rangle$
  $\langle 2 \rangle 1a.$ $messages' = WithMessage(m, messages)$
    BY $\langle 1 \rangle 2$ DEF $RawSend$
  $\langle 2 \rangle 2.$ $OpMax(i) \in (Viewstamp \cup \{NonViewstamp\})$
    BY $\langle 1 \rangle 2$, $SeenThruSafety$
  $\langle 2 \rangle$ USE DEF $GeneralMessage, CatchUpReply, Message$
  $\langle 2 \rangle 3.$ $m \in Message$
    BY $\langle 1 \rangle 2$, $\langle 2 \rangle 2$
  $\langle 2 \rangle$ QED
    BY $\langle 2 \rangle 1a$, $\langle 2 \rangle 3$, $MessageTypeSafety1$
$\langle 1 \rangle$ QED
  BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$ DEF $allstate, OtherSafety$

THEOREM $ActionSafety9 \triangleq$
    ASSUME $TypeInv, \exists i, j \in Proc : SendCatchUp(i, j)$
    PROVE $TypeInv' \land OtherSafety$
$\langle 1 \rangle$ USE DEF $Proc,$
            $ViewNum, NonViewNum,$
            $Viewstamp, NonViewstamp,$
            $LogEntry, NonLogEntry,$
            $NonRequest,$
            helper
            $TypeInv, SendCatchUp$
$\langle 1 \rangle 2.$ PICK $i, j \in Proc : SendCatchUp(i, j)$
  OBVIOUS
$\langle 1 \rangle 3.$ UNCHANGED $allstate$
  BY $\langle 1 \rangle 2$
$\langle 1 \rangle 4.$ $TypeInv ! msg_t'$
  $\langle 2 \rangle$ DEFINE $m \triangleq \langle$"CatchUp"$, i, j, \langle rvn[i], OpMax(i) \rangle \rangle$
  $\langle 2 \rangle 1a.$ $messages' = WithMessage(m, messages)$

BY $\langle 1 \rangle 2$ DEF *RawSend*

$\langle 2 \rangle 2.$ $rvn[i] \in ViewNum \land OpMax(i) \in (Viewstamp \cup \{NonViewstamp\})$

BY $\langle 1 \rangle 2$ DEF *OpMax*

$\langle 2 \rangle$ USE DEF *GeneralMessage*, *CatchUpReply*, *Message*

$\langle 2 \rangle 3.$ $m \in Message$

BY $\langle 1 \rangle 2$, $\langle 2 \rangle 2$

$\langle 2 \rangle$ QED

BY $\langle 1 \rangle 2$, $\langle 2 \rangle 1a$, $\langle 2 \rangle 3$, *MessageTypeSafety*1

$\langle 1 \rangle$ QED

BY $\langle 1 \rangle 3$, $\langle 1 \rangle 4$ DEF *allstate*, *OtherSafety*

THEOREM *ActionSafety*10 $\triangleq$

ASSUME $TypeInv$, $\exists\, i \in Proc,\ req \in ClientRequests : HandleClientRequest(i, req)$

PROVE $TypeInv' \land OtherSafety$

$\langle 1 \rangle$ USE DEF *Proc*,
ViewNum, *NonViewNum*,
Viewstamp, *NonViewstamp*,
LogEntry, *NonLogEntry*,
NonRequest,
helper
TypeInv, *HandleClientRequest*, *var_log*, *var_elect*

$\langle 1 \rangle 2.$ PICK $i \in Proc,\ req \in ClientRequests : HandleClientRequest(i, req)$

OBVIOUS

$\langle 1 \rangle 3.$ UNCHANGED var_log $\land$ UNCHANGED var_elect $\land$ UNCHANGED $messages$

OBVIOUS

$\langle 1 \rangle 4.$ $proposing' \in [Proc \rightarrow LogEntry \cup \{NonLogEntry\}]$

BY $\langle 1 \rangle 2$ DEF *SetProposing*

$\langle 1 \rangle 5.$ $TypeInv!accepts_t'$

BY $\langle 1 \rangle 2$ DEF *SetAcceptSet*

$\langle 1 \rangle 6.$ $next' \in [Proc \rightarrow Nat]$

BY $\langle 1 \rangle 2$ DEF *SetNextOpnum*

$\langle 1 \rangle$ QED

BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$, $\langle 1 \rangle 5$, $\langle 1 \rangle 6$ DEF *OtherSafety*

THEOREM *ActionSafety*11 $\triangleq$

ASSUME $TypeInv$,
$(\exists\, m \in$ DOMAIN $messages :$
$messages[m] > 0 \land m[1] = \text{"ElectPrepare"} \land ProcessElectPrepare(m))$

PROVE $TypeInv' \land OtherSafety$

$\langle 1 \rangle$ USE DEF *Proc*,
ViewNum, *NonViewNum*,
Viewstamp, *NonViewstamp*,
LogEntry, *NonLogEntry*,
NonRequest,
helper

TypeInv, *ProcessElectPrepare*

⟨1⟩2. PICK $m \in$ DOMAIN *messages* : $messages[m] > 0 \land m[1] =$ "ElectPrepare" $\land$ *ProcessElectPrepare*(m)
  OBVIOUS
⟨1⟩ DEFINE $s \triangleq m[2] d \triangleq m[3]$
⟨1⟩a. $m \in$ DOMAIN *messages*
  BY ⟨1⟩2
⟨1⟩b. $messages[m] > 0$
  BY ⟨1⟩2
⟨1⟩c. $m \in$ *Message*
  BY ⟨1⟩a
⟨1⟩limit. $s \in Proc \land d \in Proc$
  BY ⟨1⟩c DEF *GeneralMessage*, *CatchUpReply*, *Message*
⟨1⟩3.CASE $\neg$*ProcessElectPrepare*(m)! : ! *enabled*
  ⟨2⟩0. UNCHANGED *allstate* $\land$ *Discard*(m)
    BY ⟨1⟩2, ⟨1⟩3
  ⟨2⟩4. *TypeInv* ! *msg_t'*
    BY ⟨2⟩0, ⟨1⟩a, ⟨1⟩b, ⟨1⟩c, *MessageTypeSafety*2 DEF *Discard*
  ⟨2⟩ QED
    BY ⟨2⟩0, ⟨2⟩4 DEF *allstate*, *OtherSafety*
⟨1⟩4.CASE *ProcessElectPrepare*(m)! : ! *enabled*
  ⟨2⟩ USE DEF *GeneralMessage*, *CatchUpReply*, *Message*
  ⟨2⟩4. $m[4] \in$ *GeneralMessage*
    BY ⟨1⟩c, ⟨1⟩2
  ⟨2⟩5. *TypeInv* ! *leader_t'*
    BY ⟨1⟩2, ⟨1⟩4, ⟨2⟩4 DEF *SetRVN*
  ⟨2⟩6. *TypeInv* ! *disk_log_t'*
    ⟨3⟩a. *FlushLog*(d)
      BY ⟨1⟩2, ⟨1⟩4
    ⟨3⟩ QED
      BY ⟨1⟩limit, ⟨3⟩a, *TypeInv* ! *mem_log_t* DEF *FlushLog*
  ⟨2⟩7. *TypeInv* ! *state_t'*
    BY ⟨1⟩2, ⟨1⟩4, ⟨1⟩limit DEF *State*, *LeaderState*, *SetState*
  ⟨2⟩8. $proposing' \in [Proc \rightarrow LogEntry \cup \{NonLogEntry\}]$
    BY ⟨1⟩2, ⟨1⟩4 DEF *StopProposing*
  ⟨2⟩ DEFINE $v \triangleq \langle m[4][1], OpMax(d)\rangle reply \triangleq \langle$"ElectPrepareOK", $d, s, v\rangle$
  ⟨2⟩9. *TypeInv* ! *msg_t'*
    ⟨3⟩1. $reply \in$ *Message*
      ⟨4⟩a. $v \in$ *GeneralMessage*
        BY ⟨1⟩limit, *SeenThruSafety*, ⟨2⟩4
      ⟨4⟩ QED
        BY ⟨1⟩limit, ⟨2⟩4, ⟨4⟩a
    ⟨3⟩2. *CompleteRPC*$(reply, m)$
      BY ⟨1⟩2, ⟨1⟩4 DEF *CompleteRPC*
    ⟨3⟩ QED
      BY ⟨1⟩2, ⟨1⟩a, ⟨1⟩b, ⟨1⟩c, ⟨1⟩4, ⟨3⟩1, ⟨3⟩2, *MessageTypeSafety*3 DEF *CompleteRPC*

$\langle 2 \rangle$unchange. UNCHANGED *memLog* $\wedge$ UNCHANGED $\langle next, accepts \rangle$
  BY $\langle 1 \rangle 2$, $\langle 1 \rangle 4$
$\langle 2 \rangle$other. *OtherSafety*
  $\langle 3 \rangle$a. $rvn[d] \prec Sender(m)$
    $\langle 4 \rangle$a. $Sender(m) = m[4][1]$
      BY $\langle 1 \rangle$c, $\langle 1 \rangle 2$ DEF *Sender*
    $\langle 4 \rangle$b. $rvn[d] \prec m[4][1]$
      BY $\langle 1 \rangle 4$, $\langle 4 \rangle$a
    $\langle 4 \rangle$ QED
      BY $\langle 4 \rangle$a, $\langle 4 \rangle$b
  $\langle 3 \rangle$ QED
    BY $\langle 1 \rangle 2$, $\langle 1 \rangle 4$ DEF *OtherSafety*, *SetRVN*
$\langle 2 \rangle$ QED
  BY $\langle 1 \rangle 2$, $\langle 1 \rangle 4$, $\langle 2 \rangle$unchange, $\langle 2 \rangle 5$, $\langle 2 \rangle 6$, $\langle 2 \rangle 7$, $\langle 2 \rangle 8$, $\langle 2 \rangle 9$, $\langle 2 \rangle$other DEF *var_log*
$\langle 1 \rangle$ QED
  BY $\langle 1 \rangle 3$, $\langle 1 \rangle 4$

THEOREM $ActionSafety12 \triangleq$
  ASSUME *TypeInv*,
    $\exists\, m \in$ DOMAIN *messages* : $\wedge\ messages[m] > 0$
      $\wedge\ m[1] =$ "ElectPrepareOK"
      $\wedge\ ProcessElectPrepareOK(m)$
      $\wedge\ Discard(m)$
  PROVE $TypeInv' \wedge OtherSafety$
$\langle 1 \rangle$ USE DEF *Proc*,
    ViewNum, *NonViewNum*,
    Viewstamp, *NonViewstamp*,
    LogEntry, *NonLogEntry*,
    NonRequest,
    helper
    TypeInv, *ProcessElectPrepareOK*
$\langle 1 \rangle 2$. PICK $m \in$ DOMAIN *messages* :
    $messages[m] > 0 \wedge m[1] =$ "ElectPrepareOK" $\wedge$
    $ProcessElectPrepareOK(m) \wedge Discard(m)$
  OBVIOUS
$\langle 1 \rangle$ DEFINE $s \triangleq m[2] d \triangleq m[3]$
$\langle 1 \rangle$a. $m \in$ DOMAIN *messages*
  BY $\langle 1 \rangle 2$
$\langle 1 \rangle$c. $m \in Message$
  BY $\langle 1 \rangle$a
$\langle 1 \rangle$limit. $s \in Proc \wedge d \in Proc$
  BY $\langle 1 \rangle$c DEF *GeneralMessage*, *CatchUpReply*, *Message*
$\langle 1 \rangle 3$. $messages' \in [Message \rightarrow Nat]$
  $\langle 2 \rangle 1$. $Discard(m)$
    BY $\langle 1 \rangle 2$

```
⟨2⟩ QED
  BY ⟨2⟩1, MessageTypeSafety2 DEF Discard
⟨1⟩4.CASE ProcessElectPrepareOK(m)! : !enabled
  ⟨2⟩chg. TypeInv!state_t' ∧ TypeInv!accepts_t'
    ⟨3⟩a.CASE Majority(Cardinality(accepts[d]) + 1)
      ⟨4⟩a. TypeInv!state_t'
        BY ⟨1⟩limit, ⟨1⟩2, ⟨1⟩4, ⟨3⟩a DEF SetState, LeaderState, State
      ⟨4⟩b. TypeInv!accepts_t'
        BY ⟨1⟩limit, ⟨1⟩2, ⟨1⟩4, ⟨3⟩a DEF SetAcceptSet
      ⟨4⟩ QED
        BY ⟨4⟩a, ⟨4⟩b
    ⟨3⟩b.CASE ¬Majority(Cardinality(accepts[d]) + 1)
      ⟨4⟩a. UNCHANGED state
        BY ⟨1⟩2, ⟨1⟩4, ⟨3⟩b
      ⟨4⟩b. TypeInv!accepts_t'
        ⟨5⟩a. AddAccept(d, m)
          BY ⟨1⟩2, ⟨1⟩4, ⟨3⟩b
        ⟨5⟩ QED
          BY ⟨1⟩limit, ⟨5⟩a DEF AddAccept
      ⟨4⟩ QED
        BY ⟨1⟩limit, ⟨4⟩a, ⟨4⟩b
    ⟨3⟩ QED
      BY ⟨3⟩a, ⟨3⟩b
  ⟨2⟩unchg. UNCHANGED var_log ∧ UNCHANGED ⟨rvn, next, proposing⟩
    BY ⟨1⟩2, ⟨1⟩4
  ⟨2⟩ QED
    BY ⟨2⟩chg, ⟨2⟩unchg, ⟨1⟩3 DEF var_log, OtherSafety
⟨1⟩5.CASE ¬ProcessElectPrepareOK(m)! : !enabled
  ⟨2⟩a. UNCHANGED allstate
    BY ⟨1⟩2, ⟨1⟩5
  ⟨2⟩ QED
    BY ⟨2⟩a, ⟨1⟩3 DEF allstate, OtherSafety
⟨1⟩ QED
  BY ⟨1⟩4, ⟨1⟩5

THEOREM ActionSafety13 ≜
  ASSUME TypeInv,
         (∃ m ∈ DOMAIN messages :
               messages[m] > 0 ∧ m[1] = "ElectSync" ∧ ProcessElectSync(m))
  PROVE TypeInv' ∧ OtherSafety
⟨1⟩ USE DEF Proc,
            ViewNum, NonViewNum,
            Viewstamp, NonViewstamp,
            LogEntry, NonLogEntry,
            NonRequest,
```

helper

$\quad$ *TypeInv*, *ProcessElectSync*

$\langle 1\rangle 2.$ PICK $m \in$ DOMAIN $messages$:

$\quad messages[m] > 0 \land m[1] =$ "ElectSync" $\land ProcessElectSync(m)$

OBVIOUS

$\langle 1\rangle$ DEFINE $s \triangleq m[2] d \triangleq m[3]$

$\langle 1\rangle$b. $messages[m] > 0$

BY $\langle 1\rangle 2$

$\langle 1\rangle$c. $m \in Message$

BY $\langle 1\rangle 2$

$\langle 1\rangle$limit. $s \in Proc \land d \in Proc$

BY $\langle 1\rangle$c DEF *GeneralMessage*, *CatchUpReply*, *Message*

$\langle 1\rangle 3.$CASE $\neg ProcessElectSync(m)! : !enabled$

$\langle 2\rangle 0.$ UNCHANGED $allstate \land Discard(m)$

BY $\langle 1\rangle 2$, $\langle 1\rangle 3$

$\langle 2\rangle 4.$ $TypeInv!msg_t'$

BY $\langle 2\rangle 0$, $\langle 1\rangle 2$, $\langle 1\rangle$b, $\langle 1\rangle$c, *MessageTypeSafety*2 DEF *Discard*

$\langle 2\rangle$ QED

BY $\langle 2\rangle 0$, $\langle 2\rangle 4$ DEF *allstate*, *OtherSafety*

$\langle 1\rangle 4.$CASE $ProcessElectSync(m)! : !enabled$

$\langle 2\rangle 7.$ $diskLog' \in [Proc \rightarrow Seq(LogEntry)]$

$\langle 3\rangle$a. $FlushLog(d)$

BY $\langle 1\rangle 2$, $\langle 1\rangle 4$

$\langle 3\rangle$ QED

BY $\langle 1\rangle$limit, $\langle 3\rangle$a DEF *FlushLog*

$\langle 2\rangle 9.$ $TypeInv!msg_t'$

$\langle 3\rangle$ USE DEF *GeneralMessage*, *CatchUpReply*, *Message*

$\langle 3\rangle$ DEFINE $reply \triangleq \langle$ "ElectSyncOK", $d, s, m[4][1]\rangle$

$\langle 3\rangle 1.$ $reply \in Message$

$\langle 4\rangle 1.$ $m[4][1] \in ViewNum$

BY $\langle 1\rangle$limit, $\langle 1\rangle$c, $\langle 1\rangle 2$

$\langle 4\rangle$ QED

BY $\langle 4\rangle 1$, $\langle 1\rangle$limit

$\langle 3\rangle 2.$ $CompleteRPC(reply, m)$

BY $\langle 1\rangle 2$, $\langle 1\rangle 4$ DEF *CompleteRPC*

$\langle 3\rangle$ QED

BY $\langle 1\rangle 2$, $\langle 1\rangle$b, $\langle 1\rangle$c, $\langle 1\rangle 4$, $\langle 3\rangle 1$, $\langle 3\rangle 2$, *MessageTypeSafety*3 DEF *CompleteRPC*

$\langle 2\rangle$unchange. UNCHANGED $var_leader \land$ UNCHANGED $\langle memLog, rvn, state\rangle$

BY $\langle 1\rangle 2$, $\langle 1\rangle 4$

$\langle 2\rangle$other. $rvn[d] = Sender(m)$

$\langle 4\rangle$a. $Sender(m) = m[4][1]$

BY $\langle 1\rangle$c, $\langle 1\rangle 2$ DEF *Sender*

$\langle 4\rangle$b. $rvn[d] = m[4][1]$

BY $\langle 1\rangle 4$, $\langle 4\rangle$a

$\langle 4\rangle$ QED

```
      BY ⟨4⟩a, ⟨4⟩b
  ⟨2⟩ QED
    BY ⟨1⟩2, ⟨1⟩4, ⟨2⟩unchange, ⟨2⟩7, ⟨2⟩9 DEF var_leader, OtherSafety
⟨1⟩ QED
  BY ⟨1⟩3, ⟨1⟩4

THEOREM ActionSafety14 ≜
  ASSUME TypeInv,
         ∃ m ∈ DOMAIN messages : ∧ messages[m] > 0
                                 ∧ m[1] = "ElectSyncOK"
                                 ∧ ProcessElectSyncOK(m)
                                 ∧ Discard(m)
  PROVE TypeInv′ ∧ OtherSafety
⟨1⟩ USE DEF Proc,
            ViewNum, NonViewNum,
            Viewstamp, NonViewstamp,
            LogEntry, NonLogEntry,
            NonRequest,
            helper
            TypeInv, ProcessElectSyncOK
⟨1⟩2. PICK m ∈ DOMAIN messages :
            messages[m] > 0 ∧ m[1] = "ElectSyncOK" ∧
            ProcessElectSyncOK(m) ∧ Discard(m)
  OBVIOUS
⟨1⟩ DEFINE s ≜ m[2]d ≜ m[3]
⟨1⟩c. m ∈ Message
  BY ⟨1⟩2
⟨1⟩limit. s ∈ Proc ∧ d ∈ Proc
  BY ⟨1⟩2, ⟨1⟩c DEF GeneralMessage, CatchUpReply, Message
⟨1⟩3. TypeInv!msg_t′
  BY ONLY TypeInv!msg_t, ⟨1⟩2, ⟨1⟩c, MessageTypeSafety2 DEF Discard
⟨1⟩4.CASE ProcessElectSyncOK(m)! : !enabled
  ⟨2⟩a.CASE Majority(Cardinality(accepts[d]) + 1)
    ⟨3⟩a. state′ ∈ [Proc → State]
      BY ⟨1⟩limit, ⟨1⟩2, ⟨1⟩4, ⟨2⟩a DEF SetState, LeaderState, State
    ⟨3⟩b. TypeInv!accepts_t′
      BY ⟨1⟩limit, ⟨1⟩2, ⟨1⟩4, ⟨2⟩a DEF SetAcceptSet
    ⟨3⟩c. proposing′ ∈ [Proc → LogEntry ∪ {NonLogEntry}]
      BY ⟨1⟩limit, ⟨1⟩2, ⟨1⟩4, ⟨2⟩a DEF SetProposing
    ⟨3⟩d. next′ ∈ [Proc → Nat]
      ⟨4⟩a. SetNextOpnum(d, LastOpnum(d) + 2)
        BY ⟨1⟩2, ⟨1⟩4, ⟨2⟩a
      ⟨4⟩b. LastOpnum(d) ∈ Nat
        BY ⟨1⟩limit DEF LastViewstamp, OpMax, LastOpnum
      ⟨4⟩ QED
```

BY $\langle 1\rangle$limit, $\langle 4\rangle$a, $\langle 4\rangle$b DEF $SetNextOpnum$

$\langle 3\rangle$e. UNCHANGED $var_log \land$ UNCHANGED rvn

BY $\langle 1\rangle$limit, $\langle 1\rangle 2$, $\langle 1\rangle 4$, $\langle 2\rangle$a

$\langle 3\rangle$ QED

BY $\langle 3\rangle$a, $\langle 3\rangle$b, $\langle 3\rangle$c, $\langle 3\rangle$d, $\langle 3\rangle$e, $\langle 1\rangle 3$ DEF var_log, $OtherSafety$

$\langle 2\rangle$b.CASE $\neg Majority(Cardinality(accepts[d]) + 1)$

$\langle 3\rangle$a. $TypeInv\,!\,accepts_t'$

$\langle 4\rangle$a. $AddAccept(d, m)$

BY $\langle 1\rangle 2$, $\langle 1\rangle 4$, $\langle 2\rangle$b

$\langle 4\rangle$ QED

BY $\langle 1\rangle$limit, $\langle 4\rangle$a DEF $AddAccept$

$\langle 3\rangle$b. UNCHANGED $var_log \land$ UNCHANGED $var_elect \land$ UNCHANGED $\langle proposing, next\rangle$

BY $\langle 1\rangle 2$, $\langle 1\rangle 4$, $\langle 2\rangle$b

$\langle 3\rangle$ QED

BY $\langle 3\rangle$a, $\langle 3\rangle$b, $\langle 1\rangle 3$ DEF var_log, var_elect, $OtherSafety$

$\langle 2\rangle$ QED

BY $\langle 2\rangle$a, $\langle 2\rangle$b

$\langle 1\rangle 5$.CASE $\neg ProcessElectSyncOK(m)!:!enabled$

$\langle 2\rangle$a. UNCHANGED $allstate$

BY $\langle 1\rangle 2$, $\langle 1\rangle 5$

$\langle 2\rangle$ QED

BY $\langle 2\rangle$a, $\langle 1\rangle 3$ DEF $allstate$, $OtherSafety$

$\langle 1\rangle$ QED

BY $\langle 1\rangle 4$, $\langle 1\rangle 5$

THEOREM $ActionSafety15 \triangleq$

ASSUME $TypeInv$,

$(\exists\, m \in$ DOMAIN $messages$:

$messages[m] > 0 \land m[1] =$ "Prepare" $\land ProcessPrepare(m))$

PROVE $TypeInv' \land OtherSafety$

$\langle 1\rangle$ USE DEF $Proc$,

$ViewNum$, $NonViewNum$,

$Viewstamp$, $NonViewstamp$,

$LogEntry$, $NonLogEntry$,

$NonRequest$,

helper

$TypeInv$, $ProcessPrepare$

$\langle 1\rangle 2$. PICK $m \in$ DOMAIN $messages$:

$messages[m] > 0 \land m[1] =$ "Prepare" $\land ProcessPrepare(m)$

OBVIOUS

$\langle 1\rangle$ DEFINE $s \triangleq m[2] d \triangleq m[3]$

$\langle 1\rangle$a. $m \in$ DOMAIN $messages$

BY $\langle 1\rangle 2$

$\langle 1\rangle$b. $messages[m] > 0$

BY $\langle 1\rangle 2$

$\langle 1 \rangle$c. $m \in \mathit{Message}$
BY $\langle 1 \rangle$a
$\langle 1 \rangle$d. $m[4] \in \mathit{LogEntry}$
BY $\langle 1 \rangle$c, $\langle 1 \rangle$2 DEF *GeneralMessage*, *CatchUpReply*, *Message*
$\langle 1 \rangle$limit. $s \in \mathit{Proc} \land d \in \mathit{Proc}$
BY $\langle 1 \rangle$c DEF *GeneralMessage*, *CatchUpReply*, *Message*
$\langle 1 \rangle$e. $m[4][1] \in \mathit{Viewstamp}$
BY $\langle 1 \rangle$d
$\langle 1 \rangle$3.CASE $\mathit{ProcessPrepare}(m)! : !\mathit{accept}$ accept
$\langle 2 \rangle$6. $\mathit{TypeInv}!\mathit{mem_log_t}'$
$\langle 3 \rangle$a. $\mathit{memLog}' = [\mathit{memLog} \text{ EXCEPT } ![d] = \mathit{Append}(\mathit{memLog}[d], m[4])]$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$3
$\langle 3 \rangle$b. $\mathit{Append}(\mathit{memLog}[d], m[4]) \in \mathit{Seq}(\mathit{LogEntry})$
$\langle 4 \rangle$a. $\mathit{memLog}[d] \in \mathit{Seq}(\mathit{LogEntry})$
BY $\mathit{TypeInv}!\mathit{mem_log_t}$, $\langle 1 \rangle$limit
$\langle 4 \rangle$ QED
BY $\langle 4 \rangle$a, $\langle 1 \rangle$limit, $\langle 1 \rangle$d
$\langle 3 \rangle$ QED
BY $\langle 1 \rangle$limit, $\langle 1 \rangle$d, $\langle 3 \rangle$a, $\langle 3 \rangle$b
$\langle 2 \rangle$7. $\mathit{TypeInv}!\mathit{disk_log_t}'$
$\langle 3 \rangle$a.CASE $m[4][3] = \mathit{NonRequest}$
$\langle 4 \rangle$a. $\mathit{FlushLog}(d)$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$3, $\langle 3 \rangle$a
$\langle 4 \rangle$ QED
BY $\langle 1 \rangle$limit, $\langle 4 \rangle$a DEF *FlushLog*
$\langle 3 \rangle$b.CASE $m[4][3] \neq \mathit{NonRequest}$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$3, $\langle 3 \rangle$a
$\langle 3 \rangle$ QED
BY $\langle 3 \rangle$a, $\langle 3 \rangle$b
$\langle 2 \rangle$9. $\mathit{TypeInv}!\mathit{msg_t}'$
$\langle 3 \rangle$ USE DEF *GeneralMessage*, *CatchUpReply*, *Message*
$\langle 3 \rangle$ DEFINE $\mathit{reply} \triangleq \langle \text{"PrepareOK"}, d, s, \langle m[4][1], \mathit{LastViewstamp}(\mathit{diskLog}[d]) \rangle \rangle$
$\langle 3 \rangle$1. $\mathit{reply} \in \mathit{Message}$
BY $\langle 1 \rangle$e, $\langle 1 \rangle$limit, *LastLogIDTypeSafety*
$\langle 3 \rangle$2. $\mathit{CompleteRPC}(\mathit{reply}, m)$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$3 DEF *CompleteRPC*
$\langle 3 \rangle$ QED
BY $\langle 1 \rangle$2, $\langle 1 \rangle$a, $\langle 1 \rangle$b, $\langle 1 \rangle$c, $\langle 1 \rangle$3, $\langle 3 \rangle$1, $\langle 3 \rangle$2, *MessageTypeSafety*3 DEF *CompleteRPC*
$\langle 2 \rangle$unchg. UNCHANGED $\mathit{var_leader} \land$ UNCHANGED $\mathit{var_elect}$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$3
$\langle 2 \rangle$other. $\mathit{rvn}[d] = \mathit{Sender}(m)$
$\langle 3 \rangle$a. $\mathit{Sender}(m) = m[4][1][1]$
BY $\langle 1 \rangle$c, $\langle 1 \rangle$2 DEF *Sender*
$\langle 3 \rangle$b. $\mathit{rvn}[d] = m[4][1][1]$
BY $\langle 1 \rangle$3, $\langle 3 \rangle$a

$\langle 3 \rangle$ QED
BY $\langle 3 \rangle$a, $\langle 3 \rangle$b
$\langle 2 \rangle$ QED
BY $\langle 2 \rangle$unchg, $\langle 2 \rangle$6, $\langle 2 \rangle$7, $\langle 2 \rangle$9 DEF var_leader, var_elect, $OtherSafety$
$\langle 1 \rangle$4.CASE $\neg ProcessPrepare(m)!:!accept \wedge ProcessPrepare(m)!:!learn$ learn
$\langle 2 \rangle$0. UNCHANGED $var_log \wedge$ UNCHANGED $var_leader \wedge Discard(m)$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$4
$\langle 2 \rangle$a. $TypeInv!leader_t'$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$4, $\langle 1 \rangle$limit, $\langle 1 \rangle$e DEF $SetRVN$
$\langle 2 \rangle$b. $TypeInv!msg_t'$
BY $\langle 2 \rangle$0, $\langle 1 \rangle$a, $\langle 1 \rangle$b, $\langle 1 \rangle$c, $MessageTypeSafety2$ DEF $Discard$
$\langle 2 \rangle$c. $TypeInv!state_t'$
BY $\langle 1 \rangle$4, $\langle 1 \rangle$2 DEF $State$, $SetState$
$\langle 2 \rangle$other. $OtherSafety$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$4, $\langle 1 \rangle$limit DEF $OtherSafety$, $SetRVN$
$\langle 2 \rangle$other2. $rvn[d] \prec Sender(m)$
$\langle 3 \rangle$a. $Sender(m) = m[4][1][1]$
BY $\langle 1 \rangle$c, $\langle 1 \rangle$2 DEF $Sender$
$\langle 3 \rangle$b. $rvn[d] \prec m[4][1][1]$
BY $\langle 1 \rangle$4, $\langle 3 \rangle$a
$\langle 3 \rangle$ QED
BY $\langle 3 \rangle$a, $\langle 3 \rangle$b
$\langle 2 \rangle$ QED
BY $\langle 2 \rangle$0, $\langle 2 \rangle$a, $\langle 2 \rangle$b, $\langle 2 \rangle$c, $\langle 1 \rangle$4, $\langle 2 \rangle$other DEF var_log, var_leader
$\langle 1 \rangle$5.CASE $\neg ProcessPrepare(m)!:!accept \wedge \neg ProcessPrepare(m)!:!learn$ ignore
$\langle 2 \rangle$a. UNCHANGED $allstate \wedge Discard(m)$
BY $\langle 1 \rangle$2, $\langle 1 \rangle$5
$\langle 2 \rangle$b. $messages' \in [Message \rightarrow Nat]$
BY $\langle 2 \rangle$a, $\langle 1 \rangle$a, $\langle 1 \rangle$b, $\langle 1 \rangle$c, $MessageTypeSafety2$ DEF $Discard$
$\langle 2 \rangle$ QED
BY $\langle 2 \rangle$a, $\langle 2 \rangle$b DEF $allstate$, $OtherSafety$
$\langle 1 \rangle$ QED
BY $\langle 1 \rangle$3, $\langle 1 \rangle$4, $\langle 1 \rangle$5

THEOREM $ActionSafety16 \triangleq$
ASSUME $TypeInv$,
$\exists\, m \in$ DOMAIN $messages : \wedge\, messages[m] > 0$
$\wedge\, m[1] =$ "PrepareOK"
$\wedge\, ProcessPrepareOK(m)$
$\wedge\, Discard(m)$
PROVE $TypeInv' \wedge OtherSafety$
$\langle 1 \rangle$ USE DEF $Proc$,
$ViewNum$, $NonViewNum$,
$Viewstamp$, $NonViewstamp$,
$LogEntry$, $NonLogEntry$,

$NonRequest$,
helper
$TypeInv$, $ProcessPrepareOK$
⟨1⟩2. PICK $m \in$ DOMAIN $messages$:
$messages[m] > 0 \land m[1] =$ "PrepareOK" $\land ProcessPrepareOK(m) \land$
$Discard(m)$
OBVIOUS
⟨1⟩ DEFINE $s \triangleq m[2] d \triangleq m[3]$
⟨1⟩c. $m \in Message$
BY ⟨1⟩2
⟨1⟩limit. $s \in Proc \land d \in Proc$
BY ⟨1⟩c DEF $GeneralMessage$, $CatchUpReply$, $Message$
⟨1⟩3. $TypeInv!msg_t'$
⟨2⟩1. $Discard(m)$
BY ⟨1⟩2
⟨2⟩ QED
BY ⟨2⟩1, $MessageTypeSafety2$ DEF $Discard$
⟨1⟩4.CASE $ProcessPrepareOK(m)! : !enabled$
⟨2⟩accept. $TypeInv!accepts_t'$
⟨3⟩a. $AddAccept(d, m)$
BY ⟨1⟩2, ⟨1⟩4
⟨3⟩ QED
BY ⟨3⟩a, ⟨1⟩limit DEF $AddAccept$
⟨2⟩unchg. UNCHANGED $var_log \land$ UNCHANGED $var_elect \land$ UNCHANGED $next$
BY ⟨1⟩2, ⟨1⟩4
⟨2⟩a.CASE $Majority(Cardinality(accepts[d]) + 1)$
⟨3⟩c. $TypeInv!proposing_t'$
BY ⟨1⟩limit, ⟨1⟩2, ⟨1⟩4, ⟨2⟩a DEF $StopProposing$
⟨3⟩ QED
BY ⟨3⟩c, ⟨1⟩3, ⟨2⟩accept, ⟨2⟩unchg DEF var_log, var_elect, $OtherSafety$
⟨2⟩b.CASE $\neg Majority(Cardinality(accepts[d]) + 1)$
⟨3⟩b. UNCHANGED $proposing$
BY ⟨1⟩2, ⟨1⟩4, ⟨2⟩b
⟨3⟩ QED
BY ⟨3⟩b, ⟨1⟩3, ⟨2⟩accept, ⟨2⟩unchg DEF var_log, var_elect, $OtherSafety$
⟨2⟩ QED
BY ⟨2⟩a, ⟨2⟩b
⟨1⟩5.CASE $\neg ProcessPrepareOK(m)! : !enabled$
⟨2⟩a. UNCHANGED $allstate$
BY ⟨1⟩2, ⟨1⟩5
⟨2⟩ QED
BY ⟨2⟩a, ⟨1⟩3 DEF $allstate$, $OtherSafety$
⟨1⟩ QED
BY ⟨1⟩4, ⟨1⟩5

THEOREM $BuildCatchUpReplyTypeSafety \triangleq$

ASSUME NEW $d \in Proc$, $memLog \in [Proc \rightarrow Seq(LogEntry)]$,
$rvn \in [Proc \rightarrow ViewNum \cup \{NonViewNum\}]$,
NEW $lastop \in Viewstamp \cup \{NonViewstamp\}$,
$rvn[d] \neq NonViewNum$

PROVE $BuildCatchUpReply(d, lastop) \in CatchUpReply$

$\langle 1 \rangle$ USE DEF $BuildCatchUpReply$, $CatchUpReply$, $Viewstamp$, $LogEntry$, $ViewNum$, $NonLogEntry$, $NonViewstamp$

$\langle 1 \rangle$ DEFINE $vst \triangleq BuildCatchUpReply(d, lastop)! : !vst$

$\langle 1 \rangle$a. $vst \in Viewstamp \cup \{NonViewstamp\}$

BY DEF $LowerBound$, $ViewstampLess$, $ViewstampLessEqual$

$\langle 1 \rangle$b. $NextEntry(memLog[d], vst) \in LogEntry \cup \{NonLogEntry\}$

BY DEF $NextEntry$, $ViewstampLessEqual$

$\langle 1 \rangle$ QED

BY $\langle 1 \rangle$a, $\langle 1 \rangle$b

THEOREM $ActionSafety17 \triangleq$

ASSUME $TypeInv$,
$(\exists\, m \in \text{DOMAIN } messages :$
$messages[m] > 0 \land m[1] = \text{"CatchUp"} \land ProcessCatchUp(m))$

PROVE $TypeInv' \land OtherSafety$

$\langle 1 \rangle$ USE DEF $Proc$,
$ViewNum$, $NonViewNum$,
$Viewstamp$, $NonViewstamp$,
$LogEntry$, $NonLogEntry$,
$NonRequest$,
helper
$TypeInv$, $ProcessCatchUp$

$\langle 1 \rangle$2. PICK $m \in \text{DOMAIN } messages :$
$messages[m] > 0 \land m[1] = \text{"CatchUp"} \land ProcessCatchUp(m)$

OBVIOUS

$\langle 1 \rangle$ DEFINE $s \triangleq m[2] d \triangleq m[3]$

$\langle 1 \rangle$b. $messages[m] > 0$

BY $\langle 1 \rangle$2

$\langle 1 \rangle$c. $m \in Message$

BY $\langle 1 \rangle$2

$\langle 1 \rangle$d. $m[4] \in GeneralMessage$

BY $\langle 1 \rangle$c, $\langle 1 \rangle$2 DEF $GeneralMessage$, $CatchUpReply$, $Message$

$\langle 1 \rangle$limit. $s \in Proc \land d \in Proc$

BY $\langle 1 \rangle$c DEF $GeneralMessage$, $CatchUpReply$, $Message$

$\langle 1 \rangle$e. $m[4][2] \in Viewstamp \cup \{NonViewstamp\}$

BY $\langle 1 \rangle$d, $\langle 1 \rangle$2 DEF $GeneralMessage$, $CatchUpReply$, $Message$

$\langle 1 \rangle$3.CASE $ProcessCatchUp(m)! : !enabled$ accept

$\langle 2 \rangle$9. $TypeInv!msg_t'$

$\langle 3 \rangle$ USE DEF $GeneralMessage$, $CatchUpReply$, $Message$

$\langle 3 \rangle$ DEFINE $r \triangleq BuildCatchUpReply(d, m[4][2])$
$\langle 3 \rangle$ DEFINE $reply \triangleq \langle$"CatchUpReply", $d, s, r\rangle$
$\langle 3 \rangle 1.$ $reply \in Message$
$\langle 4 \rangle$a. $r \in CatchUpReply$
BY $TypeInv!mem_log_t$, $TypeInv!leader_t$, $\langle 1 \rangle$e, $\langle 1 \rangle 3$,
$\langle 1 \rangle$limit, $BuildCatchUpReplyTypeSafety$
$\langle 4 \rangle$ QED
BY $\langle 4 \rangle$a, $\langle 1 \rangle$limit
$\langle 3 \rangle 2.$ $CompleteRPC(reply, m)$
BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$ DEF $CompleteRPC$
$\langle 3 \rangle$ QED
BY $\langle 1 \rangle 2$, $\langle 1 \rangle$c, $\langle 1 \rangle$b, $\langle 1 \rangle$c, $\langle 1 \rangle 3$, $\langle 3 \rangle 1$, $\langle 3 \rangle 2$, $MessageTypeSafety3$ DEF $CompleteRPC$
$\langle 2 \rangle$unchg. UNCHANGED $allstate$
BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$
$\langle 2 \rangle$ QED
BY $\langle 2 \rangle$unchg, $\langle 2 \rangle 9$ DEF $allstate$, $OtherSafety$
$\langle 1 \rangle 4.$CASE $\neg ProcessCatchUp(m)!:!enabled$
$\langle 2 \rangle$a. UNCHANGED $allstate \land Discard(m)$
BY $\langle 1 \rangle 2$, $\langle 1 \rangle 4$
$\langle 2 \rangle$b. $TypeInv!msg_t'$
BY $\langle 2 \rangle$a, $\langle 1 \rangle$c, $\langle 1 \rangle$b, $\langle 1 \rangle$c, $MessageTypeSafety2$ DEF $Discard$
$\langle 2 \rangle$ QED
BY $\langle 2 \rangle$a, $\langle 2 \rangle$b DEF $allstate$, $OtherSafety$
$\langle 1 \rangle$ QED
BY $\langle 1 \rangle 3$, $\langle 1 \rangle 4$

THEOREM $AppendTypeSafety \triangleq$
ASSUME NEW LOG, NEW DOM, NEW $logs \in [DOM \rightarrow Seq(LOG)]$, NEW $d \in DOM$,
NEW $e \in LOG$
PROVE $Append(logs[d], e) \in Seq(LOG)$
$\langle 1 \rangle$a. $logs[d] \in Seq(LOG)$
OBVIOUS
$\langle 1 \rangle$ QED
BY $\langle 1 \rangle$a

THEOREM $ActionSafety18 \triangleq$
ASSUME $TypeInv$,
$\exists\, m \in$ DOMAIN $messages$: $\land\; messages[m] > 0$
$\land\; m[1] =$ "CatchUpReply"
$\land\; ProcessCatchUpReply(m)$
$\land\; Discard(m)$
PROVE $TypeInv' \land OtherSafety$
$\langle 1 \rangle$ USE DEF $Proc$,
$ViewNum$, $NonViewNum$,
$Viewstamp$, $NonViewstamp$,

$LogEntry$, $NonLogEntry$,
$NonRequest$,
helper
$TypeInv$, $ProcessCatchUpReply$

$\langle 1 \rangle 2.$ PICK $m \in$ DOMAIN $messages$:
$messages[m] > 0 \land m[1] =$ "CatchUpReply" $\land$
$ProcessCatchUpReply(m) \land Discard(m)$
OBVIOUS

$\langle 1 \rangle$ DEFINE $s \triangleq m[2] d \triangleq m[3]$

$\langle 1 \rangle$b. $messages[m] > 0$
BY $\langle 1 \rangle 2$

$\langle 1 \rangle$c. $m \in Message$
BY $\langle 1 \rangle 2$

$\langle 1 \rangle$limit. $s \in Proc \land d \in Proc$
BY $\langle 1 \rangle$c DEF $GeneralMessage$, $CatchUpReply$, $Message$

$\langle 1 \rangle$d. $m[4][4] \in LogEntry \cup \{NonLogEntry\}$
BY $\langle 1 \rangle$c, $\langle 1 \rangle 2$ DEF $GeneralMessage$, $CatchUpReply$, $Message$

$\langle 1 \rangle$e. $m[4][3] \in Viewstamp \cup \{NonViewstamp\}$
BY $\langle 1 \rangle$c, $\langle 1 \rangle 2$ DEF $GeneralMessage$, $CatchUpReply$, $Message$

$\langle 1 \rangle 3.$ $TypeInv!msg_t'$
$\langle 2 \rangle 1.$ $Discard(m)$
BY $\langle 1 \rangle 2$
$\langle 2 \rangle$ QED
BY $\langle 2 \rangle 1$, $MessageTypeSafety2$ DEF $Discard$

$\langle 1 \rangle 4.$CASE $ProcessCatchUpReply(m)! : !enabled$
$\langle 2 \rangle$unchg. UNCHANGED $var_leader \land$ UNCHANGED var_elect
BY $\langle 1 \rangle 2$, $\langle 1 \rangle 4$
$\langle 2 \rangle$a.CASE $m[4][4] = NonLogEntry$ roll backward
$\langle 3 \rangle$a. $TruncateBothLogsTo(d, m[4][3])$
BY $\langle 1 \rangle 2$, $\langle 1 \rangle 4$, $\langle 2 \rangle$a
$\langle 3 \rangle$b. $TypeInv!mem_log_t'$
$\langle 4 \rangle$a. $memLog' = [memLog$ EXCEPT $![d] = TruncateLog(memLog[d], m[4][3])]$
BY $\langle 3 \rangle$a DEF $TruncateBothLogsTo$
$\langle 4 \rangle$b. $TruncateLog(memLog[d], m[4][3]) \in Seq(LogEntry)$
BY ONLY $TypeInv!mem_log_t$, $\langle 1 \rangle$limit, $\langle 1 \rangle$e, $TruncateLogTypeSafety$
$\langle 4 \rangle$ QED
BY ONLY $\langle 4 \rangle$a, $TypeInv!mem_log_t$, $\langle 1 \rangle$limit, $\langle 4 \rangle$b
$\langle 3 \rangle$c. $TypeInv!disk_log_t'$
$\langle 4 \rangle$a. $diskLog' = [diskLog$ EXCEPT $![d] = TruncateLog(diskLog[d], m[4][3])]$
BY $\langle 3 \rangle$a DEF $TruncateBothLogsTo$
$\langle 4 \rangle$b. $TruncateLog(diskLog[d], m[4][3]) \in Seq(LogEntry)$
BY ONLY $TypeInv!disk_log_t$, $\langle 1 \rangle$limit, $\langle 1 \rangle$e, $TruncateLogTypeSafety$
$\langle 4 \rangle$ QED
BY ONLY $\langle 4 \rangle$a, $TypeInv!disk_log_t$, $\langle 1 \rangle$limit, $\langle 4 \rangle$b
$\langle 3 \rangle$ QED

BY ⟨3⟩b, ⟨3⟩c, ⟨1⟩3, ⟨2⟩unchg DEF *var_leader*, *var_elect*, *OtherSafety*

⟨2⟩b.CASE $m[4][4] \neq NonLogEntry$ roll forward

⟨3⟩0. $m[4][4] \in LogEntry$

BY ⟨1⟩d, ⟨2⟩b

⟨3⟩a. *TypeInv*!*mem_log_t′*

⟨4⟩a. $memLog' = [memLog \text{ EXCEPT } ![d] = Append(memLog[d], m[4][4])]$

BY ⟨1⟩2, ⟨1⟩4, ⟨2⟩b

⟨4⟩b. $Append(memLog[d], m[4][4]) \in Seq(LogEntry)$

BY ONLY *TypeInv*!*mem_log_t*, ⟨1⟩limit, ⟨3⟩0, *AppendTypeSafety*

⟨4⟩ QED

BY ONLY *TypeInv*!*mem_log_t*, ⟨1⟩limit, ⟨4⟩a, ⟨4⟩b

⟨3⟩b. *TypeInv*!*disk_log_t′*

⟨4⟩a. $diskLog' = [diskLog \text{ EXCEPT } ![d] = Append(diskLog[d], m[4][4])]$

BY ⟨1⟩2, ⟨1⟩4, ⟨2⟩b

⟨4⟩b. $Append(diskLog[d], m[4][4]) \in Seq(LogEntry)$

BY ONLY *TypeInv*!*disk_log_t*, ⟨1⟩limit, ⟨3⟩0, *AppendTypeSafety*

⟨4⟩ QED

BY ONLY *TypeInv*!*disk_log_t*, ⟨1⟩limit, ⟨4⟩a, ⟨4⟩b

⟨3⟩ QED

BY ⟨3⟩a, ⟨3⟩b, ⟨1⟩3, ⟨2⟩unchg DEF *var_leader*, *var_elect*, *OtherSafety*

⟨2⟩ QED

BY ⟨2⟩a, ⟨2⟩b

⟨1⟩5.CASE $\neg ProcessCatchUpReply(m)! : !enabled$

⟨2⟩a. UNCHANGED *allstate*

BY ⟨1⟩2, ⟨1⟩5

⟨2⟩ QED

BY ⟨2⟩a, ⟨1⟩3 DEF *allstate*, *OtherSafety*

⟨1⟩ QED

BY ⟨1⟩4, ⟨1⟩5

THEOREM *InitSafety* $\triangleq$

ASSUME *Init*1PROVE *TypeInv*

⟨1⟩ USE DEF *Proc*,

ViewNum, *NonViewNum*,

Viewstamp, *NonViewstamp*,

LogEntry, *NonLogEntry*,

NonRequest,

helper

TypeInv, *GeneralMessage*, *CatchUpReply*, *Message*,

*Init*1

⟨1⟩1. *TypeInv*!*leader_t*

OBVIOUS

⟨1⟩2. *TypeInv*!*mem_log_t*

OBVIOUS

⟨1⟩3. *TypeInv*!*disk_log_t*

```
  OBVIOUS
⟨1⟩5. TypeInv!next_opnum_t
  OBVIOUS
⟨1⟩6. TypeInv!state_t
  BY DEF State, LeaderState
⟨1⟩7. TypeInv!accepts_t
  OBVIOUS
⟨1⟩8. TypeInv!proposing_t
  OBVIOUS
⟨1⟩9. TypeInv!msg_t
  ⟨2⟩1. messages = [m ∈ Message ↦ 0]
    BY Init1
  ⟨2⟩ QED
    BY ⟨2⟩1
⟨1⟩ QED
  BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩5, ⟨1⟩6, ⟨1⟩7, ⟨1⟩8, ⟨1⟩9

THEOREM Safety ≜
  ASSUME TypeInv, OneStep
  PROVE TypeInv' ∧ OtherSafety
⟨1⟩ USE DEF Proc,
            ViewNum, NonViewNum,
            Viewstamp, NonViewstamp,
            LogEntry, NonLogEntry,
            NonRequest,
             helper
            GeneralMessage, CatchUpReply, Message,
            OneStep, Receive
⟨1⟩ QED
  BY ActionSafety1, ActionSafety2, ActionSafety3, ActionSafety4,
     ActionSafety5, ActionSafety6, ActionSafety7, ActionSafety8, ActionSafety9,
     ActionSafety10, ActionSafety11, ActionSafety12, ActionSafety13, ActionSafety14,
     ActionSafety15, ActionSafety16, ActionSafety17, ActionSafety18


test_viewnum_compare ≜
        LET v0 ≜ ⟨1, 1⟩
            v1 ≜ ⟨1, 2⟩
            v2 ≜ ⟨2, 1⟩
            v3 ≜ ⟨2, 2⟩
        IN  ∧ ¬(NonViewNum ≺ NonViewNum)
            ∧ NonViewNum ≺ v0
            ∧ ¬(v0 ≺ NonViewNum)
            ∧ ¬(v0 ≺ v0) ∧ v0 ≺ v1 ∧ v0 ≺ v2 ∧ v0 ≺ v3
            ∧ ¬(v1 ≺ v0) ∧ ¬(v1 ≺ v1) ∧ v1 ≺ v2 ∧ v1 ≺ v3
```

$$\land \neg(v2 \prec v0) \land \neg(v2 \prec v1) \land \neg(v2 \prec v2) \land v2 \prec v3$$
$$\land \neg(v3 \prec v0) \land \neg(v3 \prec v1) \land \neg(v3 \prec v2) \land \neg(v3 \prec v3)$$

$$
\begin{array}{l}
\textit{test_constant} \triangleq \\
\quad \land \textit{NonLogEntry} = \textit{NonLogEntry} \\
\quad \land \textit{NonViewNum} = \textit{NonViewNum} \\
\quad \land \textit{NonViewstamp} = \textit{NonViewstamp}
\end{array}
$$

$$
\begin{array}{l}
\textit{test_message} \triangleq \\
\quad \land \textit{Cardinality}(\text{DOMAIN } [m \in \{\} \mapsto 0]) = 0 \\
\quad \land \text{LET } am \triangleq \textit{WithMessage}(\text{"m0"}, [m \in \{\} \mapsto 0]) \\
\quad\quad \text{IN } \textit{Cardinality}(\text{DOMAIN } am) = 1 \land am[\text{"m0"}] = 1 \\
\quad \land \text{LET } am \triangleq \textit{WithMessage}(\text{"m0"}, [m \in \{\text{"m0"}\} \mapsto 1]) \\
\quad\quad \text{IN } \textit{Cardinality}(\text{DOMAIN } am) = 1 \land am[\text{"m0"}] = 2 \\
\quad \land \text{LET } am \triangleq \textit{WithMessage}(\text{"m1"}, [m \in \{\text{"m0"}\} \mapsto 0]) \\
\quad\quad \text{IN } \textit{Cardinality}(\text{DOMAIN } am) = 2 \land am[\text{"m0"}] = 0 \land am[\text{"m1"}] = 1 \\
\quad \land \text{LET } am \triangleq \textit{WithoutMessage}(\text{"m0"}, [m \in \{\text{"m0"}\} \mapsto 1]) \\
\quad\quad \text{IN } \textit{Cardinality}(\text{DOMAIN } am) = 1 \land am[\text{"m0"}] = 0
\end{array}
$$

$$
\begin{array}{l}
\textit{test_last_viewstamp} \triangleq \\
\quad \text{LET } v0 \triangleq \langle 1, 1 \rangle \\
\quad\quad\quad vst1 \triangleq \langle v0, 1 \rangle \\
\quad\quad\quad vst2 \triangleq \langle v0, 2 \rangle \\
\quad\quad\quad vst3 \triangleq \langle v0, 3 \rangle \\
\quad\quad\quad req \triangleq \text{CHOOSE } v \in \textit{ClientRequests} : \text{TRUE} \\
\quad \text{IN } \land \textit{LastViewstamp}(\langle\rangle) = \textit{NonViewstamp} \\
\quad\quad\quad \land \textit{LastViewstamp}(\langle\langle vst2, vst1, req \rangle\rangle) = \langle v0, 2 \rangle \\
\quad\quad\quad \land \textit{LastViewstamp}(\langle\langle vst2, vst1, \textit{NonRequest} \rangle\rangle) = \langle v0, 2 \rangle \\
\quad\quad\quad \land \textit{LastViewstamp}(\langle\langle vst2, vst1, \textit{NonRequest} \rangle, \langle vst3, vst2, req \rangle\rangle) = \langle v0, 3 \rangle
\end{array}
$$

$$
\begin{array}{l}
\textit{test_lastop_vnum} \triangleq \\
\quad \text{LET } v0 \triangleq \langle 1, 1 \rangle \\
\quad\quad\quad vst1 \triangleq \langle v0, 1 \rangle \\
\quad\quad\quad vst2 \triangleq \langle v0, 2 \rangle \\
\quad\quad\quad vst3 \triangleq \langle v0, 3 \rangle \\
\quad\quad\quad req \triangleq \text{CHOOSE } v \in \textit{ClientRequests} : \text{TRUE} \\
\quad \text{IN } \land \textit{LastLogEntryVNum}(\langle\rangle) = \textit{NonViewNum} \\
\quad\quad\quad \land \textit{LastLogEntryVNum}(\langle\langle vst2, vst1, req \rangle\rangle) = v0 \\
\quad\quad\quad \land \textit{LastLogEntryVNum}(\langle\langle vst2, vst1, \textit{NonRequest} \rangle\rangle) = v0 \\
\quad\quad\quad \land \textit{LastLogEntryVNum}(\langle\langle vst2, vst1, \textit{NonRequest} \rangle, \langle vst3, vst2, req \rangle\rangle) = v0
\end{array}
$$

$$
\begin{array}{l}
\textit{test_next_ballot} \triangleq \\
\quad \text{LET } v0 \triangleq \langle 1, 1 \rangle \\
\quad\quad\quad v1 \triangleq \langle 2, 1 \rangle
\end{array}
$$

```
IN  ∧ NextViewNum(NonViewNum, 1) = v0
    ∧ NextViewNum(v0, 1) = v1
    ∧ NextViewNum(v0, 2) = ⟨2, 2⟩
    ∧ NextViewNum(v1, 2) = ⟨3, 2⟩

test_viewstamp_leq ≜
    LET v0 ≜ ⟨1, 1⟩
        vst1 ≜ ⟨v0, 1⟩
        vst2 ≜ ⟨v0, 2⟩
    IN  ∧ ViewstampLessEqual(NonViewstamp, NonViewstamp)
        ∧ ViewstampLessEqual(NonViewstamp, vst1)
        ∧ ViewstampLessEqual(vst1, vst1)
        ∧ ViewstampLessEqual(vst1, vst2)
        ∧ ViewstampLessEqual(vst2, vst2)

test_lowerbound ≜
    LET v0 ≜ ⟨1, 1⟩
        vst1 ≜ ⟨v0, 1⟩
        vst2 ≜ ⟨v0, 2⟩
        vst3 ≜ ⟨v0, 3⟩
        vst4 ≜ ⟨v0, 4⟩
        op2  ≜ ⟨vst2, vst1, NonRequest⟩
        op3  ≜ ⟨vst3, vst2, NonRequest⟩
        op4 ≜ ⟨vst4, vst3, NonRequest⟩
    IN  ∧ LowerBound(⟨⟩, NonViewstamp) = NonViewstamp
        ∧ LowerBound(⟨⟩, vst1) = NonViewstamp
        ∧ LowerBound(⟨op2⟩, NonViewstamp) = NonViewstamp
        ∧ LowerBound(⟨op2⟩, vst1) = NonViewstamp
        ∧ LowerBound(⟨op2⟩, vst2) = vst2
        ∧ LowerBound(⟨op2⟩, vst3) = vst2
        ∧ LowerBound(⟨op2, op3⟩, NonViewstamp) = NonViewstamp
        ∧ LowerBound(⟨op2, op3⟩, vst1) = NonViewstamp
        ∧ LowerBound(⟨op2, op3⟩, vst2) = vst2
        ∧ LowerBound(⟨op2, op3⟩, vst4) = vst3
        ∧ LowerBound(⟨op2, op3, op4⟩, NonViewstamp) = NonViewstamp
        ∧ LowerBound(⟨op2, op3, op4⟩, vst1) = NonViewstamp
        ∧ LowerBound(⟨op2, op3, op4⟩, vst2) = vst2
        ∧ LowerBound(⟨op2, op3, op4⟩, vst3) = vst3

test_next_entry ≜
    LET v0 ≜ ⟨1, 1⟩
        vst1 ≜ ⟨v0, 1⟩
        vst2 ≜ ⟨v0, 2⟩
        vst3 ≜ ⟨v0, 3⟩
```

$vst4 \triangleq \langle v0, 4\rangle$
$op2 \triangleq \langle vst2, vst1, NonRequest\rangle$
$op3 \triangleq \langle vst3, vst2, NonRequest\rangle$
$op4 \triangleq \langle vst4, vst3, NonRequest\rangle$

IN
$\land NextEntry(\langle op2\rangle, NonViewstamp) = op2$
$\land NextEntry(\langle op2\rangle, vst1) = op2$
$\land NextEntry(\langle op2\rangle, vst2) = NonLogEntry$
$\land NextEntry(\langle op2\rangle, vst3) = NonLogEntry$
$\land NextEntry(\langle op2, op3\rangle, NonViewstamp) = op2$
$\land NextEntry(\langle op2, op3\rangle, vst1) = op2$
$\land NextEntry(\langle op2, op3\rangle, vst2) = op3$
$\land NextEntry(\langle op2, op3\rangle, vst3) = NonLogEntry$
$\land NextEntry(\langle op2, op3, op4\rangle, vst1) = op2$
$\land NextEntry(\langle op2, op3, op4\rangle, vst2) = op3$
$\land NextEntry(\langle op2, op3, op4\rangle, vst3) = op4$

$test_truncate_log \triangleq$

LET $v0 \triangleq \langle 1, 1\rangle$
$vst1 \triangleq \langle v0, 1\rangle$
$vst2 \triangleq \langle v0, 2\rangle$
$vst3 \triangleq \langle v0, 3\rangle$
$vst4 \triangleq \langle v0, 4\rangle$
$op2 \triangleq \langle vst2, vst1, NonRequest\rangle$
$op3 \triangleq \langle vst3, vst2, NonRequest\rangle$

IN
$\land TruncateLog(\langle op2\rangle, NonViewstamp) = \langle\rangle$
$\land TruncateLog(\langle op2\rangle, vst2) = \langle op2\rangle$
$\land TruncateLog(\langle op2\rangle, vst3) = \langle op2\rangle$
$\land TruncateLog(\langle op2, op3\rangle, NonViewstamp) = \langle\rangle$
$\land TruncateLog(\langle op2, op3\rangle, vst2) = \langle op2\rangle$
$\land TruncateLog(\langle op2, op3\rangle, vst3) = \langle op2, op3\rangle$
$\land TruncateLog(\langle op2, op3\rangle, vst4) = \langle op2, op3\rangle$

$unit_test \triangleq$
$\land test_message$
$\land test_constant$
$\land test_last_viewstamp$
$\land test_lastop_vnum$
$\land test_viewnum_compare$
$\land test_next_ballot$
$\land test_lowerbound$
$\land test_next_entry$
$\land test_truncate_log$

$test_init1 \triangleq$ Follower state

LET $v0 \triangleq \langle 1, 1\rangle vst1 \triangleq \langle v0, 1\rangle$ IN

$\land\ rvn = [p \in Proc \mapsto v0]$
$\land\ memLog = [p \in Proc \mapsto \langle\langle vst1,\ NonViewstamp,\ NonRequest\rangle\rangle]$
$\land\ diskLog = [p \in Proc \mapsto \langle\langle vst1,\ NonViewstamp,\ NonRequest\rangle\rangle]$
Leader state
$\land\ next = [p \in Proc \mapsto \text{IF } p = 1 \text{ THEN } 1 \text{ ELSE } 0]$
$\land\ state = [p \in Proc \mapsto \text{IF } p = 1 \text{ THEN }$ "Leader_Decided" ELSE "Backup"$]$
$\land\ accepts = [p \in Proc \mapsto \{\}]$
$\land\ proposing = [p \in Proc \mapsto NonLogEntry]$
messages
$\land\ messages = [m \in \{\} \mapsto 0]$

$test_init2 \triangleq$ Follower state
LET $v0 \triangleq \langle 1,\ 1\rangle vst1 \triangleq \langle v0,\ 1\rangle$ IN
$\land\ rvn = [p \in Proc \mapsto v0]$
$\land\ memLog = [p \in Proc \mapsto \text{IF } p \neq 1 \text{ THEN } \langle\langle vst1,\ NonViewstamp,\ NonRequest\rangle\rangle \text{ ELSE } \langle\rangle]$
$\land\ diskLog = [p \in Proc \mapsto \text{IF } p \neq 1 \text{ THEN } \langle\langle vst1,\ NonViewstamp,\ NonRequest\rangle\rangle \text{ ELSE } \langle\rangle]$
Leader state
$\land\ next = [p \in Proc \mapsto \text{IF } p = 1 \text{ THEN } 1 \text{ ELSE } 0]$
$\land\ state = [p \in Proc \mapsto \text{IF } p = 1 \text{ THEN }$ "Leader_Decided" ELSE "Backup"$]$
$\land\ accepts = [p \in Proc \mapsto \{\}]$
$\land\ proposing = [p \in Proc \mapsto NonLogEntry]$
messages
$\land\ messages = [m \in \{\} \mapsto 0]$

$test_next1 \triangleq OneStep$

The following are helpers for manual proof. They are defined less formally
* in the following ways to improve readability :
* – $ViewnumLess(a,\ b)$ is written as $a < b$.
* – $ViewstampLess(a,\ b)$ is written as $a < b$
* – $ViewstampLessEqual(a,\ b)$ is written as $a \leq b$
* – For $m \in Message$, $m.type \triangleq m[1]$, $m.src \triangleq m[2]$, $m.dst \triangleq m[3]$.
* – For $ElectPrepare$ message, $m.nextvn \triangleq m[4][1]$, $m.opmax \triangleq m[4][2]$
* – For $id \in Viewstamp$: $id.vn \triangleq id[1]$
* – For $PrepareOK$ message, $m.logid \triangleq m[4][1]$, $m.diskopmax \triangleq m[4][2]$
* – For $Prepare$ message, $m.logentry \triangleq m[4]$

The vst precedes vst.
Obtained by mapping vst to $LogEntry$, and returns $LogEntry.prev$, i.e. $LogEntry[2]$
The formal definition of $Prev(vst)$ is ignored because it is not relavant
to the implementation
$Prev(vst) \triangleq vst$

RECURSIVE $Succeed(_,\ _)$
$Succeed(vst_next,\ vst_prev) \triangleq$
$vst_next = vst_prev \lor (vst_prev < vst_next \land Succeed(Prev(vst_next),\ vst_prev))$

Check if view identified by $(vnum,\ opmax)$ is bad to dp

$BadView(dp, vnum, opmax) \triangleq$

$\quad \land \; dp.vn < vnum$

$\quad \land \; \lor (opmax > dp \land \neg Succeed(opmax, dp))$

$\quad \quad \lor \; opmax < dp$

$SentMessage \triangleq Message$ message has ever been sent till now

$HasNotVoted(dp, M) \triangleq$

$\quad \forall m \in SentMessage :$

senario: any *ElectPrepare* message targeted anyone in M

$\land \; m.type = \text{"ElectPrepare"} \land m.dst \in M$

If the sender of the *ElectPrepare* is not well intended,

$\land \; BadView(dp, m.nextvn, m.opmax)$

then m hasn't vote for it yet!

The formal definition should be $m[3]$ has never sent a corresponding *ElectPrepareOK*.

This requires the DOMAIN of messages to be SUBSET *messages*.

However, TLA+ doesn't seem to be good at reasoning about SUBSET .

So we ignore the formal definition here.

$\Rightarrow$ TRUE

$MinViewstamp(S) \triangleq$ CHOOSE $v \in S : \forall x \in S : x \leq v$

THEOREM $LocalFlushedThruSafety \triangleq$

ASSUME NEW $m \in SentMessage$,

$\quad messages[m] > 0$,

$\quad m[1] = \text{"PrepareOK"}$

$\quad$ *m.logid.vn*: the view number of the sender of the corresponding *Prepare* message

$\quad$ *m.lastlogid*: the view number of the last on-disk *log* entry of $m[2]$ when it sent m

PROVE $m.logid.vn = m.diskopmax$

THEOREM $DurablePointSafety \triangleq$

ASSUME NEW $m \in SentMessage$, $m.type = \text{"Prepare"}$,

$\quad ProcessPrepare(m)! : !accept$, $ProcessPrepare(m)$,

$\quad$ NEW $M \in$ SUBSET $Proc$, NEW $dp \in Viewstamp$,

$\quad$ LET $MajorityAccept \triangleq \exists reply \in SentMessage :$

$\quad \quad \land \forall r \in reply : r.type = \text{"PrepareOK"} \land r.logid = m.logentry.logid$

$\quad \quad \land M = \{r \in reply : r.src\} \land Majority(Cardinality(M))$

$\quad \quad \land dp = MinViewstamp(\{r \in reply : r.logid\})$

$\quad$ IN $\Diamond \langle MajorityAccept \rangle_{allvars}$

PROVE $HasNotVoted(dp, M) \land \Box[HasNotVoted(dp, M)]_{allvars}$

\ * Modification History

\ * Last modified *Thu Jul* 10 16:47:27 *EDT* 2014 by *ydmao*

\ * Created *Thu Oct* 03 15:58:11 *EDT* 2013 by *ydmao*

Bibliography

[1] ACID. http://en.wikipedia.org/wiki/ACID.

[2] add proper support for SIGSTOP and SIGCONT (currently, on replica set primary can cause data loss). https://jira.mongodb.org/browse/SERVER-10768.

[3] Filebench - file system benchmark. http://sourceforge.net/projects/filebench/.

[4] leveldb - A fast and lightweight key/value database library by Google. https://code.google.com/p/leveldb/.

[5] MemcacheDB. http://memcachedb.org.

[6] MongoDB. http://mongodb.com.

[7] mongodb/mongo. https://github.com/mongodb/mongo/blob/master/src/mongo/db/repl/rs_rollback.cpp.

[8] Network Block Device (TCP version). http://nbd.sourceforge.net/.

[9] Protocol Buffers. http://code.google.com/p/protobuf/.

[10] Re: Questions on block drivers, REQ_FLUSH and REQ_FUA. http://www.spinics.net/lists/linux-fsdevel/msg45604.html.

[11] Redis. http://redis.io.

[12] Redis Replication. http://redis.io/topics/replication.

[13] Replica Set Oplog. http://docs.mongodb.org/manual/core/replica-set-oplog.

[14] The TLA Home Page. http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html.

[15] TLA+ Proof System. http://tla.msr-inria.inria.fr/tlaps/content/Home.html.

[16] Two primaries for the same replica set. https://jira.mongodb.org/browse/SERVER-8145.

[17] VoltDB, the NewSQL database for high velocity applications. `http://voltdb.com`.

[18] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D Davis. CORFU: A Shared Log Design for Flash Clusters. In *NSDI*, pages 1–14, 2012.

[19] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proc. 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141.

[20] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977.

[21] Kenneth P Birman. *Reliable distributed systems*. Springer, 2005.

[22] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. Main-memory index structures with fixed-size partial keys. *SIGMOD Record*, 30:163–174, May 2001.

[23] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, pages 11–11, 2011.

[24] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM PPoPP Symposium*, Bangalore, India, 2010.

[25] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proc. 27th VLDB Conference*, 2001.

[26] Sang K. Cha and Changbin Song. P*TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload. In *Proc. 30th VLDB Conference*, pages 1033–1044, 2004.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26:4:1–4:26, June 2008.

[28] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *Proc. 2001 SIGMOD Conference*, pages 235–246.

[29] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.

[30] John Cieslewicz and Kenneth A. Ross. Data partitioning on chip multiprocessors. In *Proc. 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 25–34, New York, NY, USA, 2008.

[31] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. Automatic contention detection and amelioration for data-intensive operations. In *Proc. 2010 SIGMOD Conference*, pages 483–494.

[32] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[33] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010.

[34] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proc. 10th Usenix Symposium on Operating System Design and Implementation*, volume 1, 2012.

[35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[36] Ongaro Diego, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[37] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge Computer Laboratory, 2004.

[38] Edward Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, September 1960.

[39] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28, 2011.

[40] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju G. Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, Califormnia, USA, January 2007.

[41] Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. 17th VLDB Conference*, pages 525–535, 1991.

[42] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[43] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *Proc. 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35, New York, NY, USA, 2009.

[44] Flavio Paiva Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[45] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endowment*, 1:1496–1499, August 2008.

[46] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[47] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[48] Leslie Lamport. Paxos made simple. *Sigact News*, 32(4):18–25, 2001.

[49] Edward K Lee and Chandramohan A Thekkath. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices*, volume 31, pages 84–92. ACM, 1996.

[50] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[51] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.

[52] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014.

[53] Wei Lin, Mao Yang, Lintao Zhang, and Lidong Zhou. PacificA: Replication in log-based distributed storage systems. Technical report, Citeseer, 2008.

[54] Barbara Liskov and James Cowling. Viewstamped replication revisited. *MIT Technical Report*, 2012.

[55] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the Harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 226–238. ACM, 1991.

[56] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Proc. 2002 Ottawa Linux Symposium*, pages 338–367, 2002.

[57] James Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, and Osama Khan. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 257–273, Seattle, WA, 2014. USENIX.

[58] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 1–15, 2012.

[59] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–627, 1995.

[60] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *Draft of October*, 7, 2013.

[61] Jun Rao and Kenneth A. Ross. Making B+-trees cache conscious in main memory. *SIGMOD Record*, 29:475–486, May 2000.

[62] Jun Rao, Eugene J Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4(4):243–254, 2011.

[63] Kenneth A. Ross. Optimizing read convoys in main-memory query processing. In *Proc. 6th International Workshop on Data Management on New Hardware*, DaMoN '10, pages 27–33, New York, NY, USA, 2010. ACM.

[64] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proc. 5th International Symposium on Memory Management*, ISMM '06, pages 84–94. ACM, 2006.

[65] Siddhartha Sen and Robert E. Tarjan. Deletion without rebalancing in balanced binary trees. In *Proc. 21st SODA*, pages 1490–1499, 2010.

[66] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endowment*, 4(11):795–806, August 2011.

[67] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, Petros Maniatis, et al. Zeno: Eventually Consistent Byzantine-Fault Tolerance. In *NSDI*, volume 9, pages 169–184, 2009.

[68] Michael Stonebraker, Samuel Madden, J. Daniel Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. 33rd VLDB Conference*, pages 1150–1160, 2007.