Sinfonia: A New Paradigm for Building Scalable Distributed Systems

Marcos K. Aguilera* Arif Merchant* Mehul Shah* Alistair Veitch* Christos Karamanolis†
*HP Laboratories, Palo Alto, CA, USA †VMware, Palo Alto, CA, USA

ABSTRACT

We propose a new paradigm for building scalable distributed systems. Our approach does not require dealing with message-passing protocols—a major complication in existing distributed systems. Instead, developers just design and manipulate data structures within our service called Sinfonia. Sinfonia keeps data for applications on a set of memory nodes, each exporting a linear address space. At the core of Sinfonia is a novel minitransaction primitive that enables efficient and consistent access to data, while hiding the complexities that arise from concurrency and failures. Using Sinfonia, we implemented two very different and complex applications in a few months: a cluster file system and a group communication service. Our implementations perform well and scale to hundreds of machines.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed systems—*Distributed applications*; E.1 [**Data Structures**]: Distributed data structures

General Terms

Algorithms, Design, Experimentation, Performance, Reliability

Keywords

Distributed systems, scalability, fault tolerance, shared memory, transactions, two-phase commit

1. INTRODUCTION

Developers often build distributed systems using the message-passing paradigm, in which processes share data by passing messages over the network. This paradigm is error-prone and hard to use because it involves designing, implementing, and debugging complex protocols for handling distributed state. Distributed state refers to data that application hosts need to manipulate and share with one another, such as metadata, tables, and configuration and status information. Protocols for handling distributed state include protocols for replication, file data and metadata management, cache consistency, and group membership. These protocols are highly non-trivial to develop.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

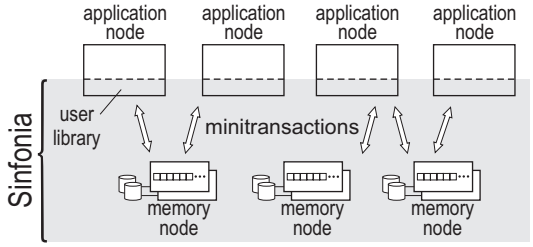

Figure 1: Sinfonia allows application nodes to share data in a fault tolerant, scalable, and consistent manner.

We propose a new paradigm for building scalable distributed systems. With our scheme, developers do not have to deal with message-passing protocols. Instead, developers just design and manipulate data structures within our service, called Sinfonia. We therefore transform the problem of protocol design into the much easier problem of data structure design. Our approach targets particularly data center *infrastructure applications*, such as cluster file systems, lock managers, and group communication services. These applications must be fault-tolerant and scalable, and must provide consistency and reasonable performance.

In a nutshell, Sinfonia is a service that allows hosts to share application data in a fault-tolerant, scalable, and consistent manner. Existing services that allow hosts to share data include database systems and distributed shared memory (DSM). Database systems lack the performance needed for infrastructure applications, where efficiency is vital. This is because database systems provide more functionality than needed, resulting in performance overheads. For instance, attempts to build file systems using a database system [24] resulted in an unusable system due to poor performance. Existing DSM systems lack the scalability or fault tolerance required for infrastructure applications. Section 8 discusses some of the DSM systems closest to Sinfonia.

Sinfonia seeks to provide a balance between functionality and scalability. The key to achieving scalability is to decouple operations executed by different hosts as much as possible, so that operations can proceed independently. Towards this goal, Sinfonia provides fine-grained address spaces on which to store data, without imposing any structure, such as types, schemas, tuples, or tables, which all tend to increase coupling. Thus, application hosts can handle data in Sinfonia relatively independently of each other. To prevent Sinfonia from becoming a bottleneck, Sinfonia itself is distributed over multiple memory nodes (Figure 1), whose number determines the space and bandwidth capacity of Sinfonia.

At the core of Sinfonia is a lightweight *minitransaction* primitive that applications use to atomically access and conditionally modify

data at multiple memory nodes. For example, consider a cluster file system, one of the applications we built with Sinfonia. With a minitransaction, a host can atomically populate an inode stored in one memory node and link this inode to a directory entry stored in another memory node, and these updates can be conditional on the inode being free (to avoid races). Like database transactions, minitransactions hide the complexities that arise from concurrent execution and failures.

Minitransactions are also useful for improving performance, in many ways. First, minitransactions allow users to batch together updates, which eliminates multiple network round-trips. Second, because of their limited scope, minitransactions can be executed within the commit protocol. In fact, Sinfonia can start, execute, and commit a minitransaction with two network round-trips. In contrast, database transactions, being more powerful and higher-level, require two round-trips just to commit, plus additional round-trips to start and execute. Third, minitransactions can execute in parallel with a replication scheme, providing availability with little extra latency.

We demonstrate Sinfonia by using it to build two complex applications that are very different from each other: a cluster file system called SinfoniaFS and a group communication service called SinfoniaGCS. These applications are known to be difficult to implement in a scalable and fault-tolerant fashion: systems achieving these goals tend to be very complicated and are the result of years of effort. Using Sinfonia, we built them with 3900 and 3500 lines of code, in one and two man-months, respectively. In SinfoniaFS, Sinfonia holds file system data, and each node in the cluster uses minitransactions to atomically retrieve and update file data and attributes, and allocate and deallocate space. In SinfoniaGCS, Sinfonia stores ordered messages broadcast by users, and users use minitransactions to add new messages to the ordering.

Experiments show that Sinfonia and its applications scale well and perform competitively. Sinfonia can execute thousands of minitransactions per second at a reasonable latency when running over a single node, and the throughput scales well with system size. SinfoniaFS over a single memory node performs as well as an NFS server and, unlike an NFS server, SinfoniaFS scales well to hundreds of nodes. SinfoniaGCS scales better than Spread [1], a high-throughput implementation of a group communication service.

2. ASSUMPTIONS AND GOALS

We consider distributed systems within a data center. A data center is a site with many fairly well-connected machines. It can have from tens to thousands of machines running from tens to hundreds of applications. Network latencies are small and have little variance most of the time, unlike in a completely asynchronous environment. Network partitions may occur within a data center, but this is rare; while there is one, it is acceptable to pause applications since, most likely, the data center is unusable. Applications in the data center and their designers are trustworthy, rather than malicious. (Access control is an orthogonal concern that could be incorporated in Sinfonia, but we have not done so.) Note that these assumptions do not hold in wide area networks, peer-to-peer systems, or the Internet as a whole.

The data center is subject to failures: a node may crash sometimes and, more rarely, all nodes may crash (e.g., due to power outages), and failures may occur at unpredictable times. Individual machines are reliable, but crashes are common because there are many machines. We do not consider Byzantine failures. Disks provide *stable storage*, that is, disks provide sufficient reliability for the target application. This may require choosing disks carefully; choices vary from low-cost disks to high-end disk arrays. Stable storage may crash, and one needs to deal with it, but such crashes are relatively rare.

Our goal is to help developers build distributed *infrastructure applications*, which are applications that support other applications. Examples include lock managers, cluster file systems, group communication services, and distributed name services. These applications need to provide reliability, consistency, and scalability. Scalability is the ability to increase system capacity proportionately to system size. In this paper, capacity refers to processing capacity, which is measured by total throughput.

3. DESIGN

We now describe Sinfonia and its principles and design.

3.1 Principles

The design of Sinfonia is based on two principles:

Principle 1. Reduce operation coupling to obtain scalability. Coupling refers to the interdependence that operations have on each other, which limits parallel execution on different hosts, and hence hinders scalability. Sinfonia avoids operation coupling by not imposing structure on the data it services.

Principle 2. Make components reliable before scaling them. We first make individual Sinfonia nodes fault-tolerant, and only then scale the system by adding more nodes. Thus, we avoid the complexities of a large system with many unreliable components.

3.2 Basic components

Sinfonia consists of a set of memory nodes and a user library that runs at application nodes (Figure 1). Memory nodes hold application data, either in RAM or on stable storage, according to application needs. The user library implements mechanisms to manipulate data at memory nodes. It is possible to place a memory node and an application node in the same host, but they are logically distinct.

Each memory node keeps a sequence of raw or uninterpreted words of some standard length; in this paper, word length is 1 byte. These bytes are organized as a *linear address space* without any structure. Each memory node has a separate address space, so that data in Sinfonia is referenced through a pair (*memory-node-id*, *address*). We also tried a design with a single global address space that is transparently mapped to memory nodes, but this design did not scale because of the lack of *node locality*. Node locality refers to placing data that is accessed together in the same node. For example, our cluster file system makes an effort to place an inode, its chaining list, and its file data in the same memory node (if space permits). This would be difficult with a transparent mapping of a single address space. Node locality is the opposite of *data striping*, which spreads data accessed together over many nodes. Data striping improves single-user throughput, but our experiments show that it impairs scalability.

Application nodes access data in Sinfonia through a user library. The user library provides basic read/write of a byte range on a single memory node, as well as *minitransactions*, described next.

3.3 Minitransactions

Minitransactions allow an application to update data in multiple memory nodes while ensuring atomicity, consistency, isolation, and (if wanted) durability. Atomicity means that a minitransaction executes completely or not at all; consistency means that data is not corrupted; isolation means that minitransactions are serializable; and durability means that committed minitransactions are not lost, even if there are failures.

We tuned the power of minitransactions so that they can execute efficiently while still being quite useful. To explain how we did that, we first describe how standard distributed transactions execute

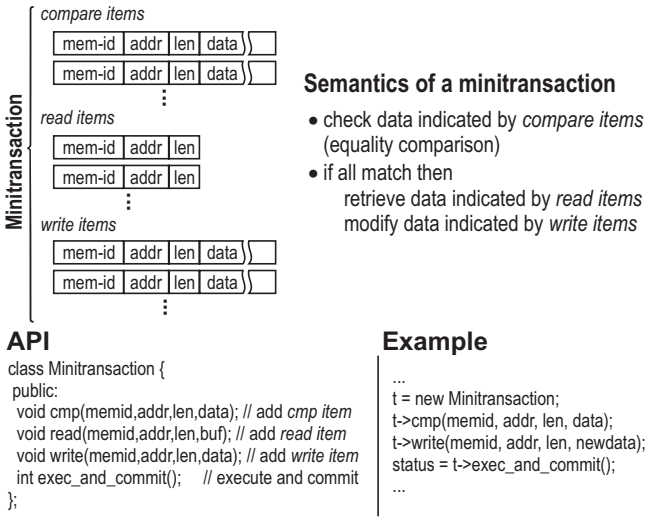

Figure 2: Minitransactions have compare items, read items, and write items. Compare items are locations to compare against given values, while read items are locations to read and write items are locations to update, if all comparisons match. All items are specified before the minitransaction starts executing. The example code creates a minitransaction with one compare and one write item on the same location—a compare-and-swap operation. Methods cmp, read, *and* write *populate a minitransaction without communication with memory nodes until* exec_and_commit *is invoked.*

and commit. Roughly speaking, a coordinator executes a transaction by asking participants to perform one or more transaction *actions*, such as retrieving or modifying data items. At the end of the transaction, the coordinator executes two-phase commit. In the first phase, the coordinator asks all participants if they are ready to commit. If they all vote yes, in the second phase the coordinator tells them to commit; otherwise the coordinator tells them to abort. In Sinfonia, coordinators are application nodes and participants are memory nodes.

We observe that it is possible to optimize the execution of some transactions, as follows. If the transaction's last action does not affect the coordinator's decision to abort or commit then the coordinator can piggyback this last action onto the first phase of two-phase commit (e.g., this is the case if this action is a data update). This optimization does not affect the transaction semantics and saves a communication round-trip.

Even if the transaction's last action affects the coordinator's decision to abort or commit, if the participant knows how the coordinator makes this decision, then we can also piggyback the action onto the commit protocol. For example, if the last action is a read and the participant knows that the coordinator will abort if the read returns zero (and will commit otherwise), then the coordinator can piggyback this action onto two-phase commit and the participant can read the item and adjust its vote to abort if the result is zero.

In fact, it might be possible to piggyback the entire transaction execution onto the commit protocol. We designed minitransactions so that this is always the case and found that it is still possible to get fairly powerful transactions.

More precisely, a minitransaction (Figure 2) consists of a set of *compare items*, a set of *read items*, and a set of *write items*. Each item specifies a memory node and an address range within that memory node; compare and write items also include data. Items are chosen before the minitransaction starts executing. Upon execution, a minitransaction does the following: (1) compare the locations in the compare items, if any, against the data in the compare items (equality comparison), (2) if all comparisons succeed, or if there are no compare items, return the locations in the read items and write to the locations in the write items, and (3) if some comparison fails, abort. Thus, the compare items control whether the minitransaction commits or aborts, while the read and write items determine what data the minitransaction returns and updates.

Minitransactions are a powerful primitive for handling distributed data. Examples of minitransactions include the following:

1. *Swap*. A read item returns the old value and a write item replaces it.
2. *Compare-and-swap*. A compare item compares the current value against a constant; if equal, a write item replaces it.
3. *Atomic read of many data*. Done with multiple read items.
4. *Acquire a lease*. A compare item checks if a location is set to 0; if so, a write item sets it to the (non-zero) *id* of the leaseholder and another write item sets the time of lease.
5. *Acquire multiple leases atomically.* Same as above, except that there are multiple compare items and write items. Note that each lease can be in a different memory node.
6. *Change data if lease is held*. A compare item checks that a lease is held and, if so, write items update data.

A frequent minitransaction idiom is to use compare items to validate data and, if data is valid, use write items to apply some changes to the same or different data. These minitransactions are common in SinfoniaFS: the file system caches inodes and metadata aggressively at application nodes, and relevant cached entries are validated before modifying the file system. For example, writing to a file requires validating a cached copy of the file's inode and chaining list (the list of blocks comprising the file) and, if they are valid, modifying the appropriate file block. This is done with compare items and write items in a minitransaction. Figure 7 shows a minitransaction used by SinfoniaFS to set a file's attributes.

Another minitransaction idiom is to have only compare items to validate data, without read or write items. Such a minitransaction modifies no data, regardless of whether it commits or aborts. But if it commits, the application node knows that all comparisons succeeded and so the validations were successful. SinfoniaFS uses this type of minitransaction to validate cached data for read-only file system operations, such as stat (NFS's getattr).

In Section 4 we explain how minitransactions are executed and committed efficiently. It is worth noting that minitransactions can be extended to include more general read-modify-write items (not just write items) and generic conditional items (not just compare items) provided that each item can be executed at a single memory node. For example, there could be an increment item that atomically increments a location; and a minitransaction could have multiple increment items, possibly at different memory nodes, to increment all of them together. These extensions were not needed for the applications in this paper, but they may be useful for other applications.

3.4 Caching and consistency

Sinfonia does not cache data at application nodes, but provides support for applications to do their own caching. Application-controlled caching has three clear advantages: First, there is greater flexibility on policies of what to cache and what to evict. Second, as a result, cache utilization potentially improves, since applications know their data better than what Sinfonia can infer. And third, Sinfonia becomes a simpler service to use because data accessed through Sinfonia is always current (not stale). Managing caches in

Mode	RAM	RAM-REPL	LOG	LOG-REPL	NVRAM	NVRAM-REPL
Description	disk image off log off replication off backup optional	disk image off log off replication on backup optional	disk image on log on disk replication off backup optional	disk image on log on disk replication on backup optional	disk image on log on nvram replication off backup optional	disk image on log on nvram replication on backup optional
Memnode resources	1 host	2 hosts	1 host, 2 disks[(c)]	2 hosts, 4 disks[(d)]	1 host, 1 disk, nvram	2 hosts, 2 disks, nvram[(d)]
Memnode space	RAM available	RAM available	disk size	disk size	disk size	disk size
Fault tolerance[(a)]	•app crash	•app crash •few memnode crashes	•app crash •all memnode crashes but with downtime	•app crash •few memnode crashes •all memnode crashes but with downtime	•app crash •all memnode crashes but with downtime	•app crash •few memnode crashes •all memnode crashes but with downtime
Performance[(b)]	first	second	third	fourth	first	second

Figure 3: Trading off fault tolerance for amount of resources and performance. Each column is a mode of operation for Sinfonia. Notes: (a) 'App crash' means tolerating crashes of any number of application nodes. 'Few memnode crashes' means tolerating crashes of memory nodes as long as not all replicas crash. 'Downtime' refers to blocking until recovery. (b) For exact performance numbers see Section 7.1.1. (c) Logging on disk uses a disk just for the log, to benefit from sequential writes. (d) If we colocate a memory node with the replica of another memory node, they can share the log disk and the image disk. This halves the total number of disks and hosts needed.

a distributed system can be a challenge for applications, but Sinfonia minitransactions simplify this, by allowing applications to atomically validate cached data and apply updates, as explained in Section 3.3.

3.5 Fault tolerance

Sinfonia provides fault tolerance in accordance with application needs. At the very least, crashes of application nodes never affect data in Sinfonia (minitransactions are atomic). In addition, Sinfonia offers some optional protections:

- *Masking independent failures*. If a few memory nodes crash, Sinfonia masks the failures so that the system continues working with no downtime.
- *Preserving data on correlated failures*. If many memory nodes crash in a short period (e.g., in a power outage) without losing their stable storage, Sinfonia ensures that data is not lost, but the system may be unavailable until enough memory nodes restart.
- *Restoring data on disasters*. If memory nodes and their stable storage crash (e.g., due to a disaster), Sinfonia recovers data using a transactionally-consistent backup.

With all protections enabled, the system remains available if there are few failures, the system loses no data if there are many failures, and the system loses non-backed up data if there are disasters. To provide fault tolerance, Sinfonia uses four mechanisms: *disk images*, *logging*, *replication*, and *backup*. A disk image keeps a copy of the data at a memory node; the space capacity of a memory node is as large as the disk size. For efficiency, the disk image is written asynchronously and so may be slightly out-of-date. To compensate for that, a log keeps recent data updates, and the log is written synchronously to ensure data durability. The log can be stored on disk or non-volatile RAM (NVRAM), if available. When a memory node recovers from a crash, it uses a recovery algorithm to replay the log. This is transparent to applications, but recovery takes time and makes the system unavailable. To provide high availability, Sinfonia replicates memory nodes, so that if a memory node fails, a replica takes over without downtime. Currently, Sinfonia uses primary-copy replication (e.g., [3]), but it is possible to use instead state machines and Paxos [19] to rely less on synchrony. Replicas are synchronized in parallel with minitransaction execution for efficiency. Backups can be made from a transactionally-consistent image of Sinfonia data. This image is generated without pausing applications, by using the log to buffer updates while writes to disk are flushed.

Figure 3 shows various Sinfonia modes of operation based on which of the above mechanisms are used. Modes differ only on their fault tolerance, memory node space, and performance, not on the application interface. Modes RAM and RAM-REPL store memory node data in RAM memory only, while other modes use disk and/or NVRAM.

3.6 Other design considerations

Load balancing. By design, Sinfonia lets applications choose where to place application data, in order to improve node locality. As a corollary, Sinfonia does not balance load across memory nodes—this is left to applications. To assist applications, Sinfonia provides per-memory-node load information to applications, including bytes read and written and minitransactions executed, retried, and aborted in various time frames (e.g., 5s, 1min, 10min, 1h, 12h). Applications can tag each minitransaction with a class identifier, and load information is provided separately for each class. Load balancing sometimes entails migrating data; for that, minitransactions can migrate many pieces of data atomically, without exposing inconsistencies due to partial migration. However, application developers still have to choose and implement a migration policy (when and where to migrate)—which is probably application specific in any case.

Colocation of data and computation. In some applications, it may be important to colocate data and computation for best performance. This is very easy in Sinfonia: one simply places an application node in the same host as a memory node, and the application node biases placement of its data toward its local memory node; Sinfonia informs the application which memory node is local, if any. In this paper, however, we do not colocate application and memory nodes.

4. IMPLEMENTATION AND ALGORITHMS

We now explain how Sinfonia is implemented, including details of our two-phase protocol to execute and commit minitransactions. This protocol differs from standard two-phase commit in several ways: it reflects some different failure assumptions in Sinfonia, it requires new schemes for recovery and garbage collection, and it incorporates minitransaction execution and a technique to avoid minitransaction deadlocks.

4.1 Basic architecture

Recall that Sinfonia comprises a set of memory nodes and a user library at each application node. The user library communicates with memory nodes through remote procedure calls, on top of which we run the minitransaction protocol. Memory nodes run a server process that keeps Sinfonia data and the minitransaction redo-log; it also runs a replication protocol.

4.2 Minitransaction protocol overview

Our minitransaction protocol integrates execution of the minitransaction into the commit protocol for efficiency. The idea is to piggyback the transaction into the first phase of two-phase commit. This piggybacking is not possible for arbitrary transactions, but minitransactions were defined so that it is possible for them.

Our two-phase commit protocol also reflects new system failure assumptions. In standard two-phase commit, if the coordinator crashes, the system has to block until the coordinator recovers. This is undesirable in Sinfonia: if the coordinator crashes, we may need to recover without it because coordinators run on application nodes, not Sinfonia memory nodes, and so they may be unstable, subject to reboots, or their recovery could be unpredictable and unsure. The traditional way to avoid blocking on coordinator crashes is to use three-phase commit [32], but we want to avoid the extra phase.

We accomplish this by blocking on participant crashes instead of coordinator crashes. This is reasonable for Sinfonia because participants are memory nodes that keep application data, so if they go down and the application needs to access data, the application has to block anyway. Furthermore, Sinfonia can optionally replicate participants (memory nodes), so that minitransactions are blocked only if there is a crash of the "logical participant", as represented by all its replicas.

In our two-phase commit protocol, the coordinator has no log, and we consider a transaction to be committed if all participants have a yes vote in their log. Standard two-phase commit requires a yes vote in the coordinator log. This modification, however, complicates the protocols for recovery and log garbage collection, which we cover in Sections 4.4–4.7.

To ensure serializability, participants lock the locations accessed by a minitransaction during phase 1 of the commit protocol. Locks are only held until phase 2 of the protocol, a short time. Lock granularity is a word, but we use range data structures to efficiently keep track of locked ranges. To avoid deadlocks, we use a simple scheme: a participant tries to acquire locks without blocking; if it fails, it releases all locks and votes "abort due to busy lock". This vote causes the coordinator to abort the minitransaction and retry after some random exponentially-increasing delay. This scheme is not appropriate when there is high contention, but otherwise it is efficient. Another deadlock-avoidance scheme is to acquire locks in some predefined order, but with that scheme, the coordinator in phase 1 has to contact participants in series (to ensure lock ordering), which could incur many extra network round-trips.

4.3 Minitransaction protocol details

Recall that a minitransaction has compare items, read items, and write items (Figure 2). Compare items are locations to be tested for equality against supplied data; if any test fails, the minitransaction aborts. If the minitransaction commits, read items are locations to be read and returned, while write items are locations to be written.

Application nodes execute and commit minitransactions using the two-phase protocol in Figure 4. Phase 1 executes and prepares the minitransaction, while phase 2 commits it. More precisely, in phase 1, the coordinator (application node) generates a new transaction id (*tid*) and sends the minitransaction to the participants (memory nodes). Each participant then (a) tries to lock the addresses of its items in the minitransaction (without blocking), (b) executes the comparisons in the compare items and, if all comparisons succeed, performs the reads in the read items and buffers the write items, and (c) decides on a vote as follows: if all locations could be locked and all compare items succeeded, the vote is for committing, otherwise it is for aborting. In phase 2, the coordinator tells participants to commit if and only if all votes are for committing. If committing, a participant applies the write items, otherwise it discards them. In either case, the participant releases all locks acquired by the minitransaction. The coordinator never logs any information, unlike in standard two-phase commit. If the minitransaction aborts because some locks were busy, the coordinator retries the minitransaction after a while using a new *tid*. This retrying is not shown in the code.

Participants log minitransactions in the redo-log in the first phase (if logging is enabled); logging occurs only if the participant votes to commit. Only write items are logged, not compare or read items, to save space. The redo-log in Sinfonia also serves as a write-ahead log to improve performance.

Code for coordinator p:

To execute and commit minitransaction (*cmpitems*, *rditems*, *writems*)
1 *tid* ← new unique identifier for minitransaction
{ Phase 1 }
2 D ← set of memory nodes referred in *cmpitems* ∪ *rditems* ∪ *writems*
3 **pfor** each $q \in D$ **do** { pfor is a parallel for }
4 **send** (EXEC&PREPARE, *tid*, D, $\pi_q(cmpitems), \pi_q(rditems), \pi_q(writems)$) **to** q
5 { π_q denotes the projection to the items handled by q }
6 *replies* ← wait for replies from all nodes in D
{ Phase 2 }
8 **if** $\forall q \in D$: *replies*[q].*vote*=OK **then** *action* ← *true* { commit }
9 **else** *action* ← *false* { abort }
10 **pfor** each $q \in D$ **do send** (COMMIT, *tid*, *action*) **to** q
11 **return** *action* { does not wait for reply of COMMIT }

Code for each participant memory node q:

upon receive (EXEC&PREPARE, *tid*, D, *cmpitems*, *rditems*, *writems*) **from** p **do**
12 *in-doubt* ← *in-doubt* ∪ {(*tid*, *cmpitems*, *rditems*, *writems*)}
13 **if** *try-read-lock*(*cmpitems* ∪ *rditems*)=*fail or try-write-lock*(*writems*)=*fail*
14 **then** *vote* ← BAD-LOCK
15 **else if** *tid* ∈ *forced-abort* **then** *vote* ← BAD-FORCED
16 { *forced-abort* is used with recovery }
17 **else if** *cmpitems* do not match data **then** *vote* ← BAD-CMP
18 **else** *vote* ← OK
19 **if** *vote*=OK **then**
20 *data* ← read *rditems*
21 add (*tid*, D, *writems*) to *redo-log* and add *tid* to *all-log-tids*
22 **else**
23 *data* ← ∅
24 release locks acquired above
25 **send-reply** (*tid*, *vote*, *data*) **to** p

upon receive (COMMIT, *tid*, *action*) **from** p **do** { *action*: *true*=commit, *false*=abort }
26 (*cmpitems*, *rditems*, *writems*) ← find (*tid*, ∗, ∗, ∗) in *in-doubt*
27 **if not** found **then return** { recovery coordinator executed first }
28 *in-doubt* ← *in-doubt* − {(*tid*, *cmpitems*, *rditems*, *writems*)}
29 **if** *tid* ∈ *all-log-tids* **then** *decided* ← *decided* ∪ {(*tid*, *action*)}
30 **if** *action* **then** apply *writems*
31 release any locks still held for *cmpitems* ∪ *rditems* ∪ *writems*

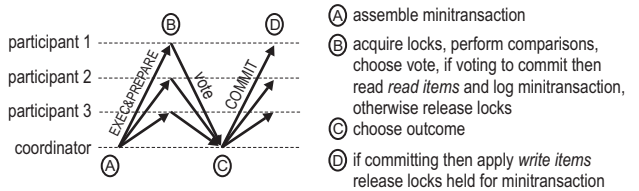

Figure 4: Protocol for executing and committing minitransactions.

Name	Description	On stable storage
redo-log	minitransaction redo-log	yes, sync
in-doubt	*tid*s not yet committed or aborted	no
forced-abort	*tid*s forced to abort (by recovery)	yes, sync
decided	*tid*s in redo-log with outcome	yes, async
all-log-tids	*tid*s in redo-log	no

Figure 5: Data structures kept at participants (memory nodes) for recovery and garbage collection. Async/sync indicates whether writes to stable storage are asynchronous or synchronous.

Participants keep an *in-doubt* list of undecided transaction *tids*, a *forced-abort* list of *tids* that must be voted to abort, and a *decided* list of finished *tids* and their outcome. These data structures are used for recovery, as explained below. Figure 5 summarizes the data structures kept by participants.

4.4 Recovery from coordinator crashes

If a coordinator crashes during a minitransaction, it may leave the minitransaction with an uncertain outcome: one in which not all participants have voted yet. To fix this problem, a recovery mechanism is executed by a third-party, the *recovery coordinator*, which runs at a dedicated management node for Sinfonia. The recovery scheme ensures the following: (a) it will not drive the system into an unrecoverable state if the recovery coordinator crashes or if there are memory node crashes during recovery; (b) it ensures correctness even if there is concurrent execution of recovery with the original coordinator (this might happen if recovery starts but the original coordinator is still running); and (c) it allows concurrent execution by multiple recovery coordinators (this might happen if recovery restarts but a previous recovery coordinator is still running).

To recover a minitransaction *tid*, the recovery coordinator tries to abort *tid* since committing requires knowledge of the minitransaction items, which the recovery coordinator does not have. More precisely, the recovery coordinator proceeds in two phases. In phase 1, it requests participants to vote "abort" on *tid*; each participant checks if it previously voted on this minitransaction, and, if so, it keeps its previous vote. Otherwise, the participant chooses to vote "abort" and places *tid* in the *forced-abort* list, to remember its vote in case the original coordinator later asks it to vote.[1] In either case, the participant returns its vote. In phase 2, the recovery coordinator tells participants to commit if and only if all votes are "commit". Participants then commit or abort accordingly, and release all locks acquired by the minitransaction. This is safe to do, even if the original coordinator is still executing, because either the minitransaction has already been committed, or it is forced to abort.

How does the recovery coordinator get triggered in the first place? The management node periodically probes memory nodes for minitransactions in the *in-doubt* list that have not yet committed after a timeout period, and starts recovery for them.

4.5 Recovery from participant crashes

When a participant memory node crashes, the system blocks any outstanding minitransactions involving the participant until the participant recovers[2]. If the participant loses its stable storage, one must recover the system from a backup (Section 4.9 explains how to produce transactionally-consistent backups). More frequently, the participant recovers without losing its stable storage, in which case it proceeds to synchronize its disk image by replaying its redo-log in order. The participant remains offline until this process completes. To avoid replaying a long redo-log, there is a *processed-pointer* variable that gets periodically written to disk, which indicates what parts of the redo-log are new; replay starts at this place. Not every minitransaction in the redo-log should be replayed, only those that committed. This is determined using the *decided* list, which includes decided minitransactions and their outcome. The *decided* list is written to disk asynchronously, so there may be a few minitransactions that are in the redo-log but not in the *decided* list. For these minitransactions, it is necessary to contact the set D of memory nodes that participated in the minitransaction to know their votes. This set D is stored in the participant's redo-log. Upon being contacted by this procedure, if a memory node has not voted, then it votes no (by placing the minitransaction's *tid* in the *forced-abort* list). This is necessary for correctness when the minitransaction's original coordinator is still running.

[1] Note that a participant serializes incoming requests from a coordinator and a recovery coordinator for the same minitransaction.

[2] This is unlike two-phase commit for database systems, where the coordinator may consider a dead participant as voting no.

4.6 Recovery from crash of the whole system

When many memory nodes or the whole system crashes and restarts, the management node starts a procedure to recover many or all memory nodes at once. Each memory node essentially uses the previously described scheme to recover from its own crash, but employs a simple optimization: memory nodes send each other their votes on recent minitransactions, so that if a memory node has a recent minitransaction in its redo-log but not in its *decided* list, it does not have to contact other memory nodes to determine its outcome.

How does the management node know that many nodes crashed and restarted? We use a simple scheme: when a memory node reboots, it sends a reboot notification to the management node. The management node tries to contact other memory nodes and waits until it gets an alive response or a reboot notification from a large fraction of memory nodes. The latter case indicates that many nodes crashed and restarted.

4.7 Log garbage collection

A memory node applies committed minitransactions to the disk image, so that the redo-log can be garbage collected. The redo-log is garbage collected in log order. If the log head has an aborted minitransaction, then it can be garbage collected immediately, because this minitransaction can never commit. The hard case is garbage collecting committed minitransactions. This is done according to the following rule:

> A committed minitransaction tid can be removed from the redo-log head only when tid has been applied to the disk image of **every** memory node involved in tid.

The reason for having "every" above is that, if some memory node q crashes and recovers, then q may need to see tid at the redo-log of other memory nodes to determine that tid committed. To implement the above rule, a memory node p periodically informs every other memory node q of the minitransactions tids that p recently applied to its disk image and that q participated in.

Besides the redo-log, the other data structures in Figure 5 are garbage collected as follows. The *all-log-tids* list, *in-doubt* list, and *decided* list are garbage collected with the redo-log: whenever a tid is removed from the redo-log, it is also removed from these other lists.

The *forced-abort* list, however, cannot be garbage collected in this way, otherwise it might corrupt the system (this may happen if a minitransaction's original coordinator is still running after the minitransaction has been garbage collected). To garbage collect *forced-abort*, we rely on an *epoch number*, which is a system-wide counter that increases monotonically very slowly (once per hour).

The *current epoch* is kept by each memory node (participants) using loosely synchronized clocks, and coordinators learn it by having participants piggyback the latest epoch onto their messages. A minitransaction is assigned an epoch by its coordinator, which is frequently the current epoch, but sometimes may lag behind. A participant votes to abort any minitransactions whose epoch is *stale*, meaning *two* or more epochs behind the current one. This allows participants to garbage collect any *tids* in *forced-abort* with a stale epoch, since such minitransactions will always get an abort vote. Epochs may abort otherwise successful minitransactions that either execute for more than 1 hour or that originate from a coordinator with a stale epoch. The former is unlikely to happen, since minitransactions are short lived. The latter could happen if the coordinator has executed no minitransactions for hours; in this case, the coordinator can simply retry its aborted minitransaction.

4.8 Further optimizations

If a minitransaction has just one participant, it can be executed in one phase because its outcome depends only on that participant. This creates further incentives for having node locality. For instance, SinfoniaFS tries to maintain a file's content and its inode in the same memory node (if space allows), to take advantage of one-phase minitransactions when manipulating this file.

Another optimization is for *read-only minitransactions*, that is, minitransactions without write items, which do not modify any data. For these, it is not necessary for memory nodes to store the minitransaction in the redo-log, because these minitransactions have no data to recover on failures.

4.9 Consistent backups

Sinfonia can perform transactionally-consistent backups of its data. To do so, each memory node takes note of the last committed minitransaction L in its redo-log, and updates the disk image up to minitransaction L. Meanwhile, new minitransactions are temporarily prevented from updating the disk image; this does not require pausing them, since they can execute by updating just the redo-log[3]. Once the disk image reflects minitransactions up to L, the disk image is copied or snapshotted, if the local file system or storage device supports snapshots. Then, updates to the disk image are resumed, while the backup is made from the copy or snapshot.

To ensure transactional consistency, we must start the above procedure "simultaneously" at all memory nodes, at a time when there are no outstanding minitransactions. To do so, we use a two-phase protocol: phase 1 locks all addresses of all nodes, and phase 2 starts the above procedure and releases all locks immediately (as soon as L is noted and subsequent minitransactions are prevented from updating the disk image). Recall that the two-phase protocol for minitransactions avoids deadlocks by acquiring locks without blocking: if some lock cannot be acquired, the coordinator releases all locks and retries after a while. This is reasonable for minitransactions that touch a small number of memory nodes. But a backup involves all memory nodes, so this scheme leads to starvation in a large system. Therefore, the two-phase protocol for creating backup images uses blocking lock requests instead, and does so in memory node order to avoid a deadlock with another backup request.

4.10 Replication

If desired, Sinfonia can replicate memory nodes for availability using standard techniques. We integrate primary-copy replication into the two-phase minitransaction protocol, so that the replica is updated while the primary writes its redo-log to stable storage in the first phase of commit. When the replica receives an update from the primary, it writes it in its redo-log and then acknowledges the primary.

Primary-copy replication is very simple, but it has a shortcoming: it relies on synchrony to fail over the primary to the replica. Used in an asynchronous system, primary-copy replication can lead to *false fail-overs*, where both primary and replica become active and the system loses consistency. We mitigate false fail-overs by using *lights-out management*—a feature available in typical data centers, which allows remote software to power on and off a machine regardless of its CPU state. Thus, Sinfonia powers down the primary when it fails over to the replica. Sinfonia could also have used an alternative solution to the false-fail-over problem: replicate memory nodes using state machines and Paxos [19] instead of primary-copy.

4.11 Configuration

Applications refer to memory nodes using a *logical memory id*, which is a small integer. In contrast, the *physical memory id* consists of a network address (IP address) concatenated with an application id. The map of logical to physical memory ids is kept at the *Sinfonia directory server*; this map is read and cached by application nodes when they initialize. This server has a fixed network name (DNS name) and can be replicated for availability. The logical to physical memory id mapping is static, except that new memory nodes can be added to it. When this happens, the application must explicitly recontact the Sinfonia directory server to obtain the extended mapping.

5. APPLICATION: CLUSTER FILE SYSTEM

We used Sinfonia to build a cluster file system called SinfoniaFS, in which a set of cluster nodes share the same files. SinfoniaFS is scalable and fault tolerant: performance can increase by adding more machines, and the file system continues to be available despite the crash of a few nodes in the system; even if all nodes crash, data is never lost or left inconsistent. SinfoniaFS exports NFS v2, and cluster nodes mount their own NFS server locally. All NFS servers export the same file system.

Cluster nodes are application nodes of Sinfonia. They use Sinfonia to store file system metadata and data, which include inodes, the free-block map, chaining information with a list of data blocks for inodes, and the contents of files. We use Sinfonia in modes `LOG` or `LOG-REPL`, which use a disk image to keep the contents of a memory node, so these data structures are stored on disk and are limited by disk size.

Sinfonia simplified the design of the cluster file system in four ways. First, nodes in the cluster need not coordinate and orchestrate updates; in fact, they need not be aware of each other's existence. Second, cluster nodes need not keep journals to recover from crashes in the middle of updates. Third, cluster nodes need not maintain the status of caches at remote nodes, which often requires complex protocols that are difficult to scale. And fourth, the implementation can leverage Sinfonia's write ahead log for performance without having to implement it again.

5.1 Data layout

Data layout (Figure 6) is similar to that of a local file system on a disk, except that SinfoniaFS is laid out over many Sinfonia memory nodes. The *superblock* has static information about the entire volume, like volume name, number of data blocks, and number of inodes. *Inodes* keep the attributes of files such as type, access mode, owner, and timestamps. Inode numbers are pairs with a memory node id and a local offset, which allows a directory to point

[3]This description is for all Sinfonia modes except `RAM` and `RAM-REPL`. These modes requires flushing RAM to disk to create a backup; while this happens, the redo-log (which is normally disabled in these modes) is enabled to buffer updates.

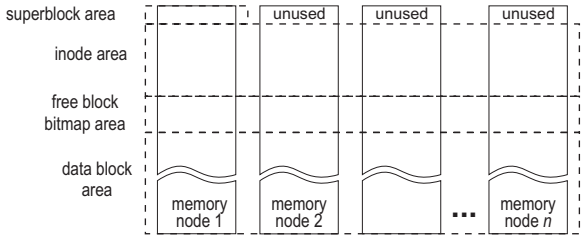

Figure 6: Data layout for SinfoniaFS. These data structures are on disk, as Sinfonia runs in mode LOG *or* LOG-REPL.

```
1 setattr(ino_t inodeNumber, sattr_t newAttributes){
2  do {
3     inode = get(inodeNumber);        // get inode from inode cache
4     newiversion = inode–>iversion+1;
5     t = new Minitransaction;
6     t–>cmp(MEMNODE(inode), ADDR_IVERSION(inode),
          LEN_IVERSION, &inode–>iversion);  // check inode iversion
7     t–>write(MEMNODE(inode), ADDR_INODE(inode),
          LEN_INODE, &newAttributes);  // update attributes
8     t–>write(MEMNODE(inode), ADDR_IVERSION(inode),
          LEN_IVERSION, &newiversion);   // bump iversion
9     status = t–>exec_and_commit();
10    if (status == fail) ... // reload inodeNumber into cache
11 } while (status == fail); }
```

Figure 7: The box shows code to create and commit a minitransaction to change an inode's attributes in SinfoniaFS. Lines 6–8 populate the minitransaction and line 9 executes and commits it.

to inodes at any memory node. *Data blocks* of 16 KB each keep the contents of files. Data block numbers are pairs with a memory node id and an offset local to the memory node. The *free-block bitmap* indicates which data blocks are in use. Chaining-list blocks indicate which blocks comprise a file; they are stored in data blocks, and have pointers to the next block in the list. Note that a 4 GB file requires only 65 chaining-list blocks (each chaining block can hold 4095 block numbers), so we did not implement indirect blocks, but they could be implemented easily. Directories and symbolic links have their contents stored like regular files, in data blocks.

5.2 Making modifications and caching

Cluster nodes use minitransactions to modify file system structures, like inodes and directories, while preserving their integrity. Memory nodes store the "truth" about the file system: the most recent version of data.

Cluster nodes can cache arbitrary amounts of data or metadata, including inodes, the free-block bitmap, and the content of files and directories. Because cache entries get stale, they are validated against data in memory nodes before being used. Validation occurs by adding compare items to a minitransaction, to check that the cached version matches what is in Sinfonia. For example, Figure 7 shows the implementation of NFS's setattr, which changes attributes of an inode. The compare item in line 6 ensures that the minitransaction only commits if the cached version matches what is in Sinfonia. If the minitransaction aborts due to a mismatch, the cache is refreshed and the minitransaction is retried. For a read-only file system operation, such as getattr (stat), cache entries are validated with a minitransaction with just compare items: if the minitransaction commits then the cache entries checked by the compare items are up-to-date.

1. if local cache is empty then load it
2. make modifications in local cache
3. issue a minitransaction that checks the validity of local cache using compare items, and updates information using write items
4. if minitransaction fails, check the reason and, if appropriate, reload cache entries and retry, or return an error indicator.

Figure 8: One minitransaction does it all: the above template shows how SinfoniaFS implements any NFS function with 1 minitransaction.

1. if file's inode is not cached then load inode and chaining list
2. find a free block in the cached free-block bitmap
3. issue a minitransaction that checks iversion of the cached inode, checks the free status of the new block, updates the inode's iversion, appends the new block to the inode's chaining list, updates the free-block bitmap, and populates the new block
4. if minitransaction failed because the inode version does not match then reload inode cache entry and retry
5. if minitransaction failed because block is not free then reload free-block bitmap and retry
6. if minitransaction failed for another reason then retry it
7. else return success

Figure 9: Implementing write that appends to a file, allocating a new block.

File read operations and file system operations that modify data always validate cache entries against memory nodes. Thus, if a cluster node writes to a file, this write will be visible immediately by all cluster nodes: if another cluster node reads the file, its read will perform a validation of cached data, which will fail, causing it to fetch the new contents from the memory nodes. For efficiency, readdir, lookup, or stat (getattr) may use cached inode attributes updated recently (within 2s) without validating them. So, these operations (but not file read) may return slightly stale results, as with NFS-mounted file systems.

In SinfoniaFS we implemented every NFS function with a single minitransaction. Figure 8 shows the general template to do this, and Figure 9 shows a specific example.

5.3 Node locality

An inode, its chaining list, and its file contents could be stored in different memory nodes, but SinfoniaFS tries to colocate them if space allows. This allows minitransactions to involve fewer memory nodes for better scalability.

5.4 Contention and load balancing

SinfoniaFS uses Sinfonia load information to decide where to write new data (load-based file migration is also possible but has not been implemented). We currently use the follow simple probabilistic scheme. Each cluster node keeps a *preferred memory node*, where new data is written. From time to time, the cluster node checks if its preferred memory node has too much load from other cluster nodes. If so, with a small *change probability*, the cluster node changes its preferred memory node to the memory node with least load. The small change probability avoids many simultaneous changes that might cause load oscillations.

Within a memory node, if a cluster node tries to allocate an inode or block and the minitransaction fails (because another cluster node allocates it first), the cluster node chooses a new inode or block at random to try to allocate again.

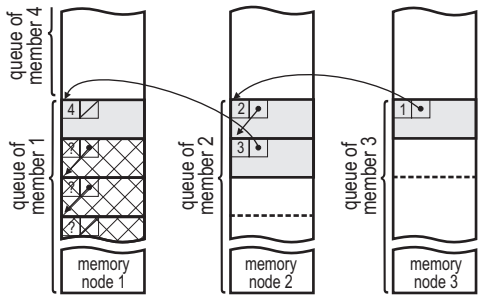

Figure 10: Basic design of SinfoniaGCS. Gray messages were successfully broadcast: they are threaded into the global list. Cross-hatched messages are waiting to be threaded into the global list, but they are threaded locally.

6. APPLICATION: GROUP COMMUNICATION

Intuitively, a group communication service [2, 8] is a chat room for distributed applications: processes can join and leave the room (called a *group*) and broadcast messages to it. Processes in the room (*members*) receive messages broadcast by members and *view change* messages indicating members have joined or left. The service ensures that all members receive the same messages and in the same order, for consistency. The current set of members is called the *latest view*.

Implementations of group communication rely on complicated protocols to ensure total ordering of messages and tolerate failures (see [10] for a survey). For example, in one scheme, a token rotates among members and a member only broadcasts if it has the token. Since no two members broadcast simultaneously, this scheme ensures a total order. The complexity, in this case, comes from handling crashes of a member who has the token and maintaining the ring to circulate tokens.

6.1 Basic design

To implement a group communication service, we store messages in Sinfonia and use minitransactions to order them. The simplest design, which performs poorly, uses a circular queue of messages stored in Sinfonia. To broadcast a message, a member finds the *tail* of the queue and adds the message to it using a minitransaction. The minitransaction may fail if a second member broadcasts concurrently and manages to execute its minitransaction before the first member. In this case, the first member has to retry by finding the new tail and trying to add its message at that location. This design performs poorly when many members broadcast simultaneously, because each time they retry they must resend their message to Sinfonia, which consumes bandwidth.

To solve this problem, we extend the above design as shown in Figure 10. Each member has a dedicated circular queue, stored on one memory node, where it can place its own broadcast messages without contention. Messages of all queues are "threaded" together with "next" pointers to create a single global list. Thus, to broadcast a message m, a member adds m to its queue, finds the end of the global list—the *global tail*—and updates the global tail to point to m, thus threading m into the global list. Only this last step (update pointer) requires a minitransaction. If unsuccessful, the member must retry, but it does not have to transfer m again to Sinfonia. This design is more efficient than the previous one because (1) it reduces the duration that the global tail location is accessed, thereby reducing contention on it, and (2) a member that broadcasts at a high rate can place multiple messages on its dedicated queue while the global tail is under contention.

Each member keeps a tail pointer indicating the latest message it received; this pointer does not have to be in Sinfonia. To receive new messages, a member just follows the "next" pointers in the global list, starting from the message indicated by its tail pointer.

View change messages are broadcast like any other messages by the member that requests the view change. We also maintain some group metadata in Sinfonia: the latest view and the location of each member's queue. To join, a member acquires a global lease on the metadata using a compare-and-swap minitransaction. Then, the member updates the latest view, finds the global tail, broadcasts a view change message, and releases the global lease. To leave, a member uses an analogous procedure.

For separation of concerns, our service does not automatically remove members that crash [28]. Rather, a member can use a separate failure detection service [5] to detect crashes and then execute a leave operation for a crashed member.

6.2 Garbage collection

If the queue of a member fills up, it must free entries that are no longer needed: those with messages that every member in the latest view has already received. For this purpose, we keep (in Sinfonia) an approximation of the last message that each member received from each queue. Each member periodically updates this last-received pointer (using a minitransaction) if it has changed. When a queue becomes full, a member that wants to broadcast frees unneeded messages in its queue.

6.3 Optimizations

Finding the global tail. We do not keep a pointer to the global tail, because it is too expensive to maintain. Instead, each member p reports the last message that p threaded (i.e., added to the global list). This information is placed (with a minitransaction) in a location $lastThreaded_p$ in Sinfonia. Moreover, when a message is threaded, it receives a monotonic global sequence number (GSN). Thus, members can efficiently find the global tail by reading $lastThreaded_p$ for each member p and picking the message with highest GSN. Another optimization is for members that only want to broadcast (without receiving messages): they periodically set their last received pointers to $lastThreaded_p$ for each member p.

Coalesced threading. Members broadcast messages asynchronously, that is, without receiving confirmation until later. If a member is broadcasting at a high rate, it delays the threading of messages into the global tail. Thus, a member's queue can have many messages waiting to be threaded. For efficiency, these are threaded together within the queue (cross-hatched messages in Figure 10), so that the member only needs to thread the first message into the global list, and then set the GSN of its latest unthreaded message. When this happens, the member receives (delayed) confirmation that its messages were broadcast.

Small messages. If a member broadcasts many small messages, we combine many of them into larger messages before placing them into the queue.

7. EVALUATION

We evaluated Sinfonia and the applications we built with it. Our testing infrastructure has 247 machines on 7 racks connected by Gigabit Ethernet switches. Intra-rack bisection bandwidth is $\approx$14 Gbps, while inter-rack bisection bandwidth is $\approx$6.5 Gbps. Each machine has two 2.8GHz Intel Xeon CPUs, 4GB of main memory, and two 10000rpm SCSI disks with 36GB each. Machines run Red Hat Enterprise Linux WS 4 with kernel version 2.6.9.

Our machines do not have NVRAM, so for Sinfonia modes that use NVRAM, we used RAM instead. This is reasonable since one type of NVRAM is battery-backed RAM, which performs identi-

Keys are addresses aligned at 4-byte boundaries, and each value is 4 bytes long, corresponding to one minitransaction item. We used the B-tree access method, synchronous logging to disk, group commit, and a cache that holds the whole address space. We ran Berkeley DB on a single host, so to use it in a distributed system, we built a multithreaded RPC server that waits for a populated minitransaction from a remote client, and then executes it locally within Berkeley DB. We also tried using Berkeley DB's RPC interface, but it performed much worse because it incurs a network round-trip for each item in a minitransaction, rather than a round-trip for the entire minitransaction.

Figure 11: Berkeley DB setup for all experiments.

cally to RAM. In all figures and tables, error bars and ± variations refer to 95% confidence intervals.

7.1 Sinfonia service

We evaluated the base performance of Sinfonia, the benefit of various minitransaction optimizations, the scalability of minitransactions, and their behavior under contention. Each experiment considered several Sinfonia modes of operation as described in Figure 3. When replication was used, we colocated the primary of a memory node with the replica of another memory node using chained declustering [18].

7.1.1 Result: base performance

We first tested base performance of a small system with 1 memory node and 1 application node. We considered minitransactions with 4-byte items with word-aligned addresses. We compared Sinfonia against Berkeley DB 4.5, a commercial open-source developer database library, configured as explained in Figure 11. Figure 12 shows throughput and latency for a workload that repeatedly issues minitransactions, each with six items[4] chosen randomly over a range of 50 000 items. We varied the number of outstanding minitransactions at the application node (# of threads) from 1 to 256. In lines labeled **Sinfonia-mode**, **mode** refers to one of Sinfonia's modes of operation in Figure 3. Modes **LOG** and **LOG-REPL** log minitransactions synchronously to disk, so these modes are more appropriate to compare against Berkeley DB, which also logs to disk.

As can be seen, the various Sinfonia modes have good performance and can reasonably trade off fault tolerance for performance. Both **Sinfonia-NVRAM** and **Sinfonia-NVRAM-REPL** can execute over 7500 minitrans/s with a latency of ≈3ms while **Sinfonia-LOG** and **Sinfonia-LOG-REPL** can execute 2400 and 2000 minitrans/s with a latency of 13ms and 16ms, respectively. All Sinfonia modes peak at around 6500-7500 minitrans/s, because the bottleneck was the CPU at the application node, which operates identically in all modes.[5] With a few threads, Berkeley DB has similar performance to **Sinfonia-LOG** and **Sinfonia-LOG-REPL**, as the system is limited by synchronous writes to disk. With a larger number of threads, Berkeley DB's performance peaks below Sinfonia. We believe this is because of contention in Berkeley DB's B-tree: it uses a page-level lock-coupling technique which locks a leaf page for every modification, blocking other transactions that access different locations in the same page.

[4]Six is the median number of items in a minitransaction of SinfoniaFS while running the Andrew benchmark.

[5]With **Sinfonia-LOG** and **Sinfonia-LOG-REPL**, if the primary memory node were colocated with the replica of another node, the bottleneck would be the memory node. We did not do that here as there is only 1 memory node.

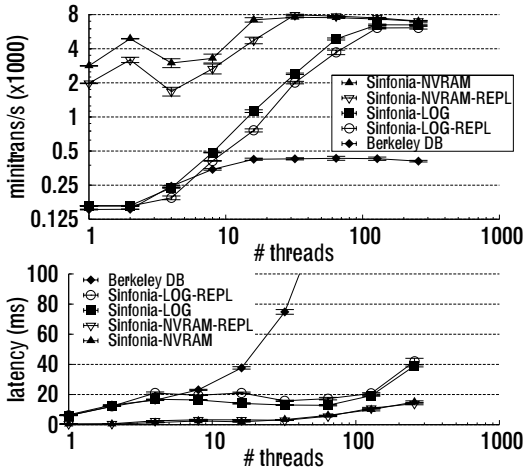

Figure 12: Performance of Sinfonia with 1 memory node.

7.1.2 Result: optimization breakdown

We measured the benefit of the various minitransaction optimizations in Sinfonia. We considered 4 systems with different levels of optimization:

- *System 1: Non-batched items.* When an application node adds an item to the minitransaction, the system immediately reads or writes the item (depending on its type) and locks its location at the proper memory node. Application nodes see the results of reads as the transaction executes, providing more flexibility than Sinfonia minitransactions do. At the end of the transaction, the system executes two-phase commit. This is the traditional way to execute transactions.
- *System 2: Batched items.* The system batches all minitransaction items at the application node. At the end of the transaction, the system reads or writes all items together, and then executes two-phase commit.
- *System 3: 2PC combined.* The system batches minitransaction items and accesses them in the first phase of two-phase commit. This saves an extra network round-trip compared to System 2, and it is how Sinfonia executes minitransactions involving many memory nodes.
- *System 4: 1PC combined.* The minitransaction executes within the first phase of one-phase commit. This saves another network round-trip compared to System 3, and it is how Sinfonia executes minitransactions involving one memory node.

We considered separately read-write and read-only minitransactions, with small (6) and large (50) numbers of minitransaction items. Read-write minitransactions consisted of compare-and-swaps, and read-only minitransactions consisted of compares only. Compare-and-swaps were set up to always succeed (by always swapping the same value). Items had 4 bytes and were chosen randomly from a set of 50 000 items. There were 4 memory nodes and 4 application nodes. Each application node had 32 threads issuing minitransactions as fast as possible. The Sinfonia mode was **LOG** (the other modes had similar results, except that in modes that do not log to disk, read-write and read-only minitransactions behaved similarly). In all systems, items were cached in RAM at memory nodes, and disk writes were synced only when committing the minitransaction.

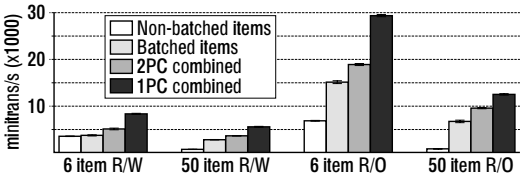

Figure 13: Performance with various combinations of techniques.

Figure 13 shows the aggregate throughput of each system (error bars are too small to see). As can be seen, all optimizations produce significant benefits. With 6 item read-only minitransactions, batching items improves throughput by 2.2x and combining execution with 2PC produces another 1.3x improvement. Large minitransactions benefit more, as expected. Read-only minitransactions benefit more than read-write minitransactions, since the latter are dominated by disk writes. Even so, for 6 item read-write minitransactions, the workload that benefits least, 2PC-combined is 44% faster than the non-batched system. For minitransactions involving just one memory node, running 1PC improves throughput over 2PC by 1.3x–1.6x, depending on the workload.

7.1.3 Result: scalability

We evaluated the scalability of Sinfonia by measuring its performance as we increased the system size and the workload together. In this experiment, there were up to 246 machines: half memory nodes and half application nodes. Each application node had 32 threads issuing minitransactions as fast as possible. Minitransactions had 6 items arranged as 3 compare-and-swaps chosen randomly, with the restriction that *minitransaction spread* be 2 (except when there is only 1 memory node). Minitransaction spread is the number of memory nodes that a minitransaction touches. Each memory node stored 1 000 000 items. Minitransaction optimizations were enabled, that is, minitransactions executed in the commit protocol.

The first graph in Figure 14 shows aggregate throughput as system size increases (the smallest size is 2, with 1 memory node and 1 application node). The table shows the exact numeric data in the graph. As can be seen, performance increases monotonically as we increase the system size, except from size 2 to 4. This is because the system of size 2 has 1 memory node, so minitransactions use 1PC, while with larger system sizes, minitransactions involve many memory nodes, which requires using the slower 2PC. We call this the *cost of distributing the system*. We also included a line for a Berkeley DB configuration (see Figure 11) to see if a single-server system outperforms small distributed systems, which sometimes happens in practice. This did not happen with Sinfonia.

We also quantify the scalability efficiency at different system sizes, by showing how well throughput increases compared to a linear increase from a base size. More precisely, we define *scalability efficiency* as

$$\frac{target\_throughput}{base\_throughput} \div \frac{target\_size}{base\_size}$$

where *target_size*>*base_size*. This metric is relative to a chosen base size. An efficiency of 0.9 as the system size doubles from 4 to 8 means that the larger system's throughput is a factor of 0.9 from doubling, a 10% loss. The second graph in Figure 14 shows scalability efficiency for each Sinfonia mode, comparing it with a perfectly scalable system, whose efficiency is always 1. As can be seen, except when the size increases from 2 to 4, efficiency is reasonable for all modes at all sizes: most have efficiency 0.9 or higher, and the lowest is 0.85.

system	system size							
	2	4	8	16	32	64	128	246
Sinfonia-NVRAM	7.5	11	19	37	73	141	255	508
Sinfonia-NVRAM-REPL	7.9	5.9	11	23	44	74	130	231
Sinfonia-LOG	2.4	2.3	5.1	10	21	37	73	159
Sinfonia-LOG-REPL	1.9	1.3	2.1	5	11	21	41	71

Table shows thousands of minitrans/s

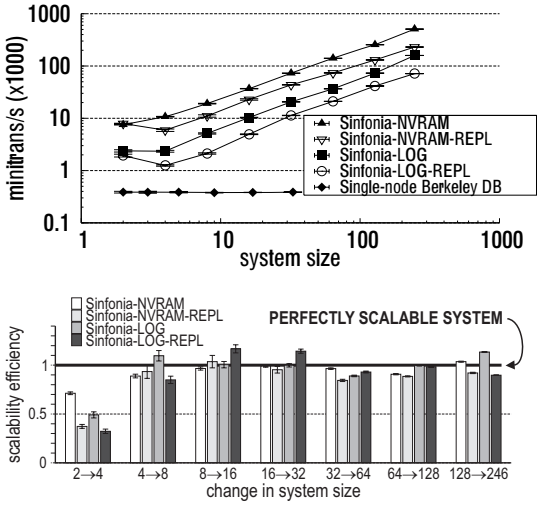

Figure 14: Sinfonia scalability.

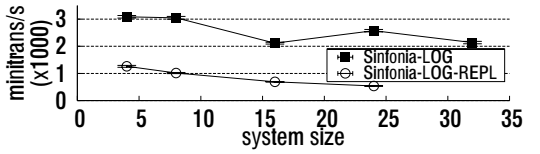

Figure 15: Effect of minitransaction spread on scalability.

In our next experiment, as we increased the system size, we also increased minitransaction spread, so that each minitransaction involved every memory node. Minitransactions consisted of 16 compare-and-swaps, so that we could increase spread up to 16. Figure 15 shows the result for Sinfonia modes LOG and LOG-REPL (other modes are similar). Note that the axes have a linear scale, which is different from Figure 14.

As can be seen, there is no scalability now. We did not know this initially, but in retrospect the explanation is simple: a minitransaction incurs a high initial cost at a memory node but much smaller incremental cost (with number of items), and so spreading it over many nodes reduces overall system efficiency. Thus, to achieve optimal scalability, we obey the following simple rule: *Across minitransactions, spread load; within a minitransaction, focus load*. In other words, one should strive for each minitransaction to involve a small number of memory nodes, and for different minitransactions to involve different nodes.

Finally, we ran experiments where we increased the number of application nodes without increasing the number of memory nodes, and vice-versa. The results were not surprising: in the first case, throughput eventually levels off as Sinfonia saturates, and in the second case, system utilization drops since there is not enough offered load.

7.1.4 Result: contention

In our next experiment, we varied the probability that two minitransactions overlap, causing contention, to see its effect on

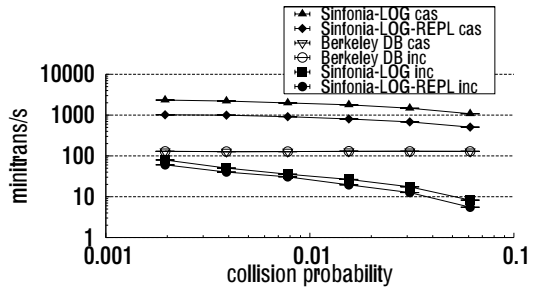

Figure 16: Effect of minitransaction overlap on performance.

throughput. Minitransactions consisted of 8 compare-and-swaps that were set up to always succeed (by always swapping the same value), so that we can measure the efficiency of our commit protocol in isolation. Sinfonia had 4 memory nodes and minitransaction spread was 2. There were 4 application nodes each with 16 outstanding minitransactions at a time. All items were selected randomly from some set whose size determined the probability of pairwise overlap. For instance, with 1024 items to choose from, the probability that two minitransactions overlap on at least one item is $1-\binom{1024-8}{8}/\binom{1024}{8} \approx 0.06$. Figure 16 shows the result on the top three lines labeled "cas" (compare-and-swap).

As can be seen, Sinfonia provides much better throughput than Berkeley DB, even with high contention. We believe this is because of contention in Berkeley DB's B-tree. We also measured latency, and the results are qualitatively similar.

In another experiment, we used Sinfonia to perform an operation not directly supported by minitransactions: increment values atomically. We did so by having a local cached copy of the values at the application node, and using a minitransaction to validate the cache and write the new value. Here, a minitransaction could fail because a compare fails. In this case, the application refreshed its cache and retried the minitransaction. In essence, this amounted to using minitransactions to implement optimistic concurrency control. The results are shown in Figure 16 by the lines labeled "inc" (increment).

Berkeley DB performs as with the compare-and-swap experiment because both experiments incur the same locking overhead. But Sinfonia performs much worse, because Sinfonia has to retry multiple times as contention increases. We conclude that, for operations not directly supported by minitransactions, Sinfonia does not perform well under high contention. We could have extended minitransactions to have "increment items" besides the read, write and compare items, as explained in Section 3.3. Had we done so, we could execute increments much more efficiently, but this was not required by our applications.

7.1.5 Result: ease of use

We evaluated Sinfonia's ease of use by building two applications with it, SinfoniaFS and SinfoniaGCS. Figure 17 shows typical development effort metrics used in software engineering. In LOC (lines of code), "glue" refers to the RPC interface in SinfoniaFS and LinuxNFS, and "core" is the rest. LinuxNFS is the NFS server in Linux. SinfoniaFS compares well with LinuxNFS even though LinuxNFS is a centralized client-server system while SinfoniaFS is distributed. SinfoniaGCS fares much better than Spread on these metrics. While SinfoniaGCS is only a prototype and Spread is open-source software (and hence, more mature code), a difference of an order of magnitude in lines of code is considerable.

We also report qualitatively on our experience in building SinfoniaFS and SinfoniaGCS, including advantages and drawbacks.

	SinfoniaFS	LinuxNFS	SinfoniaGCS	Spread
LOC (core)	2831	5400		
LOC (glue)	1024	500		
LOC (total)	3855	5900	3492	22148
(language)	C++	C	C++	C
Develop time	1 month	unknown	2 months	years
Major versions	1	2	1	4
Code maturity	prototype	open source	prototype	open source

Figure 17: Comparison of development effort.

Phase	Description
1	create 605 files in 363 directories 5 levels deep
2	remove 605 files in 363 directories 5 levels deep
3	do a stat on the working directory 250 times
4	create 100 files, and changes permissions and stats each file 50 times
4a	create 10 files, and stats each file 50 times
5a	write a 1MB file in 8KB buffers 10 times
5b	read the 1MB file in 8KB buffers 10 times
6	create 200 files in a directory, and read the directory 200 times; each time a file is removed
7a	create 100 files, and then rename and stat each file 10 times
7b	create 100 files, and link and stat each file 10 times
8	create 100 symlinks, read and remove them 20 times
9	do a statfs 1500 times

Figure 18: Connectathon NFS Testsuite modified for 10x work.

We found that the main advantages of using Sinfonia were that (1) transactions relieved us from issues of concurrency and failures, (2) we did not have to develop any distributed protocols and worry about timeouts, (3) application nodes did not have to keep track of each other, (4) the correctness of the implementation could be verified by checking that minitransactions maintained invariants of shared data structures, (5) the minitransaction log helped us debug the system by providing a detailed history of modifications; a simple Sinfonia debugging mode adds short programmer comments and current time to logged minitransactions.

The main drawback of using Sinfonia is that its address space is a low-level abstraction with which to program. Thus, we had to explicitly lay out the data structures onto the address space and base minitransactions on this layout. With SinfoniaFS, this was not much of a problem, because the file system structures led to a natural data layout. With SinfoniaGCS, we had to find a layout that is efficient in the presence of contention—an algorithmic problem of designing a concurrent shared data structure. This was still easier than developing fault-tolerant distributed protocols for group communication (a notoriously hard problem).

7.2 Cluster file system

We now consider the performance and scalability of SinfoniaFS.

7.2.1 Result: base performance

We first evaluated the performance of SinfoniaFS at a small scale, to ensure that we are scaling a reasonable system. We used 1 memory node and 1 cluster node running SinfoniaFS; a cluster node corresponds to an application node of Sinfonia. We compare against NFS with 1 client and 1 server.

We first ran SinfoniaFS with the Connectathon NFS Testsuite, which is mostly a metadata intensive microbenchmark with many phases, each of which exercises one or two file system functions. We modified some phases to increase the work by a factor of 10, shown in Figure 18, because otherwise they execute too quickly.

Figure 19 shows the benchmark results for SinfoniaFS compared to a Linux Fedora Core 3 NFS server, where smaller numbers are

Phase	Linux NFS (s)	SinfoniaFS-LOG (s)	SinfoniaFS-LOG-REPL(s)
1	19.76 ± 0.59	5.88 ± 0.04	6.05 ± 0.15
2	14.96 ± 0.99	6.13 ± 0.15	6.44 ± 0.43
3	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
4	123.4 ± 0.44	60.87 ± 0.08	61.44 ± 0.15
4a	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
5a	9.53 ± 0.27	8.27 ± 0.14	8.44 ± 0.06
5b	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
6	2.65 ± 0.09	2.68 ± 0.04	2.77 ± 0.04
7a	37.49 ± 0.21	12.25 ± 0.14	12.32 ± 0.10
7b	25.30 ± 0.21	12.47 ± 0.24	12.21 ± 0.05
8	50.90 ± 0.25	24.71 ± 0.08	25.27 ± 0.26
9	0.19 ± 0.00	0.71 ± 0.02	0.71 ± 0.02

Figure 19: Results of modified Connectathon NFS Testsuite.

better as they indicate a shorter running time. We used the NFSv2 protocol in both cases, and the underlying file system for the NFS server is ext3 using ordered journaling mode. Sinfonia was set to LOG or LOG-REPL mode, which logs to disk, and the NFS server was set to synchronous mode to provide data durability. As can be seen, except in phase 9, SinfoniaFS performs at least as well, and typically 2-3 times better than, Linux NFS. The main reason is that SinfoniaFS profits from the sequential write-ahead logging provided by Sinfonia, which is especially beneficial because Connectathon has many operations that modify metadata. Note that phases 3, 4a, and 5b executed mostly in cache, so these results are not significant. In phase 9, SinfoniaFS returns cached values to statfs, so the latency is equal to the communication overhead between the NFS client and server. LinuxNFS performs better because it is implemented in the kernel. Finally, in all phases, SinfoniaFS-LOG-REPL performs only slightly worse than SinfoniaFS-LOG because the experiments do not exhaust bandwidth, and some of the latency of replication is hidden by Sinfonia (see Section 3).

Next, we ran a macro benchmark with a more balanced mix of data and metadata operations. We modified the Andrew benchmark to use as input the Tcl 8.4.7 source code, which has 20 directories and 402 regular files with a total size of 16MB (otherwise Andrew runs too quickly). The benchmark has 5 phases: (1) duplicate the 20 directories 50 times, (2) copy all data files from one place to one of the duplicated directories, (3) recursively list the populated duplicated directories, (4) scan each copied file twice, and (5) do a "make".

Figure 20 shows the results, again comparing SinfoniaFS with Linux NFS. As can be seen, in almost all phases, SinfoniaFS performs comparably to or better than the NFS server. In phase 1 it performs far better because the phase is metadata intensive, similar to the Connectathon benchmark. In phase 4, SinfoniaFS performs worse, as lots of data is read and both Sinfonia and SinfoniaFS run in user space without buffer-copying optimizations. Sinfonia memory nodes incur several copies (disk to user memory, and vice versa when sending the data to SinfoniaFS) and SinfoniaFS incurs further copies receiving data (kernel to user), and sending it the NFS client (user to kernel). A better implementation would have both Sinfonia and SinfoniaFS in the kernel to avoid copying buffers, and would use VFS as the interface to SinfoniaFS instead of NFS. In phase 5, SinfoniaFS performs slightly better as the benefits of the write-ahead logging outweigh the copying overheads.

7.2.2 Result: scalability

We ran scalability tests by growing the system size and workload together. We started with 1 memory node and 1 cluster node

Phase	Linux NFS (s)	SinfoniaFS--LOG (s)	SinfoniaFS-LOG-REPL (s)
1 (mkdir)	26.4 ± 1.6	6.9 ± 0.2	8.8 ± 0.7
2 (cp)	53.5 ± 1.6	46.3 ± 0.2	48.6 ± 0.3
3 (ls -l)	2.8 ± 0.1	3.0 ± 0.3	2.9 ± 0.2
4 (grep+wc)	6.1 ± 0.2	6.5 ± 0.1	6.5 ± 0.1
5 (make)	67.1 ± 0.6	51.1 ± 0.1	52.1 ± 0.2

Figure 20: Results of modified Andrew benchmark.

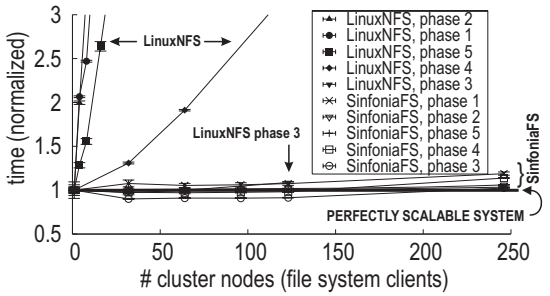

Figure 21: Results of Andrew as we scale the system and the workload together. The x-axis is the number of nodes running Andrew simultaneously. The y-axis is the duration of each phase relative to a system with 1 client. Connectathon had similar results.

and increased the number of memory nodes and cluster nodes together while running a benchmark on each cluster node, in order to see if any system overhead manifested itself as the system size increased. We synchronized the start of each phase of the benchmark at all cluster nodes, to avoid intermingling phases. We ran the same number of clients against the NFS server to be sure the benchmark overloads the server. Figure 21 shows the results for the Andrew benchmark, where the y-axis shows the duration of each phase relative to a system with 1 memory node and 1 cluster node.[6] The horizontal line at 1 represents the performance of an ideal perfectly scalable system. As can be seen, all the SinfoniaFS curves are close to the ideal system: only 18%, 6%, 2%, 14%, and 3% higher for phases 1–5 respectively, which is a reasonable efficiency. The result is very different for the NFS server, as it quickly becomes overloaded, and cannot be simply scaled as SinfoniaFS can be (the exception being phase 3, which scales because it almost exclusively uses data cached on the client). The results of Connectathon are very similar (not shown): all SinfoniaFS curves are flat at y-value 1, while NFS server curves increase extremely rapidly, except for phases 3, 4a, and 5b, which had 0 duration.

7.3 Group communication service

To evaluate the scalability characteristics of our implementation, we ran a simple workload, measured its performance, and compared it with a publicly available group communication toolkit, Spread [1], version 3.17.4 (Jan 2007). In each experiment, we had members broadcasting 64-byte messages as fast as possible (called *writers*) and members receiving messages as fast as possible (called *readers*). In all experiments, we report the aggregate read throughput of all readers.

For SinfoniaGCS, we combine up to 128 messages into a larger message as described in Section 6.3, to optimize the system for throughput. In addition, each memory node, reader, and writer was

[6]For the experiment with 246 cluster nodes, we had to colocate cluster and memory nodes, since we did not have enough machines. However, the probability a cluster node accessed a memory node in its own host was very small.

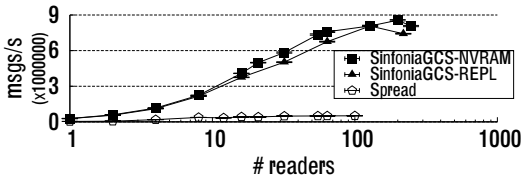

Figure 22: SinfoniaGCS base performance as we vary the number of readers. There are 8 writers, and 8 memory nodes or Spread daemons.

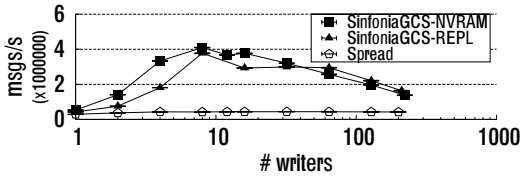

Figure 23: SinfoniaGCS base performance as we vary the number of writers. There are 16 readers, and 8 memory nodes or Spread daemons.

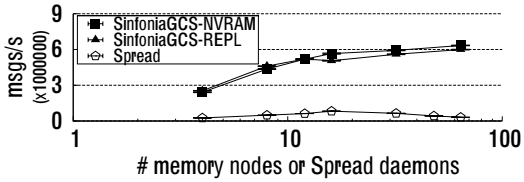

Figure 24: SinfoniaGCS scalability as we vary the number of memory nodes. There are 32 readers and 64 writers.

running on a separate machine. The Sinfonia mode was NVRAM or REPL. In mode REPL, we placed primary memory nodes and replicas in machines by themselves, without colocation.

For Spread, we evaluated two configurations. In the *separated configuration*, each daemon and each group member had its own machine, and each daemon served an equal number of members (± 1). In the *colocated configuration*, each group member had its own machine with a local Spread daemon. For both configurations, we used Spread's AGREED_MESS service type for sending messages. We found that the aggregate read throughputs for the colocated configurations were higher than those for the corresponding separated configurations (by up to a factor of 2) in small configurations, but dropped below the separated configuration as the number of daemons exceeded 16–20. We present only the results for the separated configuration, since it is more comparable to the Sinfonia GCS configuration, and its read throughput scales better than that of the colocated configuration. In addition, we ran Spread without broadcast or IP multicast, since broadcasts disrupt other applications, and IP multicast is often disabled or unsupported in data centers. Since Spread is optimized for use with broadcast or IP multicast, we expect that Spread would perform much better in settings where these broadcasts are available.

7.3.1 Result: base performance

In the first experiment, we observe how SinfoniaGCS behaves as we vary the number of readers (Figure 22). (In this and other figures, the 95% confidence intervals are too small to be visible.) We fixed the number of writers and memory nodes to 8 each, and varied the number of readers from 1 to 248. Spread used 8 daemons. For SinfoniaGCS-NVRAM, we see that the aggregate throughput increases linearly up to 20 readers and flattens out at about 64 readers to 8 million messages/second. After this point, the memory nodes have reached their read-capacity and cannot offer more read bandwidth. SinfoniaGCS using replication (SinfoniaGCS-REPL) performs similarly. The throughput is considerably higher than that of Spread up to 100 readers; we were unable to obtain reliable measurements for Spread with more than 100 readers.

In the second experiment, we observe the behavior as we vary the number of writers (Figure 23). We fixed the number of readers to 16, the number of memory nodes to 8, and varied the number of writers from 1 to 222. We see that when there are fewer writers than memory nodes, the throughput is below the peak because each queue is on a separate memory node, so not all memory nodes are utilized. When the number of writers exceed the number of memory nodes, aggregate throughput decreases gradually because (1) we reach the write capacity of the system, and additional writers cannot increase aggregate throughput, and (2) additional writers impose overhead and cause more contention for the global tail thereby decreasing aggregate throughput. The throughput of SinfoniaGCS-NVRAM considerably exceeds that of Spread throughout the measured range. For 32 writers or fewer, SinfoniaGCS-REPL performs only slightly worse than SinfoniaGCS-NVRAM ($<10\%$) due to the increased latency of having primary-copy replication. Surprisingly, SinfoniaGCS-REPL slightly outperforms SinfoniaGCS-NVRAM for large number of writers because the increased latency forces writers to write slower, which leads to less contention.

7.3.2 Results: scalability

In the third experiment, we observe the scalability as we increase the number of memory nodes (Figure 24). Since the intra-rack bisection bandwidth is higher than the inter-rack bisection bandwidth in our cluster, when readers and memory nodes were colocated in the same rack, they provided better performance. As a result, we ensured this property for 16 or fewer memory nodes, and for more memory nodes, we colocated them as much as possible, ensuring each rack had the same number of readers and memory nodes. There were 32 readers, 64 writers, and we varied the number of memory nodes from 4 to 64. We find that increasing the number of memory nodes improves the throughput up to a point (16 memory nodes in this case), but the throughput flattens out after that. This is due to the fixed number of readers and writers; as we see in the next experiment, a higher aggregate throughput can be obtained with more readers and writers. As in the other cases, the throughput exceeds the Spread throughput considerably.

In the fourth experiment, we observe the scalability with total system size (Figure 25). We used the same rack colocation rules for readers and memory nodes as before. For each system size, there was an equal number of readers, writers, and memory nodes. System size varied from 12 to 192 machines. For Sinfonia mode REPL, the system size did not count the memory node replicas. We see that the aggregate throughput increases through the entire range; the rate of growth slows gradually as the system size increases. This growth rate decrease occurs because (1) with more writers there is more global tail contention, and (2) the overhead of processing and updating per-queue metadata increases.

8. RELATED WORK

Services to help building large distributed systems include GFS [13], Bigtable [6], Chubby [4], and MapReduce [9]. GFS is a scalable file system that provides functions tailored for its use at Google, such as atomic appends but otherwise weak consistency. Bigtable is a distributed scalable store of extremely large amounts of indexed data. Chubby is a centralized lock manager for coarse-

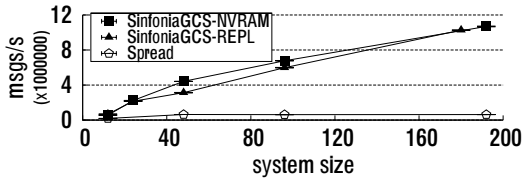

Figure 25: SinfoniaGCS scalability as we vary total system size (total number of readers, writers, and memory nodes or daemons).

grained locks; it can store critical infrequently-changing data, as with a name server or a configuration store. All these systems serve as higher-level building blocks than Sinfonia, intended for higher-level applications than Sinfonia applications. In fact, one might imagine using our approach to build these systems. MapReduce is a paradigm to concurrently process massive data sets distributed over thousands of machines. Applications include distributed grep and sort, and computing inverted indexes.

Sinfonia is inspired by database systems, transactional shared memory, and distributed shared memory. Transactional shared memory augments a machine with memory transactions [17]. Original proposals required hardware changes, but subsequent work [31] showed how to do it in software; more efficient proposals appeared later [16, 15]. These systems are all targeted at multiprocessor machines rather than distributed systems.

Distributed shared memory (DSM) emulates a shared memory environment over a distributed system. This line of work had success limited to few applications because its objective was too hard—to run existing shared memory applications transparently on a distributed system. The loosely-coupled nature of distributed systems created hard efficiency and fault tolerance problems.

Plurix (e.g., [11]) is a DSM system with optimistic transactions and a low-level implementation on its own OS to improve performance and provide transparent remote memory access through virtual memory. Plurix does not scale as Sinfonia does, since Plurix transactions require IP broadcast to fetch pages and commit. The granularity of access is a page, which produces false sharing in small data structures. The largest Plurix system has 12 hosts [11], and Plurix is not data center friendly because its frequent broadcasts are disruptive to other applications. Plurix provides less reliability than Sinfonia, as Plurix nodes are not replicated, transactions are not durable, and fault tolerance is limited to checkpointing to disk periodically (e.g., every 10s). Plurix applications include ray-tracing and interactive 3D worlds.

Perdis [12] is a transactional distributed store designed to support cooperative engineering applications in WANs. Perdis has long-lived transactions that span multiple LANs connected via a WAN. In contrast, Sinfonia's minitransactions are short-lived and streamlined to scale well in a data center. Thor [21] is a distributed object-oriented database system that provides optimistic transactions. Data is structured as objects, and objects are manipulated via type-checked method calls. While the Thor design allows transactions that span objects at many servers, actual Thor systems evaluated in the literature have only a single replicated server. Sinfonia differs from Thor in many ways. A typical Sinfonia system has tens or hundreds of memory nodes; data is unstructured, which imposes less overhead, albeit at the cost of a lower-level programming interface; and minitransactions are more streamlined than Thor transactions. For example, Sinfonia takes only 2 network round-trips to execute a minitransaction that checks that 2 flags at 2 memory nodes are clear and, if so, sets both of them. With Thor, this requires 3 round-trips: 1 round-trip to fetch both flags, plus 2 round-trips to commit.

BerkeleyDB and Stasis [30] provide transactional storage at a single node. Stasis has no support for distributed transactions, and BerkeleyDB only has minimal support for it (it lacks a distributed transaction manager). Transactional support on a disk-based system was proposed in Mime [7], which provided multi-sector atomic writes and the ability to revoke tentative writes; however, all the disks in Mime were accessed through a single controller.

Atomic transactions make a distributed system easier to understand and program, and were proposed as a basic construct in several distributed systems such as Argus [20], QuickSilver [29] and Camelot [33]. The QuickSilver distributed operating system supports and uses atomic transactions pervasively, and the QuickSilver distributed file system supports atomic access to files and directories on local and remote machines. Camelot was used to provide atomicity and permanence of server operations in the Coda file system [26] by placing the metadata in Camelot's recoverable virtual memory. This abstraction was found to be useful because it simplified crash recovery. However, the complexity of Camelot led to poor scalability; later versions of Coda replaced Camelot with the Lightweight Recoverable Virtual Memory [27], which dispensed with distributed transactions, nested transactions and recovery from media failures, providing only atomicity and permanence in the face of process failures. While this is adequate for Coda, which provides only weak consistency for file operations, distributed transactions, such as those provided by Sinfonia, are highly desirable for many distributed applications.

Remote Direct Memory Access (RDMA) is an efficient way to access data stored on a remote host while minimizing copying overhead and CPU involvement [25]. RDMA could be used in a more efficient implementation of Sinfonia than using message-passing as we do. Persistent Memory [23] is a fast persistent solid-state storage accessible over a network through RDMA, providing high bandwidth and low latency, but without transactional support.

There is a rich literature on distributed file systems, including several that are built over a high-level infrastructure designed to simplify the development of distributed applications. The Boxwood project [22] builds a cluster file system over a distributed B-tree abstraction. This work differs from ours in three main ways. First, Boxwood focuses on building storage applications rather than infrastructure applications. Second, Boxwood considered relatively small systems—up to 8 machines—rather than hundreds of machines. Third, Boxwood provides higher-level data structures than Sinfonia, but does not provide transactions. The Inversion File System [24] is built over a Postgres database; this is a fairly complex abstraction, and the performance of the file system was poor.

Gribble et al.[14] propose a scalable distributed hash table for constructing Internet services. Hash tables provide operations on key-value pairs, which are higher-level than the unstructured address spaces of Sinfonia. However, there is no support for transactions, which is an important feature of Sinfonia for handling concurrency and failures when data is distributed over many hosts.

9. CONCLUSION

We proposed a new paradigm for building scalable distributed systems, based on streamlined minitransactions over unstructured data, as provided by Sinfonia. This paradigm is quite general, given the very different nature of the applications that we built with it (a cluster file system and a group communication service). The main benefit of Sinfonia is that it can shift concerns about failures and distributed protocol design into the higher-level and easier problem of designing data structures. We are currently developing other distributed applications with Sinfonia, including a distributed lock

manager that maintains the state and wait queues for millions of locks, and a distributed B-tree for thousands of clients.

Acknowledgments. We would like to thank Terence Kelly and Jeff Mogul for helpful discussions; our shepherd Sharon Perl, the SOSP reviewers, Xiaozhou Li, and Janet Wiener for many comments that helped us improve the paper; and Yair Amir and John Schultz for suggestions on how to configure Spread.

10. REFERENCES

[1] Y. Amir and J. Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, The Johns Hopkins University, July 1998.

[2] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Symposium on Operating Systhem Principles*, pages 123–138, Nov. 1987.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. J. Mullender, editor, *Distributed Systems*, chapter 8. Addison-Wesley, 1993.

[4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation*, pages 335–350, Nov. 2006.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation*, pages 205–218, Nov. 2006.

[7] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9, HP Laboratories, Nov. 1992.

[8] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, pages 137–150, Dec. 2004.

[10] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, Dec. 2004.

[11] M. Fakler, S. Frenz, R. Goeckelmann, M. Schoettner, and P. Schulthess. Project Tetropolis—application of grid computing to interactive virtual 3D worlds. In *International Conference on Hypermedia and Grid Systems*, May 2005.

[12] P. Ferreira et al. Perdis: design, implementation, and use of a persistent distributed store. In *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*, chapter 18. Springer-Verlag, Feb. 2000.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Symposium on Operating Systems Principles*, pages 29–43, Oct. 2003.

[14] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Symposium on Operating Systems Design and Implementation*, pages 319–332, Oct. 2000.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 388–402, Oct. 2003.

[16] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300, May 1993.

[18] H.-I. Hsiao and D. DeWitt. Chained declustering: a new availability strategy for multiprocessor database machines. In *International Data Engineering Conference*, pages 456–465, Feb. 1990.

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

[20] B. Liskov. Distributed programming in Argus. *Commununications of the ACM*, 31(3):300–312, 1988.

[21] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *European Conference on Object-Oriented Programming*, pages 230–257, June 1999.

[22] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating Systems Design and Implementation*, pages 105–120, Dec. 2004.

[23] P. Mehra and S. Fineberg. Fast and flexible persistence: the magic potion for fault-tolerance, scalability and performance in online data stores. In *International Parallel and Distributed Processing Symposium - Workshop 11*, page 206a, Apr. 2004.

[24] M. A. Olson. The design and implementation of the Inversion File System. In *USENIX Winter Conference*, pages 205–218, Jan. 1993.

[25] RDMA Consortium. http://www.rdmaconsortium.org.

[26] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, Apr. 1990.

[27] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, Feb. 1994.

[28] A. Schiper and S. Toueg. From set membership to group membership: A separation of concerns. *IEEE Transactions on Dependable and Secure Computing*, 3(1):2–12, Feb. 2006.

[29] F. B. Schmuck and J. C. Wyllie. Experience with transactions in QuickSilver. In *Symposium on Operating Systems Principles*, pages 239–253, Oct. 1991.

[30] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *Symposium on Operating Systems Design and Implementation*, pages 29–44, Oct. 2006.

[31] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[32] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9(3):219–228, May 1983.

[33] A. Z. Spector et al. Camelot: a distributed transaction facility for Mach and the Internet — an interim report. Research paper CMU-CS-87-129, Carnegie Mellon University, Computer Science Dept., June 1987.