Sinfonia: A New Paradigm for Building Scalable Distributed Systems

MARCOS K. AGUILERA
Microsoft Research Silicon Valley
ARIF MERCHANT, MEHUL SHAH, and ALISTAIR VEITCH
Hewlett-Packard Laboratories
and
CHRISTOS KARAMANOLIS
VMware

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 new 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**]: Data Structures—*Distributed data structures*

General Terms: Algorithms, Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: Distributed systems, scalability, fault tolerance, shared memory, transactions, two-phase commit

ACM Reference Format:

Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. 2009. Sinfonia: A new paradigm for building scalable distributed systems. ACM Trans. Comput. Syst. 27, 3, Article 5 (November 2009), 48 pages.
DOI = 10.1145/1629087.1629088 http://doi.acm.org/10.1145/1629087.1629088

Authors' addresses: M. K. Aguilera, Microsoft Research Silicon Valley, 1288 Pear Avenue, Mountain View, CA 94043; A. Merchant, M. Shah, A. Veitch, Hewlett-Packard Laboratories, 1501 Page Mill Road, MS 1183, Palo Alto, CA 94304; C. Karamanolis, VMware, 3401 Hillview Avenue, Palo Alto, CA 94304.

DOI 10.1145/1629087.1629088 http://doi.acm.org/10.1145/1629087.1629088

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, management of file data and metadata, cache consistency, and group membership. These protocols are highly nontrivial to develop.

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) (e.g., Amza et al. [1996], Carter et al. [1991], Dasgupta et al. [1991], Li [1988]). 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 [Olson 1993] resulted in an unusable system due to poor performance. Existing DSM systems lack the scalability or fault tolerance required for infrastructure applications. Section 9 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,

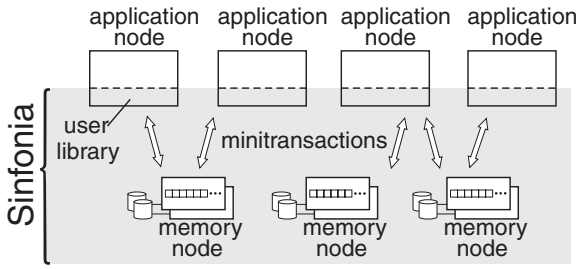

Fig. 1. Sinfonia allows application nodes to share data in a fault tolerant, scalable, and consistent manner.

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, to provide 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 hard to implement in a scalable and fault-tolerant fashion: Implementations 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 [Amir and Stanton 1998], a high-throughput implementation of a group communication service.

The article is organized as follows. We explain our assumptions and goals in Section 2. We describe the design of the Sinfonia service and minitransactions in Section 3. Section 4 explains how Sinfonia is implemented. We then give two applications of Sinfonia. The first application is a cluster file system, described in Section 5. The second application is a group communication service, described in Section 6. Section 7 reports on our evaluation of Sinfonia and its applications.

A discussion follows in Section 8. Section 9 explains related work, and Section 10 concludes.

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 a partition, it is acceptable to pause applications since, most likely, the data center is unusable because some critical service (e.g., a file server) is unreachable. 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 in proportion to system size. In this article, 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 allow parallel execution.* Coupling refers to the interdependence that operations have on each other. By avoiding coupling, we can execute operations in parallel in different machines and thereby obtain scalability by distributing work. We note that data and computation partitioning (common techniques to scale a system) both require elimination of coupling across the partitions. 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. By doing so, we avoid the complexities of a large system with many unreliable components, such as a peer-to-peer system.

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 article, 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 to achieve 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, which implements the key mechanism to access data in Sinfonia, *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 mangled; 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 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. If the transaction's last action is a write then the coordinator can piggyback this

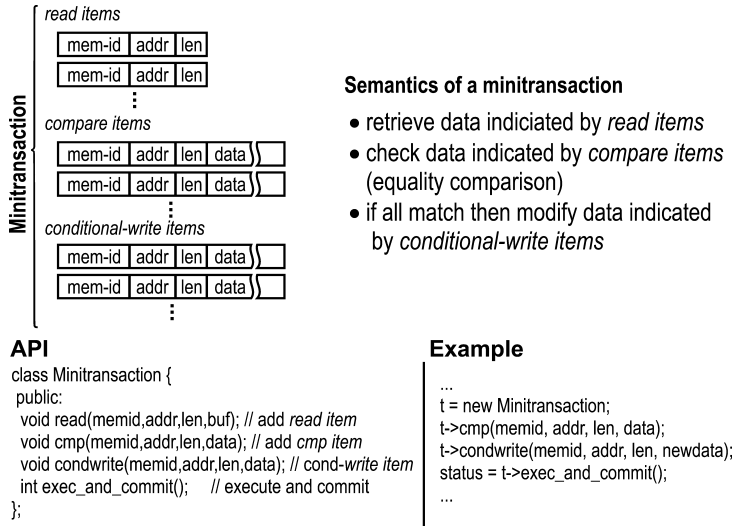

Fig. 2. Minitransactions have read items, compare items, and conditional-write items. Read items are locations to read. Compare items are locations to compare against given values, while conditional-write items are locations to update if all comparisons match. All items are specified before the minitransaction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

last action onto the first phase of two-phase commit. This optimization does not affect the transaction semantics and saves a communication round-trip.

Now suppose the transaction has a conditional write on some location W, which should occur only if some other location C holds some value v. If both locations are in the same memory node then, again, the coordinator can piggyback the action onto the commit protocol, because a single memory node can check if $C = v$ and decide whether to write to W when the transaction commits. It turns out that even if W and C are in different memory nodes, it could be possible to piggyback the action onto the commit protocol, provided that all writes in the transaction are conditional on $C = v$. The idea is to tell the memory node holding C to vote abort if $C \neq v$, so that the writes are aborted.

We designed minitransactions so that the *entire transaction* can be piggybacked onto the commit protocol, and we found that it is still possible to get fairly powerful transactions. Specifically, a minitransaction (Figure 2) consists of a set of *read items*, a set of *compare items*, and a set of *conditional-write items*. Each item specifies a memory node and an address range within that memory node; compare and conditional-write items also include data. Items must be chosen before the minitransaction starts executing. Upon execution, a minitransaction does the following: (1) read the locations specified by the read items, (2) compare the locations in the compare items, if any, against the data in the compare items (equality comparison), (3) if all comparisons succeed, or if there are no compare items, write to the locations in the conditional-write items, and (4) return to the application the locations read in (1) and the results

of the comparisons in (3). Thus, the compare items control whether the minitransaction changes any data or not, while the read and conditional-write items determine what data the minitransaction returns and modifies. What enables efficient execution is that all the writes depend on the same condition, and this condition can be evaluated during two-phase commit; if the condition fails, we omit the writes by essentially aborting the transaction.

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

—*Swap.* A read item returns the old value and a conditional-write item replaces it (here, the conditional-write always happens because there are no compare items).

—*Compare-and-swap.* A compare item compares the current value against a constant; if equal, a conditional-write item replaces it.

—*Atomic read of many locations.* This is done with multiple read items.

—*Acquire a lease.* A compare item checks if a location is set to 0; if so, a conditional-write item sets it to the (nonzero) *id* of the leaseholder and another conditional-write item sets the time of lease.

—*Acquire multiple leases atomically.* This is the same as before, except that there are multiple compare items and conditional-write items. Note that each lease can be in a different memory node.

—*Change data if lease is held.* A compare item checks that a lease is held and, if so, conditional-write items update data.

A frequent minitransaction idiom is to use compare items to validate data and, if data is valid, use conditional-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 conditional-write items in a minitransaction. Figure 11 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 conditional-write items. Such a minitransaction modifies no data, but it returns the result of the comparisons; if they all succeed, the application knows that 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 have more general conditional-update items than writes and more general conditions than equality comparisons. We discuss this extension in Section 8. This extension was not needed for the applications in this article, but it may be useful for other applications.

3.4 Caching and Consistency

Sinfonia does not cache data at application nodes, but minitransactions help 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 access patterns 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) and performance is predictable rather than varying dramatically depending on whether there is a cache hit or not. Managing caches in 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 only nonbacked-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 when minitransactions commit, to ensure data durability. The log can be stored on disk or nonvolatile 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., Budhiraja et al. [1993]), but it is possible to use instead state machines and Paxos [Lamport 1998] 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. We provide more details of how these mechanisms are implemented in Section 4.

Figure 3 shows various Sinfonia modes of operation based on which of the preceding mechanisms are used. Modes differ only on their fault tolerance, cost, space available at each memory node, 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 the applications and experiments in this article, 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. A memory node runs a server process that keeps Sinfonia data and the minitransaction redo-log.

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
Resources taken by a memnode	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]
Space available at a memnode	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

Fig. 3. Trading off fault tolerance for amount of resources and performance. Each column is a mode of operation for Sinfonia. Memnode is an abbreviation for memory node.

Table footnotes:

(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) A disk is reserved just to store the log, to benefit from sequential writes.

(d) If we colocated a memory node with the replica of another memory node, they could share the log disk and the image disk. Doing this would halve the number of disks and hosts shown here.

4.2 Minitransaction Protocol

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 [Skeen and Stonebraker 1983], but we want to avoid the extra phase.

We accomplish this by blocking on participant crashes instead of coordinator crashes. In other words, if a minitransaction starts but some participant of this minitransaction becomes unresponsive (crashes), the minitransaction blocks until the participant recovers. 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 4.3–4.6.

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. To avoid deadlocks, we use a simple scheme: A participant tries to acquire locks without blocking; if it fails (because some lock is already acquired) then it releases all locks it acquired 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 this scheme, the coordinator in phase 1 has to contact participants in series (to ensure lock ordering), which could incur many extra network round-trips.

We now explain in detail the protocol to execute and commit a minitransaction. Recall that a minitransaction has read items, compare items, and conditional-write items (Figure 2). Read items are locations to be read and returned to the application. Compare items are locations to be tested for equality against supplied data. If all comparisons succeed, conditional-write items are locations to be written.

Figure 4 shows protocol's pseudocode. There are two phases. 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 an EXEC-AND-PREPARE message to the participants (memory nodes). When a participant receives such a message, it does the following.

— The participant tries to acquire locks for all the locations that it stores and that are referred by items in the minitransaction. The participant does not block on busy locks. We use reader-writer locks with a word granularity, and we store locked intervals in a range data structure, instead of keeping the state of each location individually, which is inefficient.

— If the participant manages to acquire all locks, it performs the comparisons indicated by the compare items and reads the locations indicated by the read items. If all comparisons succeed, the participant buffers the conditional-writes in a redo-log.

— The participant chooses a vote as follows: If it managed to acquire all locks, 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 conditional-write items, otherwise it ignores them. In either case, the participant releases all locks acquired by the minitransaction. The coordinator never logs any information. (This is unlike the coordinator in standard two-phase commit.) If the minitransaction aborts because locks were busy or because it has been marked as "forced-abort" by the recovery process (explained in Section 4.3), 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 conditional-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.

Participants keep an *uncertain* list of transaction *tid s* that are in the *redo-log* but that are still undecided, a *forced-abort* list of *tid s* that must be aborted, and a *decided* list of finished *tids* and their outcome. These data structures are used for recovery, as explained in the next sections. Figure 5 summarizes the data structures kept by participants.

4.3 Recovery from Coordinator Crashes

If a coordinator crashes during the execution of a minitransaction, it may leave behind locked locations and the minitransaction may have an uncertain outcome: one in which some but not all participants have voted. To fix this problem, we use a recovery protocol, which we now describe.

Since the coordinator and its host may have crashed and may never recover, the recovery protocol is executed at a dedicated management node in Sinfonia. This node periodically probes memory nodes for minitransactions in the *uncertain* list that have not yet committed after a timeout period; if there

Code for coordinator p:

To execute and commit minitransaction ($cmpitems$, $rditems$, $cwritems$)

```
1   tid ← new unique identifier for minitransaction
    { Phase 1 }
2   D ← set of memory nodes referred in cmpitems ∪ rditems ∪ cwritems
3   pfor each q ∈ D do                                   { pfor is a parallel for }
4     send (EXEC&PREPARE, tid, D, π_q(cmpitems), π_q(rditems), π_q(cwritems)) to q
5     { π_q denotes the projection to the items handled by q }
6   replies ← wait for replies from all nodes in D
7   { Phase 2 }
8   if ∀q∈D : replies[q].vote=COMMIT-OK then action ← COMMIT
9   else action ← ABORT                                  { abort }
10  pfor each q ∈ D do send (DECISION, tid, action) to q
11  return (action, replies)                  { does not wait for reply of COMMIT }
```

Code for each participant memory node q:

```
upon receive (EXEC&PREPARE, tid, D, cmpitems, rditems, cwritems) from p do
12  add (tid, D, cmpitems, rditems, cwritems) to uncertain
13  rlocs ← set of locations in cmpitems ∪ rditems but not in cwritems
14  wlocs ← set of locations in cwritems
15  if try-read-lock(rlocs)=fail or try-write-lock(wlocs)=fail
16  then vote ← ABORT-LOCK
17  else if tid ∈ forced-abort then vote ← ABORT-FORCED     { this is needed for recovery }
18  else if cmpitems do not match data then vote ← ABORT-CMP
19  else vote ← COMMIT-OK
20  if vote ∈ {COMMIT-OK, ABORT-CMP} then data ← read rditems
21  else
22    data ← ∅
23    release locks acquired above
24  if vote=COMMIT-OK then add (tid, D, cwritems) to redo-log and add tid to all-log-tids
25  send-reply (tid, vote, data) to p

upon receive (DECISION, tid, action) from p do
26  (D, cmpitems, rditems, cwritems) ← find (tid, ∗, ∗, ∗, ∗) in uncertain
27  if not found then return                  { recovery coordinator executed first }
28  remove (tid, ∗, cmpitems, rditems, cwritems) from uncertain
29  if tid ∈ all-log-tids then decided ← decided ∪ {(tid, action)}   { if tid ∉ all-log-tids, we do
                                                  not need to remember action }
30  if action = COMMIT then apply updates in cwritems
31  release any locks still held for this minitransaction
```

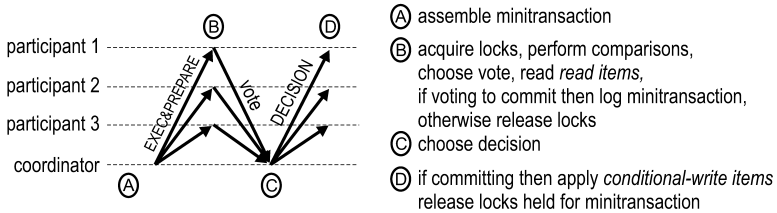

Fig. 4. Protocol for executing and committing minitransactions.

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

Fig. 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.

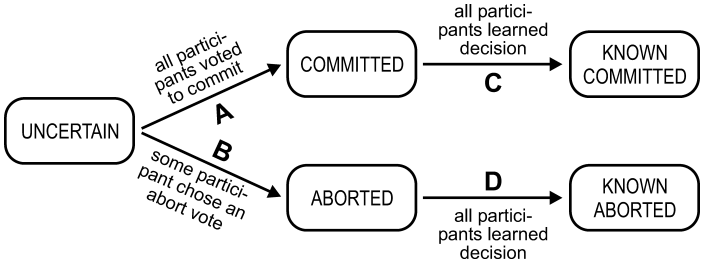

Fig. 6. States of a minitransaction.

are such minitransactions, the management node starts a recovery protocol, described shortly. We use a timeout period of a few seconds, as minitransactions are short-lived and so they are supposed to finish quickly. However, because the minitransaction coordinator may be slow rather than having crashed, we must consider the possibility that the recovery protocol executes concurrently with the minitransaction coordinator. Furthermore, the management node itself could crash during recovery, or it could appear to crash, causing another management node to be launched. This could result in many concurrent executions of the recovery protocol, possibly along with the original coordinator. We must be able to handle these situations.

The recovery protocol is executed by a *recovery coordinator*. The recovery coordinator is given the transaction id *tid* and its set D of participants, and it will try to abort the minitransaction if it has not committed yet. To understand how this is done, consider the possible states of the minitransaction shown in Figure 6. Each box represents a possible transaction *global state*, which reflects collective information from the entire system, instead of the local view of a given process. When a minitransaction is committed (or aborted), we distinguish between the state where this fact is known by all participants or not. Initially, a minitransaction is in the "uncertain" state. The state changes from "uncertain" to "committed" (transition A) when all participants have chosen a commit vote. This transition occurs even before the minitransaction coordinator has learned these votes. The state changes from "uncertain" to "aborted" (transition B) when *some* participant has chosen an abort vote. The "known committed" and "known aborted" states are reached when all participants have received the outcome from the coordinator (transitions C and D).

The key idea of the recovery protocol is that the recovery coordinator tries to produce a B transition if an A transition has not yet occurred. It does so by forcing a participant to choose an abort vote if it has not yet chosen a vote. The detailed protocol is shown in Figure 7 and it has two phases. In phase 1, the

Code for recovery coordinator p:

To recover minitransaction (tid, D) { tid =transaction id, D =minitransaction participants }
{ Phase 1 }
pfor each $q \in D$ **do** { pfor is a parallel for }
  send (REQUEST-ABORT, tid, D) **to** q
$replies \leftarrow$ wait for replies from all nodes in D
{ Phase 2 }
if $\forall q \in D$: $replies[q].vote$=COMMIT-OK **then** $action \leftarrow$ COMMIT
else $action \leftarrow$ ABORT { abort }
pfor each $q \in D$ **do send** (DECISION, tid, $action$) **to** q
return

Code for each participant memory node q:

upon receive (REQUEST-ABORT, tid, D) **from** p **do**
  if $tid \in$ *all-log-tids* **then** $vote \leftarrow$ COMMIT-OK { previously voted to commit }
  else
    if $tid \notin$ *forced-abort* **then** add tid to *forced-abort*
    $vote \leftarrow$ ABORT-FORCED
  send-reply (tid, $vote$) **to** p

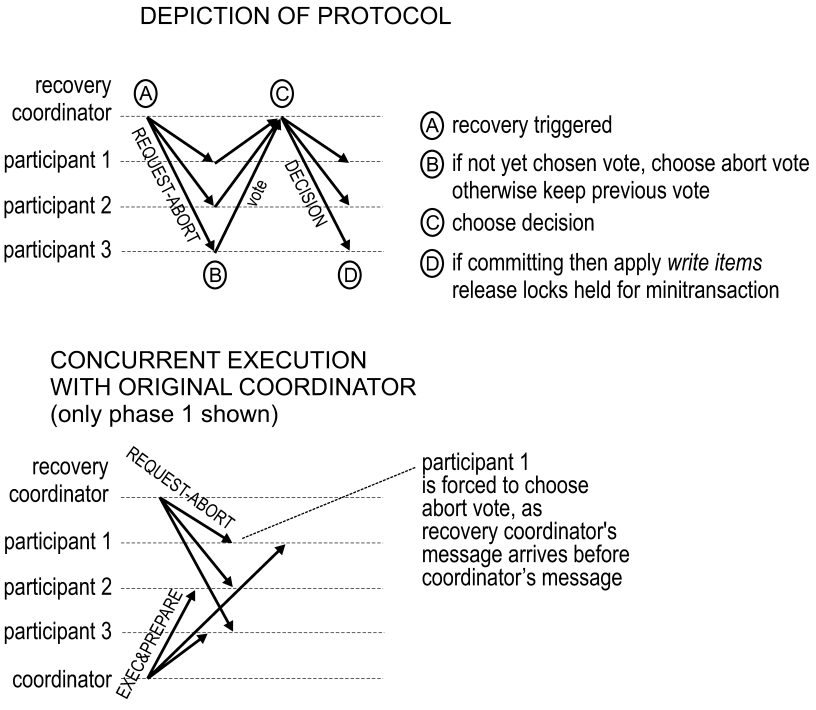

Fig. 7. Protocol for recovering from a coordinator crash.

recovery coordinator sends a REQUEST-ABORT message for transaction *tid* to each participant. Upon receiving this message, if a participant has already chosen the commit vote (*tid* $\in$ *all-log-tids*) the participant keeps its vote. Otherwise, the participant chooses the abort vote and adds *tid* to the *forced-abort* list. This list is kept in stable storage synchronously, and it is used for the participant to remember that *tid* aborted if the transaction's original coordinator subsequently asks the participant to vote (see line 17 in Figure 4). The participant then sends its vote to the recovery coordinator.

In phase 2, the recovery coordinator tells participants to commit if and only if all votes are for committing. The participants react accordingly (the participant code to handle phase 2 messages is in Figure 4).

In the protocol, it is important that each participant processes incoming messages for the same minitransaction, one at a time.

The protocol works even if the recovery coordinator executes concurrently with the original coordinator. In this case, the minitransaction's outcome depends on whether the recovery coordinator's phase 1 message reaches *some* participant before the original coordinator's phase 1 message. If so, the minitransaction will abort (see Figure 7, bottom right).

4.4 Recovery from Participant Crashes

When a participant memory node crashes, the system blocks outstanding minitransactions involving the participant while the participant is offline.[1] If the node loses its stable storage, one must recover the entire system from a backup (Section 4.8 explains how to produce transactionally consistent backups). More frequently, the node is rebooted without losing its stable storage, in which case recovery entails reconstructing the data structures of Figure 5 and synchronizing the node's disk image by replaying the redo-log. During recovery, the node remains unavailable for executing new minitransactions.

The node reconstructs its data structures as follows. It first reads the redo-log from stable storage to extract the set of *tid*s in the log, thereby reconstructing the list *all-log-tids* (see Figure 5). It also determines the minitransactions that were forced to abort, by retrieving the *forced-abort* list from stable storage. This is stage 1 of recovery. In this stage, the node postpones the servicing of all incoming messages. Figure 8 shows the details.

In the next stage, the node determines the transaction outcome of each *tid*. This information is in the list *decided*, which was kept in stable storage *asynchronously*, so its recent contents were possibly lost during the crash. As a result, there may be *tid*s in *all-log-tids* whose outcome is not in *decided*. For those *tid*s, the node queries the set D of memory nodes that participated in the minitransaction, to learn their votes and recompute the outcome. We use the REQUEST-ABORT message to perform this querying, because it is possible that some participants in the set D have not yet voted and, in this case, we want to force them to vote abort (we cannot force them to vote commit because they may

[1]This is unlike two-phase commit for database systems, where the coordinator may consider a dead participant as voting "abort".

Code for recovering node p:

```
45  stage ← 1                                          { Stage 1: reconstruct all-log-tids }
46  all-log-tids ← ∅
47  uncertain ← ∅
    { reconstruct all-log-tids from redo-log }
48  for each (tid, D, cwritems) ∈ redo-log do add tid to all-log-tids
49  retrieve forced-abort from disk                    { reconstruct forced-abort from disk }
50  stage ← 2                                          { Stage 2: reconstruct decided }
51  retrieve decided from disk                           { start with what is on disk }
52  for each tid ∈ all-log-tids do
53    if no (tid, ∗) in decided then                     { need further reconstruction }
54      find D such that (tid, D, ∗) ∈ redo-log { retrieve participants D of tid from redo-log }
55      pfor each q ∈ D do
56        send (REQUEST-ABORT, tid, D) to q            { query vote, forcing abort if none }
57      replies ← wait for replies from all nodes in D
58      if ∀q∈D : replies[q].vote=COMMIT-OK then action ← COMMIT { determine outcome }
59      else action ← ABORT
60      add (tid, action) to decided
61  stage ← 3                                          { Stage 3: replay log }
62  for each (tid, D, cwritems) ∈ redo-log in order do
63    if (tid, COMMIT) ∈ decided then apply updates in cwritems to disk image
64  stage ← 4                                          { recovery done }
```

Stage	Processing of incoming messages
1	None
2,3	REQUEST-ABORT messages only
4	All

Fig. 8. Protocol executed by a memory node when it restarts after crashing.

never have seen the minitransaction, so they would not know what to commit if the minitransaction commits). This is stage 2 of recovery. In this stage, the node must service incoming REQUEST-ABORT messages, to avoid deadlocks when multiple nodes recover simultaneously. However, the node otherwise remains offline by postponing the servicing any other messages.

In the next stage, the node replays its redo-log to update the disk image, which may be lagging behind. Replaying must be idempotent, because the node may start replaying the log and crash in the middle, so that subsequently it needs to replay the log again. To get idempotency, replay does not recompute the compare items of a minitransaction (those items are not even stored in the log), but rather simply applies the conditional-write items to the disk image if the minitransaction outcome was "commit". This is stage 3 of recovery. After this stage, recovery is done and the node can finally resume processing of incoming messages.

4.5 Optimized Recovery from Crash of the Whole System

When many memory nodes or the whole system crashes and restarts (e.g., due to a power failure), 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: To avoid too many queries of votes in stage 2 of recovery (see Figure 8), memory nodes send each other their list *all-log-tids* of *tid*s for which they chose a commit vote. Thus, in stage 2, a node only needs to query another node for *tid*s that: (a) are not in *decided* and (b) are not in the *all-log-tids* of some of its participants. This greatly reduces the number of queries done, but still there can be many queries for recent minitransactions that aborted. Another optimization batches together all such queries to the same memory node, so that in stage 2, the recovering node only sends one message per memory node, instead of one message per minitransaction per memory node.

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.

4.6 Garbage Collection

A minitransaction produces data that must be subsequently purged (garbage collected), including entries in the redo-log and in the other data structures of Figure 5.

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.

The redo-log is temporary storage for updates of minitransactions that are ultimately applied to the disk image if the minitransaction commits. Thus, one might expect that once the disk image reflects the writes of a committed minitransaction, its log entries can be garbage collected. Doing this, however, would be problematic because the redo-log also indicates the set of minitransactions that received a commit vote, and this knowledge is needed when another node recovers. For example, suppose a minitransaction *tid* commits but some memory node q has not yet applied *tid's* writes to its disk image. If all nodes crash and recover, then q forgets the outcome of *tid* and, during recovery, q needs to see *tid* at the redo-log of other memory nodes to determine that *tid* committed. Thus, because of q, the other memory nodes cannot garbage collect *tid* even though they may be done with *tid*. This motivates the following garbage collection 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*.

To implement the above rule, a memory node p must inform every other memory node q of the minitransactions *tids* that p has applied to its disk image and that q participated in. For efficiency, rather than having a memory node directly send this information to other memory nodes, we let the management

Consider a minitransaction *tid* that starts executing at a coordinator, but the coordinator is very slow. The coordinator contacts some of the participants in the first phase of the minitransaction protocol, and those participants vote to commit, but before the coordinator contacts the last participant, denoted ℓ, the system considers the coordinator to have crashed and starts the recovery procedure of Section 4.3. The recovery coordinator executes quickly and faster than the coordinator, and contacts participant ℓ before the coordinator. This forces ℓ to add *tid* to its *forced-abort* list, and it causes *tid* to abort. Next, memory nodes garbage collect *tid* from their redo-log. Now suppose that *tid* were garbage collected from the *forced-abort* list of participant ℓ. We will show that this can cause a problem. Subsequently, the slow minitransaction coordinator continues executing and finally asks ℓ to vote. Because ℓ eliminated all traces of *tid*, ℓ thinks that *tid* is a new minitransaction, and votes to commit. Now, the slow coordinator has received a commit vote from all participants, so it picks the commit decision and notifies the application. However, the participants have garbage collected *tid* from their logs and so they are unable to commit the minitransaction. This scenario violates the semantics of commit.

Fig. 9. Problematic scenario if *forced-abort* list were garbage collected in the obvious way, with the redo-log.

node collect this information and relay it to the appropriate memory nodes in batches.

Besides the redo-log, the other data structures in Figure 5 are garbage collected as follows. The *uncertain* list, *decided* list, and *all-log-tids* 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 can corrupt the system. Figure 9 shows one problematic scenario, which occurs when a minitransaction's 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. (The reason for using two epochs instead of one is to avoid aborting minitransactions during epoch boundaries.) 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 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 after it learns the current epoch.

4.7 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 conditional-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.

4.8 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: (a) updating the redo-log and (b) keeping in memory any newly committed data, for the use of minitransactions that read such data.[2] 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, the backup procedure must start the previous scheme "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 preceding 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 could cause the backup procedure to starve in a large busy system. Therefore, the backup procedure uses instead blocking lock requests, and issues the requests in memory node order to avoid a deadlock with another backup procedure.

4.9 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

[2]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 and kept in memory to buffer updates of new minitransactions.

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 avoid 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 [Lamport 1998] instead of primary-copy.

4.10 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 keep the contents of a memory node on disk and the address space of a memory node is as large as the disk (not limited by RAM).

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.

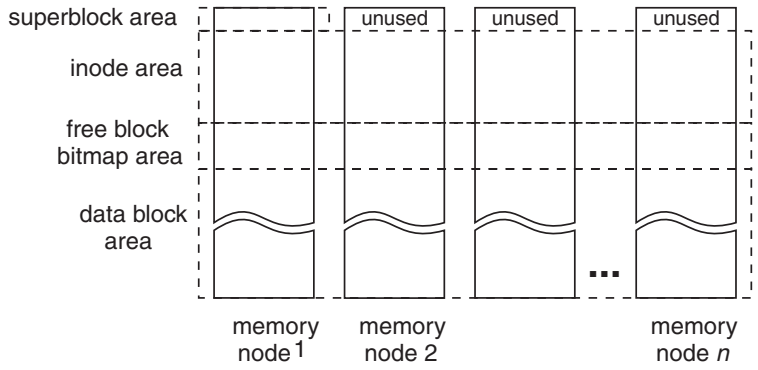

Fig. 10. Data layout for SinfoniaFS. These data structures are on disk, as Sinfonia runs in mode LOG or LOG-REPL.

5.1 Data Layout

Data layout (Figure 10) 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, such as 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 to inodes at any memory node. *Data blocks* of 16KB 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 4GB 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 before the file system takes permanent actions, such as modifying file system data or returning data to the user. Validation occurs via compare items in a minitransaction, to check that the cached version matches what is in Sinfonia. For example, Figure 11 shows the implementation of NFS's setattr, which changes attributes of an inode. The *iversion* is the current version of the inode, incremented every time the inode changes. The compare item in line 6 ensures that the minitransaction only commits if the cached version matches what is in Sinfonia. If the minitransaction

```
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;
        // cmp item to check inode iversion
6       t->cmp(MEMNODE(inode), ADDR_IVERSION(inode), LEN_IVERSION,
                &inode->iversion);
        // condwrite item to update attributes
7       t->condwrite(MEMNODE(inode), ADDR_INODE(inode), LEN_INODE,
                    &newAttributes);
        // condwrite item to bump iversion
8       t->condwrite(MEMNODE(inode), ADDR_IVERSION(inode), LEN_IVERSION,
                    &newiversion);
9       status = t->exec_and_commit();
10      if (status == fail) ...                            // reload inodeNumber into cache
11    } while (status == fail);
12  }
```

Fig. 11. Function to change an inode's attribute in SinfoniaFS, simplified and without error checking. Lines 5–9 show the 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.

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.

In addition to an *iversion*, inodes also have an *igeneration* number that increases when the inode is deleted. This extra version number is needed because some operations must fail if the *igeneration* has changed but should not fail if the *iversion* has changed. For example, if a node tries to delete a file, but another node deletes it first and creates a new file with the same inode number, then the first delete should fail. The *igeneration* can detect this, while the *iversion* can only tell that a cached entry is stale.

Inodes also store a *dversion*, which is incremented every time the file's *data* is changed (that is, when the file is written). Like *iversions* and *igenerations*, *dversions* are tested in minitransactions to validate cached data that is being acted upon. To validate the free-block bitmap, we use the bitmap's own content, instead of version numbers. For example, to allocate a new block that a node believes to be free (according to its cached free-block bitmap), a minitransaction includes a compare item to check that the block is indeed free. Because a minitransaction cannot test bits individually, we check the whole word containing the bit. This may cause the minitransaction to occasionally abort and be retried even if the block is free, if the other bits do not match. This type of false sharing could slightly affect performance, but only rarely.

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

```
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 conditional-write items
4. if minitransaction fails, check the reason and, if appropriate,
   reload cache entries and retry, or return an error indicator.
```

Fig. 12. One minitransaction does it all: The above template shows how SinfoniaFS implements any NFS function with 1 minitransaction.

```
1. if file's inode or chaining list are not cached then load them into cache
2. find a free block in the cached free-block bitmap
3. issue a minitransaction that checks iversion, igeneration, and dversion of
   the cached inode, checks the free status of the new block, updates the
   inode's iversion and dversion, 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 igeneration does not match
   then return stale filehandle error
5. if minitransaction failed because the iversion or dversion do not match
   then reload inode cache entry and retry
6. if minitransaction failed because block is not free then reload appropriate part
   of free-block bitmap and retry
7. if minitransaction failed for another reason then retry it
8. else return success
```

Fig. 13. Implementing write that appends to a file, allocating a new block.

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 12 shows the general template to do this, and Figure 13 shows a more elaborate example than Figure 11.

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

To balance load, SinfoniaFS uses load information from Sinfonia to decide where to write new data. (It is also possible to migrate files between memory nodes to balance load, but we have not yet designed this functionality.) We currently use a 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 compared to 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 but it fails (because another cluster node allocates it first), the cluster node retries the allocation using a random free inode or block within that memory node.

6. APPLICATION: GROUP COMMUNICATION

Intuitively, a group communication service [Birman and Joseph 1987; Chockler et al. 2001] 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 Défago et al. [2004] 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 abort 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 this design as shown in Figure 14. 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) involves a minitransaction that may contend with others and abort. If the minitransaction aborts, 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.

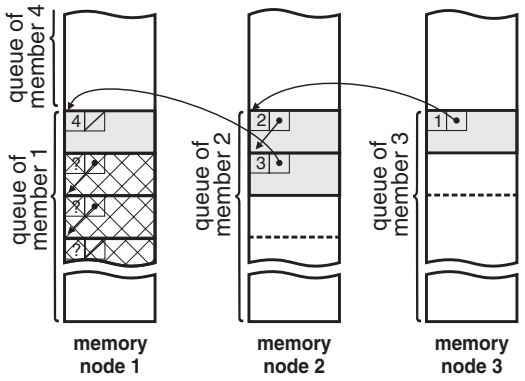

Fig. 14. Basic design of SinfoniaGCS. Gray messages were successfully broadcast: They are threaded into the global list. Cross-hatched messages are waiting to be threaded into the global list, but they are threaded locally.

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 [Schiper and Toueg 2006]. Rather, a member can use a separate failure detection service [Chandra and Toueg 1996] 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, as follows. It uses a minitransaction to read all the last received pointers for its queue. Then, it frees all messages older than and including the oldest last received pointer. Since the queue is dedicated to a member, the deallocation does not cause contention.

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 14), 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. Intrarack bisection bandwidth is ≈14Gbps, while interrack bisection bandwidth is ≈6.5Gbps. Each machine has two 2.8GHz Intel Xeon CPUs, 4GB of main memory, and two 10000 rpm 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 identically 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 the various Sinfonia modes of operation 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 [Hsiao and DeWitt 1990].

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 15. Figure 16 shows throughput and latency for a workload that repeatedly issues minitransactions,

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.

Fig. 15. Berkeley DB setup for all experiments.

where each minitransaction had 6 items[3] arranged as 3 compare-and-swaps chosen randomly over a range of 50 000 items. (Recall that compare-and-swap is implemented using a compare item and a conditional-write item on the same location.) We varied the number of outstanding minitransactions at the application node (number 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. With a few threads, Berkeley DB has similar performance to Sinfonia-LOG and Sinfonia-LOG-REPL because in this case 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 locking technique which, at a minimum, locks a leaf page for every modification, blocking other transactions that access different locations in the same page.

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 A: Nonbatched 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.

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

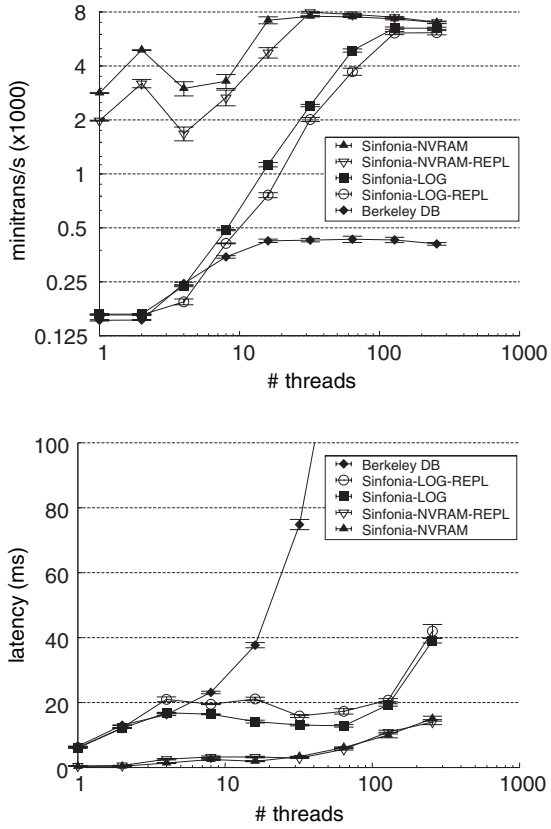

Fig. 16. Performance of Sinfonia with 1 memory node.

—*System B: 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 C: 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 B, and it is how Sinfonia executes minitransactions involving many memory nodes.

—*System D: 1PC combined.* The minitransaction executes within the first phase of one-phase commit. This saves another network round-trip compared to system C, 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

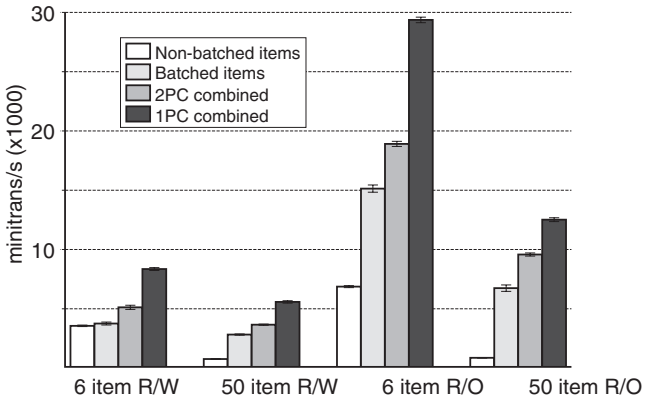

Fig. 17. Performance with various combinations of techniques.

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 this mode, items are cached in RAM at memory nodes, and log writes are synced only when committing the minitransaction. We set up the non-Sinfonia systems (Systems A and B) to behave that way as well.

Figure 17 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 nonbatched system. For minitransactions involving just one memory node, running 1PC improves throughput over 2PC by 1.3–1.6x, 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 18 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,

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

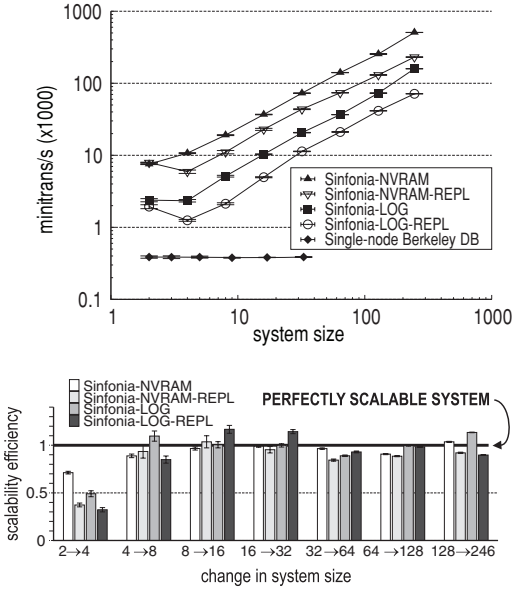

Fig. 18. Sinfonia scalability.

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. Intuitively, this is the cost of distributing the system. We also included a line for a Berkeley DB configuration (see Figure 15) 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 18 shows scalability efficiency for each Sinfonia mode, comparing it with a perfectly scalable system whose efficiency is always 1. As can be

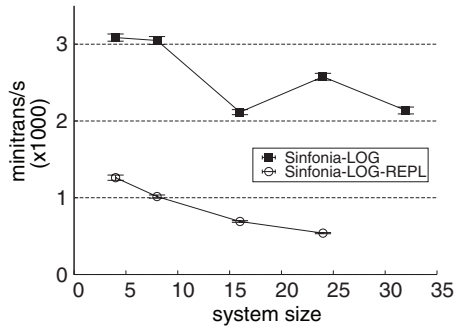

Fig. 19. Effect of minitransaction spread on scalability.

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.

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 19 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 18.

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 per memory node but much smaller incremental cost with the number of items, and so spreading a minitransaction 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 (not shown) 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 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 20 shows the result on the top three lines labeled “cas” (compare-and-swap).

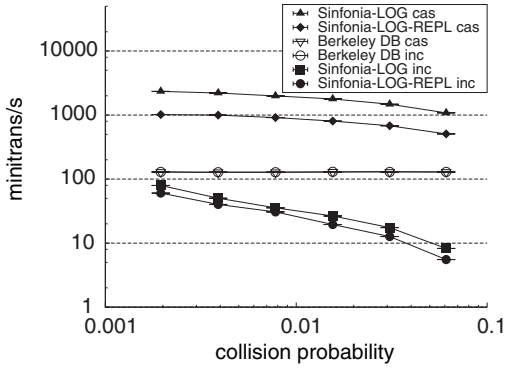

Fig. 20. Effect of minitransaction overlap on performance.

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: (a) validate the cache with compare items, and (b) write the incremented values with conditional-write items. 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 20 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, compare, and conditional-write items, as we explain in Section 8.2. 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 21 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.

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

Fig. 21. Comparison of development effort.

We also report qualitatively on our experience in building SinfoniaFS and SinfoniaGCS, including advantages and drawbacks. 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 22, because otherwise they execute too quickly.

Figure 23 shows the benchmark results for SinfoniaFS compared to a Linux Fedora Core 3 NFS server, where smaller numbers are 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

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 1 MB file in 8KB buffers 10 times
5b	read the 1 MB 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

Fig. 22. Connectathon NFS testsuite modified for 10x work.

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

Fig. 23. Results of modified Connectathon NFS Testsuite.

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 macrobenchmark with a more balanced mix of data and metadata operations. We modified the Andrew benchmark to use as input the

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

Fig. 24. Results of modified Andrew benchmark.

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 24 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 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 25 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.[4] 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

[4]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.

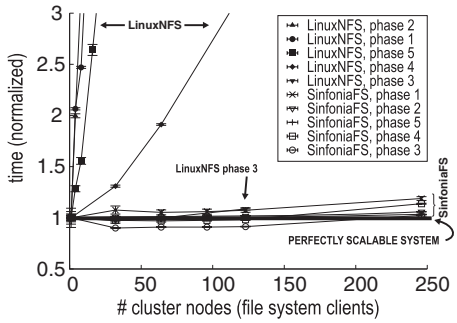

Fig. 25. Results of Andrew as we scale the system and the workload together. The x-axis is the number of nodes running Andrew simultaneously. The y-axis is the duration of each phase relative to a system with 1 client. Connectathon had similar results.

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 [Amir and Stanton 1998], version 3.17.4 (January 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 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 Spread 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

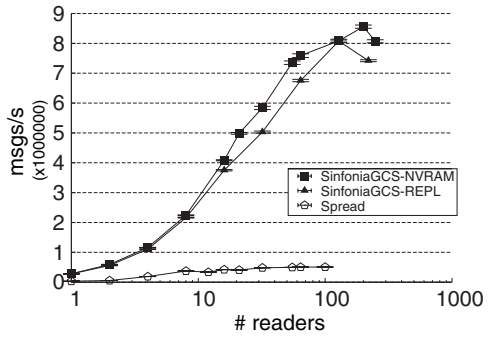

Fig. 26. SinfoniaGCS base performance as we vary the number of readers. There are 8 writers, and 8 memory nodes or Spread daemons.

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 26). (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 27). 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.

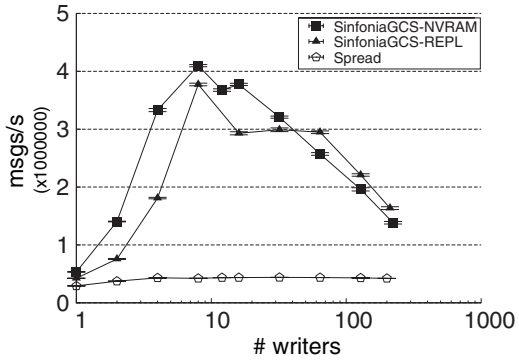

Fig. 27. SinfoniaGCS base performance as we vary the number of writers. There are 16 readers, and 8 memory nodes or Spread daemons.

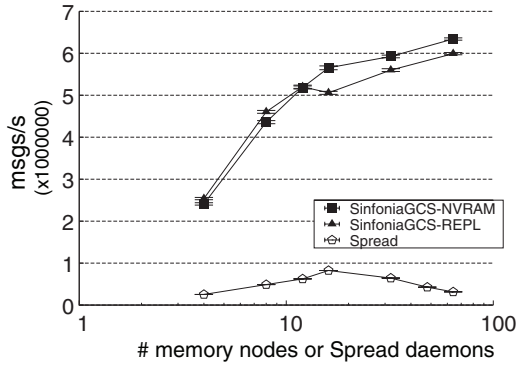

Fig. 28. SinfoniaGCS scalability as we vary the number of memory nodes. There are 32 readers and 64 writers.

7.3.2 *Results: Scalability.* In the third experiment, we observe the scalability as we increase the number of memory nodes (Figure 28). Since the intrarack bisection bandwidth is higher than the interrack 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 increase the memory nodes to more than 16, the memory nodes can handle the traffic and eventually become underutilized. We will see in the next experiment that a higher aggregate throughput can be obtained with more readers and writers. As in the other cases, the throughput exceeds the Spread throughput considerably.

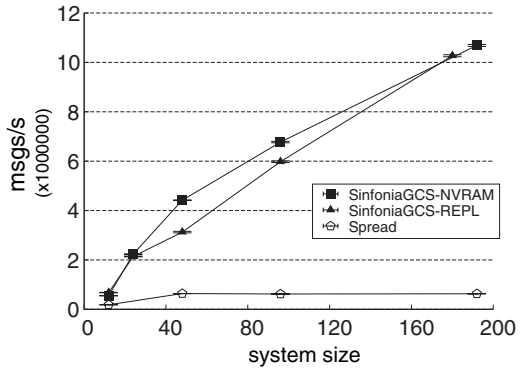

Fig. 29. SinfoniaGCS scalability as we vary total system size (total number of readers, writers, and memory nodes or daemons).

In the fourth experiment, we observe the scalability with total system size (Figure 29). 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 the per-queue metadata increases (e.g., because of the last received pointers and last threaded pointers; see Sections 6.2 and 6.3).

8. DISCUSSION

We now discuss some possible future work and extensions to Sinfonia.

8.1 Disk Arrays and Memory Nodes

A memory node is essentially a storage service connected to the network. It does not need much computational power, but it can benefit from high reliability, large storage capacity and bandwidth, and large network bandwidth. These features are generally offered by state-of-the-art network-attached disk arrays. We think it should be possible to implement the memory node functionality inside disk arrays, and thereby use them as components of Sinfonia. We now explain why this is interesting, and then explain what it entails.

Currently, network-attached disk arrays can be physically accessed by many hosts in the network, but logically it is very difficult for different hosts to share a writable disk volume, for example, to implement a cluster file system. This is because the simple read-write service provided by disk arrays is not sufficient to orchestrate concurrent accesses by different hosts. If the disk array were a memory node, however, hosts could use Sinfonia minitransactions to consistently share the volume.

Another use of minitransactions is to coordinate access to an *aggregation* of disk arrays (memory nodes). Minitransactions can provide atomic reads, writes, or conditional writes *across* different disk arrays. This ability allows clustered applications to pool multiple disk arrays transparently, and thereby obtain enormous storage space and bandwidth. For example, SinfoniaFS could serve as a cluster file system that spans many disk arrays.

We believe it is not hard to extend disk arrays to memory nodes, because most of the complexity in Sinfonia is at the application nodes (in the Sinfonia user library), not at the memory nodes. More precisely, to be a memory node, disk arrays must be extended with the following functionality:

—maintain ranges of locked locations;
—buffer the writes of a minitransaction in a redo-log;
—besides the redo-log, implement the other data structures of Figure 5, which are just lists of transaction ids;
—respond to the requests of the minitransaction protocol (Section 4.2) and recovery protocol (Section 4.3); and
—implement the memory node recovery protocol (Section 4.4).

To summarize, we believe network-attached disk arrays would benefit from having a concurrency control mechanism that allows volumes to be pooled and shared among multiple hosts. Sinfonia's minitransaction is such a mechanism, and we believe it is feasible to modify disk arrays to support minitransactions. This would be an interesting direction to investigate further as future work.

8.2 Extending Minitransactions

We showed that minitransactions are powerful, as we could use them to build very different applications: a cluster file system and a group communication service. However, as we saw in Section 7.1.4, sometimes it is not efficient to use minitransactions to execute some operations, such as atomic increment, especially under high contention. In this section, we extend minitransactions to address this limitation. In particular, we allow minitransactions to have more general conditional tests (besides equality comparisons) and updates (besides just writes). For example, an extended minitransaction can directly perform an atomic increment, without having to use optimistic concurrency control as in Section 7.1.4. This broader functionality can be incorporated while still being able to execute the minitransaction as part of the commit protocol, as we shall see.

Recall that minitransactions consist of read, compare, and conditional-write items. We can replace conditional-writes item in minitransactions with more general *conditional-update items*, which carry arbitrary functions that modify the state of a memory node. Formally, a conditional-update item specifies a memory node and a function that maps state to state, where state refers to the contents of all locations of a memory node.

Similarly, we can replace read items with *retrieve items*, which carry functions to extract data from a memory node (formally, a function from state to a

bit-string). For example, a retrieve item could compute the average of a variable-length list of numbers stored in a memory node.

We can also replace compare items with *check items*, which carry functions to check a condition in a memory node and return a boolean (formally, a function from state to a boolean). For example, a compare item could check if a location holds a number greater than 25.

In every case, each function must be local to a single memory node. However, as usual, a minitransaction could have multiple items, so the minitransaction as a whole can operate on many memory nodes.

The semantics of minitransaction are now as follows: (1) The retrieve item functions are applied (at their respective memory nodes) and the results are returned the application, (2) the check item functions are applied and the results are checked to be all true, (3) if so, the conditional-update item functions are applied to change the state of the memory nodes.

Theoretically, the functions in the retrieve, check, and conditional-update items can be shipped with the minitransaction. Practically, it is more efficient if an application uses some fixed set of functions that are compiled into the memory nodes, so that at runtime only the function name is sent over the network. For example, the memory nodes could store a function to increment a location, so that a minitransaction need only provide the function name. A function generally needs some auxiliary parameters (e.g., the location to be incremented and the word size). However, to be able to handle functions uniformly, we assume that all functions depend only on the memory node state, and we treat these auxiliary parameters as part of the function name. Thus, in the previous example, we have the function "increment location 0 (with word size 1)".

Theoretically, a function can depend on the entire state of a memory node. Practically, for efficiency reasons, it should depend on a small number of locations. We denote by $depend(f)$ the set of locations on which f depends. For example, the "increment location 0" function must read the contents of location 0, so $depend(f) = \{0\}$.

Similarly, theoretically functions in conditional-update items can modify the entire state of a memory node (these functions map the old memory node state to a new memory node state, which could be entirely different). But in practice, these functions should change only a few locations. Given such a function f, we denote by $change(f)$ the set of locations that can be changed by f. For the "increment location 0" function, $change(f) = \{0\}$.

It is easy to modify the protocol of Figure 4 to work with generalized minitransactions, as follows.

—The coordinator code remains mostly the same, except that it sends the retrieve, check, and conditional-update items to participants instead of read, compare, and conditional-write items.

—A participant must read-lock the locations $depend(f)$ for all its retrieve, check, and conditional-update items f, instead of locations in its read and compare items.

—A participant must write-lock the locations $change(f)$ for all its conditional-update items f.

—Instead of equality comparisons, a participant uses the check functions to determine its vote. Instead of the read items, a participant uses the retrieve functions to determine data to be returned. Instead of conditional-write items, a participant uses conditional-update functions to compute the changes to the memory node. These changes are stored in the log as a set of locations modified and their new values. In this manner, replaying the log remains an idempotent operation, which is a key property needed by Sinfonia's recovery scheme.

To summarize, minitransactions can be extended in a way that still permits efficient execution using the protocol of Figure 4 with some simple modifications. Currently, this extension is conceptual but as future work it would be interesting to explore further its applications.

9. RELATED WORK

Services to help developers build large distributed systems include GFS [Ghemawat et al. 2003], Bigtable [Chang et al. 2006], Chubby [Burrows 2006], and MapReduce [Dean and Ghemawat 2004]. 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-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 datasets 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 [Herlihy and Moss 1993]. Original schemes required hardware changes, but subsequent work [Shavit and Touitou 1995] proposed a software implementation; more efficient proposals appeared later [Herlihy et al. 2003; Harris and Fraser 2003]. These systems are all targeted at multiprocessor machines rather than distributed systems.

Distributed shared memory (DSM) (e.g., Amza et al. [1996], Carter et al. [1991], Dasgupta et al. [1991], Li [1988]) 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., Fakler et al. [2005]) 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 [Fakler et al. 2005], 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 raytracing and interactive 3D worlds.

Perdis [Ferreira et al. 2000] 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 [Liskov et al. 1999] 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 [Sears and Brewer 2006] 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 [Chao et al. 1992], which provided multisector atomic writes and the ability to revoke tentative writes; however, all the disks in Mime were accessed through a single controller.

Pond [Rhea et al. 2003] is a prototype of OceanStore, a distributed system for storing data objects in a wide-area network. Objects in OceanStore are versioned, where each update generates a new version. An update consists of a list of predicates associated with actions, inspired by the Bayou system [Demers et al. 1994]. Details are omitted in the Pond paper [Rhea et al. 2003], but some are given in Kubiatowicz et al. [2000]: To apply an update on an object, the system finds the first predicate that evaluates to true, and then atomically applies to the object the actions associated with that predicate. This update model was used to build a file system on top of Pond. However, the paper does not explain how to implement operations that must atomically change multiple objects stored in different hosts, such as an atomic rename of a file across directories. These operations require atomic updates of multiple objects, which requires a fault-tolerant distributed commit protocol, but such a protocol is not available in Pond or OceanStore.

Paxos Commit [Gray and Lamport 2006] is a nonblocking commit protocol, whose basic ideas are: (1) to replace participants with state machines running Paxos, (2) to consider a transaction to be committed if and only if all the state machines have voted to commit, and (3) various optimizations to save messages. We use an idea similar to (2) in Sinfonia's two-phase commit protocol: We consider a transaction to be committed if and only if all the participants have voted to commit.

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 [Liskov 1988], QuickSilver [Schmuck and Wyllie 1991], and Camelot [Spector et al. 1987]. 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 [Satyanarayanan et al. 1990] 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 [Satyanarayanan et al. 1994], 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 [RDMA Consortium]. RDMA could be used in a more efficient implementation of Sinfonia than using message-passing as we do. Persistent Memory [Mehra and Fineberg 2004] 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 [MacCormick et al. 2004] proposes a cluster file system built 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 it does not have atomic transactions. The Inversion File System [Olson 1993] is built over a Postgres database; this is a fairly complex abstraction, and the performance of the file system was poor.

Gribble et al. [2000] 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.

10. CONCLUSION

We proposed a new paradigm for building scalable distributed systems, based on streamlined minitransactions over unstructured data. This paradigm is quite general, given the very different applications that we built with it: a cluster file system and a group communication service. Recent work has additionally shown how to use the paradigm to build a scalable distributed B-tree [Aguilera et al. 2008]. 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 shared data structures.

ACKNOWLEDGMENTS

We are grateful to T. Kelly and J. Mogul for helpful discussions. We are also grateful to X. Li, S. Perl, J. Wiener, J. J. Wylie, and the anonymous reviewers for many comments that helped us improve the article.

REFERENCES

AGUILERA, M. K., GOLAB, W., AND SHAH, M. 2008. A practical scalable distributed B-tree. *Proc. VLDB Endowment 1*, 1, 598–609.

AMIR, Y. AND STANTON, J. 1998. The Spread wide area group communication system. Tech. rep. CNDS-98-4, The Johns Hopkins University.

AMZA, C. COX, A., DWARKADAS, S., KELEHER, P., LU, H., ET AL. 1996. Treadmarks: Shared memory computing on networks of workstations. *IEEE Comput. 29*, 2, 18–28.

BIRMAN, K. P. AND JOSEPH, T. A. 1987. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Symposium on Operating System Principles*. 123–138.

BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F. B., AND TOUEG, S. 1993. The primary-backup approach. In *Distributed Systems*, S. J. Mullender, Ed. Addison-Wesley, Chapter 8.

BURROWS, M. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 335–350.

CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. 1991. Implementation and performance of Munin. In *Proceedings of the Symposium on Operating Systems Principles*. 152–164.

CHANDRA, T. D. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2, 225–267.

CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. 2006. BigTable: A distributed storage system for structured data. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 205–218.

CHAO, C., ENGLISH, R., JACOBSON, D., STEPANOV, A., AND WILKES, J. 1992. Mime: A high performance storage device with strong recovery guarantees. Tech. rep. HPL-CSP-92-9, HP Laboratories.

CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. 2001. Group communication specifications: A comprehensive study. *ACM Comput. Surv. 33*, 4, 1–43.

DASGUPTA, P., LEBLANC, R. J. JR., AHAMAD, M., AND RAMACHANDRAN, U. 1991. The Clouds distributed operating system. *IEEE Comput. 24*, 11, 34–44.

DEAN, J. AND GHEMAWAT, S. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 137–150.

DÉFAGO, X., SCHIPER, A., AND URBÁN, P. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*, 4, 372–421.

DEMERS, A., PETERSEN, K., SPREITZER, M., TERRY, D., THEIMER, M., AND WELCH, B. 1994. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*. 2–7.

FAKLER, M., FRENZ, S., GOECKELMANN, R., SCHOETTNER, M., AND SCHULTHESS, P. 2005. Project Tetropolis—Application of grid computing to interactive virtual 3D worlds. In *Proceedings of the International Conference on Hypermedia and Grid Systems*.

FERREIRA, P., SHAPIRO, M., BLONDEL, X., FAMBON, O., GARCIA, J., ET AL. 2000. Perdis: Design, implementation, and use of a persistent distributed store. In *Recent Advances in Distributed Systems*. Lecture Notes in Computer Science, vol. 1752. Springer, Chapter 18.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. 2003. The Google file system. In *Proceedings of the Symposium on Operating Systems Principles*. 29–43.

GRAY, J. AND LAMPORT, L. 2006. Consensus on transaction commit. *ACM Trans. Datab. Syst. 31*, 1, 133–160.

GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. 2000. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 319–332.

HARRIS, T. AND FRASER, K. 2003. Language support for lightweight transactions. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. 388–402.

HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, W. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Symposium on Principles of Distributed Computing*. 92–101.

HERLIHY, M. AND MOSS, J. E. B. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*. 289–300.

HSIAO, H.-I. AND DEWITT, D. 1990. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of the International Data Engineering Conference*. 456–465.

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. OceanStore: An architecture for global-scale persistent storage. *ACM SIGPLAN Not. 35*, 11, 190–201.

LAMPORT, L. 1998. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2, 133–169.

LI, K. 1988. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing*. 94–101.

LISKOV, B. 1988. Distributed programming in Argus. *Comm. ACM 31*, 3, 300–312.

LISKOV, B., CASTRO, M., SHRIRA, L., AND ADYA, A. 1999. Providing persistent objects in distributed systems. In *Proceedings of the European Conference on Object-Oriented Programming*. 230–257.

MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. 2004. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 105–120.

MEHRA, P. AND FINEBERG, S. 2004. Fast and flexible persistence: The magic potion for fault-tolerance, scalability and performance in online data stores. In *Proceedings of the International Parallel and Distributed Processing Symposium - Workshop 11*. 206a.

OLSON, M. A. 1993. The design and implementation of the Inversion File System. In *Proceedings of the USENIX Winter Conference*. 205–218.

RDMA CONSORTIUM. http://www.rdmaconsortium.org.

RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. 2003. Pond: The OceanStore prototype. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 1–14.

SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput. 39*, 4, 447–459.

SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. 1994. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst. 12*, 1, 33–57.

SCHIPER, A. AND TOUEG, S. 2006. From set membership to group membership: A separation of concerns. *IEEE Trans. Depend. Secure Comput. 3*, 1, 2–12.

SCHMUCK, F. B. AND WYLLIE, J. C. 1991. Experience with transactions in QuickSilver. In *Proceedings of the Symposium on Operating Systems Principles*. 239–253.

SEARS, R. AND BREWER, E. 2006. Stasis: Flexible transactional storage. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 29–44.

SHAVIT, N. AND TOUITOU, D. 1995. Software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*. 204–213.

SKEEN, D. AND STONEBRAKER, M. 1983. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Engin. 9*, 3, 219–228.

SPECTOR, A. Z., THOMPSON, D., PAUSCH, R. F., EPPINGER, J. L., DUCHAMP, D., DRAVES, R., DANIELS, D. S., AND BLOCH, J. J. 1987. Camelot: A flexible and efficient distributed transaction processing facility for Mach and the Internet—An status report. Res. paper CMU-CS-87-129, Computer Science Department, Carnegie Mellon University.

Received December 2008; revised April 2009; accepted April 2009