Building distributed applications using Sinfonia

Marcos K. Aguilera, Christos Karamanolis, Arif Merchant, Mehul Shah, Alistair Veitch
Information Services and Process Innovation Laboratory
HP Laboratories Palo Alto
HPL-2006-147
October 18, 2006*

distributed systems, scalability, fault tolerance

We present Sinfonia, a data sharing service that simplifies the design and implementation of distributed applications that need to be reliable and scalable. At the core of Sinfonia is an efficient minitransaction primitive that allows applications to manipulate shared state consistently, while hiding concerns about fault-tolerance and concurrent execution. We show how to use Sinfonia to build two different, complex applications: a cluster file system and a group communication service. Our applications scale well and achieve performance comparable to other implementations built without Sinfonia.

* Internal Accession Date Only
Approved for External Publication

Building distributed applications using Sinfonia

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

Abstract

We present Sinfonia, a data sharing service that simplifies the design and implementation of distributed applications that need to be reliable and scalable. At the core of Sinfonia is an efficient minitransaction primitive that allows applications to manipulate shared state consistently, while hiding concerns about fault-tolerance and concurrent execution. We show how to use Sinfonia to build two different, complex applications: a cluster file system and a group communication service. Our applications scale well and achieve performance comparable to other implementations built without Sinfonia.

1 Introduction

Distributed applications, such as cluster file systems and group communication services, tend to have a complex design because they explicitly try to deal with issues of concurrency and node and network failures. Concurrency means that nodes execute simultaneously and without constant awareness of what each other is doing. Node and network failures can occur at any time and, if not addressed properly, result in a fragile system that fails if any one of a number of its components fail.

In this paper, we propose a simpler way to build applications distributed over a local area network (LAN), based on Sinfonia, a service that hides the complexity that comes from concurrent behavior and failures while providing scalable performance. In a nutshell, Sinfonia allows nodes to share application data in an general, efficient, consistent, reliable, and scalable manner.

Services that allow application nodes to share data include database systems and distributed shared memory [2, 5, 9, 15], but they lack the performance needed for some applications where efficiency is vital. For example, attempts to build file systems on top of a database system [18] resulted in an unusable system due to poor performance. For such applications, database systems provide more functionality than needed, resulting in performance overheads. Distributed shared memory tends to use expensive protocols to achieve fault tolerance and suffers from network latencies. Thus, it is not a widely adopted paradigm.

Sinfonia provides a place for application nodes to efficiently share data with each other. Similar to database systems, Sinfonia provides transactions that free the application writer from concerns of concurrent execution and partial updates due to failures. Similar to distributed shared memory systems, Sinfonia provides a fine-grained *address space*, without any structure such as schemas, tables, columns, and rows, which impose overhead.

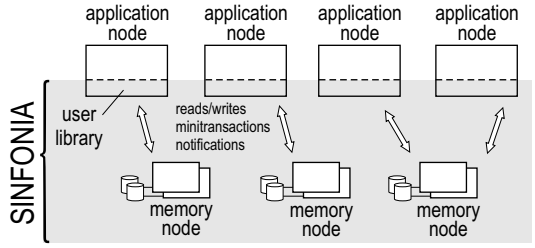

Figure 1: Sinfonia allows application nodes to share data consistently in a scalable and fault tolerant manner.

Sinfonia provides fault tolerance, if desired, in three flavors: availability, reliability, or both. Availability means that Sinfonia is available despite a small number of failures. Availability is achieved by replicating the nodes that implement Sinfonia, so that a backup node can take over if the primary fails. Reliability means that the data in Sinfonia is durable, that is, safe even if all nodes in the system crash, say, due to power failures. Reliability is achieved by logging data on disks.

Sinfonia is intended for building distributed systems that scale, so Sinfonia itself must scale. To do so, Sinfonia has a distributed implementation over a set of nodes, called *memory nodes*, whose number determine the capacity of the system.

Sinfonia provides three intuitive primitives to access data: read/write, minitransactions, and notifications. Read/write retrieves or stores data at a contiguous address range. Minitransactions atomically perform conditional updates of scattered address ranges. Minitransactions are serializable, even if minitransactions span multiple memory nodes. To implement minitransactions efficiently, an algorithmic contribution of this paper is a new type of two-phase commit protocol with associated recovery. Notifications provide callbacks when changes occur in one or more address ranges. All Sinfonia primitives provide strong consistency guarantees.

* Work done while at HP Laboratories. Current affiliation is VMware.

The core principles underlying Sinfonia are to (1) provide operations that have loose coupling, to allow parallel execution and obtain scalability, and (2) achieve fault-tolerance before scaling the system to avoid running expensive fault tolerant protocols over many nodes.

We demonstrate Sinfonia by using it to build two complex and very different applications: a cluster file system called SinfoniaFS and a group communication service called SinfoniaGCS. These applications are known to be difficult to implement in a scalable and fault-tolerant fashion: systems achieving these goals tend to be very complicated and are the result of years of trial and error. Using Sinfonia, we built and optimized them in one or two months. In SinfoniaFS, Sinfonia holds file system data, and each node in the cluster uses minitransactions to atomically retrieve and update file attributes and allocate and deallocate space. In SinfoniaGCS, Sinfonia stores ordered messages broadcast by clients, and clients use minitransactions to add new messages to the ordering; notifications inform clients of new messages.

Through experiments, we show that Sinfonia and its applications scale well and perform competitively with other implementations. Sinfonia can execute thousands of minitransactions per second at a reasonable latency when running over a single node and the throughput increases well with the number of nodes. SinfoniaFS over a single memory node performs as well as an NFS server and, unlike an NFS server, SinfoniaFS can scale. SinfoniaGCS performs comparably to Spread [1], a well-known high-performance implementation of a group communication service.

2 Assumptions and goals

We consider a distributed system with nodes that can communicate by passing messages over a network. We focus on local area networks, such as available in data centers, which for our purposes are networks with the following properties:

- Users are reasonably trustworthy, rather than malicious. Access control is an orthogonal concern that could be incorporated in Sinfonia, but we have not done so.
- Network delays are within a few orders of magnitude of each other, and do not vary wildly at each instant as in a completely asynchronous environment.
- Network partitions are not frequent and, when they do occur, it is acceptable to pause applications.

These assumptions are not true in wide area networks, peer-to-peer environments, or the Internet as a whole.

The system is subject to failures: a node may crash sometimes and more rarely all nodes may crash, say due to a power outage, and failures may occur at unpredictable times. We do not consider byzantine failures. Disks provide *stable storage*, which means disks are chosen to provide sufficient reliability for the target application; disk choices vary from low-cost ones with offline backup to high-end disk arrays with RAID.

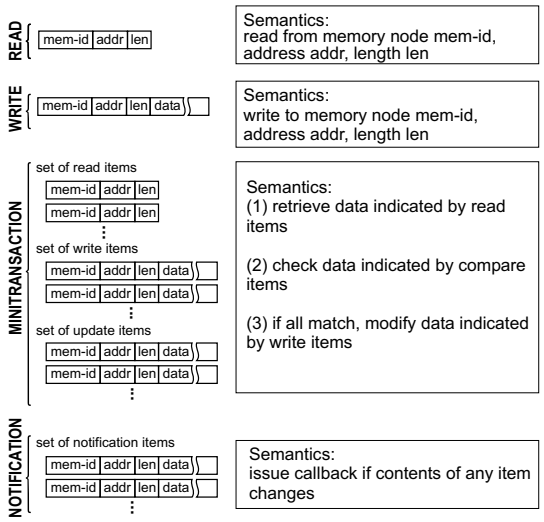

Figure 2: Primitives to access data in Sinfonia.

Our goal is to help build applications distributed over a set of *application nodes*, by providing new efficient building blocks, services, or paradigms. These should be (1) *general*, that is, broadly applicable and useful for many applications, (2) *powerful*, that is, it allows to build artifacts that are hard to build without it, (3) *easy-to-use*, that is, intuitive or quickly understood, and (4) *reasonably efficient* for its intended uses.

We target low-level and middleware applications, like a file system, a lock manager, a group communication layer, or a metadata service, or other layers between the operating system and user applications.

3 What Sinfonia provides

Sinfonia provides a service for application nodes to share data by keeping it at one of a set of memory nodes, which are logically separate from application nodes. Memory nodes are commodity off-the-shelf computers connected to the network, and they are made fault tolerant through the Sinfonia protocols, if desired.

Primitives to access data. Each memory node keeps a sequence of raw or uninterpreted words of some standard size, like 8 or 32 bits (in this paper, words have 8 bits). Those words are organized around an *address space* such that data in Sinfonia can be globally referenced through a pair $(\textit{memory-node-id}, \textit{address})$. Applications access data using three primitives (Figure 2), described next.

Read/write allows applications to read or write a contiguous address range on a *single* memory node.

Minitransactions are a key mechanism to simplify building of applications. They are an extension of the read-write service that allows applications to manipulate data that can be scattered around many memory nodes,

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

Figure 3: Box shows code to create and commit a minitransaction to change an inode's attributes in SinfoniaFS. Lines 6–8 merely populate the minitransaction, without communication with memory nodes until line 9.

and can do so *atomically*. The philosophy, borrowed from database systems, is to always keep data in Sinfonia in a consistent state, and use minitransactions to avoid partial updates due to failures, or interleaved access due to concurrent execution. In designing minitransactions, we had to balance two opposing factors: efficiency (how fast minitransactions execute) and power (how much work they do). We chose a short-lived type of minitransactions that execute quickly but are powerful enough to support optimistic concurrency control at applications. More precisely, a minitransaction has a 0 or more read items, compare items, and write items, where each item refers to an address range at a specified memory node. A minitransaction will return the data in all read items, test each compare item for equality with provided values, and if all comparisons match, write the data in all write items; compare items have a flag indicating whether, if the comparison fails, the actual data should be returned (i.e., the comparison item becomes a read item). Minitransactions are serializable, as expected. Common uses of a minitransaction include (1) swap data by having no compare items and setting the locations of read and write items to be the same, (2) compare-and-swap data, by having no read items, (3) atomically write to many memory nodes, by having several write items but no read or compare items, or (4) atomically read from many memory nodes, by having no compare or write items. Minitransactions can be started, executed, and committed all in just one or two network round-trips. One reason they are efficient is that minitransactions are *static* rather than interactive or long-lived. Figure 3 shows how an application node of SinfoniaFS implements a set attribute operation using a minitransaction with one compare item and two write items (no read items).

Notifications provide a callback from a memory node to an application node if some location within a specified address range is updated (with any value). To avoid race conditions, in which the location is updated by another application node right before notification is installed, the application provides data to be initially compared against the address range, and if comparison fails, notification is triggered immediately. The application chooses if a notification is *permanent*, meaning that it remains active after it is triggered, or *transient*, meaning that it gets canceled after being triggered once. The application also chooses whether a notification should return the contents of the data when it is triggered. Sinfonia provides *notification batches* so that the application node can install and cancel many notifications at once, to avoid multiple rounds of communication.

Fault tolerance. Crashes of application nodes never affect the data kept in Sinfonia, as outstanding writes or minitransactions either complete or get aborted. In addition, for memory nodes, which keep important application data, Sinfonia provides two different forms of fault tolerance:

- *Masking independent failures.* If few nodes crash, Sinfonia masks the failures so that the system continues working with no downtime.
- *Preserving data on correlated failures.* If many nodes crash in a short period, for example due to a power outage, Sinfonia ensures that data is not lost, but the system may become unavailable until enough nodes come back up and execute a recovery procedure.

The goal is to ensure data is always preserved, and the system remains available even when there is a small number of failures. To do so, Sinfonia uses three mechanisms, *disk images*, *disk logging*, and *replication*. These mechanisms can be individually disabled to trade off fault tolerance for performance and fewer machines. A disk image keeps a copy of the data at a memory node; for efficiency, the disk image is written asynchronously and so it may be slightly out of date. To compensate for that, a disk log is used, whereby updates to memory get logged on disk; writes to the log are sequential and done synchronously, that is, before acknowledging completion of a minitransaction. Logging is borrowed from databases, but we streamlined it for use in Sinfonia, as we describe in Section 4. When a node recovers from a crash, the log is used to update the disk image using a *recovery algorithm*; recovery is logically transparent to users of the system, but requires some time to execute, which translates into unavailability. To provide high availability, Sinfonia supports primary-backup replication of memory nodes with a hot backup, which is kept updated synchronously. If a memory nodes fail, the backup can take over without downtime. For performance, the backup node and the log of the primary are written concurrently, while the log of the backup is updated asynchronously (which sacrifices the ability to tolerate a failure of the primary, of its log disk, and of the backup simultaneously, a rare situation). Figure 4 shows the results of various combinations of enabling or

Mode	RAM	PB	LOG	PB-LOG
Description	disk off log off repl. off	disk off log off repl. on	disk on log on repl. off	disk on log on repl. on
Resources/ mem node	1 PC	2 PC's	1 PC 2 disks	2 PC's 4 disks
Fault tolerance	•app crash	•app crash •few memnode crashes	•app crash •all memnode crashes with downtime	•app crash •few memnode crashes •all memnode crashes with downtime
Performance	first	second	third	fourth

Figure 4: Trading off fault tolerance for amount of resources and performance. Each column is a mode of operation for Sinfonia. "Repl." is replication. 'App crash' means it can tolerate crashes of any number of application nodes. 'Few memnode crashes' means tolerating crashes of memory as long as both primary and backup do not crash. 'Downtime' refers to the system blocking until recovery, but no data is lost. For exact performance numbers see Section 7.1.1.

disabling the above mechanisms. Which combination is used depends on the needs of the application. We tested both extremes with two different applications: SinfoniaFS uses PB-LOG, while SinfoniaGCS uses RAM.

Scalability. Sinfonia can increase its capacity by increasing the number of memory nodes. Applications can adjust the placement of data to load balance the memory nodes, which is generally not a problem since all memory nodes are functionally equivalent. Achieving optimal scalability depends on the ability of the application to avoid hot spots, a well known problem in distributed systems that we do not solve in its full generality. But we demonstrate how we addressed this problem for the two applications that we built, when we describe them.

Principles and rationale. Sinfonia's design are based on the following:

Principle 1. Get scalability by reducing or eliminating coupling. Coupling refers to the interdependence that operations have on one other, which precludes parallel execution and hence prevents scalability through distribution. Sinfonia avoids coupling by providing a low-level address space where locations are independent of each other. For example, minitransactions can execute in parallel as long as their locations do not overlap.

Principle 2. Get fault tolerance before scaling. We avoid running expensive fault tolerant protocols over many nodes, by instead making fault-tolerant the low-level components (the memory nodes in Sinfonia) and scaling the system by adding more of them. Thus, the overhead of fault tolerance remains local.

Lessons from distributed shared memory. Sinfonia also draws lessons from the long line of work in distributed shared memory and multiprocessor memory, as distributed shared memory can be seen as a service to share data, like Sinfonia.

Distributed shared memory originated from the effort to simulate shared memory over message-passing (e.g., [2, 5, 9, 15]), as shared memory provides a higher-level and easier-to-use abstraction, allowing for simpler and more compact code than message-passing This paper draws two lessons from this effort:

- *Faithful simulation of shared memory results in poor performance.* Attempts to provide abstractions identical to shared memory failed to achieve adequate performance because of network latencies and expensive protocols.
- *Relaxed consistency to improve performance leads to hard-to-use models.* Performance of DSM can be improved by relaxing the consistency provided to permit aggressive caching. However, this means that a write is not always visible, and reads return inconsistent data. This makes it hard to understand, develop, and debug applications.

Based on these lessons, Sinfonia provides an abstraction quite different from traditional shared memory, intended to deal with network latencies. Indeed, minitransactions package together many accesses to scattered locations, and this can hide network latencies; and notifications avoids repeated polling over the network.

At the same time, Sinfonia provides strong consistency for ease of use: minitransactions are serializable and durable, and they take effect immediately.

4 Sinfonia implementation and algorithms

An algorithmic contribution of our work is a new type of two-phase commit optimized for Sinfonia to provide good performance and fault tolerance. In traditional two-phase commit, a coordinator runs the protocol *after* executing all of the transaction's actions; if the coordinator crashes, the system has to block until the coordinator recovers. In our two-phase protocol, the coordinator can start, execute, and commit a minitransaction, all in two phases. Furthermore, if the coordinator crashes, the system can recover without it. This is important because in Sinfonia the coordinators are application nodes, not Sinfonia nodes, and so they may be unstable or subject to reboots. There is a trade-off with our protocol, however: if *participants* crash the system blocks until participants recover. This is not a concern for Sinfonia because participants are memory nodes that keep application data, so if they go down, the system has to block anyways.

Our commit protocol has new recovery and garbage collection procedures to go with it, which reflect its different failure assumptions.

We now explain the basic architecture of Sinfonia's implementation and then describe the commit protocol and notifications.

4.1 Basic architecture

Implementation of Sinfonia has two parts: A user library running at application nodes provides the interface to applications, while a server running at memory nodes keep the data in Sinfonia (Figure 1). The user library provides the Sinfonia API to applications; it implements distributed protocols for the Sinfonia services, including

Code for coordinator p:

To execute minitransaction (*rditems*, *cmpitems*, *writems*)
1 *tid* ← unique identifier
{ Phase 1 }
2 D ← set of memory nodes referred in *rditems* ∪ *cmpitems* ∪ *writems*
3 **pfor** each $q \in D$ **do** { pfor is a parallel for }
4 **send** (LOCK-CMP-PREPARE, *tid*, D, $\pi_q(rditems), \pi_q(cmpitems), \pi_q(writems)$) **to** q
5 { π_q denotes the projection to the items handled by q }
6 *replies* ← wait for replies from all nodes in D
7 { Phase 2 }
8 **if** $\forall q \in D$: *replies*[q].*vote* = OK **then** *action* ← TRUE
9 **else** *action* ← FALSE
10 **pfor** each $q \in D$ **do send** (COMMIT, *tid*, *action*) **to** q
{ can always return without waiting for reply of COMMIT }

Code for each participant memory node q:

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

upon receive (COMMIT, *tid*, *action*) **from** p **do** { *action*: true=commit, false=abort }
25 (*rditems*, *cmpitems*, *writems*) ← find (*tid*, ∗, ∗, ∗) in *active*
26 *active* ← *active* − {(*tid*, *rditems*, *cmpitems*, *writems*)}
27 **if** *tid* ∈ *all-in-log list* **then** *done* ← *done* ∪ {(*tid*, *action*)}
28 **if** *action* **then** apply *writems*
29 release any locks still held for *rditems* ∪ *cmpitems* ∪ *writems*

Figure 5: Sinfonia's commit protocol for minitransactions.

the commit and recovery protocols. The server at memory nodes is a passive entity that keeps Sinfonia's address space.

4.2 Minitransactions

Recall that a minitransaction has read items, compare items, and write items (Figure 2). The read items are locations to be read and returned; the compare items are locations to be tested against supplied data; the write items are locations to be written if all comparisons succeed.

An application node executes a minitransaction using the two-phase protocol in Figure 5. In the first phase, the coordinator (application node) sends the minitransaction to participant memory nodes. Each participant does the following: (1) try to lock the address ranges in the minitransaction, (2) perform the compare items, and (3) vote to commit if successful at (1) and (2), else vote to abort. The second phase is like in the traditional protocol: the coordinator tells participants to commit iff all votes are to commit. A difference is that the coordinator never logs any information, because it may crash and never recover. Other details of the protocol will be explained as we discuss recovery.

Name	Description	On disk?
log	minitransaction log	Yes
active	*tid*s not yet committed or aborted	No
forced-abort	*tid*s forced to abort (by recovery)	Yes
done	*tid*s in log with outcome	Lazily
all-in-log	*tid*s in log	No

Figure 6: Data structures kept at participant memory nodes for recovery and garbage collection.

4.2.1 Data structures for recovery and garbage collection

Participants log minitransactions to disk in the first phase (if logging is enabled according to Sinfonia's mode); logging occurs only if the participant votes to commit. Only the participant's vote and write items are logged, not compare or read items, to save space. In essence the log in Sinfonia serves as a write-ahead log. We optimized the log using the skipping technique in [10], in which consecutive writes to the log skip sectors to avoid waiting for a full disk revolution.

To enable recovery and garbage collection, participants keep a set *active* of outstanding transaction id's ($tid's$), a set *forced-abort* set of tid's that must be voted to abort, and a set *done* of finished minitransactions with their *tid* and outcome. Figure 6 shows a summary of data structures kept by participants for recovery.

4.2.2 Recovery from coordinator crashes

If a coordinator crashes during a minitransaction, it may leave the minitransaction with an uncertain outcome. To fix this problem, a recovery mechanism is executed by a third-party, called *recovery coordinator*, which runs at a dedicated management node for Sinfonia. The recovery mechanism ensures the following: (A) it will not drive the system into an unrecoverable state if the recovery coordinator crashes; (B) it ensures correctness even if there is concurrent execution of recovery with the original coordinator, if the original coordinator is incorrectly deemed to have crashed; and (C) it allows concurrent execution by multiple recovery coordinators.

Roughly speaking, recovery works by trying to abort the outstanding minitransaction while respecting the following invariant:

> (Invariant I) A minitransaction is committed iff all participants have a yes vote in their log.

Thus, the recovery coordinator determines if each participant has voted yes and, if not, forces the participant to vote no (the participant adds the tid to the *forced-abort* list). This is necessary to ensure properties (A), (B) and (C) above. If all participants voted yes, the recovery coordinator tells them to commit. Otherwise, it tells them to abort.

How does the recovery coordinator get triggered in the first place? We use the *active* list of minitransactions. Management nodes periodically probe memory nodes for those minitransactions that have not yet committed for a long time, and starts recovery for them.

4.2.3 Recovery from participant crashes

When a participant memory node crashes, the system blocks until it comes back up[1], at which time the memory node replays the log in order. To avoid replaying a long log, there is a *processed-pointer* variable that gets periodically written to disk, which indicates what parts of the log are new; replay starts at this place. Not every minitransaction in the log should be replayed, only those that committed, which is determined by its presence in the *done* list or, if not there, by consulting the set D of memory nodes that participated in the minitransaction. This set D is stored in the log. Upon being consulted by this procedure, if a memory node has not voted, then it votes no. This is necessary for correctness of recovery while a coordinator is still running.

4.2.4 Recovery from crash of the whole system

To recover from the crash of the whole system, each memory node essentially uses the previously described scheme to recover from its own crash, but uses optimizations that batches recovery of all memory nodes for better efficiency.

4.2.5 Garbage collection

A memory node flushes dirty buffers to disk, in log order, so that the log can be garbage collected. Garbage collection respects the following property:

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

The reason for having "every" above is that if some memory node q crashes and recovers, then q may need to see tid at the log of other memory nodes to determine whether tid committed or aborted. To implement the property, a memory node p periodically informs each other memory node q of the minitransaction tid's that p recently flushed and that q participated in.

Besides the log, the other data structures in Figure 6 are garbage collected as follows. The *all-in-log* list, *active* list, and *done* list are garbage collected with the log. But the *forced-abort* list has to be garbage collected in a different way, as the following scenario shows. Suppose that a coordinator becomes very slow before finishing the first phase of a minitransaction *tid*. The system may then consider the coordinator as crashed, and trigger recovery, which then aborts *tid* since not every participant voted yes, and adds *tid* to the *forced-abort* list. Next, the system garbage collects *tid* from the log of every memory node. At this time, if the system garbage collected *tid* from the *forced-abort* list, the original coordinator could continue executing, thereby contacting participants that completely forgot about *tid*, which would then vote yes, thinking it is a new minitransaction. The coordinator would then ask to commit the minitransaction but some participants do not even know what to commit. To solve this problem, roughly speaking, we expire minitransactions that are too old using epoch numbers that participants keep among themselves. This is reasonable since minitransactions are short-lived. The scheme requires participants (not coordinators) to have loosely synchronized clocks, which is also reasonable since they are Sinfonia nodes.

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

4.2.6 Further optimizations

If a minitransaction has just one participant, it can be executed in one phase because its outcome only depends on that participant. Another optimization is for *read-only minitransactions*, that is, minitransactions without write items, which do not modify any data. For these, it is not necessary for memory nodes to log the outcome of the minitransaction to the log, because recovery is not needed.

4.2.7 Replication

We use primary-backup to replicate a memory node, if desired. The backup memory node is actively synchronized within the two-phase protocol. The second phase message is always forwarded to the backup. For the first-phase message, after the primary memory node gets the message, if it decides to vote yes then it forwards the message to synchronize the backup. This can be done concurrently with writing to the log for efficiency. The primary must wait for an acknowledgement from the backup before reporting its yes vote to the coordinator. The backup can acknowledge the primary without waiting to log its own vote, because we are not trying to tolerate the failure of both primary, the primary's disk, and the backup—a rare event. By doing so, we hide the latency of synchronizing the backup.

4.3 Notifications

Notifications are relatively simple to implement: memory nodes keep a list of active notifications and every time a minitransaction of write occurs, it checks if a notification should be triggered. For efficiency, the memory node keeps an interval list that indicates all ranges covered by notifications, and this data structure can be queried in time logarithmic on the number of notifications.

4.3.1 Fault tolerance

Notifications can survive crashes of memory nodes, but they are not persistent across crashes of the application nodes. The reason for this design choice is that if the application wants such notifications, it can store them in Sinfonia and retrieve it as it recovers.

To survive crashes of a memory node, when it recovers it asks the application nodes to resend the notifications that are in place. To know which application nodes to contact, a memory node keeps a *notification-node-set*, which is logged to disk as incremental additions and removals of nodes. Removals need not be logged immedi-

ately, because not doing so is a conservative action. Deferring logging of removals is important because if some node later adds the same notification there is no need to log the addition.

Log entries for *notification-node-set* are easy to garbage collect: as soon as the disk image is updated, all entries can be marked as ready for garbage collection. This scheme works because if a disk image of *notification-node-set* is up-to-date then applying any suffix of the log to the disk image leaves it unchanged.

4.4 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.

Cluster nodes use Sinfonia to store file system metadata and data, which include inodes, maps of free space, chaining information with a list of data blocks for inodes, and the contents of inodes. SinfoniaFS exports NFS v2, and nodes in the cluster can mount its own NFS server locally. All NFS servers export the same files.

In a nutshell, Sinfonia helps 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. We now provide details.

5.1 Data layout

Aspects similar to local file systems. Data layout (Figure 7) is somewhat similar to that of a local file system on a disk, except SinfoniaFS is laid out over several Sinfonia memory nodes. The *superblock* has static information about the entire volume, like volume name, number of data blocks, and number of inodes. *Inodes* keep the attributes of files such as type, access mode, owner, and timestamps. *Data blocks* of 16 KB each keep the contents of files. The *free block bitmap* indicates which data blocks are in use. Chaining-list blocks indicate which blocks comprise a file; they are stored in data blocks, and have pointers to the next block in the list. Note that a 4 GB file requires only 65 chaining-list blocks (each chaining block can hold 4095 block numbers), so we did not implement indirect blocks, but they could be implemented easily. Directories and symbolic links have their contents stored like regular files, in data blocks.

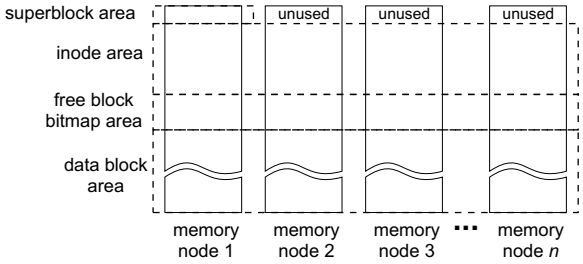

Figure 7: Data layout for SinfoniaFS.

Aspects unique to SinfoniaFS. Data block numbers are not just integers, but pairs with a memory node id and an offset local to the memory node. This enables a file to have its contents spread over many memory nodes. It also allows to add memory nodes to grow the file system without affecting existing files. Similarly, inode numbers are pairs with a memory node id and a local offset, which allows a directory to point to inodes at many memory nodes.

5.2 Making modifications and caching

Cluster nodes use minitransactions to modify file system structures, like inodes and directories, while preserving their integrity.

Cluster nodes can cache arbitrary amounts of data or metadata, including inodes, the free block bitmap, and the content of files and directories. Because cache entries get stale, they are validated against Sinfonia as they are used. Validation occurs by adding compare items to a minitransaction, to check that the cached version matches what is in Sinfonia. For example, Figure 3 shows the implementation of NFS's setattr, which changes attributes of an inode. The compare item in line 6 ensures that the minitransaction only commits if the cached version matches what is in Sinfonia. If the minitransaction aborts due to a mismatch, the cache is refreshed and the minitransaction is retried. This approach is a type of optimistic concurrency control.

Operations that modify data always validate cache entries against Sinfonia. For efficiency, *read-only* operations that involve cached entries refreshed recently (3 seconds in our experiments) execute without revalida-

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

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

```
(1) if file's inode is not cache then load inode and chaining list
(2) find a free block in the cached free-block bitmap
(3) issue a minitransaction that checks iversion of
    the cached inode, checks the free status of the new block, updates the
    inode's iversion and dversion, appends the new block to the inode's
    chaining list, and populates the new block
(4) if the minitransaction fails because the igeneration does not match
then return stale filehandle error
(5) else if failed because the iversion or dversion do not match
    then reloads cache and retry
(6) else return success
```

Figure 9: Implementation of a write that appends to a file, requiring to allocate a new block.

tion. This approach can result in slightly stale data being returned to readdir, read, lookup, or stat (getattr), but such behavior is often seen with NFS.

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

5.3 Locality

In general, an inode, its chaining list, and its file contents may all be stored in different memory nodes, but Sinfonia tries to place them in the same memory node for locality. This allows minitransactions to involve fewer memory nodes, and provides better scalability of performance, as shown by experiments.

5.4 Contention

There are two types of contention. *Memory node contention* occurs when many application nodes access the same memory node, but possibly different locations. This could cause hot spots. *Location contention* occurs when accessing the same memory node *and* location. This can cause minitransactions to abort, which requires retrying. Optimistic concurrency control is not ideal when this happens.

SinfoniaFS uses the following techniques to avoid both forms of contention:

- Spread equivalent structures across memory nodes to balance load.
- When new items are allocated and contention is detected, choose a random memory node on which to allocate.

For example, if a cluster node cannot allocate a fresh inode because another node just grabbed it, the first retries allocation at a new random memory node. This tends to spread load well if the file system is not nearly full (which is often the case since storage is cheap), without requiring cluster nodes to coordinate.

6 Application: group communication

We used Sinfonia to implement a simple group communication service — a basic abstraction for the design of fault-tolerant distributed applications [8] – that provides reliable multicast and membership services to a collection of distributed entities in a system. These entities may be processes, threads or entire hosts, but we call them processes.

6.1 Informal semantics

Intuitively, group communication [4] ensures that members agree on a sequence for membership changes and messages broadcast. A commonly agreed upon membership is called a *view*. We say that a message *m* is *broadcast in view v*, if *v* is the last view seen by the sender when the message is broadcast. Similarly, we say that *m* is received in view v, if *v* is the last view seen by the receiver before receiving *m*. The service guarantees that each member that receives a message *m* agrees on the same last view and each member is part of that view.

An *event* is either a data message or view change notice. The group communication service guarantees that there exists a total ordering of all events, such that the following properties are satisfied. The safety property specifies that every process *p* receives a gapless, *prefix* of the event sequence starting from the first view in which *p* appears. Moreover, for liveness, in failure-free conditions, the prefix includes all events up to the exclusion of *p* from a view (if *p* is never excluded, then it includes all events). If *p* crashes, the prefix may end earlier. We assume that processes can join the group once; to rejoin, they obtain a new member id.

For a process to broadcast a message, it must be a member of the latest view in the group. To receive a message, it must be a member of the view in which the message was sent. To join a group, a process invokes a *join* call and eventually a view event, which includes the new process, is received by all non-faulty group members. This view event is the first event that the new member receives from the service. For a process to be removed from the group, a *leave* operation must be invoked either by itself or by another process on its behalf, e.g., if it has crashed. We assume that failure detection is a separate service [6] used for removing failed members from the latest view.

6.2 Design

We outline a simple design for group communication using Sinfonia, and extend it to scale well with multiple broadcasters.

To implement the group communication service using Sinfonia, a straightforward design employs a single, large circular queue that contains all events in the sys-

tem. We store the queue on the Sinfonia memory nodes and the total order among events is simply the order in which they appear in the queue. We mediate each member's access to this queue using minitransactions to avoid inconsistencies in the global sequence. To broadcast an event, a writer finds the next empty location at the end of the event sequence, i.e. the *tail* of the queue, and inserts the event at that location using a minitransaction. To receive an event, a reader reads the event, using a minitransaction, from the *head* of the queue, i.e. the location following the location of the last received event. Further, we stripe the queue across the memory nodes to allow readers parallel access to different portions of the queue.

Note, in this design, each member may have a different head location, but there is only one true global tail. Thus, while this design scales well with the number of readers, it fares poorly with increasing number of writers. As writers simultaneously insert at the tail, all but one will succeed, and the others must retry the entire broadcast procedure until success. In order to reduce contention for the tail location, we extend the above design as follows.

Instead of single queue, for each group member, we maintain a separate, dedicated circular queue in which only that member stores its events for broadcast. Moreover, we determine the total order by "threading" together events. Each event contains a "next" field indicating the location of the next event in the total order. In this design, to broadcast, a writer first installs a event at tail of its queue, and since the queue is dedicated, a minitransaction is not needed. Then, the writer locates the last event in the global sequence, call it the global tail, and "threads" the new event into the global sequence by updating the "next" field in the global tail to point to the new event using a minitransaction. If unsuccessful, the writer simply retries the "thread" portion of the broadcast. Since the writer does not have to reinstall the event into the queue on a retry, this approach reduces the duration that the global tail location is accessed, thereby reducing contention for a broadcast. To receive an event, a reader uses a minitransaction to read the "next" field from the previously received event and retrieves the event at that location. For simplicity, in this design, we assign queues to memory nodes in round-robin fashion (queues do not span memory nodes).

Join and leave operations reuse broadcast functionality and in addition modify global metadata. We maintain global metadata in Sinfonia memory space that records the latest view and the location of each member's queue. In the join method, we first acquire a global lease on the metadata so other membership changes cannot interfere. We implement the global lease using a minitransaction with a single compare-and-swap. Once acquired, we update the view to include the new member, find the global tail, and broadcast a view event. Once broadcast, we release the global lease. The leave operation is exactly the same except the metadata reflects the absence of the removed member.

6.3 Garbage collection

Once a process encounters a full queue, it must garbage collect or free entries that all relevant processes have consumed. To do so, each process needs an estimate of the last event consumed from its queue by all other processes in the latest view. To record this estimate, we modify the read method to periodically post to Sinfonia memory (for the currently reading process) the location of the last event consumed for all queues in the system. We perform garbage collection in the broadcast method. Once a writer encounters a full queue, using the estimates for its queue, it determines the earliest or "minimum" of all events consumed by all other relevant processes. The writer only considers the estimates from processes in the latest view. All events from the queue's tail to the earliest consumed event can be safely removed since processes insert and read queue entries in order and we are certain all relevant processes are past the minimum. The writer clears a fixed number of entries from the tail up to the location of the minimum event in a single minitransaction. After garbage collection, the writer continues with the pending broadcast operation.

6.4 Some further optimizations

Finding the global tail in the broadcast, join, and leave operations is non-trivial and potentially expensive. A naive approach starts from a previous known position in the global event sequence and walks through each item one-by-one until encountering the end. This approach places a high overhead on writers, especially if they prefer only to broadcast. To improve the search for the global tail, with each successful thread operation, we publish the position of last threaded event for that queue to Sinfonia shared memory. We also label the last threaded event with a monotonically increasing global sequence number (GSN) and publish that number (on a per-queue basis) to Sinfonia memory space. When searching for the global tail, we read the GSN of the last threaded events for all queues and select the largest as the global tail. This modification further reduces overhead during broadcast operations.

In addition, we provide a non-blocking interface for broadcast. This interface allows the writers to submit multiple broadcast events and receive confirmation at a later time. In this case, when we install the event into the writer's queue, we also thread that event to the last unthreaded event, if any, in that queue. Thus, at any time, each queue may contain, near its tail, a batch of consecutive events strung together, yet unthreaded to the global sequence. Threading these pending events to the global sequence involves pointing the global tail's next field to the earliest unthreaded event and marking the last installed event with the next GSN. Since the GSN is only used to locate the tail, the intermediate unthreaded events

in the batch need not carry any GSNs. Also, if event sizes are small, we further coalesce those events into a single large event before installing the event into the queue. This non-blocking interface allows the client to do useful work and publish additional events during the time other processes contend for the global tail. This approach increases the number of events published per successful thread operation, resulting in improved throughput. Once successfully threaded, we post the confirmation of the pending broadcasts to the client.

7 Evaluation

In this section, we evaluate Sinfonia and our two example applications. Our testing infrastructure includes up to 24 machines connected by Intel Gigabit Ethernet interfaces. Each machine has a 1GHz Pentium 3 CPU with 2GB of main memory, and two Seagate Cheetah 32GB SCSI disks (15K rpm, 3.6ms average seek time). Each runs Fedora Core 3 with the Linux 2.6.9 kernel.

7.1 Sinfonia

We evaluated Sinfonia's minitransactions in various modes of operation, as described in Figure 4. We compared base performance against an open-source, commercial developer database library, BerkeleyDB version version 4.3. BerkeleyDB is centralized, 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 within BerkeleyDB.[2] In this experiment, we tested Sinfonia's ability to scale and deal with minitransactions that overlap causing contention.

7.1.1 Result: base performance

Figure 10 shows latency-throughput graph for a workload that repeatedly issues minitransactions, each with 32 items and minitransaction spread 1. Minitransaction spread specifies the number of memory nodes that a minitransaction touches. The Sinfonia experiments used 4 memory nodes. We started with a single application node issuing such transactions repeatedly with at most 16 transactions outstanding at any time. We increased the number of application nodes up to 6, each running the above workload.

As can be seen, the four Sinfonia modes of operation can reasonably trade off fault tolerance for performance, and performance is comparable or better than BerkeleyDB. We also did experiments where Sinfonia had only one memory node, and in all modes of operation, Sinfonia still performed well, in fact better than BerkeleyDB.

In another experiment we evaluated Sinfonia and BerkeleyDB in an idle system with 1 memory node. We had a single application node repeatedly issuing transactions with at most 1 outstanding transaction at a time with spread 1. Figure 11 shows the result. For Sinfonia modes that involve disk (LOG and PB-LOG), performance is dictated by disk latencies. Sinfonia performs better than BerkeleyDB because it optimize writes to the log.

[2] BerkeleyDB also has its own RPC interface that allows the minitransactions to be executed from a remote site, but it performed poorly.

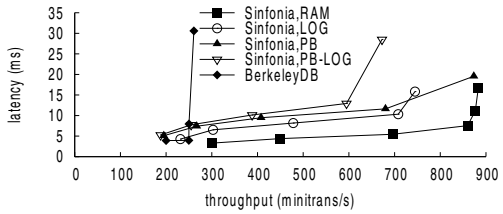

Figure 10: Base performance of Sinfonia.

Mode	RAM	PB	LOG	PB-LOG	BerkeleyDB
Minitrans latency (ms)	0.7	1.2	1.7	1.8	3.9
Throughput (minitrans/s)	1310	797	576	541	250

Figure 11: Base performance on an idle system.

7.1.2 Result: scalability

We tested scalability of Sinfonia by measuring performance as we varied the number of memory nodes. Figure 12 shows minitransaction throughput with 2-12 memory nodes and 1-11 application nodes. Sinfonia was in LOG mode. Minitransactions contained 4 items (512 bytes) with spread 2. Each application node issued 16 outstanding minitransaction at any time.

As can be seen, with a small number of application nodes, there is little benefit in adding more memory nodes since the system is under-utilized. With 11 clients, as we vary from 2 to 16 memory nodes, performance is at around 71% of linear scaling. Other experiments with minitransaction spread 1 have near linear scalability. Other Sinfonia modes have similar or better results.

Figure 13 shows minitransaction throughput as we vary the number of memory nodes and spread. There were 11 application nodes, Sinfonia was in LOG mode and minitransactions had 32 items (4KB). Absolute throughput is lower than in the previous experiment because minitransactions were larger to allow higher spread.

As can be seen, higher minitransaction spread is detrimental to scalability. We did not know this initially, but in retrospect the explanation is simple: a minitransaction incurs a high initial cost at a memory node but much smaller incremental cost (with number of items), and so spreading it over many nodes reduces overall system efficiency. Thus, to achieve optimal scalability, we obey the following simple rule:

> *Across minitransactions, spread load. Within a minitransaction, focus load.*

In other words, one should strive for each minitransaction to involve a small number of memory nodes, and for different minitransactions to involve different nodes. When we first built SinfoniaFS, we were getting no scal-

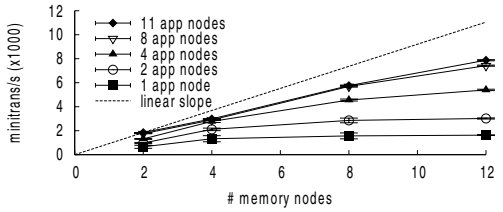

Figure 12: Sinfonia scalability.

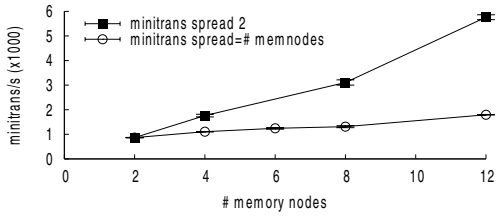

Figure 13: Effect of minitransaction spread on scalability.

ability, but after a redesign aimed at reducing spread, scalability improved dramatically.

7.1.3 Result: contention

Figure 14 shows throughput as we vary the probability that two minitransactions overlap, causing contention. Sinfonia had 4 memory nodes, minitransactions consist of 8 compare and 8 write items to perform a compare-and-swap on 8 locations. The experiment was set up such that the compares always succeeded, so that we can measure the efficiency of our commit protocol in isolation. Minitransaction spread was 2. There were 4 application nodes each with 16 outstanding minitransactions at a time. To vary the probability of pairwise overlap, we varied the range of items. For example, with range 1024, the probability of pairwise overlap is $1 - (\frac{1024-8}{1024})^8 \approx 0.06$.

As can be seen, Sinfonia provides better throughput than BerkeleyDB, even with high contention. We also measured latency, and the results are qualitatively similar.

In another experiment, we used Sinfonia to increment values atomically, by having a local cached copy of the values at the application node, and using a minitransaction to validate the cache and write the new value. Here, a minitransaction may fail because a compare fails. In this case, the application refreshed its cache and retried the minitransaction. In this experiment, BerkeleyDB performed better than Sinfonia with high probabilities of collision, because Sinfonia had to retry multiple times, whereas BerkeleyDB could just lock a location, read a value, and write the new value. This is not surprising, as optimistic concurrent control is known to be inferior when optimism is rarely accurate. Thus, we design applications to avoid this situation.

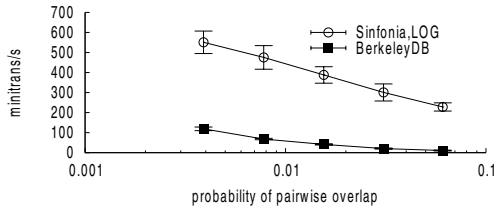

Figure 14: Effect of minitransaction overlap on performance.

7.1.4 Result: ease of use

To evaluate ease of use, we report on our experience in building two complex applications over Sinfonia, including advantages and drawbacks. We found that the main advantages of using Sinfonia were that (1) we never had to worry about failures of nodes (e.g., application nodes), (2) we did not have to develop any distributed protocols and worry about timeouts, (3) in fact, application nodes did not have to keep track of each other, and (4) the correctness of the implementation could be verified by looking at a few places in the code and ensuring that minitransactions maintained the invariants of shared data structures. The main drawbacks were that (1) Sinfonia's address space is a low-level abstraction that we had to carefully program with, (2) we had to design concurrent data structures that were efficient in the presence of contention, an algorithmic problem.

As a quantitative measure of benefits of Sinfonia, we implemented SinfoniaFS with about 2831 lines of C++ code and 1040 lines of additional glue code. We implemented SinfoniaGCS with about 2817 lines of C++ code and 505 lines of glue code.

7.2 Cluster file system

7.2.1 Result: ability to scale down

We first consider performance of SinfoniaFS at a small scale, to ensure that we are scaling a system with reasonable performance. The ability to "scale down" to small sizes is also important for when a deployment is initially small and grows over time.

We first ran SinfoniaFS with the Connectathon NFS Testsuite, which is mostly a metadata intensive microbenchmark with many phases that exercises one or two file system functions. We modified some phases to increase the work by a factor of 10, shown in Figure 15, because otherwise they execute too quickly.

Figure 16 shows the benchmark results for SinfoniaFS compared to a Linux Fedora Core 3 NFS server, where smaller numbers are better as then they indicate smaller running time. We used NFSv2 protocol in both cases, and the underlying file system for the NFS server is ext3. Sinfonia was set to LOG mode (under "SinfoniaFS") or PB-LOG mode (under "SinfoniaFS replicated") and the NFS server was set to synchronous mode to provide data durability. As can be seen, SinfoniaFS performs at least as well, sometimes better than Linux

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

Figure 15: Connectathon NFS Testsuite modified for 10x work.

Phase	Linux NFS	SinfoniaFS	SinfoniaFS replicated
1	14.21s	3.27s	3.73s
2	11.60s	3.62s	4.25s
3	0	0	0
4	82.96s	23.11s	25.94s
4a	0	0	0
5a	6.40s	4.54s	5.14s
5b	0.30s	0.30s	0.30s
6	2.60s	2.29s	2.36s
7a	25.11s	6.79s	6.99s
7b	16.72s	7.35s	7.54s
8	34.19s	18.68s	20.21s
9	0.50s	0.15s	0.15s

Figure 16: Results of Connectathon NFS Testsuite.

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 results are not significant. SinfoniaFS-replicated performs close to SinfoniaFS because logging and the backup synchronization happen simultaneously in Sinfonia (see Section 3), and so the extra latency of replication is hidden.

Next, we ran a macro benchmark with a more balanced mix of data and metadata operations. We modified the Andrew benchmark to use as input tcl 8.4.7 source code, which has 20 directories and 402 regular files with 16MB total size. 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 17 shows the results, again comparing SinfoniaFS with Linux NFS. As can be seen, Sinfonia performs better in phase 1 because this phase modifies metadata intensively. In phases 2 and 4, SinfoniaFS performs worse, as there is lots of data read and the current implementation of SinfoniaFS suffers from copying overhead,

Phase	Linux NFS	SinfoniaFS	SinfoniaFS replicated
1 (mkdir)	22.7s	9.3s	9.8s
2 (cp)	50.2s	53.8s	55.9s
3 (ls -l)	4.9s	7.5s	7.6s
4 (grep + wc)	12.7s	16.8s	16.9s
5 (make)	106.5s	98.5s	99.5s

Figure 17: Results of Andrew benchmark.

as both Sinfonia and SinfoniaFS are in user space without buffer-copying optimizations. Indeed, after a byte is read from disk, a memory node sends it over the network (user-to-kernel copy) to the SinfoniaFS server, which processes it in user-space (kernel-to-user copy), and then sends it via a local connection (user-to-kernel copy) to the NFS client (kernel-to-kernel copy), which then hands it to the application (kernel-to-user copy). A better implementation would have both Sinfonia and SinfoniaFS in kernel, would avoid copying buffers, and would use VFS as the interface to SinfoniaFS instead of NFS. In phase 3, Sinfonia also performs worse; besides the copying overhead, Sinfonia's directory structure is very simple and not optimized for large directories. In phase 5, Sinfonia performs slightly better as the benefits of the write-ahead logging outweigh the copying overheads.

7.2.2 Result: ability to scale up

We ran scalability tests in which we vary the number of memory nodes as we ran Andrew and Connectathon. The first experiment considered a fairly loaded system, where 12 cluster nodes ran the benchmarks together and synchronized at each phase of the benchmark. The results for Andrew are shown in Figure 18, where the y axis is the speed up compared to having just one memory node. As can be seen, the speed up is generous up to 4-6 memory nodes, after which there is a diminishing return (not shown) because the system is under-utilized, i.e. capacity outweighs the offered load. The speed up for different phases is different because they tax the memory nodes differently. For example, phase 3 involved listing directories, which are not implemented efficiently in Sinfonia, and so the speed up was the greatest.

The results for the Connectathon Testsuite were similar: for most phases, the speed up was near linear for up to 4 memory nodes, and started diminishing at different points depending on the phase. The exceptions are phases 3 and 4a, which always executed in 0 time, phase 5b (read), which executed in cache and had always the same small execution time, and phase 9 (statfs), which also had always the same small execution time.

We then ran scalability test for a lightly-loaded system. We started with 1 memory node and 1 cluster node, and increased the number of memory nodes and cluster nodes together (ratio 1:1) to see if any system overhead manifested itself in the larger system. Figure 19 shows the results for the Andrew benchmark, where the y-axis shows the duration of each phase relative to a system

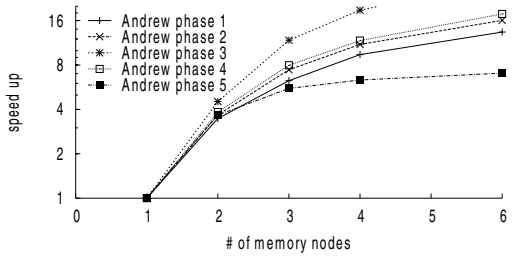

Figure 18: Speedup of Andrew as function of number memory nodes in a heavily-loaded system. Connectathon had similar results.

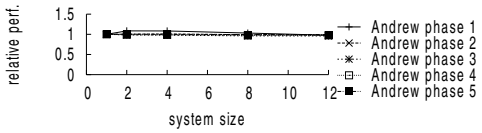

Figure 19: Results of Andrew as we grow a lightly-loaded the system. System size is the number of memory nodes which is equal to the number of cluster nodes running Andrew simultaneously. The y-axis is the duration of each phase relative to system size 1. Connectathon had similar results.

with 1 memory node and 1 cluster node. As can be seen, all curves are flat at y-value 1, which shows virtually no overhead to the larger system. The results of Connectathon were identical: all curves are flat at y-value 1.

7.3 Group communication service

To evaluate the scalability characteristics of our implementation, we ran a simple workload, measured its performance, and compared it with a publicly available group communication toolkit, Spread [1]. In each experiment, we had w writers broadcasting messages as fast as possible and r readers reading messages as fast as possible. Each machine ran a single process, either a reader, a writer, or a Sinfonia memory node. We report the average read throughput of messages for a single reader.

7.3.1 Result: base performance

In the first experiment, we observe how SinfoniaGCS behaves as we vary the number of readers (Figure 20). We fixed the number of writers to 4, the number of memory nodes to 4, and varied the number of readers from 1 to 8. SpreadGCS had 4 daemons, and each client had their own machine, connecting to one of the daemons. We see that the throughput drops only slightly, indicating that aggregate throughput for all readers increases almost linearly. We also see that SinfoniaGCS is competitive with SpreadGCS.

In the second experiment, we observe the behavior as we vary the number of writers (Figure 21). We fixed the number of readers to 4, number of memory nodes to 4, and varied writers from 1 to 8. We see 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 writers exceed the number of memory nodes, we see throughput decreases slightly because we reach the

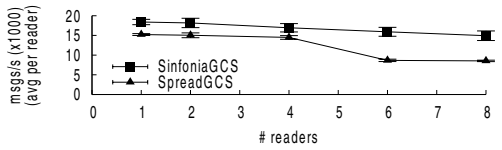

Figure 20: SinfoniaGCS base performance as we vary # readers.

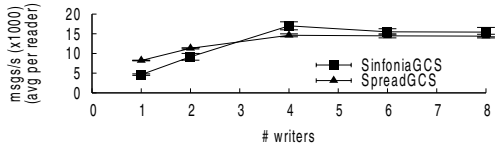

Figure 21: SinfoniaGCS base performance as we vary # writers.

write-capacity of the system, and additional writers impose a slight overhead. We also see that SinfoniaGCS is competitive with SpreadGCS.

7.3.2 Results: scalability

In the third experiment, we observe the scalability as we increase the number of memory nodes (Figure 22). We have 1 reader or 4 readers, 8 writers, and vary the memory nodes from 1 to 8. This is similar to the previous experiment, and we see that adding more memory nodes increases capacity, thereby improving throughput. Yet, throughput does not scale linearly because minitransactions with high spread, e.g. searching for the tail, impose increased overhead with additional machines.

In the fourth experiment, we observe the scalability with total system size (Figure 22). The number of readers, writers, and memory nodes are all the same. We scale system size from 6 to 24 machines. We see that at some point the system throughput does not increase much further although we add more resources. This effect is because all readers receive all messages, so eventually readers are saturated regardless of availability capacity for the rest of the system.

8 Related work

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 [16], QuickSilver [22] and Camelot [24]. 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 [20] by placing the metadata in Camelot's recoverable virtual memory. This abstraction was found to be useful because it simplified crash recovery. However, they also found that the complexity of Camelot led to poor scalability; later versions of Coda replaced Camelot with the Lightweight Recoverable Virtual Memory [21], which dispensed with distributed transactions, nested transactions and recovery

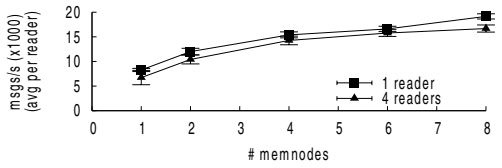

Figure 22: SinfoniaGCS scalability we vary # memory nodes.

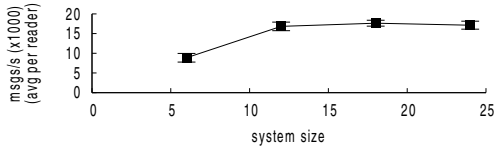

Figure 23: SinfoniaGCS scalability we vary total system size.

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. The Mesa file system [19] included a notification feature that allowed clients to be notified whenever a file became available for access; Sinfonia's notification service is more fine grained, since it allows for notification on any range of bytes, and in response to any access of those bytes.

Sinfonia's minitransactions are inspired by work in distributed shared memory (see, e.g., [2, 5, 9, 15]) and multiprocessor shared memory [14, 23, 13, 11, 12]. Herlihy [14] proposed a hardware-based transactional memory for multiprocessor systems; a software implementation of this was proposed by Shavit and Touitou [23], and more efficient implementations were proposed recently [13, 11]. Minitransactions are a generalization of the swap and compare-and-swap instructions, and of the multiword compare-and-swap operation [12], which were envisioned for multiprocessor shared memory systems.

Transaction support on a disk-based system was proposed in Mime [7], which provided multi-sector atomic writes and the ability to revoke tentative writes; however, all the disks in Mime were accessed through a single controller.

There is a rich literature on distributed file systems, including several that are built over a high-level infrastructure designed to simplify writing distributed applications. The Boxwood project [17] builds a cluster file system over a distributed B-tree abstraction. Boxwood also shares with Sinfonia the goal of providing a high-level abstraction for use by applications, but focusses on abstractions to serve as the fundamental storage infrastructure. The Inversion File System [18] is built over a Postgres database; this is a fairly complex abstraction, and the performance of the file system was substantially lower than a native NFS implementation.

Group communication [8] and virtual synchrony [4] in particular is a basic abstraction for the design of fault-tolerant distributed applications. Scalable diffusion protocols that are fairly reliable have been proposed (e.g., [3]), but require some external ordering mechanism. Sinfonia's shared memory abstraction provides a medium for reliable ordered information dissemination with some scalability.

9 Conclusion

We proposed a new approach to build distributed systems that draws ideas from database systems and distributed shared memory—technologies that have not been successfully applied to low-level applications where performance is critical, such as file systems and group communication. Sinfonia relies on a new form of two-phase commit optimized for its assumptions. The main benefit of Sinfonia is that it can shift concerns about failures and parallel protocol design into an algorithmic problem of designing concurrent data structures. We do not believe that applications built with Sinfonia will necessarily provide better performance than those built without Sinfonia—indeed, the latter could just reimplement a tailored version of Sinfonia and achieve at least as good performance. However, we have shown that it is possible to obtain reasonably competitive designs if Sinfonia is used properly.

References

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

[2] C. Amza et al. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[3] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.

[4] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM Symposium on Operating Systhem Principles (SOSP)*, pages 123–138, Austin, TX, USA, November 1987.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *SOSP*, pages 152–164, 1991.

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

[7] C. Chao et al. Mime: a high performance storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9, Concurrent Systems Project, HP Laboratories, Palo Alto, CA, Nov. 1992.

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

[9] P. Dasgupta et al. The clouds distributed operating system. *IEEE Computer*, 24(11):34–44, 1991.

[10] B. Gallagher, D. Jacobs, and A. Langen. A high-performance, transactional filestore for application servers. In *SIGMOD*, pages 868–872, 2005.

[11] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.

[12] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, pages 265–279, 2002.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[15] K. Li. Ivy: A shared virtual memory system for parallel computing. In *ICPP*, pages 94–101, Aug. 1988.

[16] B. Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.

[17] J. MacCormick et al. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, pages 105–120, 2004.

[18] M. A. Olson. The design and implementation of the inversion file system. In *USENIX Winter*, pages 205–218, 1993.

[19] L. G. Reid and P. L. Karlton. A file system supporting cooperation between programs. In *SOSP*, pages 20–19, 1983.

[20] M. Satyanarayanan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Computers*, 39(4):447–459, 1990.

[21] M. Satyanarayanan et al. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, 1994.

[22] F. B. Schmuck and J. C. Wyllie. Experience with transactions in quicksilver. In *SOSP*, pages 239–253, 1991.

[23] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

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