Sinfonia: a new paradigm for building scalable distributed systems

Marcos Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, Christos Karamanolis

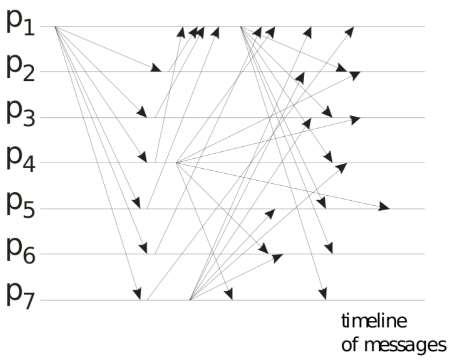
(SOSP 2007 Best Paper)

Motivation

- Corporate data centers are growing quickly
  - Companies building large data centers
  - Tens of thousands of servers
  - Businesses want to serve the world
- Need distributed applications that scale well

Current distributed applications often involve complex protocols

- Message-passing paradigm: processes share data by passing messages over network
  - Error-prone and hard to use (complex protocols to handle distributed state)



Focus

- Systems within a data center
  - Network latencies usually small and predictable
  - Nodes may crash, sometimes all of them
  - Stable storage may crash too
- Infrastructure applications
  - Applications that support other applications
  - Reliable, fault-tolerant, consistent
  - Examples: cluster file systems, distributed lock managers, group communication services, distributed name services

Design Principles

- Principle 1: Reduce operation coupling to obtain scalability
  - *Sinfonia* does this by not imposing structure on the data it services
- Principle 2: Make components reliable before scaling them
  - Individual *Sinfonia* nodes are fault-tolerant

Approach to Developing Distributed Applications

- Developers use *Sinfonia*, a data sharing service
  - Data stored in memory nodes, each exporting a linear address space
  - No structure on data imposed by Sinfonia
  - Streamlined minitransactions
- Transform problem of protocol design into easier problem of shared data structure design

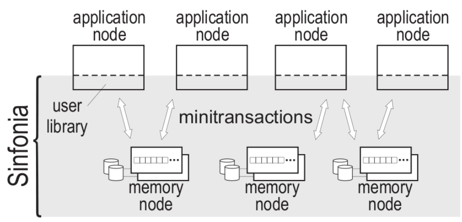
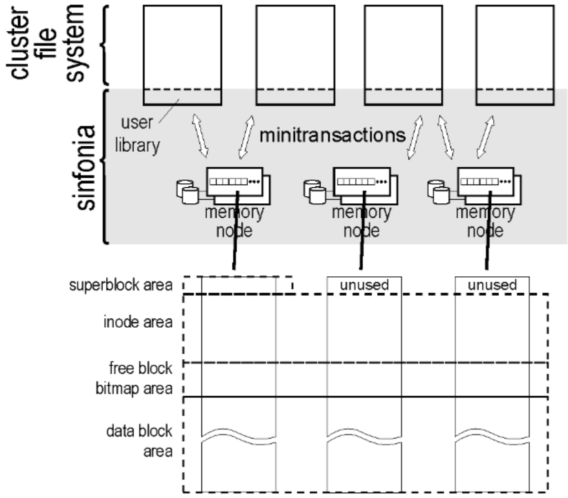


Figure 1: Sinfonia allows application nodes to share data in a fault tolerant, scalable, and consistent manner.

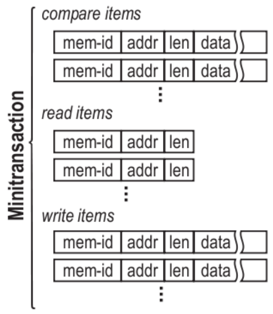
Example application: cluster file system



Sinfonia Minitransactions

- Operate on data at memory nodes
- Provide ACID properties
  - Atomicity, consistency, isolation, durability
- Designed to balance power and efficiency
- Efficiency
  - Few network roundtrips to execute
- Power
  - Flexible, general-purpose, easy to understand and use
- Result
  - A lightweight, short-lived type of transaction over unstructured data

Minitransaction in Detail



Semantics of a minitransaction

- check data indicated by *compare items* (equality comparison)
- if all match then
  retrieve data indicated by *read items*
  modify data indicated by *write items*

API

```
class Minitransaction {
 public:
  void cmp(memid,addr,len,data); // add cmp item
  void read(memid,addr,len,buf); // add read item
  void write(memid,addr,len,data); // add write item
  int exec_and_commit();     // execute and commit
};
```

Example

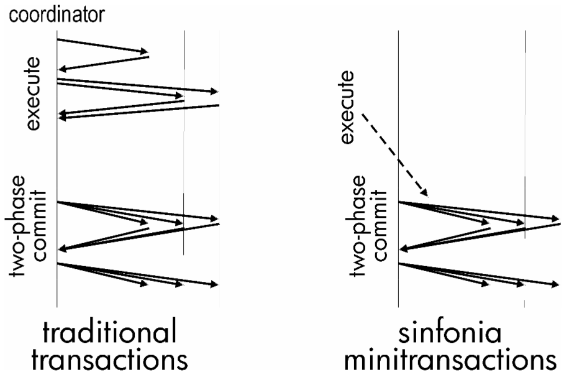
```
...
t = new Minitransaction;
t->cmp(memid, addr, len, data);
t->write(memid, addr, len, newdata);
status = t->exec_and_commit();
...
```

Power of Minitransactions

- Examples of what one minitransaction can do
  - Atomic swap operation
  - Atomic read of many data
  - Try to acquire a lease
  - Try to acquire multiple leases atomically
  - Change data if lease is held
  - Validate cache then change data (e.g., optimistic concurrency control)

Minitransaction Efficiency:

- Piggybacking execution onto two-phase commit

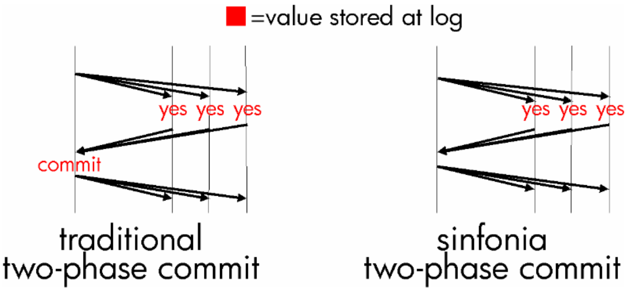


Minitransaction Efficiency: running at application node

- Commit coordinator runs at application node
  - To save a network roundtrip
- Problem: coordinator may crash and not recover
  - Application node outside of Sinfonia control
- Cannot block transactions forever in this case
- 3-phase commit expensive
- Solution: a new two-phase commit protocol

New Two-Phase Commit

- Transaction committed iff all participant memory nodes log “yes” vote
  - Compare: transaction committed iff coordinator logs “commit” decision
- But: transaction blocks while memory node is crashed
- Recovery and garbage collection more involved (see paper)



Other Features of Sinfonia

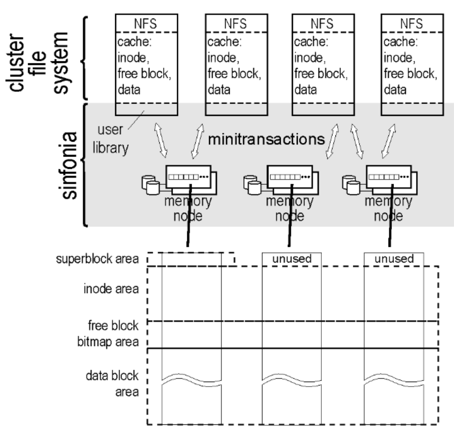
- Configurable fault tolerance
  - Cost, performance, resiliency trade offs
- Memory node replication
  - Can have mirrors of memory nodes for better availability
- Transactional backups
  - Ways to capture a transactionally consistent full image
- Debugging facilities
  - Transaction log (gc disabled) to review past

Using Sinfonia: Applications

- sinfoniaFS: Cluster File System
  - Hosts share the same set of files, files stores in Sinfonia
  - Fault tolerant
  - Scalable: performance improves with more memory nodes
- sinfoniaGCS: group communication service
  - “chat room” for distributed applications
  - Nodes can join or leave the room, notifications of who joins/leaves
  - Nodes can broadcast messages to room, messages totally ordered

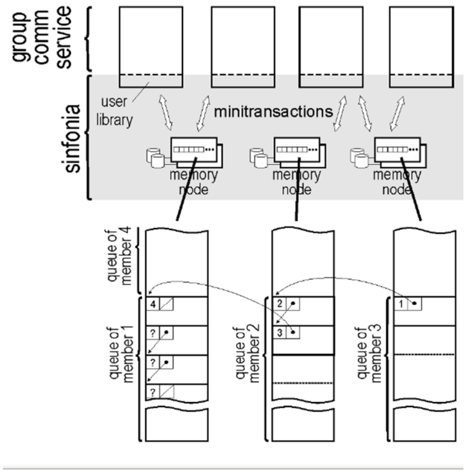
sinfoniaFS Design

- Exports NFS interface
- Each NFS operation: one minitransaction
- General Template:
  - Validate cache (cmp items)
  - Modify data (write items)



sinfoniaGCS Design

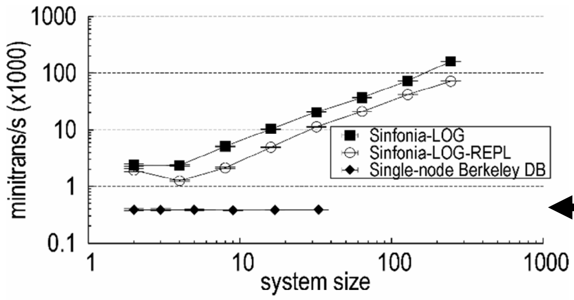
- Each member has private queue in Sinfonia
- Broadcast message:
  - Copy msg to queue
  - Thread msg in global order
- Join or leave:
  - Acquire lease
  - Update member list
  - Release lease



Evaluation

- Sinfonia service
  - Scalability
  - Performance under contention
  - Ease of use
  - Paper: benefits of streamlined minitransactions (1.4x to 11.1x throughput improvement)
- Cluster file system application
  - Performance and scalability
- Group communication application
  - Performance and scalability

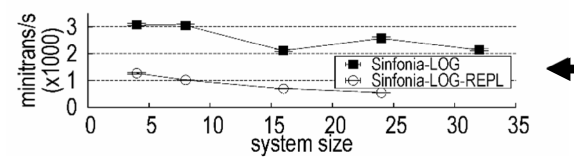
Sinfonia service: scalability



Minitransaction spread: number of memory nodes in a minitransaction

Minitransaction spread=2:

Usually within 85% of ideal scalability

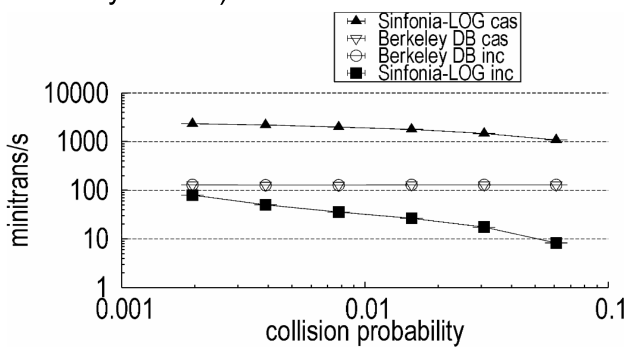


Minitransaction spread=all memory nodes (increases with system size):

No scalability

Sinfonia service: contention

- Two workloads
  - Compare-and-swap (cas) – direct minitransaction support
  - Increment (inc) – requires caching and retrying (optimistic concurrency control)



Sinfonia: Ease of Use

- Software engineering metrics

	sinfoniaFS	linuxNFS	sinfoniaGCS	spread toolkit
lines of code (language)	3,855 (C++)	5,900 (C)	2,492 (C++)	22,148 (C)
develop time	1 month	unknown	2 months	years
major versions	1	2	1	4

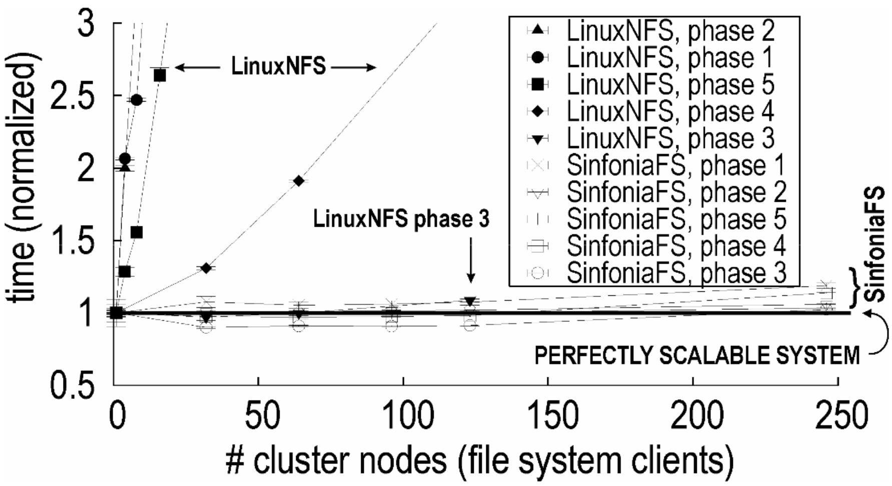
Sinfonia: Ease of Use

- Advantages
  - Transactions: relief from concurrency, failure issues
  - No distributed protocols, no timeout worries
  - Correctness verified by checking minitransactions
  - Minitransaction log useful for debugging
- Drawbacks
  - Address space is low-level abstraction
  - Had to lay out data structures manually
  - Had to find efficient layout to avoid contention (data structure design problem)

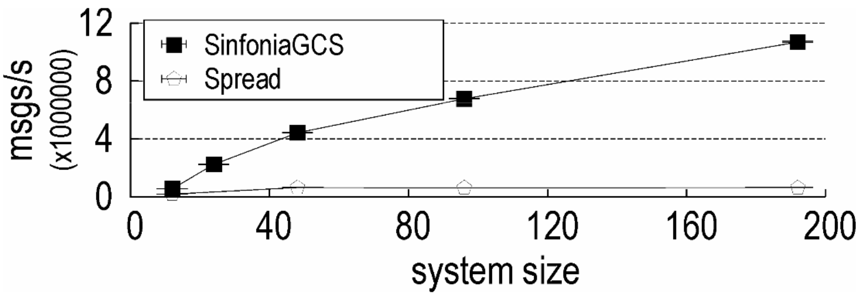
sinfoniaFS: base performance

- First considered 1-memory-node system
- Benchmarks
  - Modified andrew (tcl source code)
  - Connectathlon NFS testsuite
- sinfoniaFS performs as well as linuxNFS (details in paper)

sinfoniaFS: scalability



sinfoniaGCS: scalability



Related Work

- Database systems
- Distributed shared memory
  - Lots, plurix [Fakler et al 2005], perdis [Ferreira et al 2000]
- Camelot, coda
- Mime [Chao et al 1992]
- Thor [Liskov et al 1999]
- Sdds [Gribble et al 2000]
- Boxwood [McCormick et al 2004]
- Stasis [Sears, Brewer 2006]
- Gfs, bigtable, chubby, mapreduce [200x]

conclusions

- Sinfonia: a scalable data sharing service for building distributed applications
- Main characteristics
  - Unstructured address spaces: no unnecessary structure
  - Streamlined minitransactions
- General paradigm: built very different apps with it
- Main benefits
  - Minitransactions hide complexities of concurrency and failures (while providing good performance, fault tolerance and scalability)
  - No protocols to worry about