Sinfonia: A New Paradigm For Building Scalable Distributed Systems

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

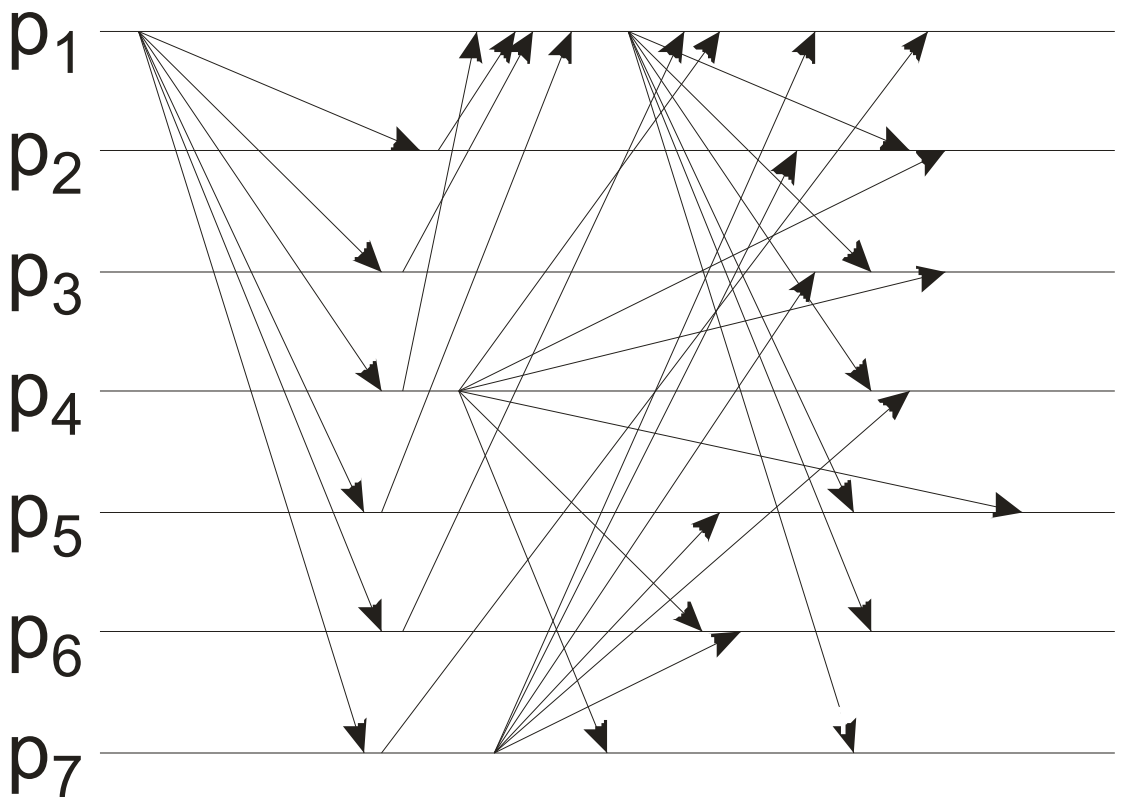
ECE1724

Authors: Marcos k. Aguilera, Arif Merchant, Mehul Shah,
Alistair Veitch, Christos Maramanolis

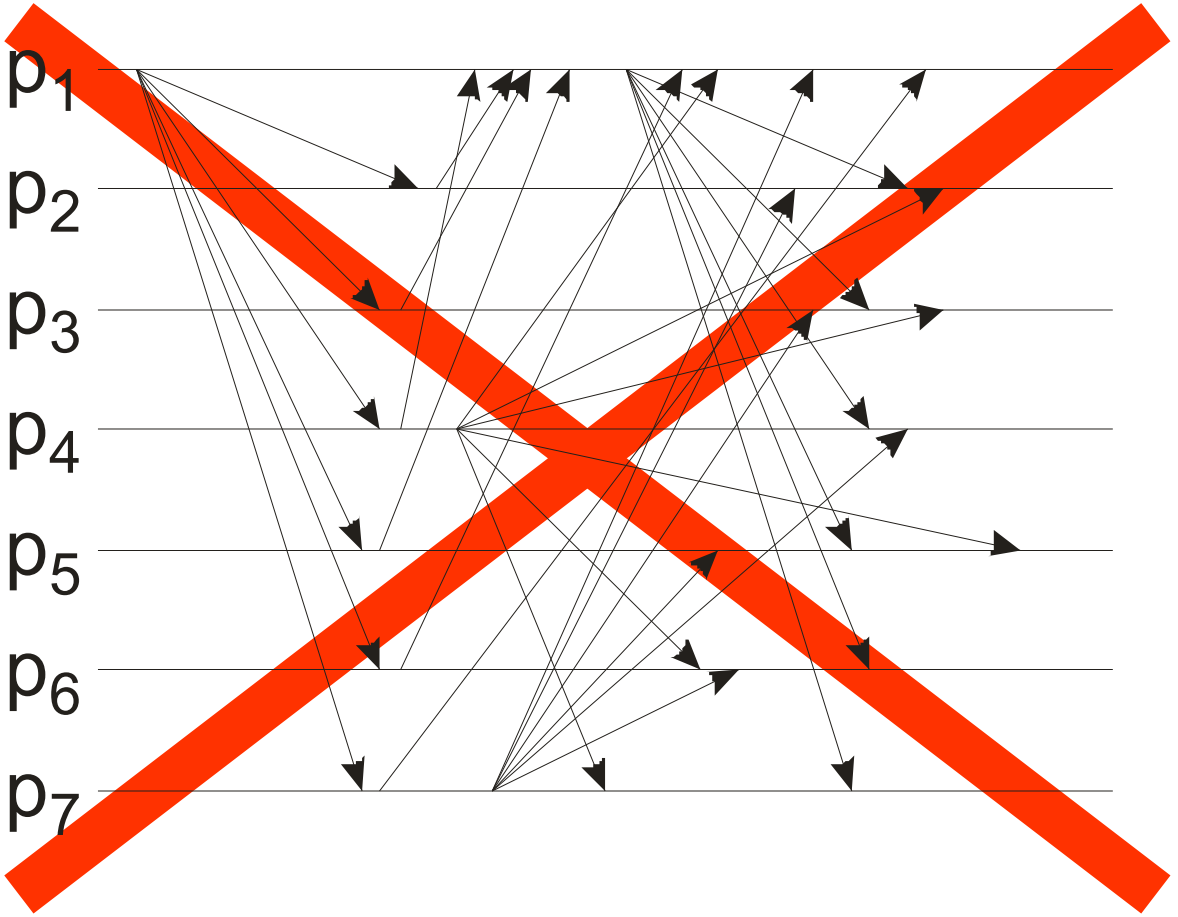
Motivation

- Corporate data centers are growing quickly
  - Companies building large data centers
  - 10000s servers and more
- Need distributed applications that scale well

Current Distributed Applications Often Involve Complex Protocols



Wouldn't it be Nice to Avoid Such Protocols?



Focus

- Systems within a data center
  - Network latencies usually small and predictable
  - Nodes may crash, sometimes all of them
  - Stable storage may crash too

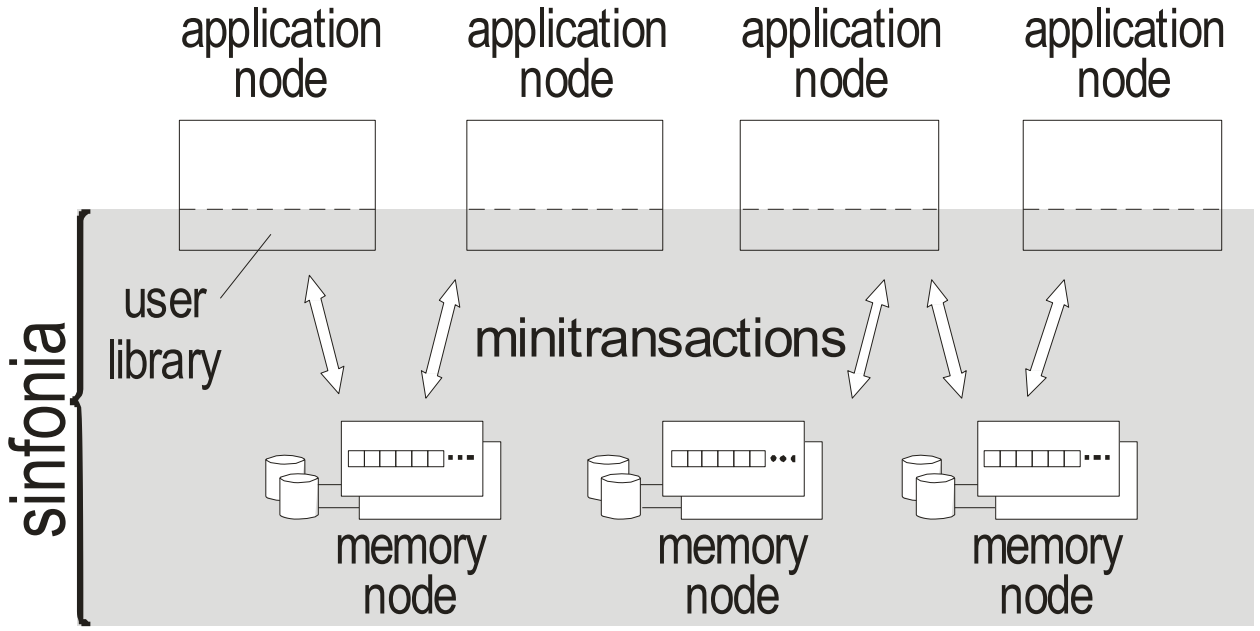
- Infrastructure applications
  - Applications that support other applications
  - Reliable, fault-tolerant, consistent
  - Examples: cluster file systems, distributed lock managers, group communication services, distributed name services

Sinfonia's Approach

- Developers use Sinfonia, a data sharing service
- Sinfonia stores data in memory nodes
  - Each memory node exports a linear address space
  - No structure (e.g., file, rows, etc.) is imposed
- Key idea: mini-transactions
  - Enables accessing and modifying data atomically
  - Performs entire transaction within the commit protocol
  - Reduces messages, locking times compared to traditional txns
- Transforms problem of distributed application and protocol design into easier problem of fault-tolerant, consistent, shared data structure design

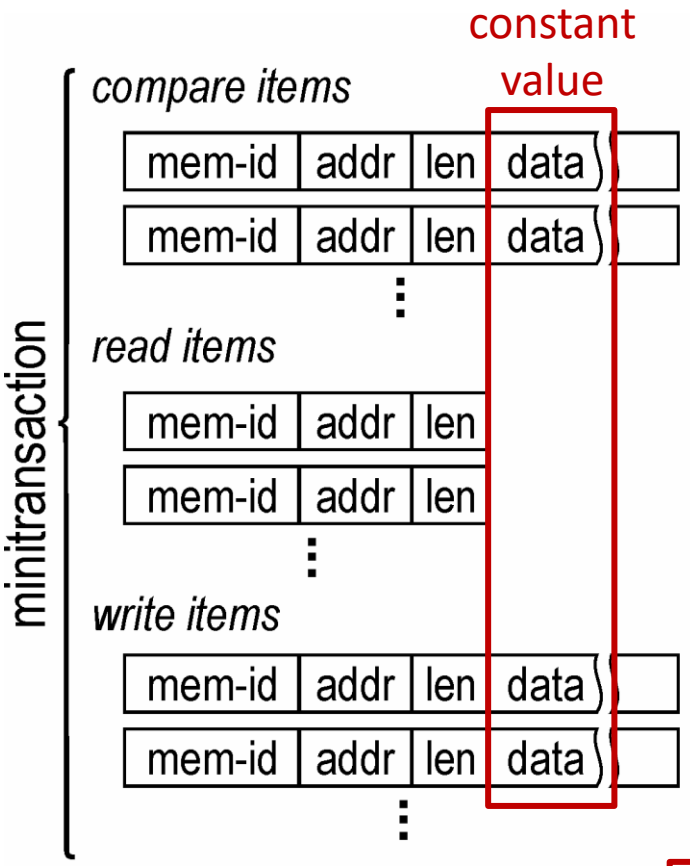
Sinfonia Architecture

- Applications run on application nodes
  - Use a Sinfonia library to access their data
- Application data stored on memory nodes (with disks)
  - Memory node location visible to applications



Sinfonia Mini-Transaction

- A limited type of distributed transaction that operates on unstructured data at memory nodes, providing ACID guarantees efficiently



semantics

- read (and return) data indicated by *read items*
- check data indicated by *compare items* (equality comparison)
- if ALL match then
  modify data indicated by *write items*

example

```
t = new Minitransaction;
t->cmp(hostX, addrX, len, 1);
t->write(hostY, addrY, len, 2);
t->write(hostZ, addrZ, len, 2);
status = t->exec_and_commit();
```

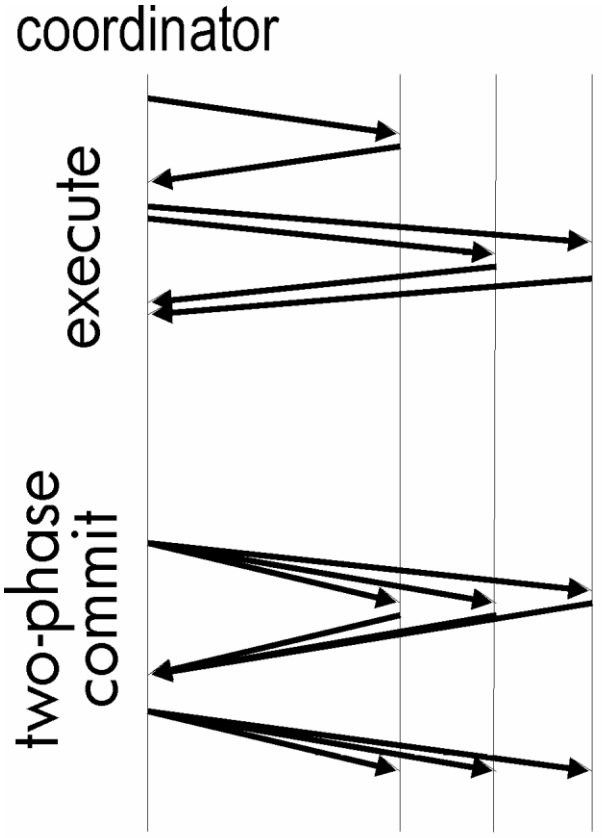
transaction executes+commits →

Mini-Transaction Requirements

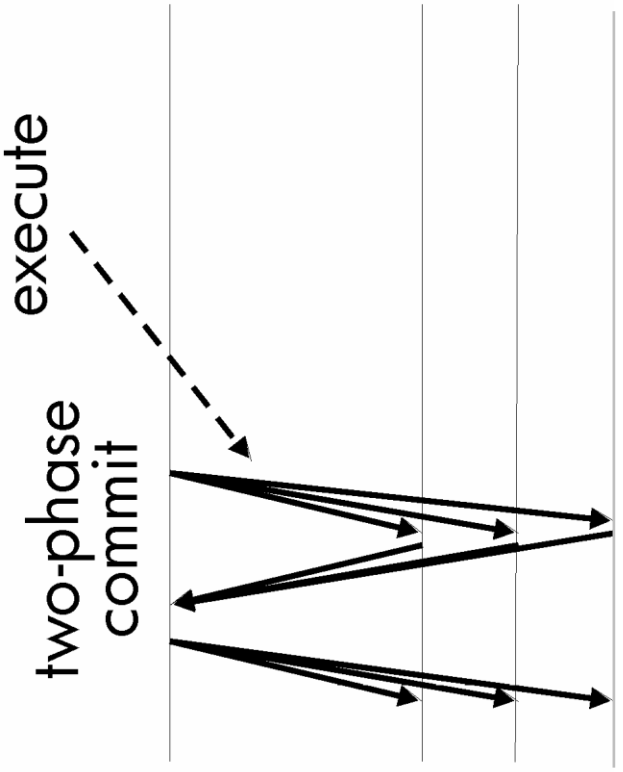
- Two restrictions: 1) read-write sets known before txn execution, 2) all writes conditioned on checks
- Many common operations meet these requirements
  - atomic compare-and-swap operation
  - atomic read of many data items
  - try to acquire a lease
  - change data if lease is held
  - validate cache then change data, e.g., similar to OCC
- Serve as building blocks for higher-level applications

Efficiency of Mini-Transactions

- Piggyback txn execution onto two-phase commit



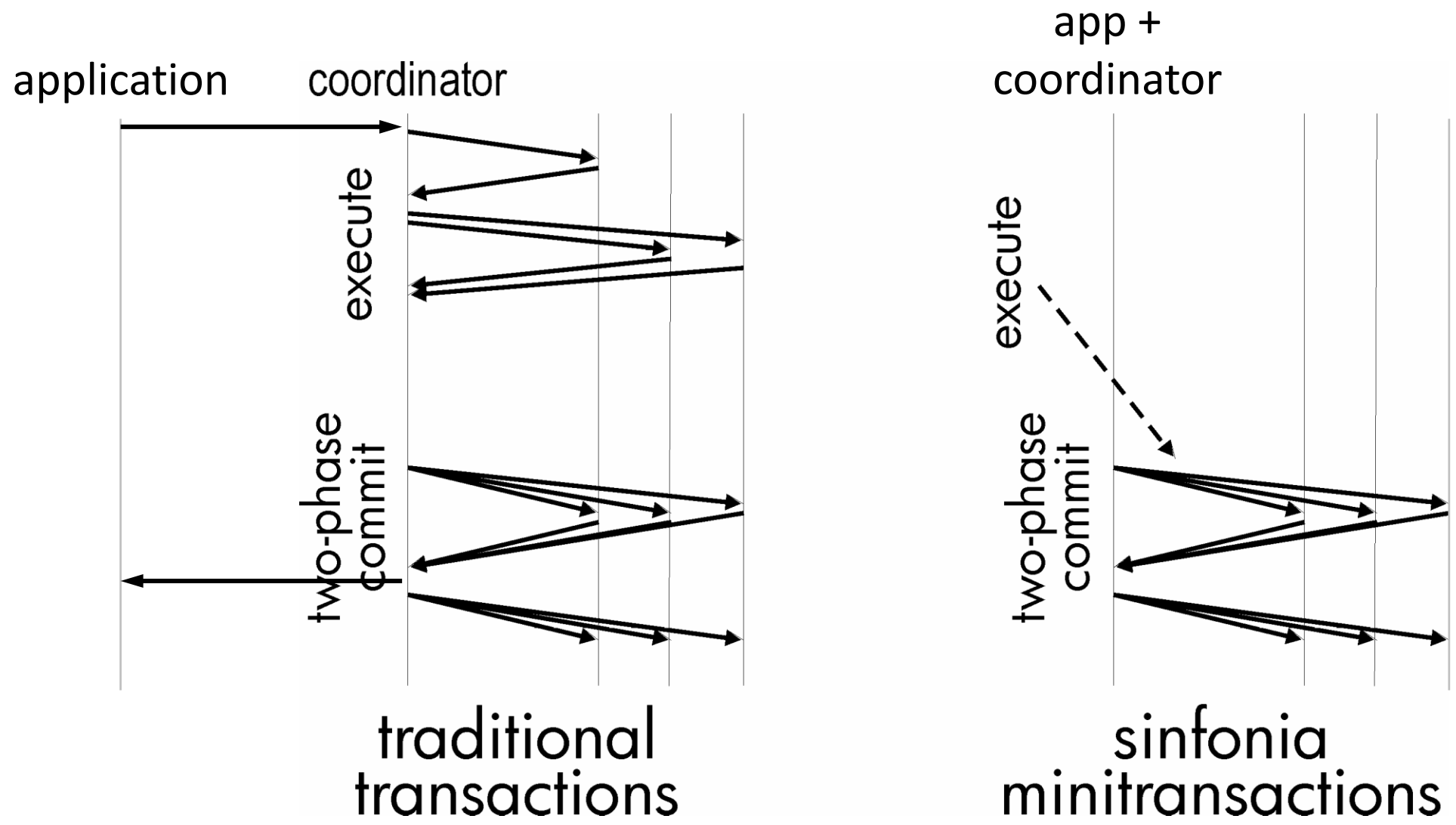
traditional transactions



sinfonia minitransactions

Efficiency of Mini-Transactions

- Run coordinator at application node

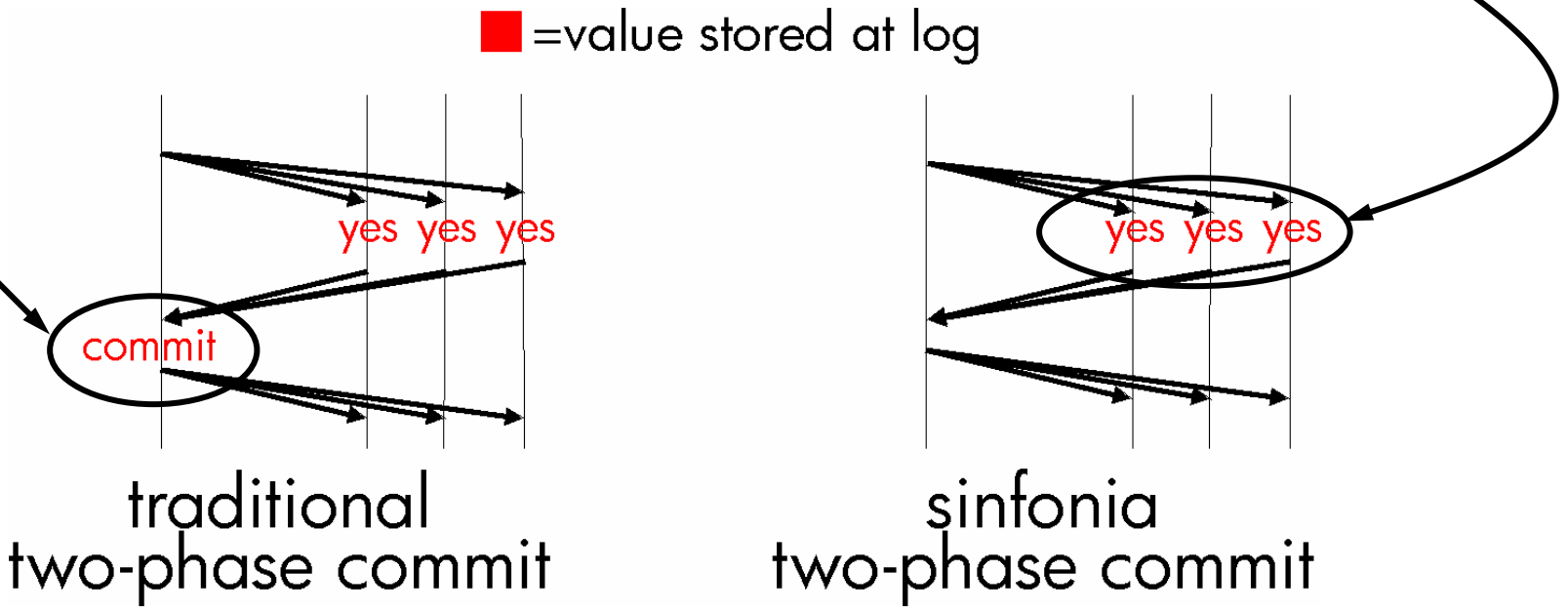


What if Coordinator Crashes?

- Application node is outside of Sinfonia control
  - Application node, coordinator may crash and not recover
- Traditional two-phase commit can block in this case
  - Why?
- Solution: new two-phase commit protocol that avoids blocking on coordinator failure

Sinfonia's Two-Phase Commit

- Traditional two-phase commit: transaction commits iff coordinator logs "commit" decision
- Sinfonia two-phase commit: transaction commits iff all participant memory nodes log "yes" vote



Commit Protocol – Phase 1

- Phase 1 (prepare)
  - Coordinator (on app node) generates unique transaction id, sends mini-transaction to the participants
  - Each participant
    - Tries to acquire read and write locks for all required addresses (at word granularity + range locking)
    - If it acquires all locks, it performs reads, comparisons
    - If all comparisons succeed, it logs writes in a (buffered) redo log to disk
    - Replies to coordinator with commit/abort vote

Commit Protocol – Phase 2

- Phase 2 (commit)
  - Coordinator tells participants to commit if and only if all participants voted to commit
  - Each participant
    - If committing, applies logged write items to their in-memory structure
    - Releases all locks

Fault Tolerance in Sinfonia

- Memory node replication for better availability
  - Memory nodes can be replicated
  - Sinfonia uses primary-backup replication
- Disk image for crash recovery
  - A copy of data in memory node on disk for crash recovery
    - Updated asynchronously, “fuzzy”
    - Use idempotent log replay for recovery
- Transactional backups for disaster recovery
  - Capture a transactionally consistent snapshot at all nodes
    - Snapshots contain all updates up to some transaction, and
    - No updates from later transactions

Recovery From Participant Crashes

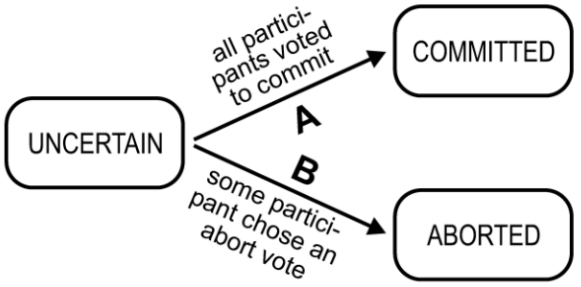
- Sinfonia blocks any mini-transaction commits that access data on a crashed participant, why?
- On recovery, participant loads disk image, replays redo log to reconstruct memory state
- If participant loses its stable storage, it must be recovered from a transactional backup
- If participant voted commit for an ongoing transaction, it needs to ask all other participants about their vote to finish phase 2 of transaction commit

Challenges With Coordinator Crashes

- When a coordinator crashes, participant may be holding locks and have logged updates for transactions that are not known to be aborted or committed (uncertain)
- Recovery and garbage collection are complicated since commit information is distributed among participants
  - Coordinator recovery needs to gather votes from participants
  - Garbage collection of logs at a participant requires knowing whether the participant's votes are known to all participants

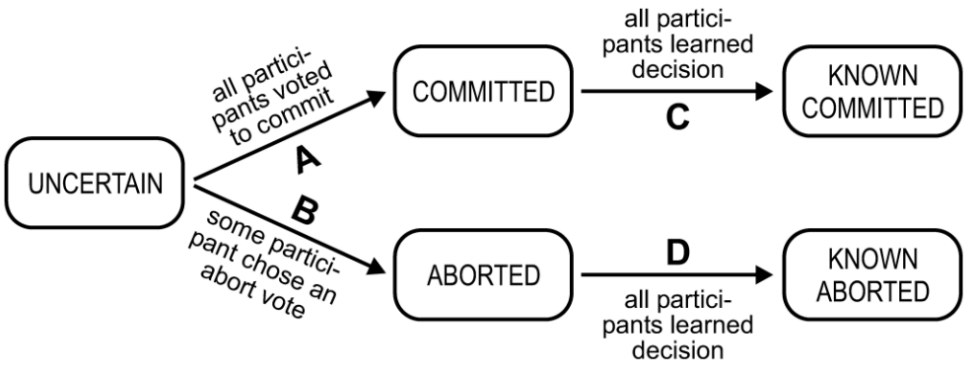
Recovery From Coordinator Crashes

- Key idea: use a separate recovery coordinator to check if transition A has occurred, otherwise force transition B



Recovery From Coordinator Crashes

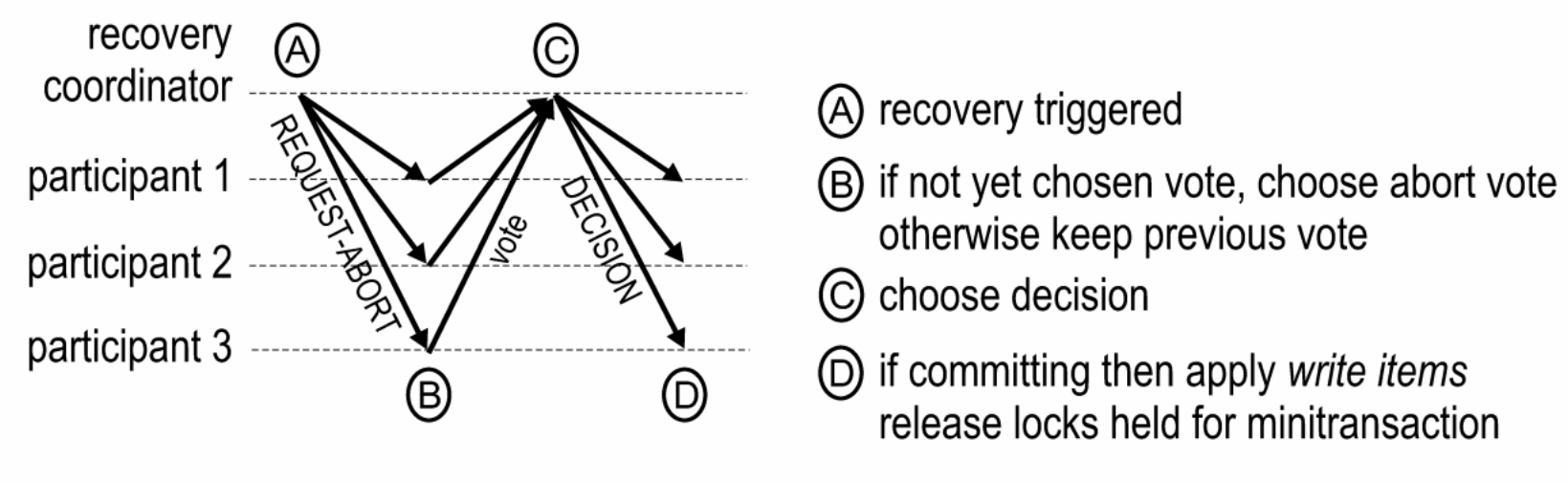
- Key idea: use a separate recovery coordinator to check if transition A has occurred, otherwise force transition B



- Transitions C and D occur when all participants receive the decision from the coordinator

Recovery From Coordinator Crashes

- Recovery coordinator periodically probes memory node logs for uncertain transactions
  - Forces participants to abort txn unless they have voted already
  - Decides to commit transaction if all participants voted “yes”

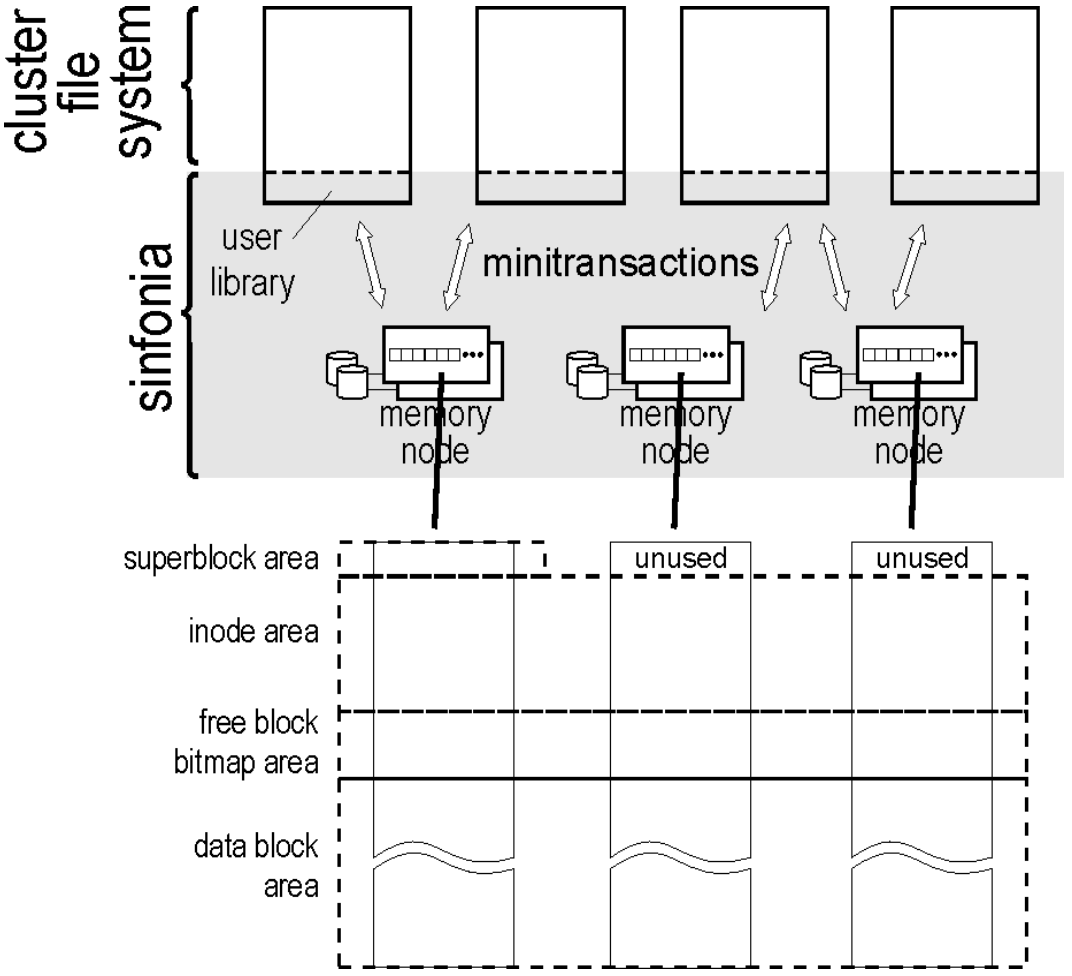


Sinfonia Applications

- sinfoniaFS: cluster file system
  - Applications share files, files stored in Sinfonia
  - Fault tolerant
  - Scalable: performance improves with more memory nodes
- sinfoniaGCS: group communication service (chat room)
  - Chat room messages stored in Sinfonia
  - Processes can join/leave room,
    notifications sent for joins/leaves
  - Processes broadcast messages to room,
    messages delivered to processes in total order

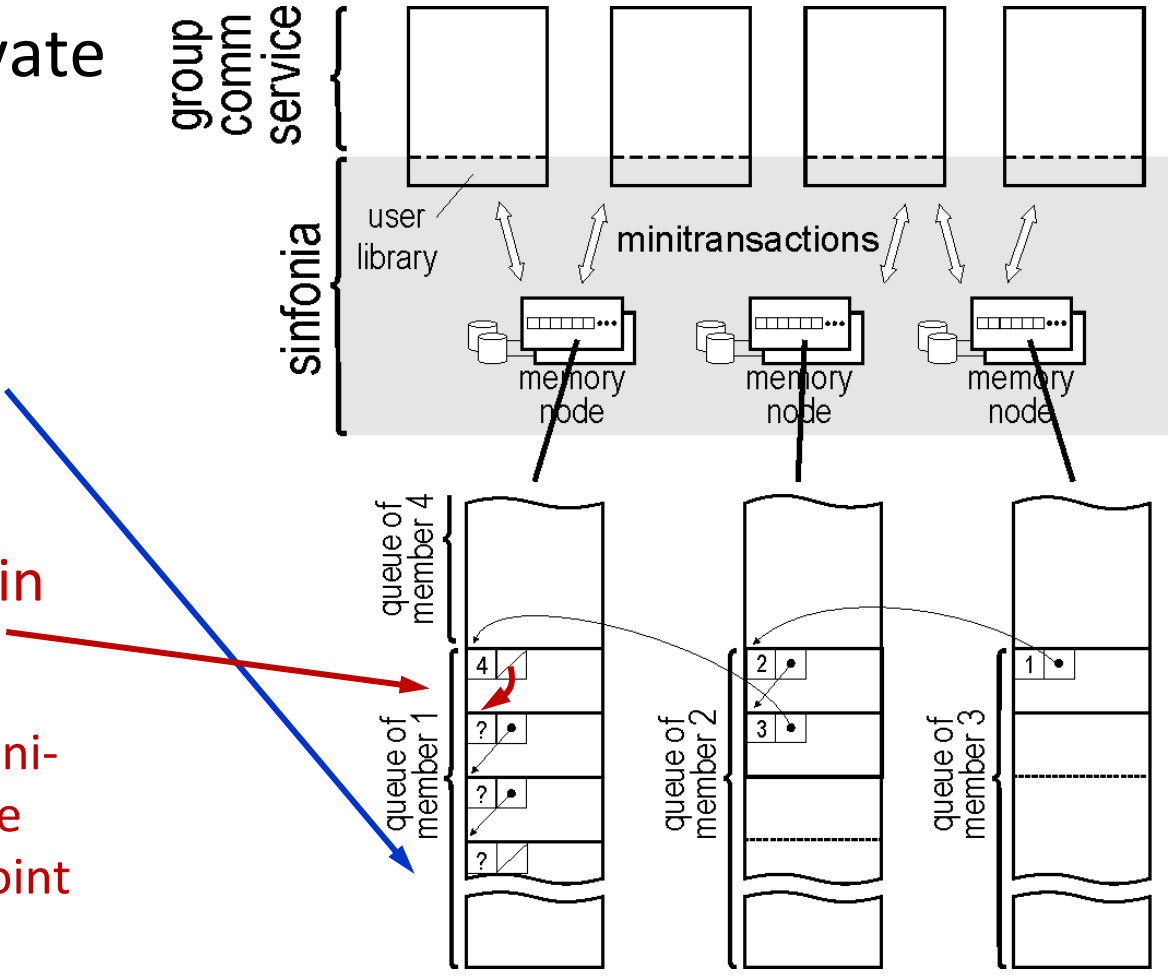
sinfoniaFS

- Exports NFS interface
- Each NFS operation is one minitransaction
- Files may be are stored across memory nodes
- General template:
  - validate cache (cmp items)
  - modify data (write items)



sinfoniaGCS

- Each member has private queue in sinfonia
- broadcast msg:
  - copy msg to queue
    - Private operation
  - thread msg in batches in global order
    - compute global tail, mini-transaction updates the tail's next pointer to point to the messages

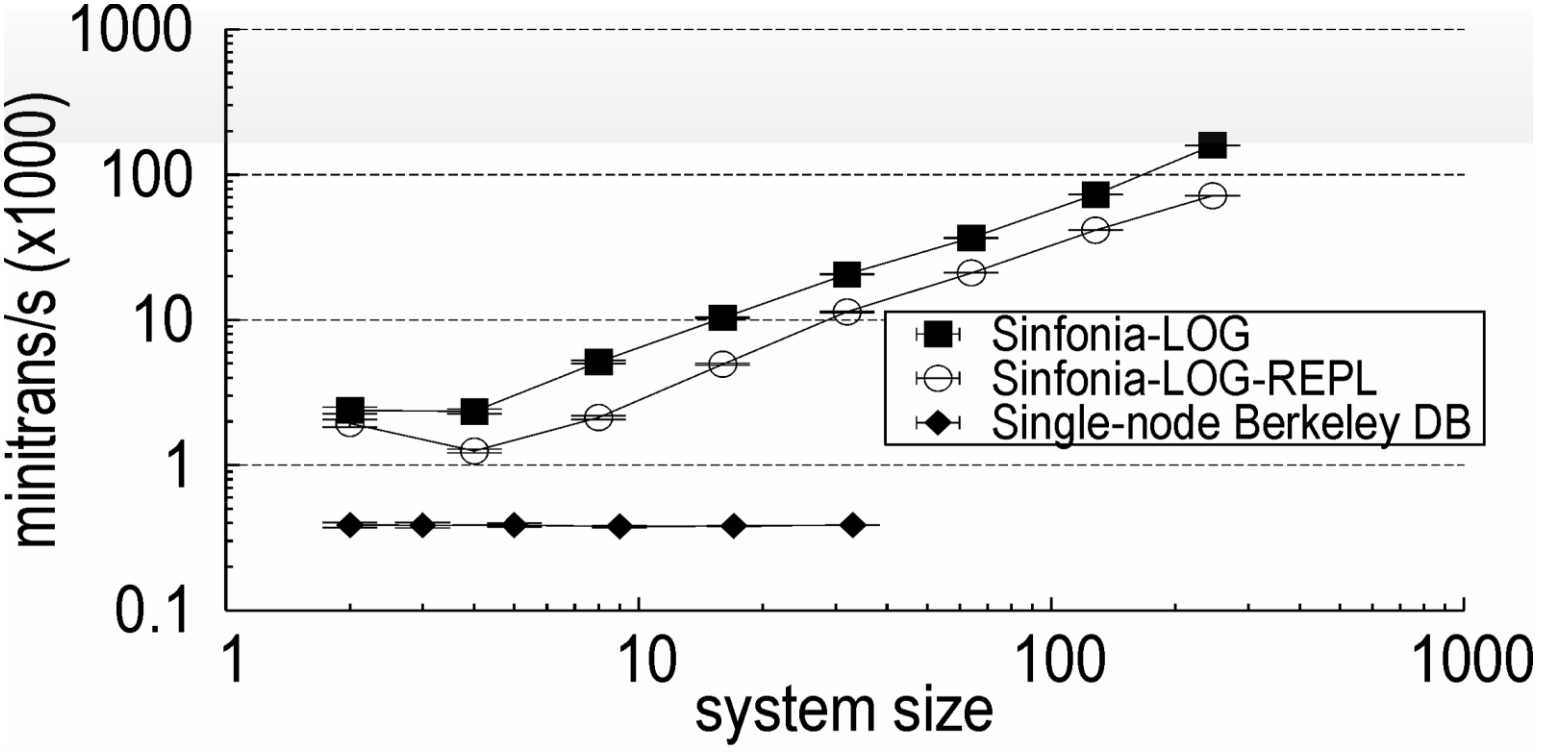


Evaluation

- Sinfonia scalability
- sinfoniaFS scalability
- sinfoniaGCS scalability

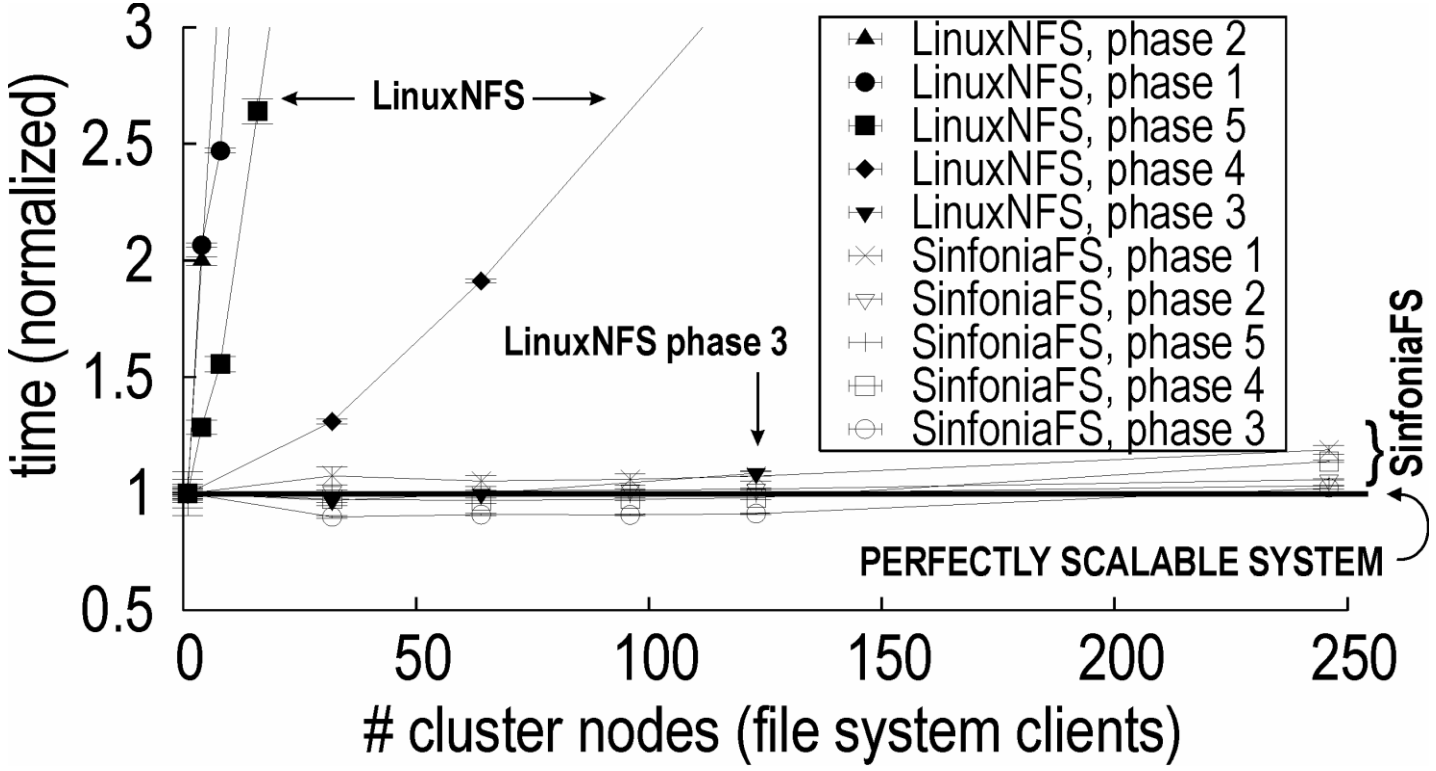
Sinfonia Scalability

- Each mini-transaction accesses two memory nodes
- Usually within 85% of ideal scalability

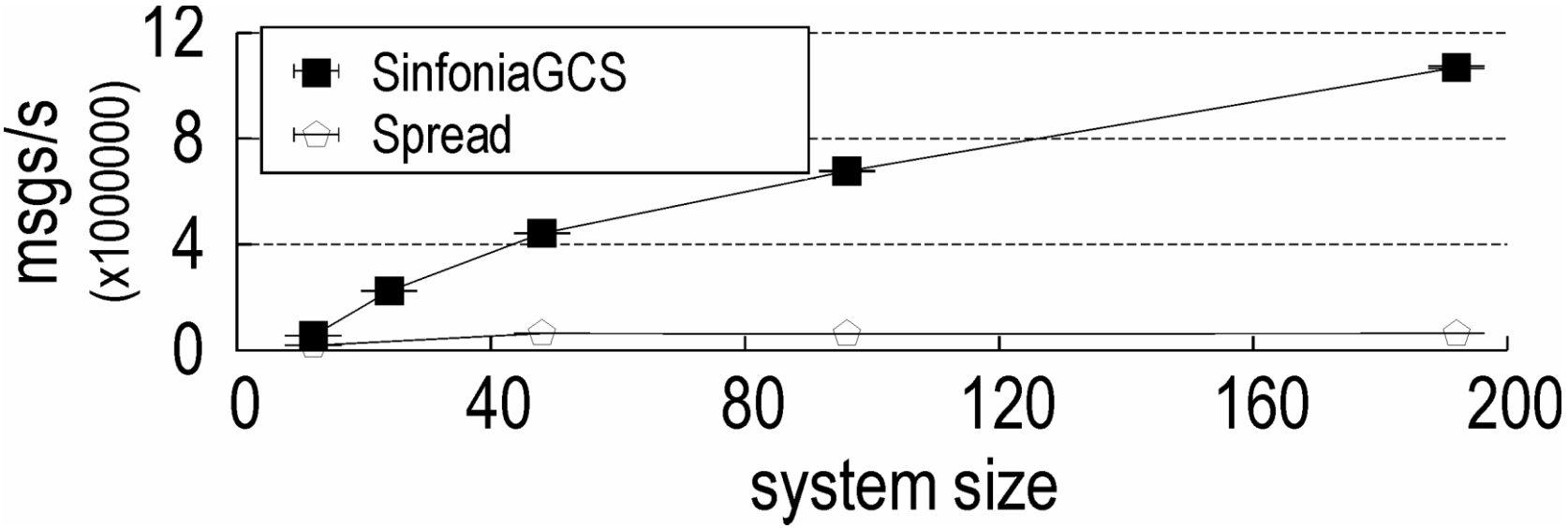


sinfoniaFS Scalability

- sinfoniaFS performs comparably to LinuxNFS with one memory node
- sinfoniaFS scales much better than LinuxNFS with increasing number of clients



sinfoniaGCS Performance



Sinfonia Ease of Use

- Advantages
  - Transactions: relief from concurrency, failure issues
  - No distributed protocols, no timeout worries
  - Correctness verified by checking minitransactions

- Drawbacks
  - Address space is low-level abstraction
  - Need to lay out data structures manually
  - Need to find efficient layout to avoid contention
    - A data structure design problem

Conclusions

- Sinfonia: a scalable data sharing service for building distributed applications
- Exposes unstructured memory address spaces
- Enables efficient mini-transaction mechanism that
  - Hides the complexities of concurrency and failures
  - While providing good performance and scalability

Discussion

Q1

- Why is this transaction not directly supported in Sinfonia?

```
transfer(A, B):
begin_tx
a = read(A)
if a < 10 then
    abort_tx
else
    write(A, a-10)
    b = read(B)
    write(B, b+10)
    commit_tx
```

Q2

- Can you implement a Sinfonia function that implements the transfer(A, B) transaction?

```
transfer(A, B):
begin_tx
a = read(A)
if a < 10 then
   abort_tx
else
   write(A, a-10)
   b = read(B)
   write(B, b+10)
   commit_tx
```

```
Mini-Transaction API:

t = new Minitransaction;
t->read(host, data_addr, data_len, &data);
t->cmp(host, data_addr, data_len, data);
t->write(host, data_addr, data_len, data);
t->exec_commit();
```

Q3

- Will Sinfonia's transaction protocol always perform better than traditional transactions and two-phase commit?

Q4

- Compare consensus and atomic commit in terms of
  - The fundamental difference between them
  - How and why are they used?
  - Liveness guarantees under failures?

Q5

- What are the most significant similarities and differences between the BigTable database and Sinfonia?