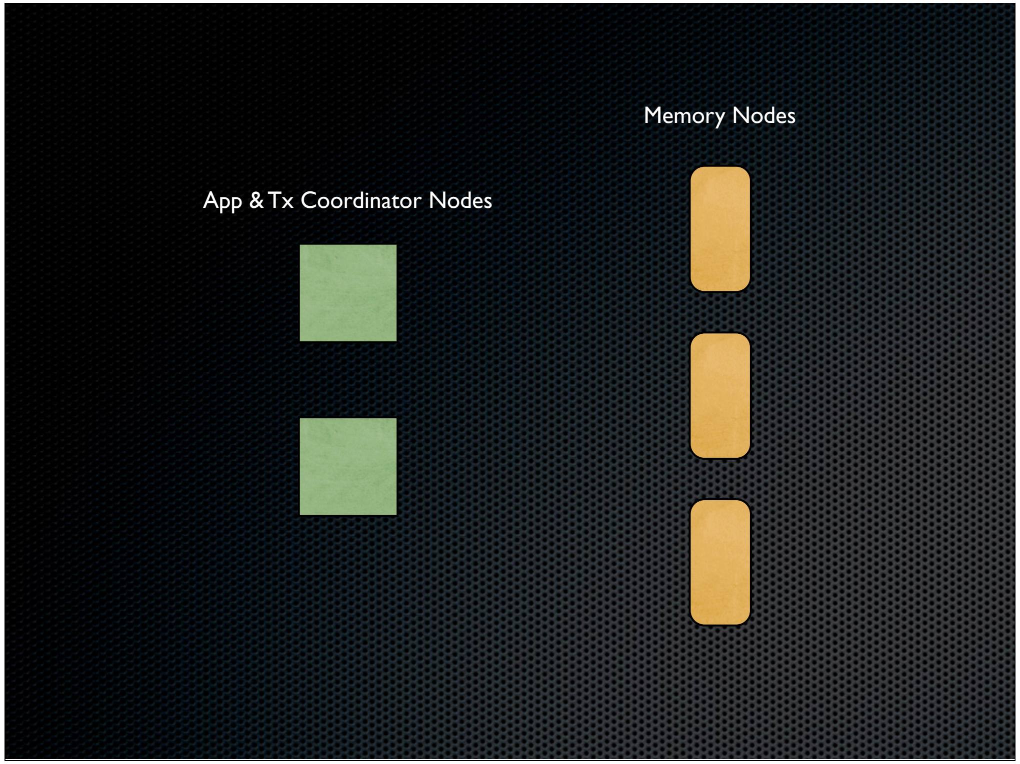
Sinfonia

a new paradigm
for building scalable
distributed systems

(SOSP 2007)

Marcos Aguilera
Arif Merchant
Mehul Shah

Alistair Veitch
Christos Karamanolis

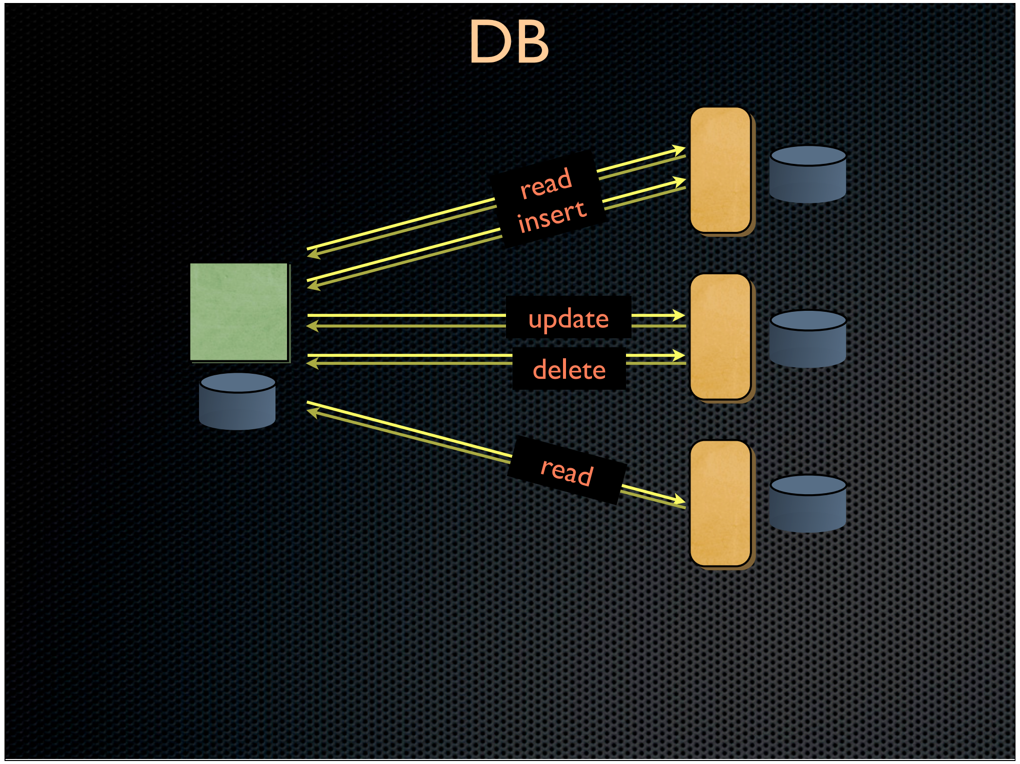


On the face of it, we have transaction coordinators alongside the application, and memory nodes to store the data. Is it just another ACID store that forces 2PC on you?

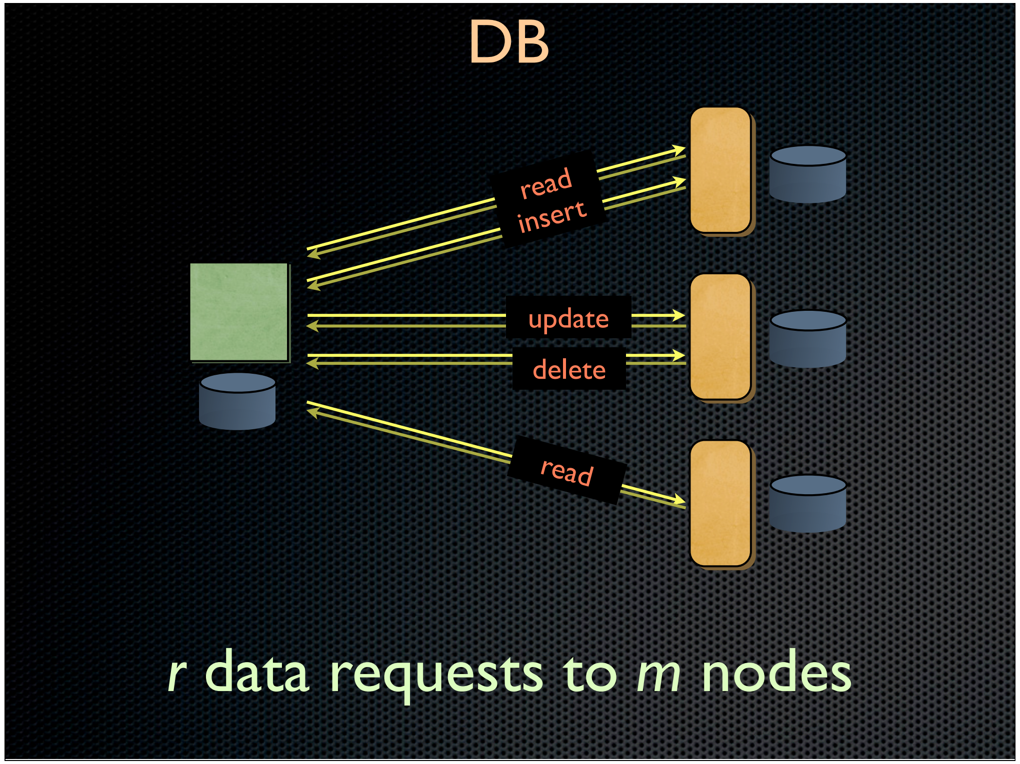
What I found interesting about this paper is that other approaches avoid 2pc at all costs; instead they relax consistency requirements (many use memcached) or avoid multi-word transactions (bigtable uses single row transactions)

On the face of it, we have transaction coordinators alongside the application, and memory nodes to store the data. Is it just another ACID store that forces 2PC on you?

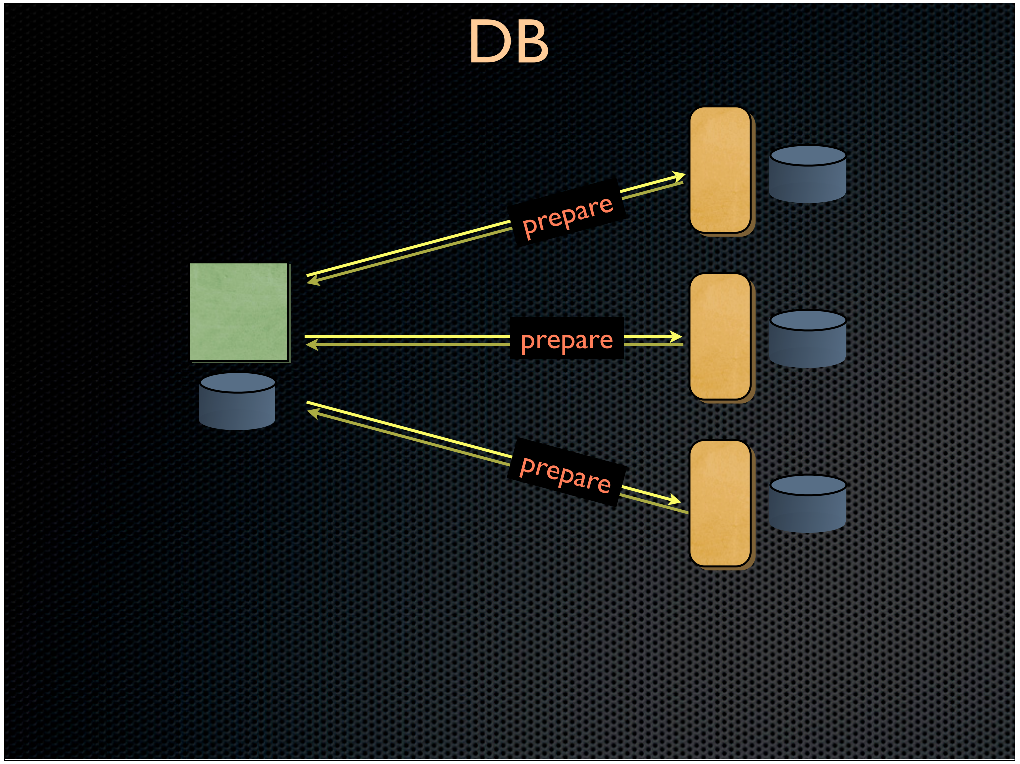
What I found interesting about this paper is that other approaches avoid 2pc at all costs; instead they relax consistency requirements (many use memcached) or avoid multi-word transactions (bigtable uses single row transactions)



A brief recap about db operation and 2pc.
App begins a transaction, makes r requests (and touches m nodes in the process).
DB often locks the data.

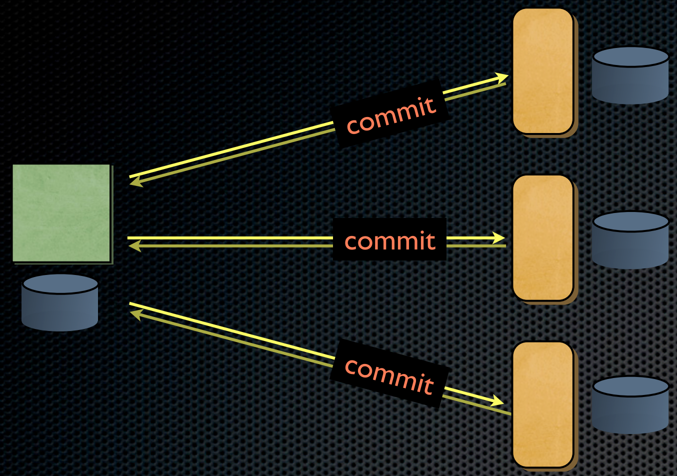


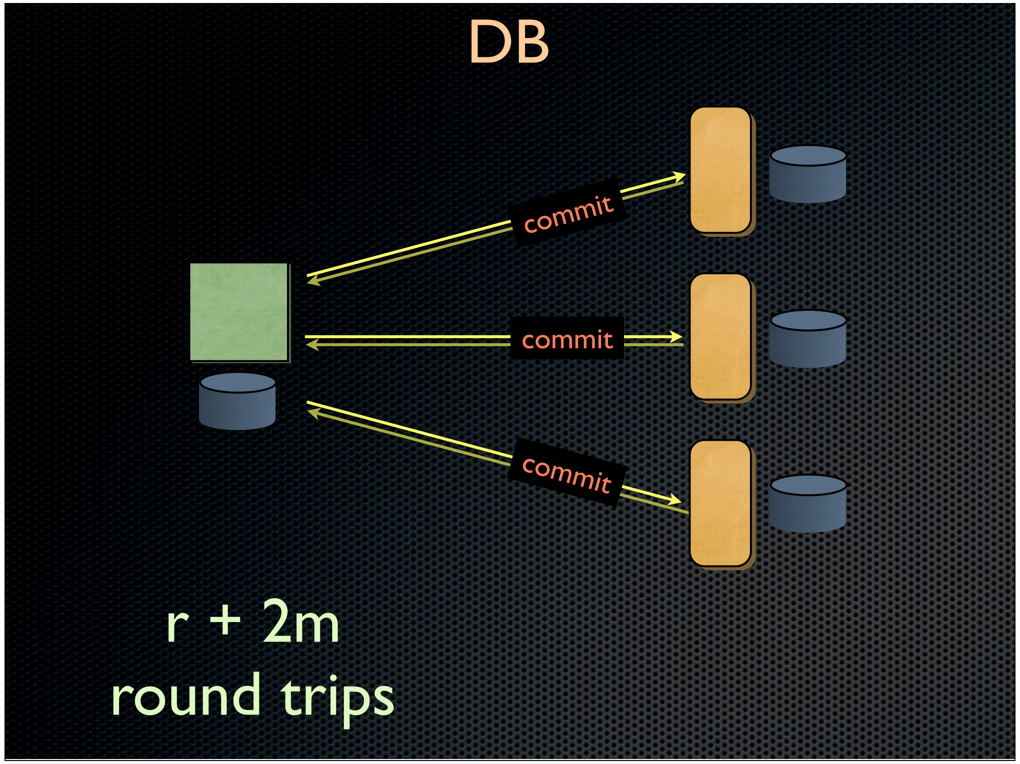
A brief recap about db operation and 2pc.
App begins a transaction, makes r requests (and touches m nodes in the process).
DB often locks the data.



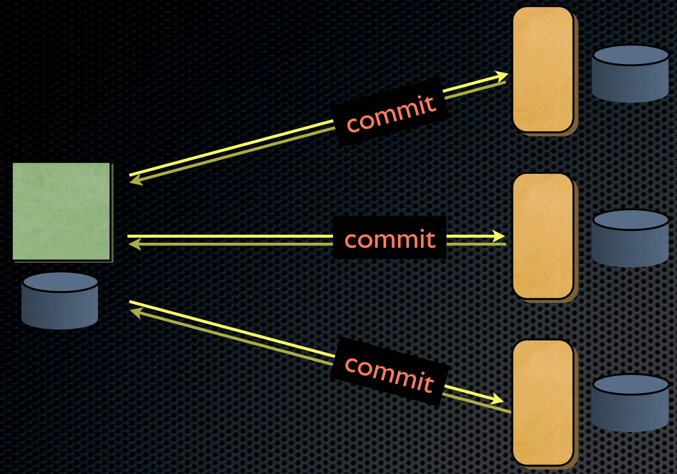
2 pc starts after data is over. The db nodes log their data and prepare decision, then the coordinator logs its commit decision.

DB



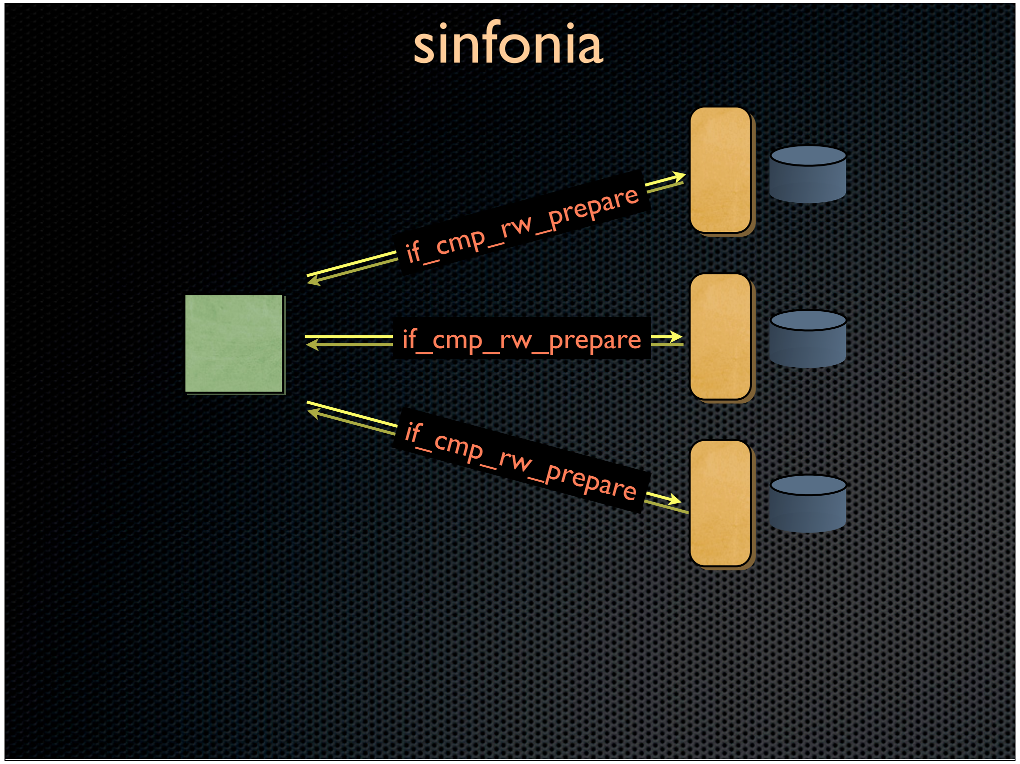

DB
commit
commit
commit
r + 2m
round trips

DB



r + 2m
round trips

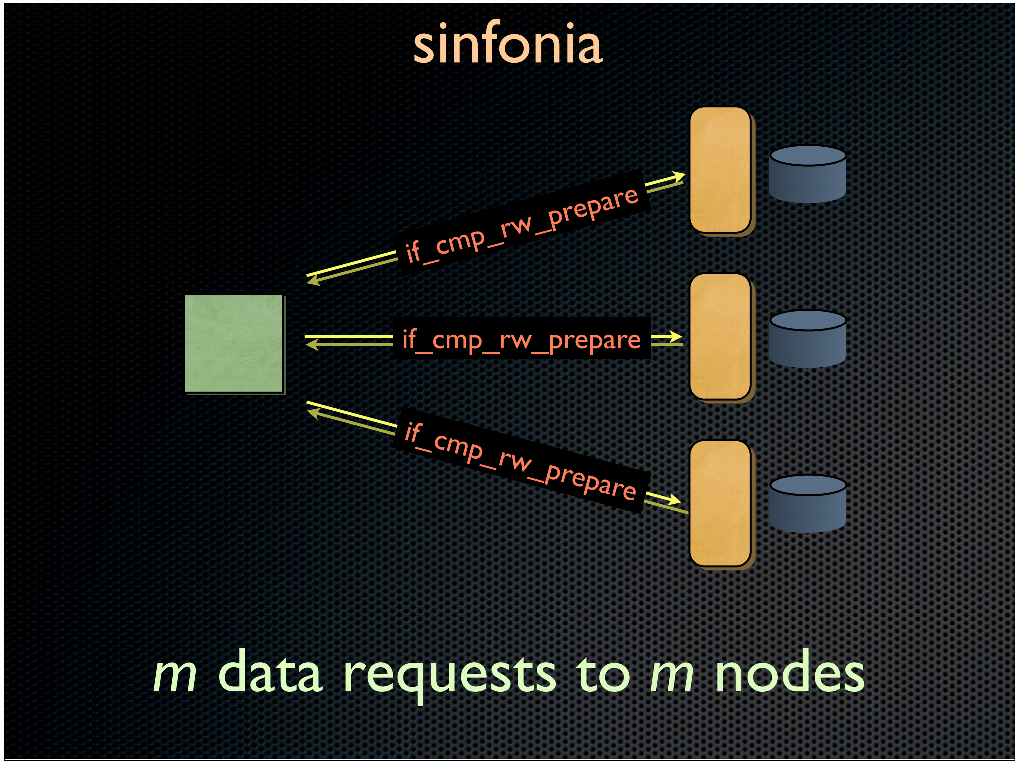
m + 1
disk writes



Sinfonia also exactly two phases, where the data transfer part is combined with prepare. Items are try-locked, then comparison is done, then the read and write is done. If the lock fails, the request fails.

It can do it because there is exactly one request allowed to read and write data. Clearly, this influences the way one writes application. The important takeaway is that such an API is useful in the real world. Multi-stage read-writes just have to be written as higher-order transactions (as with multi-word CAS)

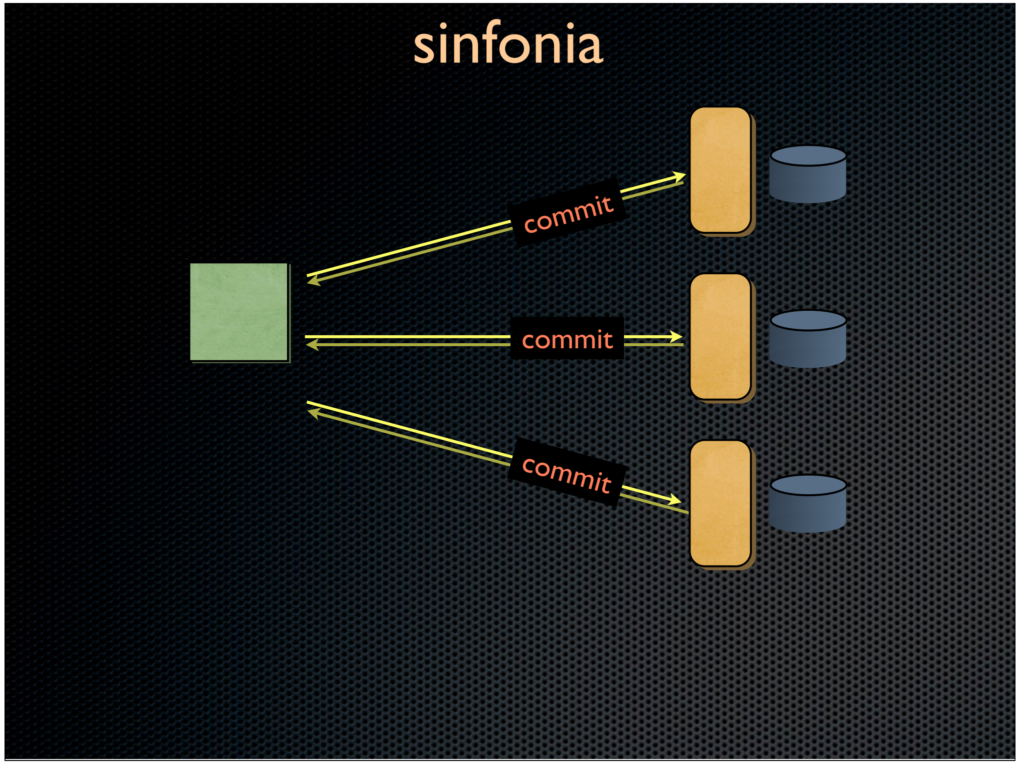
To be fair, one can do this with stored procedures, but current databases don't allow you to exec stored procedure and prepare in one network hop.



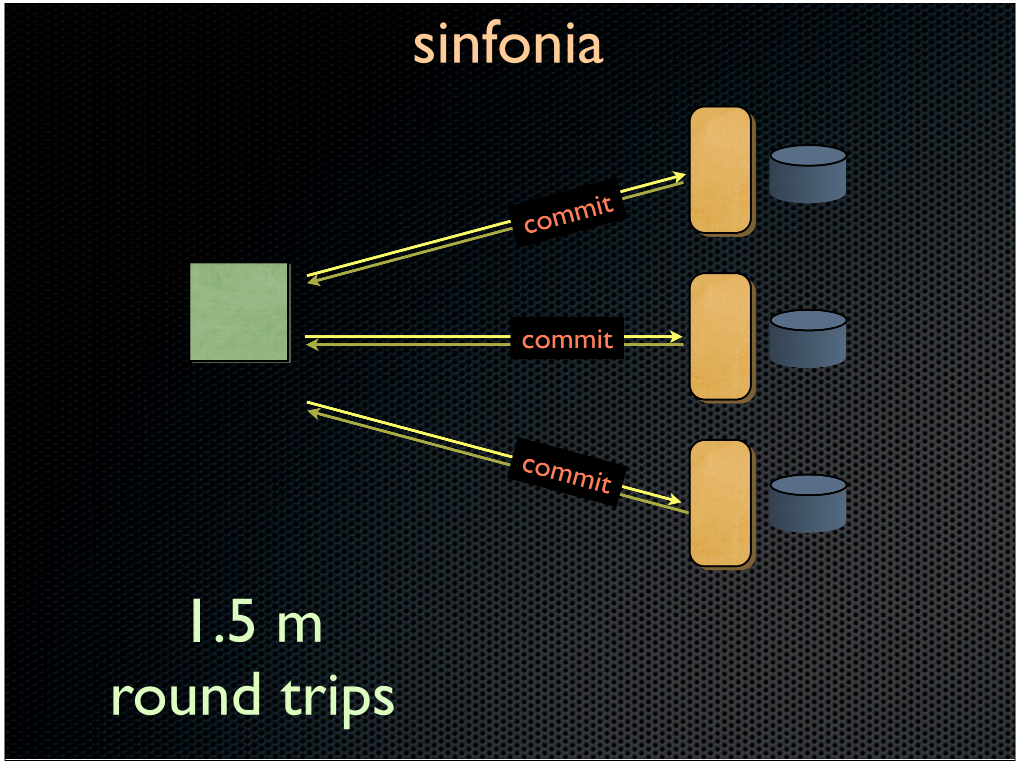
Sinfonia also exactly two phases, where the data transfer part is combined with prepare. Items are try-locked, then comparison is done, then the read and write is done. If the lock fails, the request fails.

It can do it because there is exactly one request allowed to read and write data. Clearly, this influences the way one writes application. The important takeaway is that such an API is useful in the real world. Multi-stage read-writes just have to be written as higher-order transactions (as with multi-word CAS)

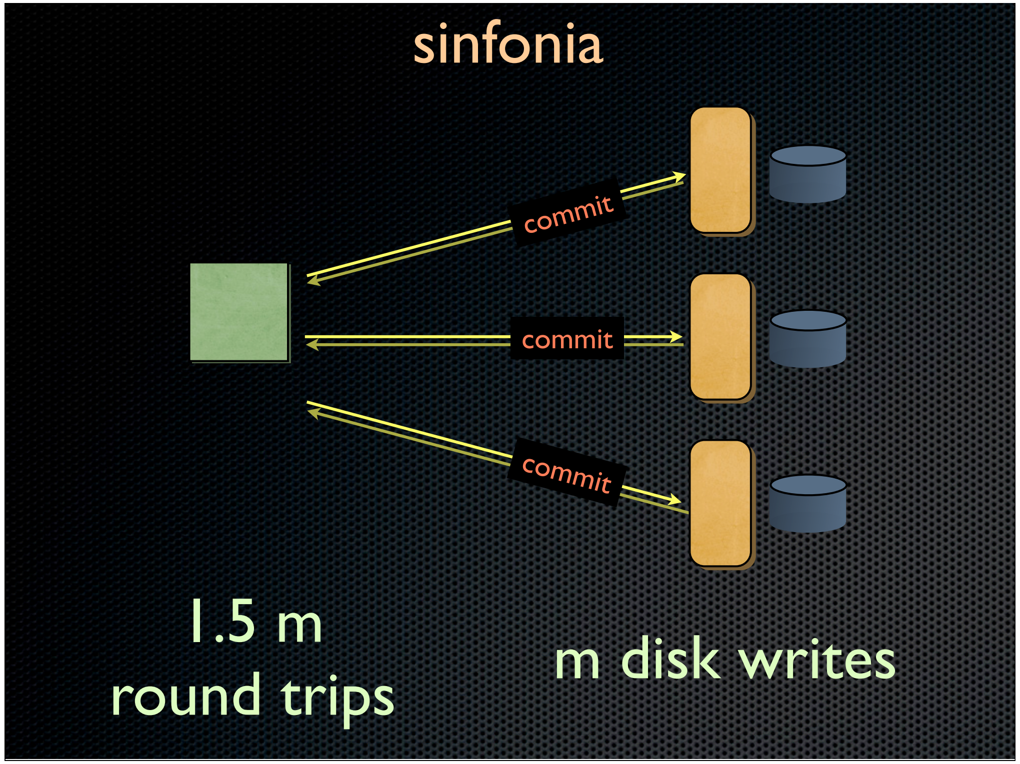
To be fair, one can do this with stored procedures, but current databases don't allow you to exec stored procedure and prepare in one network hop.



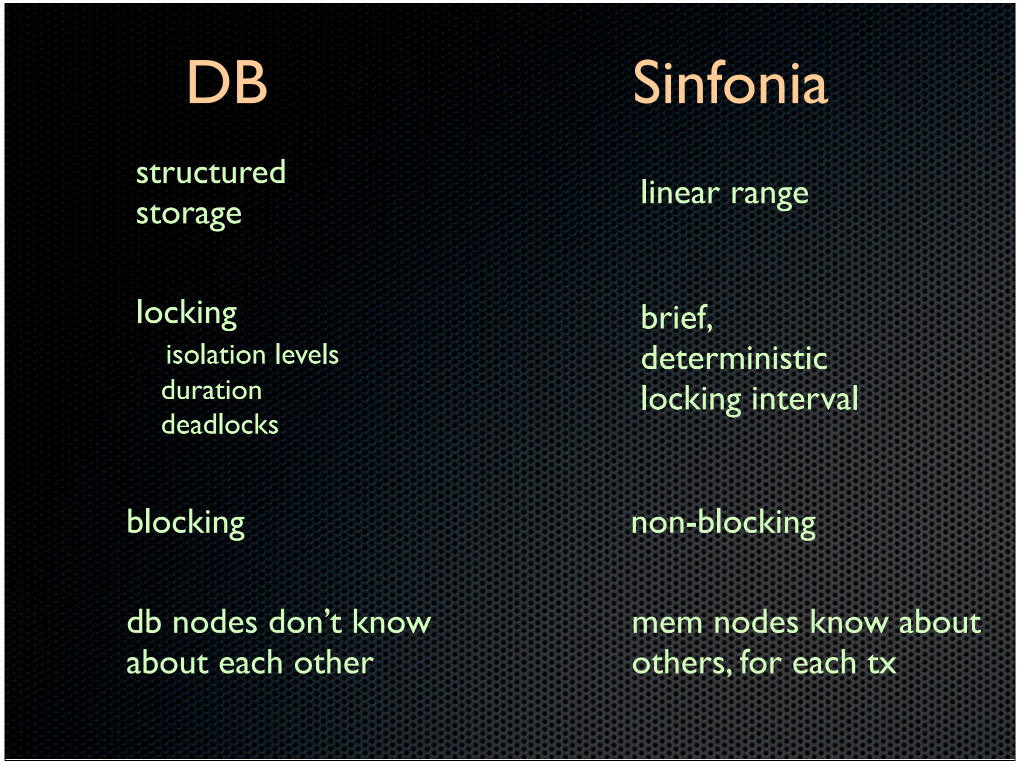
no coordinator disk write, plus commit write is lazy.



no coordinator disk write, plus commit write is lazy.



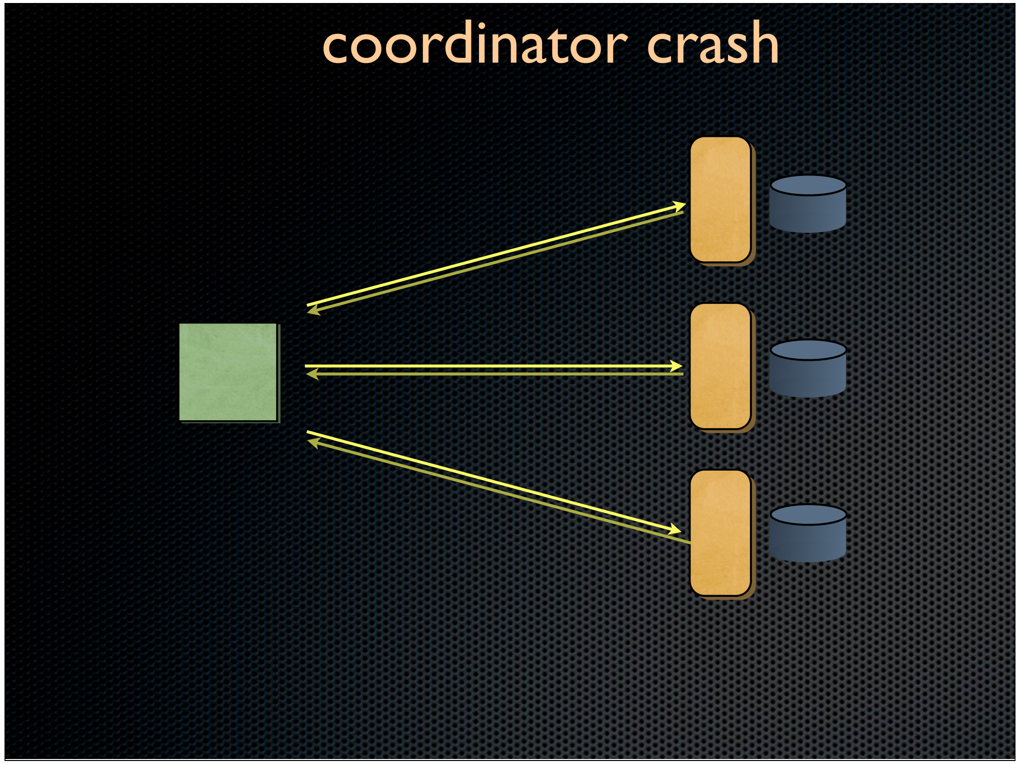
no coordinator disk write, plus commit write is lazy.



sinfonia: much lower level; app may have to worry about garbage collecting space

sinfonia: no blocking. If lock not acquired, does not prepare.

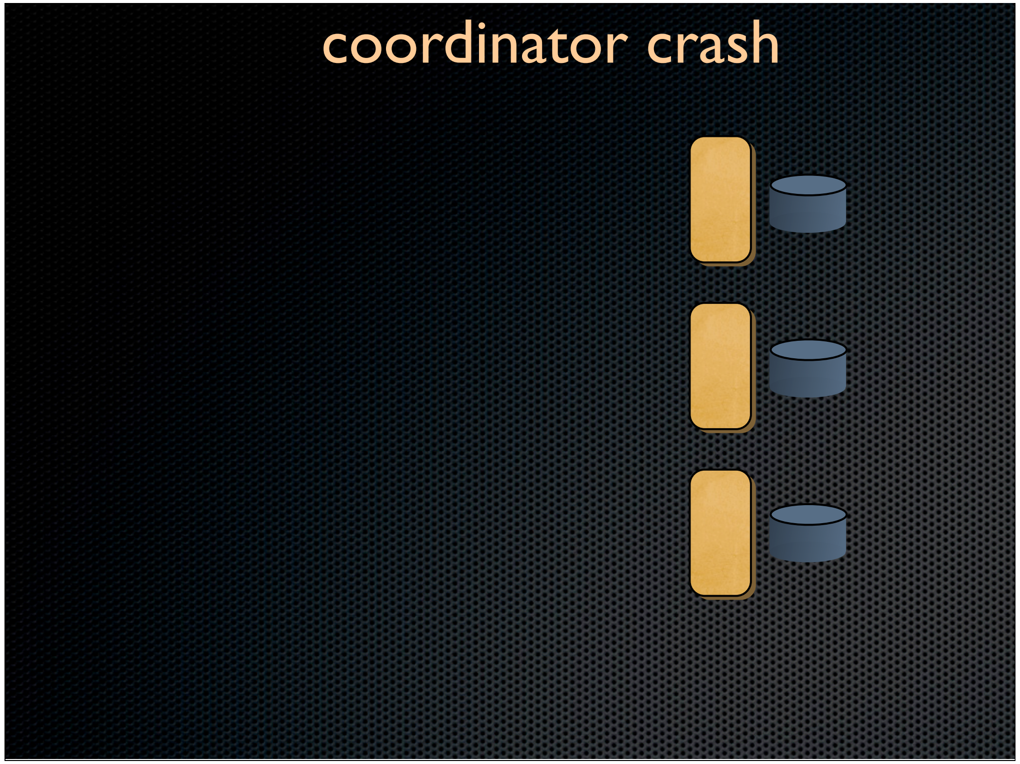
2pc: coordinator is a bottleneck for recovery because only it knows the participants.



Management infrastructure periodically polls mem nodes about in-doubt transactions and a recovery coordinator kicks in when a coordinator crashes. It tells each node, for each in-doubt tx, to abort the tx unless it voted commit. If for a particular tx, all participating nodes say they voted to commit, then the rec. coordinator drives the tx to commit.

This is correct because this scheme can run concurrently with a coordinator which may have come back (maybe it got stuck in a GC or network hiccup). It is correct because no-one changes their vote.

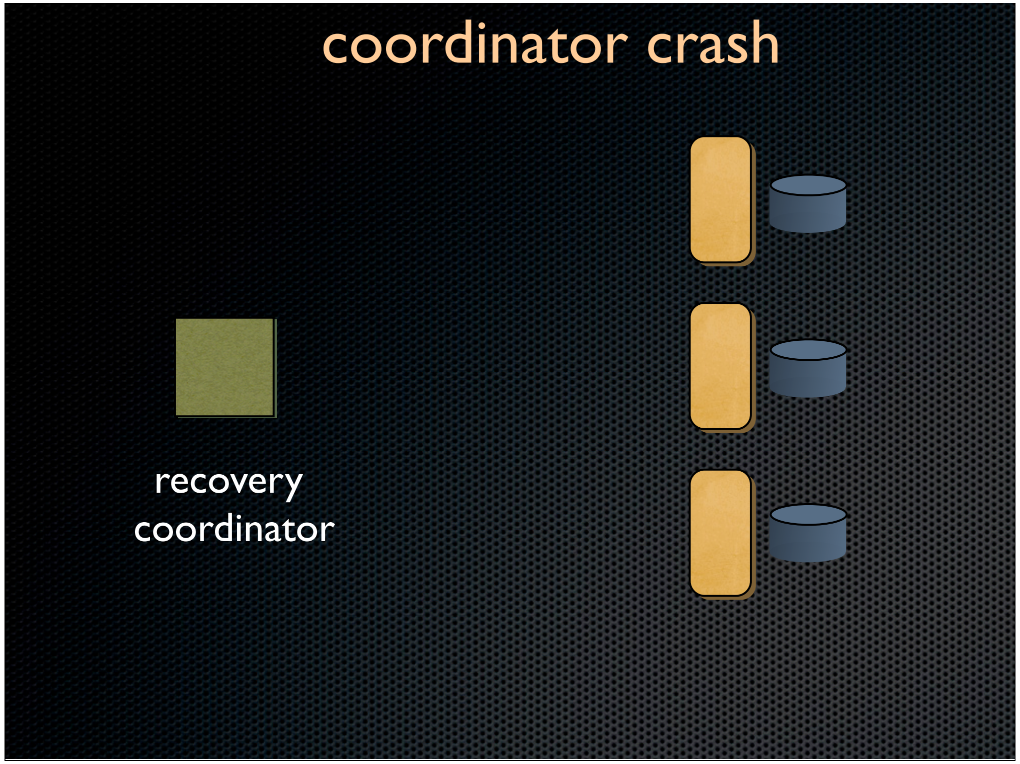
To me, it is not clear from the paper how the management infrastructure knows which nodes the crashed coordinator was responsible for (unless there is a hint in the transaction id). Otherwise, it is reckless to start aborting all transactions currently in progress at all nodes.



Management infrastructure periodically polls mem nodes about in-doubt transactions and a recovery coordinator kicks in when a coordinator crashes. It tells each node, for each in-doubt tx, to abort the tx unless it voted commit. If for a particular tx, all participating nodes say they voted to commit, then the rec. coordinator drives the tx to commit.

This is correct because this scheme can run concurrently with a coordinator which may have come back (maybe it got stuck in a GC or network hiccup). It is correct because no-one changes their vote.

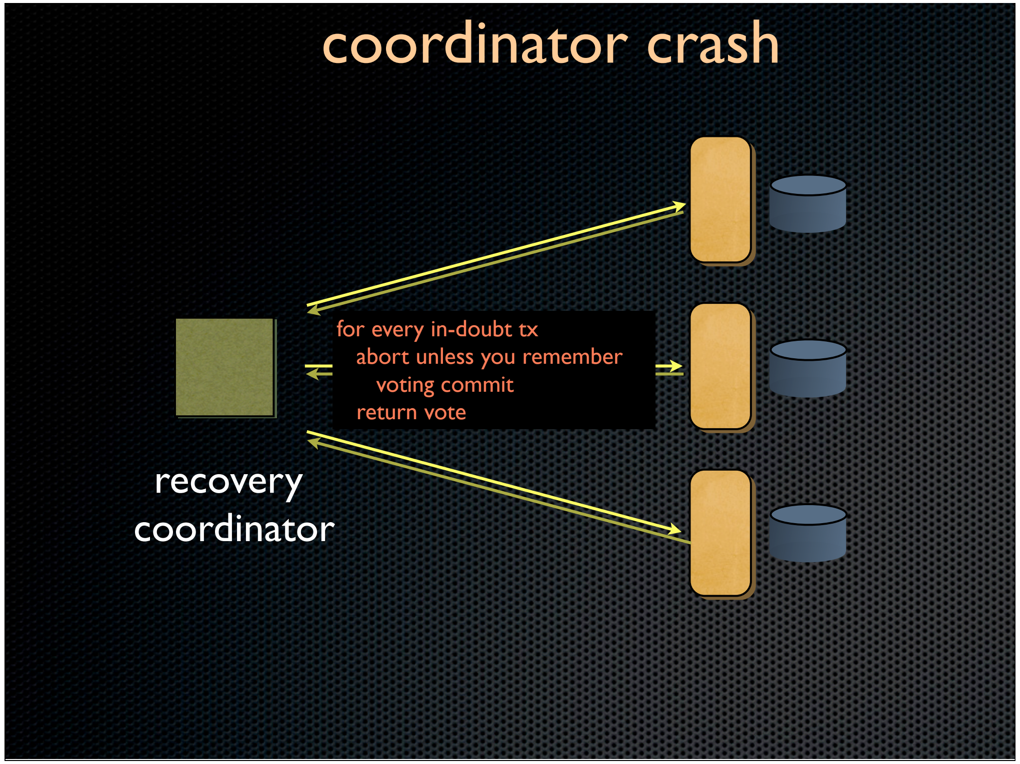
To me, it is not clear from the paper how the management infrastructure knows which nodes the crashed coordinator was responsible for (unless there is a hint in the transaction id). Otherwise, it is reckless to start aborting all transactions currently in progress at all nodes.



Management infrastructure periodically polls mem nodes about in-doubt transactions and a recovery coordinator kicks in when a coordinator crashes. It tells each node, for each in-doubt tx, to abort the tx unless it voted commit. If for a particular tx, all participating nodes say they voted to commit, then the rec. coordinator drives the tx to commit.

This is correct because this scheme can run concurrently with a coordinator which may have come back (maybe it got stuck in a GC or network hiccup). It is correct because no-one changes their vote.

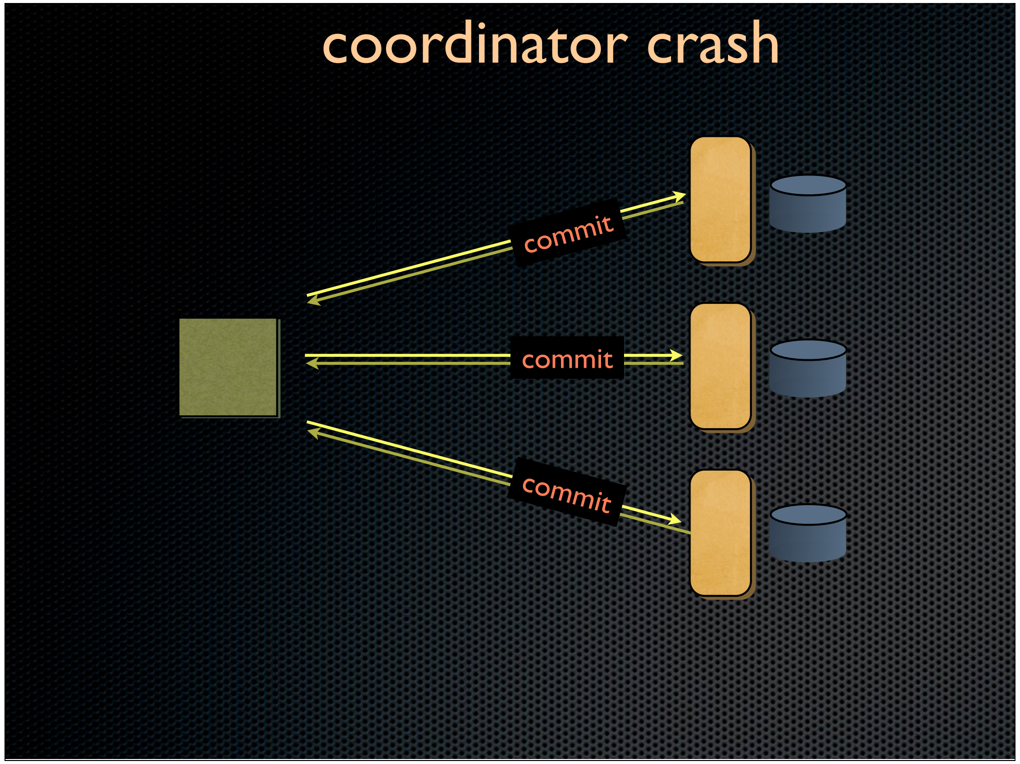
To me, it is not clear from the paper how the management infrastructure knows which nodes the crashed coordinator was responsible for (unless there is a hint in the transaction id). Otherwise, it is reckless to start aborting all transactions currently in progress at all nodes.



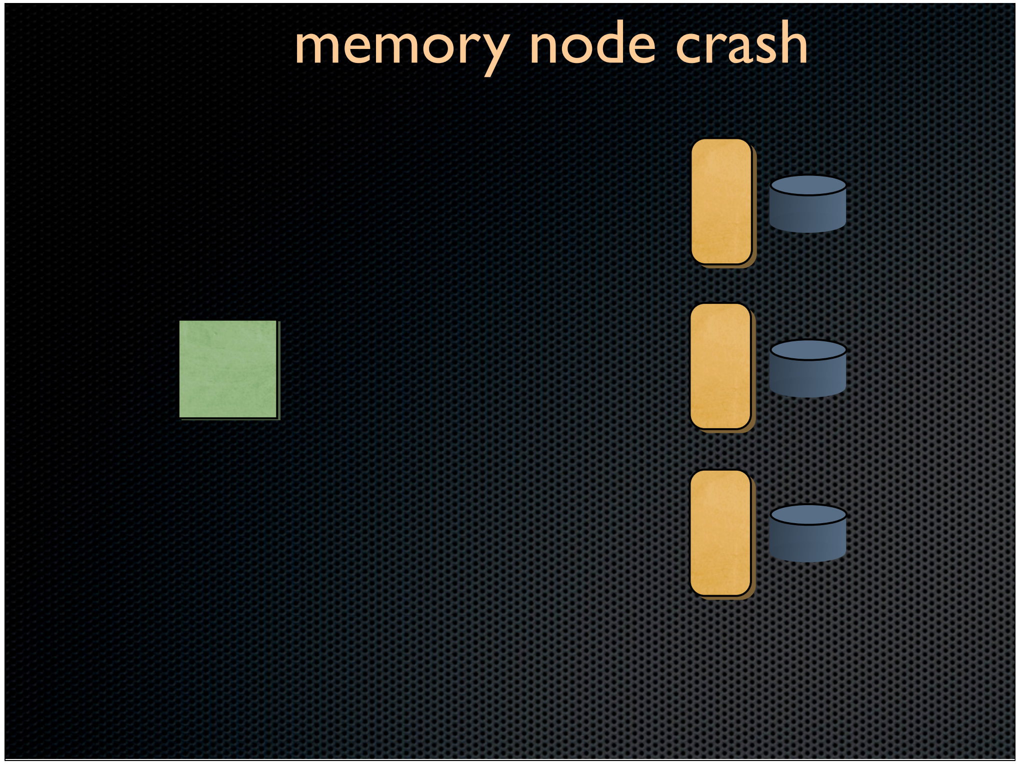
Management infrastructure periodically polls mem nodes about in-doubt transactions and a recovery coordinator kicks in when a coordinator crashes. It tells each node, for each in-doubt tx, to abort the tx unless it voted commit. If for a particular tx, all participating nodes say they voted to commit, then the rec. coordinator drives the tx to commit.

This is correct because this scheme can run concurrently with a coordinator which may have come back (maybe it got stuck in a GC or network hiccup). It is correct because no-one changes their vote.

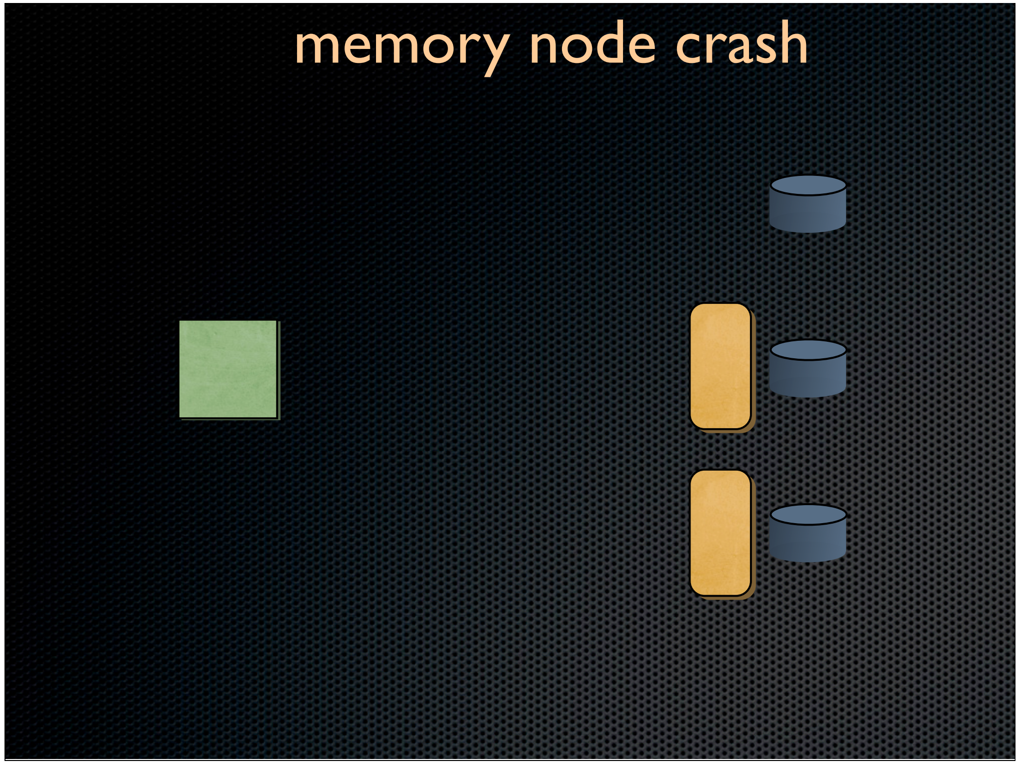
To me, it is not clear from the paper how the management infrastructure knows which nodes the crashed coordinator was responsible for (unless there is a hint in the transaction id). Otherwise, it is reckless to start aborting all transactions currently in progress at all nodes.



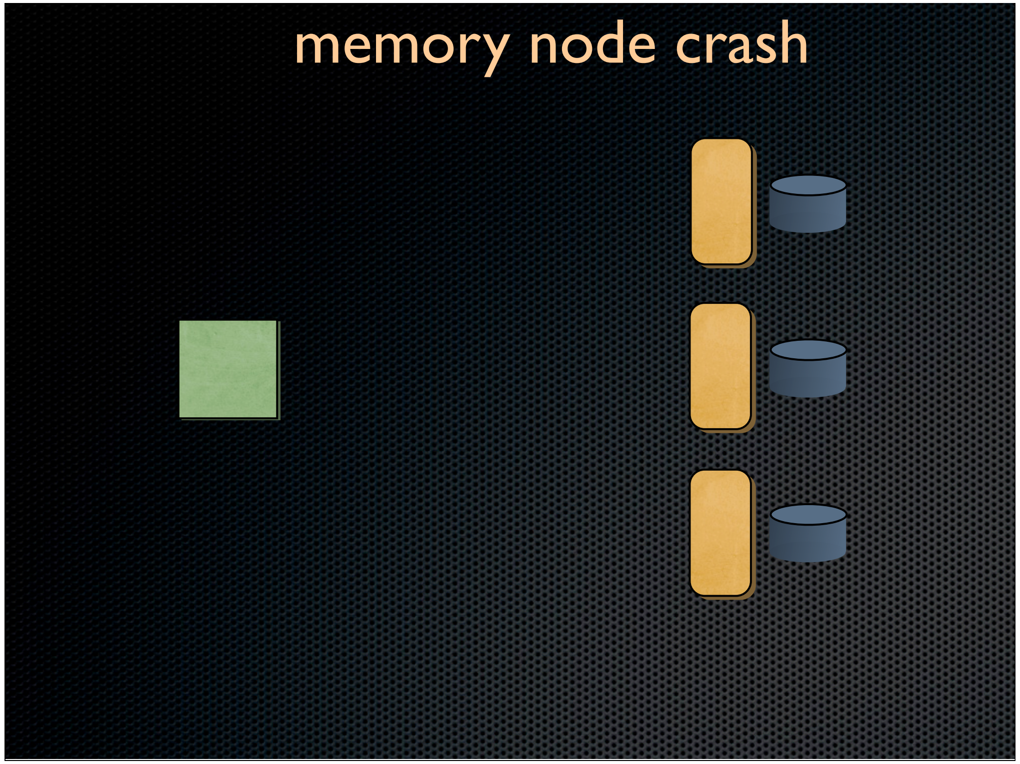
if all of them voted to commit, a commit is sent to all, else abort.



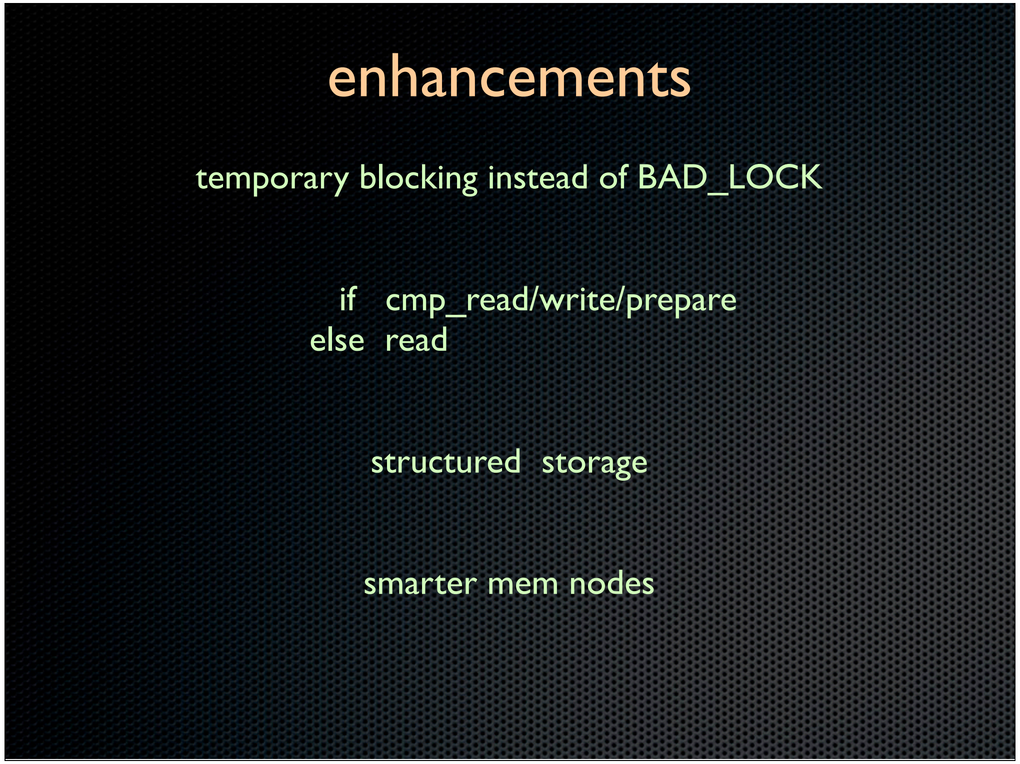
No recovery coordinator is used when a mem node crashes.
Nodes know the other nodes that were involved in the various transactions, so they ask each other while recovering.
I'm not a big fan of this architecture; I'd have preferred a recovery coordinator in all cases; it is less complex and a recovery from a system crash (like a power failure) doesn't swamp the network.



No recovery coordinator is used when a mem node crashes.
Nodes know the other nodes that were involved in the various transactions, so they ask each other while recovering.
I’m not a big fan of this architecture; I’d have preferred a recovery coordinator in all cases; it is less complex and a recovery from a system crash (like a power failure) doesn’t swamp the network.



No recovery coordinator is used when a mem node crashes.
Nodes know the other nodes that were involved in the various transactions, so they ask each other while recovering.
I'm not a big fan of this architecture; I'd have preferred a recovery coordinator in all cases; it is less complex and a recovery from a system crash (like a power failure) doesn't swamp the network.



Temporary blocking: I’d like a separate option that says block on locking for a limited amount of time instead of returning. It does introduce the possibility of limited-duration deadlocks, but may improve throughput.

Structured storage: different addressing options, not just offset, count.
Ranges are prone to “off-by-one” errors that could result in livelocks and corrupted data. Key/Value storage keeps one key’s space logically separate from another.

App will also have to worry about portability. Fig. 7 in paper writes &newAttributes. This is tied to the current structure of attributes and to the machine that used it.

smarter: Compare could be any predicate (field2 > field 3). Actions could be increment, arithmetic, insertions etc.

Unnecessary round-trips on contention.

related reading

Google: Chubby, BigTable, TaskMaster
YouTube architecture

Yahoo: , PNuts

Microsoft: Partitioning and Recovery Service

Apache/Yahoo Hadoop Project: Zookeeper