Distributed State: Transactions and Consistency

Arvind Krishnamurthy

Preliminaries

- Distribution typically addresses two needs:
  - Split the work across multiple nodes
  - Provide more reliability by replication
  - Focus of 2PC and 3PC is the first reason: splitting the work across multiple nodes

Failures

- What are the different classes/types of failures in a distributed system?
- What guarantees should we aim to provide in building fault-tolerant distributed systems?

Transactions

- Mechanism for coping with crashes and concurrency
- Example: new account creation

```
begin_transaction()
if "alice" not in password table:
    add alice to password table
    add alice to profile table
commit_transaction()
```

- transactions must be: (ACID property)
  - atomic: all writes occur, or none, even if failures
  - serializable: result is as if transactions executed one by one
  - durable: committed writes survive crash and restart
- We are interested in ***distributed*** transactions

Distributed Commit

- A bunch of computers are cooperating on some task
- Each computer has a different role
- Want to ensure atomicity: all execute, or none execute
- Challenges: failures, performance

Example

- calendar system, each user has a calendar
  - one server holds calendars of users A-M, another server holds N-Z
- sched(u1, u2, t):

```
begin_transaction()
ok1 = reserve(u1, t)
ok2 = reserve(u2, t)
if ok1 and ok2:
  if commit_transaction(): print "yes"
else abort_transaction()
```

- We want atomicity: both reserve, or neither reserves.
  - What if 1st reserve() returns true, 2nd reserve() returns false (time not available, or u2 doesn't exist); 2nd reserve() doesn't return; client fails before 2nd reserve()?

Idea #1

- tentative changes, later commit or undo (abort)

```
reserve_handler(u, t):
  if u[t] is free:
    temp_u[t] = taken    // A TEMPORARY VERSION
    return true
  else:
    return false

commit_handler():
  copy temp_u[t] to real u[t]
abort_handler():
  discard temp_u[t]
```

Idea #2

- single entity decides whether to commit to ensure agreement
- let's call it the Transaction Coordinator (TC)
- client sends RPCs to A, B
- client's commit_transaction() sends "go" to TC
- TC/A/B execute distributed commit protocol...
- TC reports "commit" or "abort" to client

Model

- For each distributed transaction T:
  - one coordinator
  - a set of participants

- Coordinator knows participants; participants don't necessarily know each other
- Each process has access to a Distributed Transaction Log (DT Log) on stable storage

The setup

- Each process has an input value, vote: Yes, No
- Each process has to compute an output value decision: Commit, Abort

Atomic Commit Specification

AC-1: All processes that reach a decision reach the same one.

AC-2: A process cannot reverse its decision after it has reached one.

AC-3: The Commit decision can only be reached if all processes vote Yes.

AC-4: If there are no failures and all processes vote Yes, then the decision will be Commit.

AC-5: If all failures are repaired and there are no more failures, then all processes will eventually decide.

2-Phase Commit

Coordinator c

Participant p_i

I. sends VOTE-REQ to all participants

2-Phase Commit

Coordinator c

Participant p_i

I. sends VOTE-REQ to all participants

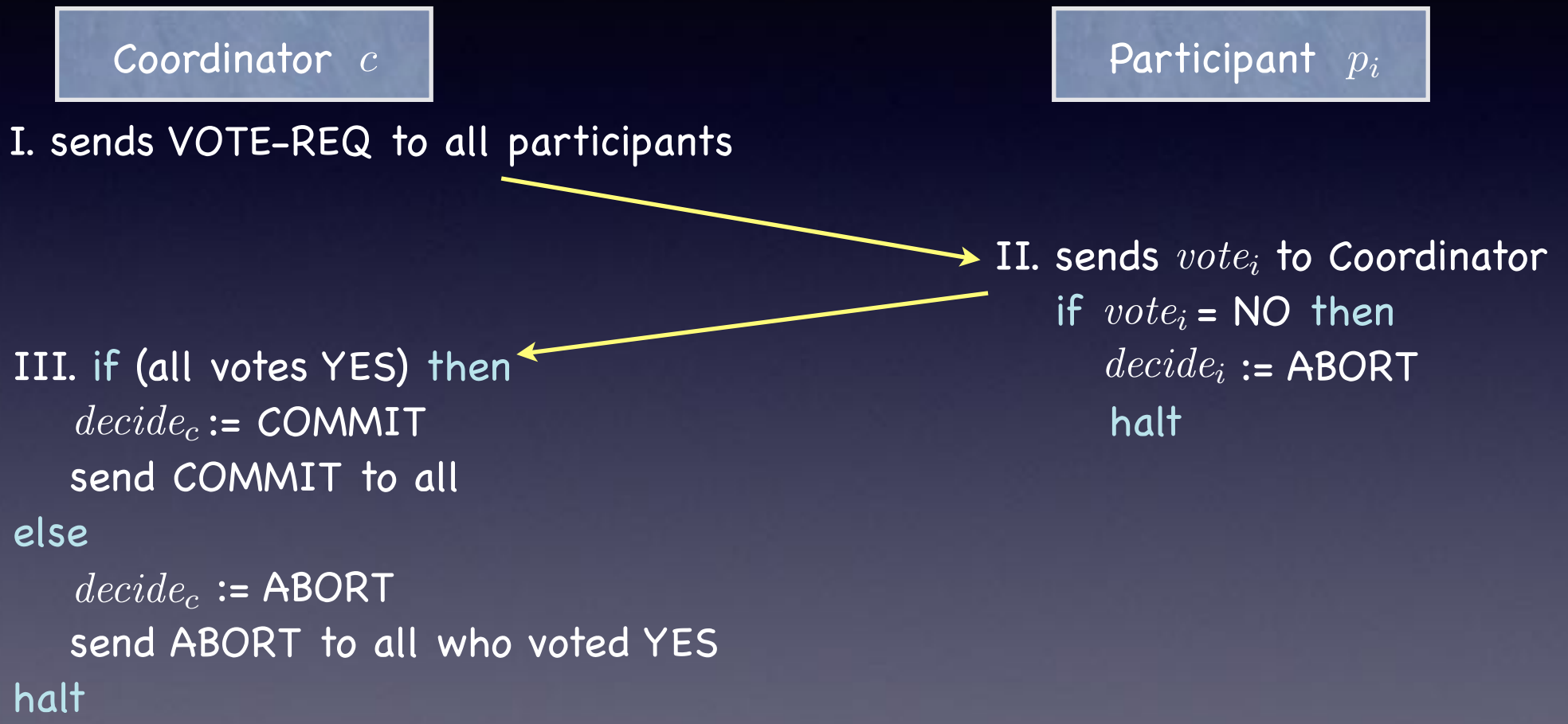
II. sends $vote_i$ to Coordinator

if $vote_i$ = NO then

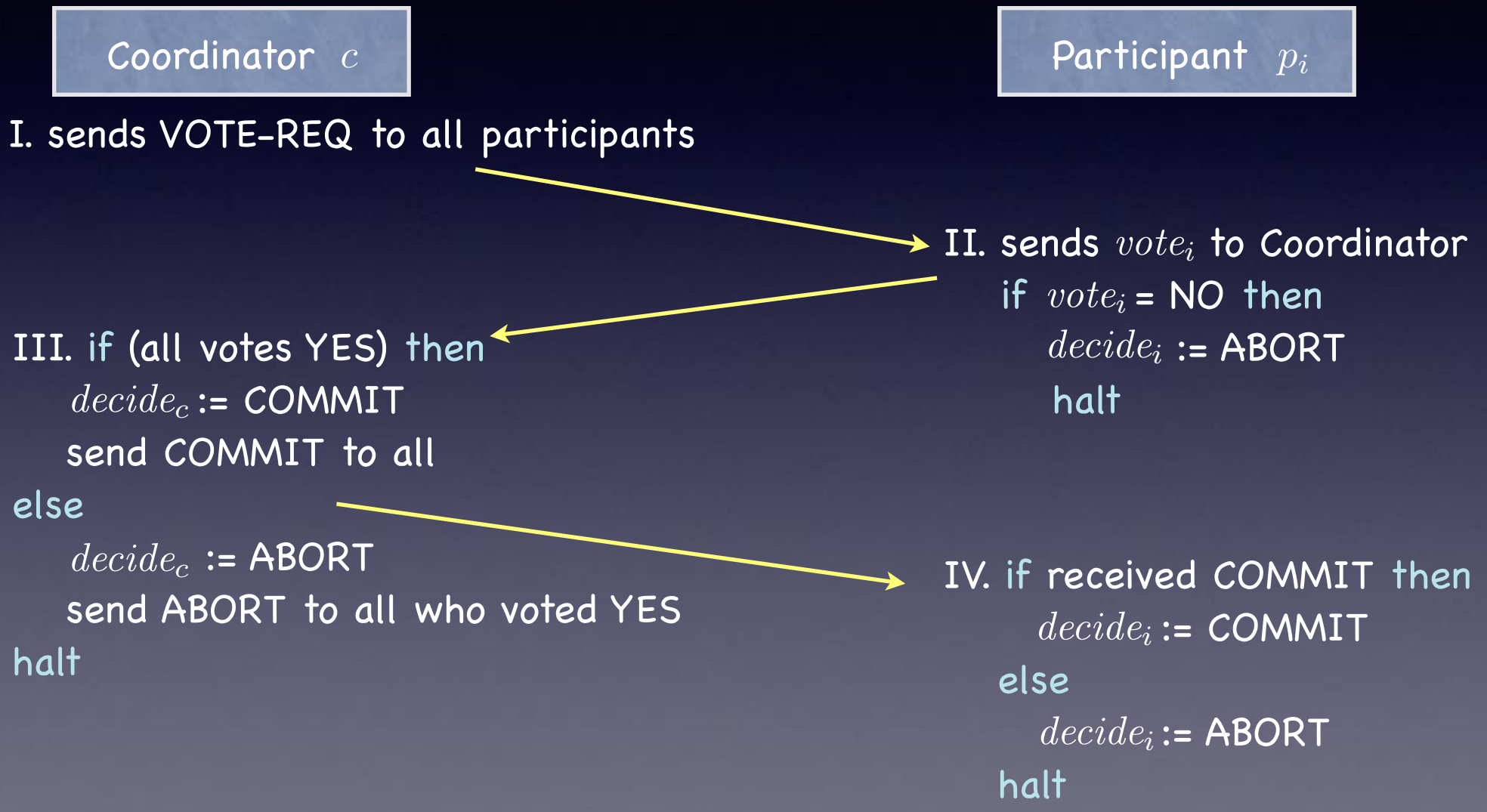
$decide_i$:= ABORT

halt

2-Phase Commit



2-Phase Commit



- How do we deal with different failures?

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Step 3 Coordinator is waiting for vote from participants

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

Termination protocols

I. Wait for coordinator to recover
   - It always works, since the coordinator is never uncertain
   - may block recovering process unnecessarily

II. Ask other participants

Logging actions

1. When coord sends VOTE-REQ, it writes START-2PC to its DT Log
2. When p_i is ready to vote YES,
   - writes YES to DT Log
   - sends YES to coord (writes also list of participants)
3. When p_i is ready to vote NO, it writes ABORT to DT Log
4. When c is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
5. When it is ready to decide ABORT, it writes ABORT to DT Log
6. After p_i receives decision value, it writes it to DT Log

p recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote Yes, writes Yes to DT Log before sending yes to coordinator (writes also list of participants)
   When participant is ready to vote No, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

- if DT Log contains START-2PC, then $p = c$:
  - if DT Log contains a decision value, then decide accordingly
  - else decide ABORT
- otherwise, p is a participant:
  - if DT Log contains a decision value, then decide accordingly
  - else if it does not contain a Yes vote, decide ABORT
  - else (Yes but no decision) run a termination protocol

- How to deal with concurrency?
  - consider transactions that transfer money from one account to another
  - how would you handle concurrency in the context of 2-PC?

Correctness: Serializability

- results should be as if transactions ran one at a time in some order
- Why is serializability good for programmers?
  - it allows application code to ignore concurrency
  - just write the transaction to take system from one legal state to another
  - internally, the transaction can temporarily violate invariants
    - but serializability guarantees other xactions won't notice

Two Phase Locking

- each database record has a lock
- the lock is stored at the server that stores the record
- transaction must wait for and acquire a record's lock before using it
  - thus update() handler implicitly acquires lock when it uses a data record
- transaction holds its locks until *after* commit or abort
  - When transactions conflict, locks delay & force serial execution
  - When they don't conflict, locks allow fast parallel execution

Locking with 2-PC

- Server must acquire locks as it executes client ops
  - client->server RPCs have two effects: acquire lock, use data
- If server says "yes" to TC's prepare:
  - Must remember locks and values across crash+restart!
  - So must write locks+values to disk log, before replying “yes”
  - If reboot, then read locks+values from disk
- If server has not said "yes" to a prepare:
  - If crash+restart, server can release locks and discard data
  - And then say "no" to TC's prepare message

- What are the strengths/weaknesses of 2PC?

Key Insight for 3-PC

- Cannot abort unless we know that no one has committed
- We need an algorithm that lets us infer the state of failed nodes
  - Introduce an additional state that helps us in our reasoning
  - But start with the assumption that there are no communication failures

3-Phase Commit

- Two approaches:

1. Focus only on site failures
   - Non-blocking, unless all sites fails
   - Timeout $\equiv$ site at the other end failed
   - Communication failures can produce inconsistencies
2. Tolerate both site and communication failures
   - partial failures can still cause blocking, but less often than in 2PC

Blocking and uncertainty

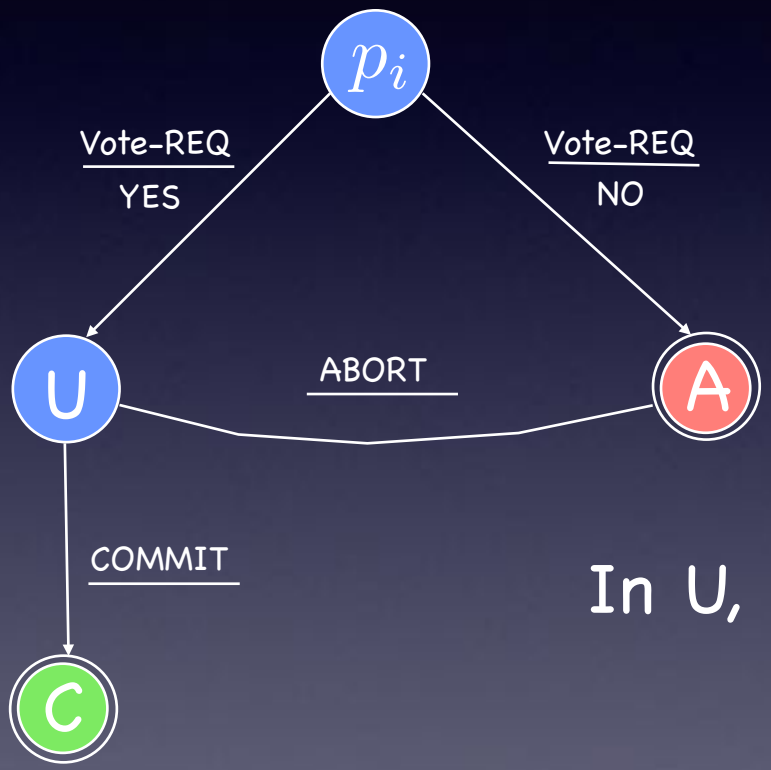
Why does uncertainty lead to blocking?

- An uncertain process does not know whether it can safely decide COMMIT or ABORT because some of the processes it cannot reach could have decided either

Non-blocking Property

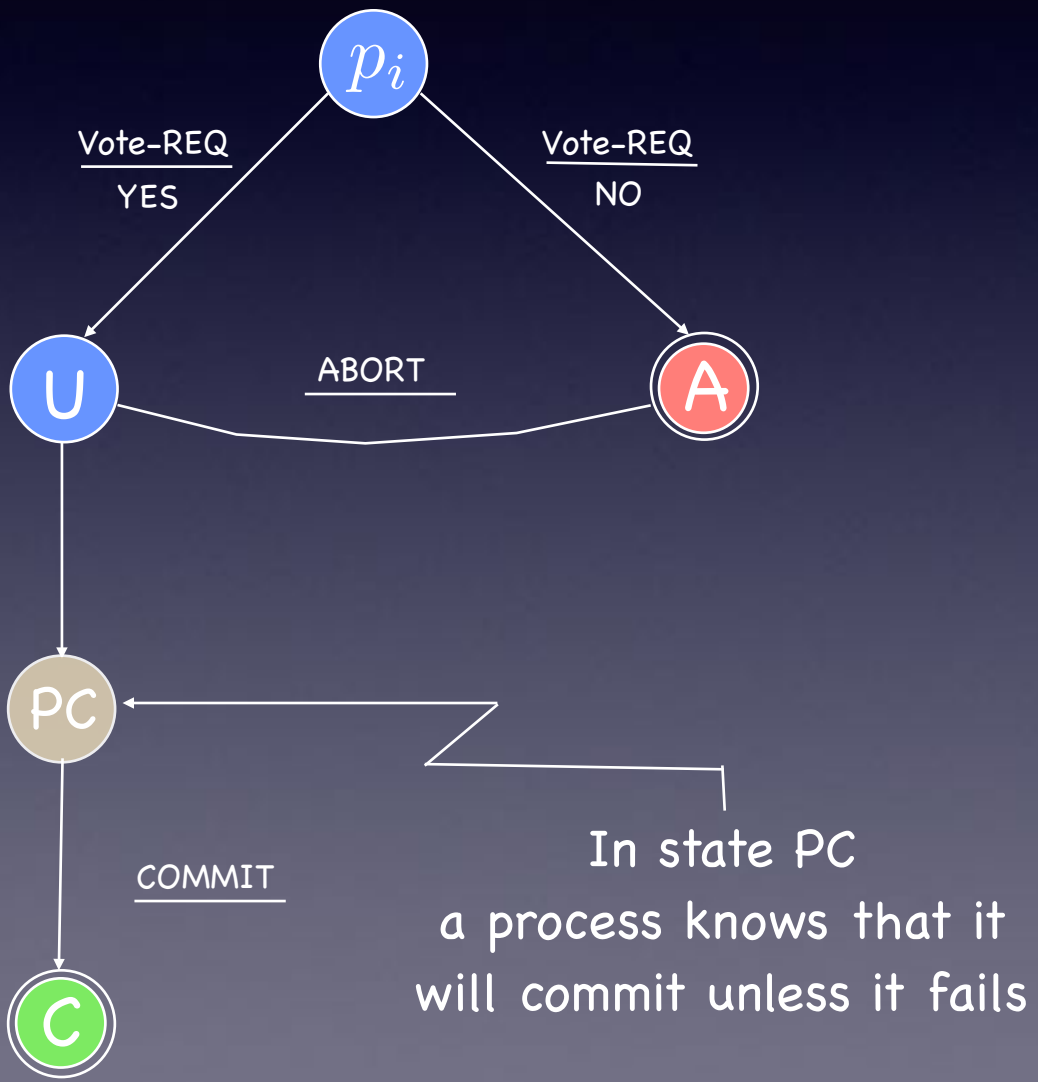
If any operational process is uncertain, then no process has decided COMMIT

2PC Revisited



In U, both A and C are reachable!

2PC Revisited



Coordinator Failure

- Elect new coordinator and have it collect the state of the system
- If any node is committed, then send commit messages to all other nodes
- If all nodes are uncertain, what should we do?

3PC: The Protocol

Dale Skeen (1982)

I. c sends VOTE-REQ to all participants.

II. When p_i receives a VOTE-REQ, it responds by sending a vote to c
if $vote_i$ = No, then $decide_i$:= ABORT and p_i halts.

III. c collects votes from all.
if all votes are Yes, then c sends PRECOMMIT to all
else $decide_c$:= ABORT; sends ABORT to all who voted Yes halts

IV. if p_i receives PRECOMMIT then it sends ACK to c

V. c collects ACKs from all.
When all ACKs have been received, $decide_c$:= COMMIT;
c sends COMMIT to all.

VI. When p_i receives COMMIT, p_i sets $decide_i$:= COMMIT and halts.

Termination protocol: Process states

At any time while running 3 PC, each participant can be in exactly one of these 4 states:

Aborted	Not voted, voted NO, received ABORT
Uncertain	Voted YES, not received PRECOMMIT
Committable	Received PRECOMMIT, not COMMIT
Committed	Received COMMIT

Not all states are compatible

	Aborted	Uncertain	Committable	Committed
Aborted	Y	Y	N	N
Uncertain	Y	Y	Y	N
Committable	N	Y	Y	Y
Committed	N	N	Y	Y

Failures

- Things to worry about:
  - timeouts: participant failure/coordinator failure
  - recovering participant
  - total failures

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator	Step 3 Coordinator is waiting for vote from participants
Step 4 p_i waits for PRECOMMIT	Step 5 Coordinator waits for ACKs
Step 6 p_i waits for COMMIT	

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator Exactly as in 2PC	Step 3 Coordinator is waiting for vote from participants Exactly as in 2PC
Step 4 p_i waits for PRECOMMIT Run some Termination protocol	Step 5 Coordinator waits for ACKs Coordinator sends COMMIT
Step 6 p_i waits for COMMIT Run some Termination protocol	

Termination protocol

- When p_i times out, it starts an election protocol to elect a new coordinator
- The new coordinator sends STATE-REQ to all processes that participated in the election
- The new coordinator collects the states and follows a termination rule

TR1. if some process decided ABORT, then?

TR2. if some process decided COMMIT, then?

TR3. if all processes that reported state are uncertain, then?

TR4. if some process is committable, but none committed, then?

Termination protocol

- When p_i times out, it starts an election protocol to elect a new coordinator
- The new coordinator sends STATE-REQ to all processes that participated in the election
- The new coordinator collects the states and follows a termination rule

TR1. if some process decided ABORT, then
  decide ABORT
  send ABORT to all
  halt

TR2. if some process decided COMMIT, then
  decide COMMIT
  send COMMIT to all
  halt

TR3. if all processes that reported state are uncertain, then
  decide ABORT
  send ABORT to all
  halt

TR4. if some process is committable, but none committed, then
  send PRECOMMIT to uncertain processes
  wait for ACKs
  send COMMIT to all
  halt

Discussion

- What are the strengths/weaknesses of 3PC?

Shared Virtual Memory

Context

- Parallel architectures & programming models
  - Bus-based shared memory multiprocessors
    - h/w support for coherent shared memory
    - can run both shared memory & message passing
    - scalable to 10's of nodes
  - Distributed memory machines/clusters of workstation
    - provides message passing interface
    - scalable up to 1000s of nodes
    - cheap! economies of scale, commodity shelf h/w

Distributed Shared Memory

- Radical idea: let us not have the hardware dictate what programming model we can use
- Provide a shared address space abstraction even on clusters

- Is this a good idea? What are the upsides/downsides of this approach?

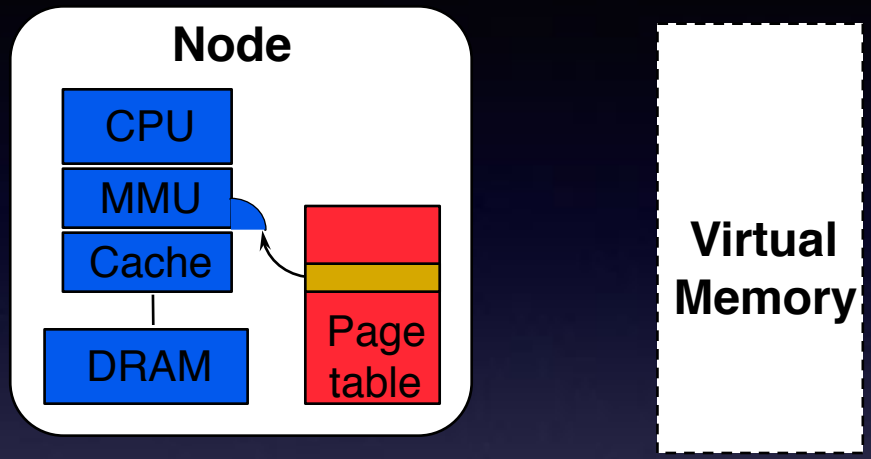
How do we provide this abstraction?

- Operating system support:
  - e.g., Ivy, Treadmarks, Munin
- Compiler support (Shasta)
  - minimize overhead through compiler analysis
  - object granularity as opposed to byte granularity
  - notions of immutable data, sharing patterns
- Limited hardware support (Wisconsin Wind Tunnel, DEC memory channel)

IVY Shared Virtual Memory

- Seminal system that sparked the entire field of DSM (distributed shared memory)
- Motivations:
  - sharing things on a network
  - “embassy” system to support a network file system between two different OSes
  - parallel ***scheme*** run time system on a cluster
- Focus: parallel computing and not distributed computing
  - less emphasis on request-reply, fault-tolerance, security

Traditional Virtual Memory



- Page Table entry:

Virt. page #	physical page #	valid

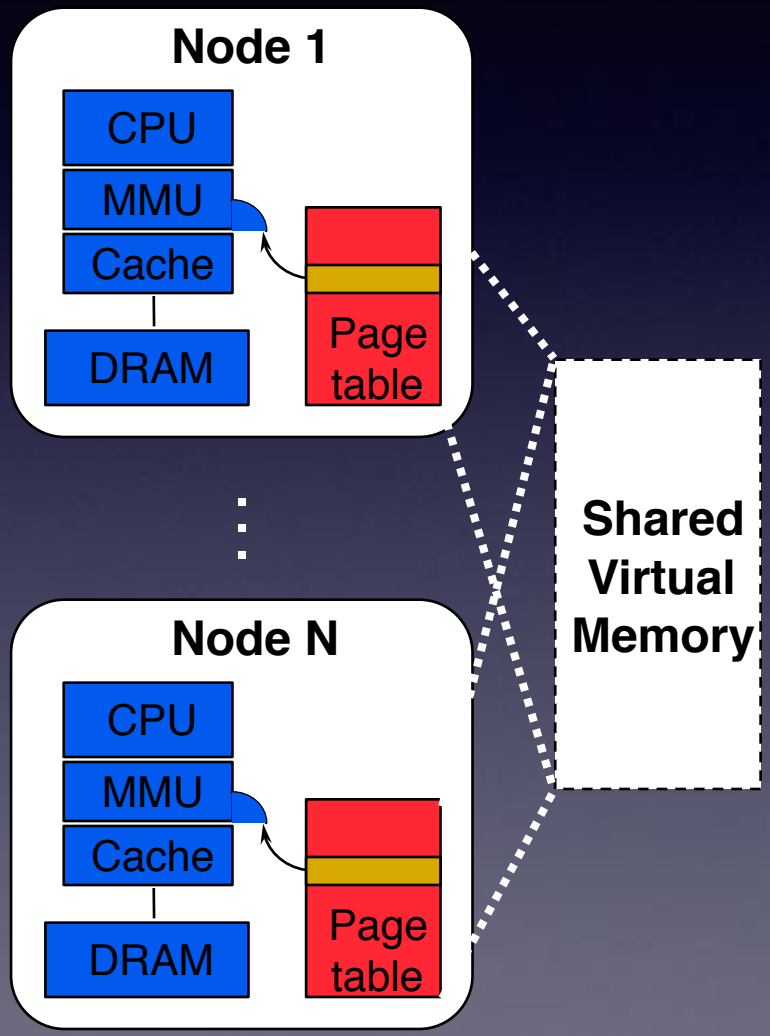
- If “valid”, translation exists
- If “not valid”, traps into the kernel, gets the page, re-executes trapped instruction
- Check is made for every access; TLB serves as a cache for the page table entries

Shared Virtual Memory

- Pool of "shared pages": if not local, page is not mapped
- Page table entry access bits

Virt. page #	physical page #	valid	access

- H/w detects read access to invalid page
  - read faults
- H/w detects writes to mapped memory with no write access
  - write faults
- OS maintains consistency at VM page level
  - copying data
  - setting access bits



Issues

- Programming model (as in coherence, consistency, etc.)
- Correctness of implementation
- Performance related issues

Programming Model

- Contract between programmer and h/w
- Shared memory abstraction typically means two related concepts:
  - Coherence
  - Consistency model (e.g., sequential consistency, linearizability)

- What is the difference between coherence and sequential consistency?

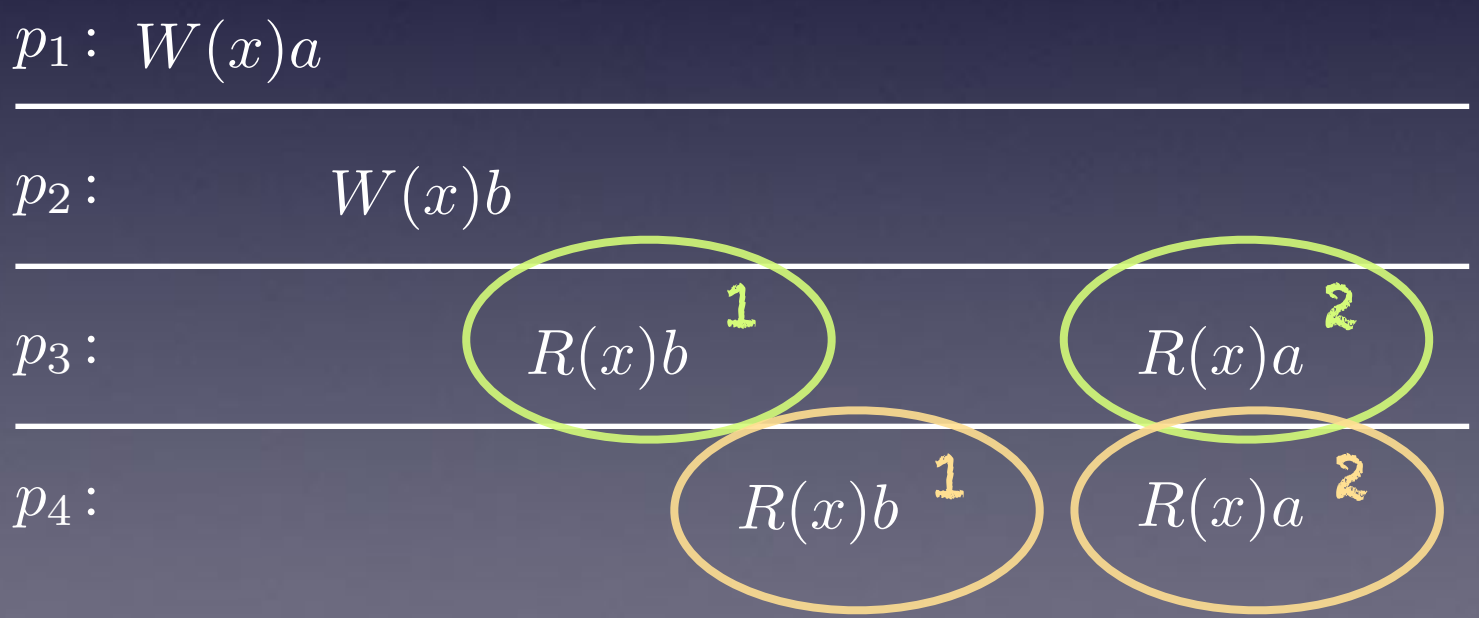
Coherence vs. Consistency

- Coherence: writes are propagated to other nodes; the writes to a particular memory location are seen in order

- Consistency: the writes to multiple distinct memory location or writes from multiple processors to the same location are seen in a well-defined order

Sequential Consistency

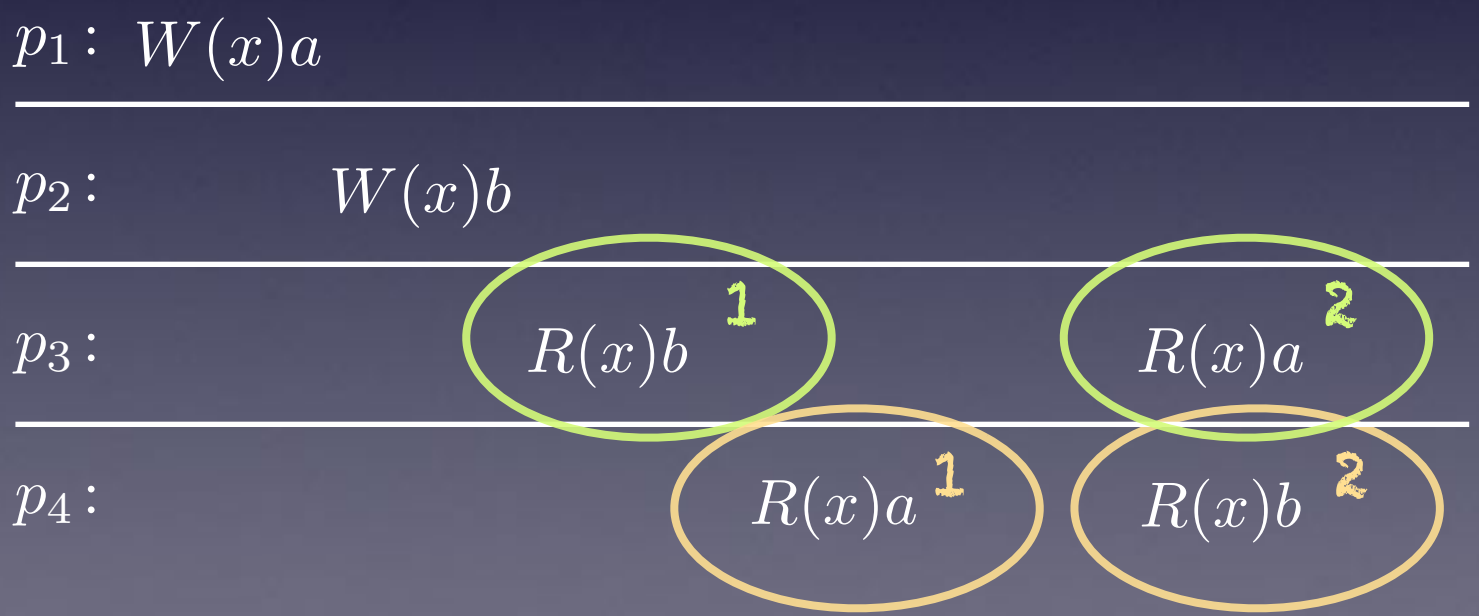
- "The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program" (Lamport, 1979)



Is this data store sequentially consistent?

Sequential Consistency

- "The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program" (Lamport, 1979)



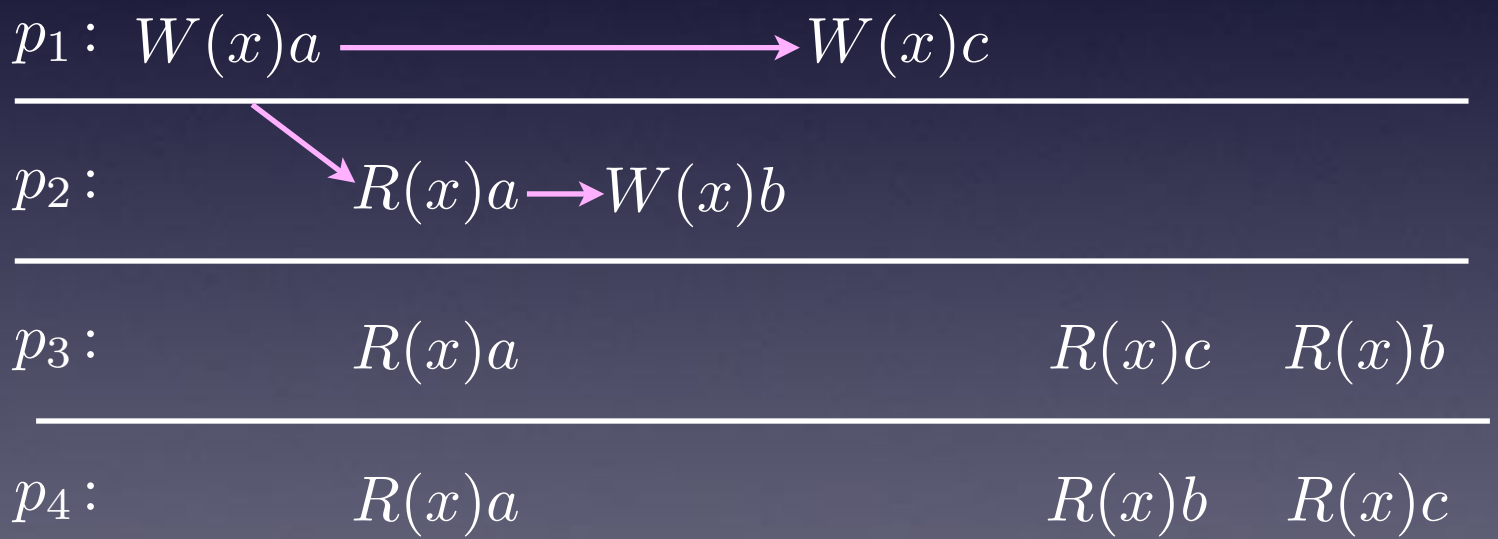
Is this data store sequentially consistent?

Other Consistency Models

- Can we have consistency models stronger than sequential consistency?
- How do we weaken sequential consistency?

Weakening Sequential Consistency: Causal Consistency

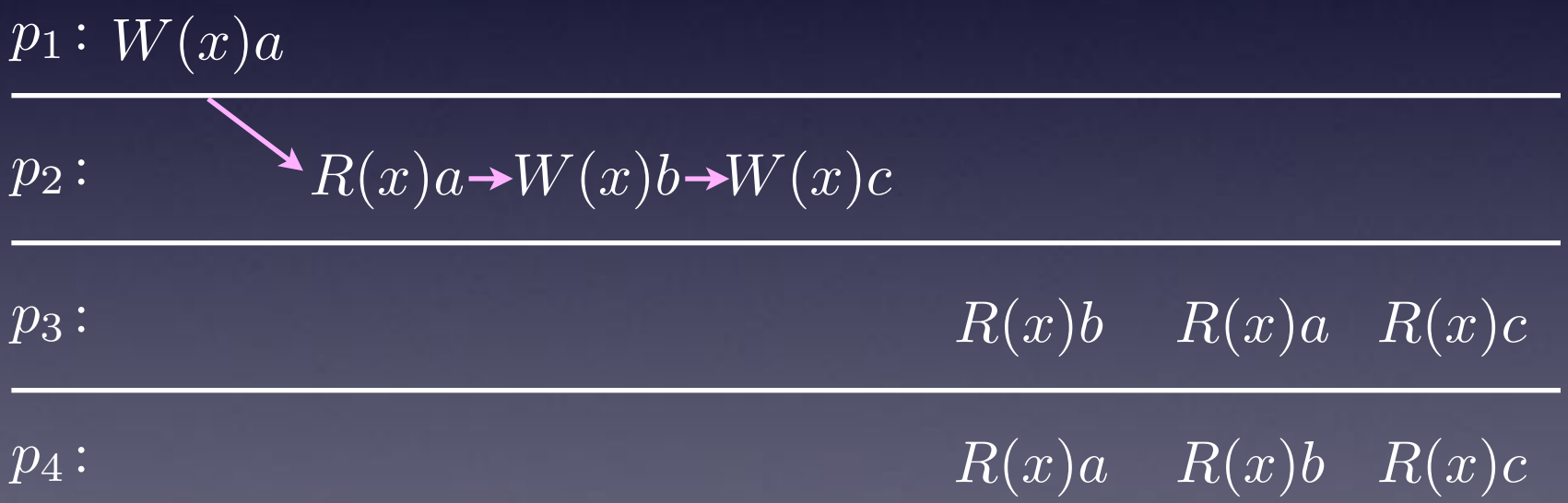
- Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines. (Hutto and Ahamad, 1990)



Is this data store sequentially consistent?
Causally consistent?

More Weakening: FIFO Consistency

- "Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes" (PRAM consistency, Lipton and Sandberg 1988)



Is this data store causally consistent?

Is this data store FIFO consistent?

Programming Complexity

Process p_1

$x := 1$

if $(y = 0)$ then

kill (p_2)

Process p_2

$y := 1$

if $(x = 0)$ then

kill (p_1)

Initially, $x = y = 0$

What are the possible outcomes?

- What do you make out of these consistency models?

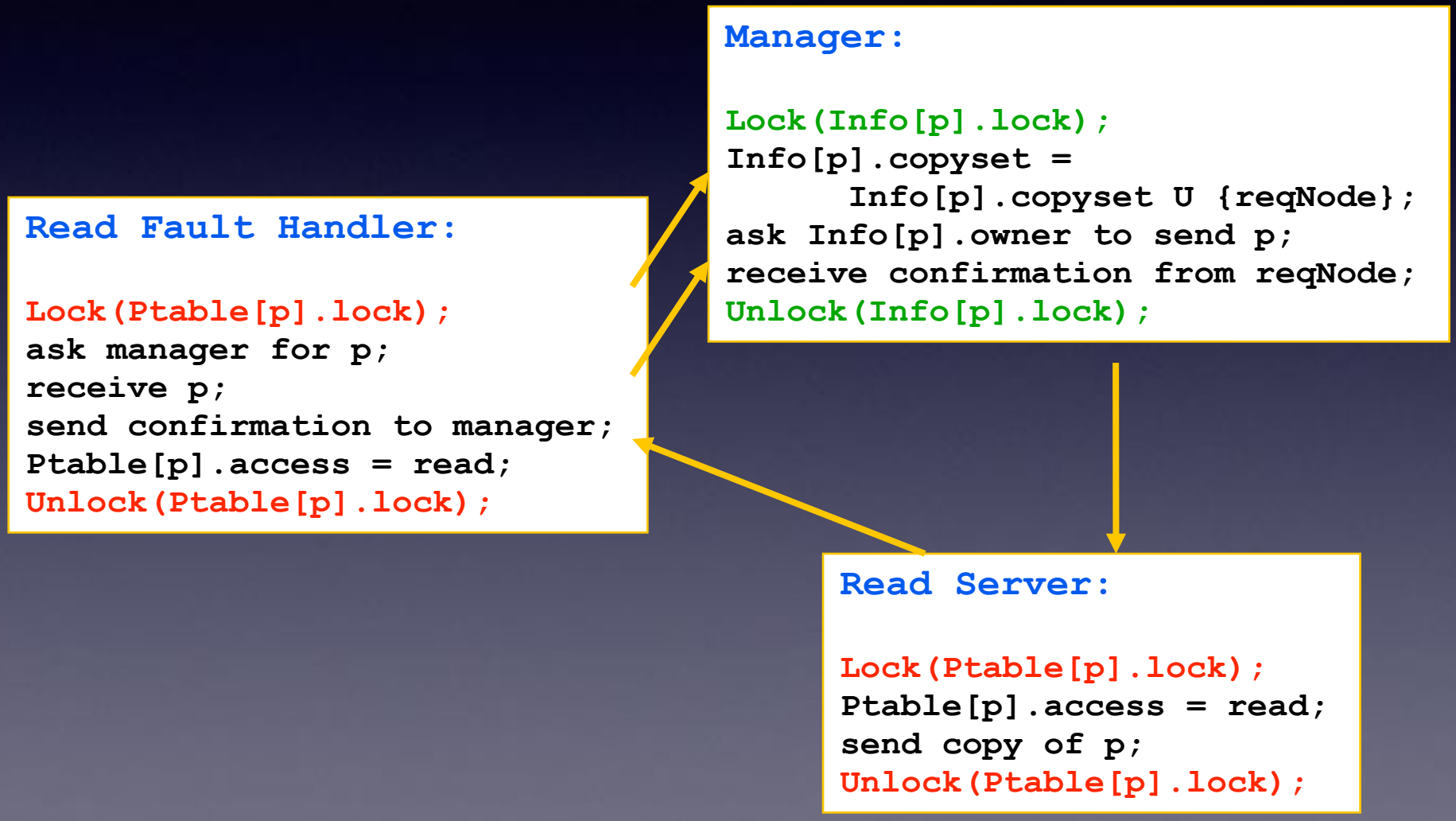
Ivy DSM

- Goal: provide sequentially consistent shared memory
- Baseline Implementation:
  - centralized manager
  - manager maintains the “owner” and the set of readers (“copyset”)

Read Faults

- Handler on client:
  - asks manager
  - manager forwards request to owner
  - owner sends the page
  - requester sends an ACK to manager

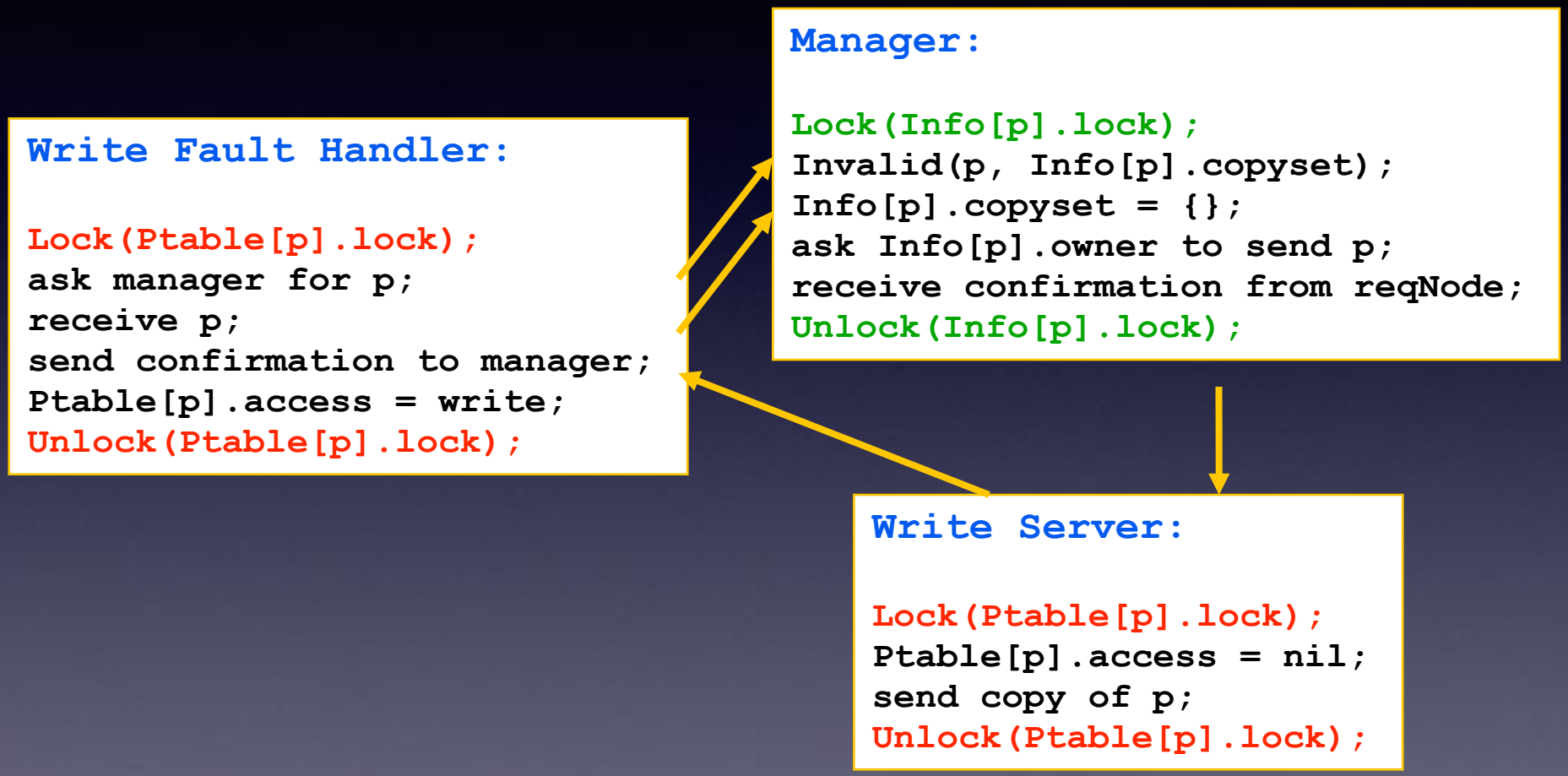
Pseudocode


Manager:
Lock(Info[p].lock);
Info[p].copyset =
Info[p].copyset U {reqNode};
ask Info[p].owner to send p;
receive confirmation from reqNode;
Unlock(Info[p].lock);
Read Fault Handler:
Lock(Ptable[p].lock);
ask manager for p;
receive p;
send confirmation to manager;
Ptable[p].access = read;
Unlock(Ptable[p].lock);
Read Server:
Lock(Ptable[p].lock);
Ptable[p].access = read;
send copy of p;
Unlock(Ptable[p].lock);

Write Faults

- Handling includes invalidations:
  - make request to manager
  - copies are invalidated
  - manager forwards request to owner
  - owner relinquishes page to requester
  - requester sends an ACK to the owner

Write Pseudocode



Scenarios

- Consider P1 and P2 caching a page with “read” perms
- What happens if both perform a “write” at the same time?

Question

- Can the confirmation messages be eliminated?

Scenarios

- Consider P1 is owner of page
- P2 performs a read
- P3 performs a write
- What if manager handles write before read is complete?

Improved Manager

- Owner serves as the manager for each page

```
Read Fault Handler:

Lock(Ptable[p].lock);
ask manager for p;
receive p;
Ptable[p].access = read;
Unlock(Ptable[p].lock);
```

```
Read Server:

Lock(Ptable[p].lock);
If I am owner {
    Ptable[p].access = read;
    Ptable[p].copyset =
        Ptable[p].copyset U {reqNode};
    send copy of p;
} else {
    forward request to probable owner;
}
Unlock(Ptable[p].lock);
```

Performance Questions

- In what situations will IVY perform well?
- How can we improve IVY's performance?