



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CONCURRENCY CONTROL

Lecture A

RPCs

WHY RPCs

- **RPC** = Remote Procedure Call
- Proposed by Birrell and Nelson in 1984
- Important abstraction for processes to call functions in other processes
- Allows code reuse
- Implemented and used in most distributed systems, including cloud computing systems
- Counterpart in Object-based settings is called RMI (Remote Method Invocation)

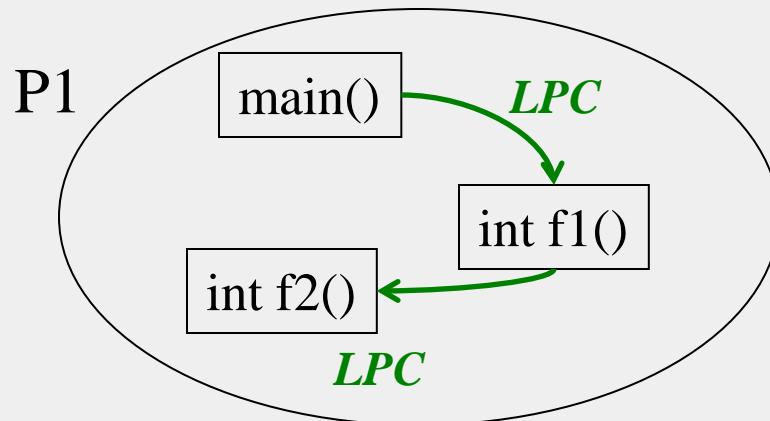
LOCAL PROCEDURE CALL (LPC)

- Call from one function to another function within the same process
 - Uses stack to pass arguments and return values
 - Accesses objects via pointers (e.g., C) or by reference (e.g., Java)
- LPC has *exactly-once* semantics
 - If process is alive, called function executed exactly once

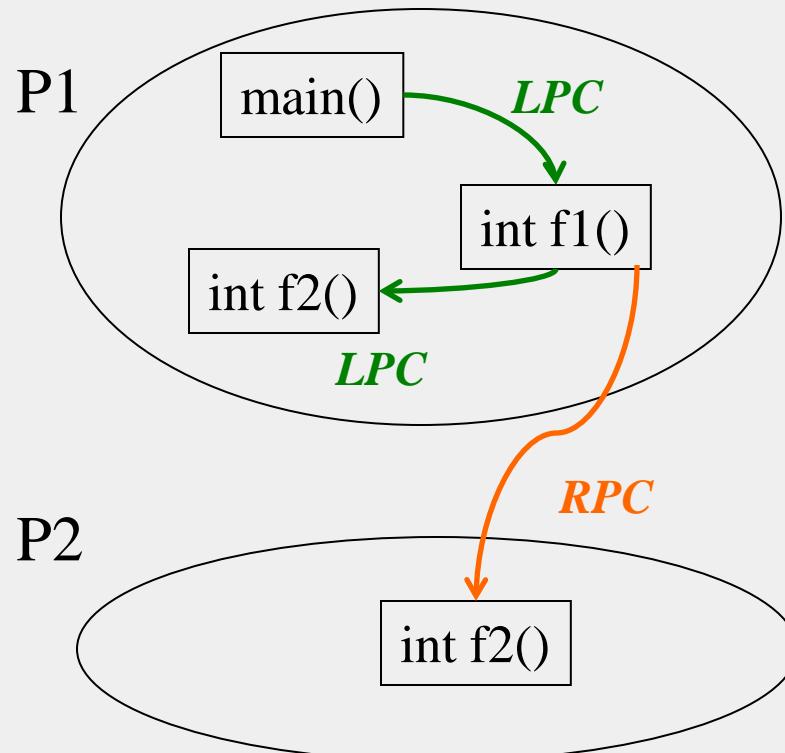
REMOTE PROCEDURE CALL

- Call from one function to another function, where caller and callee function reside in different processes
 - Function call crosses a process boundary
 - Accesses objects via global references
 - Can't use pointers across processes since a reference address in process P1 may point to a different object in another process P2
 - E.g., Object address = IP + port + object number
- Similarly, RMI (Remote Method Invocation) in Object-based settings

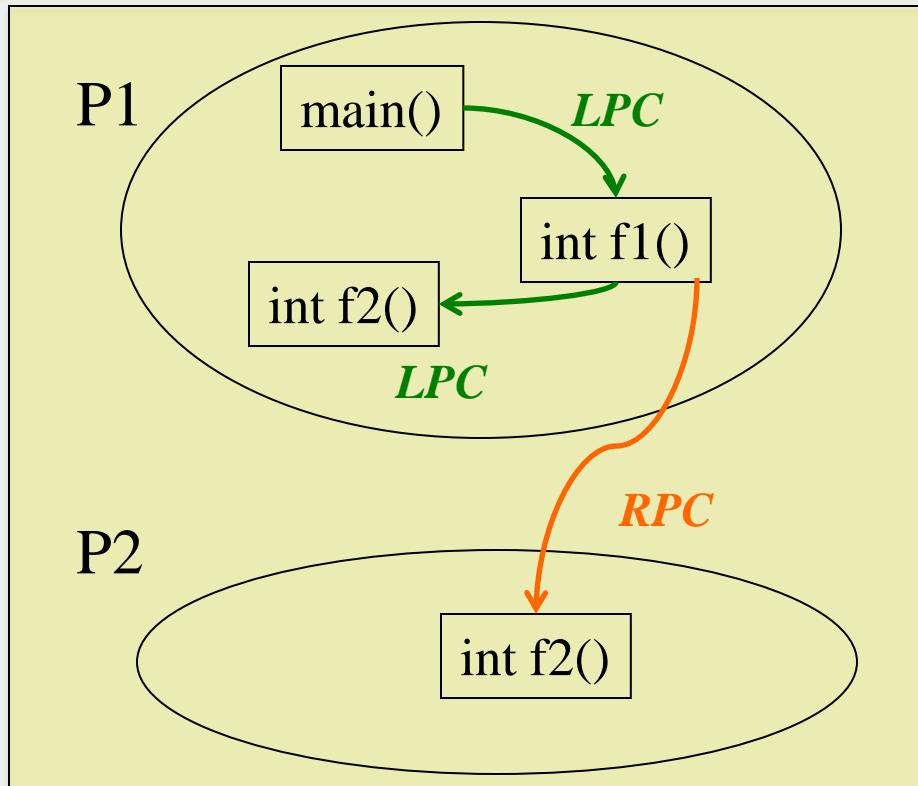
LPCs



RPCs

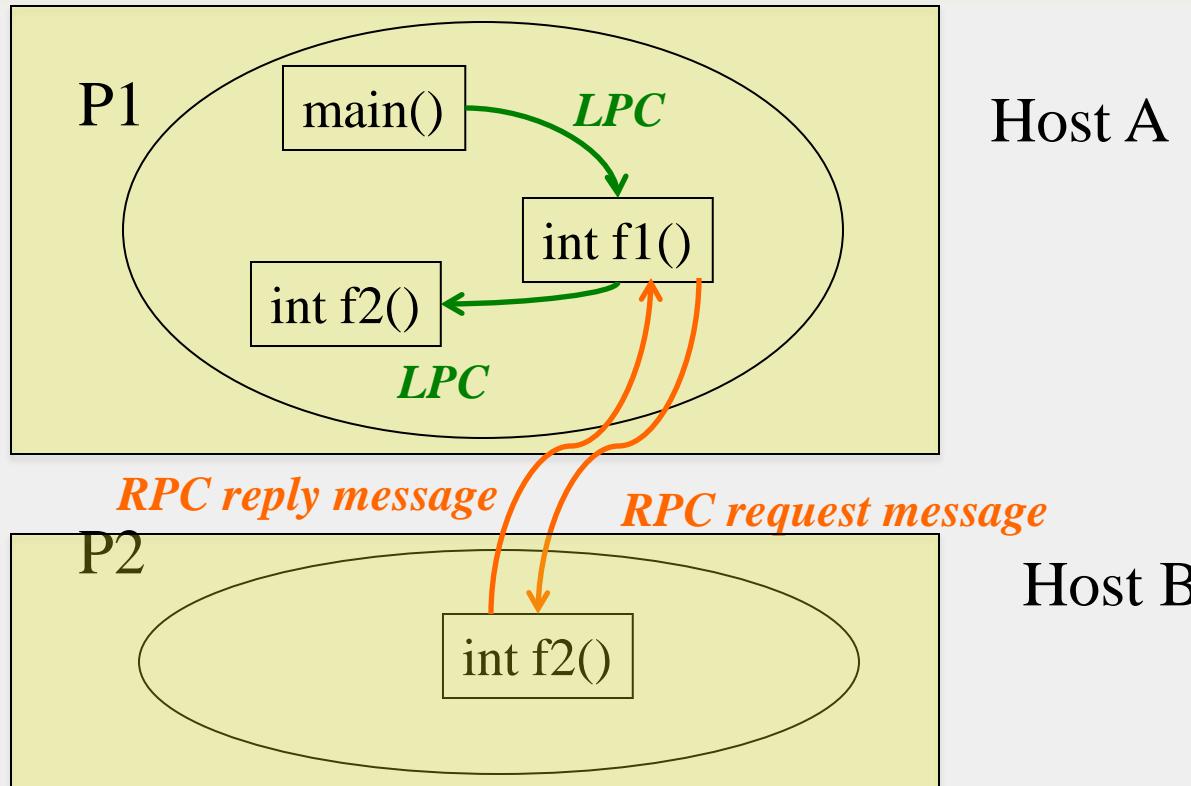


RPCs



Host A

RPCs



RPC CALL SEMANTICS

- Under failures, hard to guarantee exactly-once semantics
- Function may not be executed if
 - Request (call) message is dropped
 - Reply (return) message is dropped
 - Called process fails before executing called function
 - Called process fails after executing called function
 - Hard for caller to distinguish these cases
- Function may be executed multiple times if
 - Request (call) message is duplicated

IMPLEMENTING RPC CALL SEMANTICS

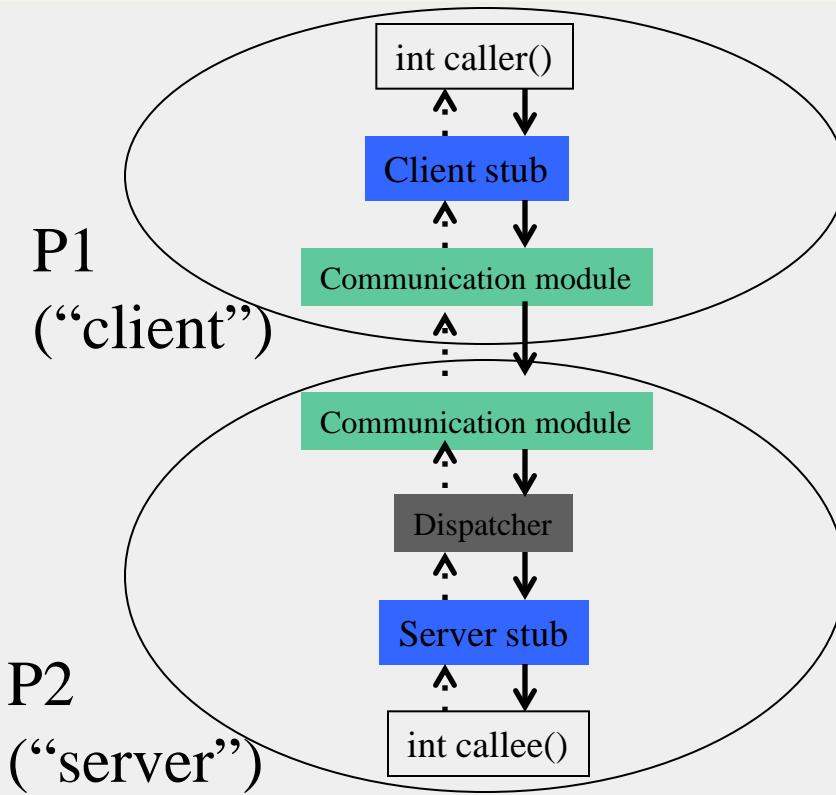
- Possible semantics
 - At most once semantics (e.g., Java RMI)
 - At least once semantics (e.g., Sun RPC)
 - Maybe, i.e., best-effort (e.g., CORBA)

Retransmit request	Filter duplicate requests	Re-execute function or retransmit reply	RPC Semantics
Yes	No	Re-execute	At least once
Yes	Yes	Retransmit	At most once
No	NA	NA	Maybe

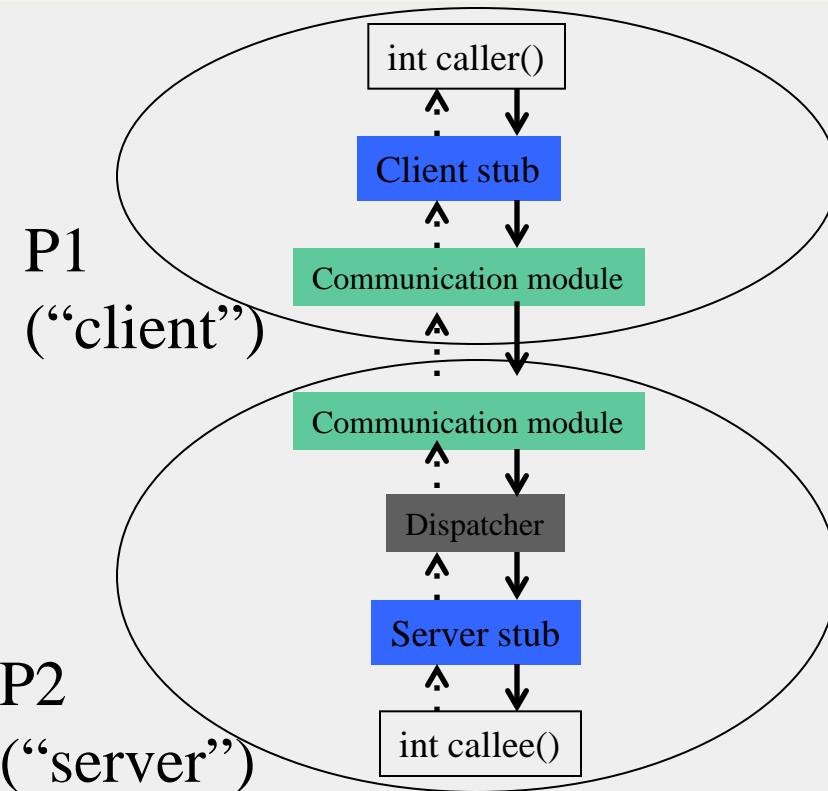
IDEMPOTENT OPERATIONS

- Idempotent operations are those that can be repeated multiple times, without any side effects
- Examples (x is server-side variable)
 - $x=1;$
 - $x=(\text{argument}) y;$
- Non-examples
 - $x=x+1;$
 - $x=x^*2;$
- Idempotent operations can be used with at-least-once semantics

IMPLEMENTING RPCS



RPC COMPONENTS



Client

- **Client stub:** has same function signature as `callee()`
 - Allows same `caller()` code to be used for LPC and RPC
- **Communication Module:** Forwards requests and replies to appropriate hosts

Server

- **Dispatcher:** Selects which server stub to forward request to
- **Server stub:** calls `callee()`, allows it to return a value

GENERATING CODE

- Programmer only writes code for caller function and callee function
- Code for remaining components all **generated automatically** from function signatures (or object interfaces in Object-based languages)
 - E.g., Sun RPC system: Sun XDR interface representation fed into rpcgen compiler
- These components together part of a Middleware system
 - E.g., CORBA (Common Object Request Brokerage Architecture)
 - E.g., Sun RPC
 - E.g., Java RMI

MARSHALLING

- Different architectures use different ways of representing data
 - **Big endian**: Hex 12-AC-33 stored with 12 in lowest address, then AC in next higher address, then 33 in highest address
 - IBM z, System 360
 - **Little endian**: Hex 12-AC-33 stored with 33 in lowest address, then AC in next higher address, then 12
 - Intel
- Caller (and callee) process uses its own *platform-dependent* way of storing data
- Middleware has a common data representation (CDR)
 - *Platform-independent*

MARSHALLING (2)

- Middleware has a common data representation (CDR)
 - Platform-independent
- Caller process converts arguments into CDR format
 - Called “Marshalling”
- Callee process extracts arguments from message into its own platform-dependent format
 - Called “Unmarshalling”
- Return values are marshalled on callee process and unmarshalled at caller process

NEXT

- Now that we know RPCs, we can use them as a building block to understand transactions



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CONCURRENCY CONTROL

Lecture B

TRANSACTIONS

TRANSACTION

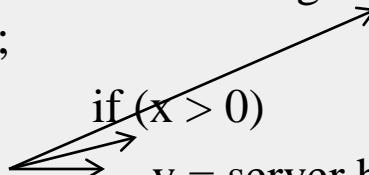
- Series of operations executed by client
- Each operation is an RPC to a server
- Transaction either
 - Completes and *commits* all its operations at server
 - Commit = reflect updates on server-side objects
 - Or *aborts* and has no effect on server

EXAMPLE: TRANSACTION

Client code:

```
int transaction_id = openTransaction();
x = server.getFlightAvailability(ABC, 123,
date);
if(x > 0)
    y = server.bookTicket(ABC, 123, date);
    server.putSeat(y, "aisle");
// commit entire transaction or abort
closeTransaction(transaction_id);
```

RPCs

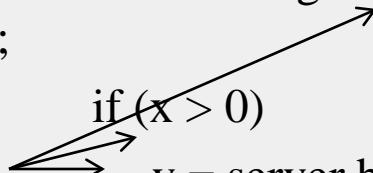


EXAMPLE: TRANSACTION

Client code:

```
int transaction_id = openTransaction();
x = server.getFlightAvailability(ABC, 123, date);           // read(ABC, 123, date)
if(x > 0)                                                 // write(ABC, 123, date)
    y = server.bookTicket(ABC, 123, date);                  // write(y)
    server.putSeat(y, "aisle");
// commit entire transaction or abort
closeTransaction(transaction_id);
```

RPCs



ATOMICITY AND ISOLATION

- Atomicity: All or nothing principle: a transaction should either i) complete successfully, so its effects are recorded in the server objects; or ii) the transaction has no effect at all.
- Isolation: Need a transaction to be indivisible (atomic) from the point of view of other transactions
 - No access to intermediate results/states of other transactions
 - Free from interference by operations of other transactions
- But...
- Clients and/or servers might crash
- Transactions could run concurrently, i.e., with multiple clients
- Transactions may be distributed, i.e., across multiple servers

ACID PROPERTIES FOR TRANSACTIONS

- **Atomicity:** All or nothing
- **Consistency:** If the server starts in a consistent state, the transaction ends the server in a consistent state.
- **Isolation:** Each transaction must be performed without interference from other transactions, i.e., non-final effects of a transaction must not be visible to other transactions.
- **Durability:** After a transaction has completed successfully, all its effects are saved in permanent storage.

MULTIPLE CLIENTS, ONE SERVER

- What could go wrong?

1. LOST UPDATE PROBLEM

Transaction T1

```
x = getSeats(ABC123);  
    // x = 10  
if(x > 1)  
    x = x - 1;  
write(x, ABC123);  
  
commit
```

Transaction T2

```
x = getSeats(ABC123);  
    if(x > 1)    // x = 10  
  
    x = x - 1;  
    write(x, ABC123);  
  
commit
```

At Server: seats = 10

T1's or T2's update was lost!

seats = 9

seats = 9

2. INCONSISTENT RETRIEVAL PROBLEM

Transaction T1

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);  
// ABC123 = 5 now  
  
write(y+5, ABC789);  
  
commit
```

Transaction T2

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
// x = 5, y = 15  
print("Total:" x+y);  
// Prints "Total: 20"  
  
commit
```

At Server:
ABC123 = 10
ABC789 = 15

**T2's sum is the wrong value!
Should have been "Total: 25"**

NEXT

- How to prevent transactions from affecting each other



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CONCURRENCY CONTROL

Lecture C

SERIAL EQUIVALENCE

CONCURRENT TRANSACTIONS

- To prevent transactions from affecting each other
 - Could execute them one at a time at server
 - But reduces number of concurrent transactions
 - *Transactions per second* directly related to revenue of companies
 - This metric needs to be maximized
- Goal: increase concurrency while maintaining correctness (ACID)

SERIAL EQUIVALENCE

- An interleaving (say O) of transaction operations is serially equivalent iff (if and only if):
 - There is some ordering (O') of those transactions, one at a time, which
 - Gives the same end-result (for all objects and transactions) as the original interleaving O
 - Where the operations of each transaction occur consecutively (in a batch)
- Says: Cannot distinguish end-result of real operation O from (fake) serial transaction order O'

CHECKING FOR SERIAL EQUIVALENCE

- An operation has an **effect** on
 - The server object if it is a write
 - The client (returned value) if it is a read
- Two operations are said to be conflicting operations, if their *combined effect* depends on the order they are executed
 - read(x) and write(x)
 - write(x) and read(x)
 - write(x) and write(x)
 - NOT read(x) and read(x): swapping them doesn't change end-results
 - NOT read/write(x) and read/write(y): swapping them ok

CHECKING FOR SERIAL EQUIVALENCE (2)

- *Two transactions are serially equivalent if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.*
 - Take all pairs of conflict operations, one from T1 and one from T2
 - If the T1 operation was reflected first on the server, mark the pair as “(T1, T2),” otherwise mark it as “(T2, T1)”
 - All pairs should be marked as either “(T1, T2)” or all pairs should be marked as “(T2, T1).”

1. LOST UPDATE PROBLEM – CAUGHT!

Transaction T1

```
x = getSeats(ABC123);  
    // x = 10  
if(x > 1)  
    x = x - 1;  
write(x, ABC123);  
  
commit
```

Transaction T2

```
x = getSeats(ABC123);  
    if(x > 1)    // x = 10  
        x = x - 1;  
write(x, ABC123);  
  
commit
```

At Server: seats = 10

T1's or T2's update was lost!

seats = 9

seats = 9

2. INCONSISTENT RETRIEVAL PROBLEM – CAUGHT!

Transaction T1

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);
```

```
write(y+5, ABC789);
```

```
commit
```

Transaction T2

```
(T1, T2) → x = getSeats(ABC123);  
y = getSeats(ABC789);
```

```
(T2, T1) // x = 5, y = 15  
print("Total:" x+y);
```

```
// Prints "Total: 20"  
commit
```

At Server:
ABC123 = 10
ABC789 = 15

**T2's sum is the wrong value!
Should have been “Total: 25”**

WHAT'S OUR RESPONSE?

- At commit point of a transaction T, check for serial equivalence with all other transactions
 - Can limit to transactions that overlapped with T
- If not serially equivalent
 - Abort T
 - Roll back (undo) any writes that T did to server objects

CAN WE DO BETTER?

- Aborting => wasted work
- Can you prevent violations from occurring?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CONCURRENCY CONTROL

Lecture D

PESSIMISTIC CONCURRENCY

TWO APPROACHES

- Preventing isolation from being violated can be done in two ways
 1. **Pessimistic** concurrency control
 2. **Optimistic** concurrency control

PESSIMISTIC VS. OPTIMISTIC

- **Pessimistic**: assume the worst, prevent transactions from accessing the same object
 - E.g., Locking
- **Optimistic**: assume the best, allow transactions to write, but check later
 - E.g., Check at commit time, multi-version approaches

PESSIMISTIC: EXCLUSIVE LOCKING

- Each object has a lock
- At most one transaction can be inside lock
- Before reading or writing object O, transaction T must call **lock(O)**
 - Blocks if another transaction already inside lock
- After entering lock T can read and write O multiple times
- When done (or at commit point), T calls **unlock(O)**
 - If other transactions waiting at lock(O), allows one of them in
- Sound familiar? (This is mutual exclusion!)

CAN WE IMPROVE CONCURRENCY?

- More concurrency => more transactions per second => more revenue (\$\$\$)
- Real-life workloads have a lot of read-only or read-mostly transactions
 - Exclusive locking reduces concurrency
 - Hint: Ok to allow two transactions to concurrently read an object, since read-read is not a conflicting pair

ANOTHER APPROACH: READ-WRITE LOCKS

- Each object has a lock that can be held in one of two modes
 - **Read mode**: multiple transactions allowed in
 - **Write mode**: exclusive lock
- Before first reading O, transaction T calls `read_lock(O)`
 - T allowed in only if *all* transactions inside lock for O all entered via read mode
 - Not allowed if *any* transaction inside lock for O entered via write mode

READ-WRITE LOCKS (2)

- Before first writing O, call `write_lock(O)`
 - Allowed in only if no other transaction inside lock
- If T already holds `read_lock(O)`, and wants to write, call `write_lock(O)` to *promote* lock from read to write mode
 - Succeeds only if no other transactions in write mode or read mode
 - Otherwise, T blocks
- `Unlock(O)` called by transaction T releases any lock on O by T

GUARANTEEING SERIAL EQUIVALENCE WITH LOCKS

- Two-phase locking
 - A transaction cannot acquire (or promote) any locks after it has started releasing locks
 - Transaction has two phases
 1. Growing phase: only acquires or promotes locks
 2. Shrinking phase: only releases locks
 - Strict two phase locking: releases locks only at commit point

WHY TWO-PHASE LOCKING => SERIAL EQUIVALENCE?

- Proof by contradiction
- Assume two-phase locking system where serial equivalence is violated for some two transactions T1, T2
- Two facts must then be true:
 - (A) For some object O1, there were conflicting operations in T1 and T2 such that the time ordering pair is (T1, T2)
 - (B) For some object O2, the conflicting operation pair is (T2, T1)
 - (A) \Rightarrow T1 released O1's lock and T2 acquired it after that
 \Rightarrow T1's shrinking phase is before or overlaps with T2's growing phase
- Similarly, (B) \Rightarrow T2's shrinking phase is before or overlaps with T1's growing phase
- But both these cannot be true!

DOWNSIDE OF LOCKING

- Deadlocks!

DOWNSIDE OF LOCKING – DEADLOCKS!

Transaction T1

```
Lock(ABC123);
```

```
x = write(10, ABC123);
```

```
Lock(ABC789);
```

// Blocks waiting for T2

...

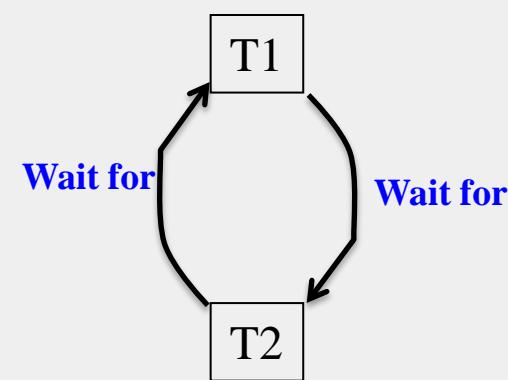
Transaction T2

```
Lock(ABC789);
```

```
y = write(15, ABC789);
```

```
Lock(ABC123);
```

... // Blocks waiting for T1



WHEN Do DEADLOCKS Occur?

- 3 necessary conditions for a deadlock to occur
 1. Some objects are accessed in exclusive lock modes
 2. Transactions holding locks cannot be preempted
 3. There is a circular wait (cycle) in the Wait-for graph
- “Necessary” = if there’s a deadlock, these conditions are all definitely true
- (Conditions not sufficient: if they’re present, it doesn’t imply a deadlock is present.)

COMBATING DEADLOCKS

1. Lock **timeout**: abort transaction if lock cannot be acquired within timeout
 - ⌚ Expensive, wasted work
2. Deadlock **Detection**:
 - Keep track of Wait-for graph (e.g., via Global Snapshot algorithm), and
 - Find cycles in it (e.g., periodically)
 - If find cycle, there's a deadlock => Abort one or more transactions to break cycle
 - ⌚ Still allows deadlocks to occur

COMBATING DEADLOCKS (2)

3. Deadlock Prevention

- Set up the system so one of the *necessary conditions* is violated
 - 1. *Some objects are accessed in exclusive lock modes*
 - Fix: Allow read-only access to objects
 - 2. *Transactions holding locks cannot be preempted*
 - Fix: Allow preemption of some transactions
 - 3. *There is a circular wait (cycle) in the Wait-for graph*
 - Fix: Lock all objects in the beginning; if fail any, abort transaction
=> No cycles in Wait-for graph

NEXT

- Can we allow more concurrency?
- Optimistic Concurrency Control



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CONCURRENCY CONTROL

Lecture E

OPTIMISTIC CONCURRENCY CONTROL

OPTIMISTIC CONCURRENCY CONTROL

- Increases concurrency more than pessimistic concurrency control
- Increases transactions per second
- For non-transaction systems, increases operations per second and lowers latency
- Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon's Dynamo
- Preferable than pessimistic when conflicts are *expected to be rare*
 - But still need to ensure conflicts are caught!

FIRST-CUT APPROACH

- Most basic approach
 - Write and read objects at will
 - Check for serial equivalence at commit time
 - If abort, roll back updates made
 - An abort may result in other transactions that read dirty data, also being aborted
 - Any transactions that read from *those* transactions also now need to be aborted
- ⌚ Cascading aborts

SECOND APPROACH: TIMESTAMP ORDERING

- Assign each transaction an id
- Transaction id determines its position in **serialization order**
- Ensure that for a transaction T, both are true:
 1. T's **write** to object O allowed only if **transactions that have read or written O had lower ids than T.**
 2. T's **read** to object O is allowed only if **O was last written by a transaction with a lower id than T.**
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
 - Can we do better?

THIRD APPROACH: MULTI-VERSION CONCURRENCY CONTROL

- For each object
 - A per-transaction version of the object is maintained
 - Marked as *tentative* versions
 - And a **committed** version
- Each tentative version has a timestamp
 - Some systems maintain both a read timestamp and a write timestamp
- On a read or write, find the “correct” tentative version to read or write from
 - “Correct” based on transaction id, and tries to make transactions only read from “immediately previous” transactions

EVENTUAL CONSISTENCY ...

- ... that you've seen in key-value stores ...
- ... is a form of optimistic concurrency control
 - In Cassandra key-value store
 - In DynamoDB key-value store
 - In Riak key-value store
- But since non-transaction systems, the optimistic approach looks different

EVENTUAL CONSISTENCY IN CASSANDRA AND DYNAMO DB

- Only one version of each data item (key-value pair)
- Last-write-wins (LWW)
 - Timestamp, typically based on *physical time*, used to determine whether to overwrite

```
if(new write's timestamp > current object's timestamp)
    overwrite;
else
    do nothing;
```
- With unsynchronized clocks
 - If two writes are close by in time, older write might have a newer timestamp, and might win

EVENTUAL CONSISTENCY IN RIAK KEY-VALUE STORE

- Uses **vector clocks**! (Should sound familiar to you!)
- Implements causal ordering
- Uses vector clocks to detect whether
 1. New write is strictly newer than current value, or
 2. If new write conflicts with existing value
- In case (2), a *sibling value* is created
 - Resolvable by user, or automatically by application (but not by Riak)
- To prevent vector clocks from getting too many entries
 - Size-based pruning
- To prevent vector clocks from having entries updated a long time ago
 - Time-based pruning

SUMMARY

- RPCs and RMIs
- Transactions
- Serial Equivalence
 - Detecting it via conflicting operations
- Pessimistic Concurrency Control:
locking
- Optimistic Concurrency Control



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

REPLICATION CONTROL

Lecture A

REPLICATION

SERVER-SIDE FOCUS

- Concurrency Control = how to coordinate multiple concurrent clients executing operations (or transactions) with a server

Next:

- Replication Control = how to handle operations (or transactions) when there are **multiple servers**, with data perhaps replicated across servers

REPLICATION: WHAT AND WHY

- **Replication** = An object has identical copies, each maintained by a separate server
 - Copies are called “replicas”
- Why replication?
 - **Fault-tolerance**: With k replicas of each object, can tolerate failure of any ($k-1$) servers
 - **Load balancing**: Spread read/write operations out over the k replicas => load lowered by a factor of k compared to a single replica
 - Replication => Higher Availability

AVAILABILITY

- If each server is down a fraction f of the time
 - Server's failure probability
- With no replication, availability of object =
 - = Probability that single copy is up
 - = $(1 - f)$
- With k replicas, availability of object =
 - Probability that at least one replica is up
 - = $1 - \text{Probability that all replicas are down}$
 - = $(1 - f^k)$

NINES AVAILABILITY

- With no replication, availability of object =
 $= (1 - f)$
- With k replicas, availability of object =
 $= (1 - f^k)$

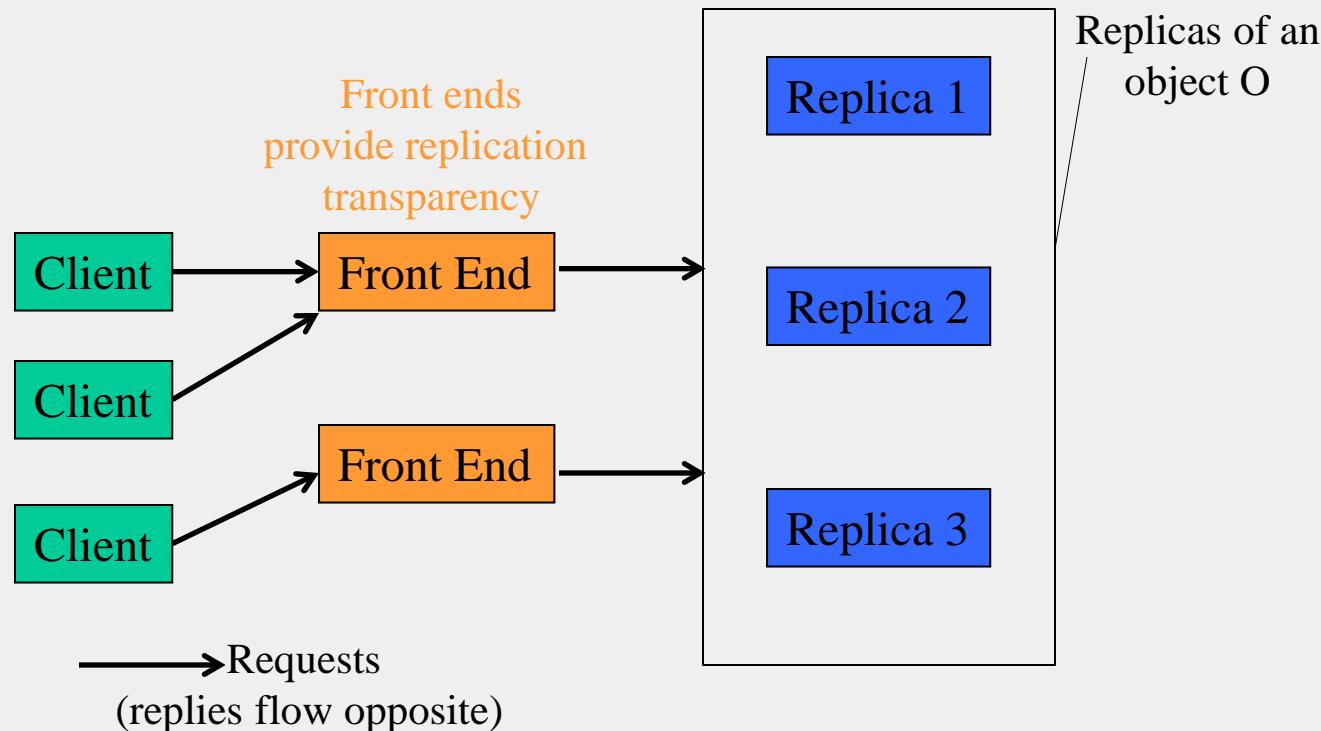
Availability Table

f =failure probability	No replication	$k=3$ replicas	$k=5$ replicas
0.1	90%	99.9%	99.999%
0.05	95%	99.9875%	6 Nines
0.01	99%	99.9999%	10 Nines

WHAT'S THE CATCH?

- Challenge is to maintain two properties
 1. Replication Transparency
 - A client ought not to be aware of multiple copies of objects existing on the server side
 2. Replication Consistency
 - Ensure that clients see single consistent copy of data, in spite of replication
 - For transactions, guarantee ACID

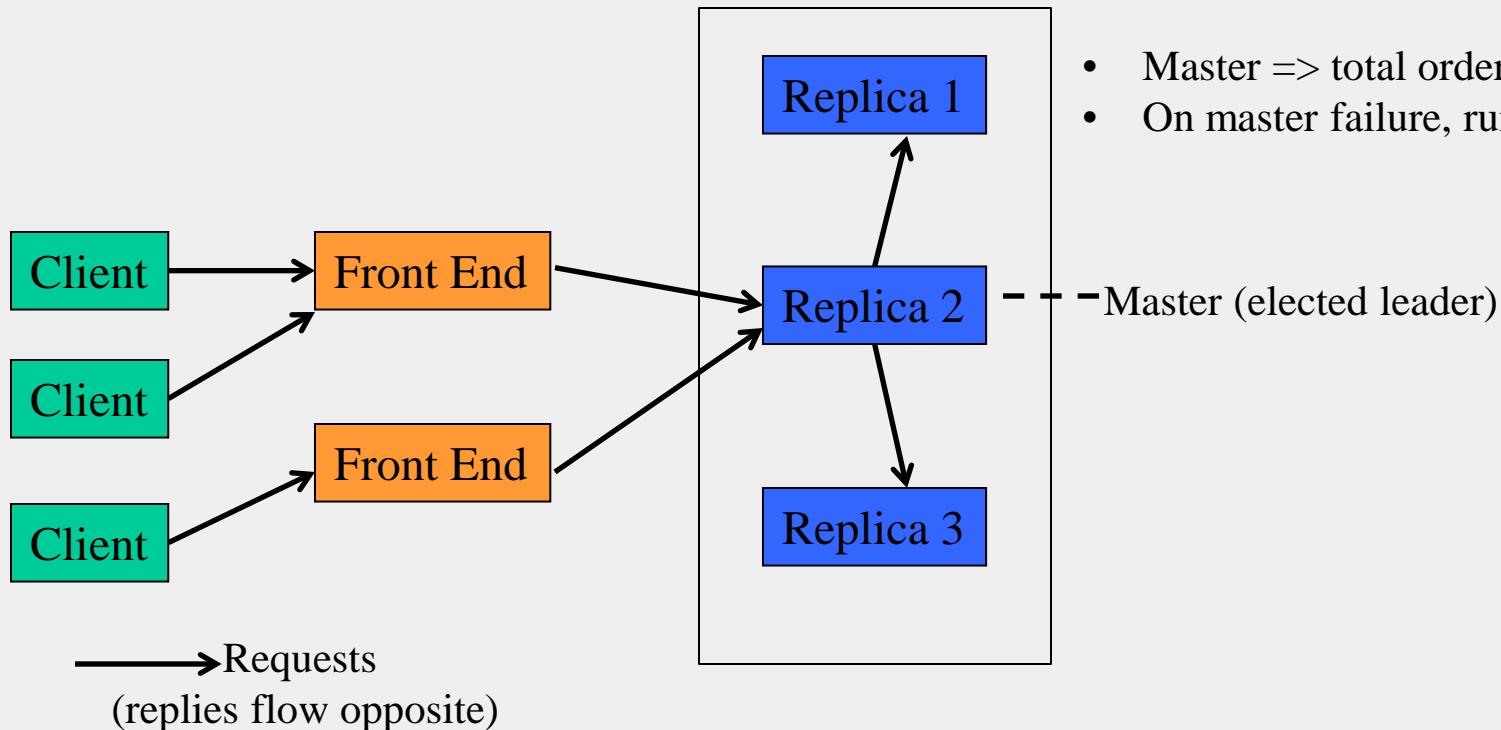
REPLICATION TRANSPARENCY



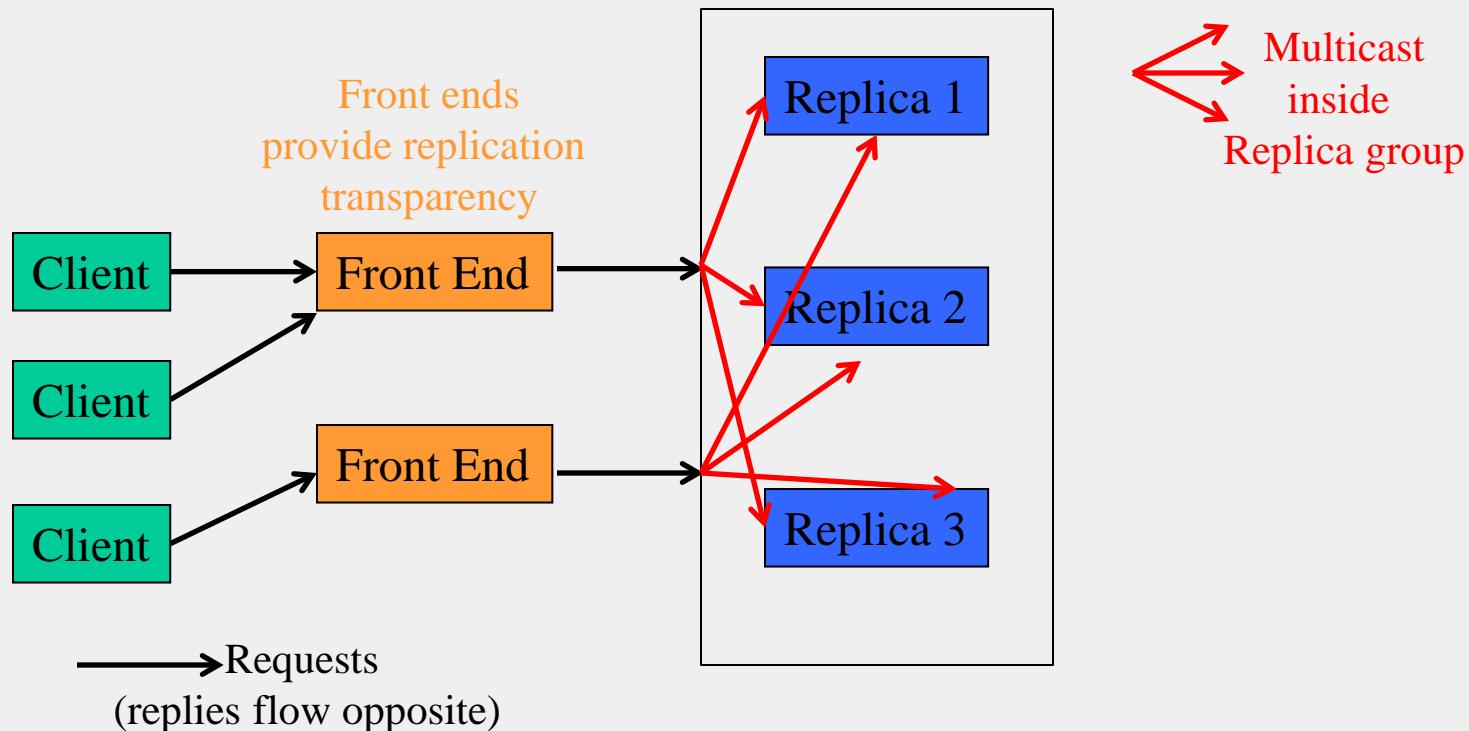
REPLICATION CONSISTENCY

- Two ways to forward updates from front-ends (FEs) to replica group
 - **Passive Replication**: uses a primary replica (master)
 - **Active Replication**: treats all replicas identically
- Both approaches use the concept of “Replicated State Machines”
- Each replica’s code runs the same state machine
 - *Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs. [Schneider 1990]*

PASSIVE REPLICATION



ACTIVE REPLICATION



ACTIVE REPLICATION USING CONCEPTS YOU'VE LEARNT EARLIER

- Can use any flavor of **multicast ordering**, depending on application
 - FIFO ordering
 - Causal ordering
 - Total ordering
 - Hybrid ordering
- Total or Hybrid (*-Total) ordering + Replicated State machines approach
 - => all replicas reflect the same sequence of updates to the object

ACTIVE REPLICATION USING CONCEPTS YOU'VE LEARNT EARLIER (2)

- What about failures?
 - Use **virtual synchrony** (i.e., **view synchrony**)
- Virtual synchrony with total ordering for multicasts =>
 - All replicas see all failures/joins/leaves and all multicasts in the same order
 - Could also use causal (or even FIFO) ordering if application can tolerate it

TRANSACTIONS AND REPLICATION

- One-copy serializability
 - *A concurrent execution of transactions in a replicated database is one-copy-serializable if it is equivalent to a serial execution of these transactions over a single logical copy of the database.*
 - (Or) The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects (i.e., 1 replica per object).
- In a non-replicated system, transactions appear to be performed one at a time in some order.
 - Correctness means **serial equivalence** of transactions
- When objects are replicated, transaction systems for correctness need
 - Serial equivalence + One-copy serializability

NEXT

- More on transactions with distributed servers



CLOUD COMPUTING CONCEPTS

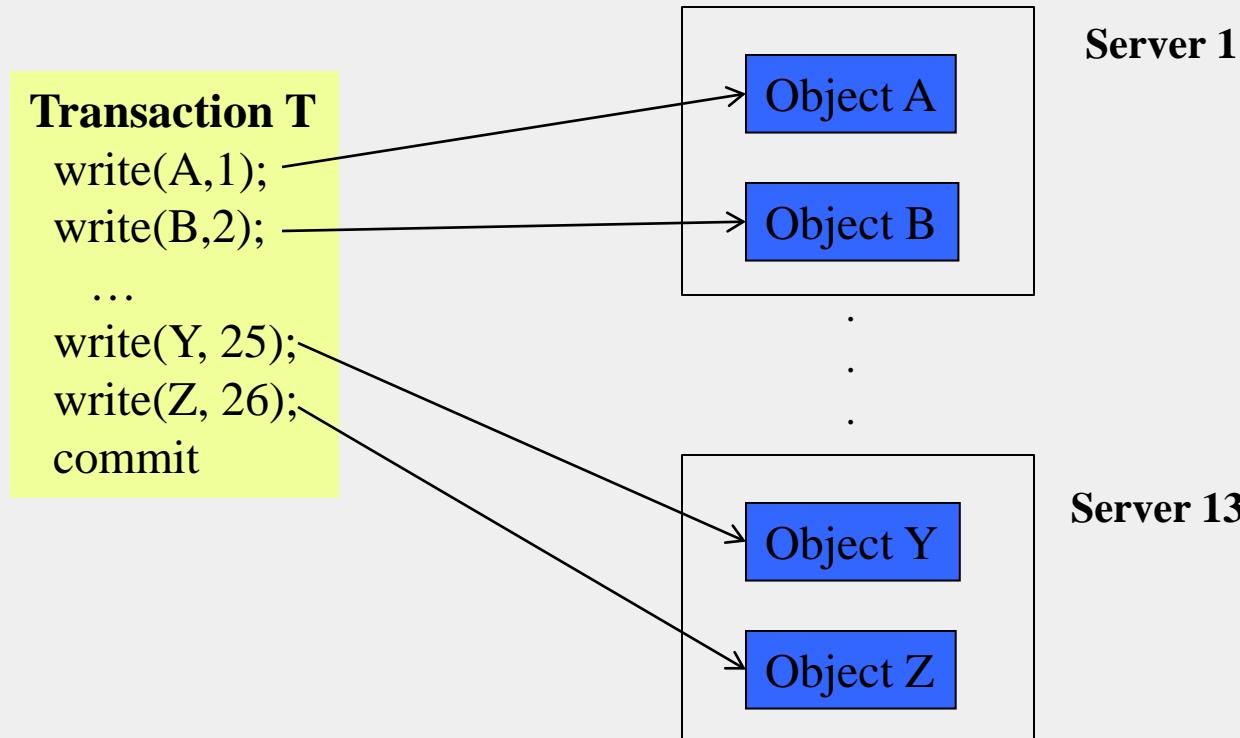
with Indranil Gupta (Indy)

REPLICATION CONTROL

Lecture B

TWO-PHASE COMMIT

TRANSACTIONS WITH DISTRIBUTED SERVERS



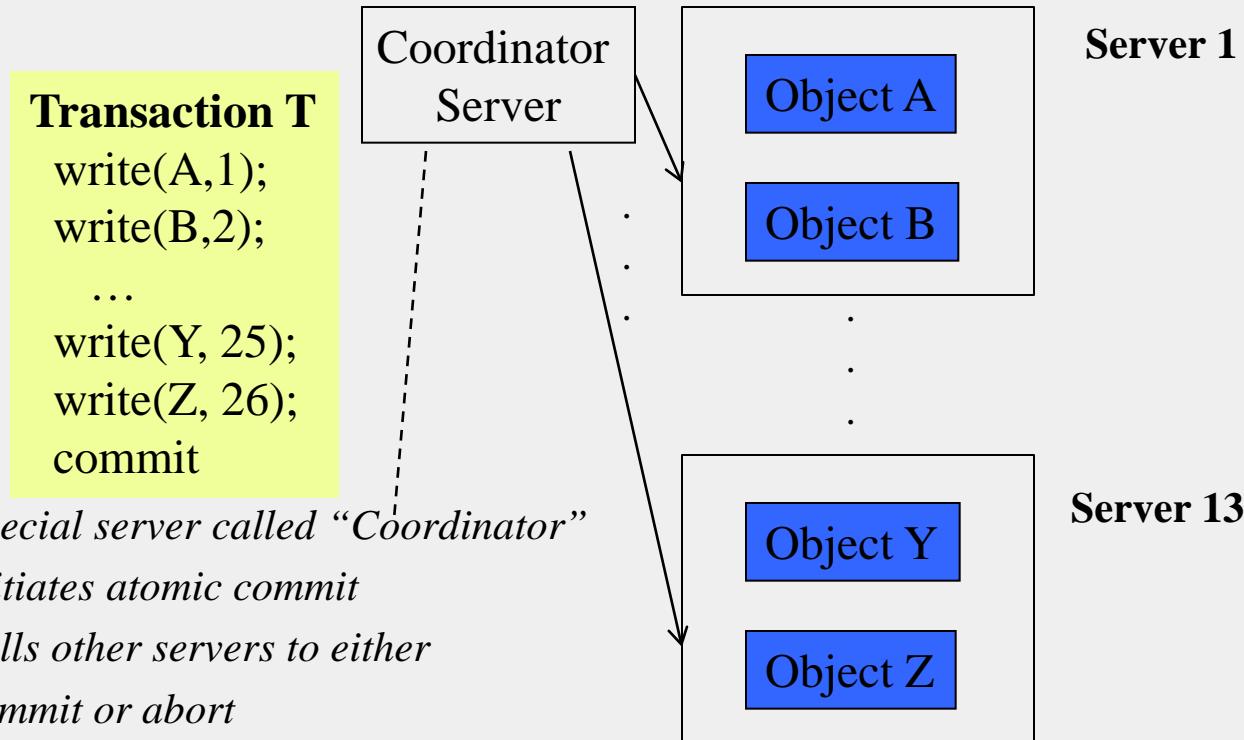
TRANSACTIONS WITH DISTRIBUTED SERVERS (2)

- Transaction T may touch objects that reside on different servers
- When T tries to commit
 - Need to ensure all these servers commit their updates from T => T will commit
 - Or none of these servers commit => T will abort
- What problem is this?

TRANSACTIONS WITH DISTRIBUTED SERVERS (2)

- Transaction T may touch objects that reside on different servers
- When T tries to commit
 - Need to ensure all these servers commit their updates from T => T will commit
 - Or none of these servers commit => T will abort
- What problem is this?
 - Consensus!
 - (It's called the “Atomic Commit problem”)

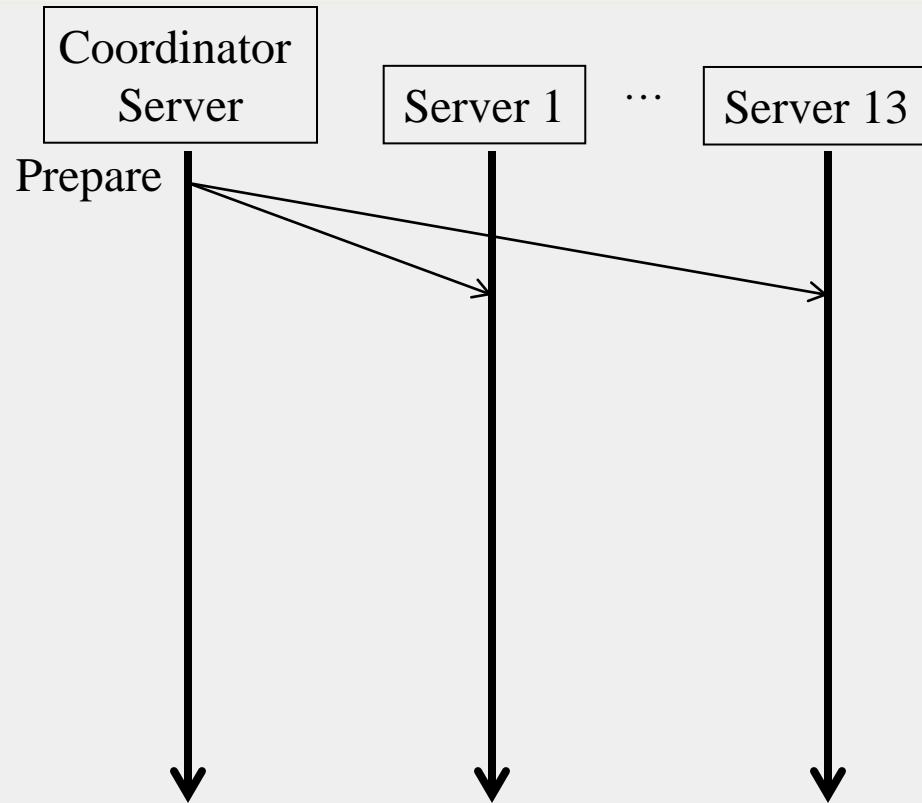
ONE-PHASE COMMIT



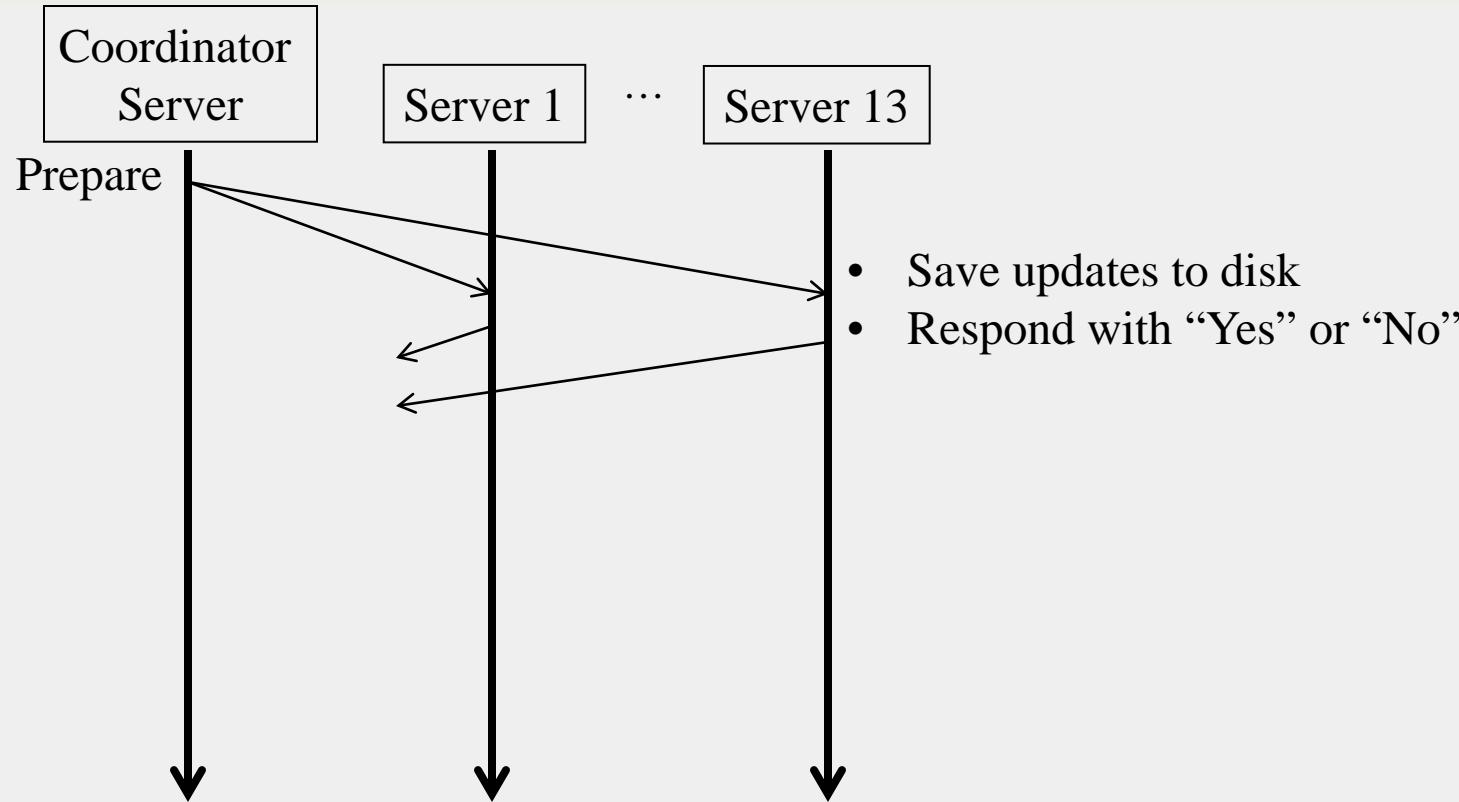
ONE-PHASE COMMIT: ISSUES

- Server with object has no say in whether transaction commits or aborts
 - If object corrupted, it cannot commit (while other servers have committed)
- Server may crash before receiving commit message, with some updates still in memory

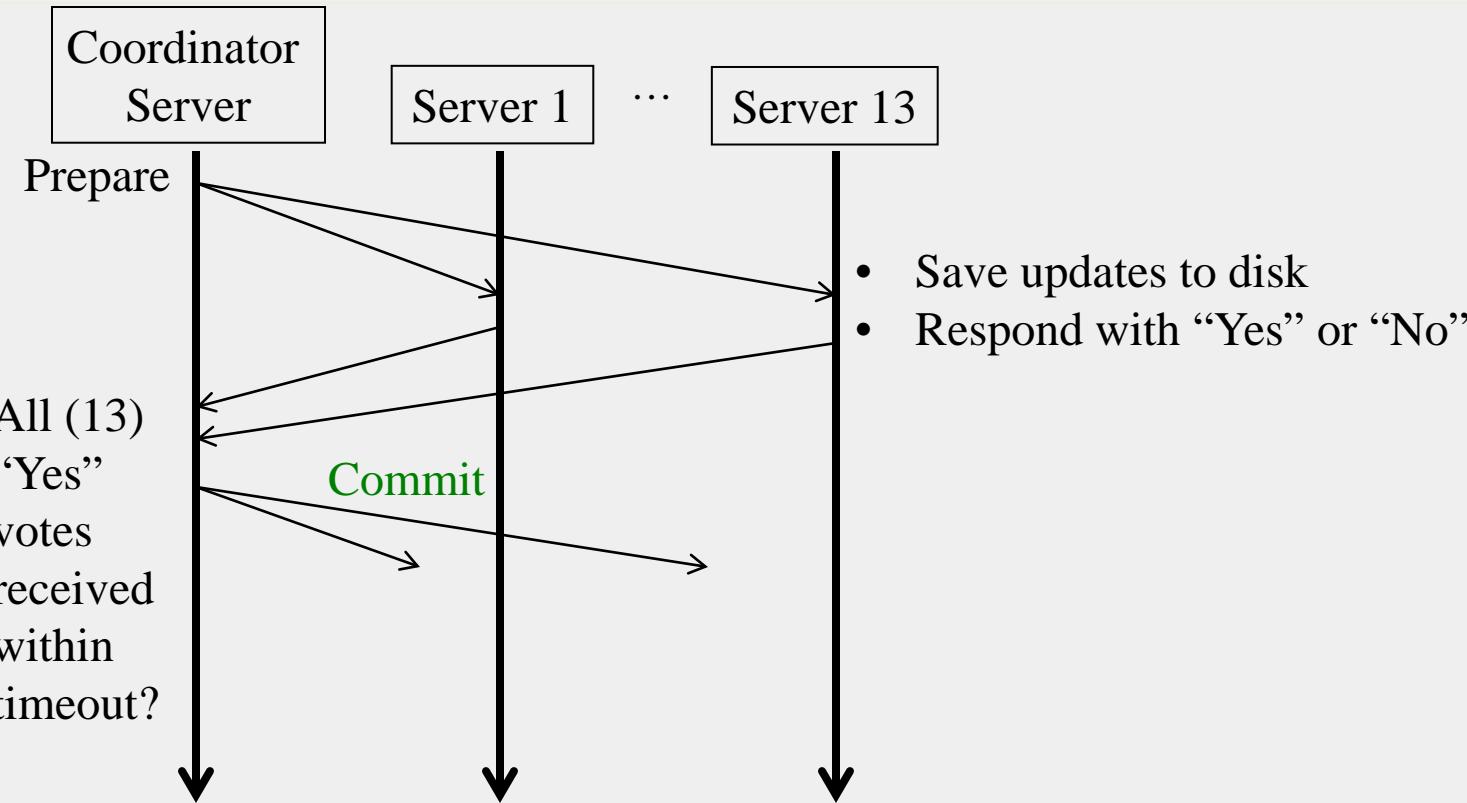
TWO-PHASE COMMIT



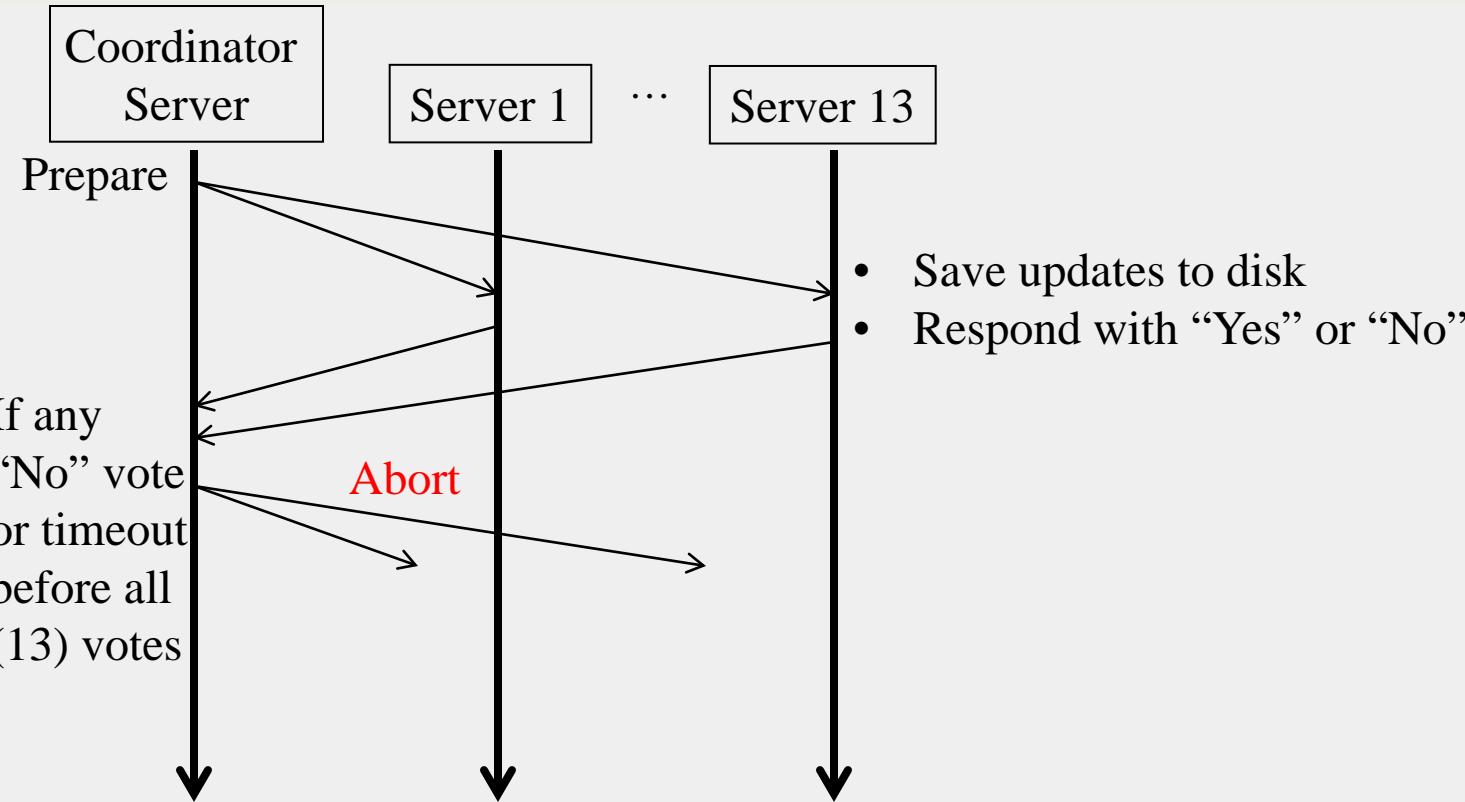
TWO-PHASE COMMIT



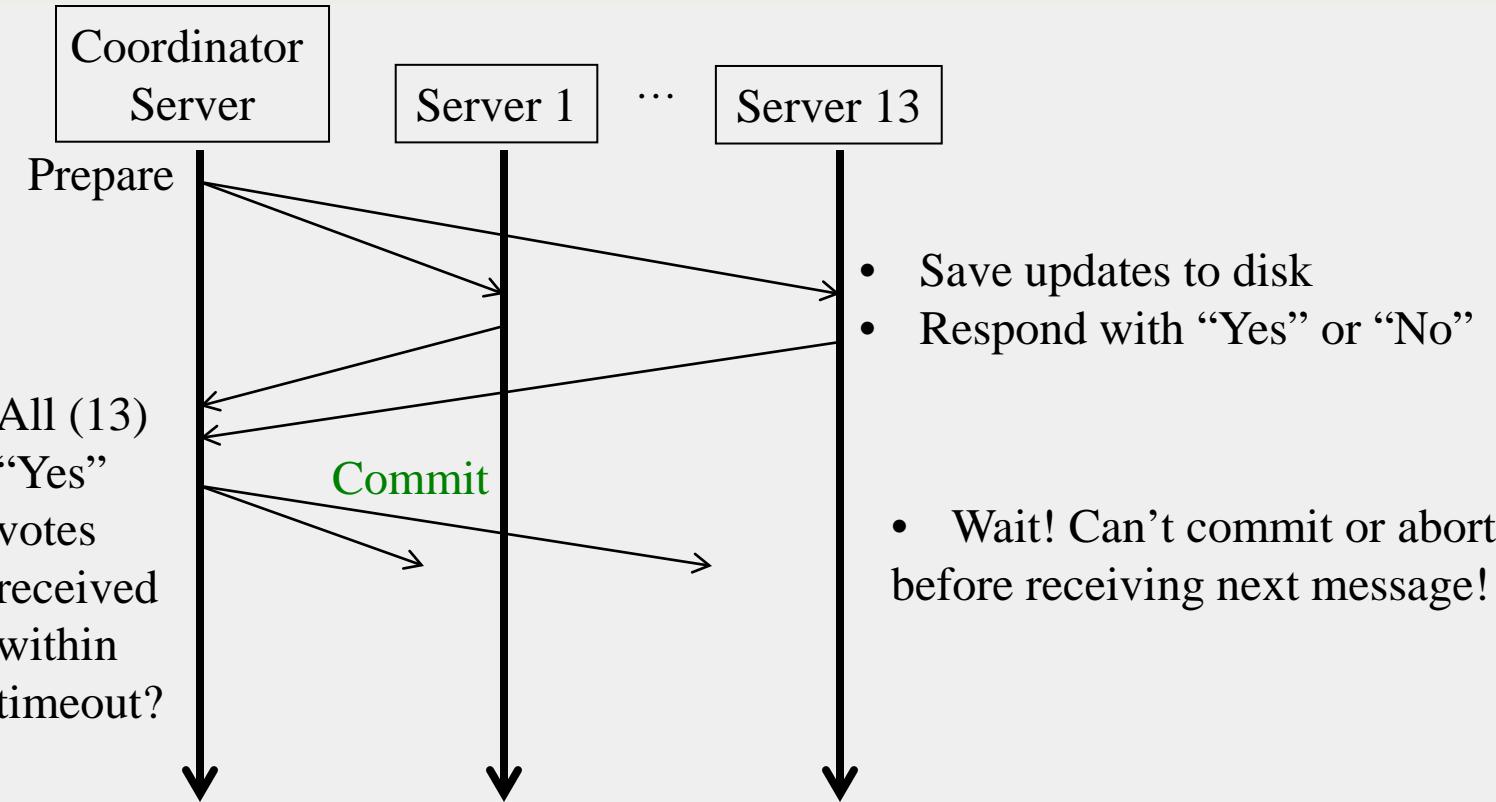
TWO-PHASE COMMIT



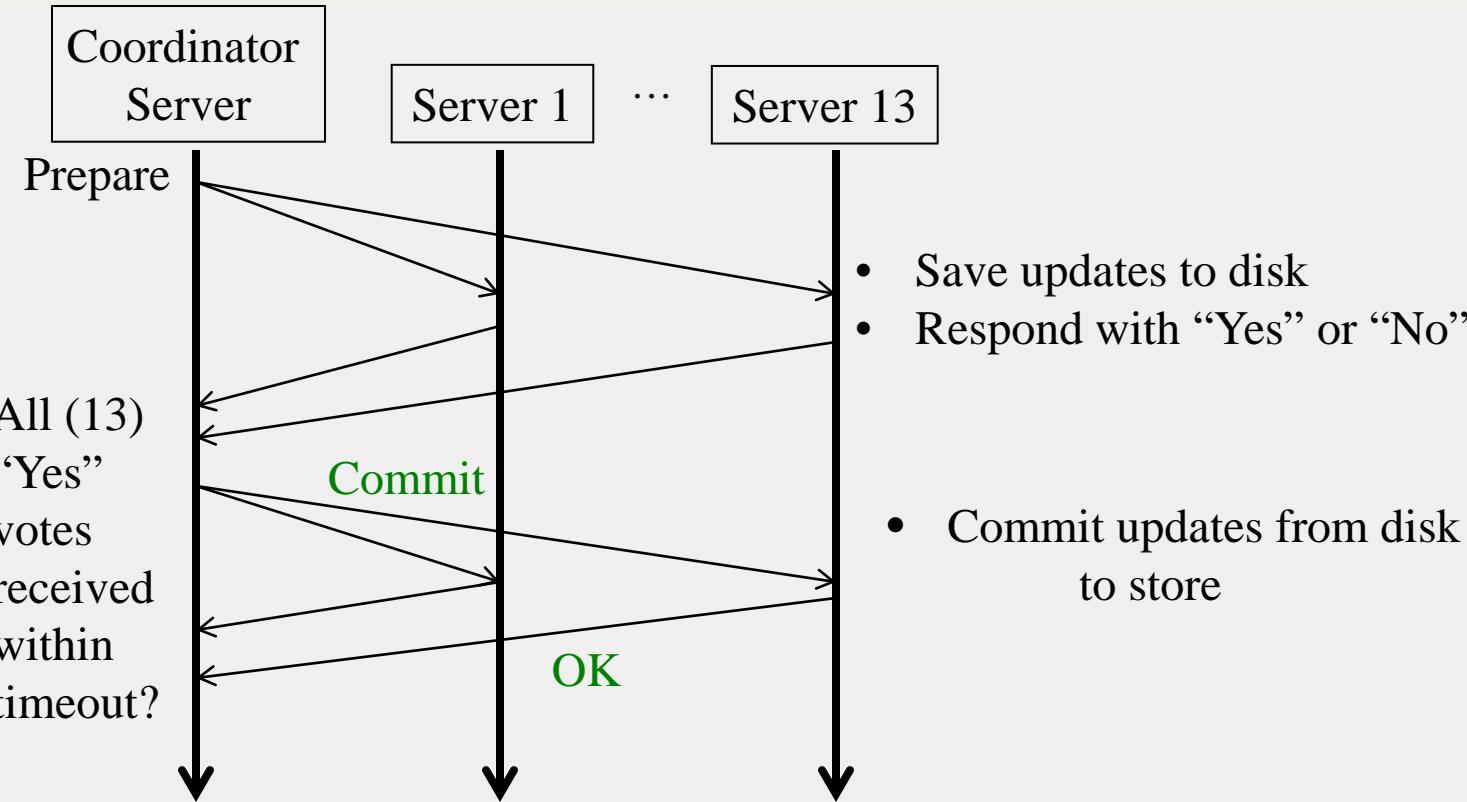
TWO-PHASE COMMIT



TWO-PHASE COMMIT



TWO-PHASE COMMIT



FAILURES IN TWO-PHASE COMMIT

- To deal with server crashes
 - Each server saves tentative updates into permanent storage, right before replying Yes/No in first phase. Retrievable after crash recovery.
 - Coordinator logs votes and decisions too
- To deal with coordinator crashes
 - Coordinator logs all decisions and received/sent messages on disk
 - After recovery or new election => new coordinator takes over
- To deal with Prepare message loss
 - The server may decide to abort unilaterally after a timeout for first phase (server will always vote No, and so coordinator will also eventually abort)

FAILURES IN TWO-PHASE COMMIT (2)

- To deal with Yes/No message loss, coordinator aborts the transaction after a timeout (pessimistic!). It must announce Abort message to all.
- To deal with Commit or Abort message loss
 - Server can poll coordinator (repeatedly)
- If server voted Yes, it cannot commit unilaterally before receiving Commit message
- If server voted No, can abort right away (why?)

USING PAXOS IN DISTRIBUTED SERVERS

Atomic Commit

- Can also use Paxos to decide whether to commit a transaction or not
- But need to ensure that if any server votes No, everyone aborts

Ordering updates

- Paxos can also be used by replica group (for an object) to order all updates
 - Server proposes message for next sequence number
 - Group reaches consensus (or not)

SUMMARY

- Multiple servers in cloud
 - Replication for fault-tolerance
 - Load balancing across objects
- Replication flavors using concepts we learnt earlier
 - Active replication
 - Passive replication
- Transactions and distributed servers
 - Two-phase commit



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

DISTRIBUTED GRAPH PROCESSING

Lecture A

DISTRIBUTED GRAPH PROCESSING

WHAT WE'LL COVER

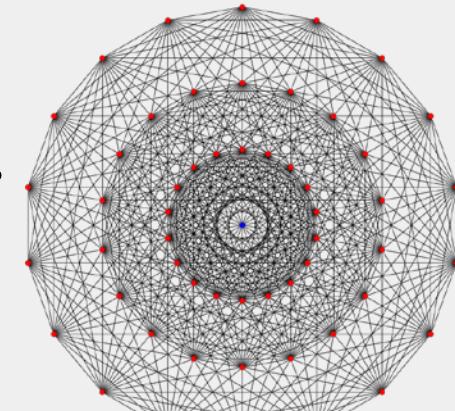
- Distributed Graph Processing
- Google's Pregel system
 - Inspiration for many newer graph processing systems: Piccolo, Giraph, GraphLab, PowerGraph, LFGraph, X-Stream, etc.

WHAT'S A GRAPH?

- A graph is not a plot!
- A graph is a “network”
- A graph has **vertices** (i.e., nodes)
 - E.g., in the Facebook graph, each user = a vertex (or a node)
- A graph has **edges** that connect pairs of vertices
 - E.g., in the Facebook graph, a friend relationship = an edge

LOTS OF GRAPHS

- Large graphs are all around us
 - Internet Graph: vertices are routers/switches and edges are links
 - World Wide Web: vertices are webpages, and edges are URL links on a webpage pointing to another webpage
 - Called “Directed” graph as edges are uni-directional
 - Social graphs: Facebook, Twitter, LinkedIn
 - Biological graphs: DNA interaction graphs, ecosystem graphs, etc.



Source: Wikimedia Commons

GRAPH PROCESSING OPERATIONS

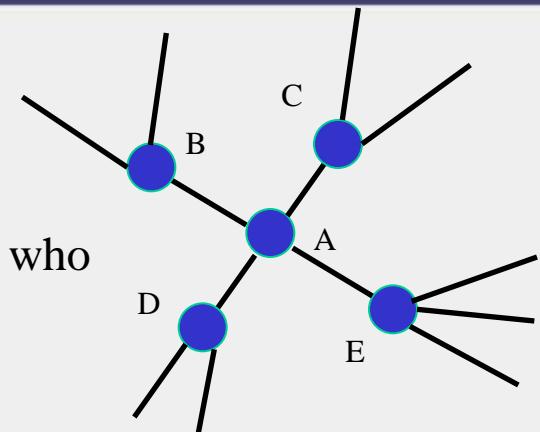
- Need to derive properties from these graphs
- Need to summarize these graphs into statistics
- E.g., find shortest paths between pairs of vertices
 - Internet (for routing)
 - LinkedIn (degrees of separation)
- E.g., do matching
 - Dating graphs in match.com (for better dates)
- And many (many) other examples!

WHY HARD?

- Because these graphs are large!
 - Human social network has 100s Millions of vertices and Billions of edges
 - WWW has Millions of vertices and edges
- Hard to store the entire graph on one server and process it
 - Slow on one server (even if beefy!)
- Use distributed cluster/cloud!

TYPICAL GRAPH PROCESSING APPLICATION

- Works in *iterations*
- Each vertex assigned a *value*
- In each iteration, each vertex:
 1. Gathers values from its immediate neighbors (vertices who join it directly with an edge). E.g., @A: B→A, C→A, D→A,...
 2. Does some computation using its own value and its neighbors values.
 3. Updates its new value and sends it out to its neighboring vertices. E.g., A→B, C, D, E
- Graph processing terminates after: i) fixed iterations, or ii) vertices stop changing values

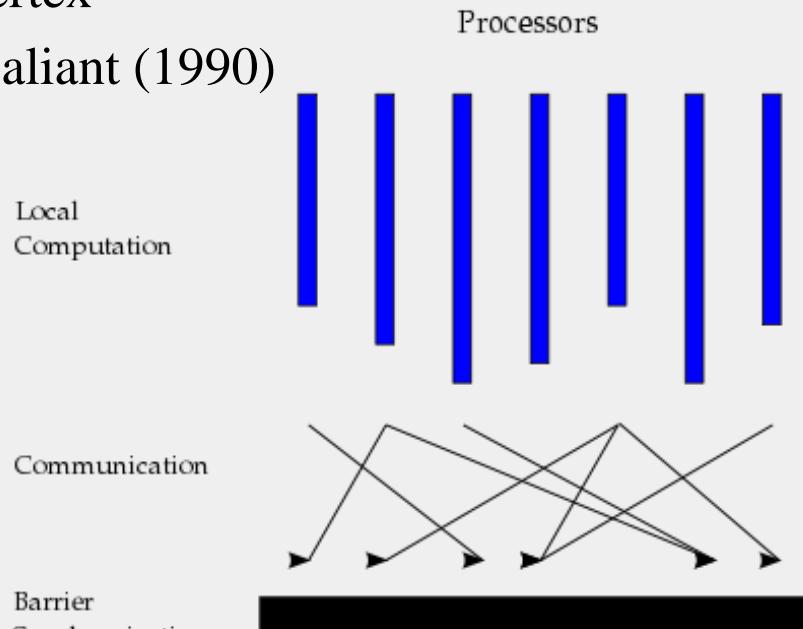


HADOOP/MAPREDUCE TO THE RESCUE?

- Multi-stage Hadoop
- Each stage == 1 graph iteration
- Assign vertex ids as keys in the reduce phase
 - ☺ Well-known
 - ☹ At the end of every stage, transfer all vertices over network
 - ☹ All vertex values written to HDFS (file system)
 - ☹ Very slow!

BULK SYNCHRONOUS PARALLEL MODEL

- “Think like a vertex”
- Originally by Valiant (1990)



Source: http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

BASIC DISTRIBUTED GRAPH PROCESSING

- “Think like a vertex”
- Assign each vertex to one server
- Each server thus gets a subset of vertices
- In each iteration, each server performs **Gather-Apply-Scatter** for all its assigned vertices
 - Gather: get all neighboring vertices’ values
 - Apply: compute own new value from own old value and gathered neighbors’ values
 - Scatter: send own new value to neighboring vertices

ASSIGNING VERTICES

- How to decide which server a given vertex is assigned to?
- Different options
 - Hash-based: $\text{Hash}(\text{vertex id}) \bmod \text{number of servers}$
 - Remember consistent hashing from P2P systems?!
 - Locality-based: Assign vertices with more neighbors to the same server as its neighbors
 - Reduces server to server communication volume after each iteration

PREGEL SYSTEM BY GOOGLE

- Pregel uses the master/worker model
 - Master (one server)
 - Maintains list of worker servers
 - Monitors workers; restarts them on failure
 - Provides Web-UI monitoring tool of job progress
 - Worker (rest of the servers)
 - Processes its vertices
 - Communicates with the other workers
- Persistent data is stored as files on a distributed storage system (such as GFS or BigTable)
- Temporary data is stored on local disk

PREGEL EXECUTION

1. Many copies of the program begin executing on a cluster
2. The master assigns a partition of input (vertices) to each worker
 - Each worker loads the vertices and marks them as *active*
3. The master instructs each worker to perform a iteration
 - Each worker loops through its active vertices & computes for each vertex
 - Messages can be sent whenever, but need to be delivered before the end of the iteration (i.e., the barrier)
 - When all workers reach iteration barrier, master starts next iteration
4. Computation halts when, in some iteration: no vertices are inactive and when no messages are in transit
5. Master instructs each worker to save its portion of the graph

FAULT-TOLERANCE IN PREGEL

- **Checkpointing**
 - Periodically, master instructs the workers to save state of their partitions to persistent storage
 - e.g., Vertex values, edge values, incoming messages
- **Failure detection**
 - Using periodic “ping” messages from master → worker
- **Recovery**
 - The master reassigns graph partitions to the currently available workers
 - The workers all reload their partition state from most recent available checkpoint

How FAST Is It?

- Shortest paths from one vertex to all vertices
 - SSSP: “Single Source Shortest Path”
- On 1 Billion vertex graph (tree)
 - 50 workers: 180 seconds
 - 800 workers: 20 seconds
- 50 B vertices on 800 workers: 700 seconds (~12 minutes)
- Pretty Fast!

SUMMARY

- Lots of (large) graphs around us
- Need to process these
- MapReduce not a good match
- Distributed Graph Processing systems: Pregel by Google
- Many follow-up systems
 - Piccolo, Giraph: Pregel-like
 - GraphLab, PowerGraph, LFGraph, X-Stream: more advanced



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SCHEDULING

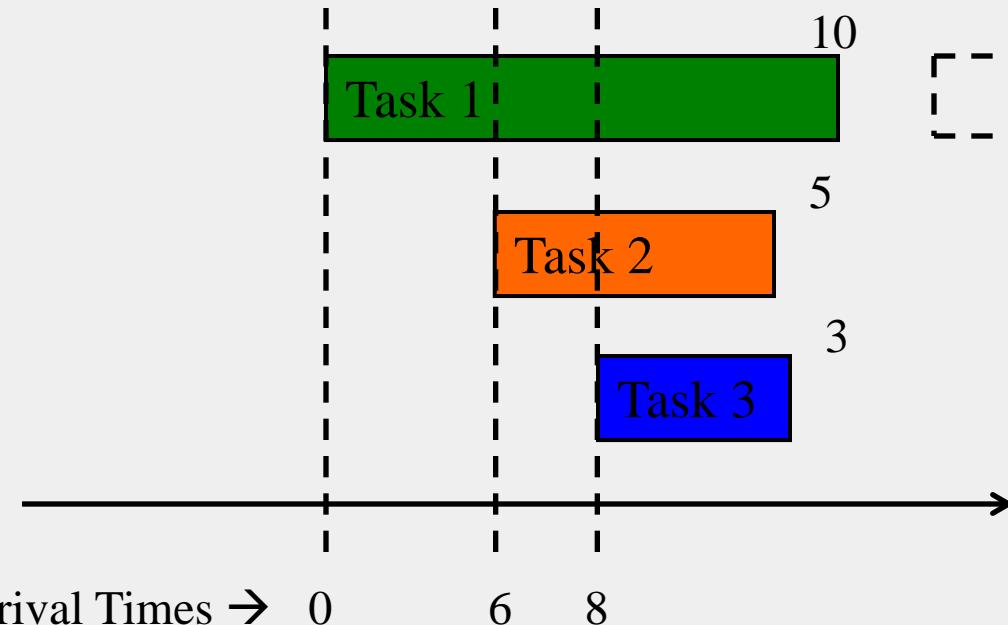
Lecture A

SINGLE-PROCESSOR SCHEDULING

WHY SCHEDULING?

- Multiple “tasks” to schedule
 - The processes on a single-core OS
 - The tasks of a Hadoop job
 - The tasks of multiple Hadoop jobs
- Limited resources that these tasks require
 - Processor(s)
 - Memory
 - (Less contentious) disk, network
- Scheduling goals
 1. Good throughput or response time for tasks (or jobs)
 2. High utilization of resources

SINGLE PROCESSOR SCHEDULING

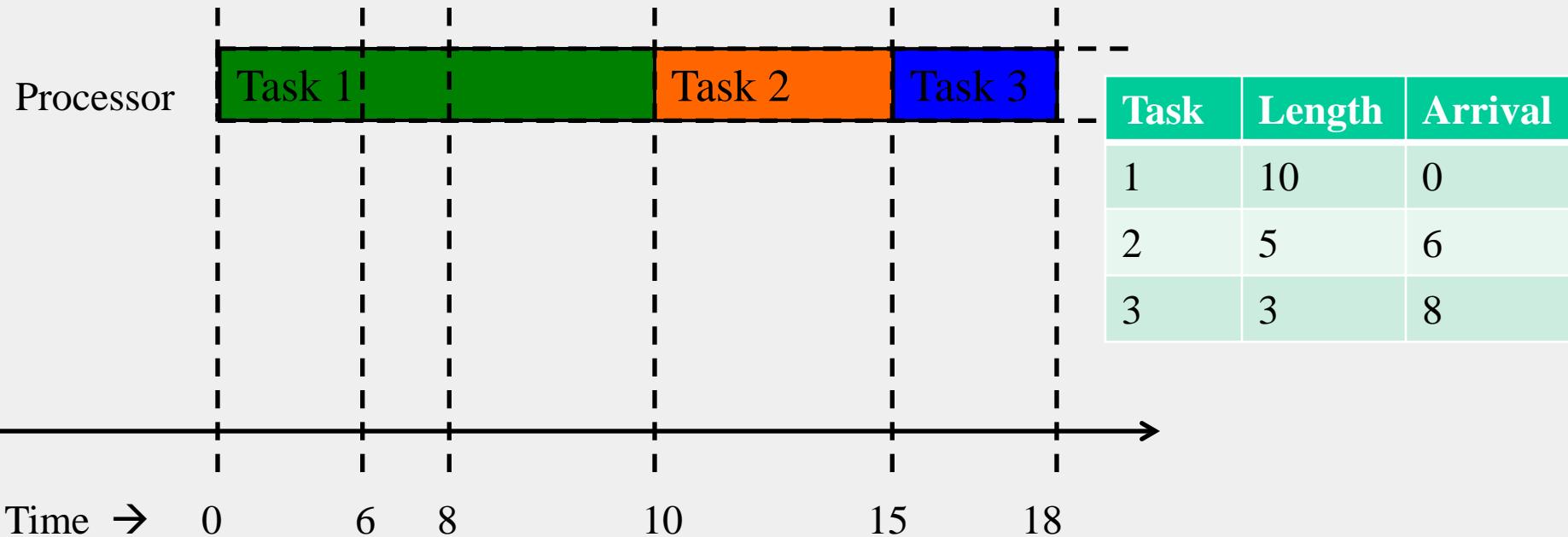


Which tasks run when?

Processor

Task	Length	Arrival
1	10	0
2	5	6
3	3	8

FIFO SCHEDULING (FIRST-IN FIRST-OUT)/FCFS

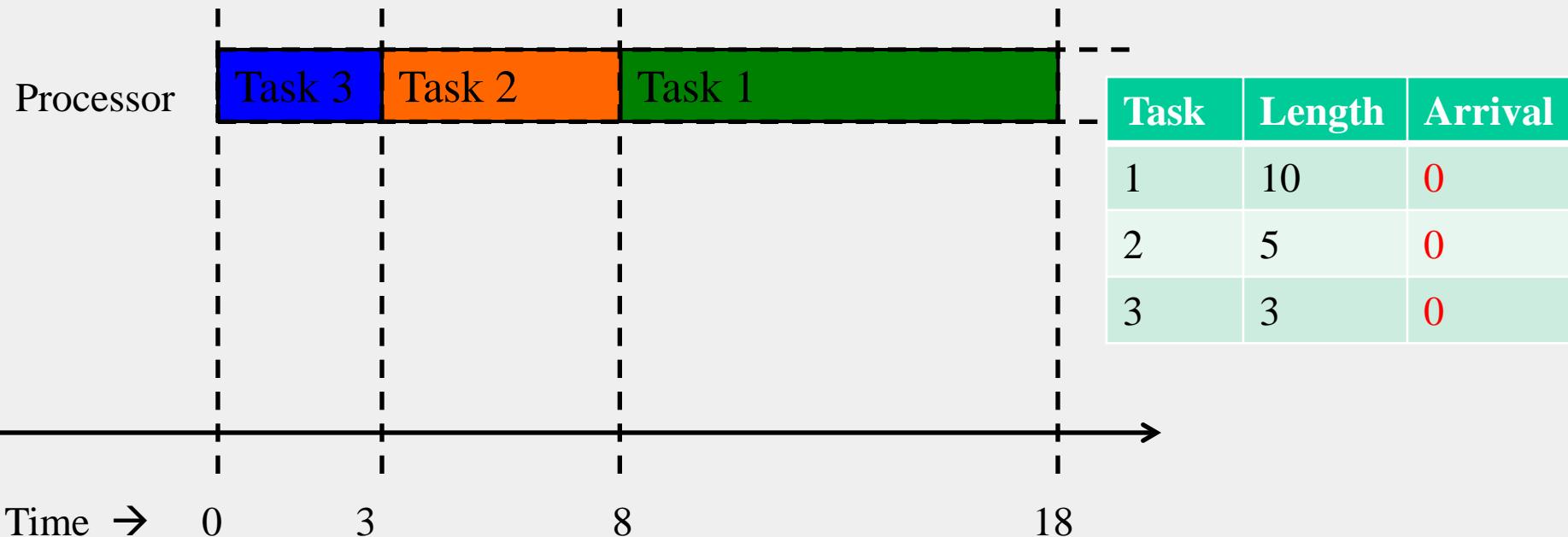


- *Maintain tasks in a queue in order of arrival*
- *When processor free, dequeue head and schedule it*

FIFO/FCFS PERFORMANCE

- Average completion time may be high
- For our example on previous slides,
 - Average completion time of FIFO/FCFS =
 $(\text{Task 1} + \text{Task 2} + \text{Task 3})/3$
= $(10+15+18)/3$
= $43/3$
= 14.33

STF SCHEDULING (SHORTEST TASK FIRST)

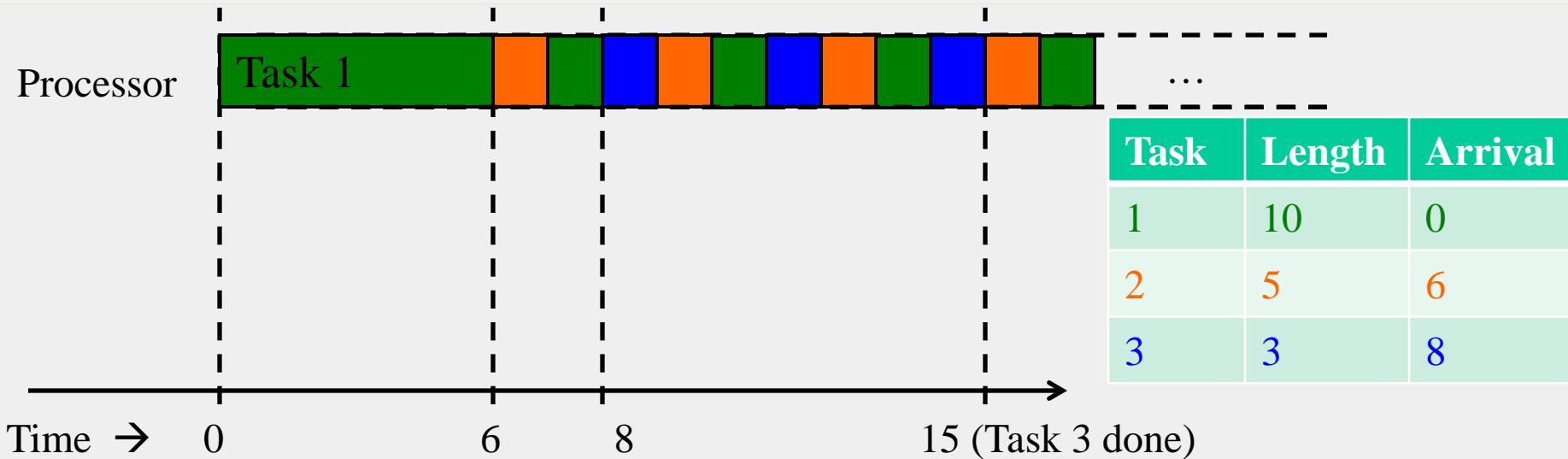


- *Maintain all tasks in a queue, in increasing order of running time*
- *When processor free, dequeue head and schedule*

STF Is OPTIMAL!

- Average completion of STF is the shortest among all scheduling approaches!
- For our example on previous slides,
 - Average completion time of STF =
$$\begin{aligned} & (\text{Task 1} + \text{Task 2} + \text{Task 3})/3 \\ &= (18+8+3)/3 \\ &= 29/3 \\ &= 9.66 \end{aligned}$$
(versus 14.33 for FIFO/FCFS)
- In general, STF is a special case of priority scheduling
 - Instead of using time as priority, scheduler could use user-provided priority

ROUND-ROBIN SCHEDULING



- Use a quantum (say 1 time unit) to run portion of task at queue head
- Pre-empts processes by saving their state, and resuming later
- After pre-empting, add to end of queue

ROUND-ROBIN VS. STF/FIFO

- Round-Robin preferable for
 - Interactive applications
 - User needs quick responses from system
- FIFO/STF preferable for Batch applications
 - User submits jobs, goes away, comes back to get result

SUMMARY

- Single processor scheduling algorithms
 - FIFO/FCFS
 - Shortest task first (optimal!)
 - Priority
 - Round-robin
 - Many other scheduling algorithms out there!
- What about cloud scheduling?
 - Next!



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SCHEDULING

Lecture B

HADOOP SCHEDULING

HADOOP SCHEDULING

- A Hadoop job consists of Map tasks and Reduce tasks
- Only one job in entire cluster => it occupies cluster
- Multiple customers with multiple jobs
 - Users/jobs = “tenants”
 - **Multi-tenant system**
- => Need a way to schedule all these jobs (and their constituent tasks)
- => Need to be *fair* across the different tenants
- Hadoop YARN has two popular schedulers
 - *Hadoop Capacity Scheduler*
 - *Hadoop Fair Scheduler*

HADOOP CAPACITY SCHEDULER

- Contains multiple queues
- Each queue contains multiple jobs
- Each queue guaranteed some portion of the cluster capacity
 - E.g.,
 - Queue 1 is given 80% of cluster
 - Queue 2 is given 20% of cluster
 - Higher-priority jobs go to Queue 1
- For jobs within same queue, FIFO typically used
- Administrators can configure queues

ELASTICITY IN HCS

- Administrators can configure each queue with limits
 - Soft limit: how much % of cluster is the queue guaranteed to occupy
 - (Optional) Hard limit: max % of cluster the queue is guaranteed
- Elasticity
 - A queue allowed to occupy more of cluster if resources free
 - But if other queues below their capacity limit, now get full, need to give these other queues resources
- Pre-emption not allowed!
 - Cannot stop a task part-way through
 - When reducing % cluster to a queue, wait until some tasks of that queue have finished

OTHER HCS FEATURES

- Queues can be hierarchical
 - May contain child sub-queues, which may contain child sub-queues, and so on
 - Child sub-queues can share resources equally
- Scheduling can take memory requirements into account
(memory specified by user)

HADOOP FAIR SCHEDULER

- Goal: all jobs get equal share of resources
- When only one job present, occupies entire cluster
- As other jobs arrive, each job given equal % of cluster
 - E.g., Each job might be given equal number of cluster-wide YARN containers
 - Each container == 1 task of job

HADOOP FAIR SCHEDULER (2)

- Divides cluster into pools
 - Typically one pool per user
- Resources divided equally among pools
 - Gives each user fair share of cluster
- Within each pool, can use either
 - Fair share scheduling, or
 - FIFO/FCFS
 - (Configurable)

PRE-EMPTION IN HFS

- Some pools may have *minimum shares*
 - Minimum % of cluster that pool is guaranteed
- When minimum share not met in a pool, for a while
 - Take resources away from other pools
 - By pre-empting jobs in those other pools
 - By *killing* the currently-running tasks of those jobs
 - Tasks can be re-started later
 - Ok since tasks are idempotent!
 - To kill, scheduler picks most-recently-started tasks
 - Minimizes wasted work

OTHER HFS FEATURES

- Can also set limits on
 - Number of concurrent jobs per user
 - Number of concurrent jobs per pool
 - Number of concurrent tasks per pool
- Prevents cluster from being hogged by one user/job

ESTIMATING TASK LENGTHS

- HCS/HFS use FIFO
 - May not be optimal (as we know!)
 - Why not use shortest-task-first instead? It's optimal (as we know!)
- Challenge: Hard to know expected running time of task (before it's completed)
- Solution: Estimate length of task
- Some approaches
 - Within a job: Calculate running time of task as proportional to size of its input
 - Across tasks: Calculate running time of task in a given job as average of other tasks in that given job (weighted by input size)
- Lots of recent research results in this area!

SUMMARY

- Hadoop Scheduling in YARN
 - Hadoop Capacity Scheduler
 - Hadoop Fair Scheduler
- Yet, so far we've talked of only one kind of resource
 - Either processor, or memory
 - How about multi-resource requirements?
 - Next!



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SCHEDULING

Lecture C

DOMINANT-RESOURCE
FAIR SCHEDULING

CHALLENGE

- What about scheduling VMs in a cloud (cluster)?
- Jobs may have multi-resource requirements
 - Job 1's tasks: 2 CPUs, 8 GB
 - Job 2's tasks: 6 CPUs, 2 GB
- How do you schedule these jobs in a “fair” manner?
- That is, how many tasks of each job do you allow the system to run concurrently?
- What does fairness even mean?

DOMINANT RESOURCE FAIRNESS (DRF)

- Proposed by researchers from U. California Berkeley
- Proposes notion of fairness across jobs with multi-resource requirements
- They showed that DRF is
 - Fair for multi-tenant systems
 - Strategy-proof: tenant can't benefit by lying
 - Envy-free: tenant can't envy another tenant's allocations

WHERE IS DRF USEFUL?

- DRF is
 - Usable in scheduling VMs in a cluster
 - Usable in scheduling Hadoop in a cluster
- DRF used in Mesos, an OS intended for cloud environments
- DRF-like strategies also used some cloud computing company's distributed OS's

How DRF WORKS

- Our example
 - Job 1's tasks: 2 CPUs, 8 GB
=> Job 1's resource vector = <2 CPUs, 8 GB>
 - Job 2's tasks: 6 CPUs, 2 GB
=> Job 2's resource vector = <6 CPUs, 2 GB>
- Consider a cloud with <18 CPUs, 36 GB RAM>

How DRF WORKS (2)

- Our example
 - Job 1's tasks: 2 CPUs, 8 GB
=> Job 1's resource vector = <2 CPUs, 8 GB>
 - Job 2's tasks: 6 CPUs, 2 GB
=> Job 2's resource vector = <6 CPUs, 2 GB>
- Consider a cloud with <18 CPUs, 36 GB RAM>
- Each Job 1's task consumes % of total **CPUs** = $2/18 = \mathbf{1/9}$
- Each Job 1's task consumes % of total **RAM** = $8/36 = \mathbf{2/9}$
- $\mathbf{1/9} < \mathbf{2/9}$
 - => Job 1's dominant resource is RAM, i.e., Job 1 is more memory-intensive than it is CPU-intensive

How DRF WORKS (3)

- Our example
 - Job 1's tasks: 2 CPUs, 8 GB
=> Job 1's resource vector = <2 CPUs, 8 GB>
 - Job 2's tasks: 6 CPUs, 2 GB
=> Job 2's resource vector = <6 CPUs, 2 GB>
- Consider a cloud with <18 CPUs, 36 GB RAM>
- Each Job 2's task consumes % of total CPUs = $6/18 = 6/18$
- Each Job 2's task consumes % of total RAM = $2/36 = 1/18$
- $6/18 > 1/18$
 - => Job 2's dominant resource is CPU, i.e., Job 1 is more CPU-intensive than it is memory-intensive

DRF FAIRNESS

- For a given job, the % of its dominant resource type that it gets cluster-wide, is the same for all jobs
 - Job 1's % of RAM = Job 2's % of CPU
- Can be written as linear equations, and solved

DRF SOLUTION, FOR OUR EXAMPLE

- DRF Ensures
 - Job 1's % of RAM = Job 2's % of CPU
- Solution for our example:
 - Job 1 gets 3 tasks each with <2 CPUs, 8 GB>
 - Job 2 gets 2 tasks each with <6 CPUs, 2 GB>
 - Job 1's % of RAM
= Number of tasks * RAM per task / Total cluster RAM
= $3 \times 8 / 36 = 2/3$
 - Job 2's % of CPU
= Number of tasks * CPU per task / Total cluster CPUs
= $2 \times 6 / 18 = 2/3$

OTHER DRF DETAILS

- DRF generalizes to multiple jobs
- DRF also generalizes to more than 2 resource types
 - CPU, RAM, Network, Disk, etc.
- DRF ensures that each job gets a fair share of that type of resource which the job desires the most
 - Hence fairness

SUMMARY: SCHEDULING

- Scheduling very important problem in cloud computing
 - Limited resources, lots of jobs requiring access to these jobs
- Single-processor scheduling
 - FIFO/FCFS, STF, Priority, Round-Robin
- Hadoop scheduling
 - Capacity scheduler, Fair scheduler
- Dominant-Resources Fairness



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

STREAM PROCESSING

Lecture A

STREAM PROCESSING IN STORM

WHAT WE'LL COVER

- Why Stream Processing
- Storm

STREAM PROCESSING CHALLENGE

- Large amounts of data => Need for real-time views of data
 - Social network trends, e.g., Twitter real-time search
 - Website statistics, e.g., Google Analytics
 - Intrusion detection systems, e.g., in most datacenters
- Process large amounts of data
 - With latencies of few seconds
 - With high throughput

MAPREDUCE?

- Batch Processing => Need to wait for entire computation on large dataset to complete
- Not intended for long-running stream-processing

ENTER STORM

- Apache Project
- <https://storm.incubator.apache.org/>
- Highly active JVM project
- Multiple languages supported via API
 - Python, Ruby, etc.
- Used by over 30 companies including
 - Twitter: For personalization, search
 - Flipboard: For generating custom feeds
 - Weather Channel, WebMD, etc.

STORM COMPONENTS

- Tuples
- Streams
- Spouts
- Bolts
- Topologies

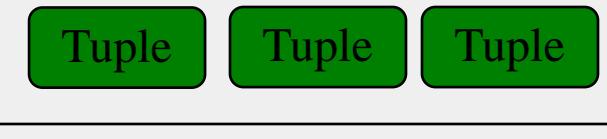
TUPLE

- An ordered list of elements
- E.g., <tweeter, tweet>
 - E.g., <“Miley Cyrus”, “Hey! Here’s my new song!”>
 - E.g., <“Justin Bieber”, “Hey! Here’s MY new song!”>
- E.g., <URL, clicker-IP, date, time>
 - E.g., <coursera.org, 101.201.301.401, 4/4/2014, 10:35:40>
 - E.g., <coursera.org, 901.801.701.601, 4/4/2014, 10:35:42>

Tuple

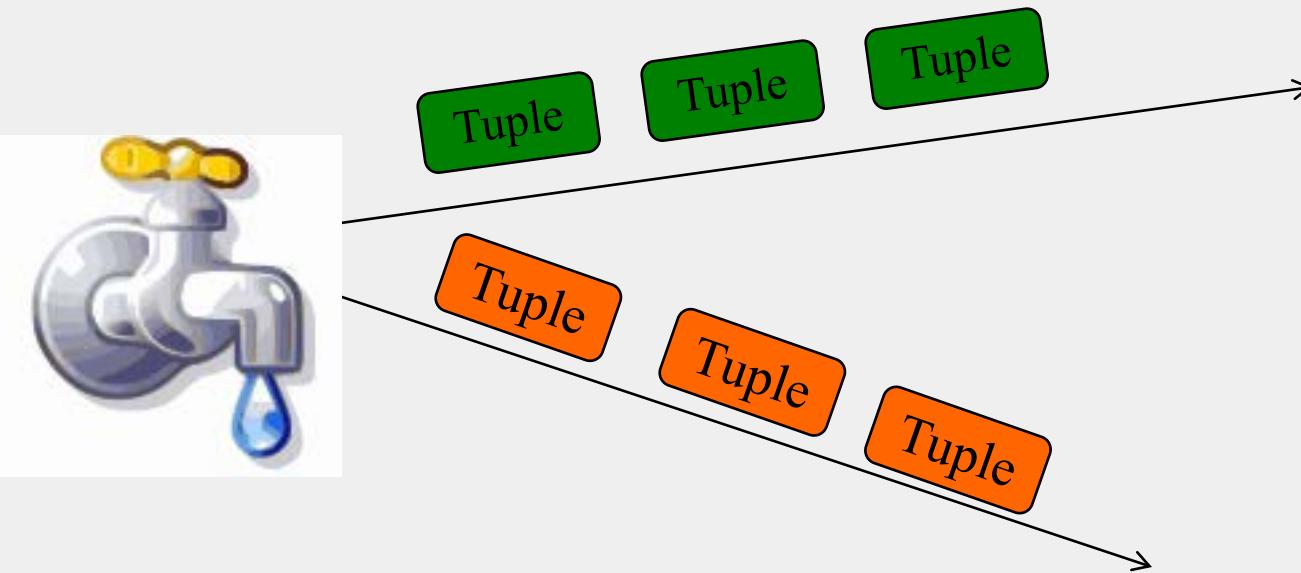
STREAM

- Sequence of tuples
 - Potentially unbounded in number of tuples
- Social network example:
 - <“Miley Cyrus”, “Hey! Here’s my new song!”>, <“Justin Bieber”, “Hey! Here’s MY new song!”>, <“Rolling Stones”, “Hey! Here’s my old song that’s still a super-hit!”>, ...
- Website example:
 - <coursera.org, 101.201.301.401, 4/4/2014, 10:35:40>, <coursera.org, 901.801.701.601, 4/4/2014, 10:35:42>, ...



SPOUT

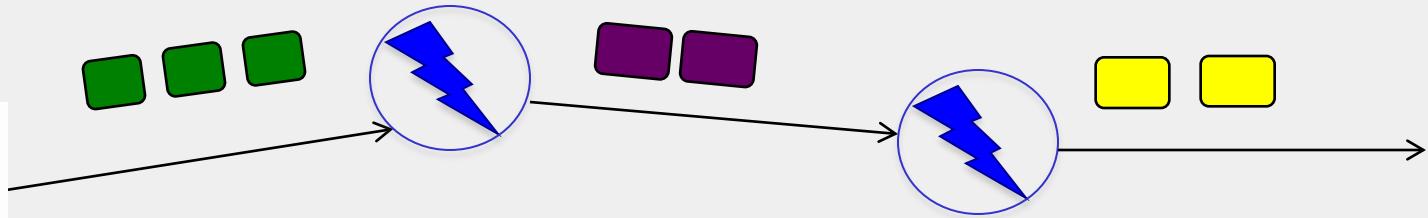
- A Storm entity (process) that is a source of streams
- Often reads from a crawler or DB



BOLT

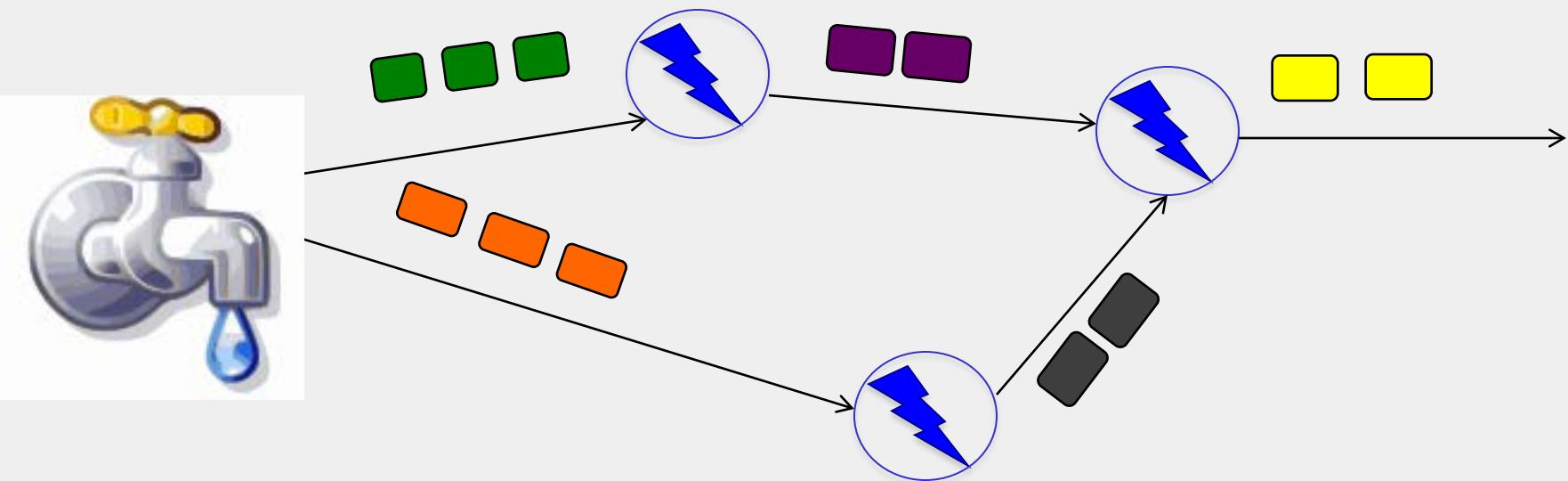


- A Storm entity (process) that
 - Processes input streams
 - Outputs more streams for other bolts



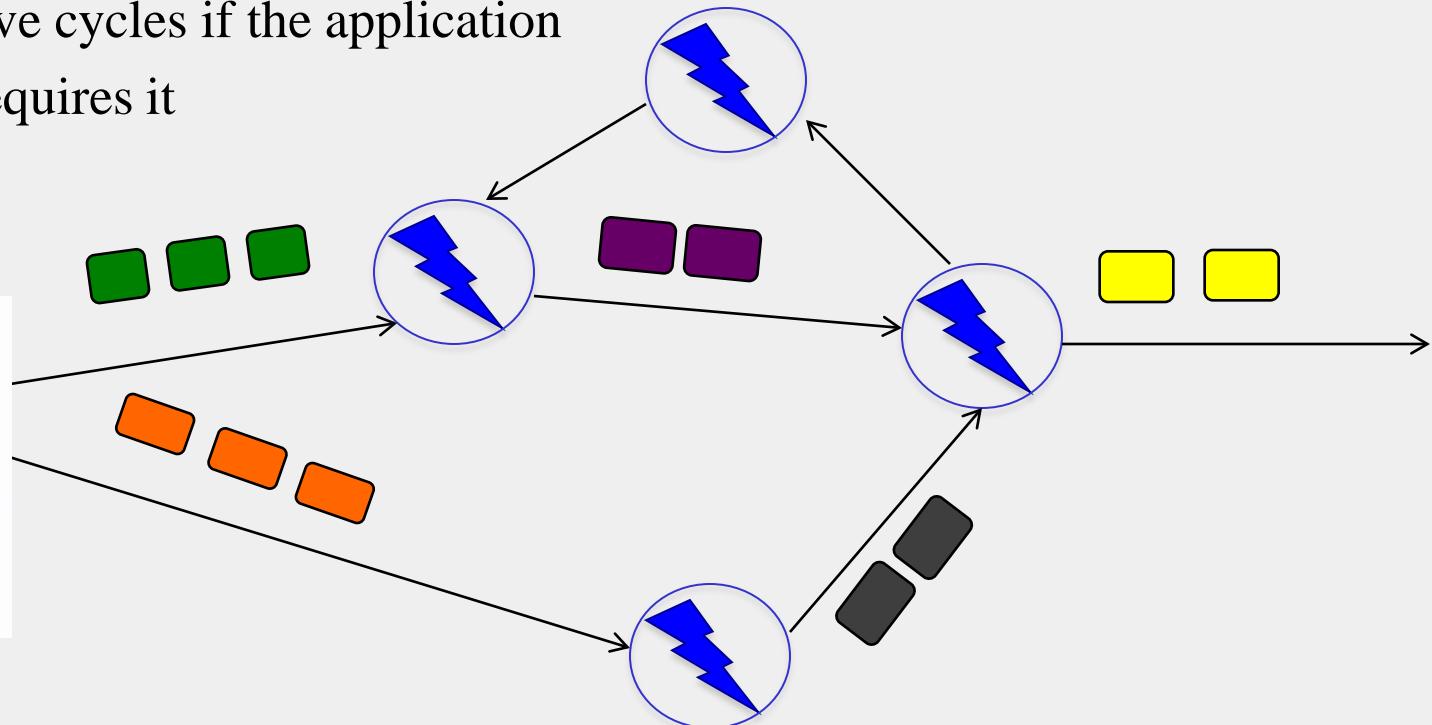
TOPOLOGY

- A directed graph of spouts and bolts (and output bolts)
- Corresponds to a Storm “application”



TOPOLOGY

- Can have cycles if the application requires it



BOLTS COME IN MANY FLAVORS

- Operations that can be performed
 - **Filter**: forward only tuples which satisfy a condition
 - **Joins**: When receiving two streams A and B, output all pairs (A,B) which satisfy a condition
 - **Apply/transform**: Modify each tuple according to a function
 - And many others
- But bolts need to process a lot of data
 - Need to make them fast

PARALLELIZING BOLTS

- Have multiple processes (“tasks”) constitute a bolt
- Incoming streams split among the tasks
- Typically each incoming tuple goes to one task in the bolt
 - Decided by “**Grouping strategy**”
- Three types of grouping are popular

GROUPING

- **Shuffle Grouping**
 - Streams are distributed evenly across the bolt's tasks
 - Round-robin fashion
- **Fields Grouping**
 - Group a stream by a subset of its fields
 - E.g., All tweets where twitter username starts with [A-M,a-m,0-4] goes to task 1, and all tweets starting with [N-Z,n-z,5-9] go to task 2
- **All Grouping**
 - All tasks of bolt receive all input tuples
 - Useful for joins



STORM CLUSTER

- Master node
 - Runs a daemon called *Nimbus*
 - Responsible for
 - Distributing code around cluster
 - Assigning tasks to machines
 - Monitoring for failures of machines
- Worker node
 - Runs on a machine (server)
 - Runs a daemon called *Supervisor*
 - Listens for work assigned to its machines
- Zookeeper
 - Coordinates Nimbus and Supervisors communication
 - All state of Supervisor and Nimbus is kept here

FAILURES

- A tuple is considered failed when its topology (graph) of resulting tuples fails to be fully processed within a specified timeout
- **Anchoring:** Anchor an output to one or more input tuples
 - Failure of one tuple causes one or more tuples to replayed

API For FAULT-TOLERANCE (OUTPUTCOLLECTOR)

- **Emit(tuple, output)**
 - Emits an output tuple, perhaps anchored on an input tuple (first argument)
- **Ack(tuple)**
 - Acknowledge that you finish processing a tuple
- **Fail(tuple)**
 - Immediately fail the spout tuple at the root of tuple topology if there is an exception from the database, etc.
- Must remember to ack/fail each tuple
 - Each tuple consumes memory. Failure to do so results in memory leaks.

SUMMARY

- Processing data in real-time a big requirement today
- Storm
 - And other sister systems, e.g., Spark Streaming
- Parallelism
- Application topologies
- Fault-tolerance



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

STRUCTURE OF NETWORKS

Lecture A

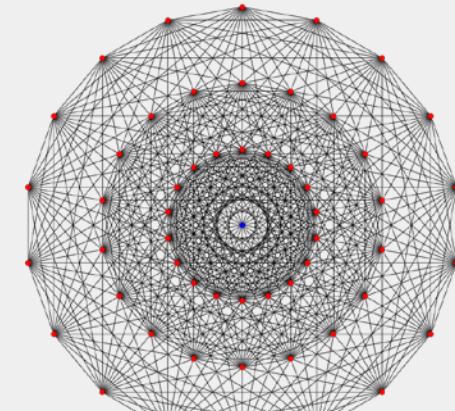
STRUCTURE OF NETWORK

WHAT'S A NETWORK/GRAFH?

- Has **vertices** (i.e., nodes)
 - E.g., in the Facebook graph, each user = a vertex (or a node)
- Has **edges** that connect pairs of vertices
 - E.g., in the Facebook graph, a friend relationship = an edge

LOTS OF GRAPHS/NETWORKS

- Large graphs/network are all around us
 - Internet : vertices are routers/switches and edges are links
 - World Wide Web: vertices are webpages, and edges are URL links on a webpage pointing to another webpage
 - Called “Directed” graph as edges are uni-directional
 - Social networks: Facebook, Twitter, LinkedIn
 - Biological networks: DNA interaction graphs, ecosystem graphs, etc.



Source: Wikimedia Commons

COMPLEXITY OF NETWORKS

- **Structural:** *human population has ~7 B nodes, there are millions of computers on the Internet...*
- **Evolution:** *people make new friends all the time, ISP's change hands all the time...*
- **Diversity:** *some people are more popular, some friendships are more important...*
- **Node Complexity:** *Endpoints have different CPUs, Windows is a complicated OS, Mobile devices ...*
- **Emergent phenomena:** simple end behavior → leads to → complex system-wide behavior.
 - *If we understand the basics of climate change, why is the weather so unpredictable?*

NETWORK STRUCTURE

- “Six degrees of Kevin Bacon”
- Milgram’s experiment in 1970
- Recent work shows similarities between the structures of: Internet, WWW, human social networks, p2p overlays, Electric power grid, protein networks
- These networks have “evolved naturally”
- Many of these are “**small world networks**”

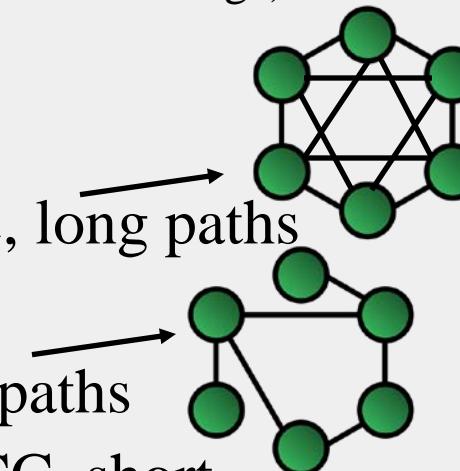
Two IMPORTANT NETWORK PROPERTIES

1. Clustering Coefficient: CC

$\Pr(\text{A-B edge, given an A-C edge and a C-B edge})$

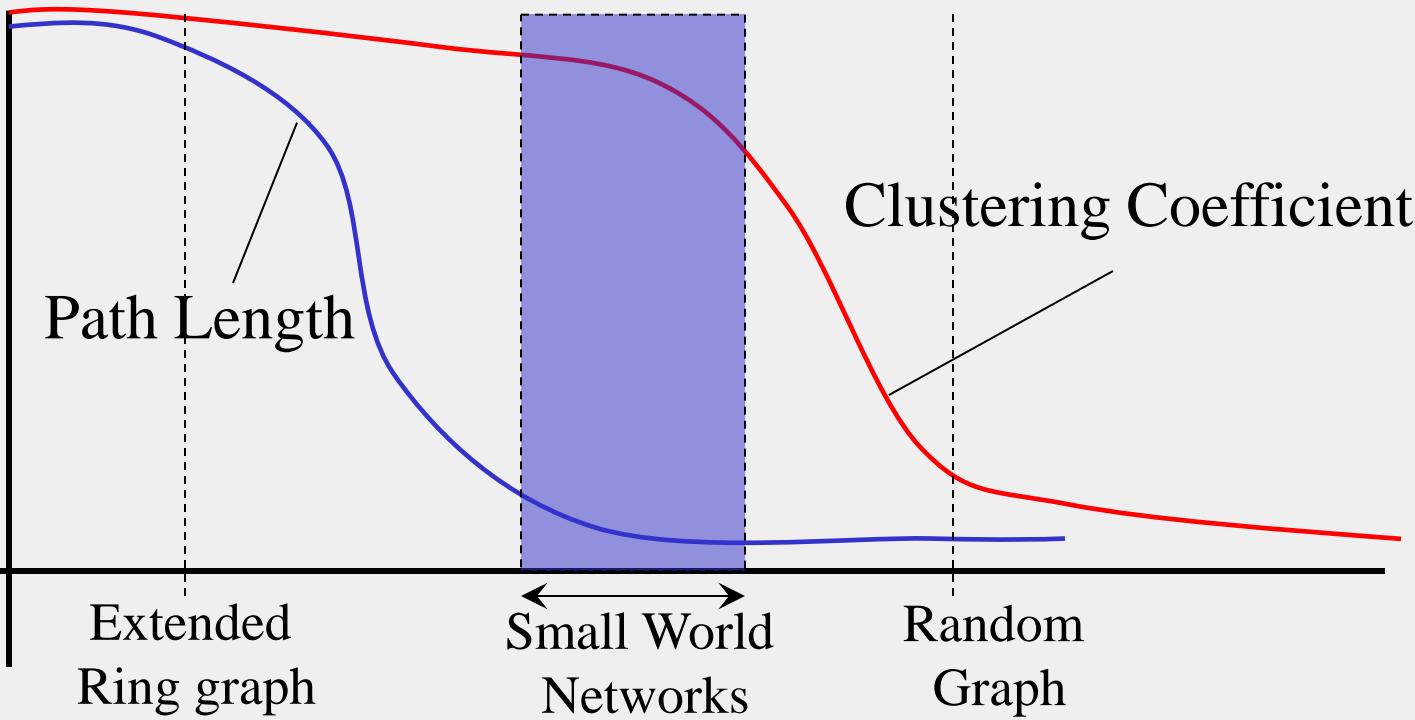
2. Path Length of shortest path

- Extended Ring graph: high CC, long paths
- Random graph: low CC, short paths
- Small World Networks: high CC, short paths



DERIVING SMALL-WORLD GRAPHS

Convert more and more edges to point to random nodes →



SMALL-WORLD NETWORKS ALL AROUND

Most “natural evolved” networks are small world

- Network of actors → six degrees of Kevin Bacon
- Network of humans → Milgram’s experiment
- Co-authorship network → “Erdos Number”
- World Wide Web, the Internet, ...

Many of these networks also “grow incrementally”

“Preferential” model of growth

- When adding a vertex to graph, connect it to existing vertex v with probability proportional to $\text{num_neighbors}(v)$

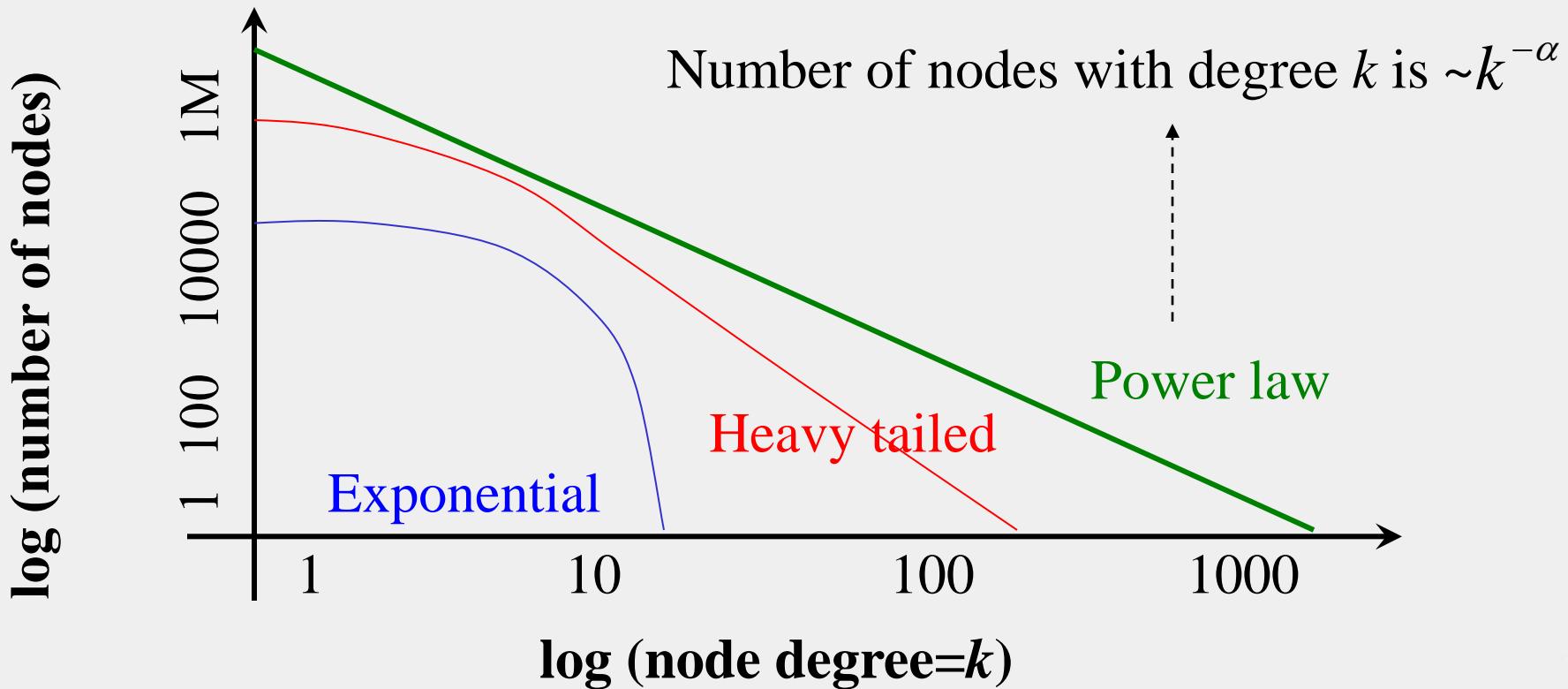
DEGREES

Degree of a vertex = number of its immediate neighbor vertices

Degree distribution – what is the probability of a given node having k edges (neighbors, friends, ...)

- Regular graph: all nodes same degree
- **Gaussian**
- Random graph: **Exponential** $e^{-k.c}$
- **Power law:** $k^{-\alpha}$

POWER LAW GRAPHS



SMALL-WORLD AND POWER-LAW

- A lot of small world networks are power law graphs
 - Internet backbone, telephone call graph, protein networks
 - WWW is a small-world graph and also a power-law graph with $\alpha=2.1-2.4$
 - Gnutella p2p system network has heavy-tailed degree distribution
- Power law networks also called *scale-free*
 - Gnutella has 3.4 edges per vertex, *independent of scale* (*i.e.*, *number of vertices*)

SMALL-WORLD \neq POWER-LAW

- Not all small world networks are power law
 - E.g., co-author networks
- Not all power-law networks are small world
 - E.g., Disconnected power-law networks

RESILIENCE OF SMALL-WORLD+POWER-LAW

Most nodes have small degree, but a few nodes have high degree

Attacks on small world networks

- Killing a large number of randomly chosen nodes does not disconnect graph
- Killing a few high-degree nodes will disconnect graph

“A few (of the many thousand) nutrients are very important to your body”

“The Electric Grid is very vulnerable to attacks”

ROUTING IN SMALL-WORLD/POWER-LAW NETWORKS

- Build shortest-path routes between every pair of vertices
- => Most of these routes will pass via the few high-degree vertices in the graphs
 - => High-degree vertices are heavily overloaded
 - High-degree vertices more likely to suffer congestions or crash
- Same phenomenon in Electric power grid
- Solution may be to introduce some randomness in path selection; don't always use shortest path

SUMMARY

- Networks (graphs) are all around us
 - Man-made networks like Internet, WWW, p2p
 - Natural networks like protein networks, human social network
- Yet, many of these have common characteristics
 - Small-world
 - Power-law
- Useful to know this: when designing distributed systems that run on such networks
 - Can better predict how these networks might behave



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

DISTRIBUTED FILE SYSTEMS

Lecture A

FILE SYSTEM ABSTRACTION

FILE SYSTEM

- Contains files and directories (folders)
- Higher level of abstraction
 - Prevents users and processes from dealing with disk blocks and memory blocks

FILE CONTENTS

- Typical File



- Timestamps: creation, read, write, header
- File type, e.g., .c, .java
- Ownership, e.g., edison
- Access Control List: who can access this file and in what mode
- Reference Count: Number of directories containing this file
 - May be > 1 (hard linking of files)
 - When 0, can delete file

WHAT ABOUT DIRECTORIES?

- They're just files!
- With their “data” containing
 - The meta-information about files the directory contains
 - Pointers (on disk) to those files

UNIX FILE SYSTEM: OPENING AND CLOSING FILES

- Uses notion of *file descriptors*
 - Handle for a process to access a file
- Each process: Needs to open a file before reading/writing file
 - OS creates an internal datastructure for a file descriptor, returns handle
- *filedes=open(name, mode)*
 - mode = access mode, e.g., r, w, x
- *filedes=creat(name, mode)*
 - Create the file, return the file descriptor
- *close(filedes)*

UNIX FILE SYSTEM: READING AND WRITING

- *status=read(filedes, buffer, num_bytes)*
 - File descriptor maintains a *read-write pointer* pointing to an offset within file
 - *read()* reads *num_bytes* starting from that pointer (into buffer), and *automatically advances pointer by num_bytes*
- *status=write(filedes, buffer, num_bytes)*
 - Writes from buffer into file at position pointer
 - Automatically advances pointer by *num_bytes*
- *pos=lseek(filedes, offset, whence)*
 - Moves read-write pointer to position offset within file
 - *whence* says whether offset absolute or relative (relative to current pointer)

UNIX FILE SYSTEM: CONTROL OPERATIONS

- *status=link(old_link, new_link)*
 - Creates a new link at second arg to the file at first arg
 - Old_link and new_link are Unix-style names, e.g.,
“/usr/edison/my_invention”
 - Increments reference count of file
 - Known as a “hard link”
 - Vs. “Symbolic/Soft linking” which creates another file pointing to this file;
does not change reference count
- *status=unlink(old_link)*
 - Decrements reference count
 - If count=0, can delete file
- *status=stat/fstat(file_name, buffer)*
 - Get attributes (header) of file into *buffer*

DISTRIBUTED FILE SYSTEMS (DFS)

- Files are stored on a server machine
 - Client machine does RPCs to server to perform operations on file

Desirable Properties from a DFS

- Transparency: client accesses DFS files as if it were accessing local (say, Unix) files
 - Same API as local files, i.e., client code doesn't change
 - Need to make location, replication, etc. invisible to client
- Support concurrent clients
 - Multiple client processes reading/writing the file concurrently
- Replication: for fault-tolerance

CONCURRENT ACCESSES IN DFS

- One-copy update semantics: when file is replicated, its contents, as visible to clients, are no different from when the file has exactly 1 replica
- At most once operation vs. At least once operation
 - Choose carefully
 - At most once, e.g., append operations cannot be repeated
 - *Idempotent* operations have no side effects when repeated: they can use at least once semantics, e.g., read at absolute position in file



SECURITY IN DFS

- Authentication
 - Verify that a given user is who they claim to be
- Authorization
 - After a user is authenticated, verify that the file they're trying to access
 - Two popular flavors
 - **Access Control Lists (ACLs)** = per file, list of allowed users and access allowed to each
 - **Capability Lists** = per user, list of files allowed to access and type of access allowed
 - Could split it up into capabilities, each for a different (user,file)

LET'S BUILD A DFS!

- We'll call it our "Vanilla DFS"
- Vanilla DFS runs on a server, and at multiple clients
- Vanilla DFS consists of three types of processes
 - Flat file service: at server
 - Directory service: at server, talks to (i.e., "client of") Flat file service
 - Client service: at client, talks to Directory service and Flat file service

VANILLA DFS: FLAT FILE SERVICE API

- **Read**(*file_id, buffer, position, num_bytes*)
 - Reads *num_bytes* from absolute *position* in file *file_id* into *buffer*
 - *File_id* is not a file descriptor, it's a unique id of that file
 - No automatic read-write pointer!
 - Why not? Need operation to be *idempotent* (at least once semantics)
 - No file descriptors!
 - Why not? Need servers to be *stateless*: easier to recover after failures (no state to restore!)
 - In contrast, Unix file system operations are neither idempotent nor stateless

VANILLA DFS: FLAT FILE SERVICE API (2)

- `write(file_id, buffer, position, num_bytes)`
 - Similar to read
- `create/delete(file_id)`
- `get_attributes/set_attributes(file_id, buffer)`

VANILLA DFS: DIRECTORY SERVICE API

- $\text{file_id} = \text{lookup}(dir, file_name)$
 - file_id can then be used to access file via Flat file service
- $\text{add_name}(dir, file_name)$
 - Increments reference count
- $\text{un_name}(dir, file_name)$
 - Decrements reference count; if =0, can delete
- $list=\text{get_names}(dir, pattern)$
 - Like ls –al or dir, followed by grep or find

CAN WE BUILD A REAL DFS ALREADY?

- Next: Two popular distributed file systems
 - NFS and AFS



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

DISTRIBUTED FILE SYSTEMS

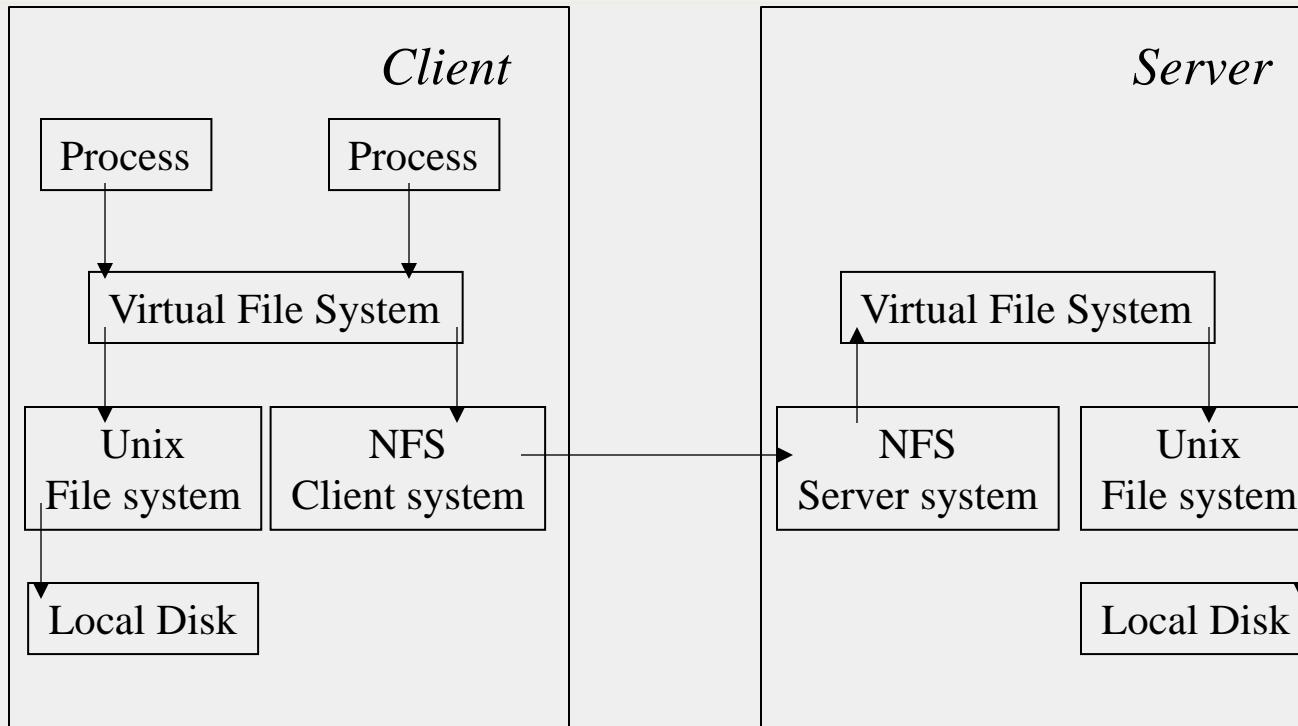
Lecture B

NFS AND AFS

NFS

- Network File System
- Sun Microsystems, 1980s
- Used widely even today

NFS ARCHITECTURE



NFS CLIENT AND SERVER SYSTEMS

- NFS Client system
 - Similar to our “Client service” in our Vanilla DFS
 - Integrated with kernel (OS)
 - Performs RPCs to NFS Server system for DFS operations
- NFS Server system
 - Plays the role of both Flat file service + Directory service from our Vanilla DFS
 - Allows *mounting* of files and directories
 - Mount /usr/edison/inventions into /usr/tesla/my_competitors
 - => Now, /usr/tesla/my_competitors/foo refers to /usr/edison/inventions/foo
 - Mount: Doesn’t clone (copy) files, just point to that directory now

VIRTUAL FILE SYSTEM MODULE

- Allows processes to access files via file descriptors
 - Just like local Unix files! So, local and remote files are indistinguishable (i.e., gives transparency)
 - For a given file access, decides whether to route to local file system or to NFS client system
- Names all files (local or remote) uniquely using “NFS file handles”
- Keeps a data structure for each mounted file system
- Keeps a data structure called **v-node** for all open files
 - If local file, v-node points to local disk i-node
 - If remote, v-node contains address of remote NFS server

SERVER OPTIMIZATIONS

- **Server caching** is one of the big reasons NFS is so fast with reads
 - Server Caching = Store, in memory, some of the recently-accessed blocks (of files and directories)
 - Most programs (written by humans) tend to have *locality of access*
 - Blocks accessed recently will be accessed soon in the future
- Writes: two flavors
 - **Delayed write**: write in memory, flush to disk every 30 s (e.g., via Unix sync operation)
 - Fast but not consistent
 - **Write-through**: Write to disk immediately before ack-ing client
 - Consistent but may be slow

CLIENT CACHING

- Client also caches recently-accessed blocks
- Each block in cache is tagged with
 - Tc : the time when the cache entry was last validated.
 - Tm : the time when the block was last modified at the server.
 - A cache entry at time T is valid if
$$(T-Tc < t) \text{ or } (Tm_{client} = Tm_{server}).$$
 - $t=freshness\ interval$
 - Compromise between consistency and efficiency
 - Sun Solaris: t is set adaptively between 3-30 s for files, 30-60 s for directories
- When block is written, do a delayed-write to server



ANDREW FILE SYSTEM (AFS)

- Designed at CMU
 - Named after Andrew Carnegie and Andrew Mellon, the “C” and “M” in CMU
- In use today in some clusters (especially University clusters)

INTERESTING DESIGN DECISIONS IN AFS

- Two unusual design principles:
 - Whole file serving
 - Not in blocks
 - Whole file caching
 - Permanent cache, survives reboots
- Based on (validated) assumptions that
 - Most file accesses are by a single user
 - Most files are small
 - Even a client cache as “large” as 100MB is supportable (e.g., in RAM)
 - File reads are much more frequent than file writes, and typically sequential

AFS DETAILS

- Clients system = *Venus* service
- Server system = *Vice* service
- Reads and writes are **optimistic**
 - Done on local copy of file at client (*Venus*)
 - When file closed, writes propagated to *Vice*
- When a client (*Venus*) opens a file, *Vice*:
 - Sends it entire file
 - Gives client a *callback promise*
- Callback promise
 - Promise that if another client modifies then closes the file, a callback will be sent from *Vice* to *Venus*
 - Callback state at *Venus* only binary: valid or canceled



SUMMARY

- Distributed File Systems
 - Widely used today
- Vanilla DFS
- NFS
- AFS
- Many other file systems out there today!



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

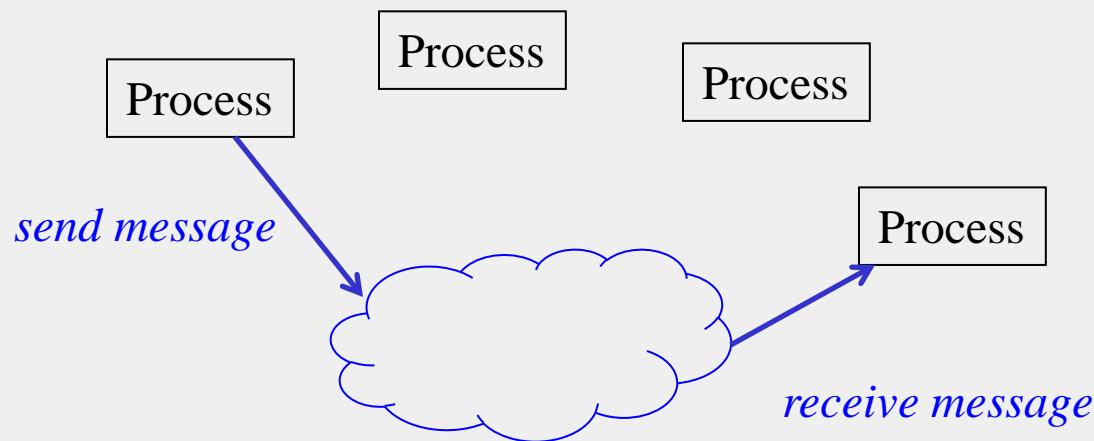
DISTRIBUTED SHARED MEMORY

Lecture A

DISTRIBUTED SHARED MEMORY

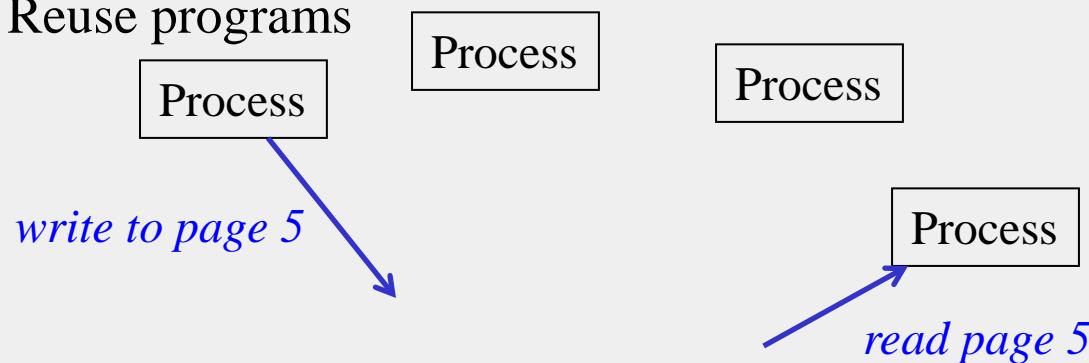
So FAR ...

- Message passing network



BUT WHAT IF ...

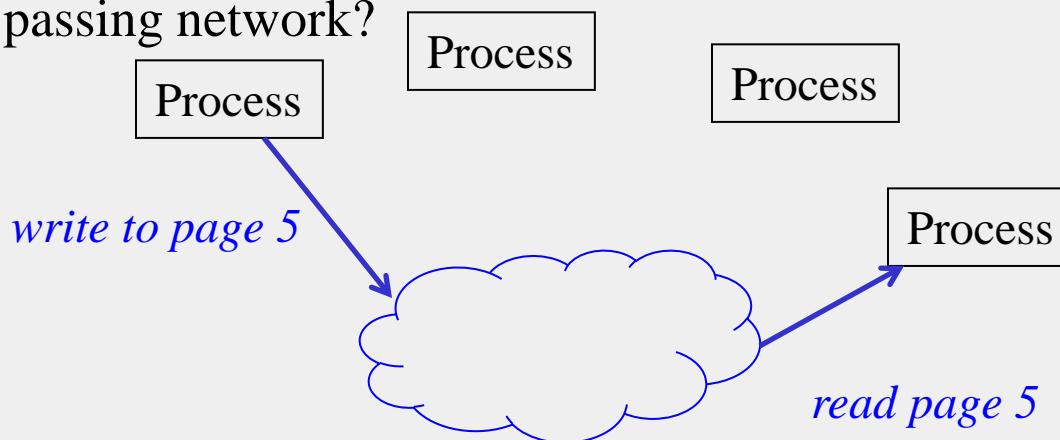
- Processes could *share* memory pages instead?
- Makes it convenient to write programs
- Reuse programs



Page 0	Page 1	Page 2	...	Page N-1
--------	--------	--------	-----	----------

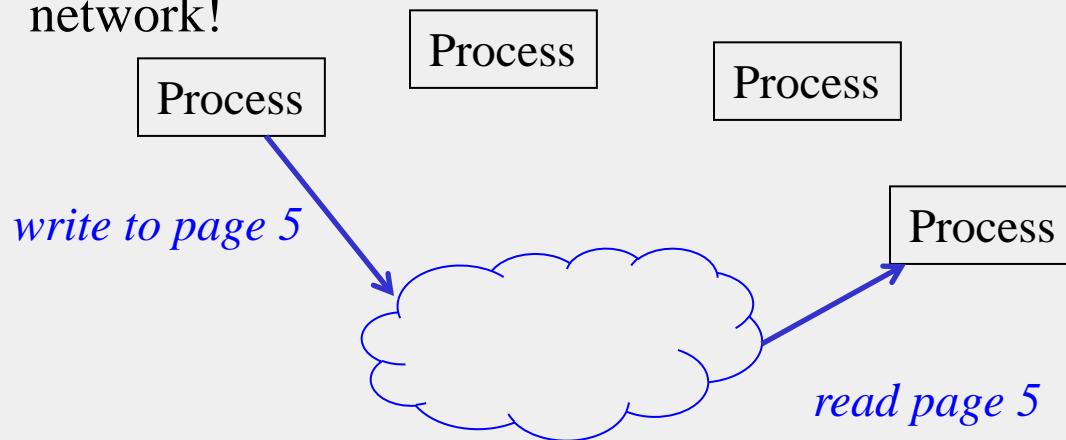
DISTRIBUTED SHARED MEMORY

- Distributed Shared Memory = processes virtually share pages
- How do you implement DSM over a message-passing network?



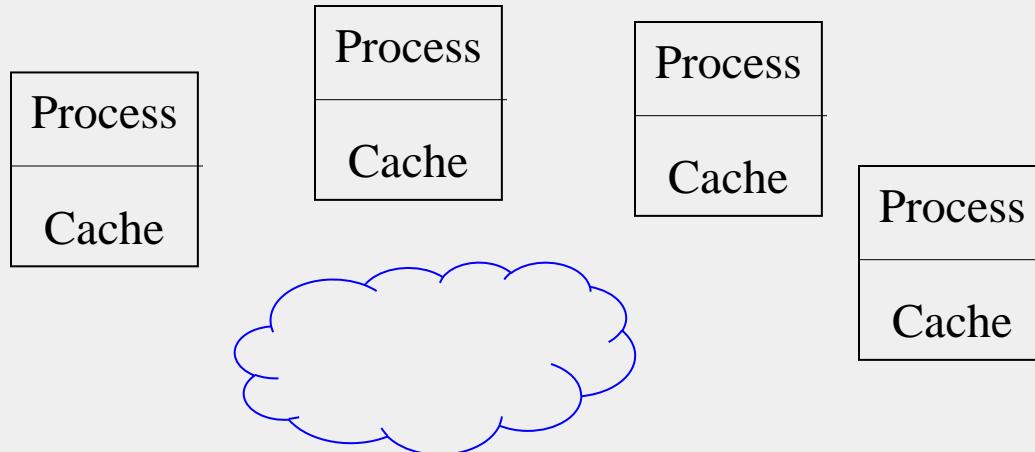
IN FACT ...

1. Message-passing can be implemented over DSM!
 - Use a common page as buffer to read/write messages
2. DSM can be implemented over a message-passing network!



DSM OVER MESSAGE-PASSING NETWORK

- *Cache* maintained at each process
 - Cache stores pages accessed recently by that process
- Read/write first goes to cache



DSM OVER MESSAGE-PASSING NETWORK (2)

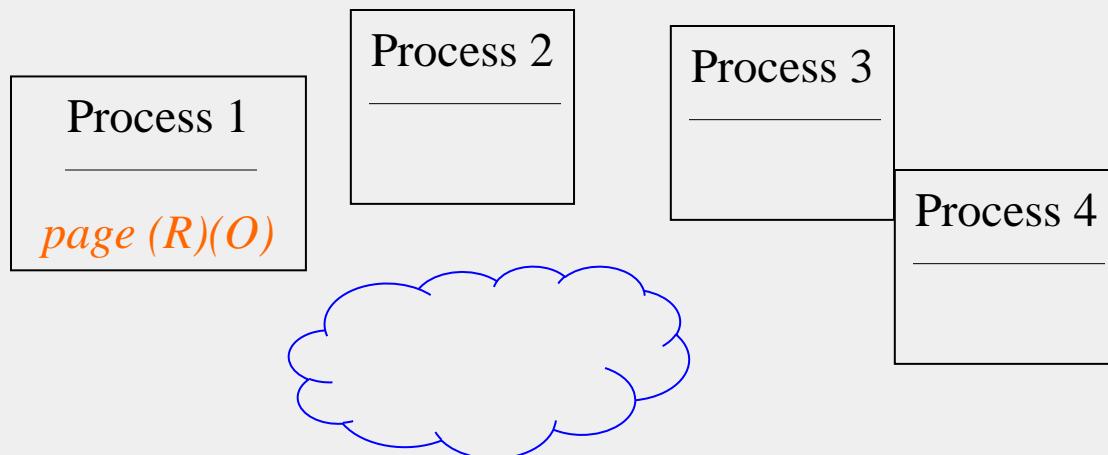
- Pages can be mapped in local memory
- When page is present in memory, page hit
- Otherwise, *page fault* (kernel trap) occurs
 - Kernel trap handler: invokes the DSM software
 - May contact other processes in DSM group, via multicast

DSM: INVALIDATE PROTOCOL

- Owner = Process with latest version of page
- Each page is in either R or W state
- When page in R state, owner has an R copy, but other processes may also have R copies
 - but no W copies exist
- When page is in W state, only owner has a copy

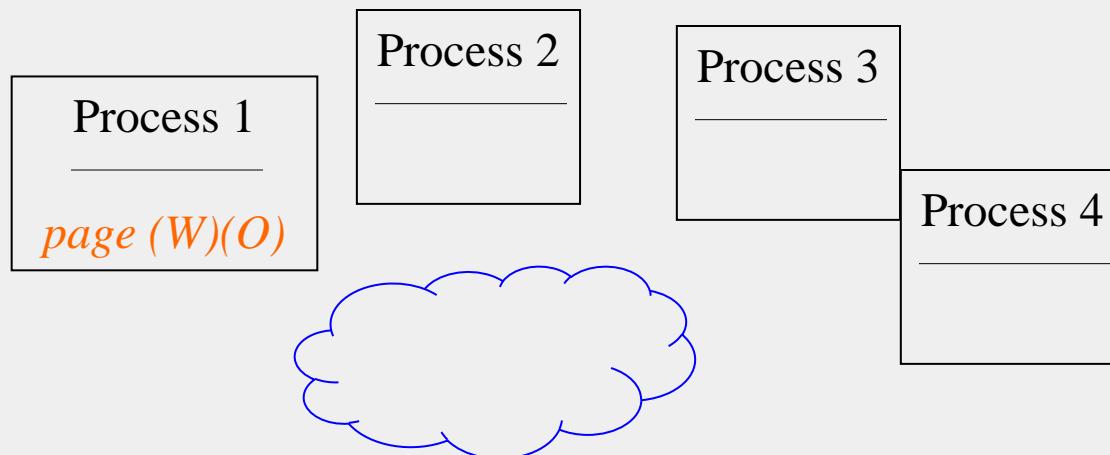
PROCESS 1 ATTEMPTING A READ: SCENARIO 1

- Process 1 is owner (O) and has page in R state
- *Read from cache. No messages sent.*



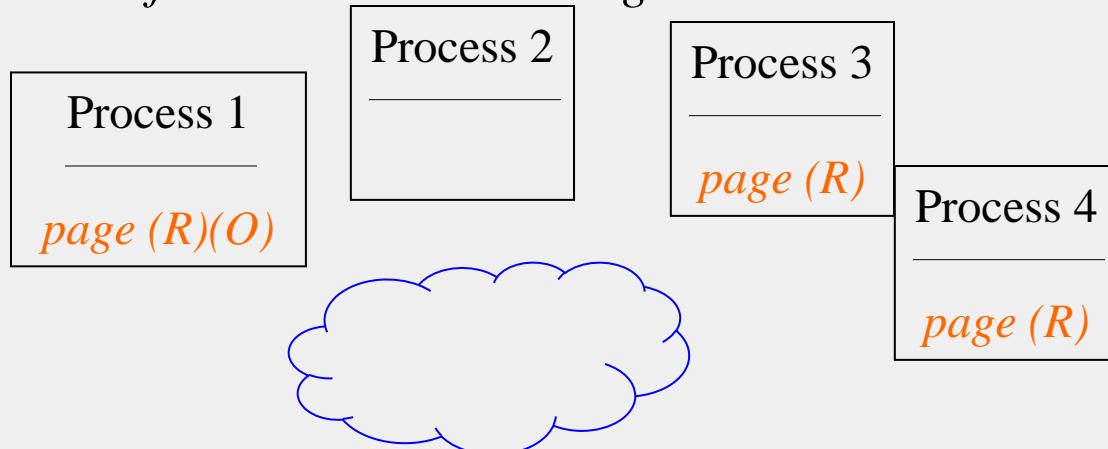
PROCESS 1 ATTEMPTING A READ: SCENARIO 2

- Process 1 is owner (O) and has page in W state
- *Read from cache. No messages sent.*



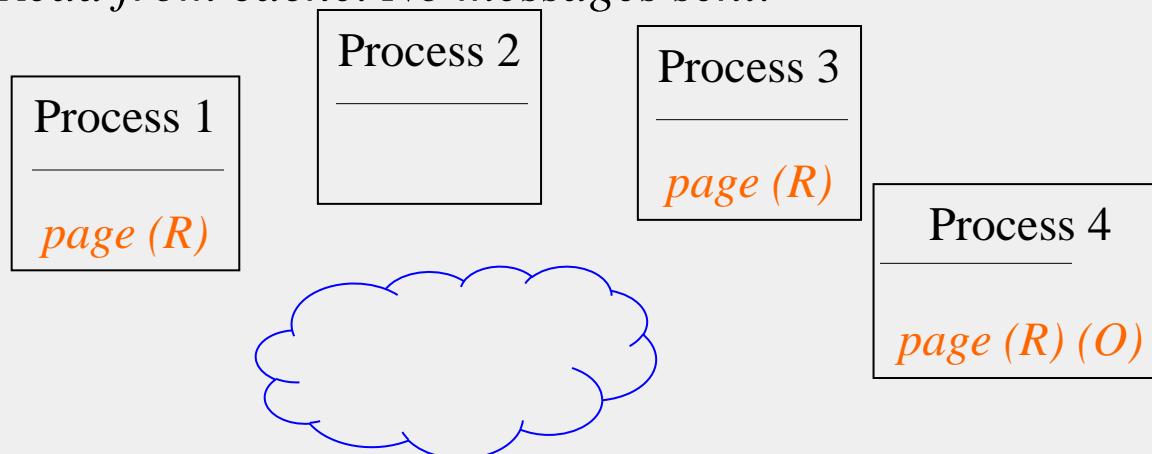
PROCESS 1 ATTEMPTING A READ: SCENARIO 3

- Process 1 is owner (O) and has page in R state
- Other processes also have page in R state
- *Read from cache. No messages sent.*



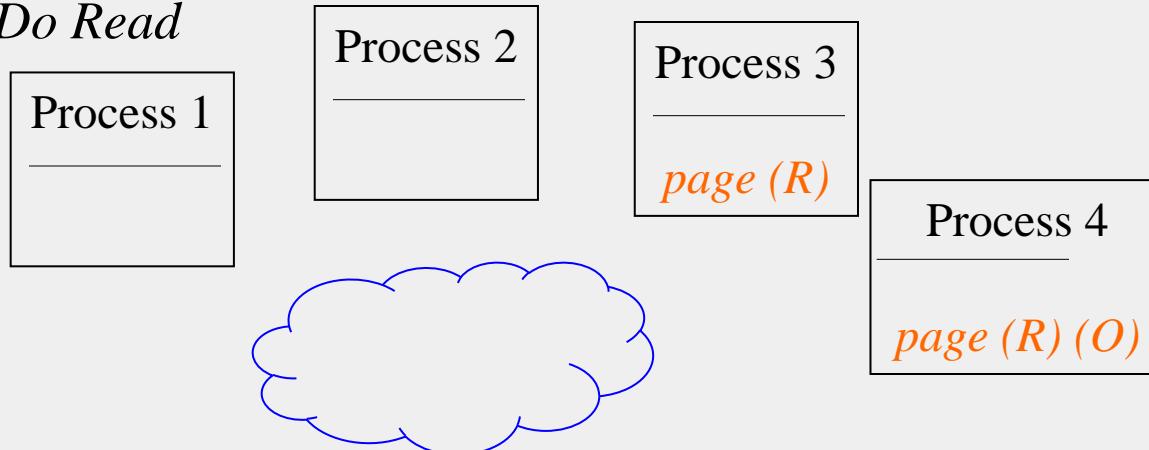
PROCESS 1 ATTEMPTING A READ: SCENARIO 4

- Process 1 has page in R state
- Other processes also have page in R state, and someone else is owner
- *Read from cache. No messages sent.*



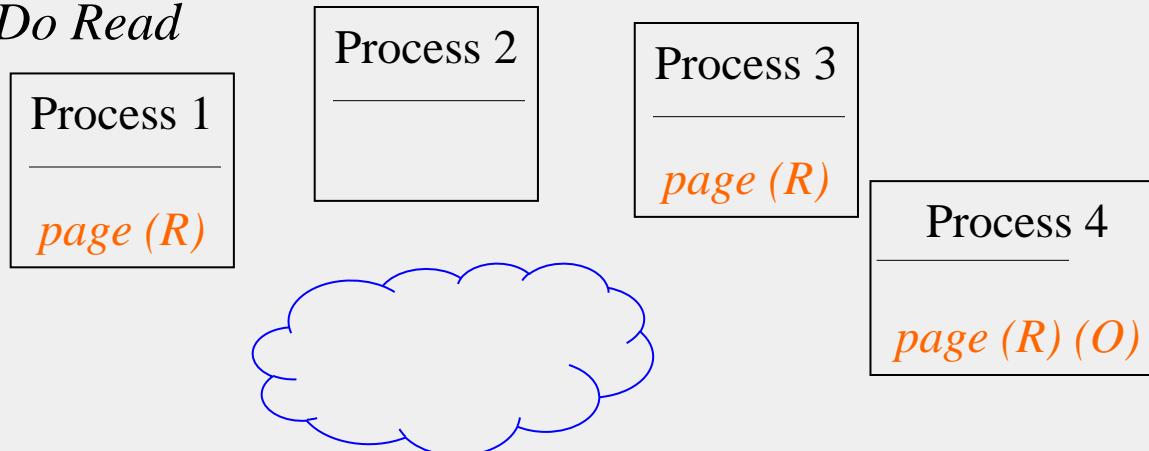
PROCESS 1 ATTEMPTING A READ: SCENARIO 5

- Process 1 does not have page
- Other process(es) has/have page in (R) state
- *Ask for a copy of page. Use multicast.*
- *Mark it as R*
- *Do Read*



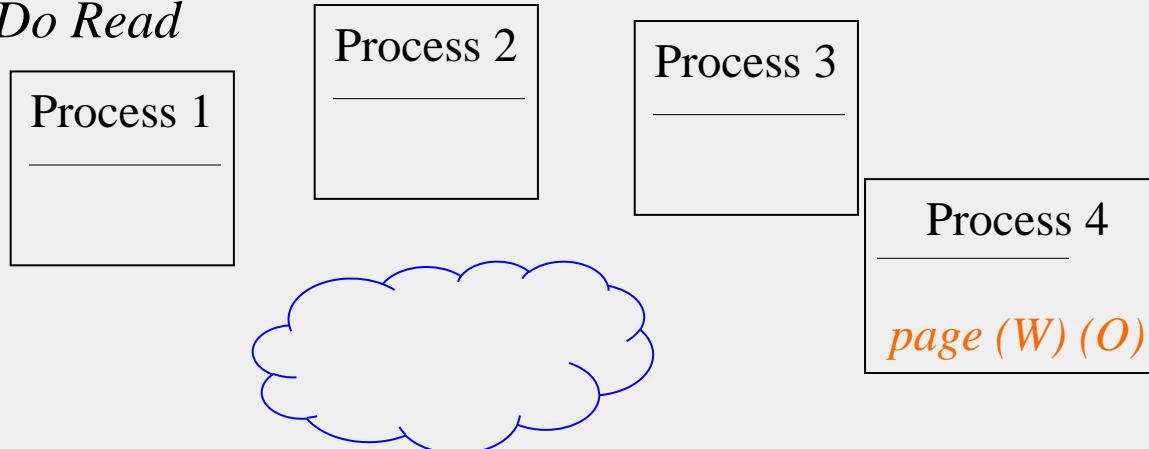
END STATE: READ SCENARIO 5

- Process 1 does not have page
- Other process(es) has/have page in (R) state
- *Ask for a copy of page. Use multicast.*
- *Mark it as R*
- *Do Read*



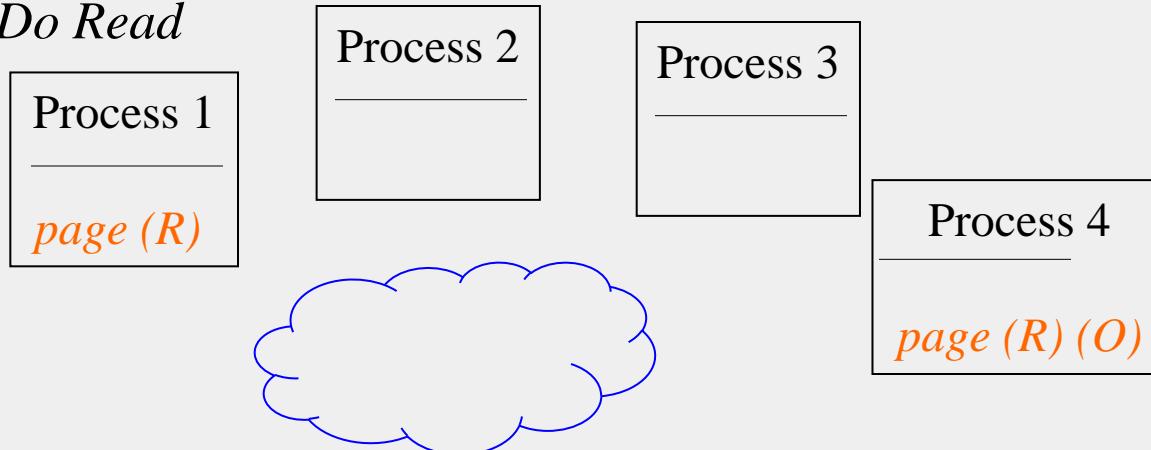
PROCESS 1 ATTEMPTING A READ: SCENARIO 6

- Process 1 does not have page
- Another process has page in (W) state
- *Ask other process to degrade its copy to (R). Locate process via multicast*
- *Get page; mark it as R*
- *Do Read*



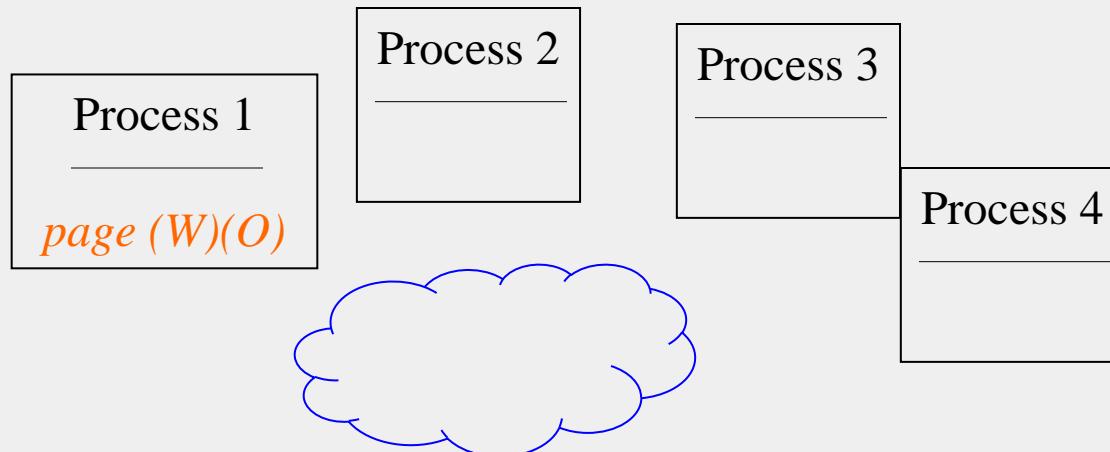
END STATE: READ SCENARIO 6

- Process 1 does not have page
- Another process has page in (W) state
- *Ask other process to degrade its copy to (R). Locate process via multicast*
- *Get page; mark it as R*
- *Do Read*



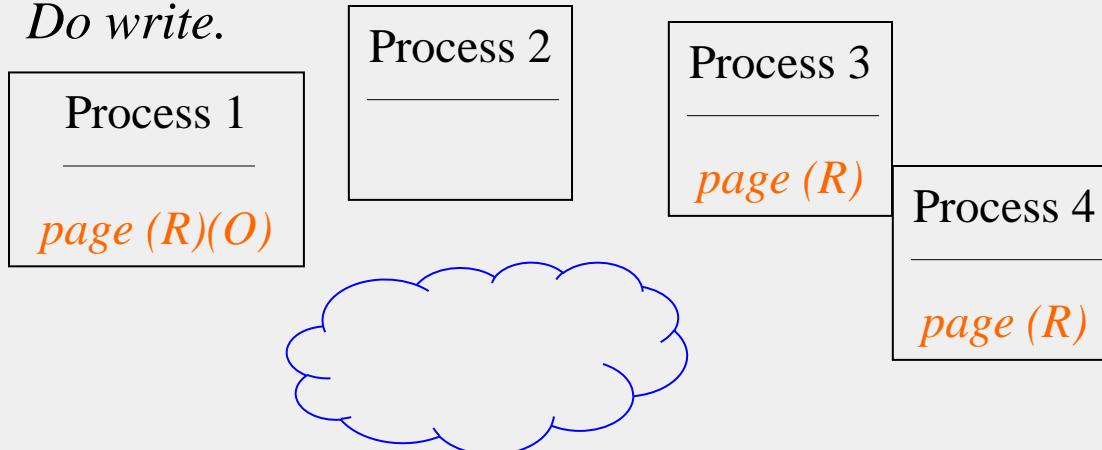
PROCESS 1 ATTEMPTING A WRITE: SCENARIO 1

- Process 1 is owner (O) and has page in W state
- *Write to cache. No messages sent.*



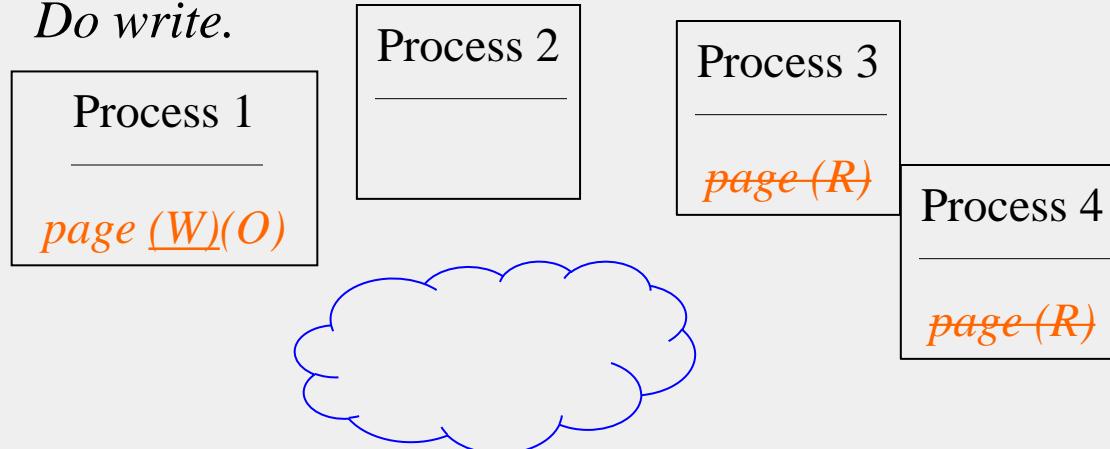
PROCESS 1 ATTEMPTING A WRITE: SCENARIO 2

- Process 1 is owner (O) has page in R state
- Other processes may also have page in R state
- Ask other processes to invalidate their copies of page. Use multicast.
- Mark page as (W).
- Do write.



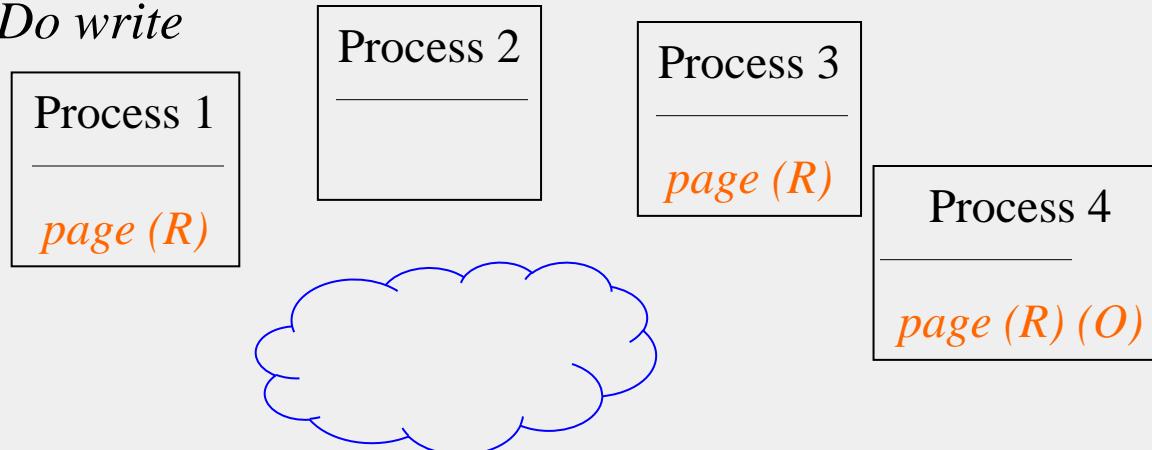
END STATE: WRITE SCENARIO 2

- Process 1 is owner (O) has page in R state
- Other processes may also have page in R state
- Ask other processes to invalidate their copies of page. Use multicast.
- *Mark page as (W).*
- *Do write.*



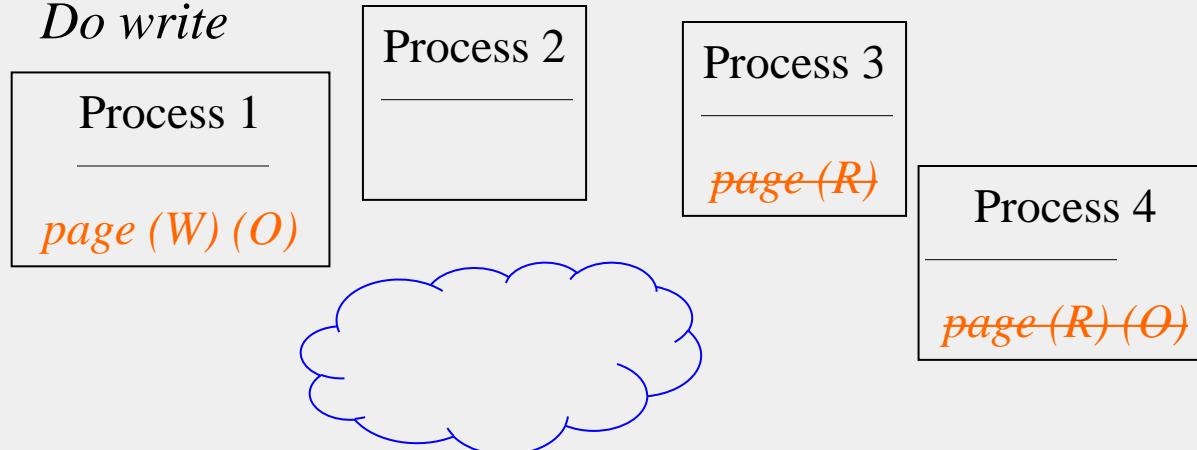
PROCESS 1 ATTEMPTING A WRITE: SCENARIO 3

- Process 1 has page in R state
- Other processes may also have page in R state, and someone else is owner
- Ask *other processes* to *invalidate their copies of page. Use multicast.*
- *Mark page as (W), become owner*
- *Do write*



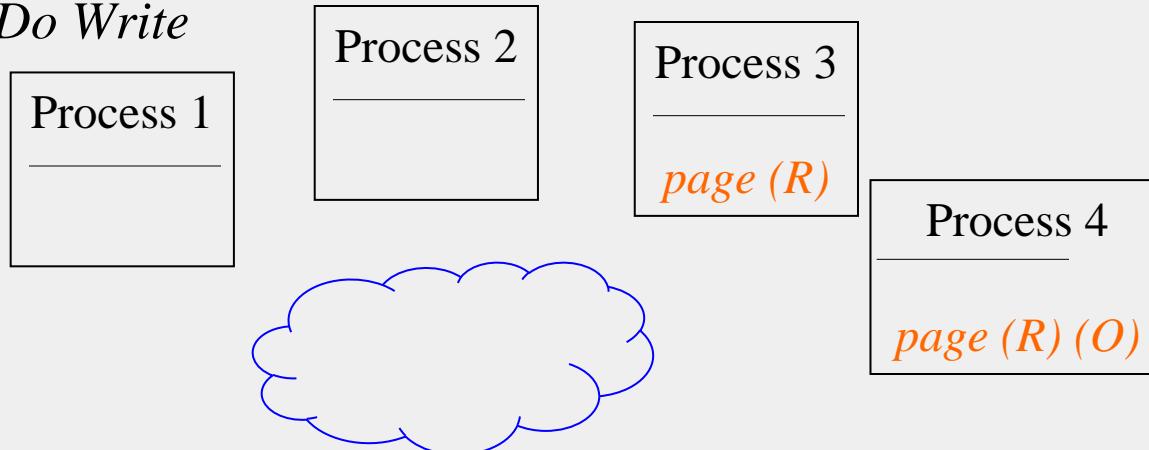
END STATE: WRITE SCENARIO 3

- Process 1 has page in R state
- Other processes may also have page in R state, and someone else is owner
- Ask other processes to invalidate their copies of page. Use multicast.
- *Mark page as (W), become owner*
- *Do write*



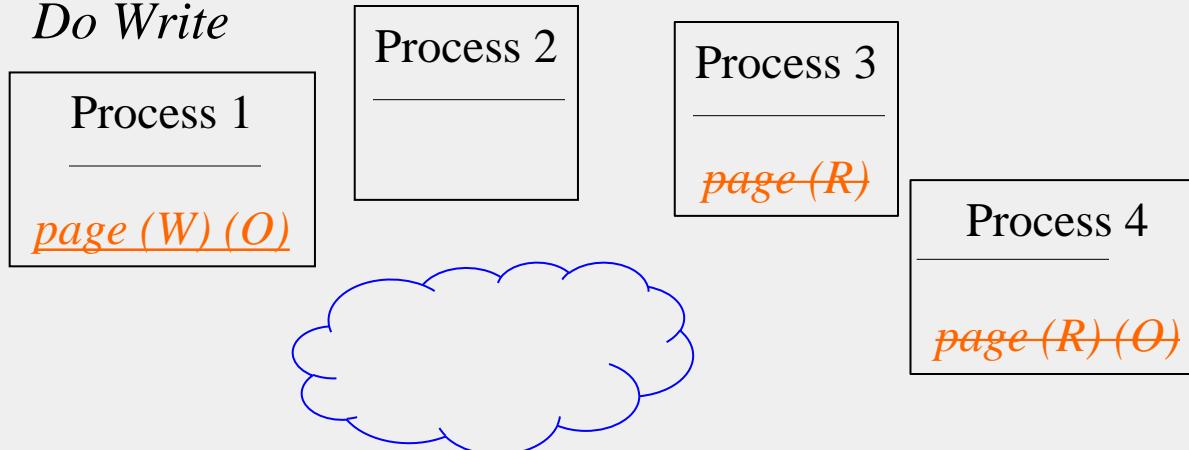
PROCESS 1 ATTEMPTING A WRITE: SCENARIO 4

- Process 1 does not have page
- Other process(es) has/have page in (R) or (W) state
- *Ask other processes to invalidate their copies of the page. Use multicast.*
- *Fetch all copies; use the latest copy; mark it as (W); become owner*
- *Do Write*



END STATE: WRITE SCENARIO 4

- Process 1 does not have page
- Other process(es) has/have page in (R) or (W) state
- *Ask other processes to invalidate their copies of the page. Use multicast.*
- *Fetch all copies; use the latest copy; mark it as (W); become owner*
- *Do Write*



INVALIDATE DOWNSIDES

- That was the invalidate approach
- If two processes write same page concurrently
 - Flip-flopping behavior where one process invalidates the other
 - Lots of network transfer
 - Can happen when unrelated variables fall on same page
 - Called **false sharing**
- Need to set page size to capture a process' *locality of interest*
- If page size much larger, then have false sharing
- If page size much smaller, then too many page transfers => also inefficient

AN ALTERNATIVE APPROACH: UPDATE

- Instead: could use **Update** approach
 - Multiple processes allowed to have page in W state
 - On a write to a page, multicast newly written value (or part of page) to all other holders of that page
 - Other processes can then continue reading and writing page
- Update preferable Invalidate
 - When lots of sharing among processes
 - Writes are to small variables
 - Page sizes large
- Generally though, Invalidate better and preferred option

CONSISTENCY

- Whenever multiple processes share data, consistency comes into picture
- DSM systems can be implemented with:
 - Linearizability
 - Sequential Consistency
 - Causal Consistency
 - Pipelined RAM (FIFO) Consistency
 - Eventual Consistency
 - (Also other models like Release consistency)
 - These should be familiar to you from the course!
- As one goes down this order, speed increases while consistency gets weaker

Is it ALIVE?

- DSM was very popular over a decade ago
- But may be making a comeback now
 - Faster networks like Infiniband + SSDs => Remote Direct Memory Access (RDMA) becoming popular
 - Will this grow? Or stay the same as it is right now?
 - Time will tell!

SUMMARY

- **DSM = Distributed Shared Memory**
 - Processes share pages, rather than sending/receiving messages
 - Useful abstraction: allows processes to use same code as if they were all running over the same OS (multiprocessor OS)
- DSM can be implemented over a message-passing interface
- Invalidate vs. Update protocols



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SENSOR NETWORKS

Lecture A

SENSORS AND THEIR NETWORKS

EVERYTHING'S GETTING SMALLER

- Smallest state-of-the-art transistor today is made of a single gold atom
 - Still in research, not yet in industry.
- Pentium P4 contains 42 M transistors
- Gold atomic weight is 196 ~ 200.
- 1 g of Au contains 3×10^{21} atoms => 7.5×10^{18} P4 processors from a gram of Au => 1 billion P4's per person
- CPU speedup $\sim \sqrt{(\# \text{ transistors on die})}$

SENSORS HAVE BEEN AROUND FOR CENTURIES

- Coal mines have always had CO/CO₂ sensors: “canary in a coal mine”
- Industry has used sensors for a long time, e.g., in assembly line

Today...

- Excessive information
 - Environmentalists collecting data on an island
 - Army needs to know about enemy troop deployments
 - Humans in society face information overload
- Sensor networking technology can help filter and process this information

TRENDS

Growth of any technology requires

- I. Hardware
- II. Operating Systems and Protocols
- III. Killer applications
 - Military and Civilian

SENSOR NODES

- Motivating factors for emergence:
applications, Moore's Law (or variants),
wireless comm., MEMS (micro electro
mechanical sensors)
- Canonical *Sensor Node* contains
 1. Sensor(s) to convert a different energy form to
an electrical impulse – e.g., to measure
temperature
 2. Microprocessor
 3. Communications link – e.g., wireless
 4. Power source – e.g., battery

SENSOR MOTES

- Size: small
 - MICA motes: Few inches
 - MicaDot: Few centimeters
 - Intel Motes: Few centimeters
 - Even smaller: Golem Dust=11.7 cu. mm
- Everything on one chip: micro-everything
 - processor, transceiver, battery, sensors, memory, bus
 - MICA: 4 MHz, 40 Kbps, 4 KB SRAM / 512 KB Serial Flash, lasts 7 days at full blast on 2 x AA batteries

TYPES OF SENSORS

- Micro-sensors (MEMS, Materials, Circuits)
 - Acceleration, vibration, sound, gyroscope, tilt, magnetic, motion, pressure, temp, light, moisture, humidity, barometric
- Chemical
 - CO, CO₂, radon
- Biological
 - Pathogen detectors
- [In some cases, actuators too (mirrors, motors, smart surfaces, micro-robots)]

I₂C Bus

- Developed By Philips
- Inter-IC connect
 - e.g., connect sensor to microprocessor
- Simple features
 - Has only 2 wires
 - Bi-directional
 - Serial data (SDA) and serial clock (SCL) bus
- Up to 3.4 Mbps

TRANSMISSION MEDIUM

- Spec, MICA: Radio Frequency (RF)
 - Broadcast medium, routing is “store and forward,” links are bidirectional
- Smart Dust: smaller size but RF needs high frequency => higher power consumption

Optical transmission: simpler hardware, lower power

- Directional antennas only, broadcast costly
- Line of sight required
- Switching links costly: mechanical antenna movements
- Passive transmission (reflectors) => “wormhole” routing
- Unidirectional links

SUMMARY: SENSOR NODE

- Small size: few mm to a few inches
- Limited processing and communication
 - Mhz clock, MB flash, KB RAM, 100's Kbps (wireless) bandwidth
- Limited power (MICA: 7-10 days at full blast)
- Failure prone nodes and links (due to deployment, fab, wireless medium, etc.)
- But easy to manufacture and deploy in large numbers
- *Need to offset this with scalable and fault-tolerant OS's and protocols*

SENSOR NODE OPERATING SYSTEM

Issues

- Size of code and run-time memory footprint
 - Embedded system OS's inapplicable: need hundreds of KB ROM
- Workload characteristics
 - Continuous? Bursty?
- Application diversity
 - Want to reuse sensor nodes
- Tasks and processes
 - Scheduling
 - Hard and soft real-time
- Power consumption
- Communication

TINYOS FOR SENSOR NODES

Developed at Berkeley (2000's), then @Crossbow Inc.

- Bursty dataflow-driven computations
- Multiple data streams => concurrency-intensive
- Real-time computations (hard and soft)
- Power conservation
- Size
- Accommodate diverse set of applications

TinyOS:

- - Event-driven execution (*reactive* mote)
- - Modular structure (components) and clean interfaces

PROGRAMMING TINYOS MOTES

- Use a variant of C called NesC
- NesC defines *components*
- A component is either:
 - A *module* specifying a set of methods and internal storage (~like a Java static class)
 - A module corresponds to either a hardware element on the chip (e.g., the clock or the LED), or to a user-defined software module
 - Modules implement and use *interfaces*
 - Or a *configuration*, a set of other components *wired* together by specifying the unimplemented methods
- A complete NesC application then consists of one top level configuration

TINYOS COMPONENTS

- Component invocation is event driven, arising from hardware events
- Static allocation only avoids run-time overhead
- Scheduling: dynamic, hard (or soft) real-time
- Explicit interfaces accommodate different applications

DEPLOYING YOUR APPLICATION

(applies to MICA Mote)

- On your PC
 - Write NesC program
 - Compile to an executable for the mote
 - (Simulate and Debug)
 - Plug the mote into the port through a connector board
 - Install the program
- On the mote
 - Turn the mote on, and it's already running your application

ENERGY SAVINGS

- Power saving modes:
 - MICA: active, idle, sleep
- Tremendous variance in energy supply and demand
 - Sources: batteries, solar, vibration, AC
 - Requirements: long term deployment v. short term deployment, bandwidth intensiveness
 - 1 year on 2xAA batteries => 200 uA average current

FALLOUT

- TinyOS is small: Software Footprint = 3.4 KB
 - Can't load a lot of data
- Power saving modes:
 - MICA: active, idle, sleep
- Radio Transmit is the most expensive (12 mA)
 - CPU Active: 4.6 mA
 - => Better compute than transmit
- => Lead to **in-network aggregation** approaches
 - Build trees among sensor nodes, base station at root of tree
 - Internal nodes receive values from children, calculate summaries (e.g., averages) and transmit these
 - More power-efficient than transmitting raw values or communicating directly with base station

FALLOUT (2)

- Correct direction for future technology
 - Today's growth rates: data > storage > CPU > communication > batteries
- Due to hostile environments (battlefields, environmental observation) and cheap fabrication
 - High failure rates in sensor nodes
 - Need sensor networks to be
 - Self-organizing
 - Self-managing
 - Self-healing
 - Scalable: Number of messages as function of number of nodes
- Broader (but related direction)
 - ASICs: Application-Specific Integrated Chips
 - FPGAs: Field Programmable Gate Arrays
 - Faster because move more action into hardware!

SUMMARY

- Sensor nodes are cheap and battery-limited
- Deploy them in inhospitable terrains =>
 - Need to conserve power
 - Be smart about design of OS and distributed protocol
- TinyOS design
- Distributed protocol challenges

SOME TOPICS FOR YOU TO LOOK UP

- Raspberry PI
 - Cheap computer, programmable
- Arduino
- Home automation systems: Nest, AMX, Homelogic, Honeywell, etc.
 - Power concerns smaller (since connected to power), but key security and accuracy concerns
- Network such devices together
 - Often called “Internet of Things”
 - Also called “cyberphysical systems”
- Cars today are networks of sensors
- Combination of humans and machines often called “Cyberphysical systems”
 - Operation theater (in hospitals) are becoming networks of sensors



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

DATA CENTER OUTAGE STUDIES

Lecture A

WHAT CAUSES DISASTERS?

TAKE A GUESS!

Which of the following do you think is the leading cause of datacenter outages?

1. Power outage
2. Overheating
3. Human error
4. Fire
5. DOS attacks

TAKE A GUESS!

Which of the following do you think is the leading cause of datacenter outages?

1. Power outage
2. Overheating
3. Human error (70%)
4. Fire
5. DOS attacks



HUMAN ERROR EXAMPLES

- A system operator mistakenly deleted the \$38 billion Alaska Permanent Fund database and then deleted its backup.
- A maintenance contractor's mistake shut down the Oakland Air Traffic Control Center.
- A State of Virginia technician pulled the wrong controller and crashed a redundant SAN that already had suffered a controller failure.
- A technician with DBS Bank made an unauthorized repair on a redundant SAN and took down both sides.

Source:

http://www.availabilitydigest.com/public_articles/0704/data_center_outages-lessons.pdf



HUMAN ERROR EXAMPLES (2)

- A system administrator closed all applications on one server in an active/active pair to upgrade it and then shut down the operating server.
- A test technician failed to disable a fire alarm actuator prior to testing the fire suppression system.
- Siren noise damaged several disks, including the virtual backup disks.
- (hosting.com) Incorrect breaker operation sequence executed by servicing vendor caused a shutdown of UPS and offline time to websites of 1-5 hours
- Thirteen million German websites went dark when an operator mistakenly uploaded an empty zone file.
- (And many more!)

Source:

http://www.availabilitydigest.com/public_articles/0704/data_center_outages-lessons.pdf

WHY STUDY OUTAGES?

- They're fun! (Schadenfreude!)
- But really – so that we can learn lessons
- Learn more about the actual behavior of systems in the real world
- Design better systems in the future
- Not our goal to say some companies are worse than others
 - In fact, companies that suffer outages run better and more robust infrastructure afterwards!
 - “What doesn't kill you, makes you stronger”
- We'll see a few case studies of outages
 - And learn lessons from them



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

DATA CENTER OUTAGE STUDIES

Lecture B

AWS OUTAGE



OVERVIEW

- Occurred on April 21st, 2011
- AWS published a post-mortem analysis
 - <http://aws.amazon.com/message/65648/>
- Not our goal to say AWS is a bad infrastructure
 - In fact, quite the opposite – AWS treated customers very well
 - After the outage, AWS is still market leader
 - AWS fixed infrastructure to prevent recurrence
- During the outage
 - Several companies using AWS EC2 went down, e.g., Reddit, Foursquare
 - AWS dashboard showed problems with EC2, and other storage
 - Lasted 3.5 days (at least)
 - Led to some data loss

BACKGROUND

- AWS **Regions**: Separate datacenters
 - E.g., us-east-1, us-west-1, etc.
 - Each region consists of **availability zones**
 - Can have automatic data replication across zones in a region (though not all customers do it)
- AWS Elastic Block Storage (**EBS**) – mountable storage “devices,” accessible from EC2 instances
- 1 EBS volume runs inside an Availability Zone
 - Two networks: primary n/w used for EC2 and EBS control plane; secondary n/w used for overflow – has *lower capacity*
 - Control information replicated across zones (for availability)
- EBS volumes replicated for durability
 - Each volume has a primary replica
 - If out of sync or node failure, replicas programmed to do aggressive re-mirroring of data

TIMELINE OF OUTAGE

- *12:47 AM: Routine primary n/w capacity upgrade in an av. zone in US East Region*
- Traffic shifted off several primary n/w routers to other primary n/w routers
 - Critical Error: someone shifted traffic for one such router to a secondary n/w router
- => Several EBS volumes now had no/bad primary n/w
 - Primary n/w disconnected
 - Second n/w has low capacity and thus overwhelmed
 - Many primary replicas had no backup
- Team discovered critical error and rolled it back

(Is it over yet?)

TIMELINE OF OUTAGE (2)

- Team discovered critical error and rolled it back
 - Due to network partitioning, many primary replicas thought they had no backup: these automatically started re-mirroring aggressively
 - *All at once*: free n/w cap quickly used, replicas stuck in loop
 - Re-mirroring *storm*: 13% of EBS volumes
- N/w unavailable for control plane
 - Unable to serve “create volume” API requests for EBS
 - Control plane ops have long time-out; began to back up
 - When thread pool filled up, control plane started to reject create volume requests
- *2:40 AM: Team disabled all new “create volume” API requests*
- *2:50 AM: all error rates and latencies for EBS APIs start to recover*

(Is it over yet?)

TIMELINE OF OUTAGE (3)

- Two issues made things worse
 - Primaries searching for potential replicas did not back off
 - A race condition existed in EBS code that was only triggered by high request rates: activated now, caused more node failures
- *5:30 AM: Error rates and latencies increase again*
- Re-mirroring is negotiation b/w EC2 node, EBS node, and EBS control plane (to ensure 1 primary)
 - Due to race condition, EBS nodes started to fail
 - Rate of negotiations increased
 - Caused more node failures (via race), and rinse-n-repeat
 - “Brown-out” of EBS API functionalities
- *8:20 AM: Team starts disabling all communication b/w EBS cluster in affected av. zone and EBS control plane*
 - Av. zone still down, but control plane recovering slowly

TIMELINE OF OUTAGE (4)

- *11:30 am: Team figures out how to prevent EBS servers in av. zone from futile re-mirroring*
 - Affected av. zone slowly recovers
- Customers still continued to face high error rates for new EBS-backed EC2 instances until noon
 - Another new EBS control plane API had recently been launched (for attaching new EC2 instances to volumes)
 - Its error rates were being shadowed by new errors
- Noon: No more volumes getting stuck
- But 13% volumes still in stuck state

TIMELINE OF OUTAGE (5)

- Long tail of recovery
 - Read more on the post-mortem to find out how team addressed this
 - By noon April 24th, all but 1.04% of volumes had been restored
 - Eventually, 0.07% volumes could not be recovered, and were lost forever
- This outage also affected relational database service (RDS) that were single – av. zone.

GENERAL LESSONS LEARNT

Large outages/failures

- Often start from human error
- But balloon due to *cascading* sub-failures

SPECIFIC LESSONS LEARNT

Ways this outage could have been avoided:

- Audit n/w configuration change processes, create a step-by-step protocol for upgrades
- Higher capacity in secondary n/w
- Prevent re-mirroring storm: backing off rather than aggressively retry
- Fixing race condition
- Users who wrote code to take advantage of multiple av. zones within region not affected
- Better tools for communication, health (AWS Dashboard), service credit for customers (multi-day credit)



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

DATA CENTER OUTAGE STUDIES

Lecture C

FACEBOOK OUTAGE

OVERVIEW

- Outage occurred on 23rd September, 2010
- FB unreachable for 2.5 hours (worst in past 4 years)
- Facebook published post-mortem
 - <https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919>
- Not our goal to say Facebook is a bad infrastructure
 - In fact, after the outage, Facebook is still market leader in social networks
 - Facebook fixed infrastructure to prevent recurrence

BACKGROUND

- Data stored in a persistent store and cache
 - Persistent store = many servers
 - Cache = many servers running a distributed cache system
- Includes configuration data
- FB has automated system for verifying configuration values in the cache
 - And replace invalid values with updated values from the store



TIMELINE

- On Sep 23, FB made a change to the persistent copy of a configuration
 - Change was invalid
- All clients (FB cache servers) saw invalid value
 - All attempted to fix it
 - All queried cluster of databases
 - Databases overwhelmed quickly by 100K's queries per second
- Team fixed the invalid configuration

(Is it over yet?)

TIMELINE (2)

- When client received error from DB, it interpreted it as invalid and deleted cache entry
 - When DB failed to respond => client created more queries
 - No back off
 - Rinse-n-repeat
 - (Cascading failures)

TIMELINE (3)

- FB's solution
 - Turn off entire FB website
 - Stop all traffic to DB cluster
 - DB recovers
 - Slowly allow users back on: allowed clients to slowly update caches
 - Took until later in day for entire site to be back up

LESSONS LEARNT

- New configuration system design
- When cannot access resource
 - Don't retry aggressively
 - But instead, back off
 - Each time a request fails, wait twice as long as last time
 - Called “Exponential backoff”
 - Used in networking protocols like 802.11 and TCP to avoid congestion



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

DATA CENTER OUTAGE STUDIES

Lecture D

THE PLANET OUTAGE

OVERVIEW

- Outage occurred on 31st May, 2008
- Source:
http://www.availabilitydigest.com/public_articles/0309/planet_explosion.pdf
- The Planet – 4th largest web hosting company,
supported 22K websites
 - 6 datacenters: Houston (2), Dallas (4)
- Took down 9K servers and 7.5K businesses

TIMELINE

- 5.55 pm: Explosion in H1 Houston DC
 - Short circuit in transformer set it on fire
 - Caused an explosion of battery-acid fumes from UPS backup
 - (Cascading failures)
 - Blew out 3 walls of first floor

TIMELINE (2)

- No servers were damaged, but 9K servers brought down
- Fire department evacuated building
 - Directed that backup generators could not be turned on
 - Due to fire hazard, no staff allowed back in until 10 pm
- The rumor is that the Planet staff had to physically ship some critical servers to their other DCs (on pickups)
 - But limited by power and cooling at other DCs

TIMELINE (3)

- 5 pm Jun 2: Power restored to second floor
- Jun 4: First floor servers were being restored one rack at a time
- All the while: The Planet provided frequent updates to customers (15 min to 1 hour)

LESSONS LEARNT

- Backup data & services across DCs, perhaps across different providers
 - “Business Continuity Plans”
 - Whose responsibility would this be?
 - Provider?
 - Customer? More difficult due to extra work and data lock-in across providers.
- May cost customers more
 - Like insurance premiums?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

DATA CENTER OUTAGE STUDIES

Lecture E

WRAP-UP



OUTAGES ARE INEVITABLE

- Outages are inevitable
- We've seen how AWS, Facebook, The Planet kept affected users updated throughout
 - Frequent updates
 - Coupons/discounts
 - Published post-mortems afterwards
 - All these bolster customer confidence
- Many companies run dashboards with real-time information
 - Google Apps status dashboard
 - AWS dashboard

NOT ALL COMPANIES ...

Not all companies are as open as those discussed

- RIM Apr 2007 – day-long outage; no details
- Hostway Jul 2007 – informed customers that it would move its DC Miami → Tampa, and that outage would be 12 hours
 - Outage was 3-7 days

OVERALL LESSONS LEARNT

- Datacenter fault-tolerance akin to human ailments/medicine today
 - Most common illnesses (crash failures) addressed
 - But uncommon cases can be horrible (unexpected outages)
- *Testing* is important
 - American Eagle, during a disaster, discovered that they could not fail over to backup DC
- Failed upgrades common cause of outage
 - Need a fallback plan

OVERALL LESSONS LEARNT (2)

- Data availability and recovery
 - BCP, Disaster-tolerance
 - Cross-DC replication, either by provider or by customer
- Consistent documentation
 - A Google AppEngine outage prolonged because ops did not know which version of docs to use for recovery
 - Google's fix: mark old documents explicitly as “deprecated”
- Outages always a cascading series of failures
 - Need more ways to break the chain and prevent outages

OVERALL LESSONS LEARNT (3)

- Other sources of outages
 - DOS-resistance
 - Internet outages
 - Under-sea cable cut, DNS failures, government blocking Internet (mostly via DNS)
 - Solution: Alternate DNS services
- Many failures are unexpected
- But there are also planned outages (e.g., kernel upgrades)
 - Need to be planned well
 - Steps documented and followed
 - Fallback plans in place

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

Part 2
CONCLUSION

WHAT YOU'VE LEARNT SO FAR - C3 PART 1

- Introduction: Clouds, Mapreduce, Key-value stores
- Classical Precursors: Peer-to-peer systems, Grids
- Widely-used algorithms: Gossip, Membership, Paxos
- Classical algorithms: Time and Ordering, Snapshots, Multicast
- Fun: Interviews with leading managers and researchers, from both industry and academia

WHAT YOU'VE LEARNT SO FAR - C3 PART 2

- Classical algorithms: Leader Election, Mutual Exclusion, Scheduling
- Scalability: Concurrency control, Replication Control
- Trending Areas: Stream processing, Graph processing, Structure of Networks, Sensor Networks
- Miscellaneous: Distributed File systems, Distributed shared memory, Security, Datacenter outage studies
- Fun: Interviews with leading managers and researchers, from both industry (Google, Microsoft, Yahoo) and academia

WHAT YOU'VE DONE SO FAR (PART 1 + PART 2)

- Homeworks
- 2 Programming Assignment
 - Implement a membership protocol inside an emulator
 - Implement a key-value store inside an emulator
- 2 Exams

SUMMARY

- You've now learnt the **distributed systems concepts** that underlie today's cloud computing technologies
- You've learnt
 - Concepts
 - Techniques
 - Industry systems, including open-source (from the inside)
- You've seen
 - Distributed systems
 - Distributed algorithms
 - As applied to cloud computing

ONWARD!

- You now have the background to go exploring through the various cloud computing systems out there
- Go ahead and make changes to them!
- For those of you in the Cloud Specialization
 - Other courses in the specialization: *Cloud Applications* and *Cloud Networking*



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SECURITY

Lecture A

BASIC SECURITY CONCEPT

SECURITY THREATS

- **Leakage**
 - Unauthorized access to service or data
 - E.g., someone knows your bank balance
- **Tampering**
 - Unauthorized modification of service or data
 - E.g., someone modifies your bank balance
- **Vandalism**
 - Interference with normal service, without direct gain to attacker
 - E.g., denial-of-service attacks

COMMON ATTACKS

- **Eavesdropping**
 - Attacker taps into network
- **Masquerading**
 - Attacker pretends to be someone else, i.e., identity theft
- **Message tampering**
 - Attacker modifies messages
- **Replay attack**
 - Attacker replays old messages
- **Denial-of-service:** bombard a port

ADDRESSING THE CHALLENGES: CIA PROPERTIES

- Confidentiality
 - *Protection against disclosure to unauthorized individuals*
 - Addresses leakage threat
- Integrity
 - *Protection against unauthorized alteration or corruption*
 - Addresses tampering threat
- Availability
 - *Service/data is always readable/writable*
 - Addresses vandalism threat

POLICIES VS. MECHANISMS

- Many scientists (e.g., Hansen) have argued for a separation of policy vs. mechanism
- A security policy indicates *what* a secure system accomplishes
- A security mechanism indicates *how* these goals are accomplished
- E.g.,
 - Policy: in a file system, only authorized individuals allowed to access files (i.e., CIA properties)
 - Mechanism: Encryption, capabilities, etc.

MECHANISMS: GOLDEN A's

- **Authentication**
 - Is a user (communicating over the network) claiming to be Alice, really Alice?
- **Authorization**
 - Yes, the user is Alice, but is she allowed to perform her requested operation on this object?
- **Auditing**
 - How did Eve manage to attack the system and breach defenses? Usually done by continuously logging all operations.

DESIGNING SECURE SYSTEMS

- Don't know how powerful attacker is
- When designing a security protocol need to
 1. Specify [attacker model](#): Capabilities of attacker
(Attacker model should be tied to reality)
 2. Design security mechanisms to satisfy policy under the attacker model
 3. Prove that mechanisms satisfy policy under attacker model
 4. Measure effect on overall performance (e.g., throughput) in the common case, i.e., no attacks

NEXT

- Basic cryptography



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SECURITY

Lecture B

BASIC CRYPTOGRAPHY CONCEPTS

BASIC SECURITY TERMINOLOGY

- **Principals:** processes that carry out actions on behalf of users
 - Alice
 - Bob
 - Carol
 - Dave
 - Eve (typically evil)
 - Mallory (typically malicious)
 - Sara (typically server)

KEYS

- Key = sequence of bytes assigned to a user
 - Can be used to “lock” a message, and only this key can be used to “unlock” that locked message

ENCRYPTION

- Message (sequence of bytes) + Key →
(Encryption) →
Encoded message (sequence of bytes)
- Encoded Message (sequence of bytes) + Key →
(Decryption) →
Original message (sequence of bytes)
- No one can decode an encoded message without
the key

Two CRYPTOGRAPHY SYSTEMS

I. Symmetric Key systems:

- K_A = Alice's key; secret to Alice
- K_{AB} = Key shared only by Alice and Bob
- Same key used to both encrypt and decrypt a message

• E.g., DES (Data Encryption Standard): 56 b key
operates on 64 b blocks from the message

Two CRYPTOGRAPHY SYSTEMS (2)

II. Public-Private Key systems:

- $K_{A\text{priv}}$ = Alice's **private key**; known only to Alice
- $K_{A\text{pub}}$ = Alice's **public key**; known to *everyone*
- Anything encrypted with $K_{A\text{priv}}$ can be decrypted only with $K_{A\text{pub}}$
- Anything encrypted with $K_{A\text{pub}}$ can be decrypted only with $K_{A\text{priv}}$

•RSA and PGP fall into these categories

- RSA = Rivest Shamir Adleman
- PGP = Pretty Good Privacy
- Keys are several 100s or 1000s of b long
- Longer keys => harder for attackers to break
- Public keys maintained via PKI (Public Key Infrastructure)

PUBLIC-PRIVATE KEY CRYPTOGRAPHY

- If Alice wants to send a secret message M that can be read only by Bob
 - Alice encrypts it with Bob's public key
 - $K_{B\text{pub}}(M)$
 - Bob only one able to decrypt it
 - $K_{B\text{priv}}(K_{B\text{pub}}(M)) = M$
 - Symmetric too, i.e., $K_{A\text{pub}}(K_{A\text{priv}}(M)) = M$

SHARED/SYMMETRIC VS. PUBLIC/PRIVATE

- Shared keys reveal too much information
 - Hard to *revoke* permissions from principals
 - E.g., group of principals shares one key
 - want to remove one principal from group
 - need everyone in group to change key
- Public/private keys involve costly encryption or decryption
 - At least one of these 2 operations is costly
- Many systems use public/private key system to generate shared key, and use latter on messages

NEXT

- How to use cryptography to implement security mechanisms



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SECURITY

Lecture C

IMPLEMENTING MECHANISM
USING CRYPTOGRAPHY

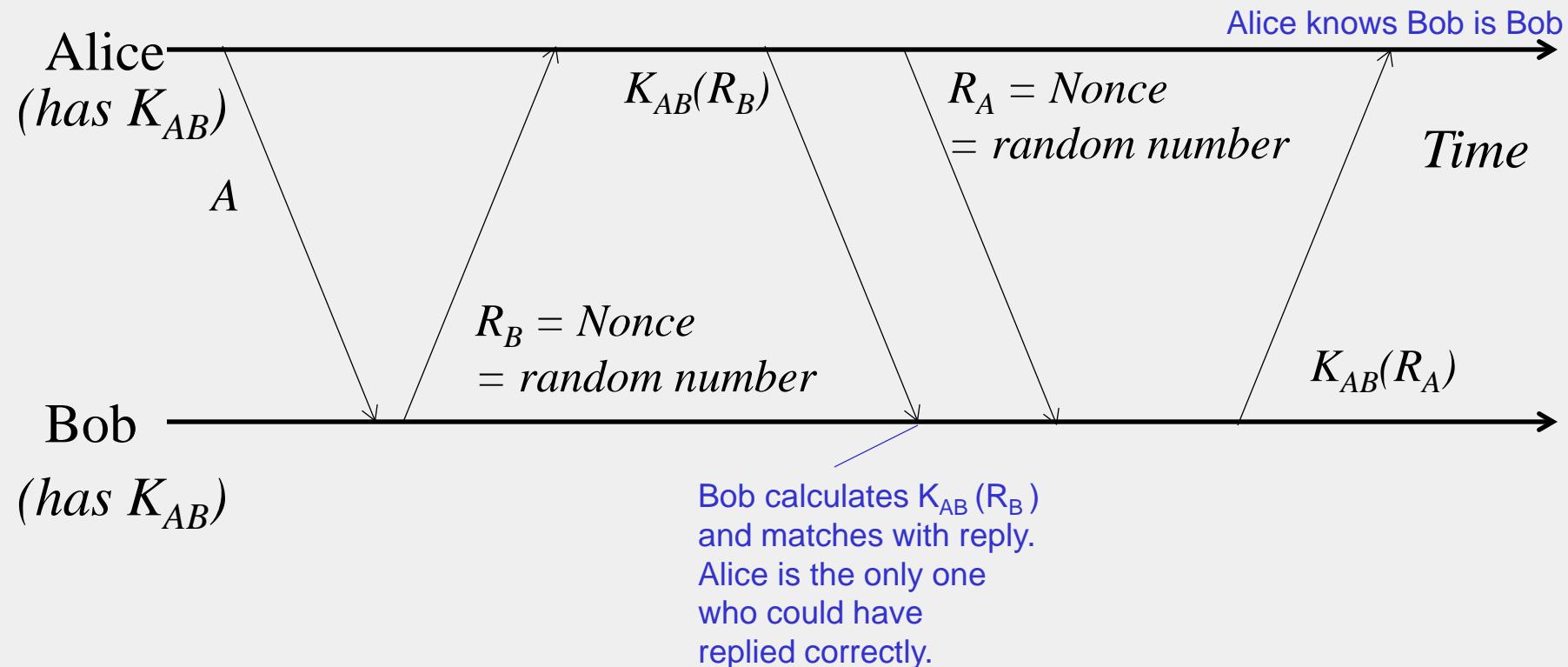
IN THIS LECTURE

- How to use cryptography to implement
 - I. Authentication
 - II. Digital Signatures
 - III. Digital Certificates

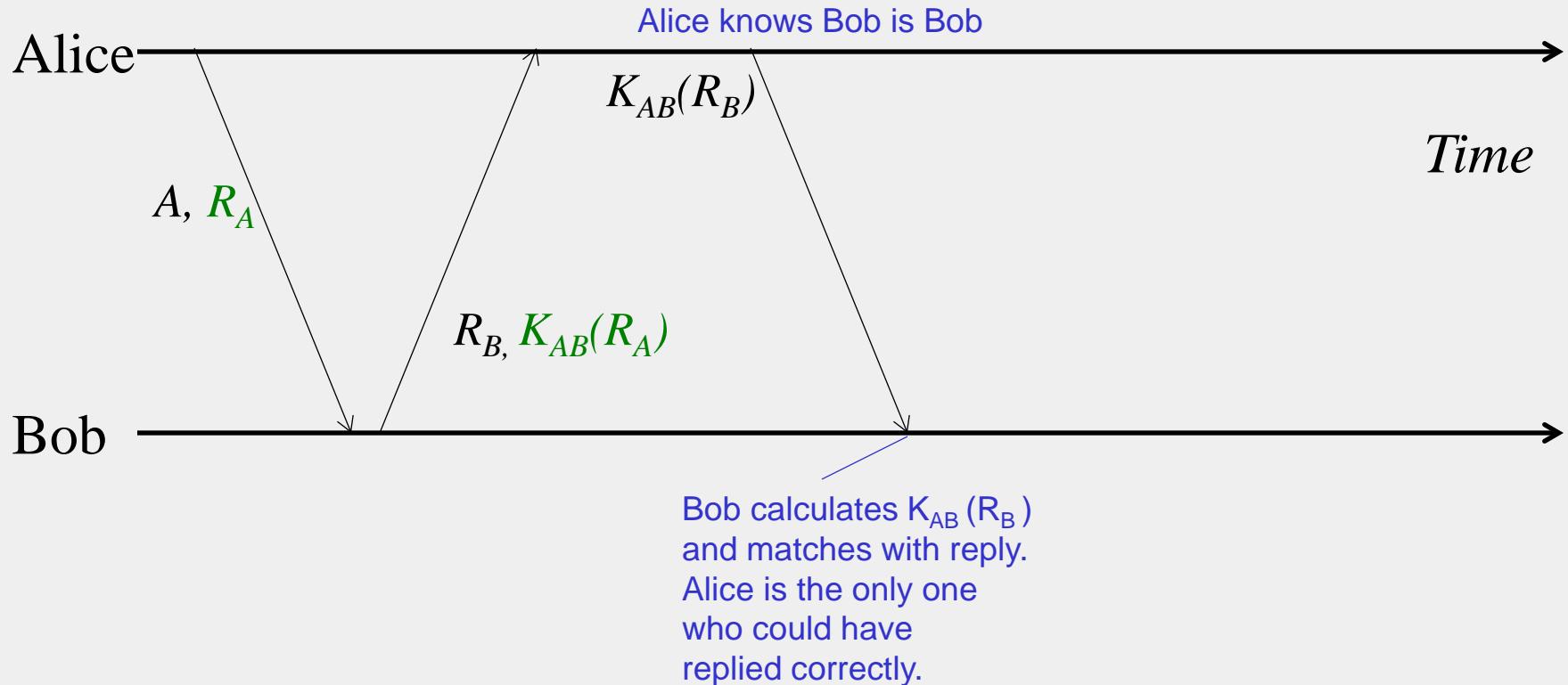
I. AUTHENTICATION

- Two principals verify each other's identities
- Two flavors
 - **Direct authentication:** directly between two parties
 - **Indirect authentication:** uses a trusted third-party server
 - Called authentication server
 - E.g., a Verisign server

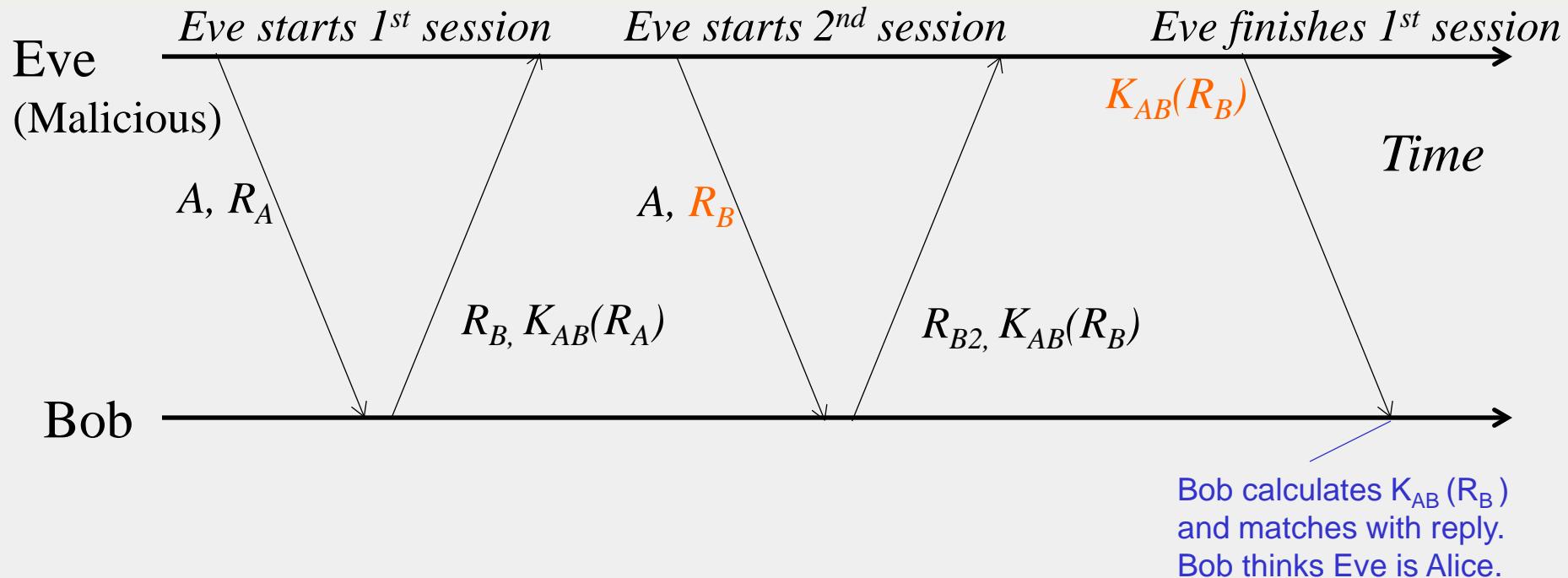
DIRECT AUTHENTICATION USING SHARED KEY



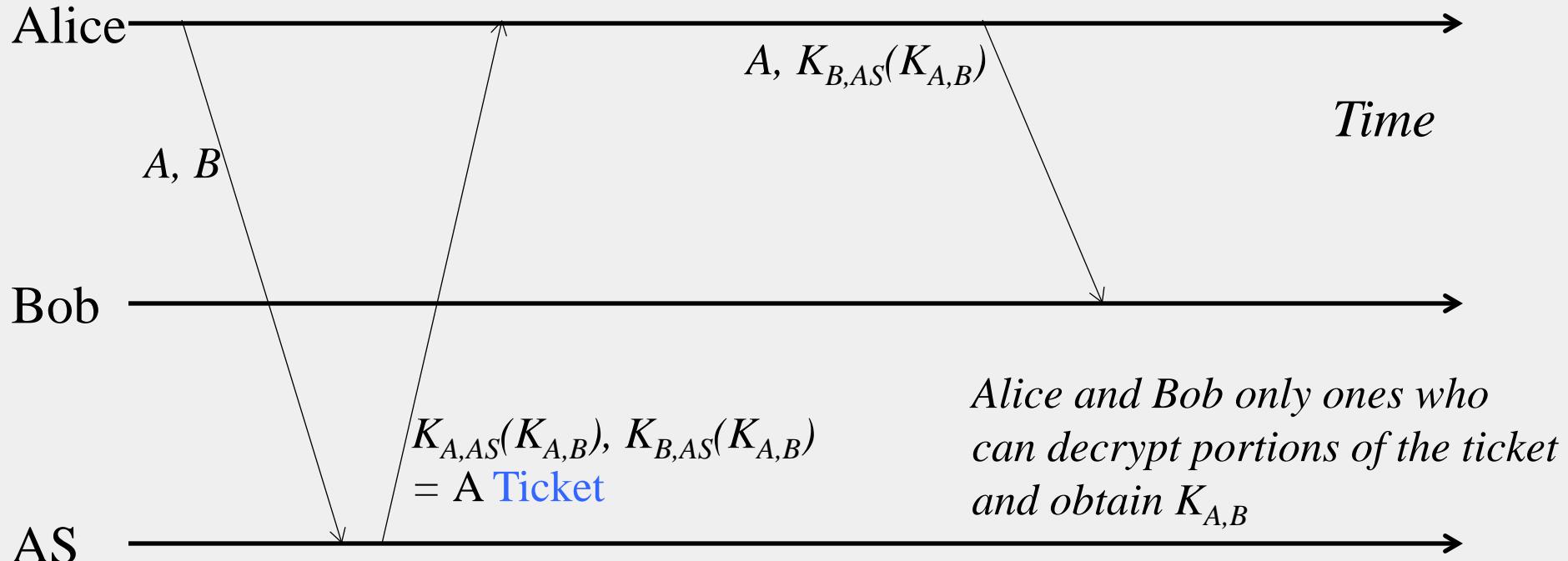
WHY NOT OPTIMIZE NUMBER OF MESSAGES?



UNFORTUNATELY, THIS SUBJECT TO REPLAY ATTACK



INDIRECT AUTHENTICATION USING AUTHENTICATION SERVER AND SHARED KEYS



II. DIGITAL SIGNATURES

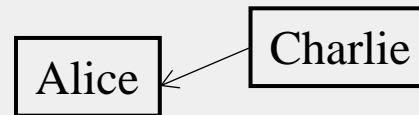
- Just like “real” signatures
 - Authentic, Unforgeable
 - Verifiable, Non-repudiable
- To sign a message M , Alice encrypts message with her own private key
 - Signed message: $[M, K_{A\text{priv}}(M)]$
 - Anyone can verify, with Alice’s public key, that Alice signed it
- To make it more efficient, use a one-way hash function, e.g., SHA-1, MD-5, etc.
 - Signed message: $[M, K_{A\text{priv}}(\text{Hash}(M))]$
 - Efficient since hash is fast and small; don’t need to encrypt/decrypt full message

III. DIGITAL CERTIFICATES

- Just like “real” certificates
- Implemented using digital signatures
- Digital Certificates have
 - Standard format
 - Transitivity property, i.e., chains of certificates
 - Tracing chain backwards must end at trusted authority (at root)

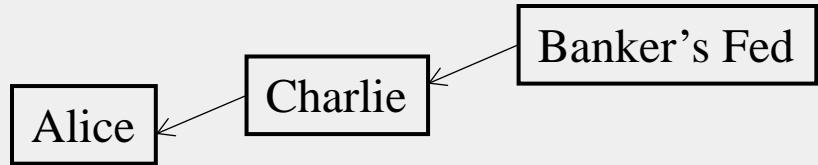
EXAMPLE: ALICE'S BANK ACCOUNT

1. Certificate Type: Account
2. Name: Alice
3. Account number: 12345
4. Certifying Authority: Charlie's Bank
5. Signature
 - $K_{C\text{priv}}(\text{Hash}(\text{Name}+\text{Account number}))$



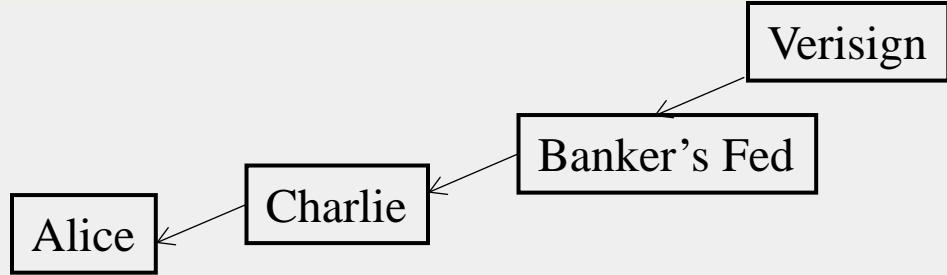
CHARLIE'S BANK, IN TURN HAS ANOTHER CERTIFICATE

1. Certificate Type: Public Key
2. Name: Charlie's Bank
3. Public Key: $K_{C\text{pub}}$
4. Certifying Authority: Banker's Federation
5. Signature
 - $K_{F\text{priv}}(\text{Hash}(\text{Name}+\text{Public key}))$



BANKER'S FEDERATION, HAS ANOTHER CERTIFICATE FROM THE ROOT SERVER

1. Certificate Type: Public Key
2. Name: Banker's Federation
3. Public Key: $K_{F\text{pub}}$
4. Certifying Authority: Verisign
5. Signature
 - $K_{\text{verisign priv}}(\text{Hash}(\text{Name+Public key}))$



IV. AUTHORIZATION

- **Access Control Matrix**
 - For every combination of (principal,object) say what mode of access is allowed
 - May be very large (1000s of principals, millions of objects)
 - May be sparse (most entries are “no access”)
- **Access Control Lists (ACLs)** = per object, list of allowed principals and access allowed to each
- **Capability Lists** = per principal, list of files allowed to access and type of access allowed
 - Could split it up into capabilities, each for a different (principal,file)

SECURITY: SUMMARY

- Security challenges abound
 - Lots of threats and attacks
- CIA properties are desirable policies
- Encryption and decryption
- Shared key vs public/private key systems
- Implementing authentication, signatures, certificates
- Authorization



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

INTRODUCTION TO CLOUDS

Lecture A

WHY CLOUDS?

THE HYPE!

- Gartner in 2009 – Cloud computing revenue will soar faster than expected and will exceed \$150 billion by 2013. It will represent 19% of IT spending by 2015.
- IDC in 2009: “Spending on IT cloud services will triple in the next 5 years, reaching \$42 billion.”
- Forrester in 2010 – Cloud computing will go from \$40.7 billion in 2010 to \$241 billion in 2020.
- Companies and even federal/state governments using cloud computing now: fdbizopps.gov



MANY CLOUD PROVIDERS

- AWS: Amazon Web Services
 - EC2: Elastic Compute Cloud
 - S3: Simple Storage Service
 - EBS: Elastic Block Storage
- Microsoft Azure
- Google Compute Engine
- Rightscale, Salesforce, EMC, Gigaspaces, 10gen, Datastax, Oracle, VMWare, Yahoo, Cloudera
- And many, many more!



Two CATEGORIES OF CLOUDS

- Can be either a (i) public cloud, or (ii) private cloud
- Private clouds are accessible only to company employees
- Public clouds provide service to any paying customer:
 - Amazon S3 (Simple Storage Service): store arbitrary datasets, pay per GB-month stored
 - Amazon EC2 (Elastic Compute Cloud): upload and run arbitrary OS images, pay per CPU hour used
 - Google App Engine/Compute Engine: develop applications within their App Engine framework, upload data that will be imported into their format, and run



CUSTOMERS SAVE TIME AND \$\$\$

- Dave Power, Associate Information Consultant at Eli Lilly and Company: "With AWS, a new server can be up and running in three minutes (it used to take Eli Lilly seven and a half weeks to deploy a server internally) and a 64-node Linux cluster can be online in five minutes (compared with three months internally).
... It's just shy of instantaneous."
- Ingo Elfering, Vice President of Information Technology Strategy, GlaxoSmithKline: "With Online Services, we are able to reduce our IT operational costs by roughly 30% of what we're spending."
- Jim Swartz, CIO, Sybase: "At Sybase, a private cloud of virtual servers inside its datacenter has saved nearly \$US2 million annually since 2006, because the company can share computing power and storage resources across servers."
- Hundreds of startups in Silicon Valley can harness large computing resources without buying their own machines.



BUT WHAT EXACTLY IS A CLOUD?

- Next lecture!





CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

INTRODUCTION TO CLOUDS

Lecture B

WHAT IS A CLOUD?

WHAT IS A CLOUD?

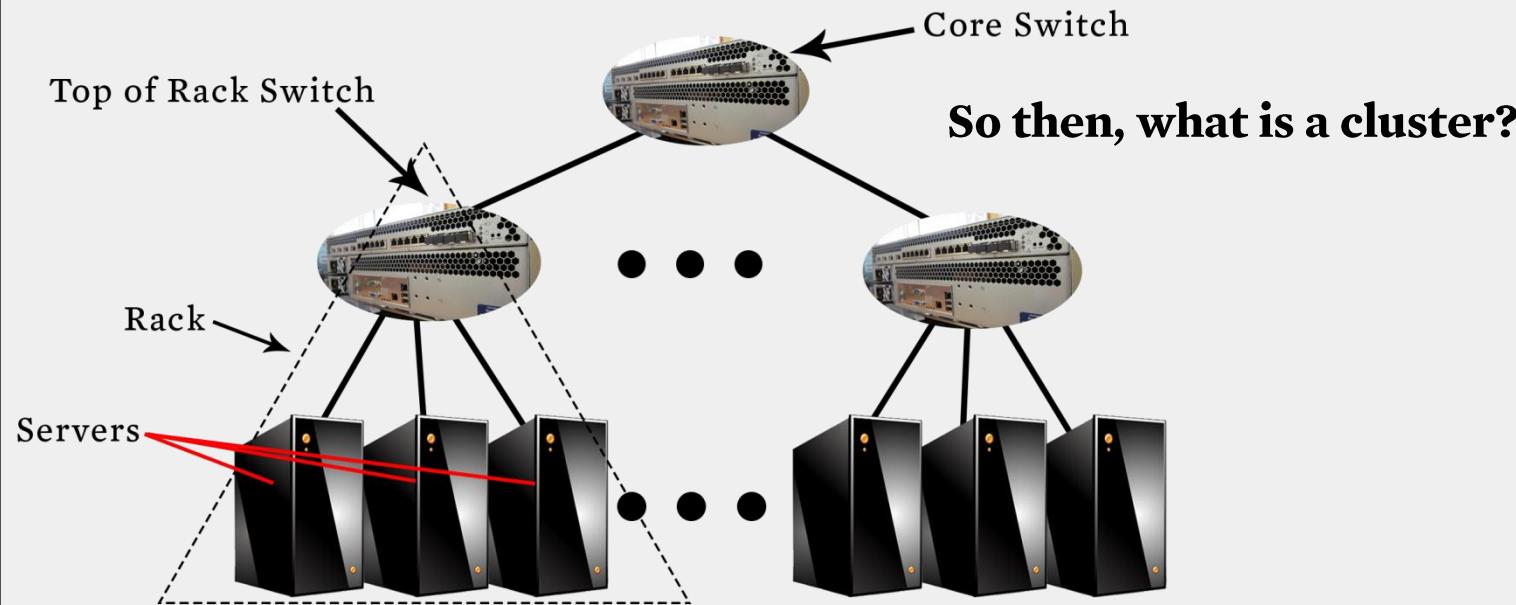
- It's a cluster!
- It's a supercomputer!
- It's a datastore!
- It's Superman!
- None of the above
- All of the above
- Cloud = **Lots of storage + compute cycles nearby**



WHAT IS A CLOUD?

- A single-site cloud (aka “datacenter”) consists of
 - Compute nodes (grouped into racks)
 - Switches, connecting the racks
 - A network topology, e.g., hierarchical
 - Storage (backend) nodes connected to the network
 - Front-end for submitting jobs and receiving client requests
 - Software services
- A geographically distributed cloud consists of
 - Multiple such sites
 - Each site perhaps with a different structure and services

A SAMPLE CLOUD TOPOLOGY





CLOUD COMPUTING CONCEPTS

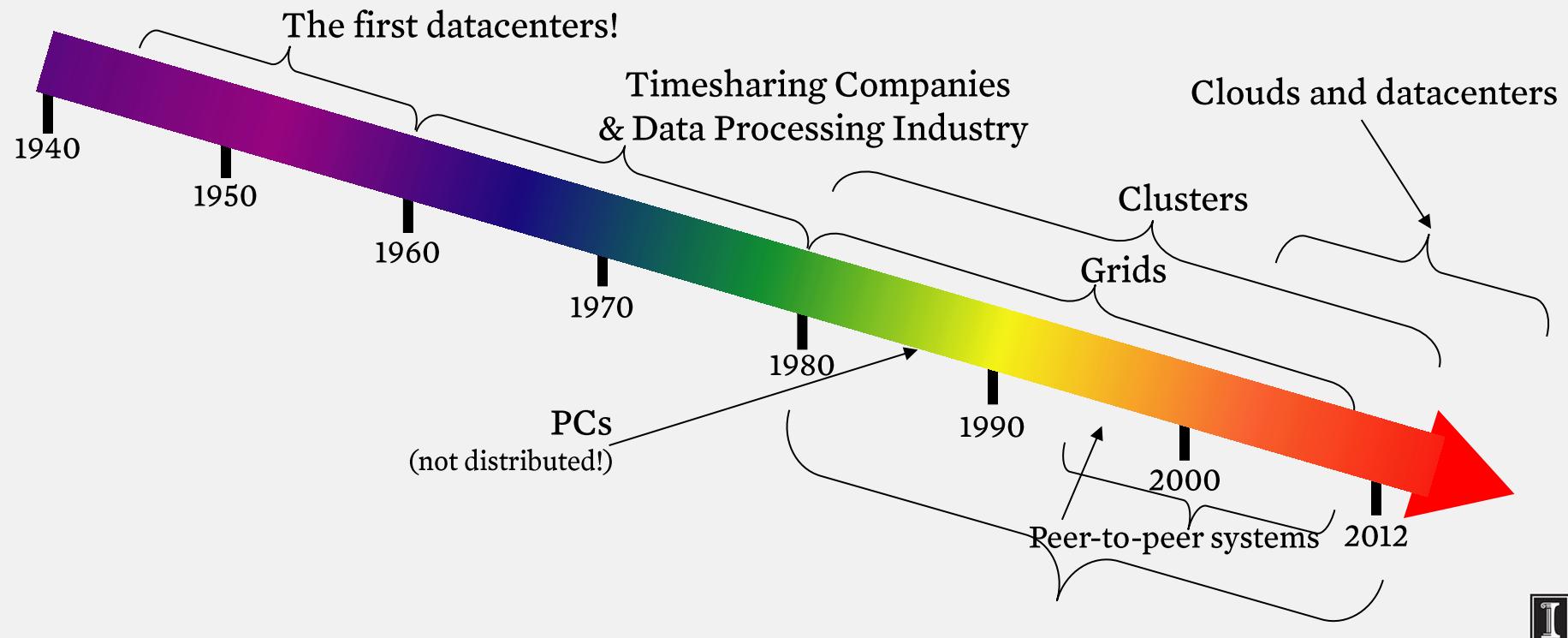
with Indranil Gupta (Indy)

INTRODUCTION TO CLOUDS

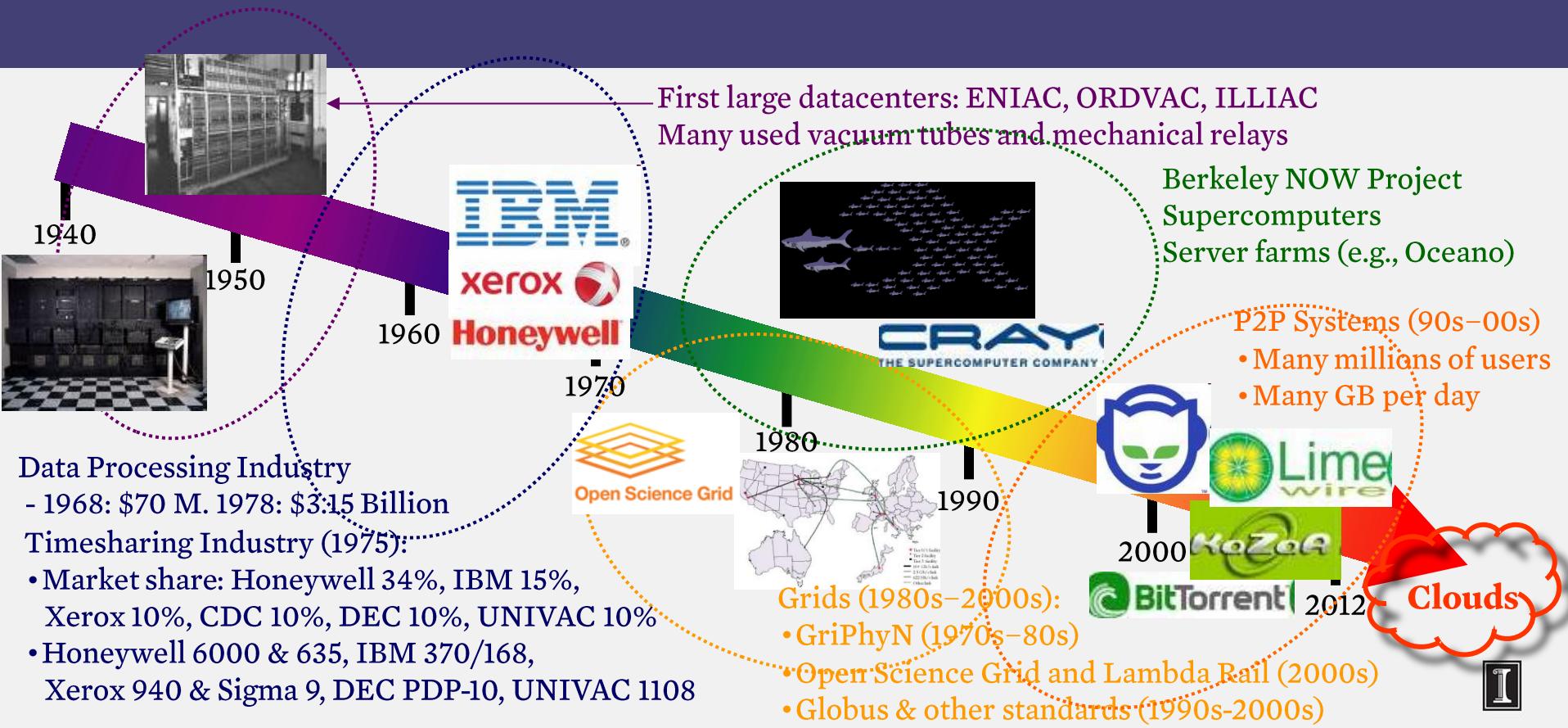
Lecture C

HISTORY

"A CLOUDY HISTORY OF TIME"



"A CLOUDY HISTORY OF TIME"



TRENDS: TECHNOLOGY

- Doubling periods – storage: 12 months, bandwidth: 9 months, and (what law is this?) CPU compute capacity: 18 months
- Then and Now
 - Bandwidth
 - 1985: mostly 56Kbps links nationwide
 - 2012: Tbps links widespread
 - Disk capacity
 - Today's PCs have TBs, far more than a 1990 supercomputer



TRENDS: USERS

- Then and Now
 - Biologists:
 - 1990: were running small single-molecule simulations
 - 2012: CERN's Large Hadron Collider producing many PB/year



PROPHECIES

- In 1965, MIT's Fernando Corbató and the other designers of the Multics operating system envisioned a computer facility operating "like a power company or water company."
- **Plug** your thin client into the computing utility **and play** your favorite Intensive Compute & Communicate Application
 - Have today's clouds brought us closer to this reality? Think about it.





CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

INTRODUCTION TO CLOUDS

Lecture D

WHAT'S NEW IN
TODAY'S CLOUDS

FOUR FEATURES NEW IN TODAY'S CLOUDS

- I. **Massive scale.**
- II. **On-demand access:** Pay-as-you-go, no upfront commitment.
 - Anyone can access it
- III. **Data-intensive Nature:** What was MBs has now become TBs, PBs and XB.s.
 - Daily logs, forensics, Web data, etc.
 - Humans have data numbness: Wikipedia (large) compress is only about 10 GB!
- IV. **New Cloud Programming Paradigms:** MapReduce/Hadoop, NoSQL/Cassandra/MongoDB and many others.
 - High in accessibility and ease of programmability
 - Lots of open-source

Combination of one or more of these gives rise to novel and unsolved distributed computing problems in cloud computing.



I. MASSIVE SCALE

- Facebook [GigaOm, 2012]
 - 30K in 2009 -> 60K in 2010 -> 180K in 2012
- Microsoft [NYTimes, 2008]
 - 150K machines
 - Growth rate of 10K per month
 - 80K total running Bing
- Yahoo! [2009]:
 - 100K
 - Split into clusters of 4000
- AWS EC2 [Randy Bias, 2009]
 - 40,000 machines
 - 8 cores/machine
- eBay [2012]: 50K machines
- HP [2012]: 380K in 180 DCs
- Google: A lot

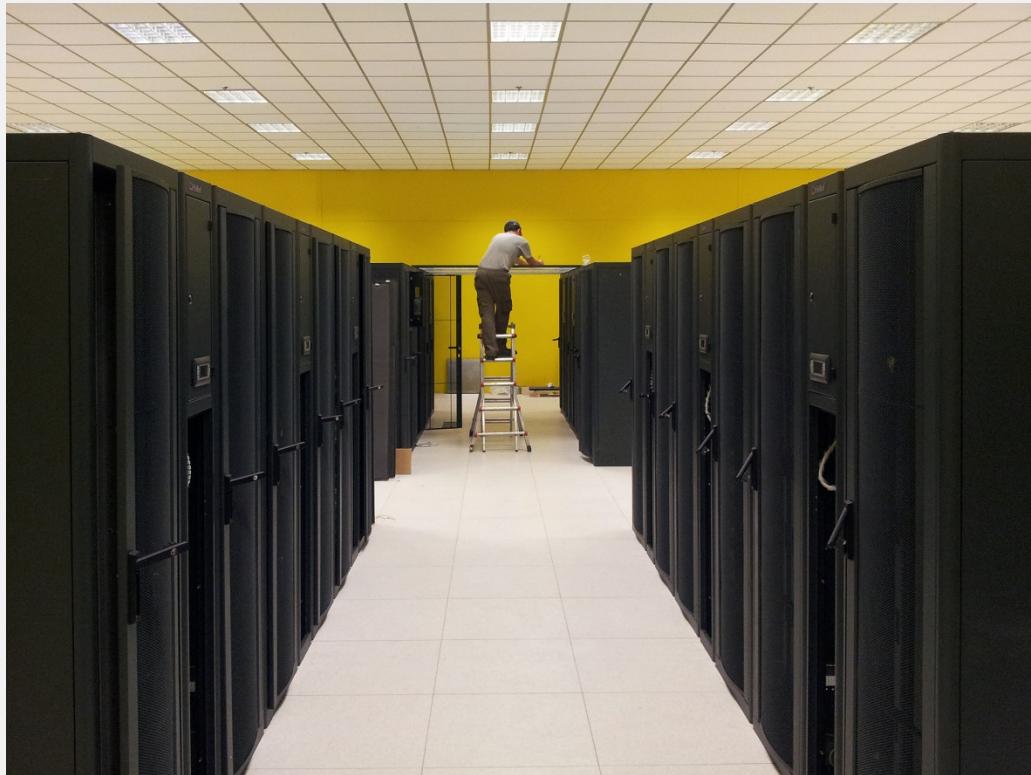


WHAT DOES A DATACENTER LOOK LIKE FROM INSIDE?

- A virtual walk through a datacenter
- Additional reference:
<http://gigaom.com/cleantech/a-rare-look-inside-facebook-s-oregon-data-center-photos-video/>



SERVERS



Front



SERVERS

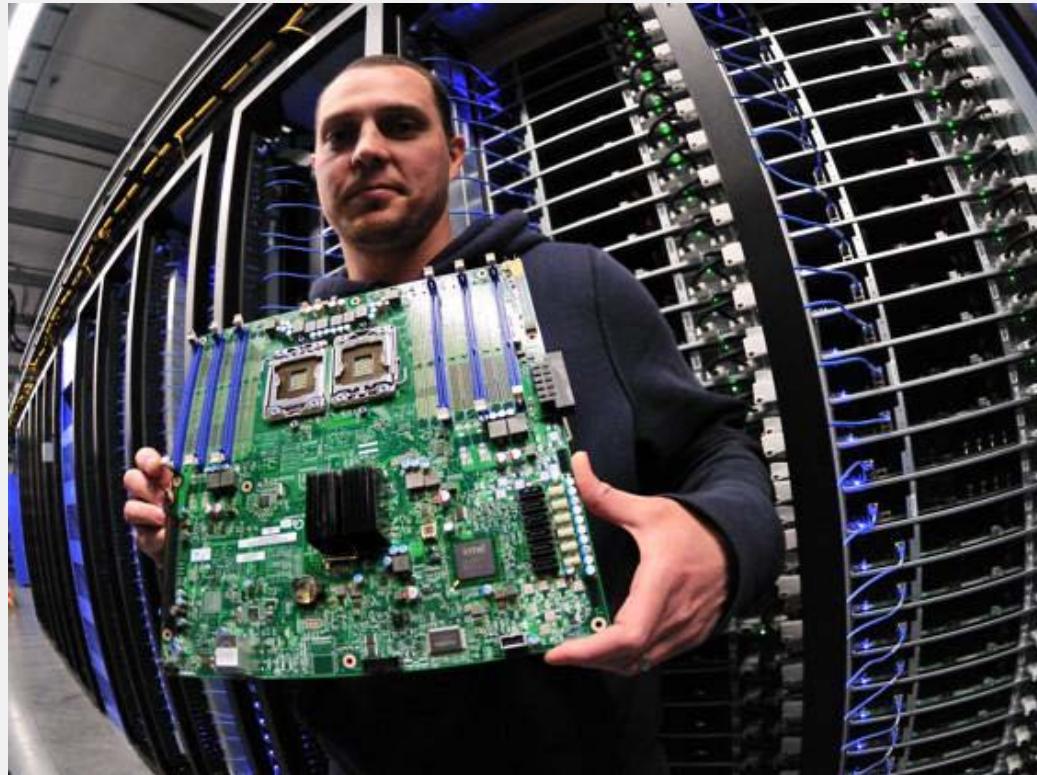


Back

Robert Scoble | CC BY 2.0



SERVERS



Inside



SERVERS



Some highly secure
(e.g., financial info)



POWER



U.S. Army | CC BY 2.0



Greg Goebel | CC BY-SA 2.0



David Goehring | CC BY 2.0

- WUE = Annual Water Usage / IT Equipment Energy (L/kWh) (low is good)
- PUE = Total Facility Power / IT Equipment Power (low is good - e.g., Google = 1.11)

Off-site

On-site



COOLING



Tim Dorr | CC BY-SA 2.0



ChrisDag | CC BY 2.0

- Air sucked in
- Combined with purified water
- Moves cool air through system

EXTRA - FUN VIDEOS TO WATCH

- Microsoft GFS Datacenter Tour (Youtube)
 - <http://www.youtube.com/watch?v=hOxA1llpQIw>
- Timelapse of a Datacenter Construction on the Inside (Fortune 500 company)
 - <http://www.youtube.com/watch?v=ujO-xNvXj3g>





CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

INTRODUCTION TO CLOUDS

Lecture E

NEW ASPECTS OF CLOUDS

II. ON-DEMAND ACCESS: *AAS CLASSIFICATION

On-demand: renting a cab vs. (previously) renting a car, or buying one. Ex.:

- AWS Elastic Compute Cloud (EC2): a few cents to a few \$ per CPU hour
- AWS Simple Storage Service (S3): a few cents to a few \$ per GB-month
- HaaS: Hardware as a Service
 - You get access to barebones hardware machines, do whatever you want with them, ex: your own cluster
 - Not always a good idea because of security risks
- IaaS: Infrastructure as a Service
 - You get access to flexible computing and storage infrastructure. Virtualization is one way of achieving this (what's another way, e.g., using Linux). Often said to subsume HaaS.
 - Ex: Amazon Web Services (AWS: EC2 and S3), Eucalyptus, Rightscale, Microsoft Azure



II. ON-DEMAND ACCESS: *AAS CLASSIFICATION

- PaaS: Platform as a Service
 - You get access to flexible computing and storage infrastructure, coupled with a software platform (often tightly)
 - Ex: Google's AppEngine (Python, Java, Go)
- SaaS: Software as a Service
 - You get access to software services, when you need them. Often said to subsume SOA (Service Oriented Architectures).
 - Ex: Google docs, MS Office on demand



III. DATA-INTENSIVE COMPUTING

- Computation-Intensive Computing
 - Example areas: MPI-based, high-performance computing, grids
 - Typically run on supercomputers (e.g., NCSA Blue Waters)
- Data-Intensive
 - Typically store data at datacenters
 - Use compute nodes nearby
 - Compute nodes run computation services
- In data-intensive computing, the **focus shifts from computation to the data**:
CPU utilization no longer the most important resource metric, instead I/O is (disk and/or network)



IV. NEW CLOUD PROGRAMMING PARADIGMS

- Easy to write and run highly parallel programs in new cloud programming paradigms:
 - Google: MapReduce and Sawzall
 - Amazon: Elastic MapReduce service (pay-as-you-go)
 - Google (MapReduce)
 - Indexing: a chain of 24 MapReduce jobs
 - ~200K jobs processing 50PB/month (in 2006)
 - Yahoo! (Hadoop + Pig)
 - WebMap: a chain of 100 MapReduce jobs
 - 280 TB of data, 2500 nodes, 73 hours
 - Facebook (Hadoop + Hive)
 - ~300TB total, adding 2TB/day (in 2008)
 - 3K jobs processing 55TB/day
 - Similar numbers from other companies, e.g., Yieldex, eharmony.com, etc.
 - NoSQL: MySQL is an industry standard, but Cassandra is 2400 times faster!





CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

INTRODUCTION TO CLOUDS

Lecture F

ECONOMICS OF CLOUDS

Two CATEGORIES OF CLOUDS

- Can be either a (i) public cloud, or (ii) private cloud
- Private clouds are accessible only to company employees
- Public clouds provide service to any paying customer
- You're starting a new service/company: should you use a public cloud or purchase your own private cloud?



SINGLE SITE CLOUD: TO OUTSOURCE OR OWN?

- Medium-sized organization: wishes to run a service for M months
 - Service requires 128 servers (1024 cores) and 524 TB
 - Same as UIUC CCT cloud site
- **Outsource** (e.g., via AWS): monthly cost
 - S3 costs: \$0.12 per GB month. EC2 costs: \$0.10 per CPU hour (costs from 2009)
 - Storage = $\$0.12 \times 524 \times 1000 \sim \62 K
 - Total = Storage + CPUs = $\$62\text{ K} + \$0.10 \times 1024 \times 24 \times 30 \sim \136 K
- **Own:** monthly cost
 - Storage $\sim \$349\text{ K} / M$
 - Total $\sim \$1555\text{ K} / M + 7.5\text{ K}$ (includes 1 sysadmin / 100 nodes)
 - using 0.45:0.4:0.15 split for hardware:power:network and 3 year lifetime of hardware



SINGLE SITE CLOUD: TO OUTSOURCE OR OWN?

- Breakeven analysis: more preferable to own if:
 - $\$349 K / M < \$62 K$ (storage)
 - $\$1555 K / M + 7.5 K < \$136 K$ (overall)
- *Breakeven points*
 - $M > 5.55$ months (storage)
 - $M > 12$ months (overall)
- As a result
 - Startups use clouds a lot
 - Cloud providers benefit monetarily most from storage



SUMMARY

- Clouds build on many previous generations of distributed systems
- Especially the timesharing and data processing industry of the 1960–70s.
- Need to identify unique aspects of a problem to classify it as a new cloud computing problem
 - Scale, On-demand access, data-intensive, new programming
- Otherwise, the solutions to your problem may already exist!





CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CLOUDS ARE
DISTRIBUTED SYSTEMS

Lecture A

A CLOUD IS
A DISTRIBUTED SYSTEM

A CLOUD...

- A cloud consists of
 - Hundreds to thousands of machines in a datacenter (server side)
 - Thousands to millions of machines accessing these services (client side)
- Servers communicate amongst one another
- Clients communicate with servers
- Clients also communicate with each other

A CLOUD... IS A DISTRIBUTED SYSTEM

- Servers communicate amongst one another → Distributed System
 - Essentially a cluster!
- Clients communicate with servers
 - Also a distributed system!
- Clients may also communicate with each other
 - In peer-to-peer systems like BitTorrent
 - Also a distributed system!

FOUR FEATURES OF CLOUDS = ALL DISTRIBUTED SYSTEMS FEATURES!

- I. Massive Scale: many servers
- II. On-demand nature
 - access (multiple) servers anywhere
- III. Data-Intensive Nature
 - lots of data => need a cluster (multiple machines) to store
- IV. New Cloud Programming Paradigms
 - Hadoop/Mapreduce, NoSQL all need clusters

CLOUD = A FANCY WORD FOR A DISTRIBUTED SYSTEM

- A “cloud” is the latest nickname for a distributed system
- Previous nicknames for “distributed system” have included
 - Peer-to-peer systems
 - Grids
 - Clusters
 - Timeshared computers (Data Processing Industry)

(See Lecture Video on History of Clouds)

CLOUD = A FANCY WORD FOR A DISTRIBUTED SYSTEM (2)

- Nicknames come and go, but the core concepts underlying distributed systems stay the same
 - And they are used decade after decade
 - E.g., Lamport Timestamps were invented in the 1970s, and they are used in almost all distributed/cloud systems today
 - This course is about these distributed systems concepts
- A few years from now, there may be a new nickname for distributed systems
 - The core concepts will remain the same, and they will continue to be used in real systems

So **WHAT IS A “DISTRIBUTED SYSTEM”?**

- Let's try to define the term!



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CLOUDS ARE
DISTRIBUTED SYSTEMS

Lecture B

WHAT IS
A DISTRIBUTED SYSTEM?

So **WHAT IS A “DISTRIBUTED SYSTEM”?**

- Let's try to define the term!

WAIT, WAIT...

- Let's first take a step back
- And define the term “Operating System”
- All machines (computers, devices, etc.)
run an Operating System
 - Also called an “OS”
- If you've used a computer, you've used
an Operating System

CAN YOU NAME SOME EXAMPLES OF OPERATING SYSTEMS?

CAN YOU NAME SOME EXAMPLES OF OPERATING SYSTEMS?

- For **big devices**:
 - Linux
 - Mac OS X (Yosemite, Mavericks, ...)
 - Windows (10, 8, 7, Vista, XP, ...)
 - Unix
 - FreeBSD ...
- For **small devices**:
 - Android
 - iOS
 - TinyOS ...

WHAT IS AN OPERATING SYSTEM?

WHAT IS AN OPERATING SYSTEM?

- User interface to hardware (device drivers)
- Provides abstractions (processes, file system)
- Resource manager (scheduler)
- Means of communication (networking)
- ...

FOLDOC DEFINITION (FREE ONLINE DICTIONARY OF COMPUTING)

- The low-level software which handles the interface to peripheral hardware, schedules tasks, allocates storage, and presents a default interface to the user when no application program is running.
- The OS may be split into a kernel which is always present and various system programs which use facilities provided by the kernel to perform higher-level house-keeping tasks, often acting as servers in a client-server relationship.
- Some would include a graphical user interface and window system as part of the OS, others would not. The operating system loader, BIOS, or other firmware required at boot time or when installing the operating system would generally not be considered part of the operating system, though this distinction is unclear in the case of a roamable operating system such as RISC OS.
- The facilities an operating system provides and its general design philosophy exert an extremely strong influence on programming style and on the technical cultures that grow up around the machines on which it runs.

**LET'S REPEAT THE SAME EXERCISE FOR THE
TERM "DISTRIBUTED SYSTEM"**

CAN YOU NAME SOME EXAMPLES OF DISTRIBUTED SYSTEMS?

CAN YOU NAME SOME EXAMPLES OF DISTRIBUTED SYSTEMS?

- Client communicating with a server
- BitTorrent (peer to peer overlay)
- The Internet
- The Web (servers and clients)
- Hadoop
- Datacenters

NOT DISTRIBUTED SYSTEMS

- The following are NOT distributed systems
 - Humans Interacting with each other
 - A standalone machine not connected to the network, and with only one process running on it

Is THERE A GOOD DEFINITION OUT THERE?

DEFINITION FROM FOLDOC (FREE ONLINE DICTIONARY OF COMPUTING)

A collection of (probably heterogeneous) automata whose distribution is transparent to the user so that the system appears as one local machine. This is in contrast to a network, where the user is aware that there are several machines, and their location, storage replication, load balancing and functionality is not transparent. Distributed systems usually use some kind of client-server organization.

(Definition last updated in 1994)

THE FOLDOC DEFINITION FOR DISTRIBUTED SYSTEM IS INCORRECT

A collection of (probably heterogeneous) automata whose distribution is transparent to the user so that the system appears as one local machine. This is in contrast to a network, where the user is aware that there are several machines, and their location, storage replication, load balancing and functionality is not transparent. Distributed systems usually use some kind of client-server organization.

- However, it's never the case that all sites on the Web are all up or all down, e.g., site A may be up while B down; later B is up while A is down.
Yet, the Web is a distributed system!
- Peer to peer systems like BitTorrent rely largely on only clients, no servers.

DEFINITIONS FROM TEXTBOOKS

- A distributed system is a collection of independent computers that appear to the users of the system as a single computer.

[Andrew Tanenbaum]

- A distributed system is several computers doing something together. Thus, a distributed system has three primary characteristics: multiple computers, interconnections, and shared state.

[Michael Schroeder]

DEFINITIONS LOOK UNSATISFACTORY TO US

- Why are these definitions short?
- Why do these definitions look inadequate to us?
- Because we are interested in the insides of a distributed system
 - algorithmics
 - design and implementation
 - maintenance
 - study

EASY TO KNOW WHEN YOU SEE IT

- The following are NOT distributed systems
 - Humans Interacting with each other
 - A standalone machine not connected to the network, and with only one process running on it

EASY TO KNOW WHEN YOU SEE IT

I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so.
But I know it when I see it...

[Potter Stewart, Associate Justice, US Supreme Court (talking about his interpretation of a technical term laid down in the law, case Jacobellis versus Ohio 1964)]

A WORKING DEFINITION OF "DISTRIBUTED SYSTEM"

*A distributed system is a collection of entities, each of which is **autonomous**, **programmable**, **asynchronous** and **failure-prone**, and which communicate through an **unreliable** communication medium.*

A WORKING DEFINITION OF "DISTRIBUTED SYSTEM"

*A distributed system is a collection of entities, each of which is **autonomous**, **programmable**, **asynchronous** and **failure-prone**, and which communicate through an **unreliable** communication medium.*

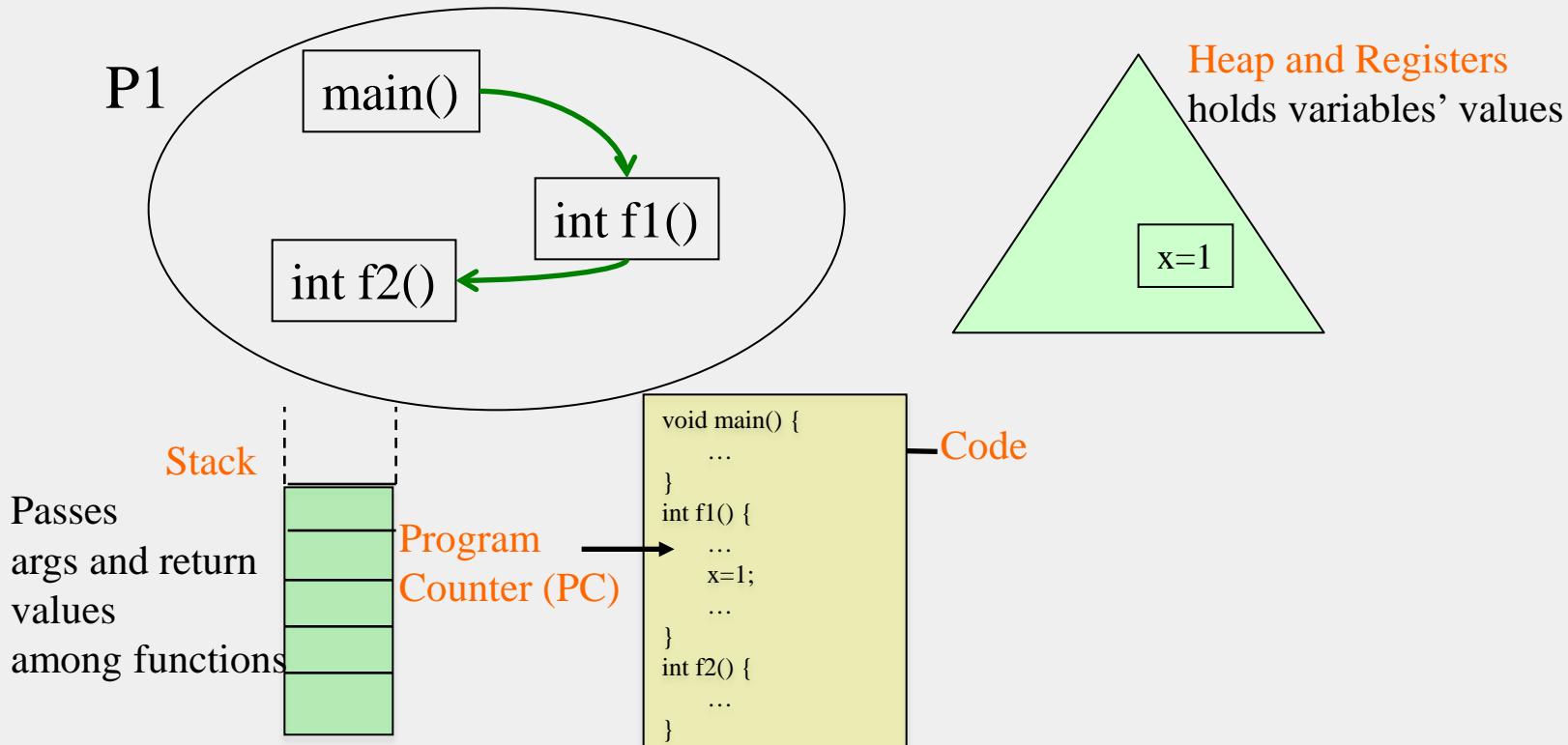
- Our interest in distributed systems involves
 - algorithmics, design and implementation, maintenance, study
- Entity=a process on a device (PC, PDA, mote)
- Communication Medium=Wired or wireless network

A WORKING DEFINITION OF "DISTRIBUTED SYSTEM"

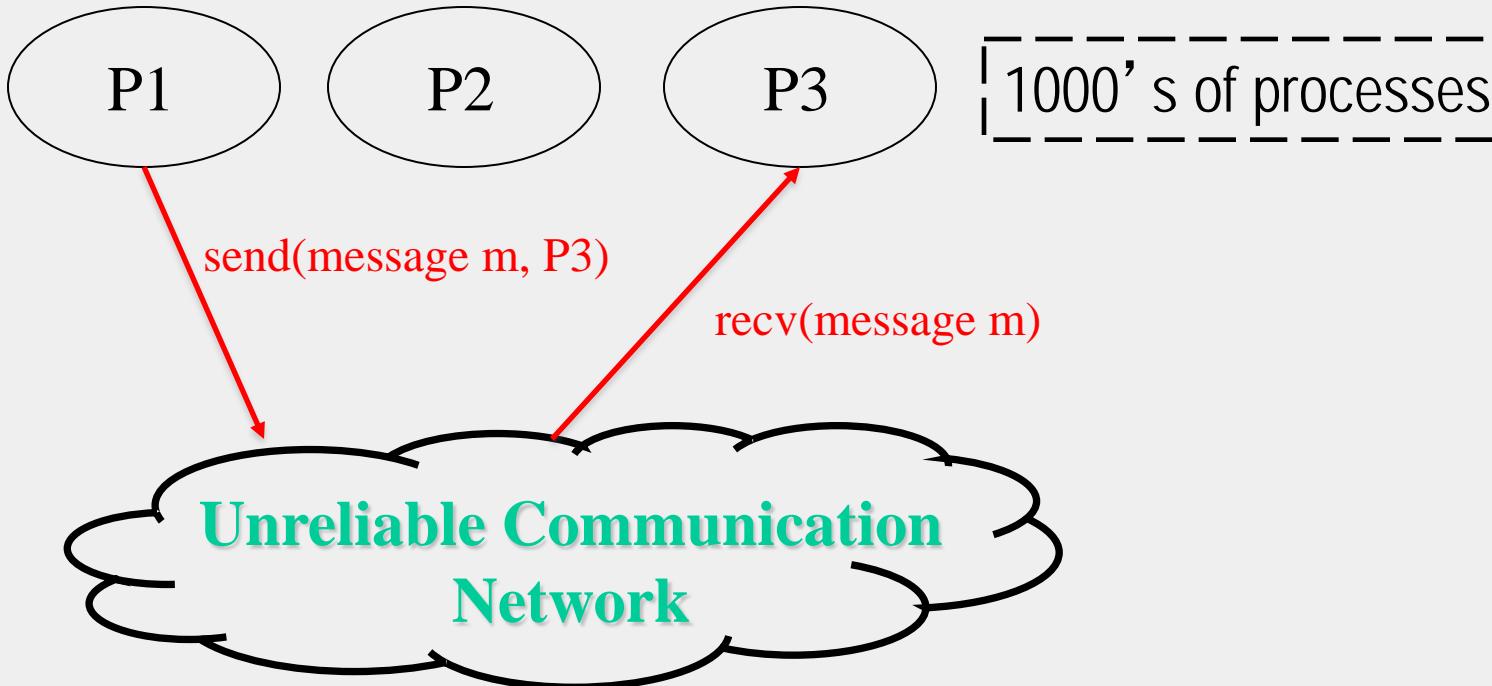
A distributed system is a collection of entities, each of which is autonomous, programmable, asynchronous and failure-prone, and which communicate through an unreliable communication medium.

- Eliminates “Humans Interacting with each other”
- Distinguishes distributed systems from parallel systems (e.g., multiprocessor systems)

REMEMBER A PROCESS? (ORIENTATION LECTURE)



DISTRIBUTED SYSTEM = MANY PROCESSES SENDING AND RECEIVING MESSAGES



NOT MEANT TO BE A PERFECT DEFINITION

- That's only a working definition,
good for this course
- Feel free to come up with your
own definition for distributed
systems!
- Try the exercise after you've seen
the many different examples of
distributed systems in this course!

A RANGE OF INTERESTING PROBLEMS FOR DISTRIBUTED SYSTEM DESIGNERS

- P2P systems [Gnutella, Kazaa, BitTorrent]
- Cloud Infrastructures [AWS, Azure, Google Cloud]
- Cloud Storage [Key-value stores, NoSQL, Cassandra]
- Cloud Programming [MapReduce, Storm, Pregel]
- Coordination [Paxos, Leader Election, Snapshots]
- Managing Many Clients and Servers Concurrently
[Concurrency Control, Replication Control]
- (and many more that you'll see in this course!)

IN SOLVING THESE PROBLEMS, MANY CHALLENGES ABOUND...

- **Failures**: no longer the exception, but rather a norm
- **Scalability**: 1000s of machines, Terabytes of data
- **Asynchrony**: clock skew and clock drift
- **Concurrency**: 1000s of machines interacting with each other accessing the same data
- ...

LOOKING FORWARD

- Over the next few weeks, we will see several core concepts of distributed systems
 - Gossip
 - Membership
 - Distributed Hash Tables (DHTs)
- ... and alternate this with their use in real systems
 - Peer-to-peer systems (which use DHTs)
 - Key-value/NoSQL stores (which use DHTs, gossip, membership)

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MAPREDUCE

Lecture A

MAPREDUCE PARADIGM

WHAT IS MAPREDUCE?

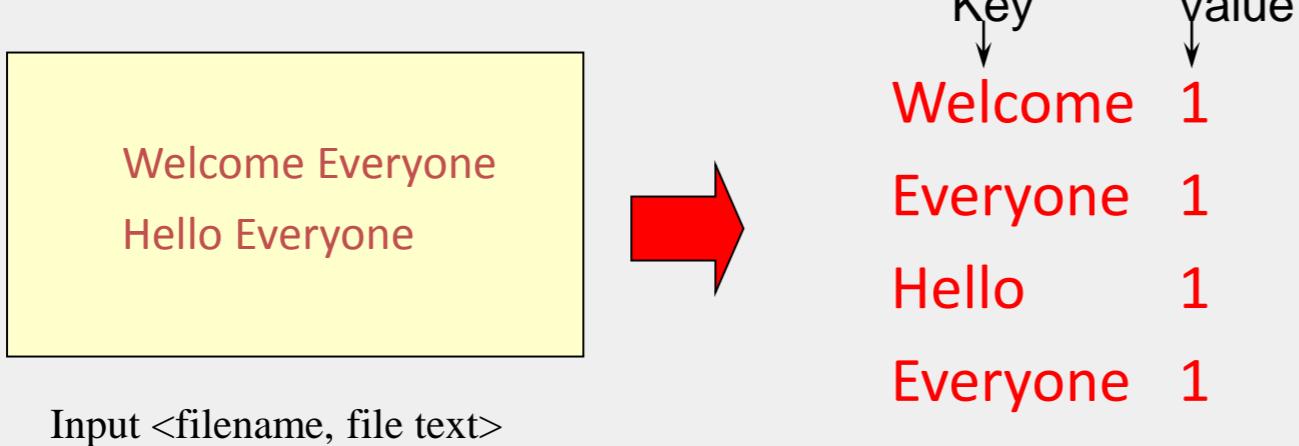
- Terms are borrowed from functional language (e.g., Lisp)

Sum of squares:

- `(map square '(1 2 3 4))`
 - Output: `(1 4 9 16)`
[processes each record sequentially and independently]
- `(reduce + '(1 4 9 16))`
 - `(+ 16 (+ 9 (+ 4 1)))`
 - Output: 30
[processes set of all records in batches]
- Let's consider a sample application: [WordCount](#)
 - You are given a huge dataset (e.g., Wikipedia dump or all of Shakespeare's works) and asked to list the count for each of the words in each of the documents therein.

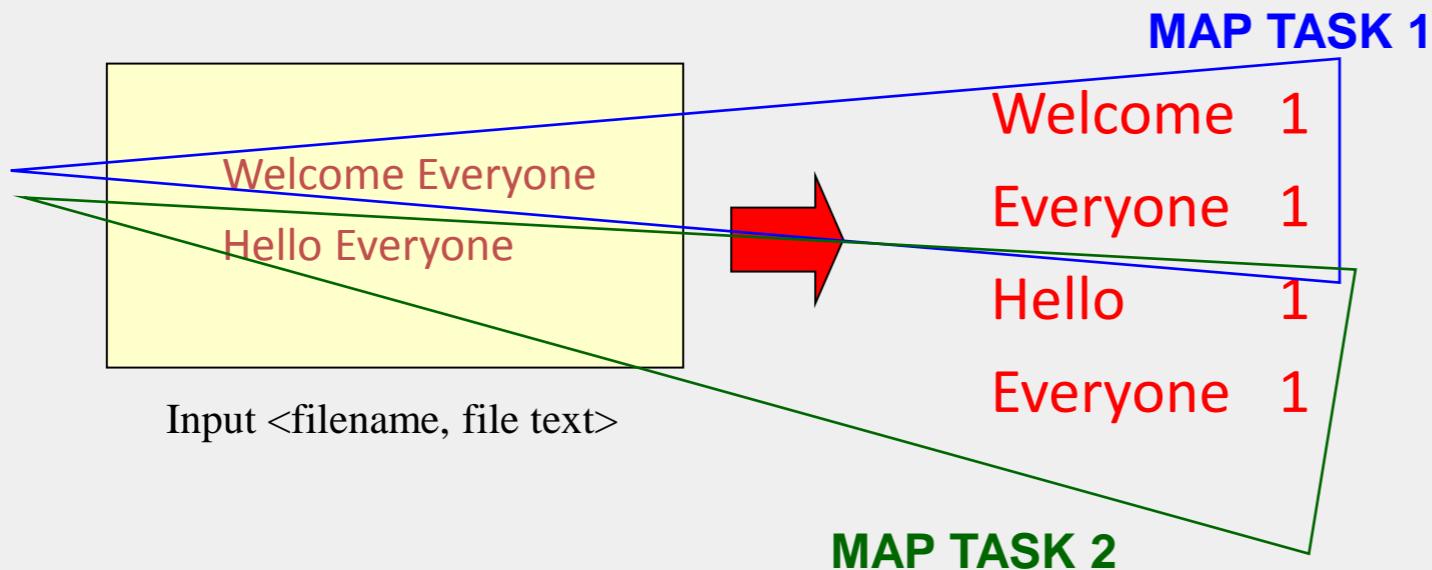
MAP

- Process individual records to generate intermediate key/value pairs.



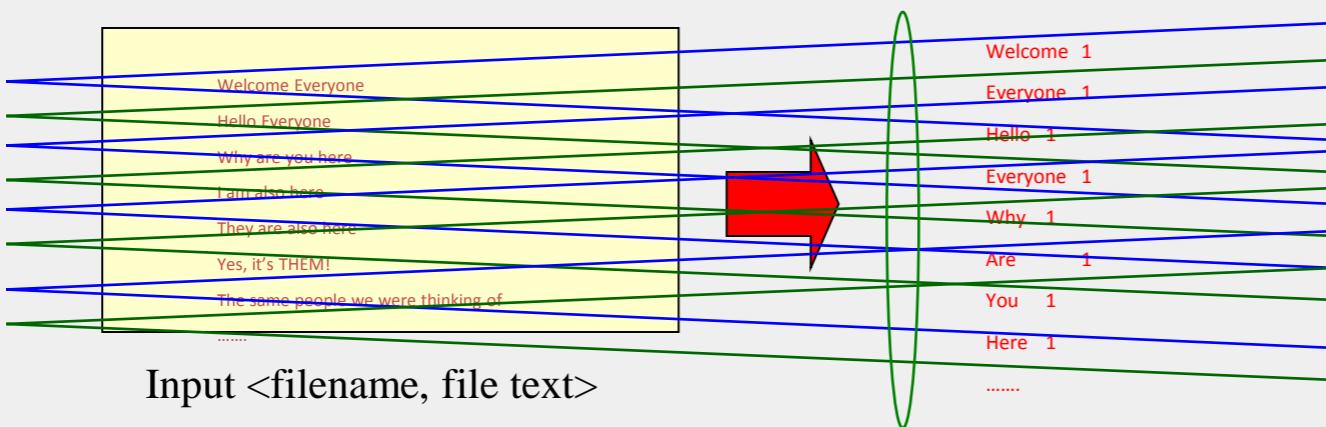
MAP

- Parallelly process individual records to generate intermediate key/value pairs.



MAP

- Parallelly process a large number of individual records to generate intermediate key/value pairs.



MAP TASKS

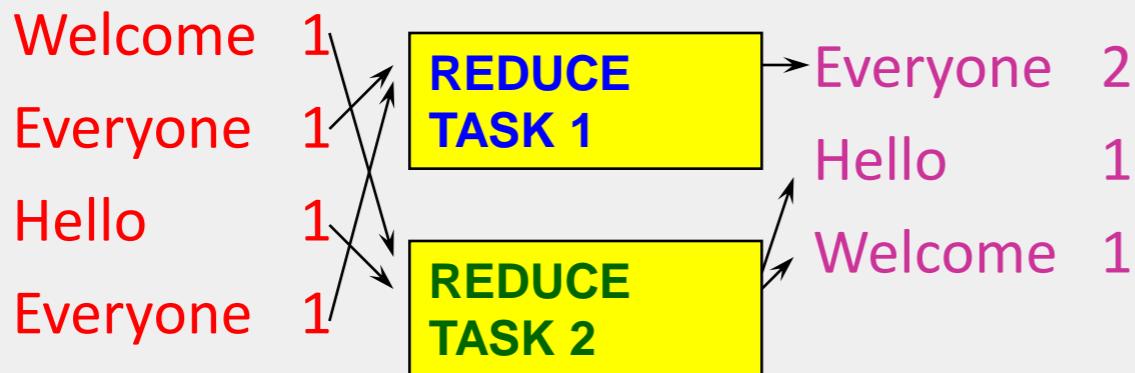
REDUCE

- Reduce processes and merges all intermediate values associated per key



REDUCE

- Each key assigned to one Reduce
- Parallelly processes and merges all intermediate values by partitioning keys



- Popular: *hash partitioning*, i.e., key is assigned to reduce # = $\text{hash}(\text{key}) \% \text{number of reduce servers}$

HADOOP CODE - MAP

```
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text,
    IntWritable> {
    private final static IntWritable one =
        new IntWritable(1);
    private Text word = new Text();

    public void map( LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter)
        throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

HADOOP CODE - REDUCE

```
public static class ReduceClass extends MapReduceBase
    implements Reducer<Text, IntWritable, Text,
    IntWritable> {
    public void reduce(
        Text key,
        Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter)
    throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

HADOOP CODE - DRIVER

```
// Tells Hadoop how to run your Map-Reduce job
public void run (String inputPath, String outputPath)
    throws Exception {
    // The job. WordCount contains MapClass and Reduce.
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("mywordcount");
    // The keys are words
    (strings) conf.setOutputKeyClass(Text.class);
    // The values are counts (ints)
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(MapClass.class);
    conf.setReducerClass(ReduceClass.class);
    FileInputFormat.addInputPath(
        conf, newPath(inputPath));
    FileOutputFormat.setOutputPath(
        conf, new Path(outputPath));
    JobClient.runJob(conf);
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MAPREDUCE

Lecture B

MAPREDUCE EXAMPLES

SOME APPLICATIONS OF MAPREDUCE

Distributed Grep:

- Input: large set of files
- Output: lines that match pattern
- Map – *Emits a line if it matches the supplied pattern*
- Reduce – *Copies the intermediate data to output*

SOME APPLICATIONS OF MAPREDUCE (2)

Reverse Web-Link Graph

- Input: Web graph: tuples (a, b) where (page $a \rightarrow$ page b)
- Output: For each page, list of pages that link *to* it
- Map – *process web log and for each input $\langle source, target \rangle$, it outputs $\langle target, source \rangle$*
- Reduce – *emits $\langle target, list(source) \rangle$*

SOME APPLICATIONS OF MAPREDUCE (3)

Count of URL access frequency

- Input: Log of accessed URLs, e.g., from proxy server
- Output: For each URL, % of total accesses for that URL

- Map – *Process web log and outputs <URL, 1>*
- Multiple reducers – *Emits <URL, URL_count>*

(So far, like WordCount. But still need %)

- Chain another MapReduce job after above one
- Map – *Processes <URL, URL_count> and outputs <1, (<URL, URL_count>)>*
- 1 Reducer – Sums up *URL_count's* to calculate overall_count.

Emits multiple <URL, URL_count/overall_count>

SOME APPLICATIONS OF MAPREDUCE (4)

Map task's output is sorted (e.g., quicksort)

Reduce task's input is sorted (e.g., mergesort)

Sort

- Input: Series of (key, value) pairs
- Output: Sorted <value>s
- Map – $\langle \text{key}, \text{value} \rangle \rightarrow \langle \text{value}, _ \rangle$ (identity)
- Reducer – $\langle \text{key}, \text{value} \rangle \rightarrow \langle \text{key}, \text{value} \rangle$ (identity)
- Partitioning function – partition keys across reducers based on ranges
 - Take data distribution into account to balance reducer tasks



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MAPREDUCE

Lecture C

MAPREDUCE SCHEDULING

PROGRAMMING MAPREDUCE

Externally: For **user**

1. Write a Map program (short), write a Reduce program (short)
2. Submit job; wait for result
3. Need to know nothing about parallel/distributed programming!

Internally: For the Paradigm and Scheduler

1. Parallelize Map
2. Transfer data from Map to Reduce
3. Parallelize Reduce
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output

(Ensure that no Reduce starts before all Maps are finished. That is, ensure the **barrier** between the Map phase and Reduce phase)

INSIDE MAPREDUCE

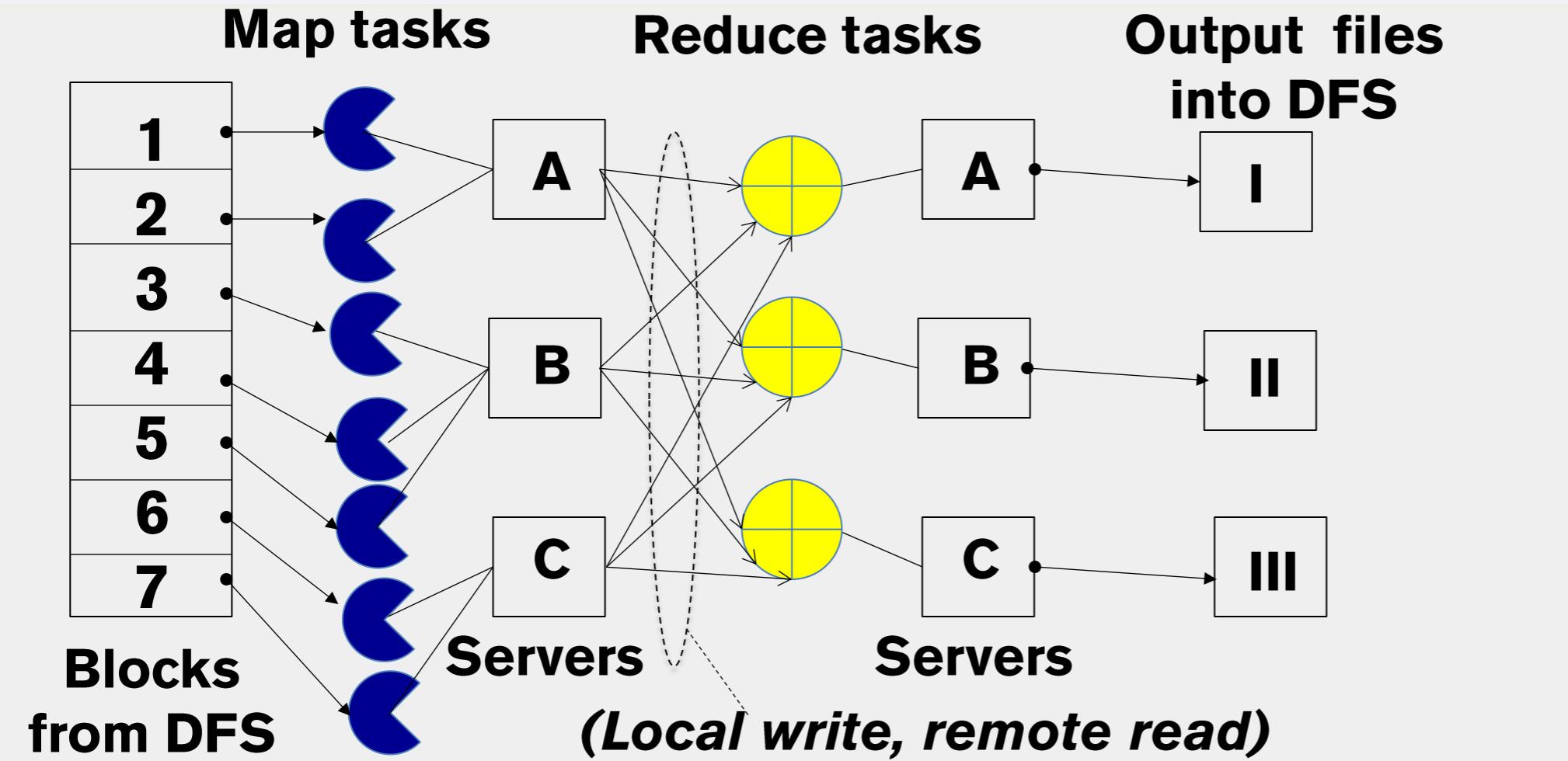
For the cloud:

1. Parallelize Map: **easy!** each map task is independent of the other!
 - All Map output records with same key assigned to same Reduce
2. Transfer data from Map to Reduce:
 - All Map output records with same key assigned to same Reduce task
 - Use **partitioning function, e.g., $\text{hash}(\text{key}) \% \text{number of reducers}$**
3. Parallelize Reduce: **easy!** Each reduce task is independent of the other!
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output
 - Map input: from **distributed file system**
 - Map output: to local disk (at Map node); uses **local file system**
 - Reduce input: from (multiple) remote disks; uses local file systems
 - Reduce output: to distributed file system

local file system = Linux FS, etc.

distributed file system = GFS (Google File System), HDFS (Hadoop Distributed File System)

INTERNAL WORKINGS OF MAPREDUCE



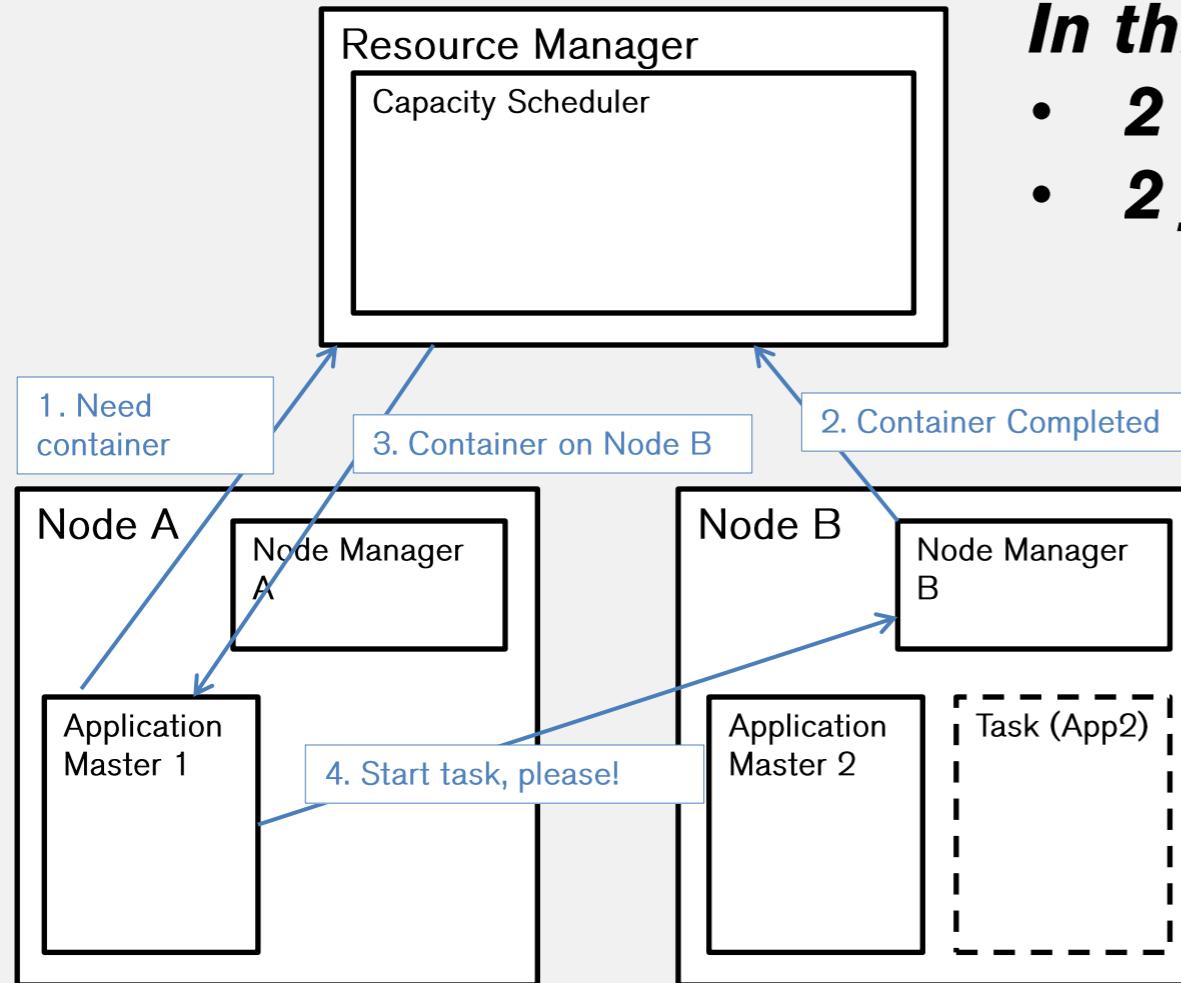
Resource Manager (assigns maps and reduces to servers)

THE YARN SCHEDULER

- Used in Hadoop 2.x +
- YARN = Yet Another Resource Negotiator
- Treats each server as a collection of *containers*
 - Container = some CPU + some memory
- Has 3 main components
 - Global *Resource Manager (RM)*
 - Scheduling
 - Per-server *Node Manager (NM)*
 - Daemon and server-specific functions
 - Per-application (job) *Application Master (AM)*
 - Container negotiation with RM and NMs
 - Detecting task failures of that job



YARN: How a Job Gets a Container



In this figure

- **2 servers (A, B)**
- **2 jobs ($1, 2$)**

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MAPREDUCE

Lecture D

MAPREDUCE FAULT-TOLERANCE

FAULT TOLERANCE

- Server failure
 - NM heartbeats to RM
 - If server fails, RM lets all affected AMs know, and AMs take action
 - NM keeps track of each task running at its server
 - If task fails while in-progress, mark the task as idle and restart it
 - AM heartbeats to RM
 - On failure, RM restarts AM, which then syncs up with its running tasks
- RM failure
 - Use old checkpoints and bring up secondary RM
- Heartbeats also used to piggyback container requests
 - Avoids extra messages

SLOW SERVERS

Stragglers (slow nodes)

- The slowest machine slows the entire job down (why?)
- Due to bad disk, network bandwidth, CPU, or memory
- Keep track of “progress” of each task (% done)
- Perform backup (replicated) execution of straggler task: task considered done when first replica complete. Called **speculative execution**.

LOCALITY

- Locality
 - Since cloud has hierarchical topology (e.g., racks)
 - GFS/HDFS stores 3 replicas of each of chunks (e.g., 64 MB in size)
 - Maybe on different racks, e.g., 2 on a rack, 1 on a different rack
 - MapReduce attempts to schedule a map task on
 - A machine that contains a replica of corresponding input data, or failing that,
 - On the same rack as a machine containing the input, or failing that,
 - Anywhere

MAPREDUCE: SUMMARY

- MapReduce uses parallelization + aggregation to schedule applications across clusters
- Need to deal with failure
- Plenty of ongoing research work in scheduling and fault-tolerance for MapReduce and Hadoop



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

INTRODUCTION TO
PART 1

WHAT THIS COURSE IS ABOUT

- This course is about the internals of cloud computing
 - Not how to use cloud systems or write cloud applications (separate course in Cloud Specialization: *Cloud Applications*)
 - Not about networking (separate course in Cloud Specialization: *Cloud Networking*)
- We'll go underneath the hood and look at **distributed systems** that underlie today's cloud computing technologies

WHAT THIS COURSE IS ABOUT (2)

- We'll discuss
 - Concepts
 - Techniques
 - Industry systems, including open source (from the inside)
- The course is a mix of
 - Distributed systems
 - Distributed algorithms
 - As applied to cloud computing

SYLLABUS FOR PART 1

- Introduction: Clouds, MapReduce, Key-value stores
- Classical precursors: Peer-to-peer systems, Grids
- Widely-used algorithms: Gossip, Membership, Paxos
- Classical algorithms: Time and Ordering, Snapshots, Multicast
- Fun: Interviews with leading managers and researchers, from both industry and academia

EXERCISES

- 5 Homeworks
- 1 Programming Assignment (C++)
 - Implement a membership protocol inside an emulator
- 1 Exam

ONWARD!

- Cloud computing is an exciting area to be studying, very dynamic and continuously changing
- I'm looking forward to working with you!
- Come, let's tour the landscape.



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

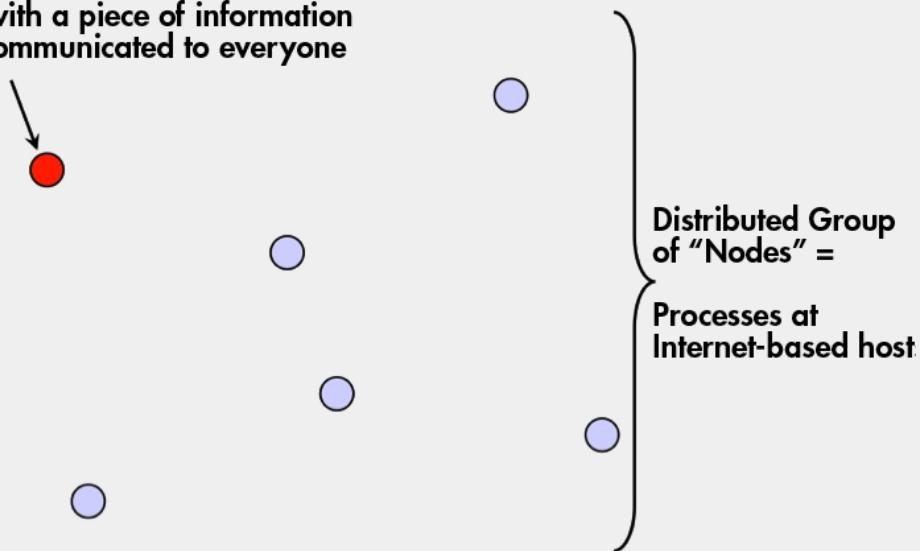
GOSSIP

Lecture A

MULTICAST PROBLEM

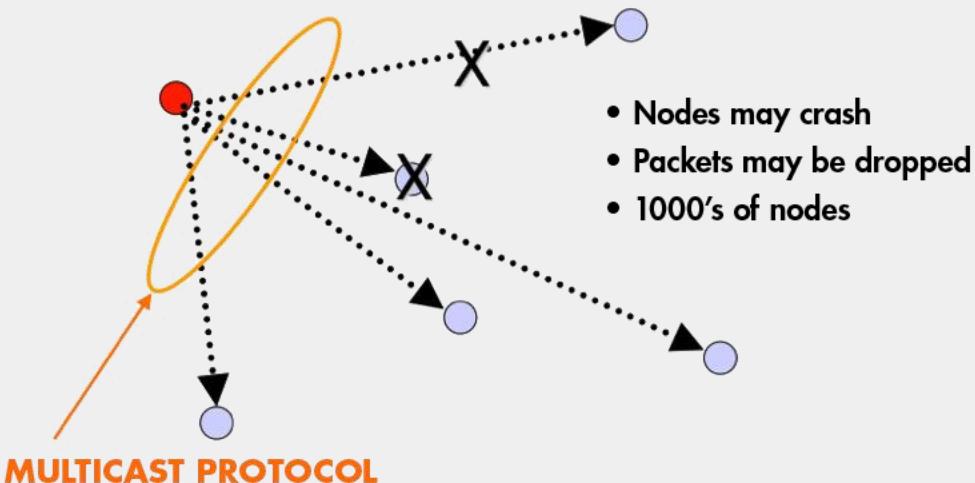
MULTICAST

Node with a piece of information
to be communicated to everyone

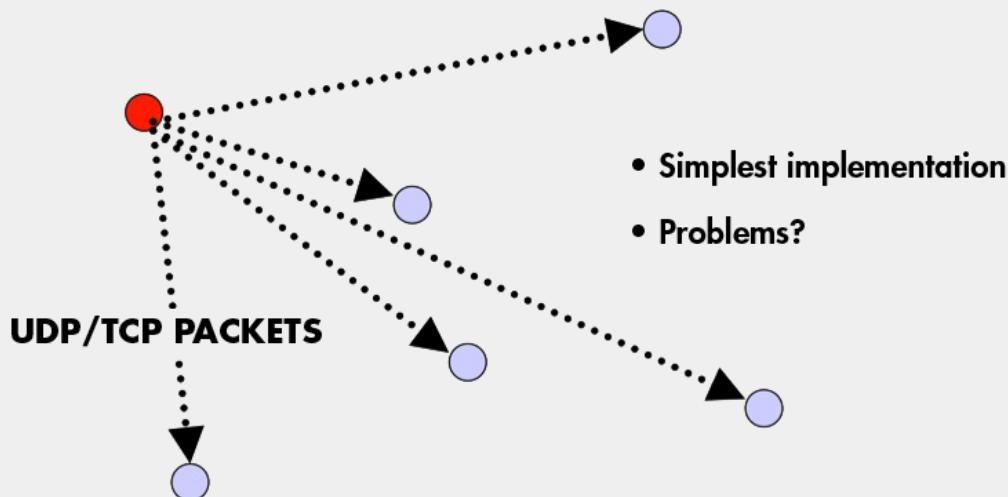


FAULT-TOLERANCE AND SCALABILITY

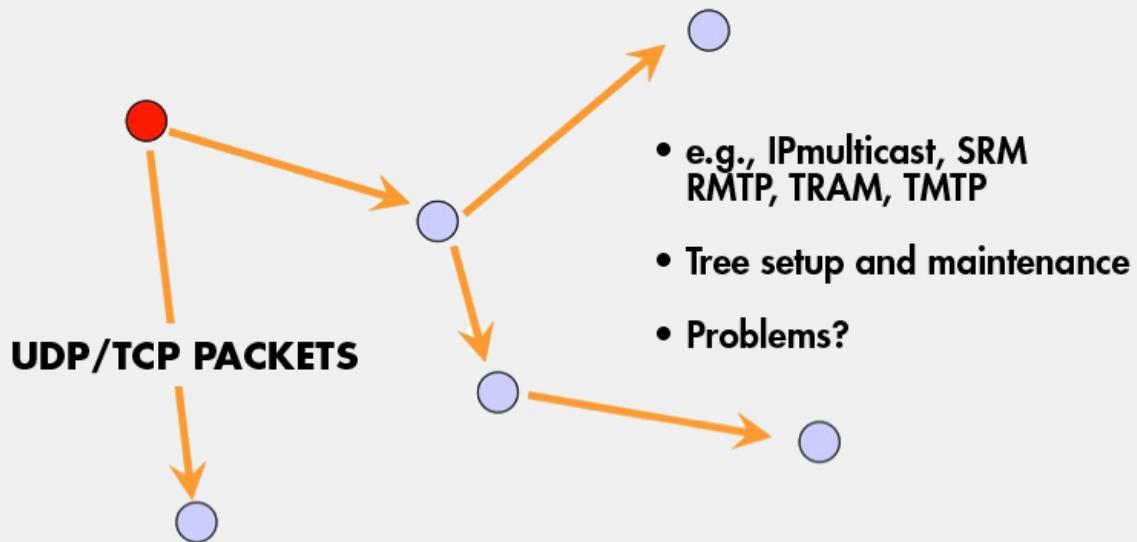
MULTICAST SENDER



CENTRALIZED



TREE-BASED



TREE-BASED MULTICAST PROTOCOLS

- Build a spanning tree among the processes of the multicast group
- Use spanning tree to disseminate multicasts
- Use either acknowledgments (ACKs) or negative acknowledgements (NAKs) to repair multicasts not received
- SRM (Scalable Reliable Multicast)
 - Uses NAKs
 - But adds random delays, and uses exponential backoff to avoid NAK storms
- RMTP (Reliable Multicast Transport Protocol)
 - Uses ACKs
 - But ACKs only sent to designated receivers, which then re-transmit missing multicasts
- These protocols still cause an $O(N)$ ACK/NAK overhead



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

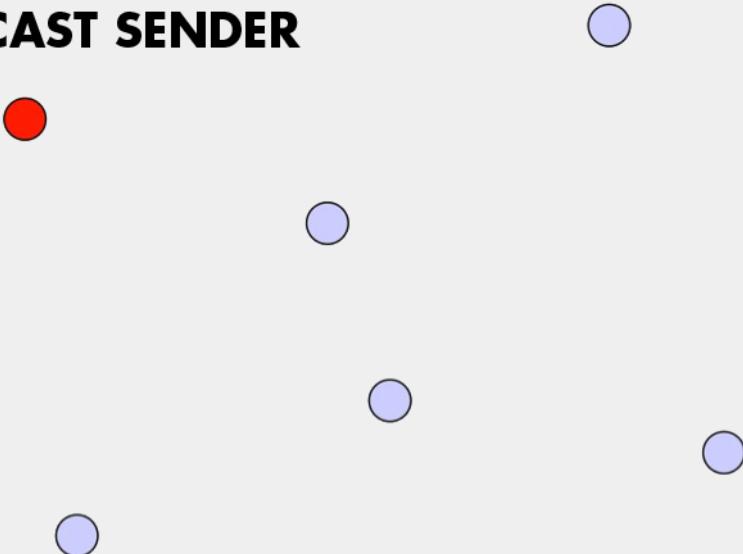
GOSSIP

Lecture B

THE GOSSIP PROTOCOL

A THIRD APPROACH

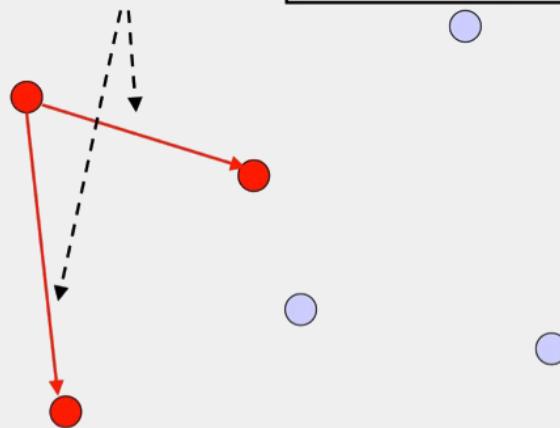
MULTICAST SENDER



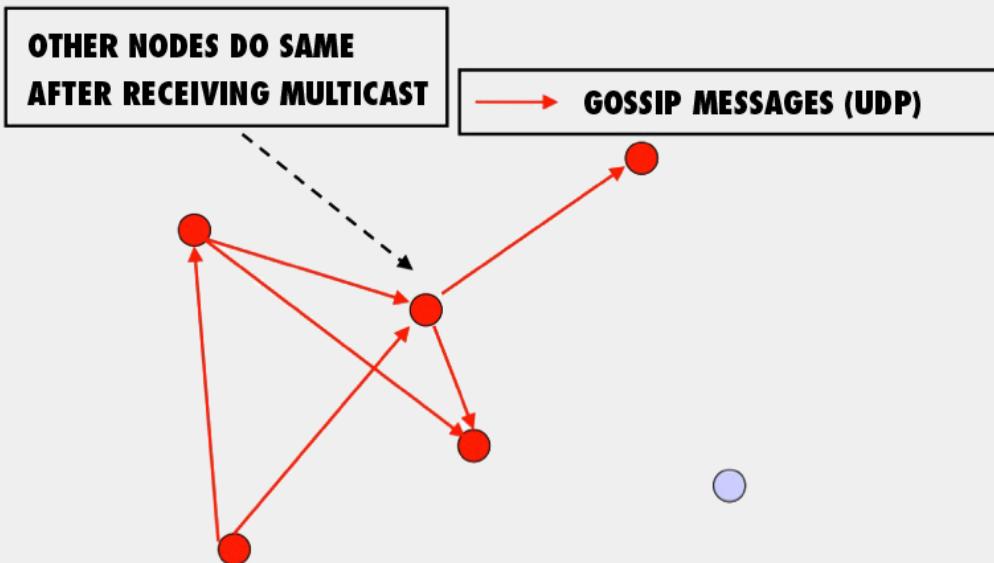
A THIRD APPROACH

**PERIODICALLY, TRANSMIT TO
 b RANDOM TARGETS**

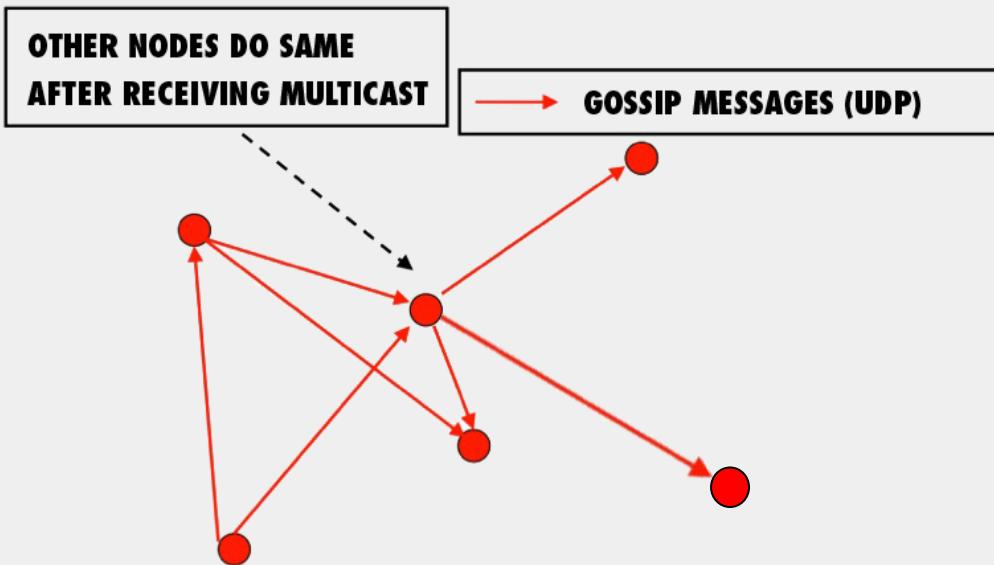
→ GOSSIP MESSAGES (UDP)



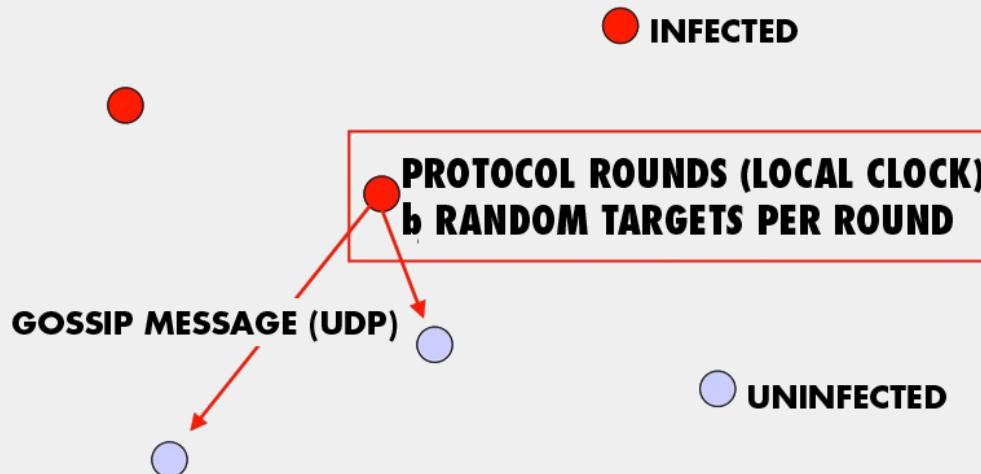
A THIRD APPROACH



A THIRD APPROACH



"EPIDEMIC" MULTICAST (OR "GOSSIP")



PUSH VS. PULL

- So that was “Push” gossip
 - Once you have a multicast message, you start gossiping about it
 - Multiple messages? Gossip a random subset of them, or recently-received ones, or higher priority ones
- There’s also “Pull” gossip
 - Periodically poll a few randomly selected processes for new multicast messages that you haven’t received
 - Get those messages
- Hybrid variant: Push-Pull
 - As the name suggests



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

GOSSIP

Lecture C

GOSSIP ANALYSIS

PROPERTIES

Claim that the simple Push protocol

- Is lightweight in large groups
- Spreads a multicast quickly
- Is highly fault-tolerant

ANALYSIS

From old mathematical branch of *Epidemiology* [Bailey 75]

- Population of $(n+1)$ individuals mixing homogeneously
- Contact rate between any individual pair is β
- At any time, each individual is either uninfected (numbering x) or infected (numbering y)
- Then, $x_0 = n$, $y_0 = 1$
and at all times $x + y = n + 1$
- Infected–uninfected contact turns latter infected, and it stays infected

ANALYSIS (CONTD.)

- Continuous time process
- Then

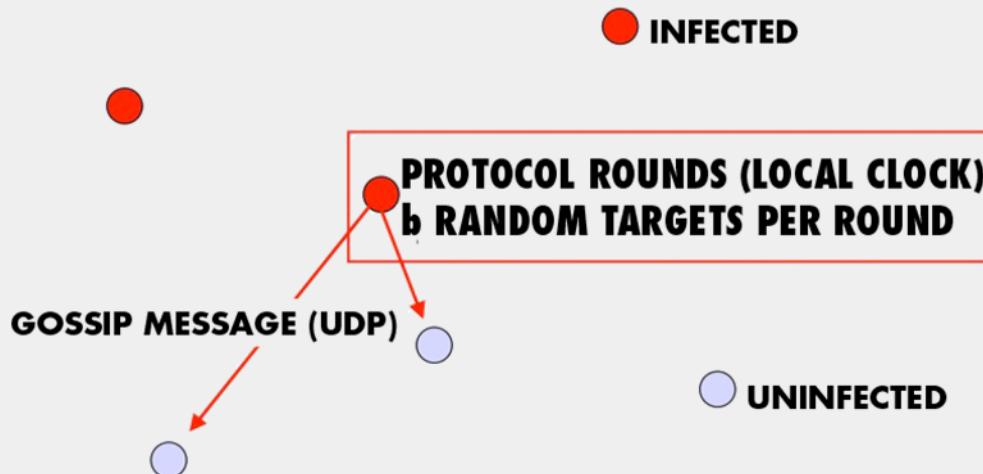
$$\frac{dx}{dt} = -\beta xy \quad (\text{why?})$$

with solution:

$$x = \frac{n(n+1)}{n + e^{\beta(n+1)t}}, \quad y = \frac{(n+1)}{1 + ne^{-\beta(n+1)t}}$$

(can you derive it?)

EPIDEMIC MULTICAST



EPIDEMIC MULTICAST ANALYSIS

$$\beta = \frac{b}{n} \quad (\text{why?})$$

Substituting, at time $t=c\log(n)$, the number of infected is

$$y \approx (n+1) - \frac{1}{n^{cb-2}}$$

(correct? can you derive it?)

ANALYSIS (CONTD.)

- Set c, b to be small numbers independent of n
- Within $c \log(n)$ rounds, **[low latency]**
 - all but $\frac{1}{n^{cb-2}}$ number of nodes receive the multicast
[reliability]
 - each node has transmitted no more than $c b \log(n)$ gossip messages **[lightweight]**

WHY IS LOG(N) LOW?

- Log(N) is not constant in theory
- But pragmatically, it is a very slowly growing number
- Base 2
 - Log(1000) ~ 10
 - Log(1M) ~ 20
 - Log(1B) ~ 30
 - Log(all IPv4 address) = 32

FAULT-TOLERANCE

- Packet loss
 - 50% packet loss: analyze with b replaced with $b/2$
 - To achieve same reliability as 0% packet loss, takes twice as many rounds
- Node failure
 - 50% of nodes fail: analyze with n replaced with $n/2$ and b replaced with $b/2$
 - Same as above

FAULT-TOLERANCE

- With failures, is it possible that the epidemic might die out quickly?
- Possible, but improbable:
 - Once a few nodes are infected, with high probability, the epidemic will not die out
 - So the analysis we saw in the previous slides is actually behavior *with high probability*
- [Galey and Dani 98]
- Think: Why do rumors spread so fast? Why do infectious diseases cascade quickly into epidemics? Why does a virus or worm spread rapidly?

PULL GOSSIP: ANALYSIS

- In all forms of gossip, it takes $O(\log(N))$ rounds before about $N/2$ gets the gossip
 - Why? Because that's the fastest you can spread a message – a spanning tree with fanout (degree) of constant degree has $O(\log(N))$ total nodes
- Thereafter, pull gossip is faster than push gossip
- After the i th, round let p_i be the fraction of non-infected processes. Then
(k =number of gossip pulls per round per process)

$$p_{i+1} = (p_i)^{k+1}$$

- This is super-exponential
- Second half of pull gossip finishes in time $O(\log(\log(N)))$

TOPOLOGY-AWARE GOSSIP

- Network topology is hierarchical

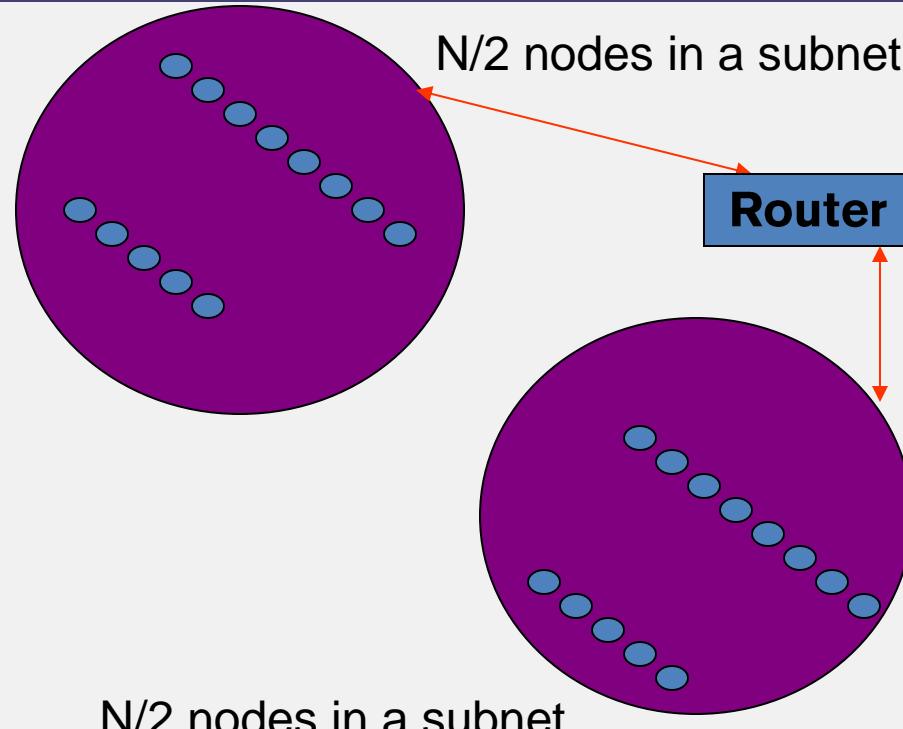
- Random gossip target selection => core routers face $O(N)$ load (Why?)

- Fix: In subnet i , which contains n_i nodes, pick gossip target in your subnet with probability $1/n_i$

- Router load = $O(1)$

- Dissemination time = $O(\log(N))$

- Why?



ANSWER - PUSH ANALYSIS (CONTD.)

Using: $\beta = \frac{b}{n}$

Substituting, at time $t=c\log(n)$

$$\begin{aligned}y &= \frac{n+1}{1 + ne^{-\frac{b}{n}(n+1)c\log(n)}} \approx \frac{n+1}{1 + \frac{1}{n^{cb-1}}} \\&\approx (n+1)\left(1 - \frac{1}{n^{cb-1}}\right) \\&\approx (n+1) - \frac{1}{n^{cb-2}}\end{aligned}$$



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

GOSSIP

Lecture D

GOSSIP IMPLEMENTATIONS

SO, ...

- Is this all theory and a bunch of equations?
- Or are there implementations yet?

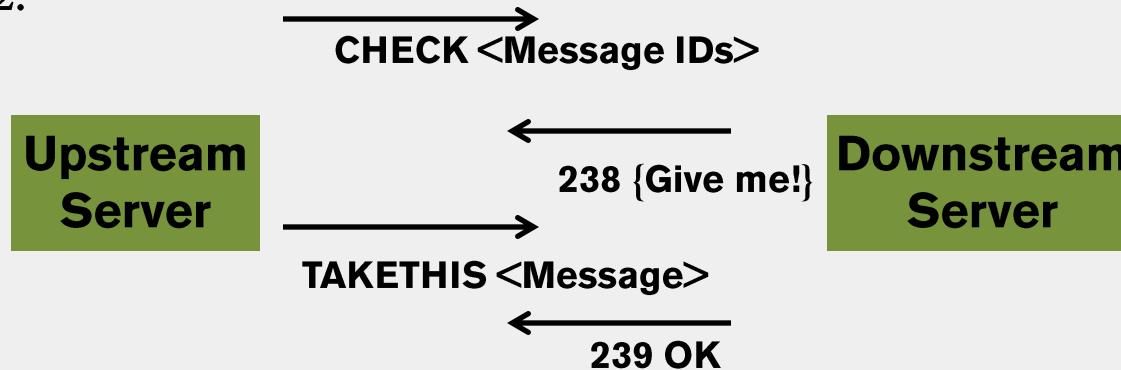
SOME IMPLEMENTATIONS

- Clearinghouse and Bayou projects: email and database transactions [PODC '87]
- refDBMS system [Usenix '94]
- Bimodal Multicast [ACM TOCS '99]
- Sensor networks [Li Li et al, Infocom '02, and PBBF, ICDCS '05]
- AWS EC2 and S3 Cloud (rumored). ['00s]
- Cassandra key-value store (and others) use gossip for maintaining membership lists
- Usenet NNTP (Network News Transport Protocol) ['79]

NNTP INTER-SERVER PROTOCOL

1. Each client uploads and downloads news posts from a news server

2.



Server retains news posts for a while,
transmits them lazily, deletes them after a while.

SUMMARY

- Multicast is an important problem
- Tree-based multicast protocols
- When concerned about scale and fault-tolerance, gossip is an attractive solution
- Also known as epidemics
- Fast, reliable, fault-tolerant, scalable, topology-aware

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

GRIDS

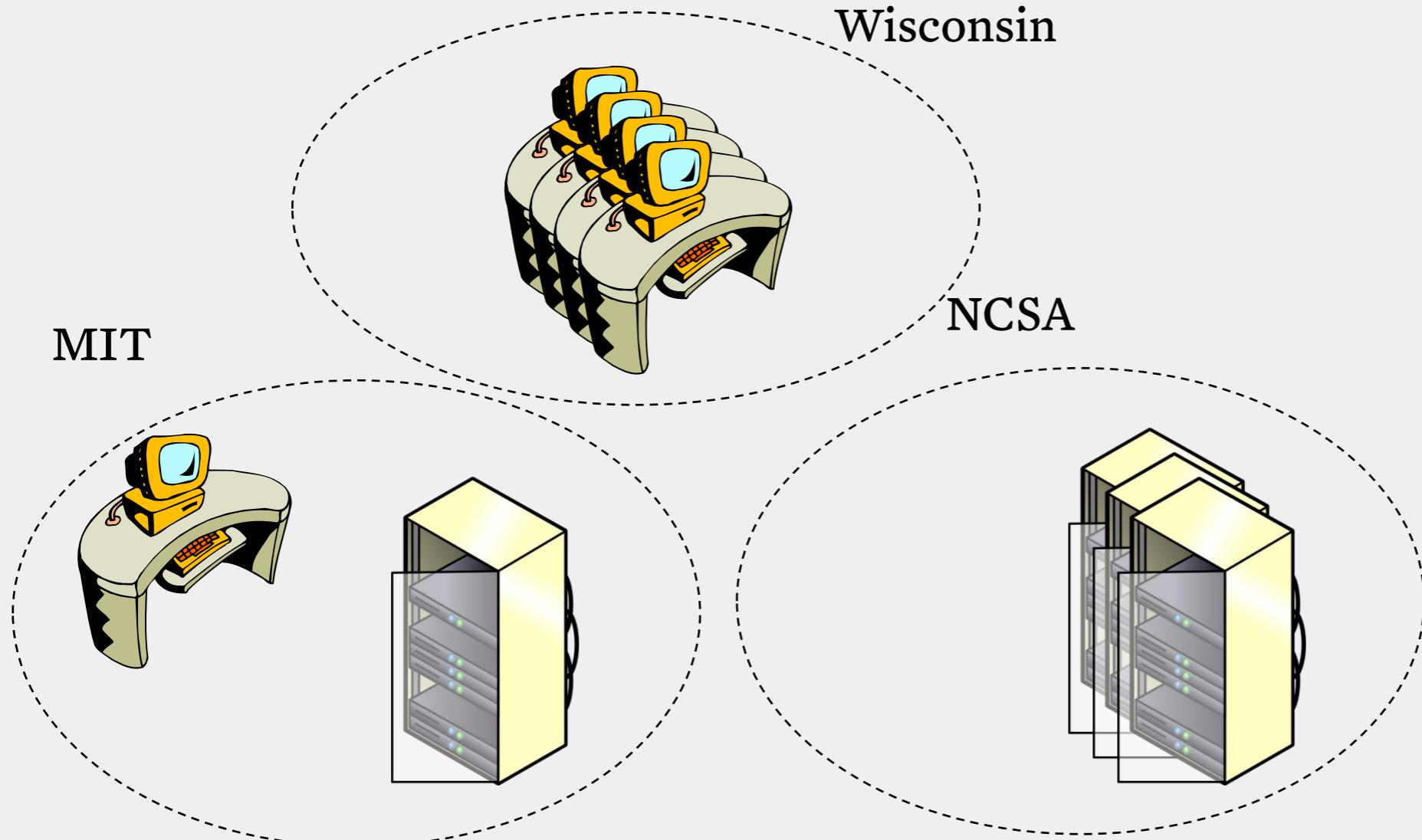
Lecture A

GRID APPLICATIONS

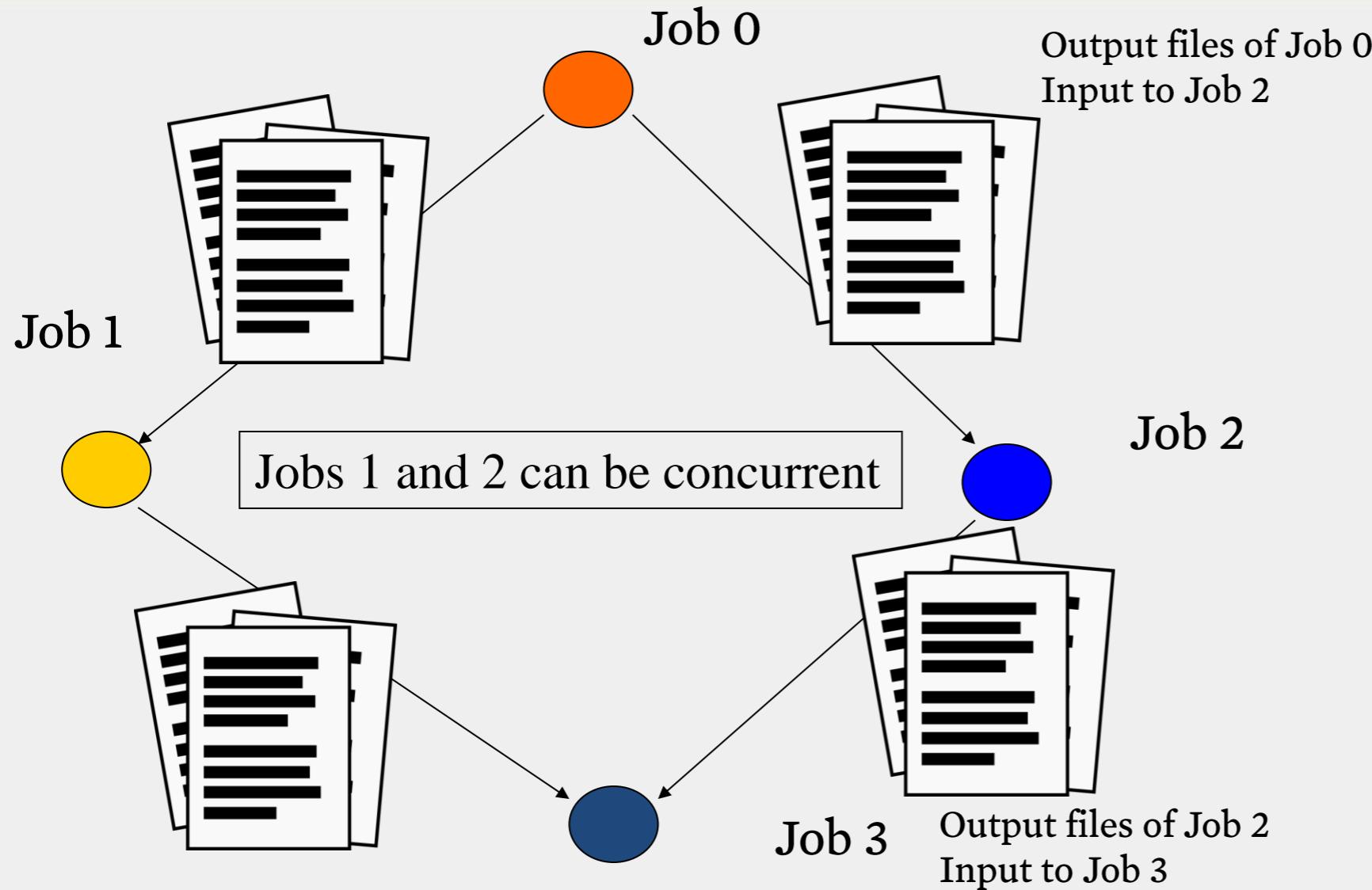
EXAMPLE: RAPID ATMOSPHERIC MODELING SYSTEM, COLOSTATE U

- Hurricane Georges, 17 days in Sept 1998
 - “RAMS modeled the mesoscale convective complex that dropped so much rain, in good agreement with recorded data”
 - Used 5 km spacing instead of the usual 10 km
 - Ran on 256+ processors
- Computation-intensive computing (or HPC = High Performance Computing)
- *Can one run such a program without access to a supercomputer?*

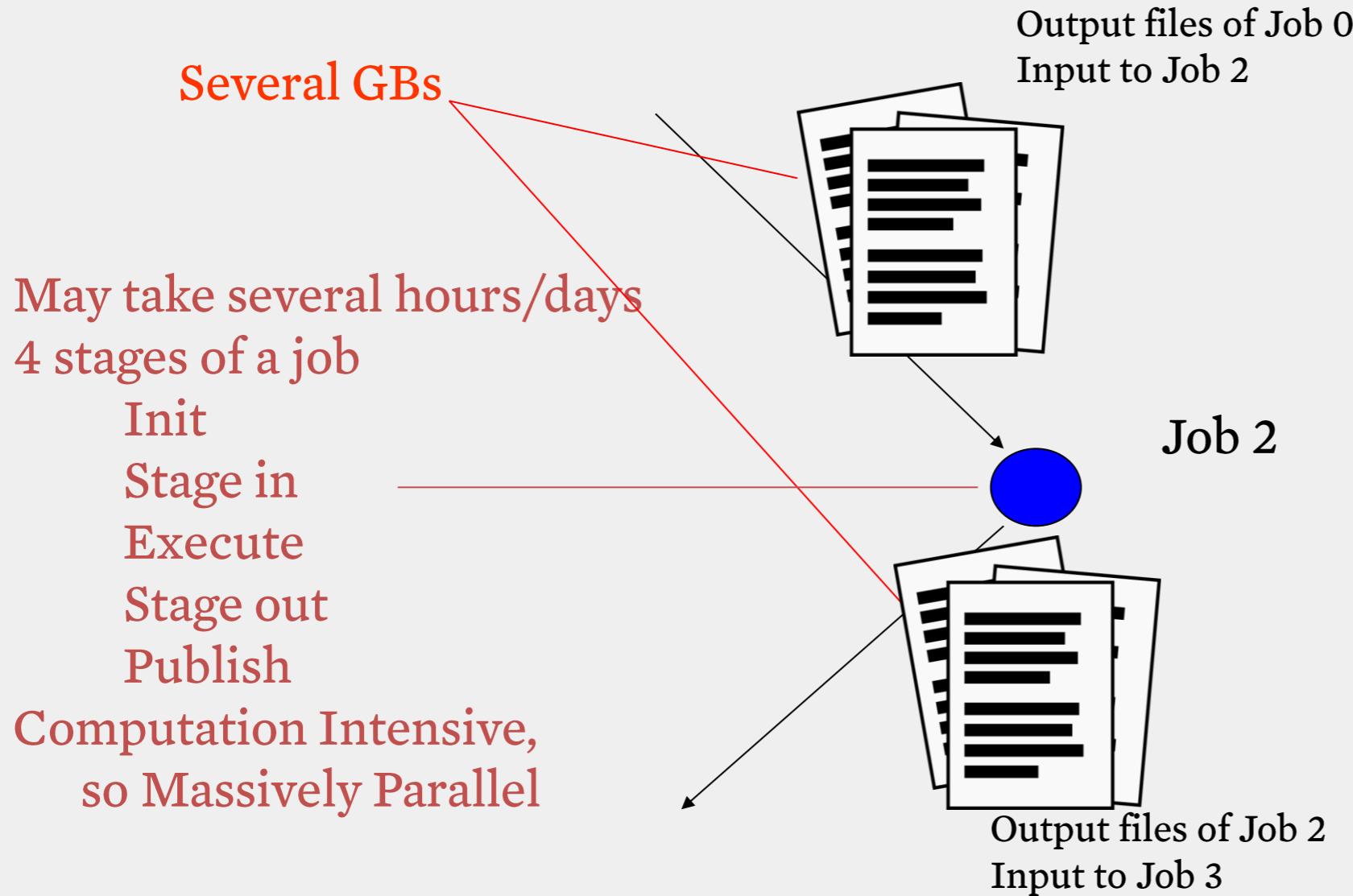
DISTRIBUTED COMPUTING RESOURCES



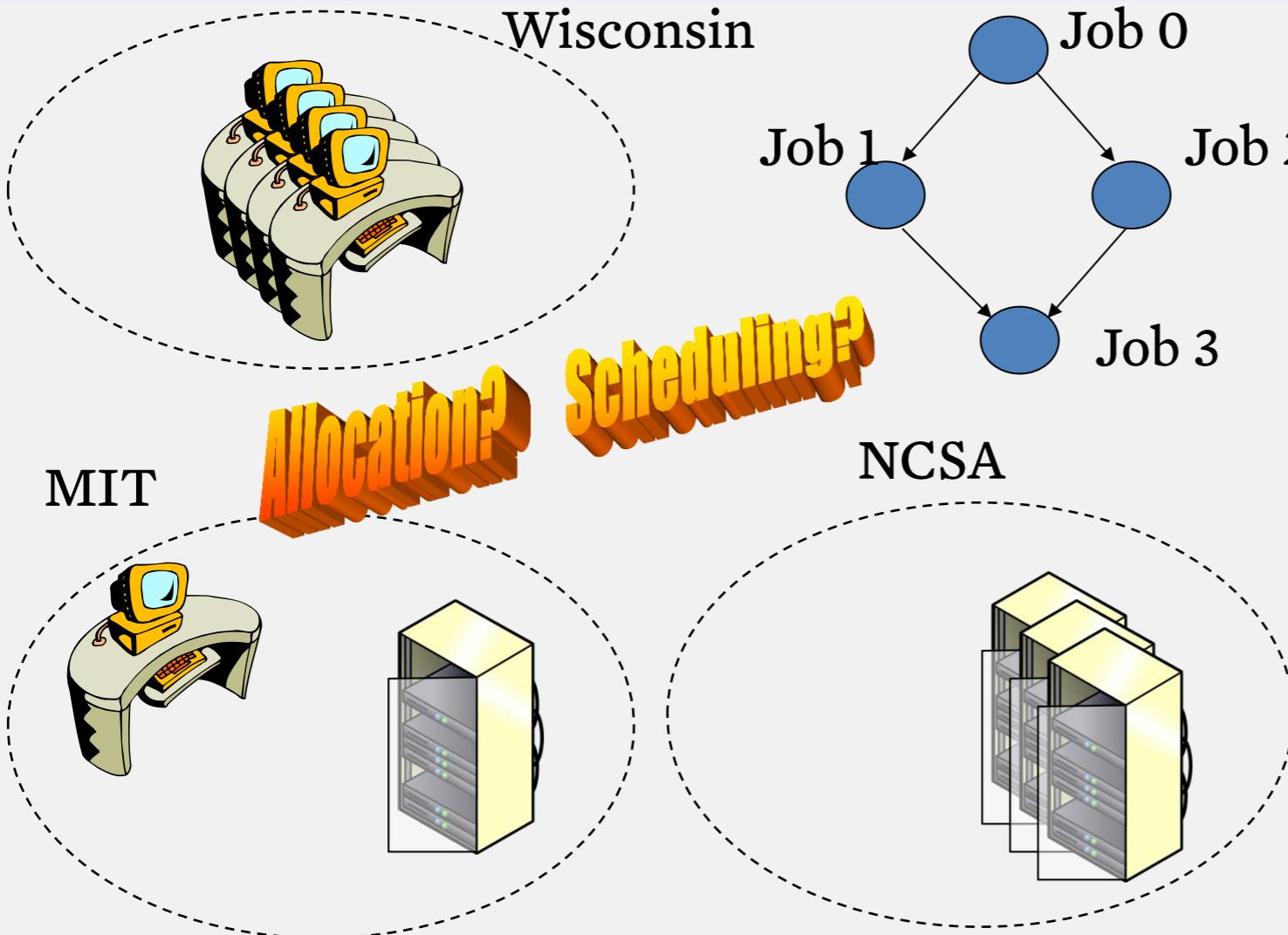
AN APPLICATION CODED BY A PHYSICIST/BIOLOGIST/METEOROLOGIST



AN APPLICATION CODED BY A PHYSICIST/BIOLOGIST/METEOROLOGIST



NEXT: SCHEDULING PROBLEM



CLOUD COMPUTING CONCEPTS

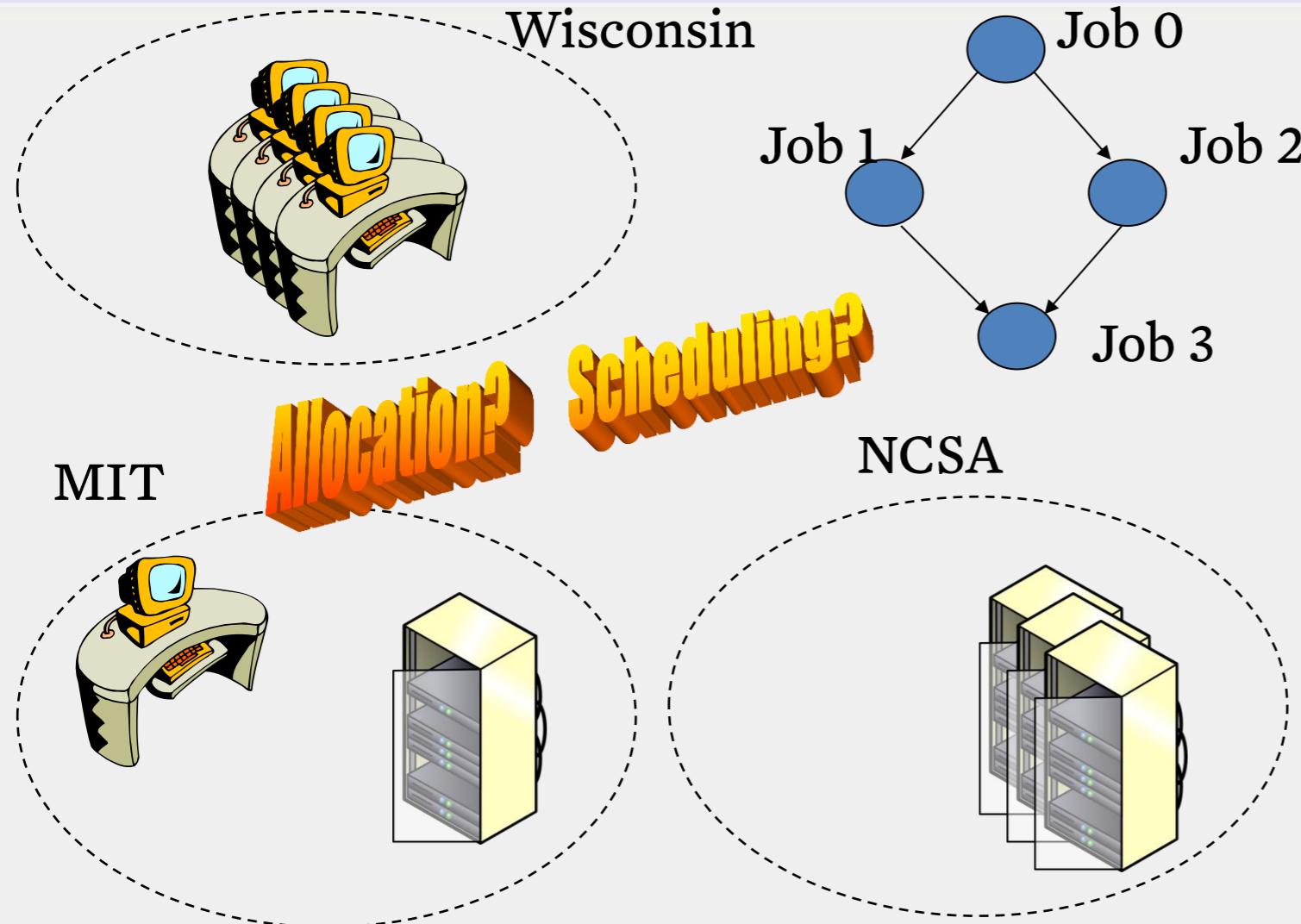
with Indranil Gupta (Indy)

GRIDS

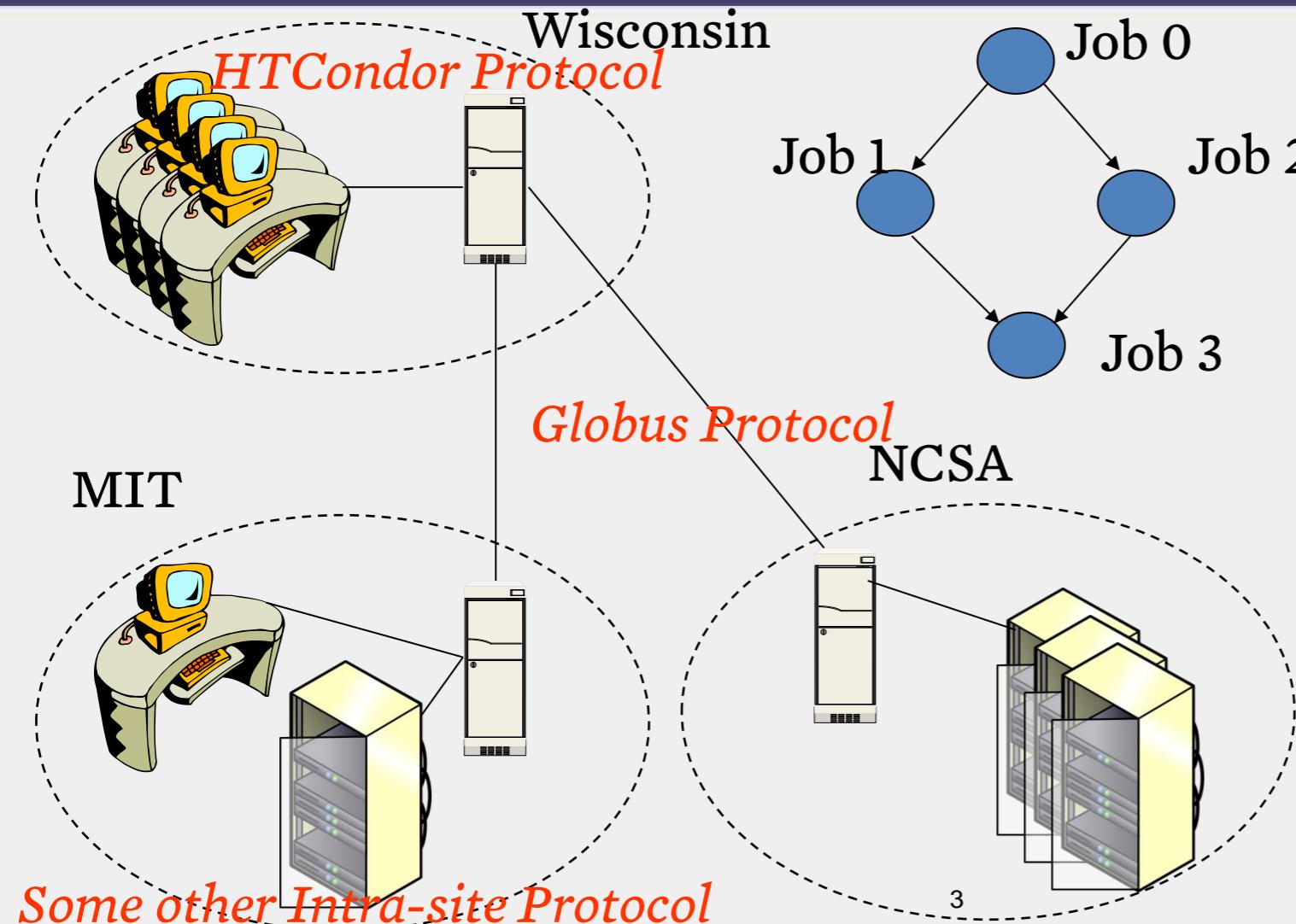
Lecture B

GRID INFRASTRUCTURE

SCHEDULING PROBLEM

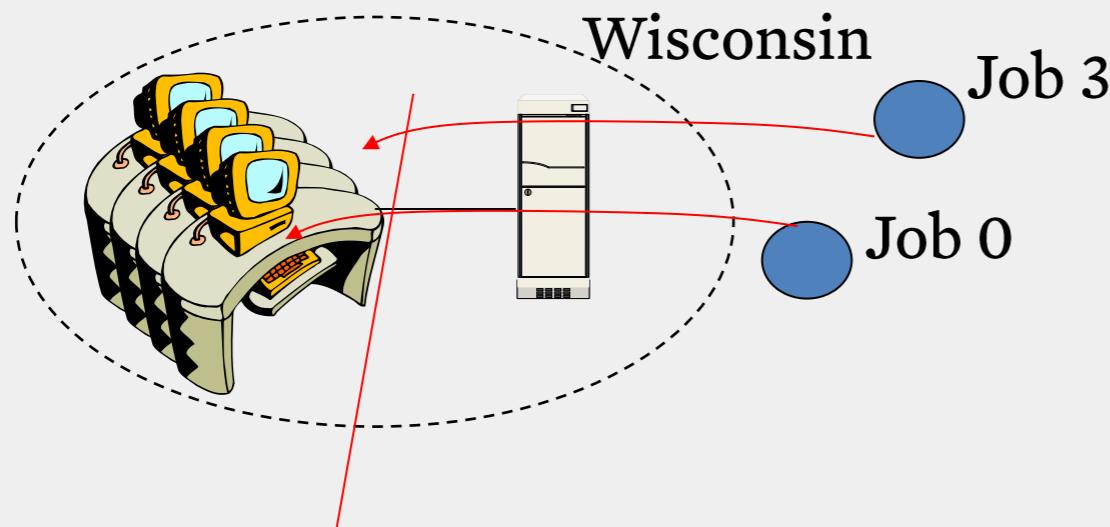


2-LEVEL SCHEDULING INFRASTRUCTURE



INTRA-SITE PROTOCOL

HTCondor Protocol



*Internal Allocation & Scheduling
Monitoring
Distribution and Publishing of Files*

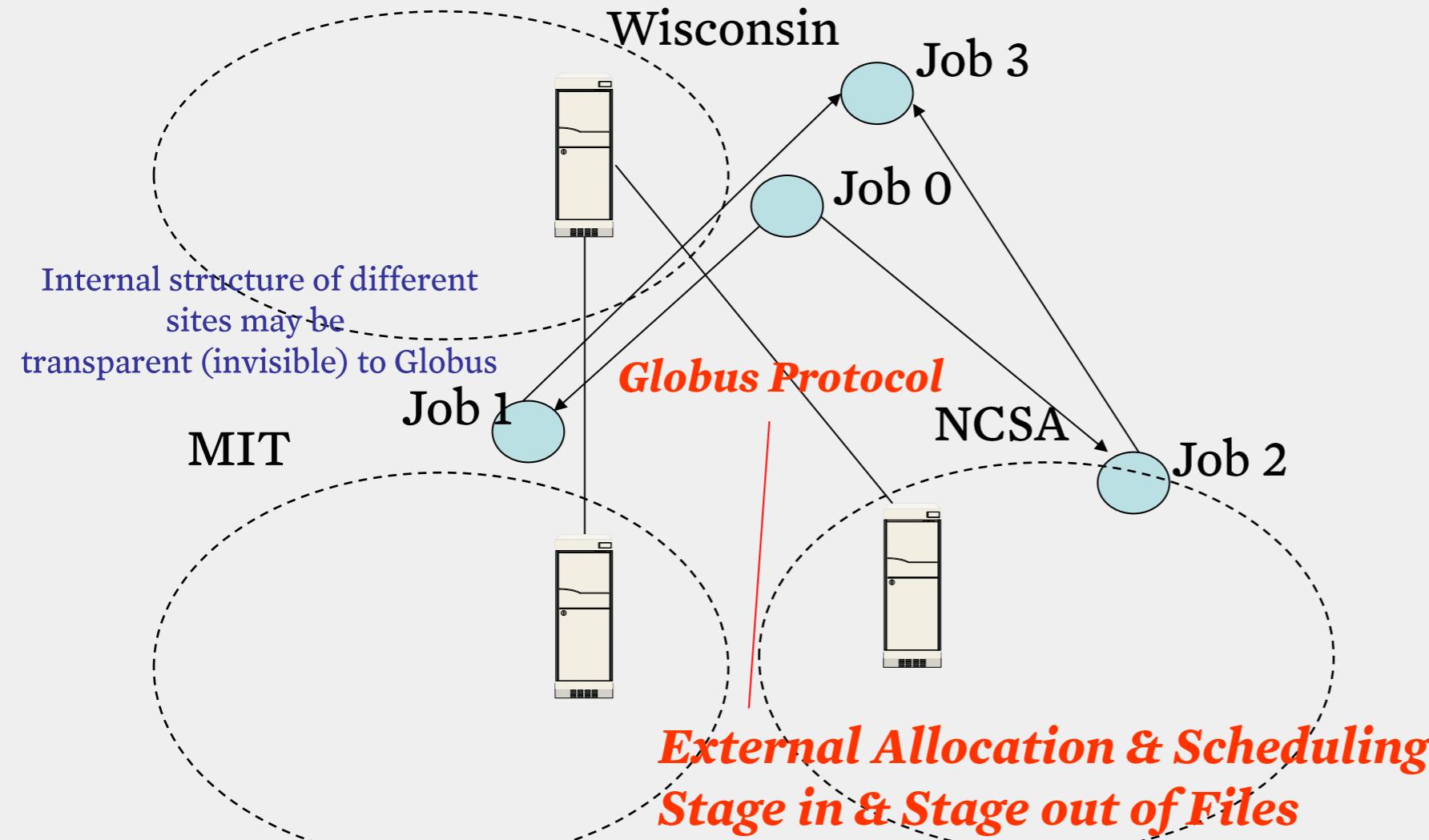
CONDOR (NOW HTCONDOR)

- High-throughput computing system from U. Wisconsin Madison
- Belongs to a class of Cycle-scavenging systems

Such systems

- Run on a lot of workstations
- When workstation is free, ask site's central server (or Globus) for tasks
- If user hits a keystroke or mouse click, stop task
 - Either kill task or ask server to reschedule task
- Can also run on dedicated machines

INTER-SITE PROTOCOL



GLOBUS

- Globus Alliance involves universities, national US research labs, and some companies
- Standardized several things, especially software tools
- Separately, but related: Open Grid Forum
- Globus Alliance has developed the Globus Toolkit

<http://toolkit.globus.org/toolkit/>

GLOBUS TOOLKIT

- Open-source
- Consists of several components
 - [GridFTP](#): Wide-area transfer of bulk data
 - [GRAM5](#) (Grid Resource Allocation Manager): submit, locate, cancel, and manage jobs
 - Not a scheduler
 - Globus communicates with the schedulers in intra-site protocols like HTCondor or Portable Batch System (PBS)
 - [RLS](#) (Replica Location Service): Naming service that translates from a file/dir name to a target location (or another file/dir name)
 - Libraries like [XIO](#) to provide a standard API for all Grid IO functionalities
 - Grid Security Infrastructure ([GSI](#))

SECURITY ISSUES

- Important in Grids because they are *federated*, i.e., no single entity controls the entire infrastructure
- **Single sign-on**: collective job set should require once-only user authentication
- **Mapping to local security mechanisms**: some sites use Kerberos, others using Unix
- **Delegation**: credentials to access resources inherited by subcomputations, e.g., job 0 to job 1
- **Community authorization**: e.g., third-party authentication
- These are also important in clouds, but less so because clouds are typically run under a central control
- In clouds the focus is on failures, scale, on-demand nature

SUMMARY

- Grid computing focuses on computation-intensive computing (HPC)
- Though often federated, architecture and key concepts have a lot in common with that of clouds
- Are Grids/HPC converging towards clouds?
 - E.g., Compare OpenStack and Globus



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MEMBERSHIP

Lecture A

WHAT IS GROUP MEMBERSHIP LIST?

A CHALLENGE

- You've been put in charge of a datacenter, and your manager has told you, "Oh no! We don't have any failures in our datacenter!"
- Do you believe him/her?
 - What would be your first responsibility?
 - Build a failure detector
 - What are some things that could go wrong if you didn't do this?

FAILURES ARE THE NORM

... not the exception, in datacenters.

- Say, the rate of failure of one machine (OS/disk/motherboard/network, etc.) is once every 10 years (120 months) on average.
- When you have 120 servers in the DC, the **mean time to failure (MTTF)** of the next machine is 1 month.
- When you have 12,000 servers in the DC, the MTTF is about once every 7.2 hours!

A CHALLENGE

- You've been put in charge of a datacenter, and your manager has told you, "Oh no! We don't have any failures in our datacenter!"
- Do you believe him/her?
 - What would be your first responsibility?
 - Build a failure detector
 - What are some things that could go wrong if you didn't do this?

TO BUILD A FAILURE DETECTOR

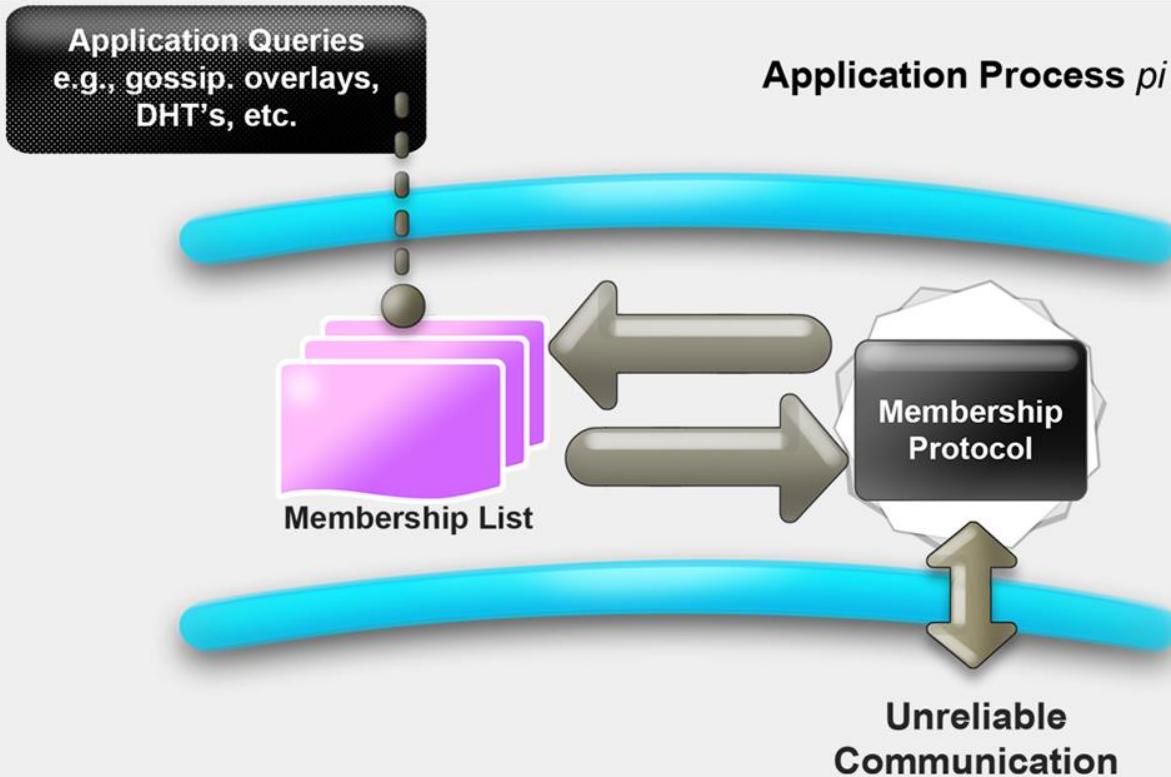
- You have a few options:
 1. Hire 1000 people, each to monitor one machine in the datacenter and report to you when it fails.
 2. Write a failure detector program (distributed) that automatically detects failures and reports to your workstation.

Which is more preferable, and why?

TARGET SETTINGS

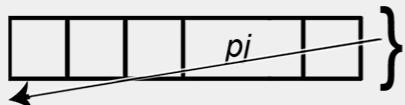
- Process ‘group’-based systems
 - Clouds/Datacenters
 - Replicated servers
 - Distributed databases
- Crash-stop/Fail-stop process failures

GROUP MEMBERSHIP SERVICE



TWO SUB-PROTOCOLS

Application Process π_i



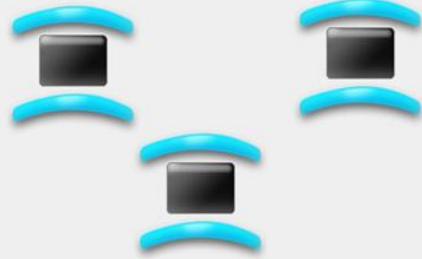
- Complete list all the time (**strongly consistent**)
 - Virtual synchrony
- Almost-Complete list (**weakly consistent**)
 - Gossip-style, SWIM, ... (FOCUS OF THIS LECTURE SERIES)
- Or Partial-random list (**other systems**)
 - SCAMP, T-MAN, Cyclon, ...



Unreliable
Communication

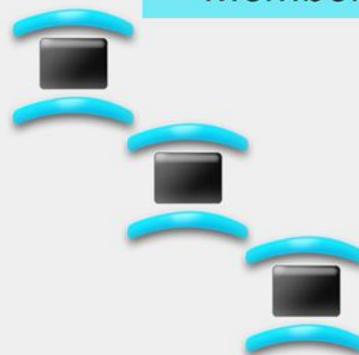
LARGE GROUP: SCALABILITY A GOAL

this is us (pi)

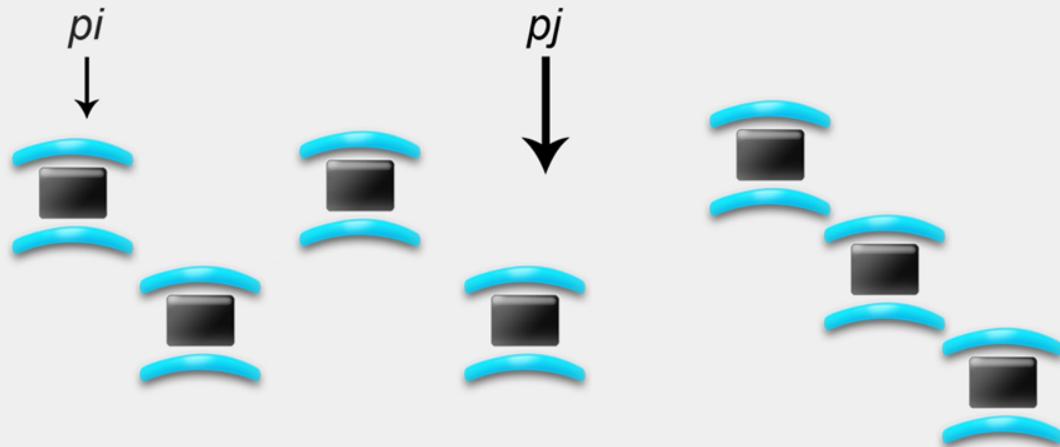


1000's of processes

Process Group
“Members”

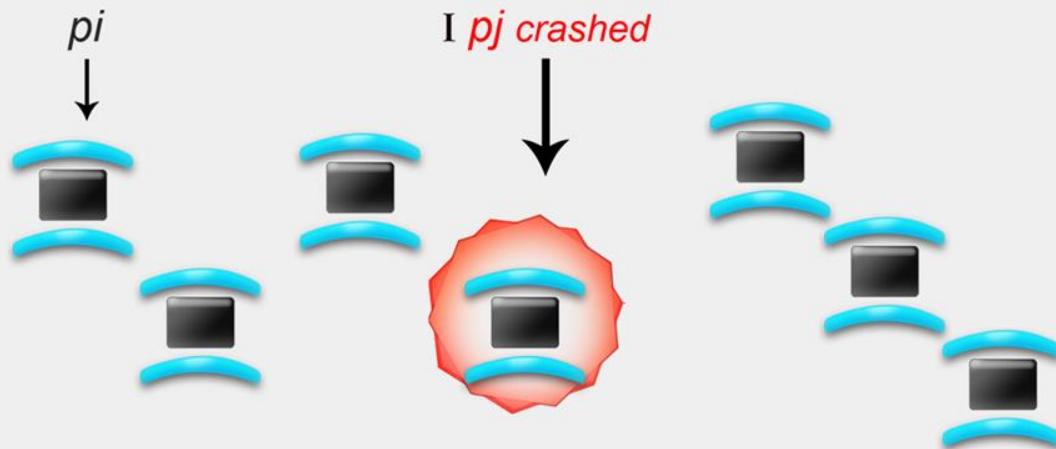


GROUP MEMBERSHIP PROTOCOL



Crash-stop Failures only

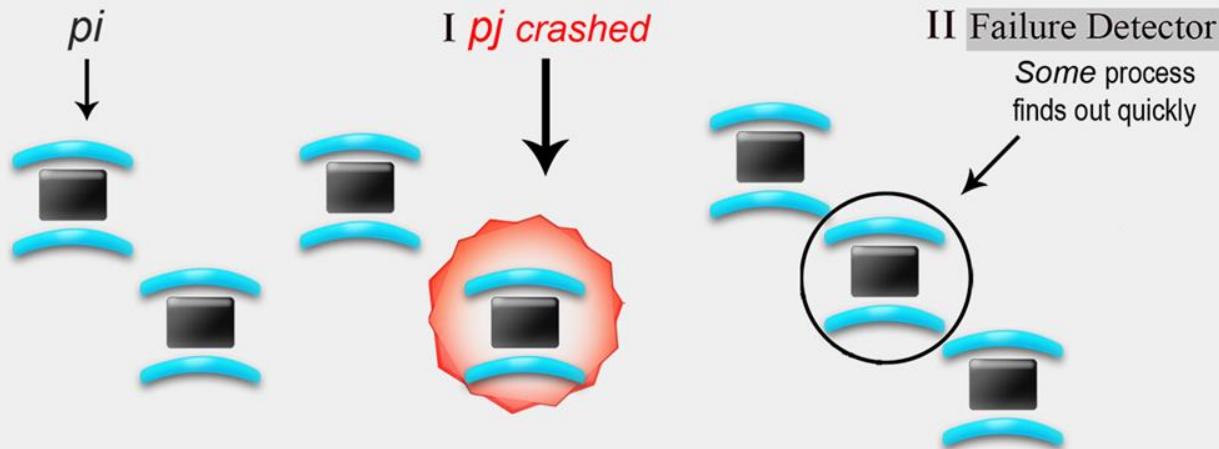
GROUP MEMBERSHIP PROTOCOL



Crash-stop Failures only



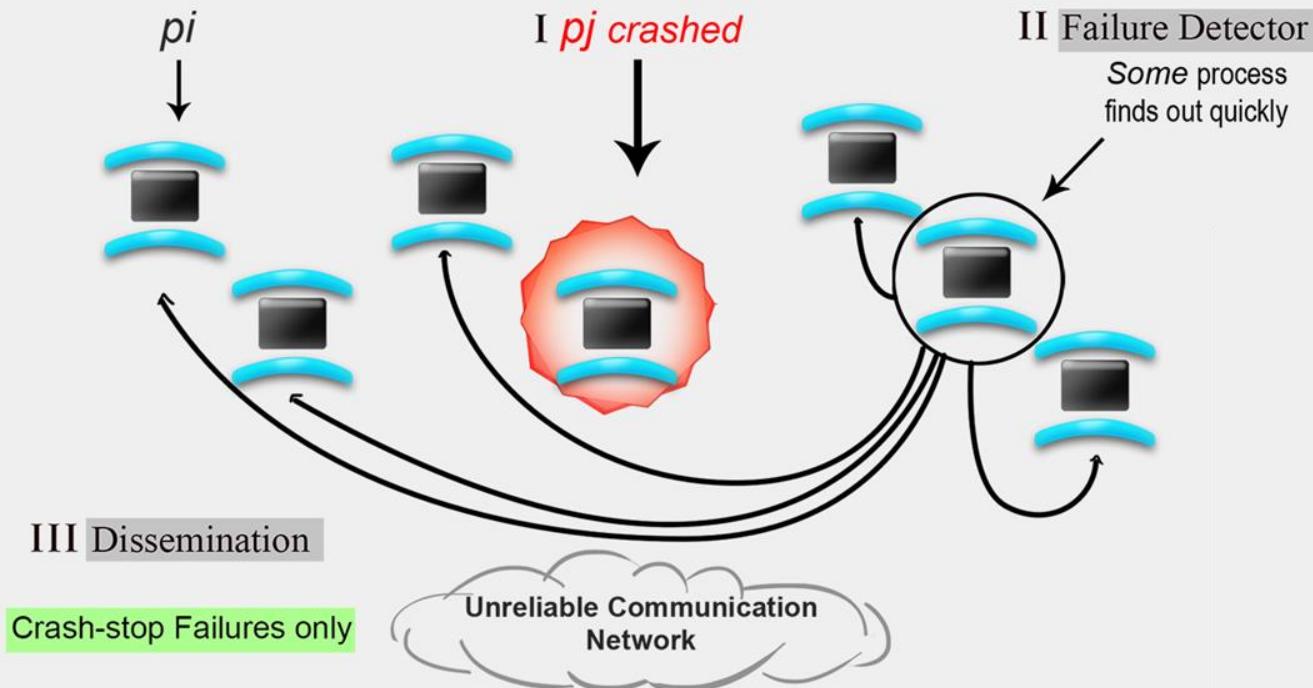
GROUP MEMBERSHIP PROTOCOL



Crash-stop Failures only

Unreliable Communication Network

GROUP MEMBERSHIP PROTOCOL



NEXT

- How do you design a group membership protocol?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

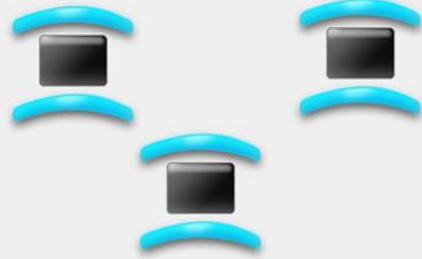
MEMBERSHIP

Lecture B

FAILURE DETECTORS

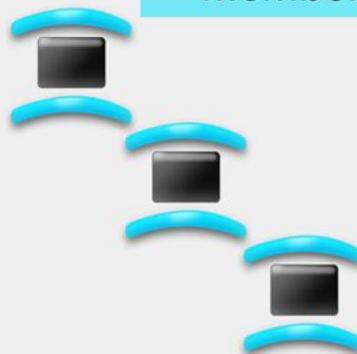
LARGE GROUP: SCALABILITY A GOAL

this is us (pi)

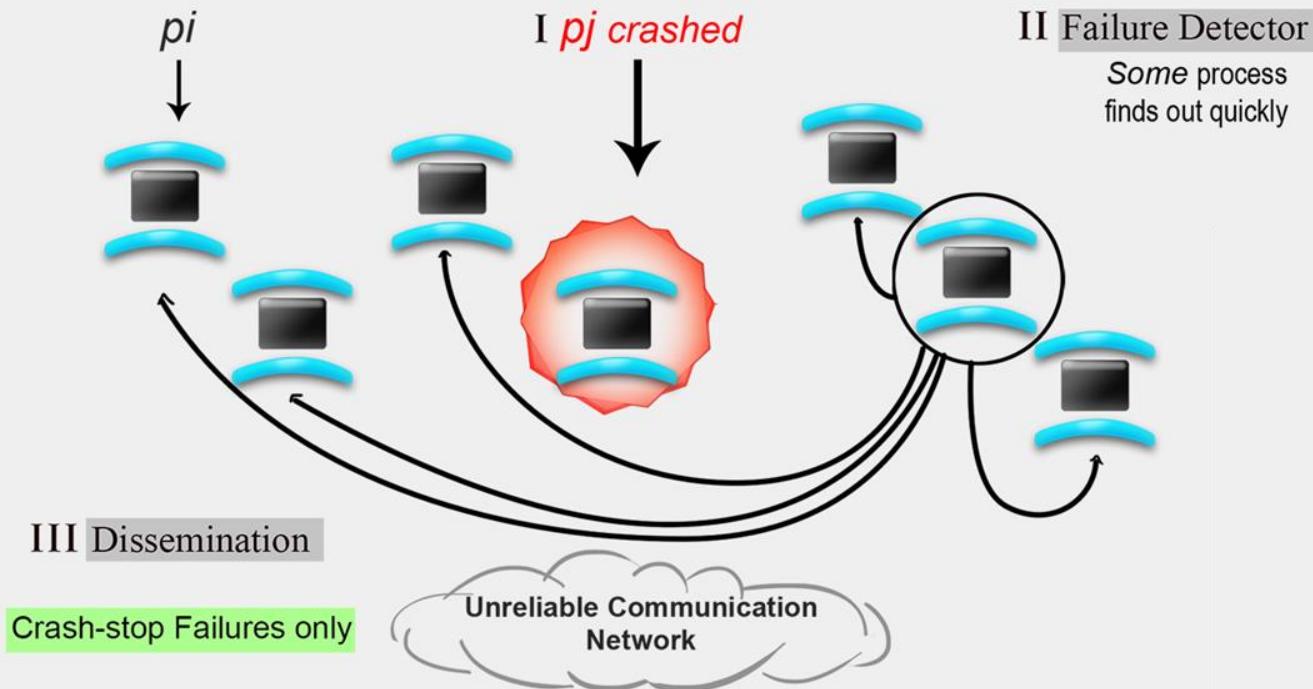


1000's of processes

Process Group
“Members”



GROUP MEMBERSHIP PROTOCOL



I. *pj* CRASHES

- Nothing we can do about it!
- A frequent occurrence
- Common case rather than exception
- Frequency goes up linearly with size of datacenter

II. DISTRIBUTED FAILURE DETECTORS: DESIRABLE PROPERTIES

- **Completeness** = each failure is detected
- **Accuracy** = there is no mistaken detection
- Speed
 - Time to first detection of a failure
- Scale
 - Equal Load on each member
 - Network Message Load

DISTRIBUTED FAILURE DETECTORS: PROPERTIES

- Completeness
 - Accuracy
-
- Speed
 - Time to first detection of failure
 - Scale
 - Equal Load on each member
 - Network Message Load

Impossible together in
lossy networks [Chandra
and Toueg]

If possible, then can
solve consensus!

WHAT REAL FAILURE DETECTORS PREFER

- Completeness
- Accuracy
- Speed
 - Time to first detection of failure
- Scale
 - Equal Load on each member
 - Network Message Load

Guaranteed

Partial/Probabilistic
guarantee

FAILURE DETECTOR PROPERTIES

- Completeness

Guaranteed

- Accuracy

Partial/Probabilistic
guarantee

- Speed
 - Time to first detection of failure

Time until **some**
process detects the failure

- Scale
 - Equal Load on each member
 - Network Message Load

FAILURE DETECTOR PROPERTIES

- Completeness

Guaranteed

- Accuracy

Partial/Probabilistic
guarantee

- Speed

- Time to first detection of failure

Time until **some**
process detects the failure

- Scale

- Equal Load on each member
 - Network Message Load

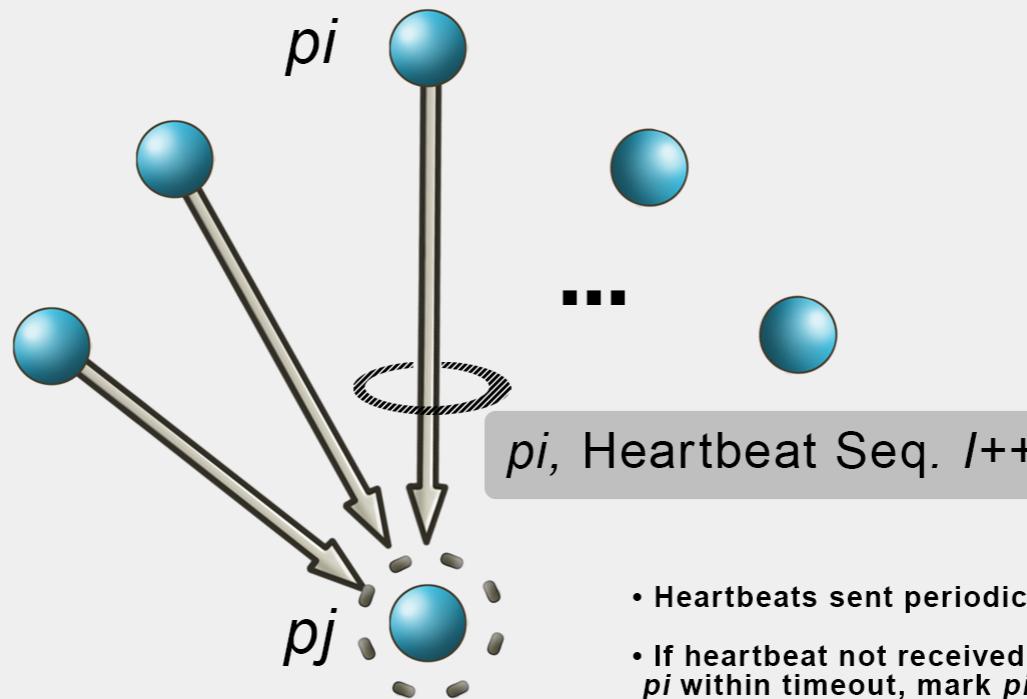
No bottlenecks/single
failure point

FAILURE DETECTOR PROPERTIES

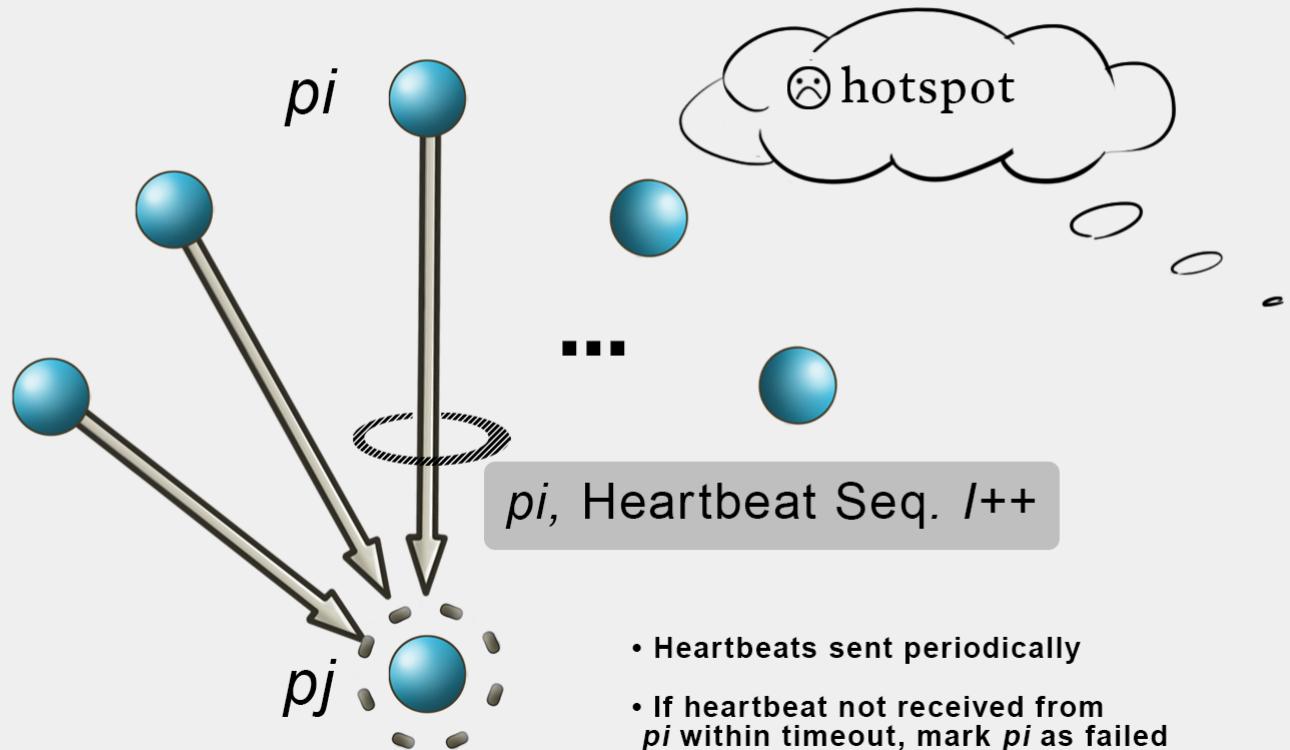
- Completeness
- Accuracy
- Speed
 - Time to first detection of failure
- Scale
 - Equal Load on each member
 - Network Message Load

In spite of
arbitrary simultaneous
process failures

CENTRALIZED HEARTBEATING

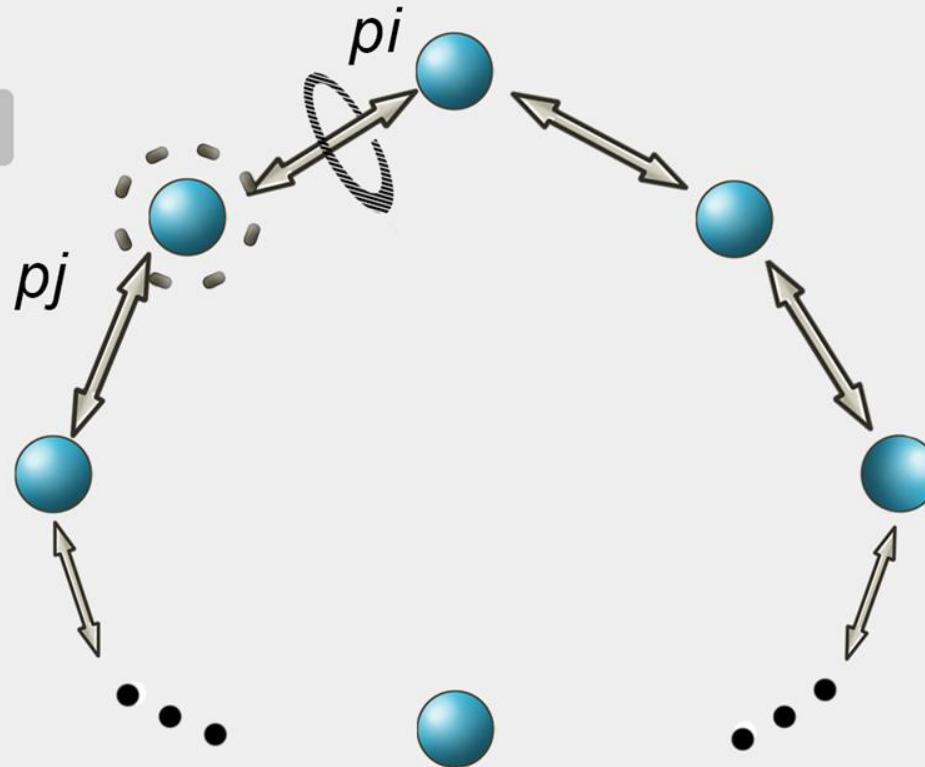


CENTRALIZED HEARTBEATING



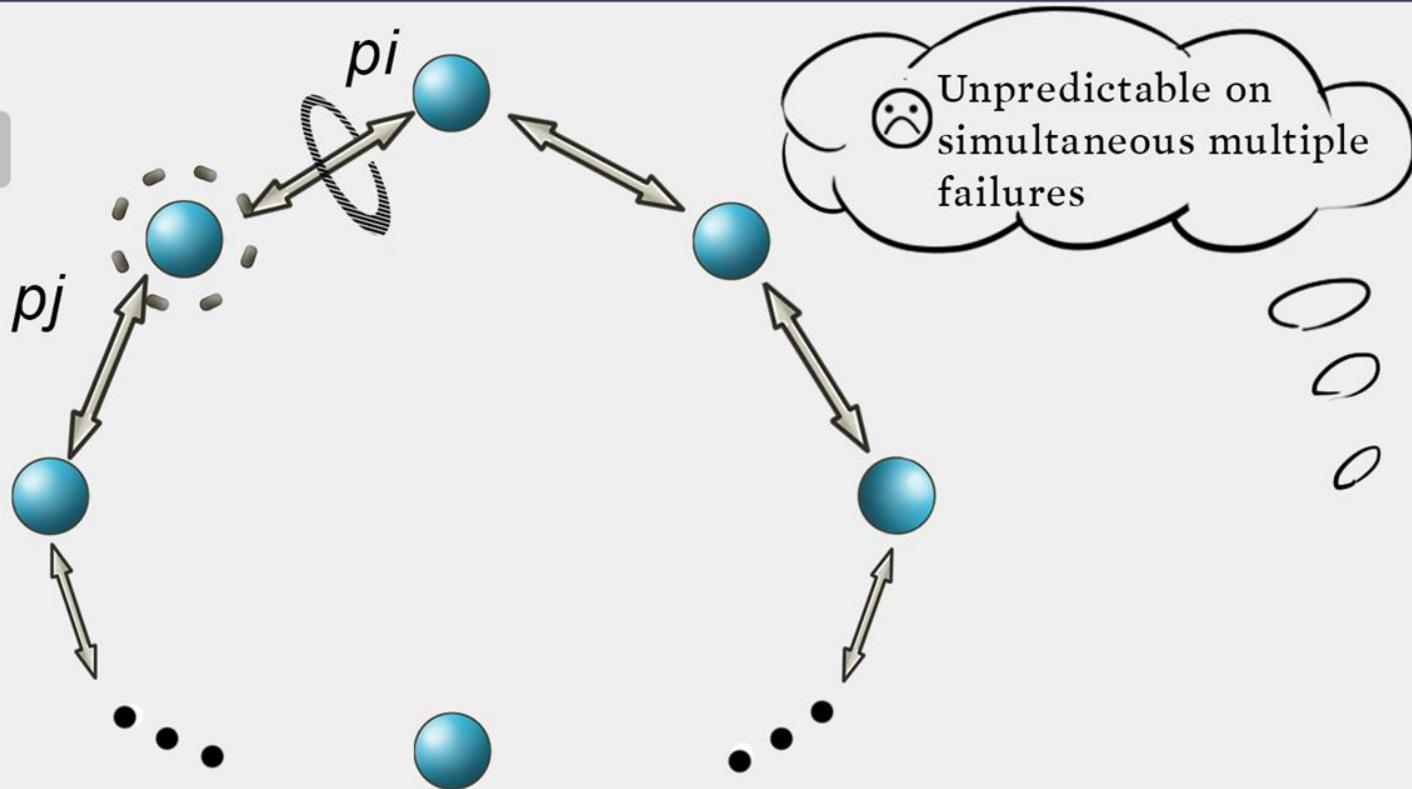
RING HEARTBEATING

p_i , Heartbeat Seq. /++

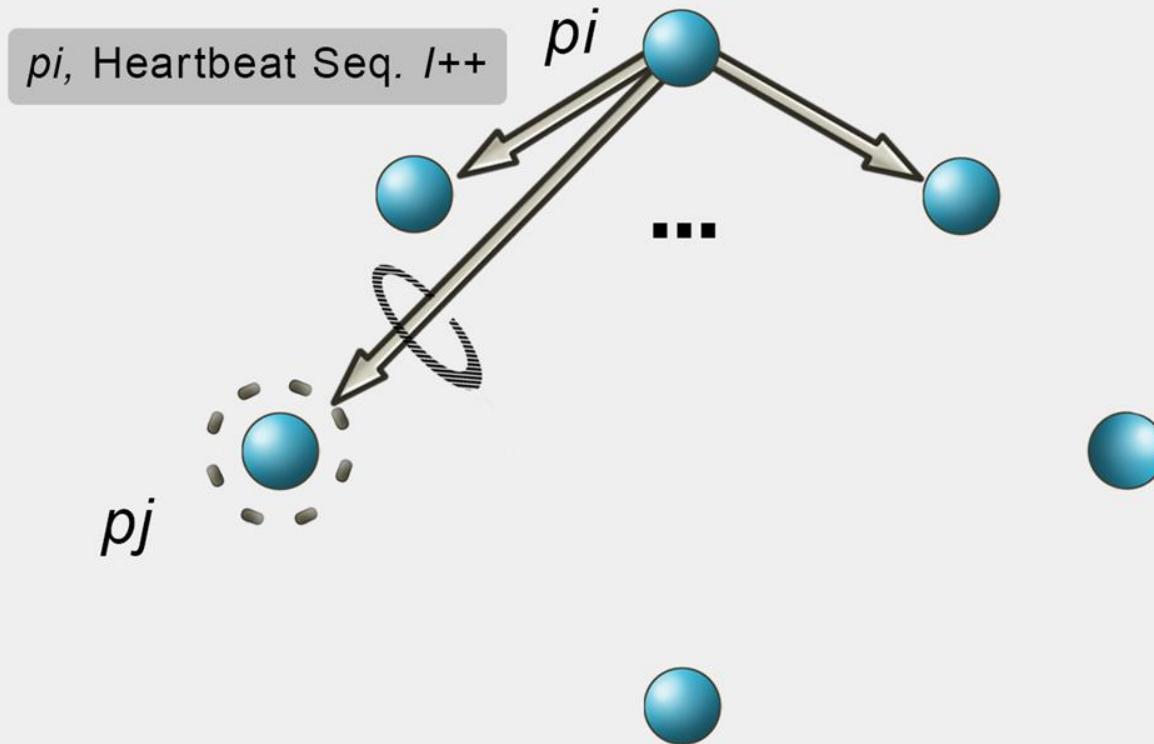


RING HEARTBEATING

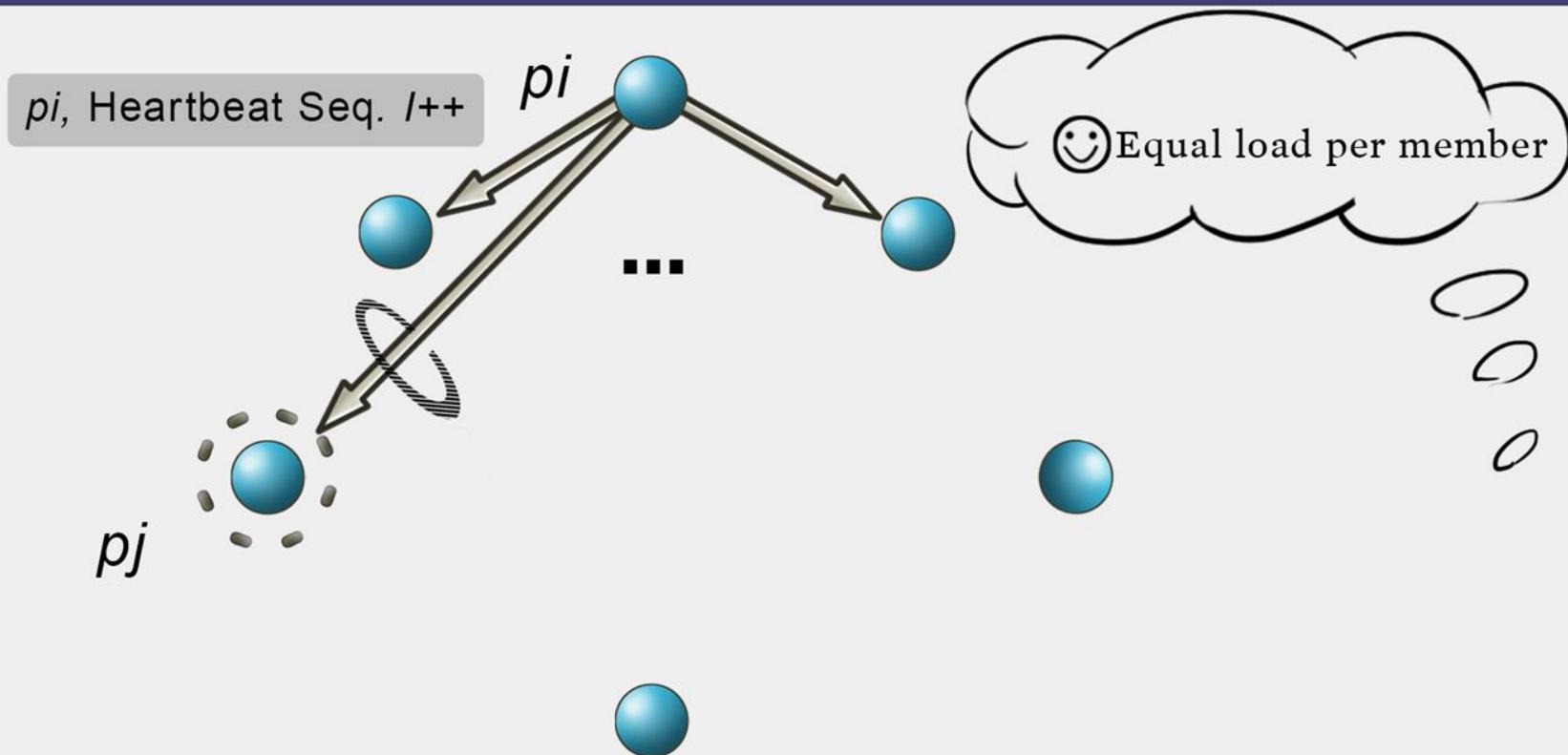
p_i , Heartbeat Seq. /++



ALL-TO-ALL HEARTBEATING



ALL-TO-ALL HEARTBEATING



NEXT

- How do you increase the robustness of all-to-all heartbeating?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

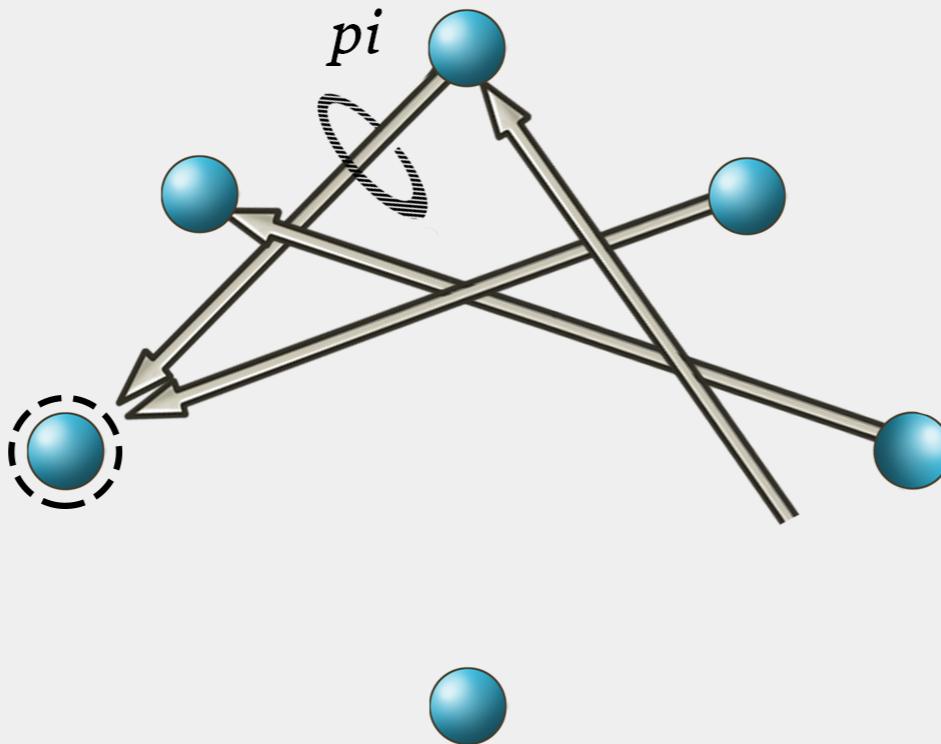
MEMBERSHIP

Lecture C

GOSSIP-STYLE MEMBERSHIP

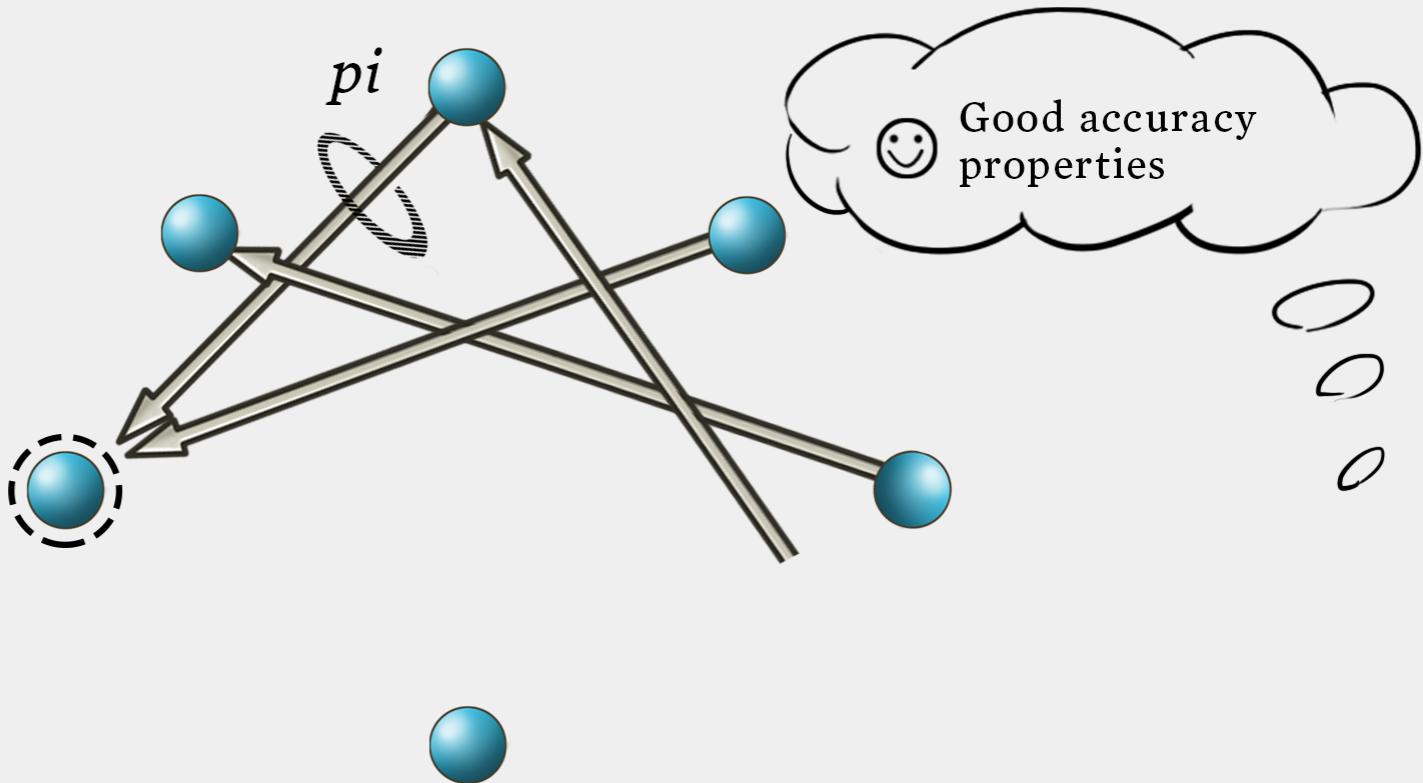
GOSSIP-STYLE HEARTBEATING

Array of
Heartbeat seq. /
for member subset

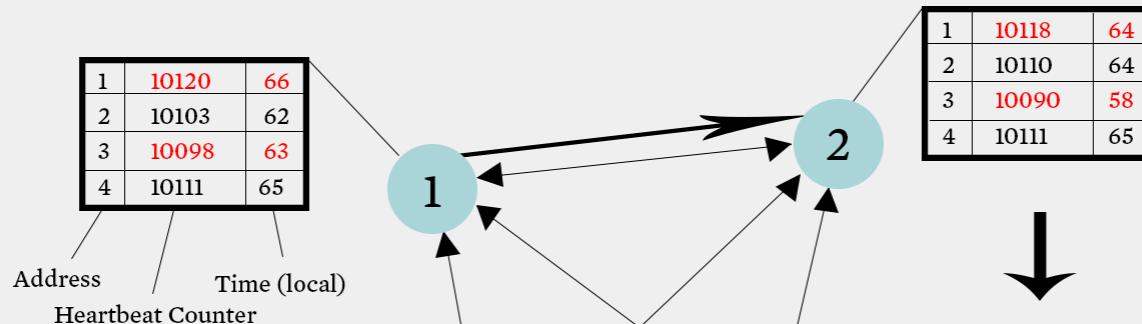


GOSSIP-STYLE HEARTBEATING

Array of
Heartbeat seq. /
for member subset



GOSSIP-STYLE FAILURE DETECTION



Protocol

- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated

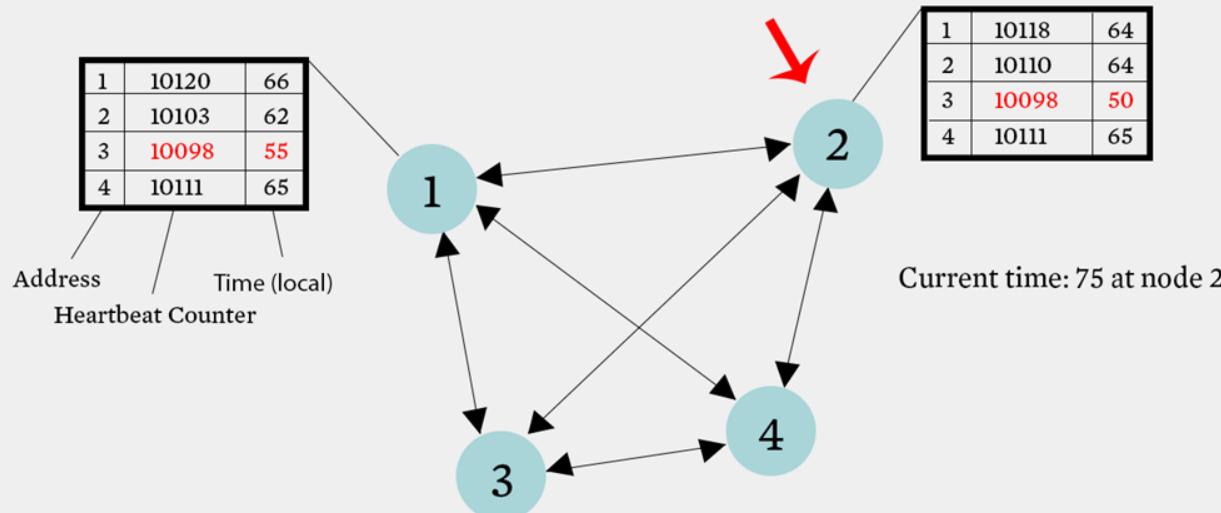
Current time: 70 at node 2
(asynchronous clocks)

GOSSIP-STYLE FAILURE DETECTION

- If the heartbeat has not increased for more than T_{fail} seconds, the member is considered failed
- And after $T_{cleanup}$ seconds, it will delete the member from the list
- Why two different timeouts?

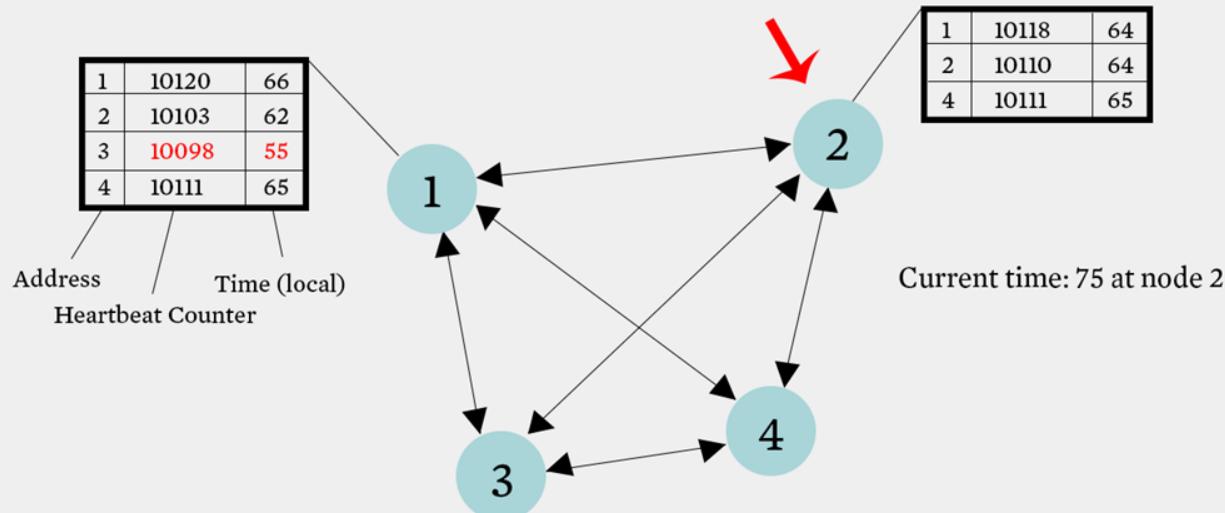
GOSSIP-STYLE FAILURE DETECTION

- What if an entry pointed to a failed node is deleted right after T_{fail} ($=24$) seconds?



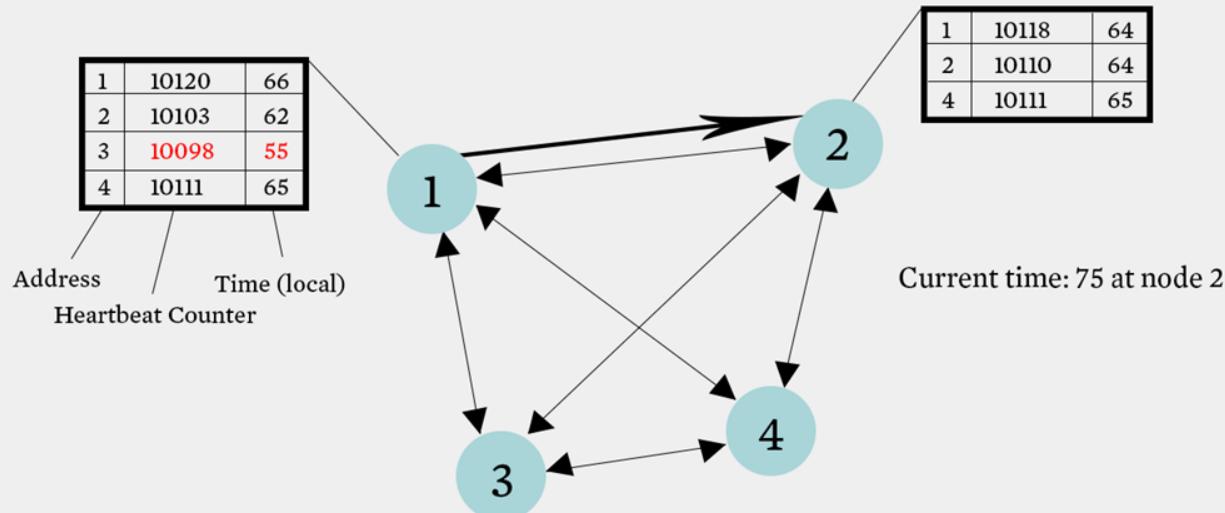
GOSSIP-STYLE FAILURE DETECTION

- What if an entry pointed to a failed node is deleted right after T_{fail} ($=24$) seconds?



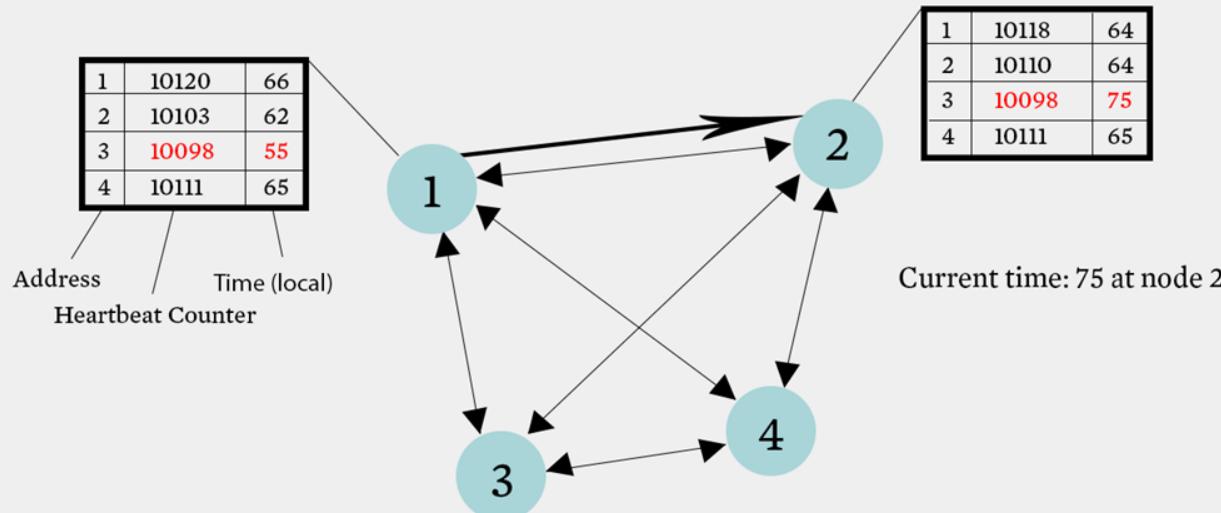
GOSSIP-STYLE FAILURE DETECTION

- What if an entry pointed to a failed node is deleted right after T_{fail} ($=24$) seconds?



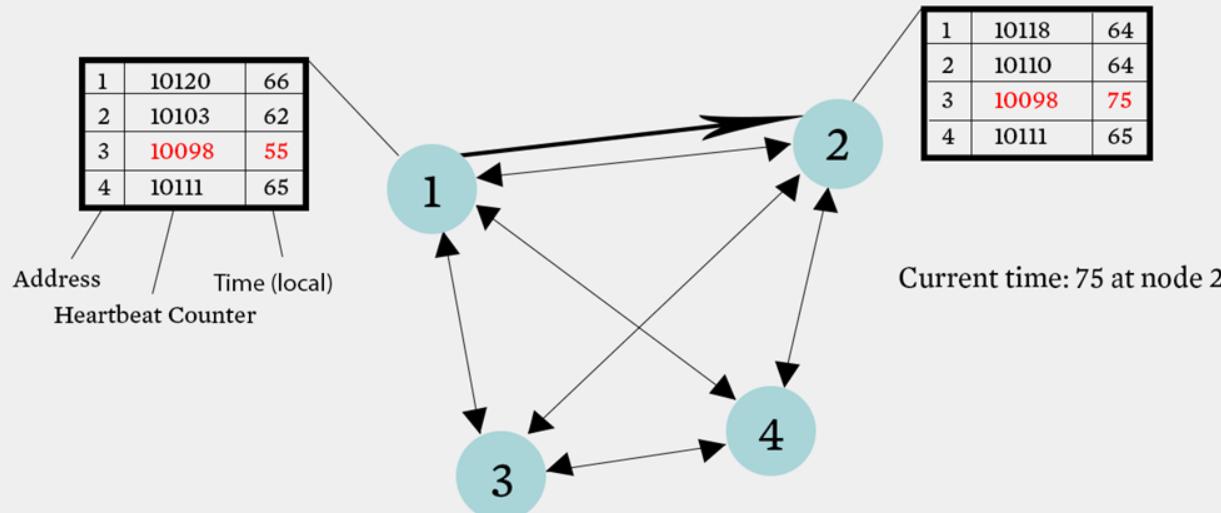
GOSSIP-STYLE FAILURE DETECTION

- What if an entry pointed to a failed node is deleted right after T_{fail} ($=24$) seconds?



GOSSIP-STYLE FAILURE DETECTION

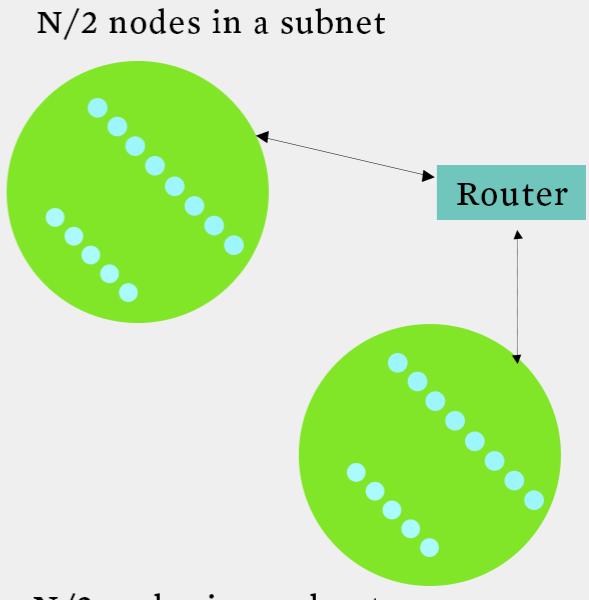
- What if an entry pointed to a failed node is deleted right after T_{fail} ($=24$) seconds?



- Fix: remember for another T_{fail}

MULTI-LEVEL GOSSIPING

- Network topology is hierarchical
- Random gossip target selection => core routers face **O(N) load** (Why?)
- **Fix:** Select gossip target in subnet i , which contains n_i nodes, with probability $1/n_i$
- Router load = $O(1)$
- Dissemination time = $O(\log(N))$
 - Why?
- What about latency for multi-level topologies?
[Gupta et al, TPDS 06]



ANALYSIS/DISCUSSION

- What happens if gossip period T_{gossip} is decreased?
- A single heartbeat takes $O(\log(N))$ time to propagate.

So: N heartbeats take:

- $O(\log(N))$ time to propagate, if bandwidth allowed per node is allowed to be $O(N)$
- $O(N \cdot \log(N))$ time to propagate, if bandwidth allowed per node is only $O(1)$
- What about $O(k)$ bandwidth?
- What happens to $P_{mistake}$ (false positive rate) as $T_{fail}, T_{cleanup}$ is increased?
- Tradeoff: False positive rate vs. detection time vs. bandwidth

NEXT

- So, is this the best we can do? What is the best we can do?

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MEMBERSHIP

Lecture D

WHICH IS THE BEST FAILURE DETECTOR?

FAILURE DETECTOR PROPERTIES ...

- Completeness
- Accuracy
- Speed
 - Time to first detection of a failure
- Scale
 - Equal Load on each member
 - Network Message Load

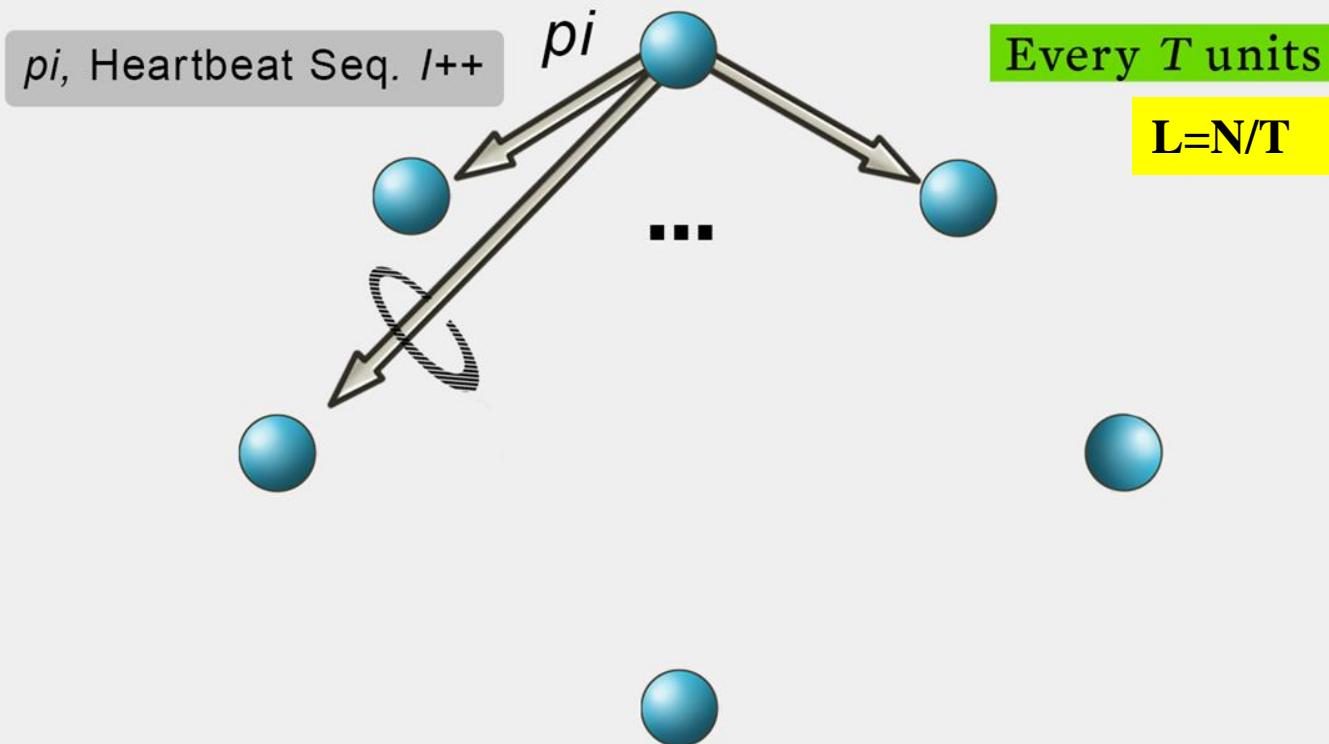
FAILURE DETECTOR PROPERTIES ...

- Completeness Guarantee always
 - Accuracy Probability PM(T)
 - Speed T time units
 - Time to first detection of a failure
 - Scale
 - Equal Load on each member
 - Network Message Load

FAILURE DETECTOR PROPERTIES ...

- Completeness
 - Accuracy
 - Speed
 - Time to first detection of a failure
 - Scale
 - Equal Load on each member
 - Network Message Load

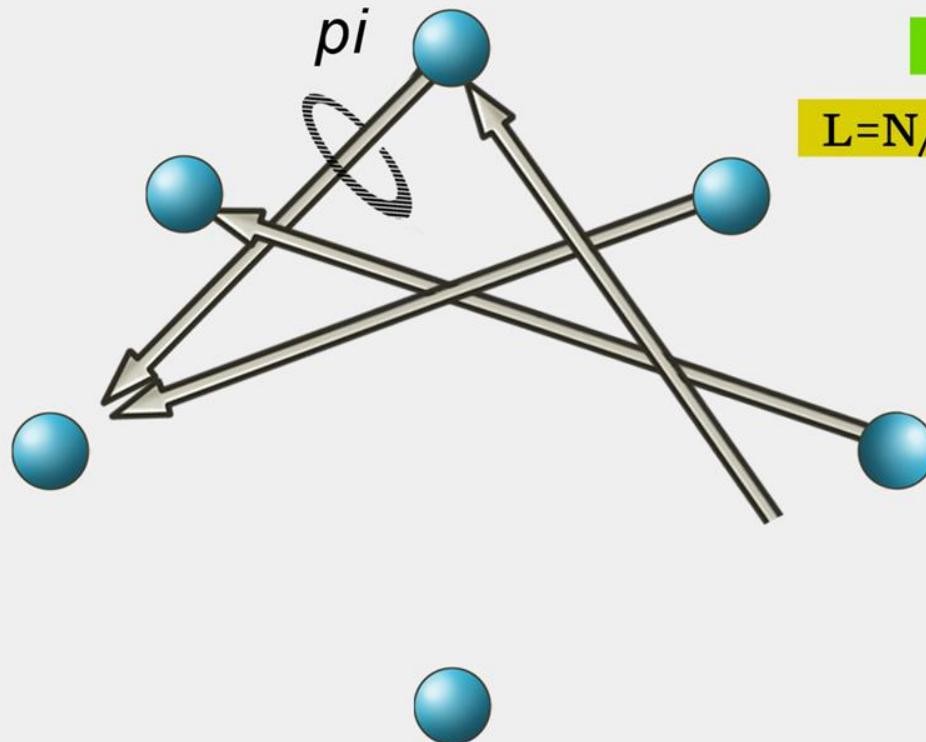
ALL-TO-ALL HEARTBEATING



ALL-TO-ALL HEARTBEATING

Array of
Heartbeat seq. I
for member subset

Every tg units
=gossip period,
send $O(N)$ gossip
message



$$T = \log N * tg$$

$$L = N / tg = N * \log N / T$$

WHAT'S THE BEST/OPTIMAL WE CAN DO?

- Worst case load L^*

as a function of $T, PM(T), N$

Independent Message Loss probability p_{ml}

$$L^* = \frac{\log(PM(T))}{\log(p_{ml})} \cdot \frac{1}{T}$$

(try to work out the proof)

WHAT'S THE BEST/OPTIMAL WE CAN DO?

- Optimal L is independent of N (!)
- All-to-all and gossip-based: sub-optimal
 - $L=O(N/T)$
 - try to achieve simultaneous detection at *all* processes
 - fail to distinguish *Failure Detection* and *Dissemination* components

Key:

- Separate the two components
- Use a non heartbeat-based Failure Detection Component

NEXT

- Is there a better failure detector?



CLOUD COMPUTING CONCEPTS

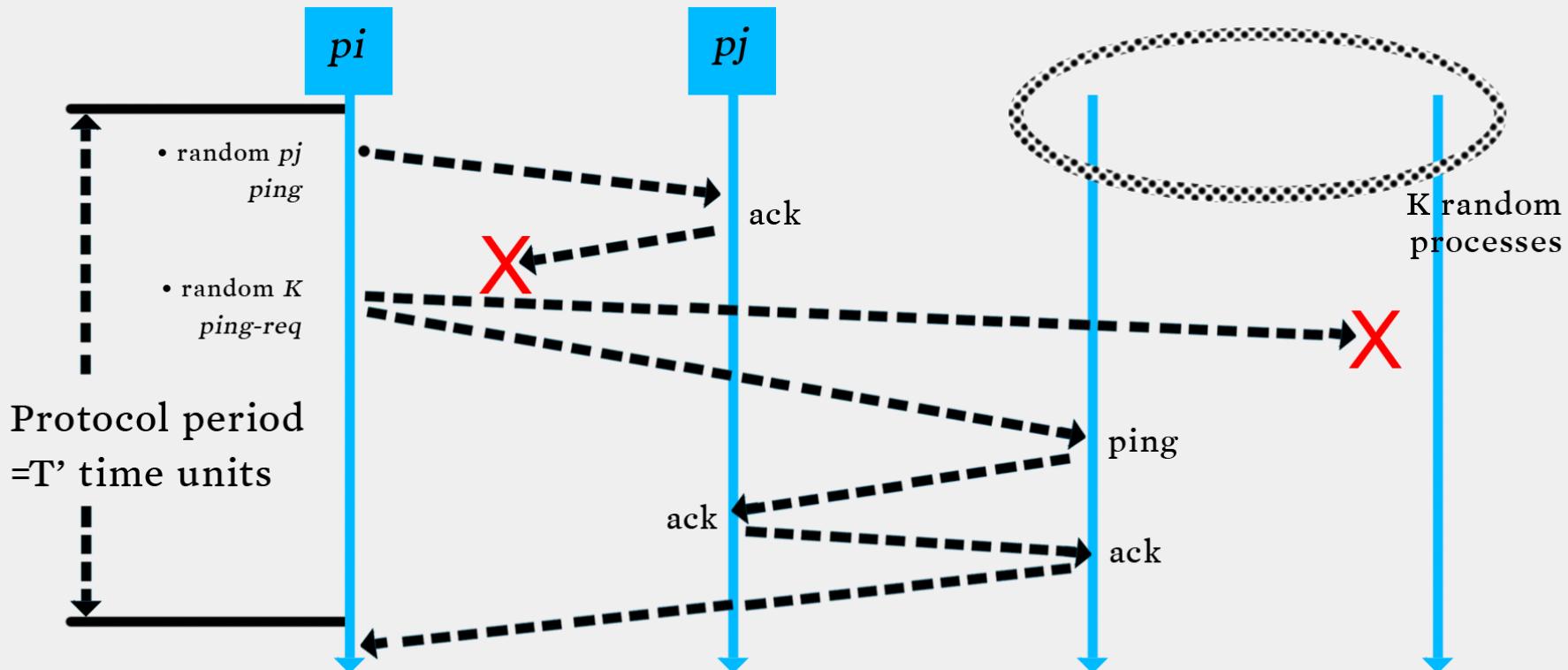
with Indranil Gupta (Indy)

MEMBERSHIP

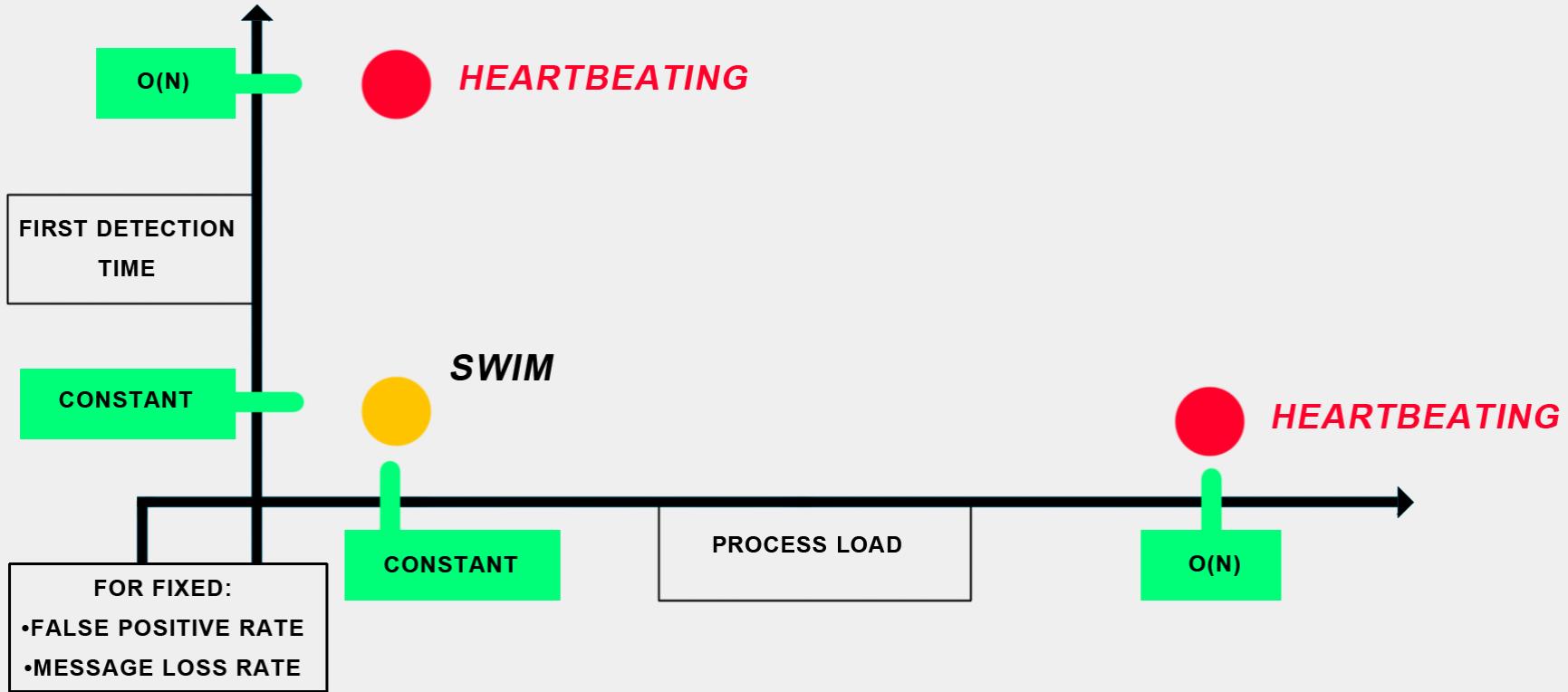
Lecture E

ANOTHER PROBABILISTIC
FAILURE DETECTOR

SWIM FAILURE DETECTOR PROTOCOL



SWIM VERSUS HEARTBEATING



SWIM FAILURE DETECTOR



PARAMETER	SWIM
First Detection Time	<ul style="list-style-type: none">• Expected $\left\lceil \frac{\epsilon}{\epsilon - 1} \right\rceil$ periods• Constant (independent of groupsize)
Process Load	<ul style="list-style-type: none">• Constant per period• $< 8L^*$ for 15% loss
False Positive Rate	<ul style="list-style-type: none">• Tunable (via K)• Falls exponentially as load is scaled
Completeness	<ul style="list-style-type: none">• Deterministic time-bounded• Within $O(\log(N))$ periods w.h.p.

ACCURACY, LOAD

- $PM(T)$ is exponential in $-K$. Also depends on pml (and pf)
- See paper

$$\frac{L}{L^*} < 28$$

$$\frac{E[L]}{L^*} < 8$$

for up to 15 % loss rates

DETECTION TIME

- Prob. of being pinged in T' = $1 - (1 - \frac{1}{N})^{N-1} = 1 - e^{-1}$
- $E[T] = T! \cdot \frac{e}{e-1}$
- Completeness: *Any* alive member detects failure
 - Eventually
 - By using a trick: within worst case $O(N)$ protocol periods

TIME-BOUNDED COMPLETENESS

- Key: select each membership element once as a ping target in a traversal
 - Round-robin pinging
 - Random permutation of list after each traversal
- Each failure is detected in worst case $2N-1$ (local) protocol periods
- Preserves FD properties

NEXT

- How do failure detectors fit into the big picture of a group membership protocol?
- What are the missing blocks?



CLOUD COMPUTING CONCEPTS

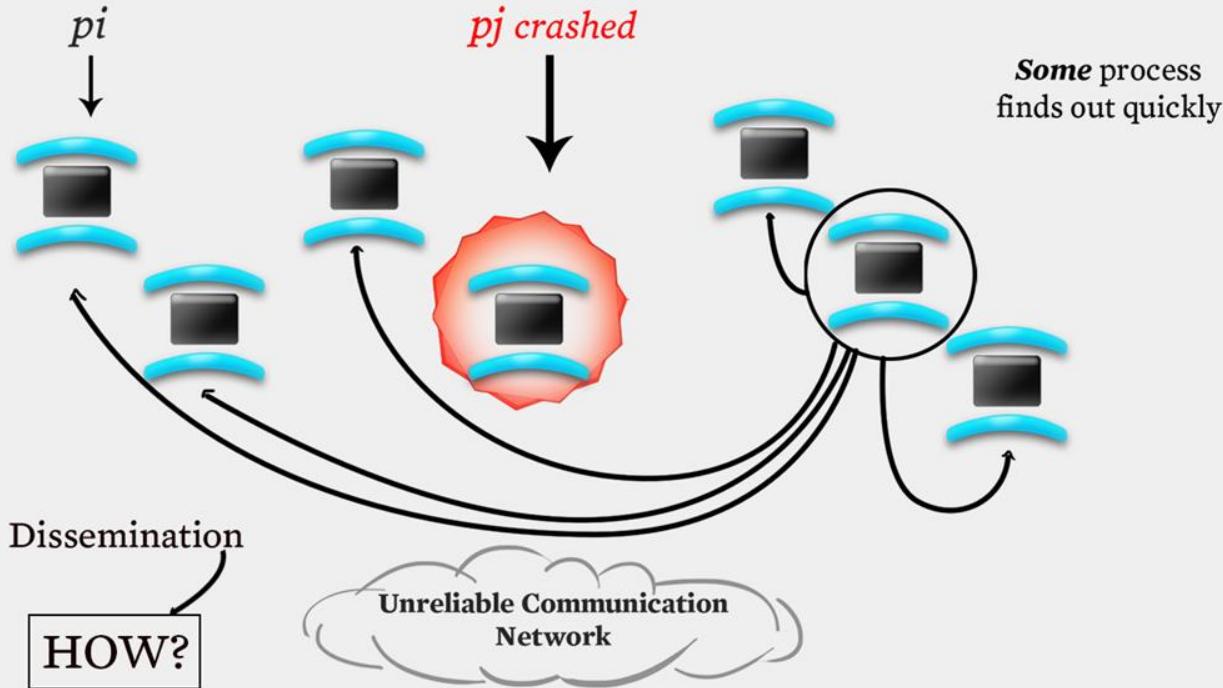
with Indranil Gupta (Indy)

MEMBERSHIP

Lecture F

DISSEMINATION AND SUSPICION

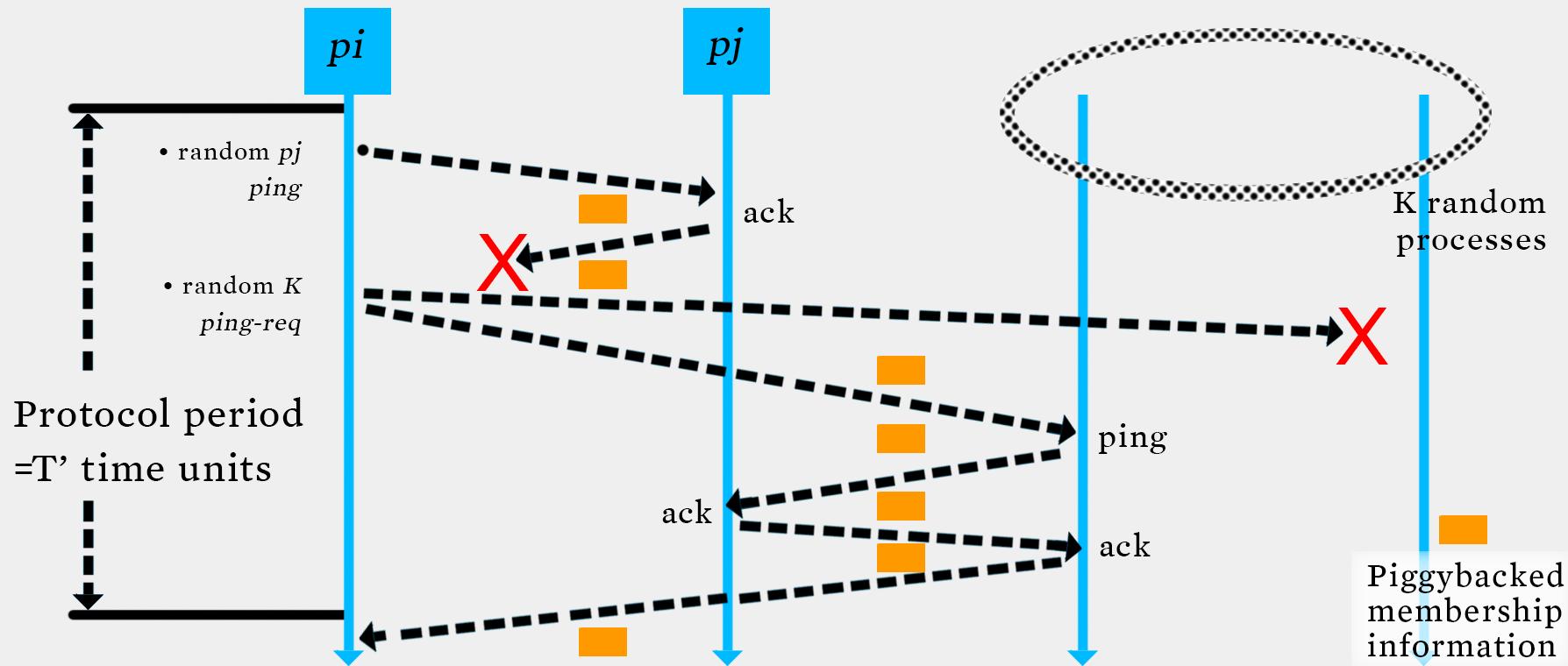
III. DISSEMINATION



DISSEMINATION OPTIONS

- Multicast (Hardware / IP)
 - unreliable
 - multiple simultaneous multicasts
- Point-to-point (TCP / UDP)
 - expensive
- Zero extra messages: Piggyback on Failure Detector messages
 - Infection-style Dissemination

SWIM FAILURE DETECTOR PROTOCOL



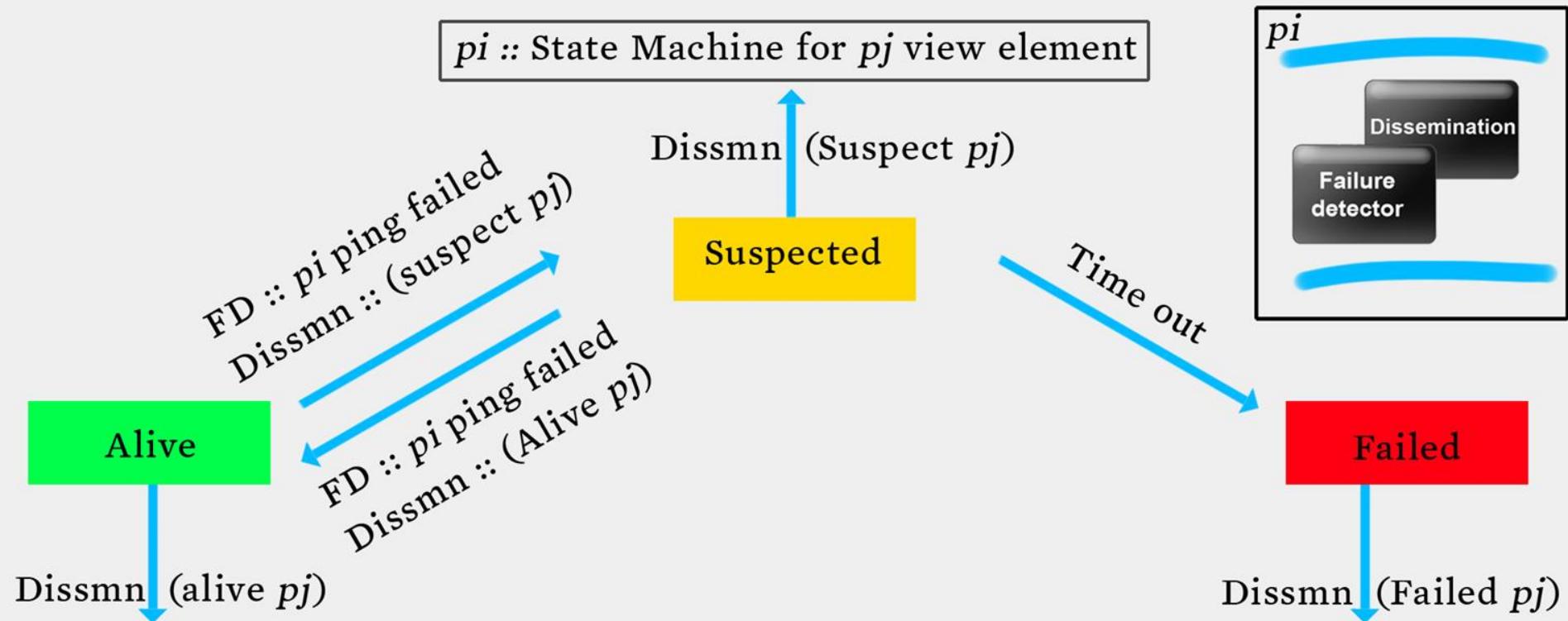
INFECTION-STYLE DISSEMINATION

- Epidemic style dissemination
- After $\lambda \log(N)$ protocol periods, $N^{-(2\lambda-2)}$ processes would not have heard about an update
- Maintain a buffer of recently joined/evicted processes
- Piggyback from this buffer
- Prefer recent updates
- Buffer elements are garbage collected after a while
- After $\lambda \log(N)$ protocol periods; this defines weak consistency

SUSPICION MECHANISM

- False detections, due to:
 - Perturbed processes
 - Packet losses, e.g., from congestion
- Indirect pinging may not solve the problem
 - e.g., correlated message losses near pinged host
- Key: *suspect* a process before *declaring* it as failed in the group

SUSPICION MECHANISM



SUSPICION MECHANISM

- Distinguish multiple suspicions of a process
 - Per-process *incarnation number*
 - *Inc #* for π_i can be incremented only by π_i
 - e.g., when it receives a (Suspect, π_i) message
 - Somewhat similar to DSDV
- Higher inc# notifications over-ride lower inc#'s
- Within an inc#: (Suspect inc #) > (Alive, inc #)
- (Failed, inc #) overrides everything else

WRAP UP

- Failures the norm, not the exception in datacenters
- Every distributed system uses a failure detector
- Many distributed systems use a membership service
- Ring failure detection underlies
 - IBM SP2 and many other similar clusters/machines
- Gossip-style failure detection underlies
 - Amazon EC2/S3 (rumored!)

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

P2P SYSTEMS

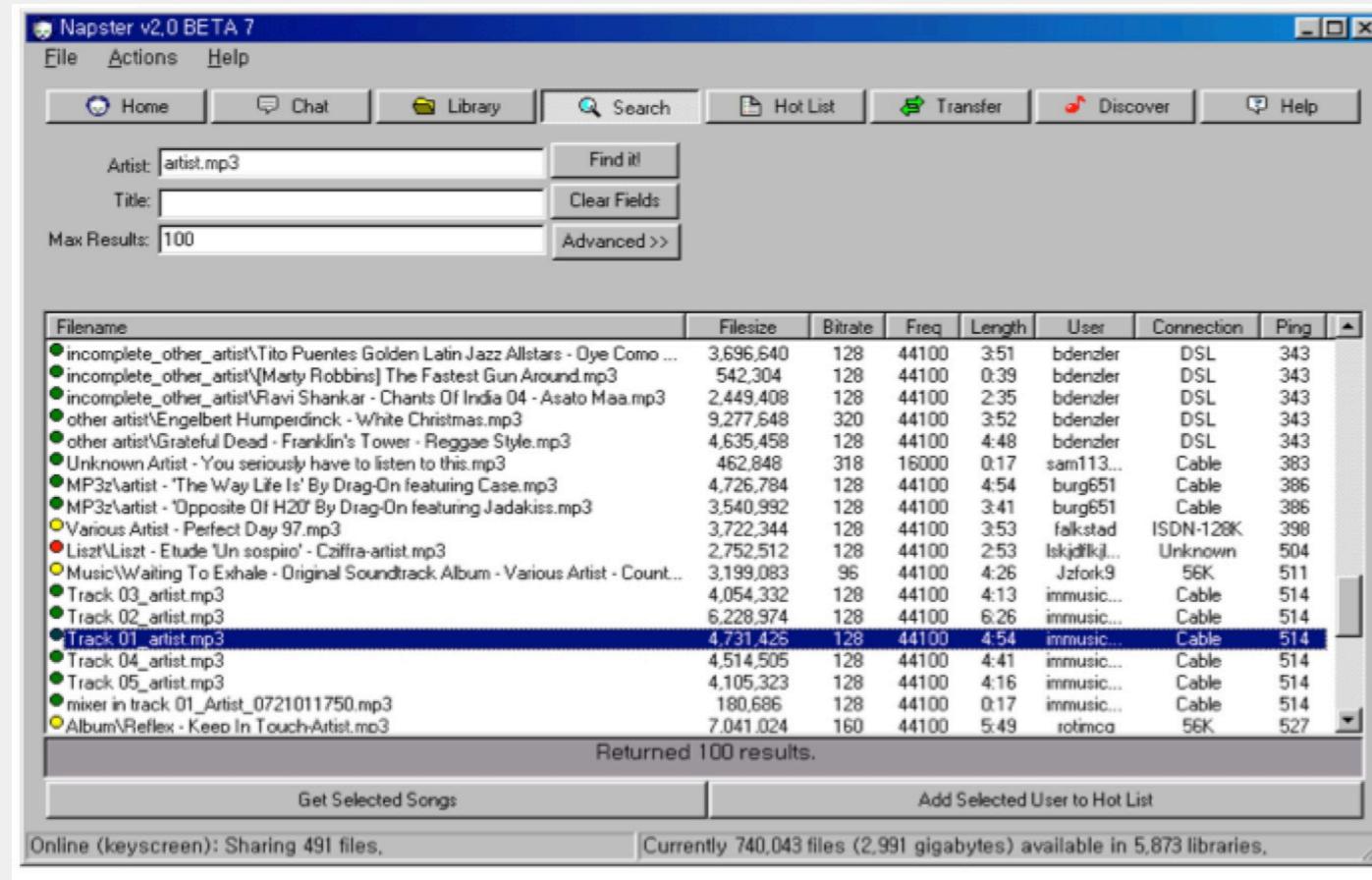
Lecture A

INTRODUCTION

WHY STUDY PEER-TO-PEER SYSTEMS?

- First distributed systems that seriously focused on scalability with respect to number of nodes
- P2P techniques abound in cloud computing systems
 - Key-value stores (e.g., Cassandra, Riak, Voldemort) use Chord p2p hashing

WHY STUDY PEER TO PEER SYSTEMS?



A BRIEF HISTORY

- [6/99] Shawn Fanning (freshman Northeastern University) releases Napster online music service
- [12/99] RIAA sues Napster, asking \$100K per download
- [3/00] 25% University of Wisconsin traffic Napster, many universities ban it
- [00] 60M users
- [2/01] US Federal Appeals Court: users violating copyright laws, Napster is abetting this
- [9/01] Napster decides to run paid service, pay % to songwriters and music companies
- [Today] Napster protocol is open, people free to develop OpenNap clients and servers
<http://opennap.sourceforge.net>
 - Gnutella: <http://www.limewire.com> (deprecated)
 - Peer-to-peer working groups: <http://p2p.internet2.edu>

WHAT WE WILL STUDY

- Widely-deployed P2P Systems
 1. Napster
 2. Gnutella
 3. Fasttrack (Kazaa, Kazaalite, Grokster)
 4. BitTorrent
- P2P Systems with Provable Properties
 1. Chord
 2. Pastry
 3. Kelips

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

P2P SYSTEMS

Lecture B

NAPSTER

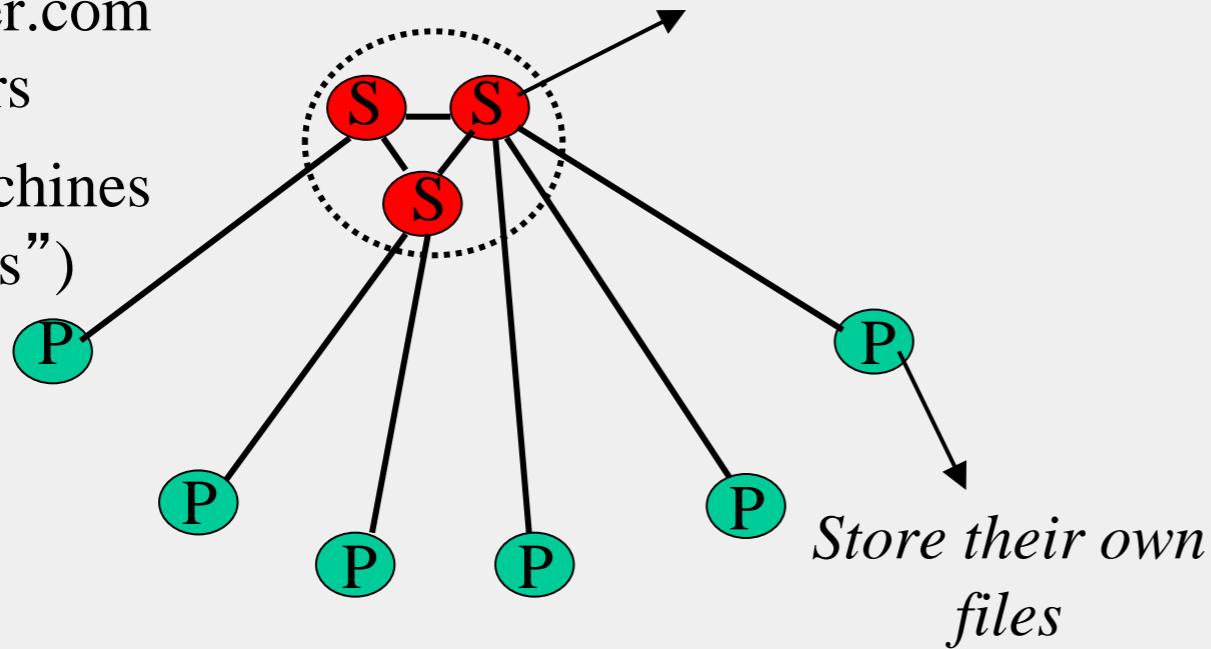
NAPSTER STRUCTURE

*Store a directory, i.e.,
filenames with peer pointers*

Filename	Info about
PennyLane.mp3	Beatles, @ 128.84.92.23:1006
.....	

napster.com
Servers

Client machines
("Peers")



NAPSTER OPERATIONS

Client

- Connect to a Napster server
 - Upload list of music files that you want to share
 - Server maintains list of <filename, ip_address, portnum> tuples. **Server stores no files.**

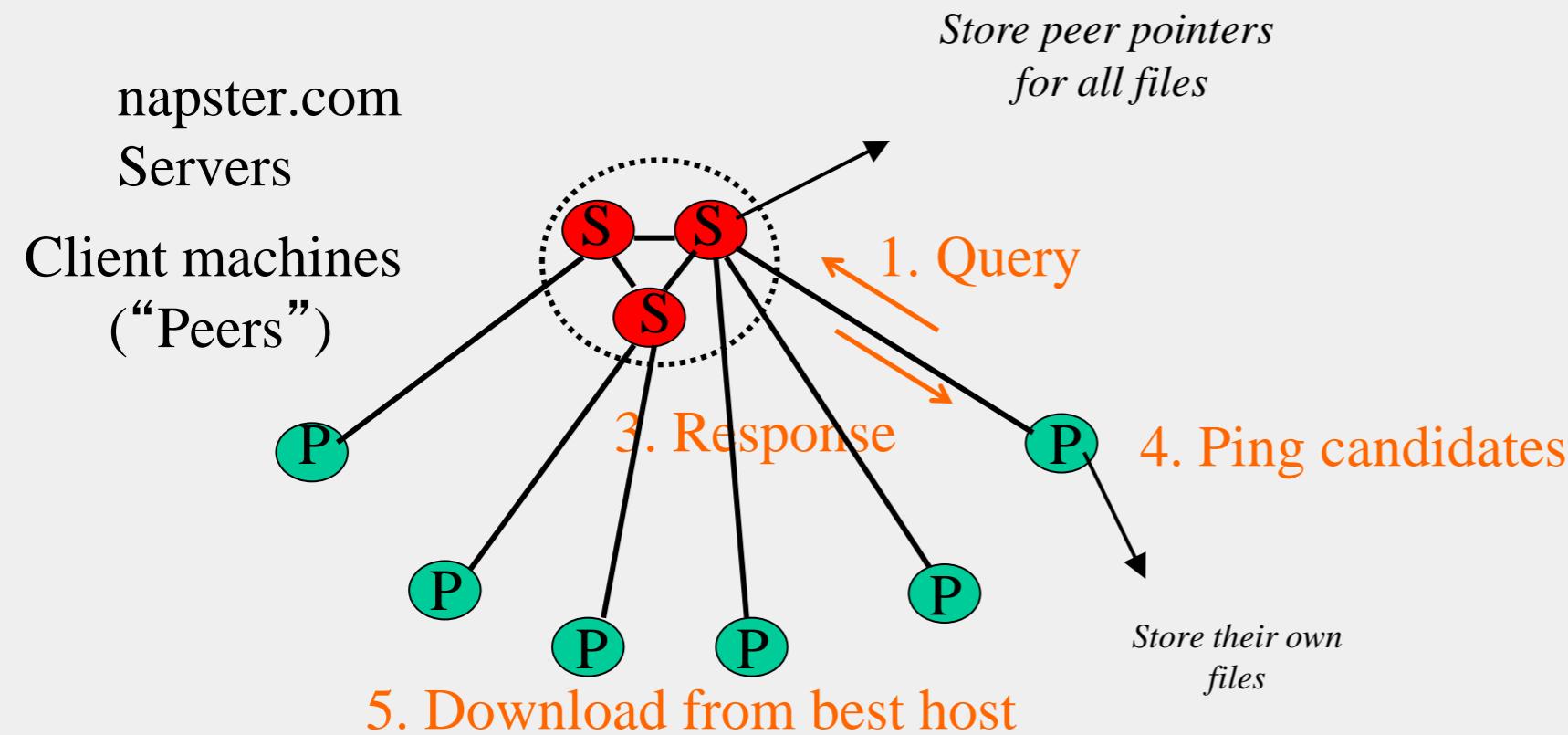
NAPSTER OPERATIONS

Client (contd.)

- Search
 - Send server keywords to search with
 - (Server searches its list with the keywords)
 - Server returns a list of hosts – <ip_address, portnum> tuples – to client
 - Client pings each host in the list to find transfer rates
 - Client fetches file from best host
- All communication uses TCP (Transmission Control Protocol)
 - Reliable and ordered networking protocol

NAPSTER SEARCH

2. All servers search their lists (ternary tree algorithm)



JOINING A P2P SYSTEM

- Can be used for any p2p system
 - Send an http request to well-known url for that P2P service - `http://www.myp2pservice.com`
 - Message routed (after lookup in DNS=Domain Name System) to introducer, a well known server that keeps track of some recently joined nodes in p2p system
 - Introducer initializes new peers' neighbor table

PROBLEMS

- Centralized server a source of congestion
- Centralized server single point of failure
- No security: plaintext messages and passwd
- Napster.com declared to be responsible for users' copyright violation
 - “Indirect infringement”
 - Next system: Gnutella

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

P2P SYSTEMS

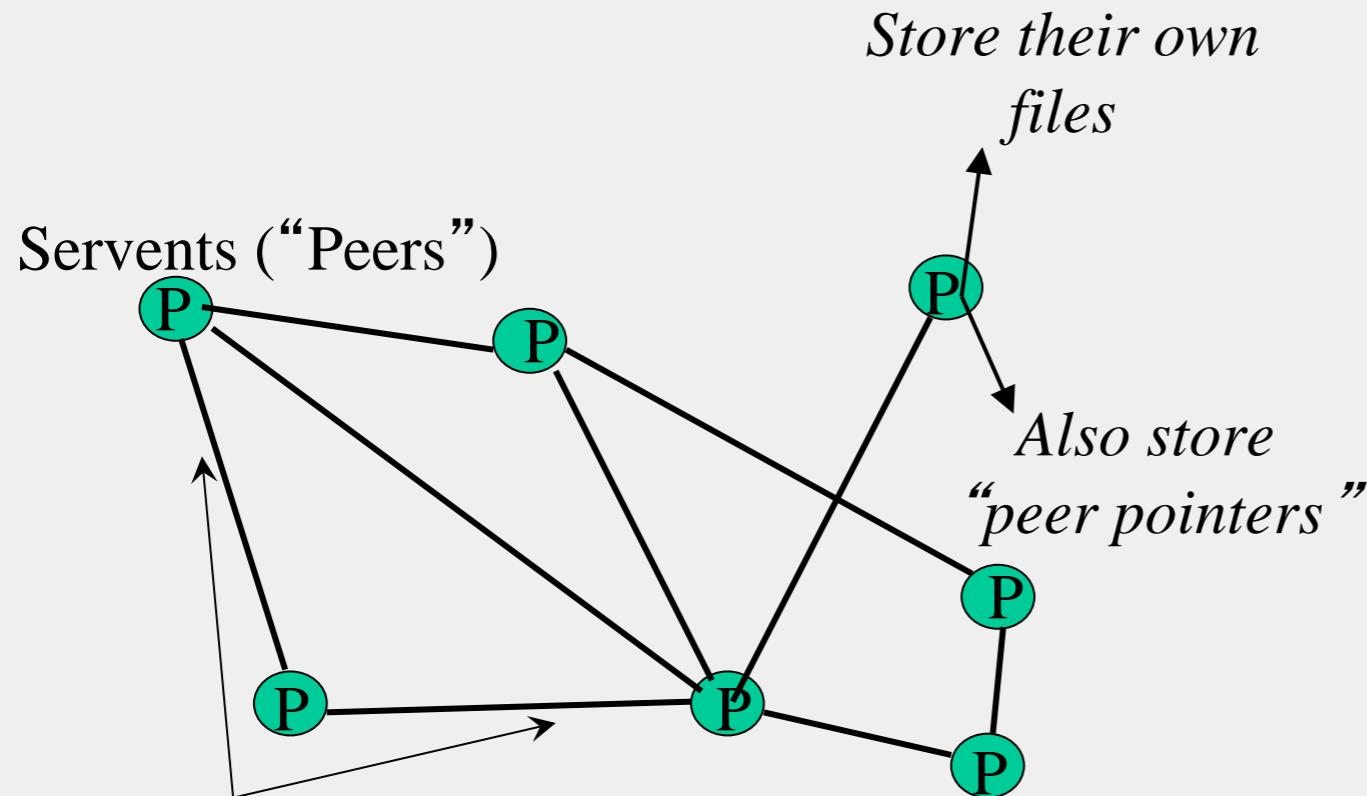
Lecture C

GNUTELLA

GNUTELLA

- Eliminate the servers
- Client machines search and retrieve amongst themselves
- Clients act as servers too, called **servents**
- [3/00] release by AOL, immediately withdrawn, but 88K users by 3/03
- Original design underwent several modifications

GNUTELLA



Connected in an **overlay** graph
(== each link is an implicit Internet path)

HOW DO I SEARCH FOR MY BEATLES FILE?

- Gnutella *routes* different messages within the overlay graph
- Gnutella protocol has 5 main message types
 - **Query** (search)
 - **QueryHit** (response to query)
 - **Ping** (to probe network for other peers)
 - **Pong** (reply to ping, contains address of another peer)
 - Push (used to initiate file transfer)
- We'll go into the message structure and protocol now
 - All fields except IP address are in little-endian format
 - Ox12345678 stored as 0x78 in lowest address byte, then 0x56 in next higher address, and so on.

HOW DO I SEARCH FOR MY BEATLES FILE?

Descriptor Header



0
↓
ID of this search transaction

15
↓
Type of payload
0x00 Ping
0x01 Pong
0x40 Push
0x80 Query
0x81 Queryhit

16
↓
*Decrement at each hop,
Message dropped when ttl=0
ttl_initial usually 7 to 10*

Payload

17 18
↓
Number of bytes of message following this header

22
↓
Incremented at each hop

Gnutella Message Header Format

How do I SEARCH FOR MY BEATLES FILE?

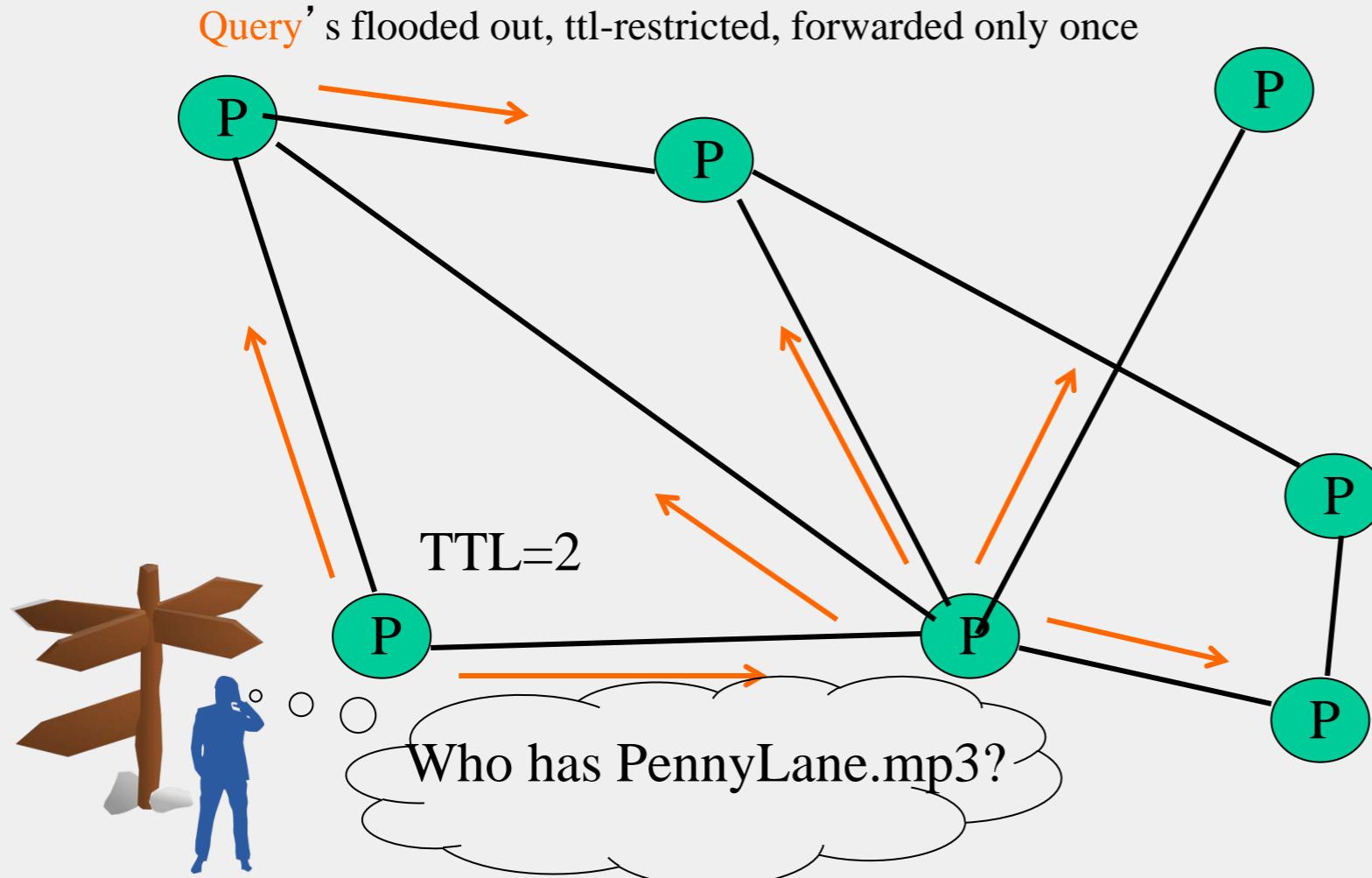
Query (0x80)

Minimum Speed	Search criteria (keywords)
---------------	----------------------------

0 1

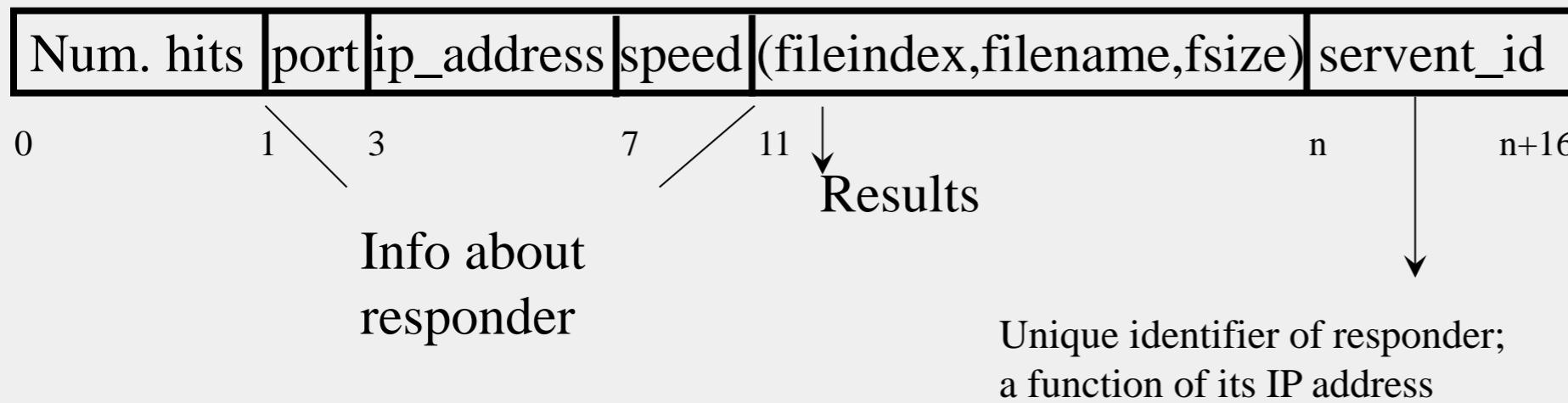
Payload Format in Gnutella **Query** Message

GNUTELLA SEARCH



GNUTELLA SEARCH

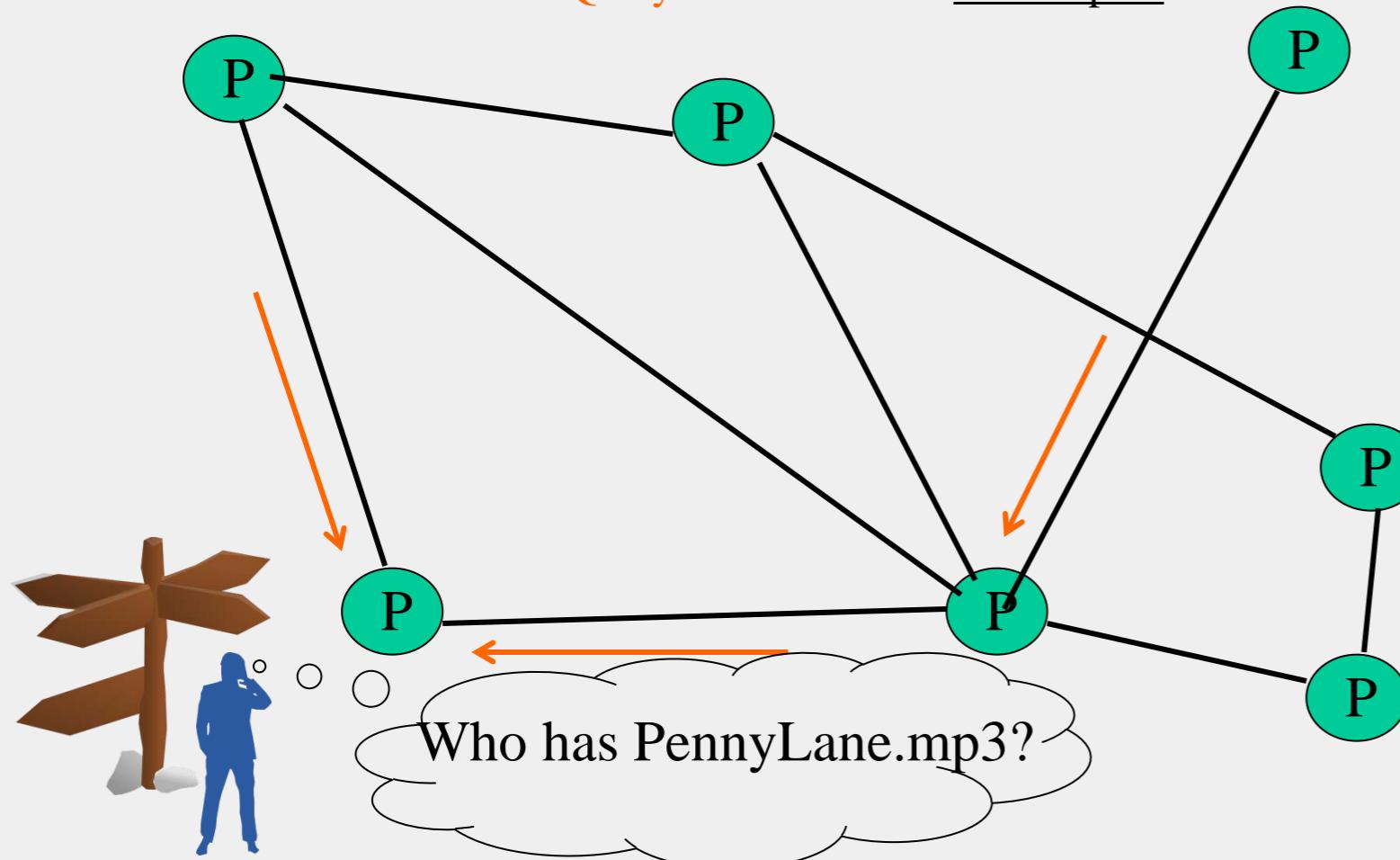
QueryHit (0x81) : successful result to a query



Payload Format in Gnutella **QueryHit** Message

GNUTELLA SEARCH

Successful results **QueryHit**'s routed on reverse path



AVOIDING EXCESSIVE TRAFFIC

- To avoid duplicate transmissions, each peer maintains a list of recently received messages
- Query forwarded to all neighbors except peer from which received
- Each Query (identified by DescriptorID) forwarded only once
- QueryHit routed back only to peer from which Query received with same DescriptorID
- Duplicates with same DescriptorID and Payload descriptor (msg type) are dropped
- QueryHit with DescriptorID for which Query not seen is dropped

AFTER RECEIVING QUERYHIT MESSAGES

- Requestor chooses “best” QueryHit responder
 - Initiates HTTP request directly to responder’s ip+port

GET /get/<File Index>/<File Name>/HTTP/1.0\r\n

Connection: Keep-Alive\r\n

Range: bytes=0-\r\n

User-Agent: Gnutella\r\n

\r\n

- Responder then replies with file packets after this message:

HTTP 200 OK\r\n

Server: Gnutella\r\n

Content-type:application/binary\r\n

Content-length: 1024 \r\n

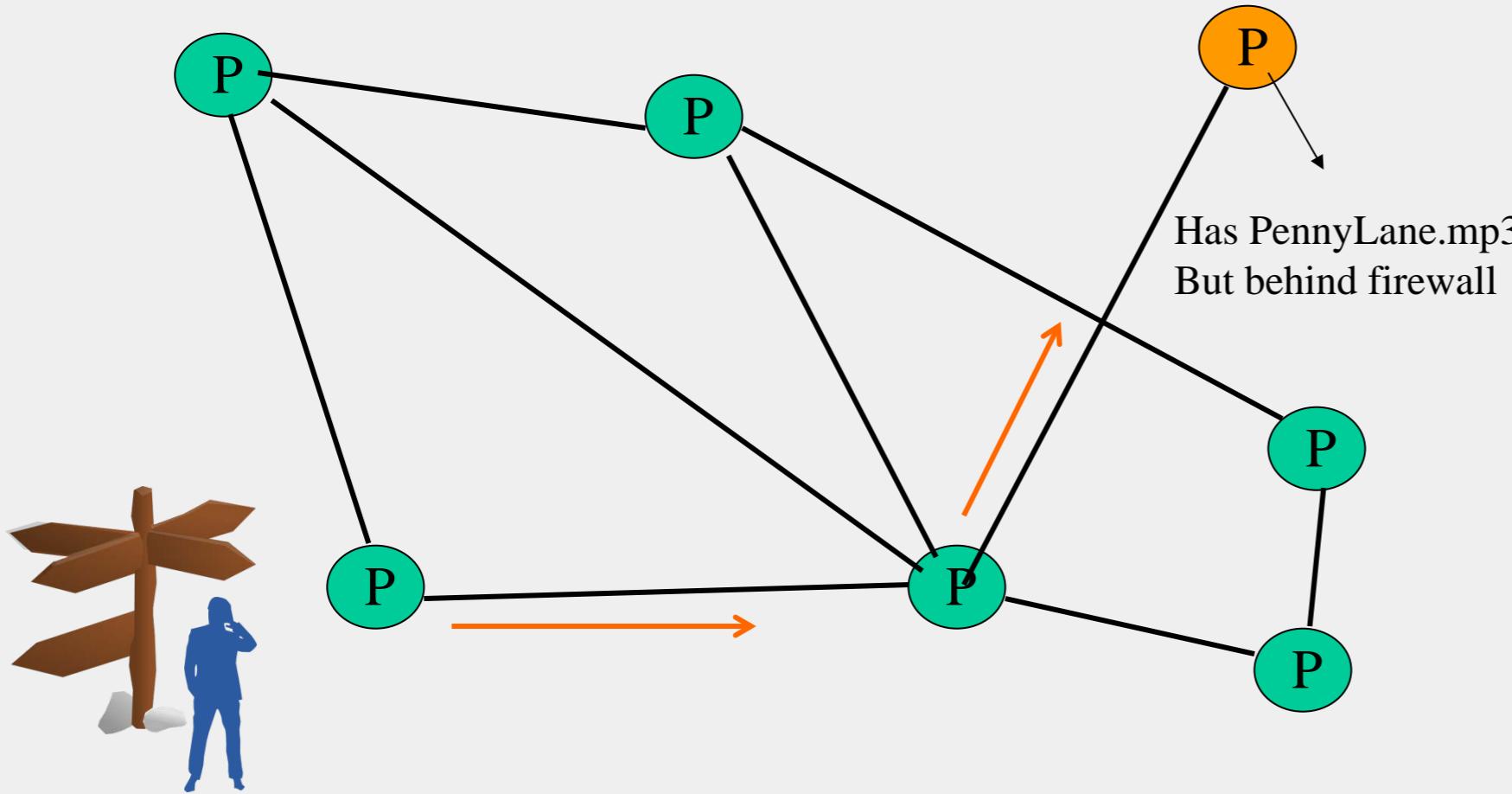
\r\n

AFTER RECEIVING QUERY HIT MESSAGES (2)

- HTTP is the file transfer protocol. Why?
 - Because it's standard, well-debugged, and widely used.
- Why the “range” field in the GET request?
 - To support partial file transfers.
- What if responder is behind firewall that disallows incoming connections?

DEALING WITH FIREWALLS

Requestor sends **Push** to responder asking for file transfer



DEALING WITH FIREWALLS

Push (0x40)

servent_id	fileindex	ip_address	port
------------	-----------	------------	------

same as in
received QueryHit

Address at which
requestor can accept
incoming connections

DEALING WITH FIREWALLS

- Responder establishes a TCP connection at ip_address, port specified. Sends

GIV <File Index>:<Servent Identifier>/<File Name>\n\n
- Requestor then sends GET to responder (as before) and file is transferred as explained earlier
- What if requestor is behind firewall too?
 - Gnutella gives up
 - Can you think of an alternative solution?

PING-PONG

Ping (0x00)

no payload

Pong (0x01)

Port	ip_address	Num. files shared	Num. KB shared
------	------------	-------------------	----------------

- Peers initiate Ping's periodically
- Ping's flooded out like Query's, Pong's routed along reverse path like QueryHit's
- Pong replies used to update set of neighboring peers
 - To keep neighbor lists fresh in spite of peers joining, leaving and failing

GNUTELLA SUMMARY

- No servers
- Peers/servents maintain “neighbors,” this forms an overlay graph
- Peers store their own files
- Queries flooded out, ttl restricted
- QueryHit (replies) reverse path routed
- Supports file transfer through firewalls
- Periodic ping-pong to continuously refresh neighbor lists
 - List size specified by user at peer: heterogeneity means some peers may have more neighbors
 - Gnutella found to follow **power law** distribution:
$$P(\#links = L) \sim L^{-k} \quad (k \text{ is a constant})$$

PROBLEMS

- Ping/Pong constituted 50% traffic
 - Solution: Multiplex, *cache* and reduce frequency of pings/pongs
- Repeated searches with same keywords
 - Solution: *Cache* Query, QueryHit messages
- Modem-connected hosts do not have enough bandwidth for passing Gnutella traffic
 - Solution: use a central server to act as proxy for such peers
 - Another solution:
 - ➔ FastTrack System (soon)

PROBLEMS (CONTD.)

- Large number of *freeloaders*
 - 70% of users in 2000 were freeloaders
 - Only download files, never upload own files
- Flooding causes excessive traffic
 - Is there some way of maintaining meta-information about peers that leads to more intelligent routing?
 - ➔ Structured peer-to-peer systems
 - e.g., Chord System (coming up soon)

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

P2P SYSTEMS

Lecture D

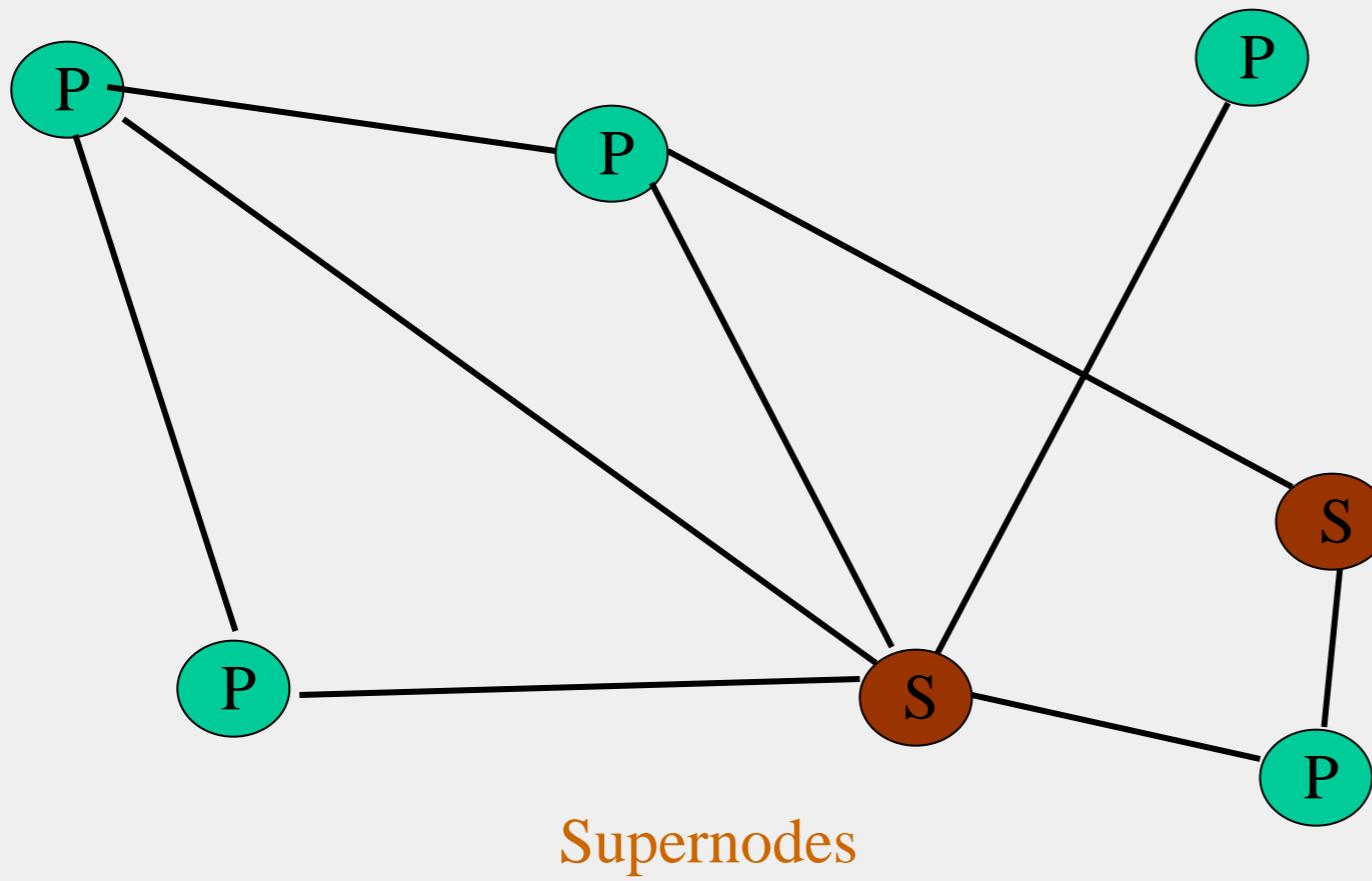
FASTTRACK AND BITTORENT

FASTTRACK

- Hybrid between Gnutella and Napster
- Takes advantage of “healthier” participants in the system
- Underlying technology in Kazaa, KazaaLite, Grokster
- Proprietary protocol, but some details available
- Like Gnutella, but with some peers designated as *supernodes*

A FASTTRACK-LIKE SYSTEM

Peers

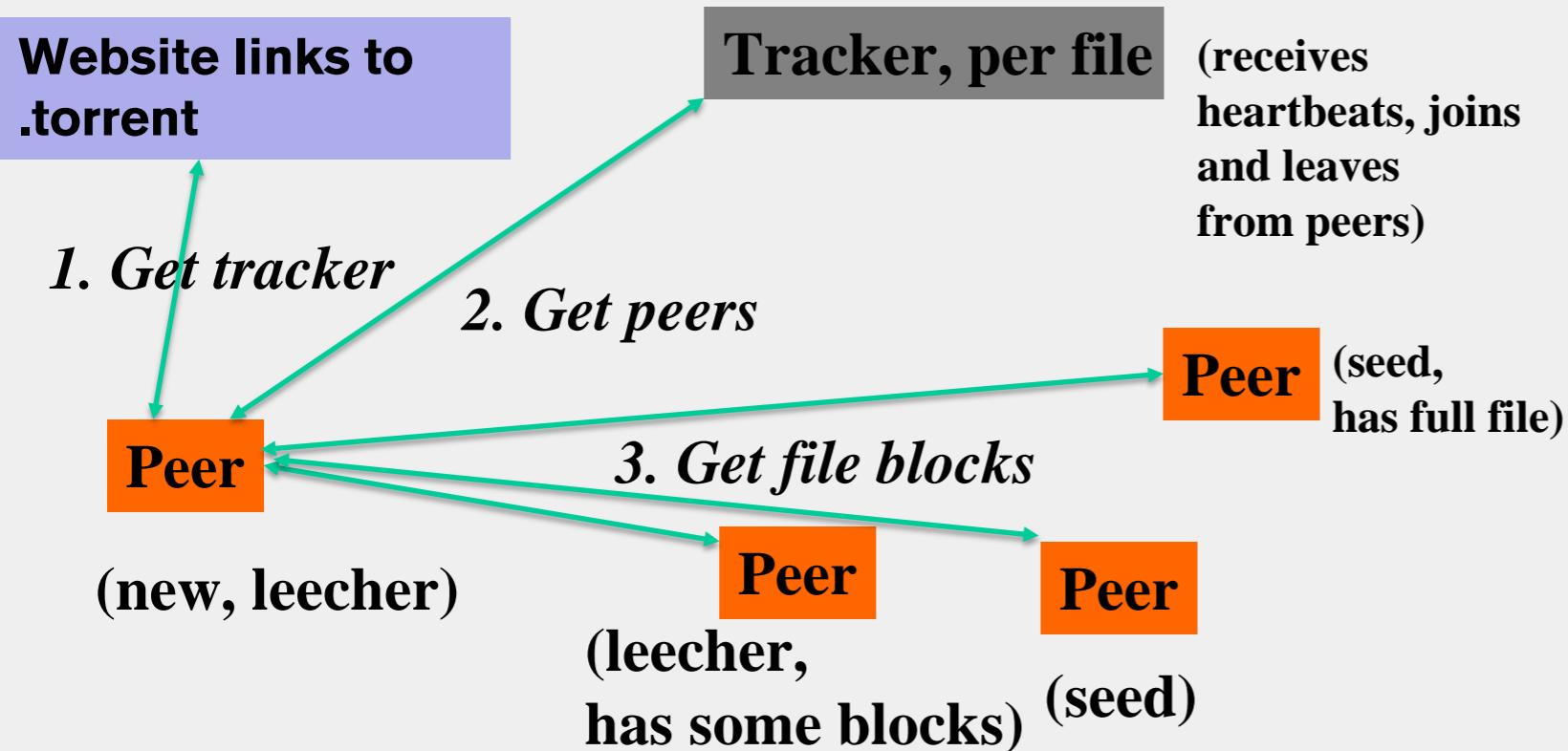


Supernodes

FASTTRACK (CONTD.)

- A supernode stores a directory listing a subset of nearby (<filename, peer pointer>), similar to Napster servers
- Supernode membership changes over time
- Any peer can become (and stay) a supernode, provided it has earned enough *reputation*
 - Kazaalite: participation level (=reputation) of a user between 0 and 1000, initially 10, then affected by length of periods of connectivity and total number of uploads
 - More sophisticated Reputation schemes invented, especially based on economics (See P2PEcon workshop)
- A peer searches by contacting a nearby supernode

BITTORRENT



BITTORRENT (2)

- File split into blocks (32 KB – 256 KB)
- Download **Local Rarest First** block policy: prefer early download of blocks that are least replicated among neighbors
 - Exception: New node allowed to pick one random neighbor: helps in bootstrapping
- **Tit for tat** bandwidth usage: Provide blocks to neighbors that provided it the best download rates
 - Incentive for nodes to provide good download rates
 - Seeds do the same too
- **Choking**: Limit number of neighbors to which concurrent uploads <= a number (5), i.e., the “best” neighbors
 - Everyone else choked
 - Periodically re-evaluate this set (e.g., every 10 s)
 - **Optimistic unchoke**: periodically (e.g., ~30 s), unchoke a random neighbor – helps keep unchoked set fresh

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

P2P SYSTEMS

Lecture E

CHORD

DHT=DISTRIBUTED HASH TABLE

- A hash table allows you to insert, lookup, and delete objects with keys
- A *distributed* hash table allows you to do the same in a distributed setting (objects=files)
- Performance concerns:
 - Load balancing
 - Fault-tolerance
 - Efficiency of lookups and inserts
 - Locality
- Napster, Gnutella, FastTrack are all DHTs (sort of)
- So is Chord, a structured peer-to-peer system that we study next

COMPARATIVE PERFORMANCE

	Memory	Lookup Latency	#Messages for a lookup	
Napster	$O(1)$ $(O(N) \text{ @ server})$	$O(1)$	$O(1)$	
Gnutella	$O(N)$	$O(N)$	$O(N)$	

COMPARATIVE PERFORMANCE

	Memory	Lookup Latency	#Messages for a lookup	
Napster	$O(1)$ $(O(N) \text{ @ server})$	$O(1)$	$O(1)$	
Gnutella	$O(N)$	$O(N)$	$O(N)$	
Chord	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$	

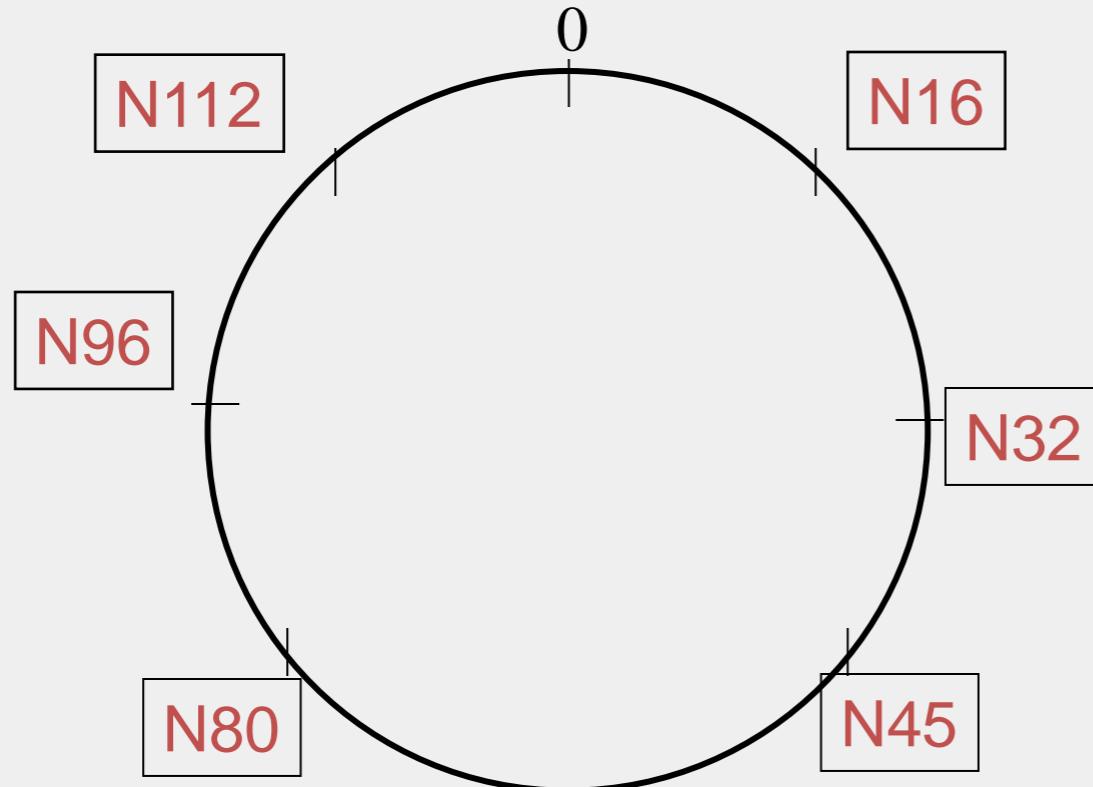
CHORD

- Developers: I. Stoica, D. Karger, F. Kaashoek, H. Balakrishnan, R. Morris, Berkeley, and MIT
- Intelligent choice of neighbors to reduce latency and message cost of routing (lookups/inserts)
- Uses *Consistent Hashing* on node's (peer's) address
 - **SHA-1(ip_address, port) → 160 bit string**
 - Truncated to m bits
 - Called peer id (number between 0 and $2^m - 1$)
 - Not unique but id conflicts very unlikely
 - Can then map peers to one of 2^m logical points on a circle

RING OF PEERS

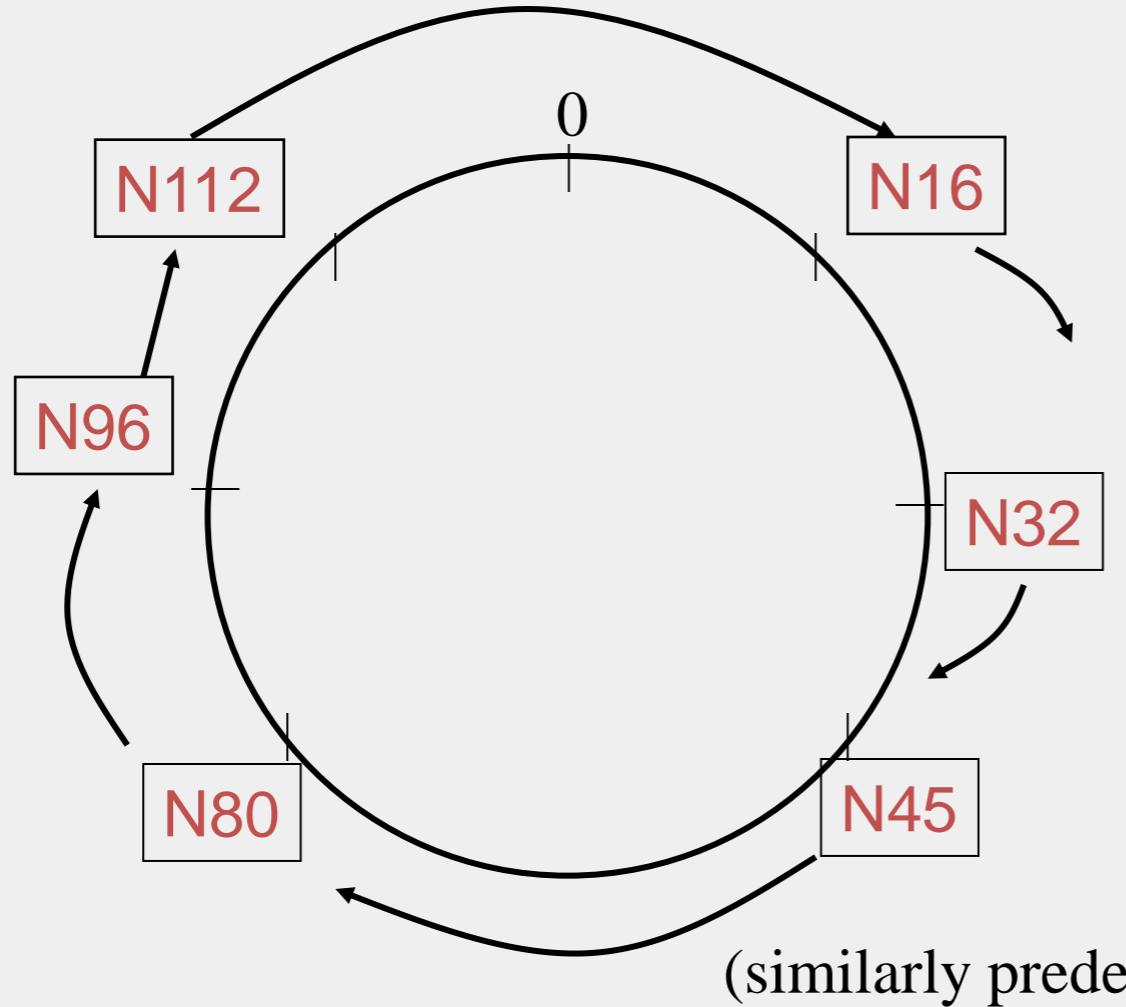
Say $m=7$

6 nodes



PEER POINTERS (1): SUCCESSORS

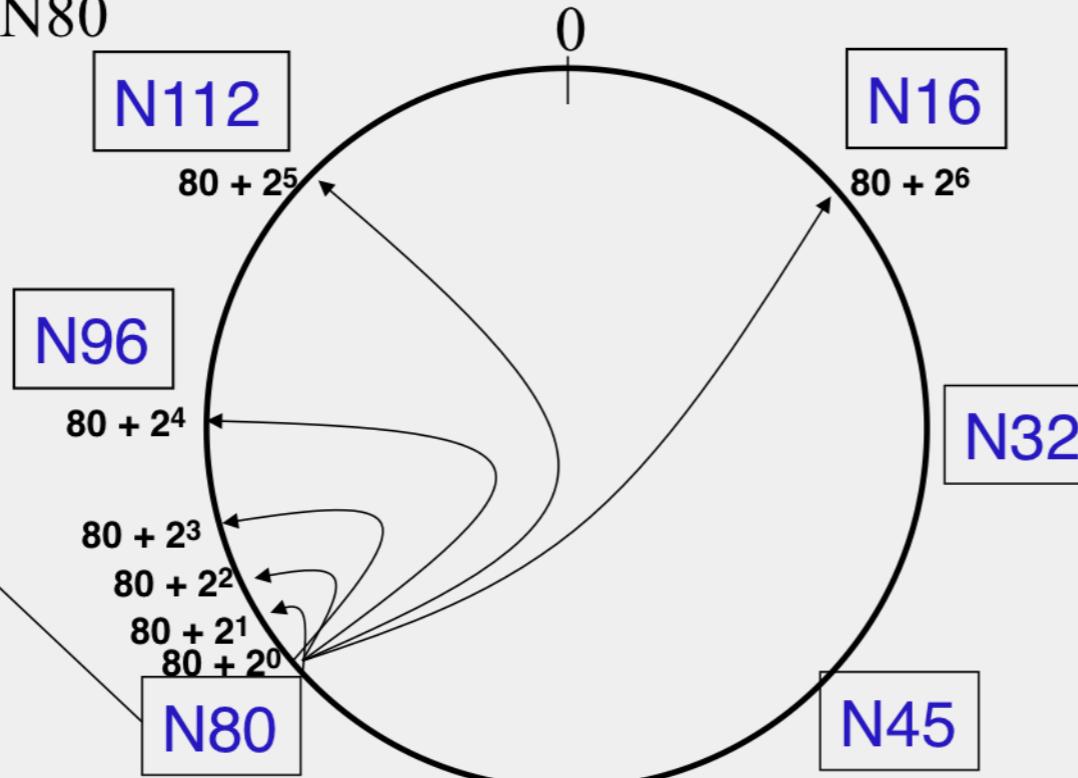
Say $m=7$



PEER POINTERS (2): FINGER TABLES

Finger Table at N80

i	$ft[i]$
0	96
1	96
2	96
3	96
4	96
5	112
6	16



Say $m=7$

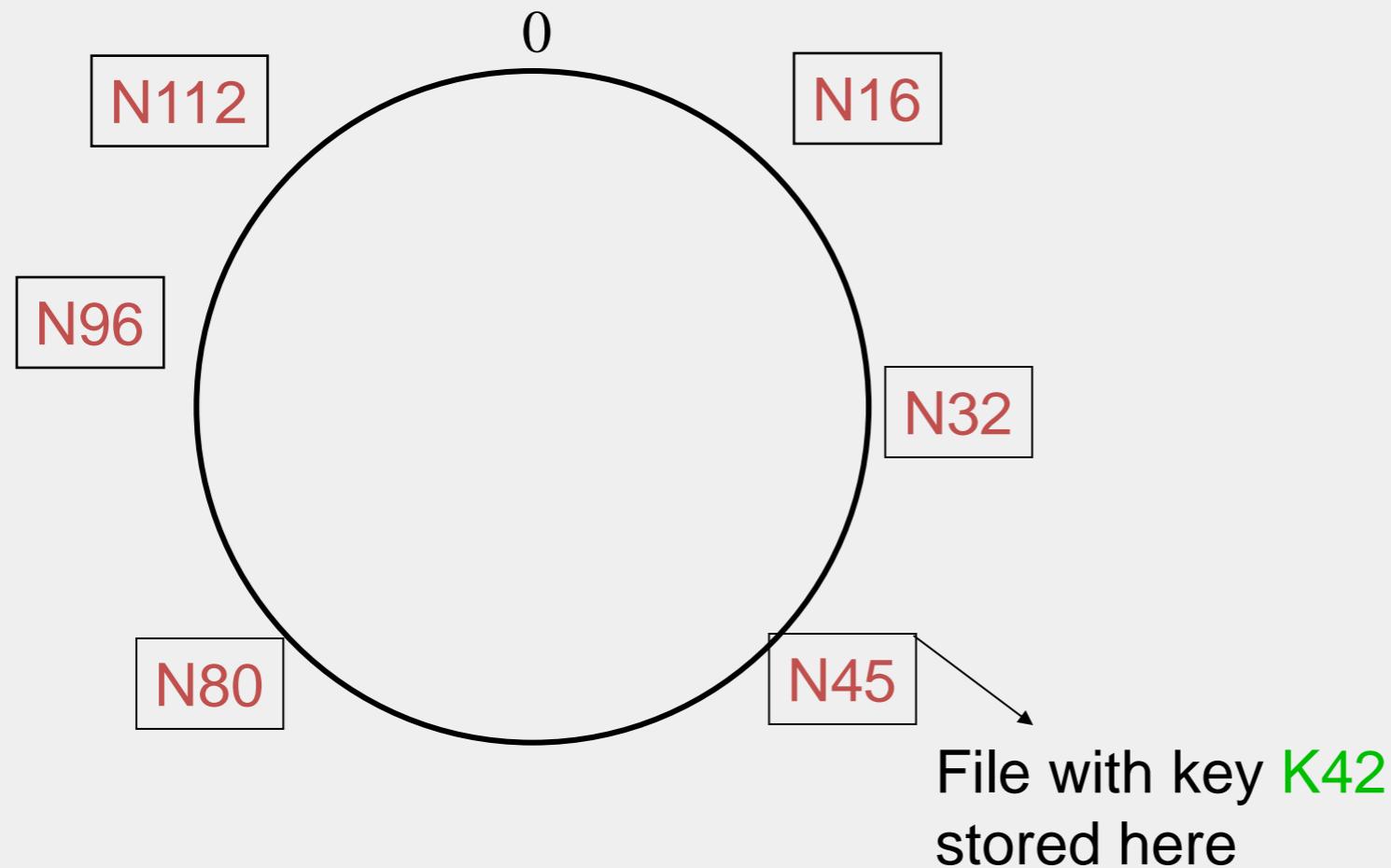
i th entry at peer with id n is first peer with id $\geq (n + 2^i)(\text{mod } 2^m)$

WHAT ABOUT THE FILES?

- Filenames also mapped using same consistent hash function
 - SHA-1(filename) → 160 bit string (*key*)
 - File is stored at first peer with id greater than its *key* ($\text{mod } 2^m$)
- File *cnn.com/index.html* that maps to key K42 is stored at first peer with id greater than 42
 - Note that we are considering a different file-sharing application here: *cooperative web caching*
 - The same discussion applies to any other file sharing application, including that of mp3 files.
- Consistent Hashing => with K keys and N peers, each peer stores $O(K/N)$ keys. (i.e., $< c.K/N$, for some constant c)

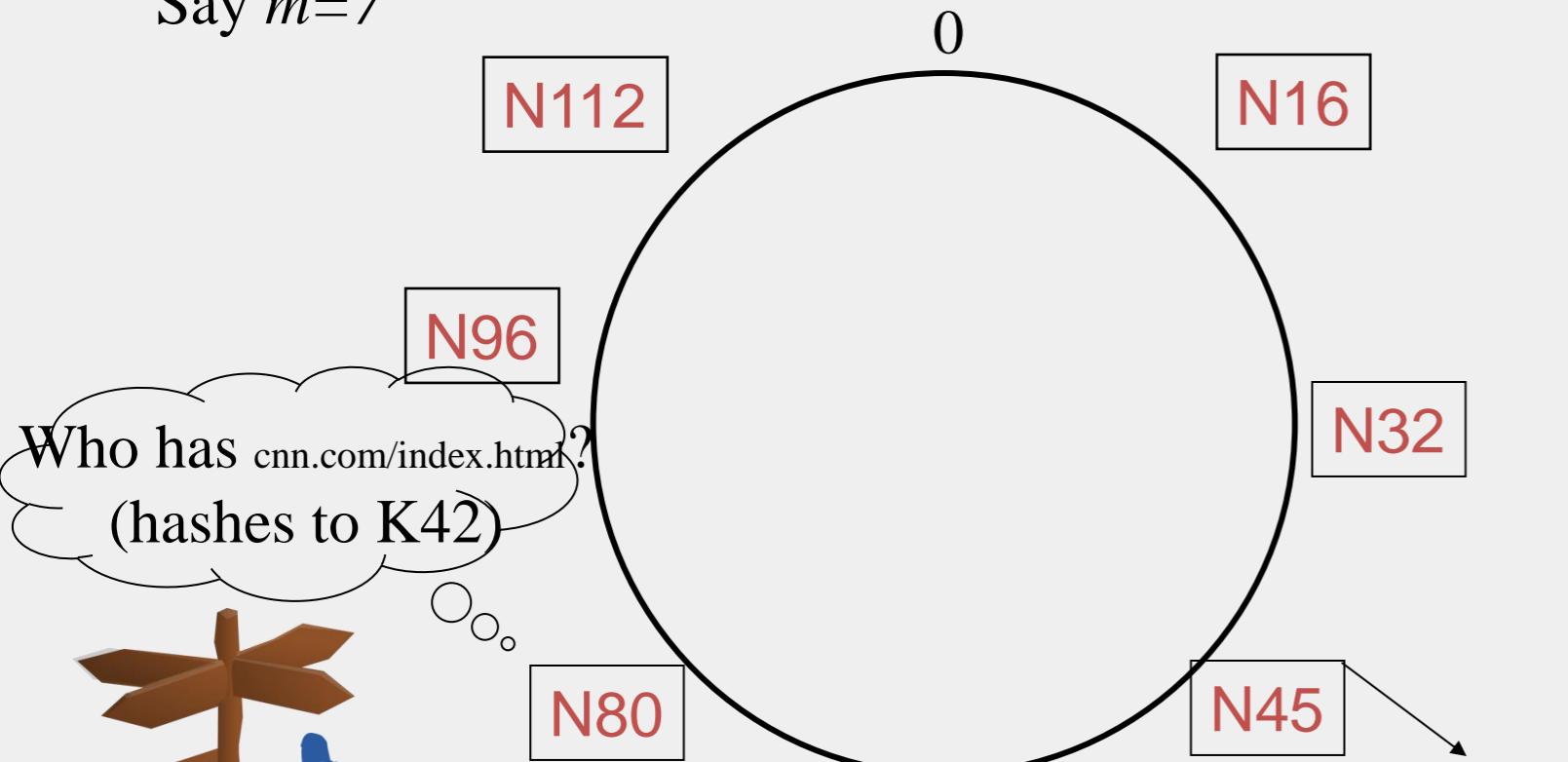
MAPPING FILES

Say $m=7$



SEARCH

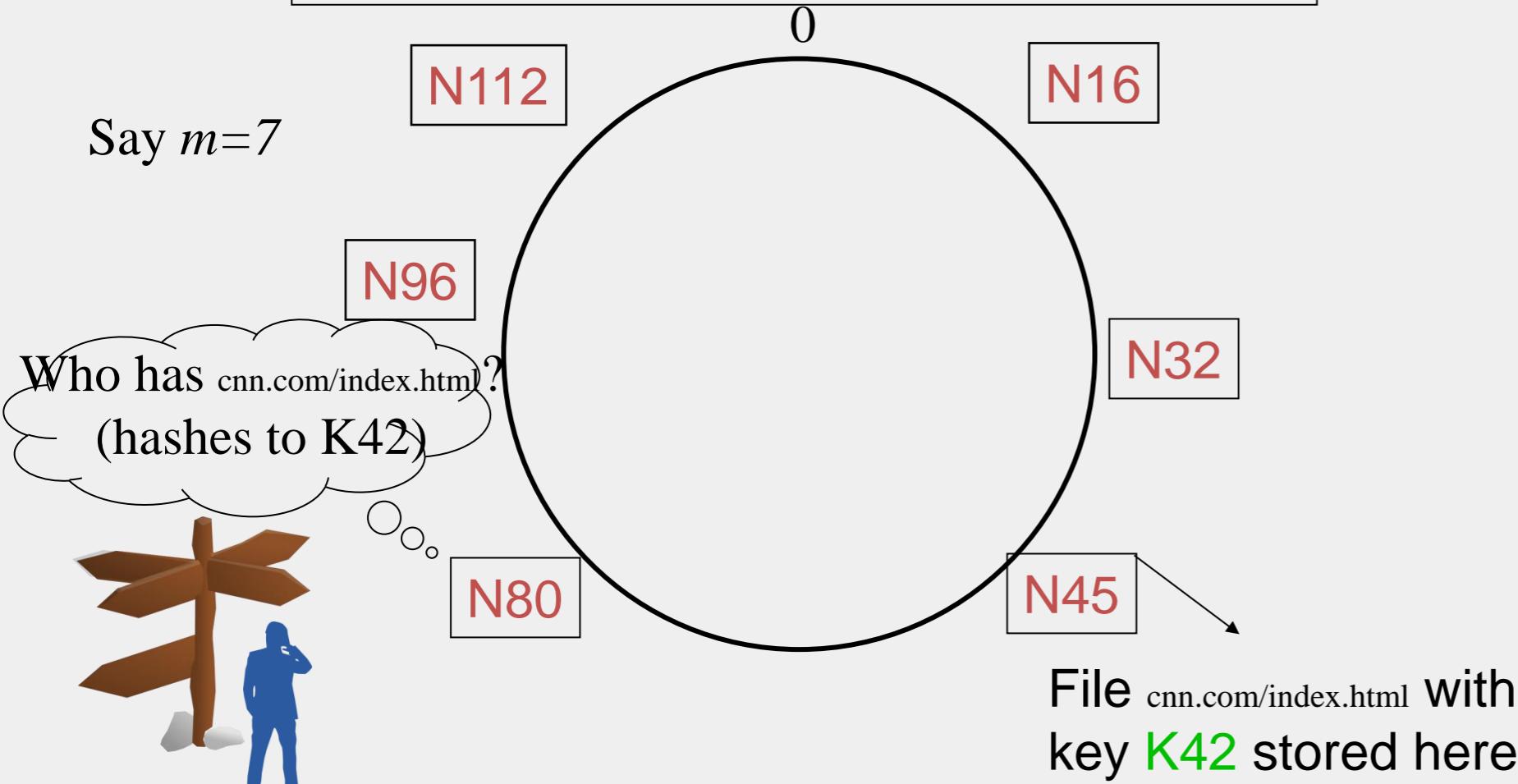
Say $m=7$



File `cnn.com/index.html` with
key **K42** stored here

SEARCH

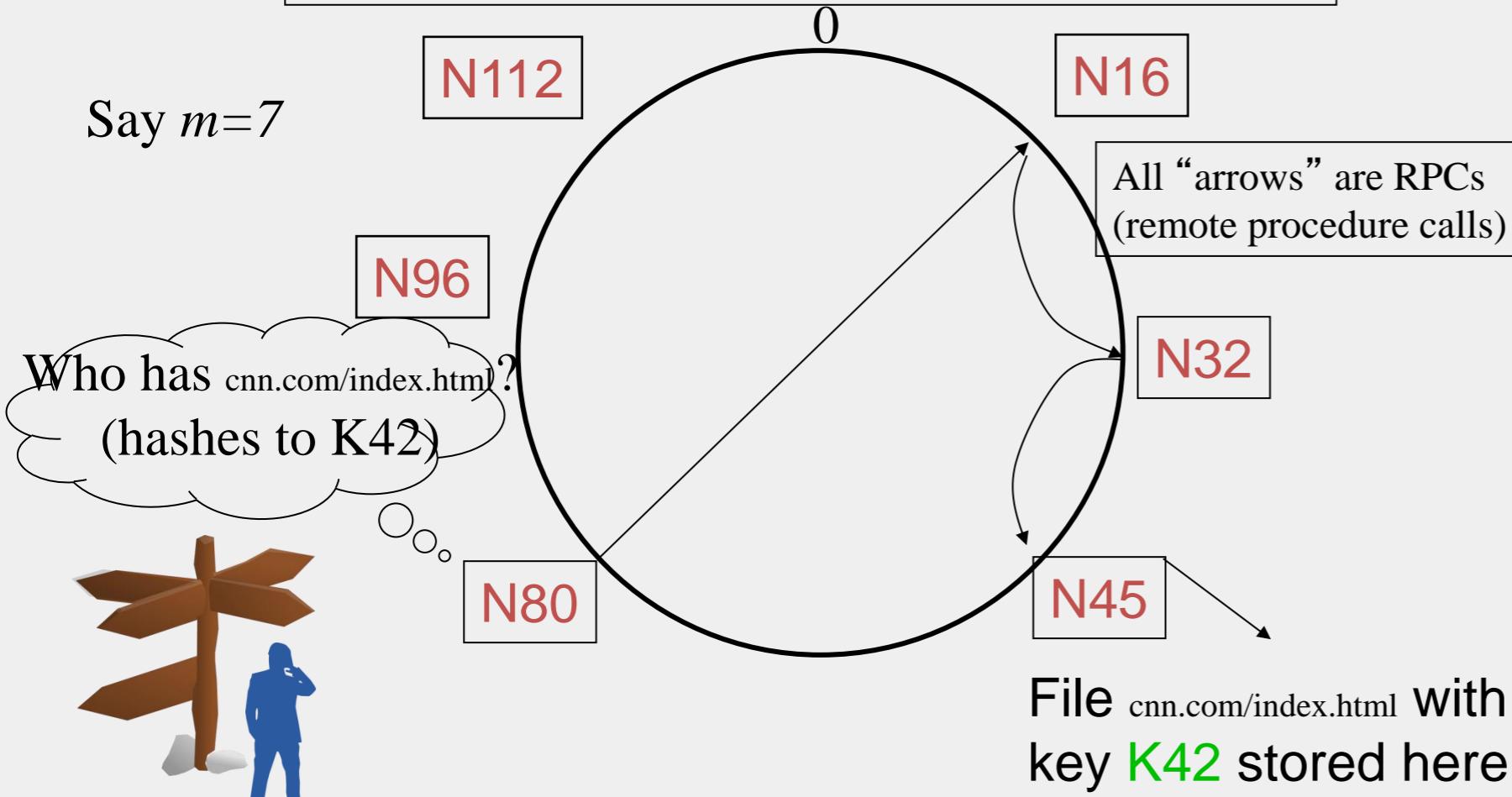
At node n , send query for key k to largest successor/finger entry $\leq k$
if none exist, send query to $\text{successor}(n)$



SEARCH

At node n , send query for key k to largest successor/finger entry $\leq k$
if none exist, send query to $\text{successor}(n)$

Say $m=7$

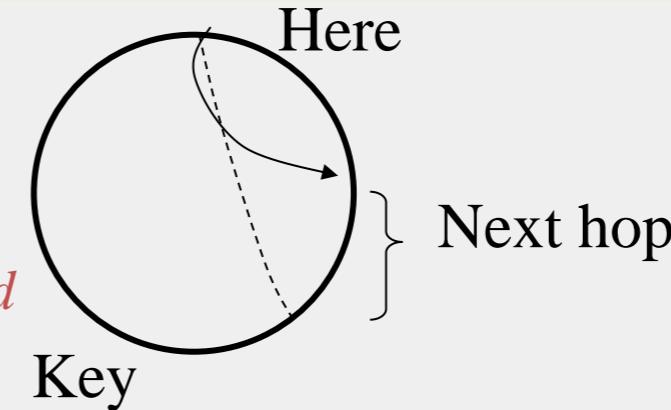


ANALYSIS

Search takes $O(\log(N))$ time

Proof

- (Intuition): *at each step, distance between query and peer-with-file reduces by a factor of at least 2*



- (Intuition): after $\log(N)$ forwardings, distance to key is at most $2^m / 2^{\log(N)} = 2^m / N$
- Number of node identifiers in a range of $2^m / N$ is $O(\log(N))$ with high probability (why? SHA-1! and “Balls and Bins”)

So using *successors* in that range will be ok, using another $O(\log(N))$ hops

ANALYSIS (CONTD.)

- $O(\log(N))$ search time holds for file insertions too (in general for *routing to any key*)
 - “Routing” can thus be used as a **building block** for
 - All operations: insert, lookup, delete
- $O(\log(N))$ time true only if finger and successor entries correct
- When might these entries be wrong?
 - When you have failures

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

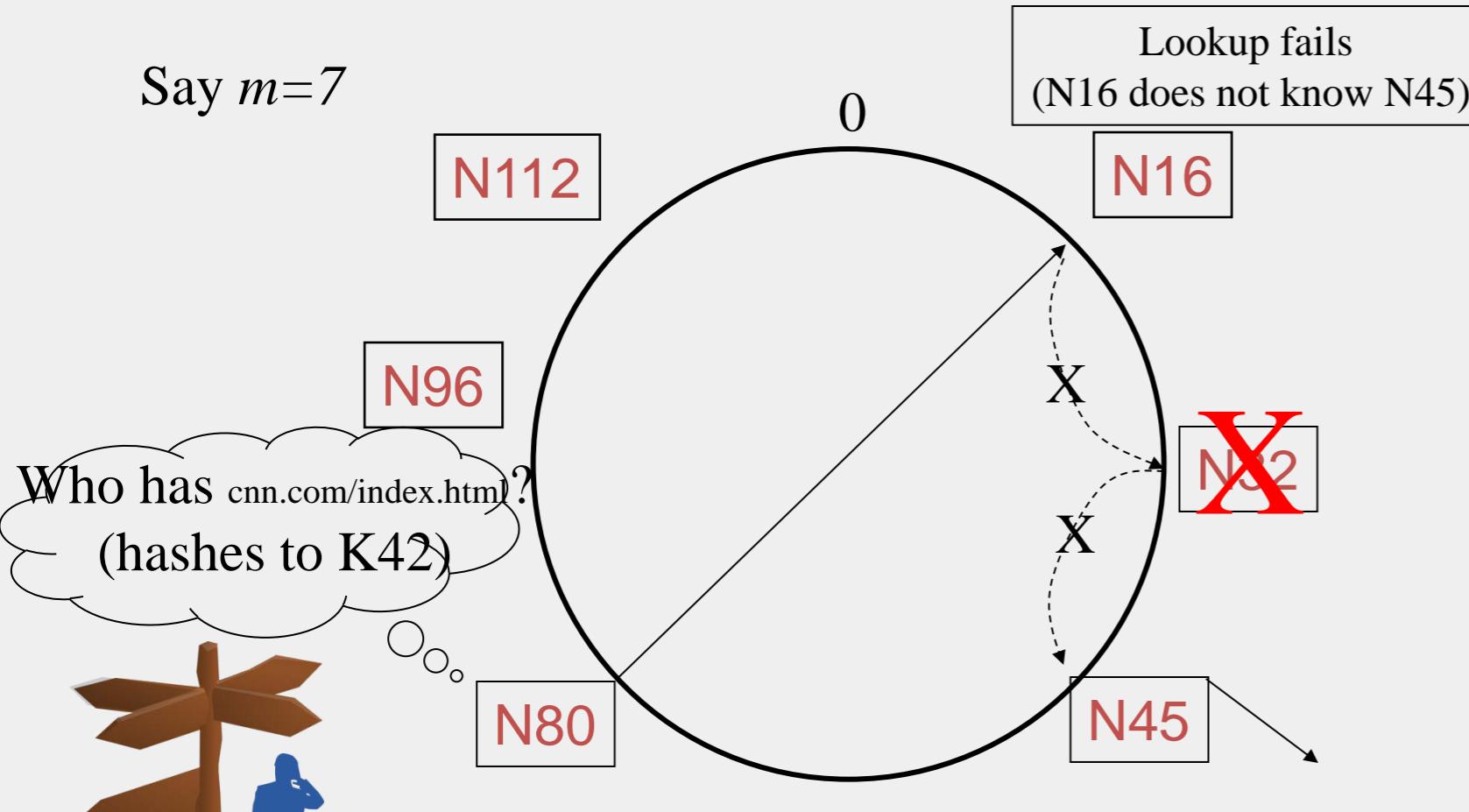
P2P SYSTEMS

Lecture F

FAILURES IN CHORD

SEARCH UNDER PEER FAILURES

Say $m=7$

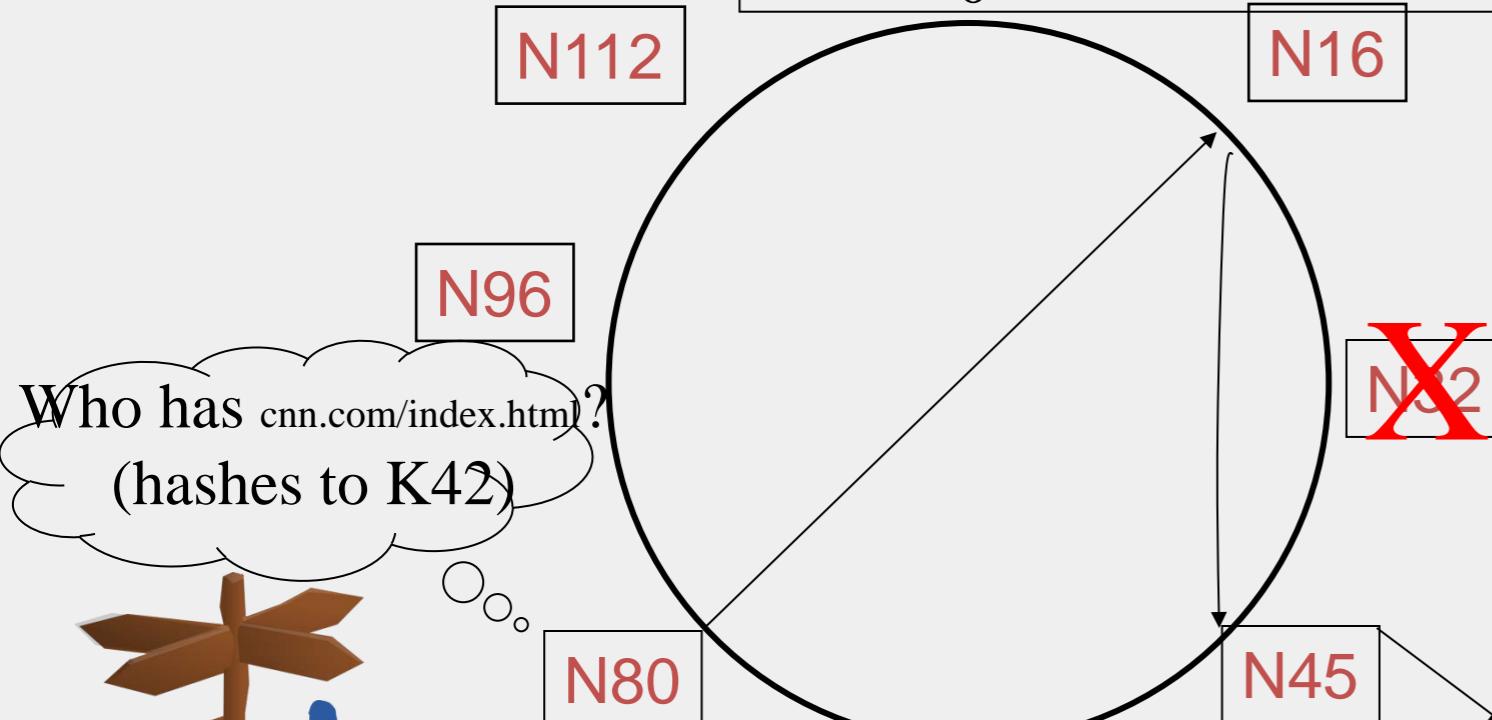


File `cnn.com/index.html` with
key `K42` stored here

SEARCH UNDER PEER FAILURES

Say $m=7$

One solution: maintain r multiple *successor* entries
0 In case of failure, use successor entries



File `cnn.com/index.html` with
key `K42` stored here

SEARCH UNDER PEER FAILURES

- Choosing $r=2\log(N)$ suffices to maintain *lookup correctness* w.h.p. (i.e., ring connected)
 - Say 50% of nodes fail
 - $\Pr(\text{at given node, at least one successor alive}) =$

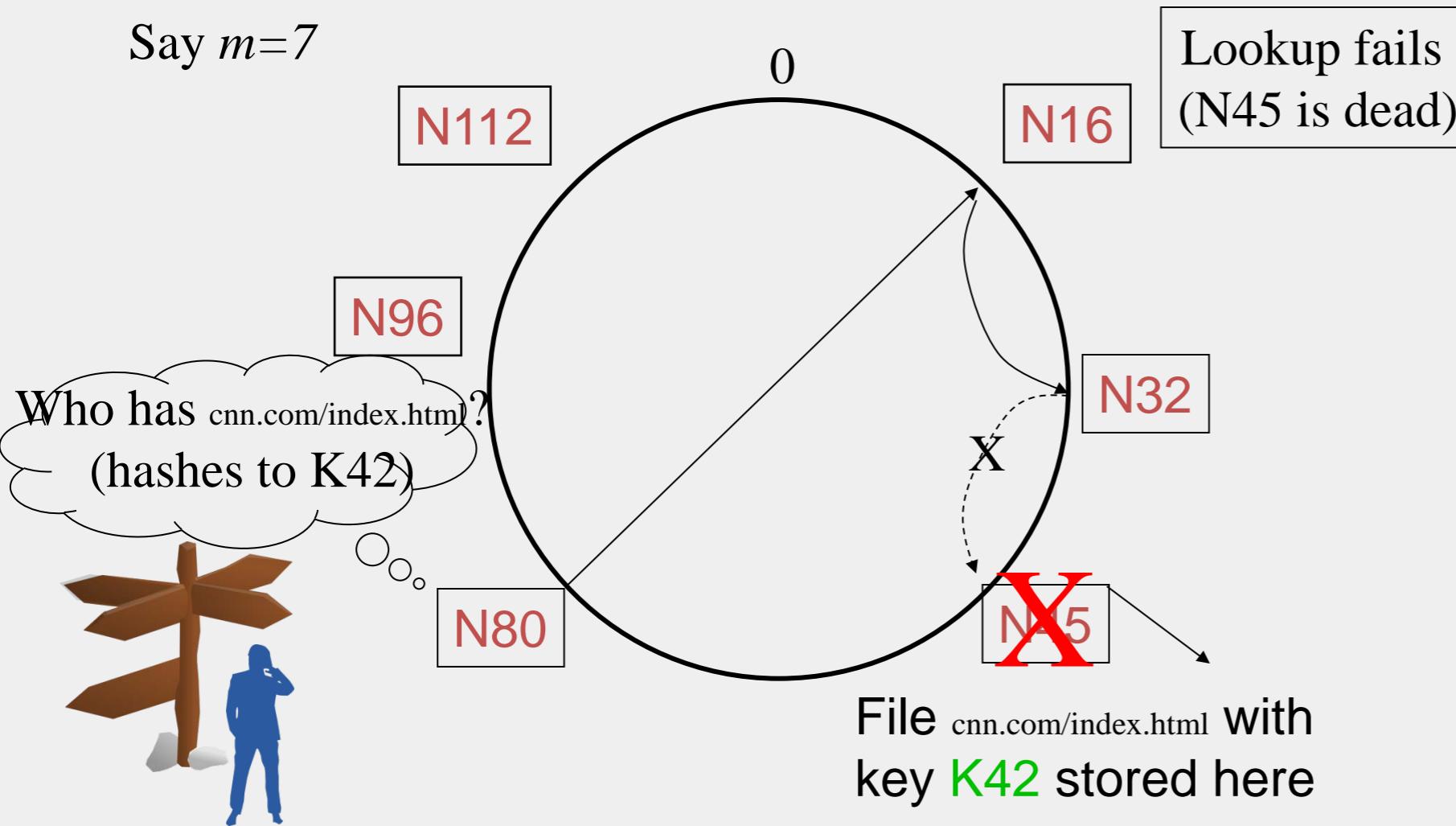
$$1 - \left(\frac{1}{2}\right)^{2\log N} = 1 - \frac{1}{N^2}$$

- $\Pr(\text{above is true at all alive nodes}) =$

$$\left(1 - \frac{1}{N^2}\right)^{N/2} = e^{-\frac{1}{2N}} \approx 1$$

SEARCH UNDER PEER FAILURES (2)

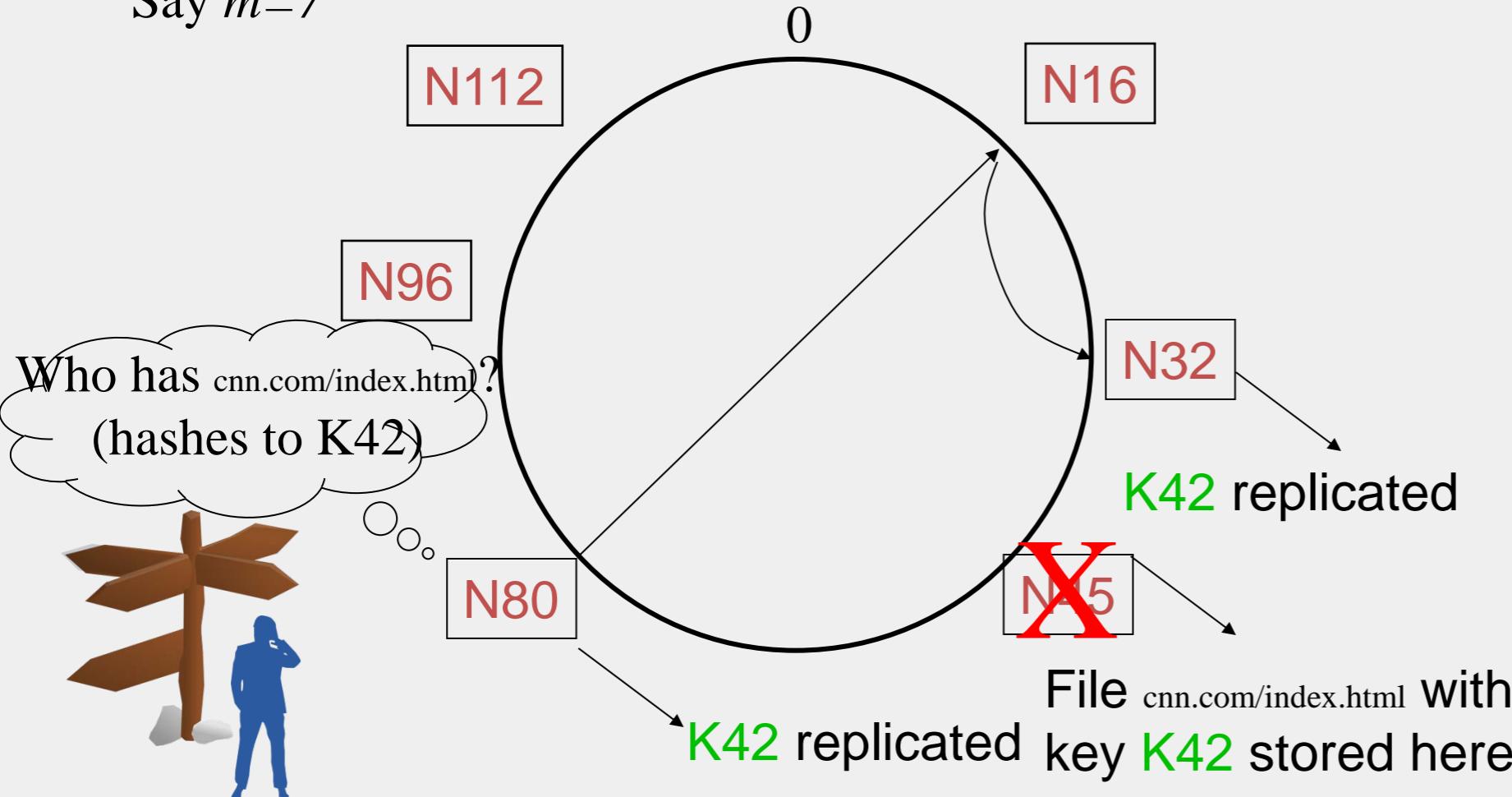
Say $m=7$



SEARCH UNDER PEER FAILURES (2)

Say $m=7$

One solution: replicate file/key at r successors and predecessors



NEED TO DEAL WITH DYNAMIC CHANGES

- ✓ Peers fail
- New peers join
- Peers leave
 - P2P systems have a high rate of *churn* (node join, leave and failure)
 - 25% per hour in Overnet (eDonkey)
 - 100% per hour in Gnutella
 - Lower in managed clusters
 - Common feature in all distributed systems, including wide-area (e.g., PlanetLab), clusters (e.g., Emulab), clouds (e.g., AWS), etc.

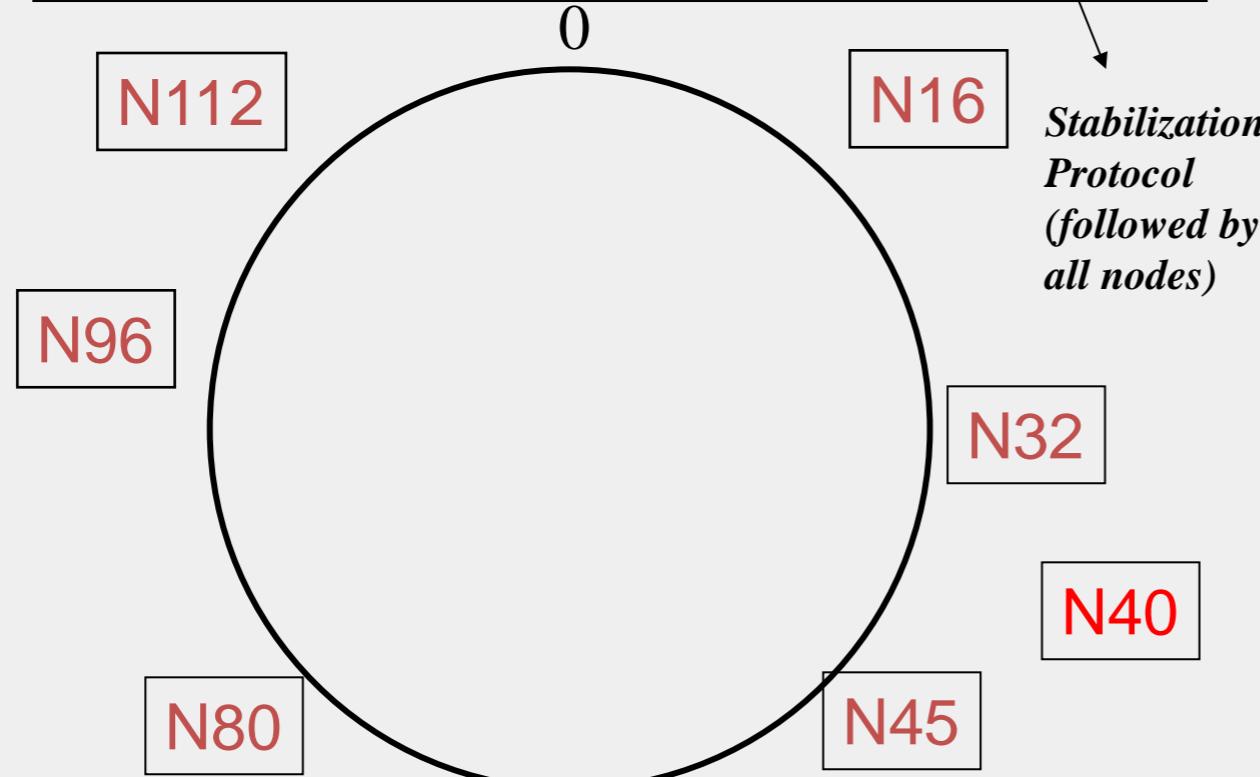
So, all the time, need to:

→ update *successors* and *fingers*, and copy keys

NEW PEERS JOINING

Say $m=7$

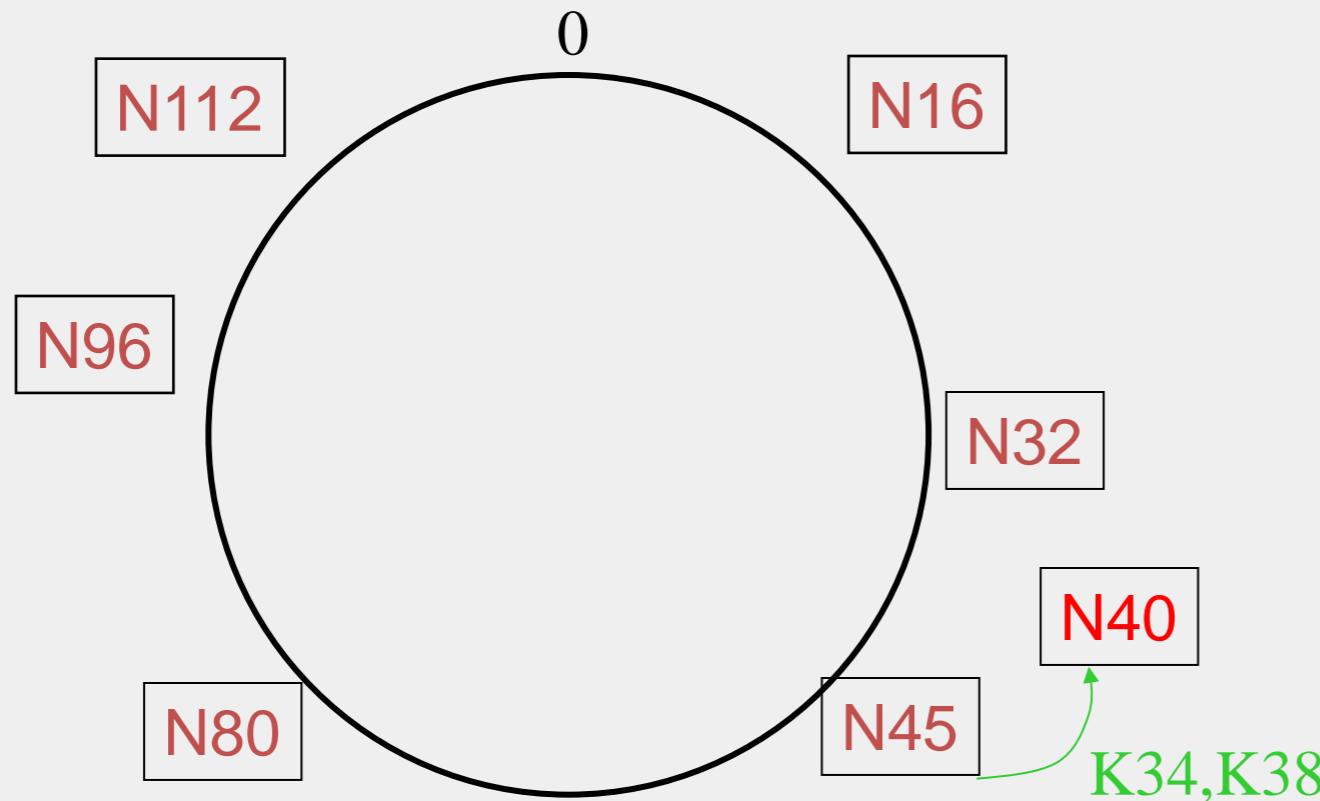
Introducer directs N40 to N45 (and N32)
N32 updates successor to N40
N40 initializes successor to N45, and inits fingers from it
N40 periodically talks to neighbors to update finger table



NEW PEERS JOINING (2)

N40 may need to copy some files/keys from N45
(files with fileid between 32 and 40)

Say $m=7$



NEW PEERS JOINING (3)

- A new peer affects $O(\log(N))$ other finger entries in the system, on average [Why?]
- Number of messages per peer join= $O(\log(N)*\log(N))$
- Similar set of operations for dealing with peers leaving
 - For dealing with failures, also need *failure detectors* (we'll see these later in the course!)

STABILIZATION PROTOCOL

- Concurrent peer joins, leaves, failures might cause loopiness of pointers and failure of lookups
 - Chord peers periodically run a *stabilization* algorithm that checks and updates pointers and keys
 - Ensures *non-loopiness* of fingers, eventual success of lookups and $O(\log(N))$ lookups w.h.p.
 - Each stabilization round at a peer involves a constant number of messages
 - Strong stability takes $O(N^2)$ stabilization rounds
 - For more see [TechReport on Chord webpage]

CHURN

- When nodes are constantly joining, leaving, failing
 - Significant effect to consider: traces from the Overnet system show *hourly* peer turnover rates (**churn**) could be 25–100% of total number of nodes in system
 - Leads to excessive (unnecessary) key copying (remember that keys are replicated)
 - Stabilization algorithm may need to consume more bandwidth to keep up
 - Main issue is that files are replicated, while it might be sufficient to replicate only meta information about files
 - Alternatives
 - Introduce a level of indirection (any p2p system)
 - Replicate metadata more, e.g., Kelips (later in this lecture series)

VIRTUAL NODES

- Hash can get non-uniform → Bad load balancing
 - Treat each node as multiple virtual nodes behaving independently
 - Each joins the system
 - Reduces variance of load imbalance

WRAP-UP NOTES

- Virtual Ring and Consistent Hashing used in Cassandra, Riak, Voldemort, DynamoDB, and other key-value stores
- Current status of Chord project:
 - File systems (CFS, Ivy) built on top of Chord
 - DNS lookup service built on top of Chord
 - Internet Indirection Infrastructure (I3) project at UC Berkeley
 - Spawning research on many interesting issues about p2p systems

<http://www.pdos.lcs.mit.edu/chord/>

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

P2P SYSTEMS

Lecture G

PASTRY

PASTRY

- Designed by Antony Rowstron (Microsoft Research) and Peter Druschel (Rice University)
- Assigns ids to nodes, just like Chord (using a virtual ring)
- **Leaf Set** – Each node knows its successor(s) and predecessor(s)

PASTRY NEIGHBORS

- **Routing tables based prefix matching**
 - Think of a hypercube
- Routing is thus based on prefix matching and is thus $\log(N)$
 - And hops are short (in the underlying network)

PASTRY ROUTING

- Consider a peer with id 01110100101. It maintains a neighbor peer with an id matching each of the following prefixes:
 - 0^*
 - 01^*
 - 011^*
 - ... 0111010010^*
- When it needs to route to a peer, say $011101\underline{1}1001$, it starts by forwarding to a neighbor with the largest matching prefix, i.e., 011101^*

PASTRY LOCALITY

- For each prefix, say 011^* , among all potential neighbors with a matching prefix, the neighbor with the shortest round-trip time is selected
- Since shorter prefixes have many more candidates (spread out throughout the Internet), the neighbors for shorter prefixes are likely to be closer than the neighbors for longer prefixes
- Thus, in the prefix routing, early hops are short and later hops are longer
- Yet overall “stretch,” compared to direct Internet path, stays short

SUMMARY OF CHORD AND PASTRY

- Chord and Pastry protocols
 - More structured than Gnutella
 - Black box lookup algorithms
 - Churn handling can get complex
 - $O(\log(N))$ memory and lookup cost
 - $O(\log(N))$ lookup hops may be high
 - Can we reduce the number of hops?

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

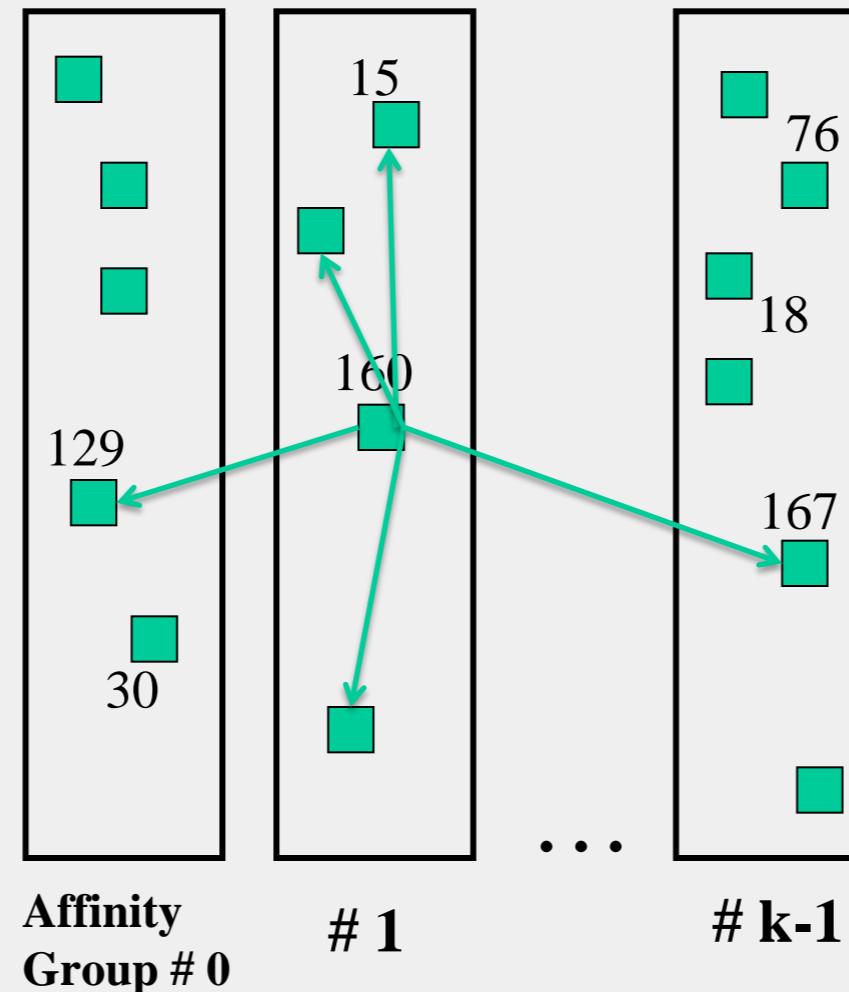
P2P SYSTEMS

Lecture H

KELIPS

KELIPS - A 1 HOP Lookup DHT

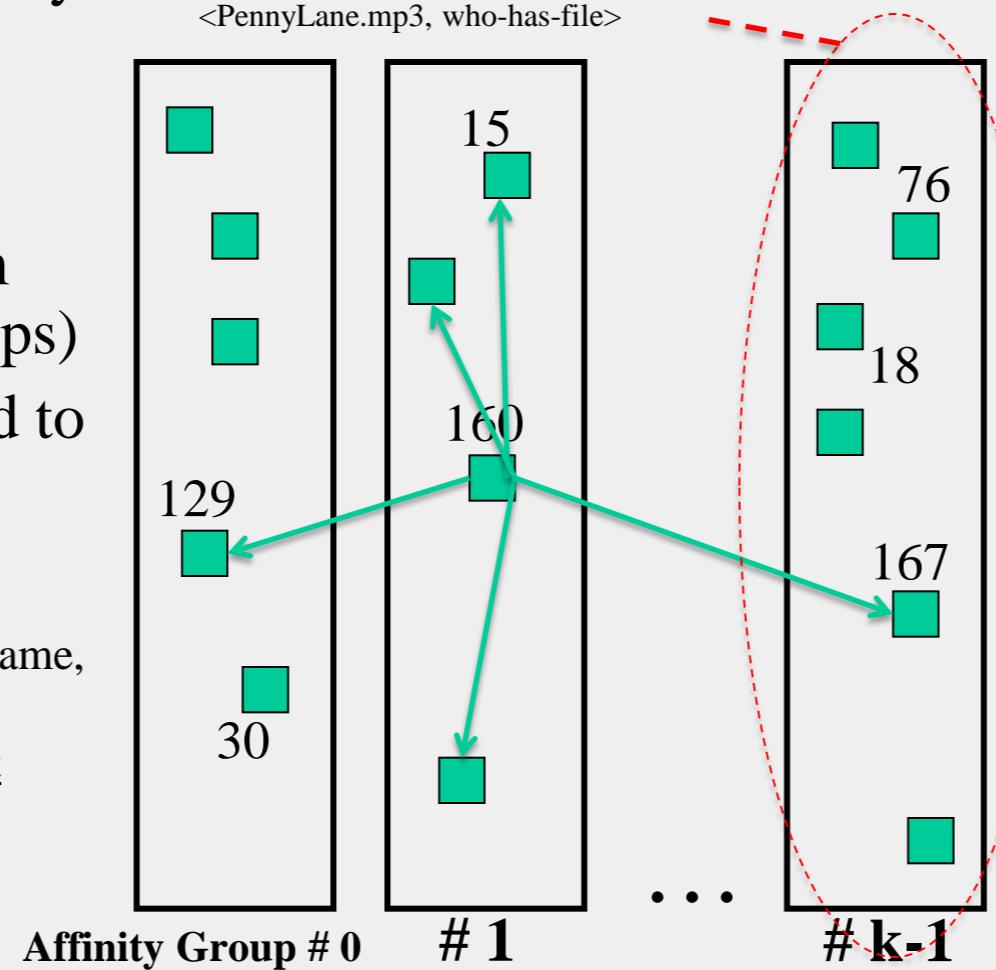
- k “affinity groups”
 - $k \sim \sqrt{N}$
- Each node hashed to a group (hash mod k)
- Node’s neighbors
 - (Almost) all other nodes in its own affinity group
 - One contact node per foreign affinity group



KELIPS FILES AND METADATA

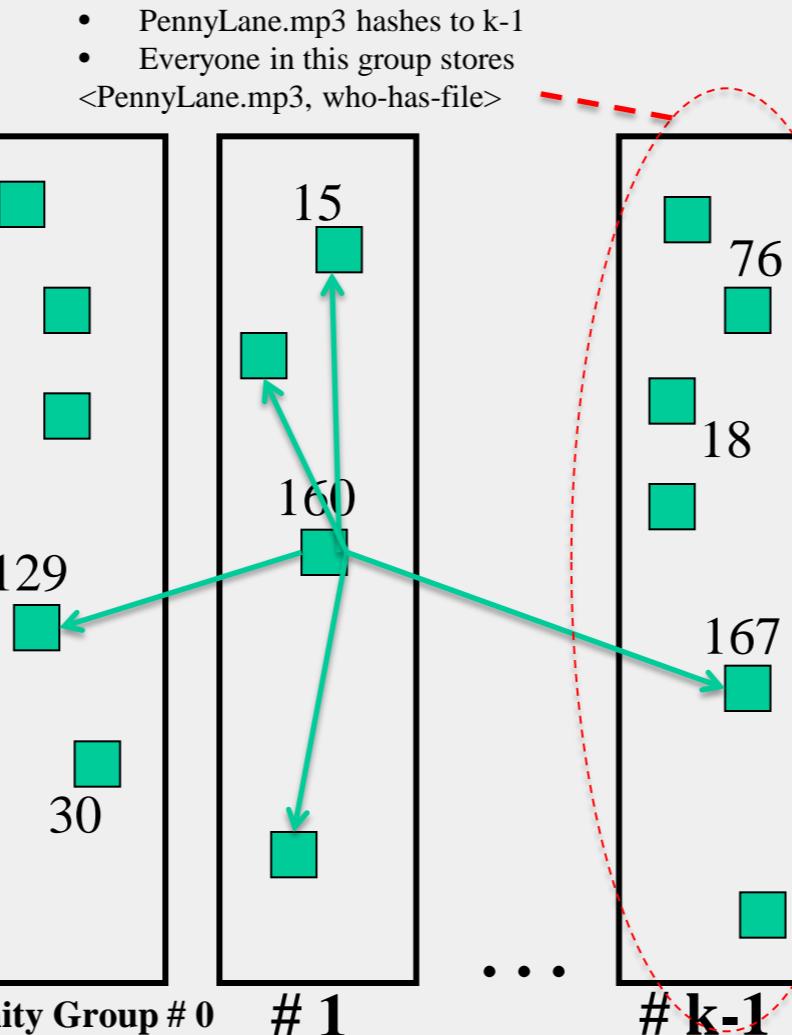
- File can be stored at any (few) node(s)
- Decouple file replication/location (outside Kelips) from file querying (in Kelips)
- Each filename hashed to a group
 - All nodes in the group replicate pointer information, i.e., <filename, file location>
 - Affinity group does not store files

- PennyLane.mp3 hashes to k-1
- Everyone in this group stores <PennyLane.mp3, who-has-file>



KELIPS LOOKUP

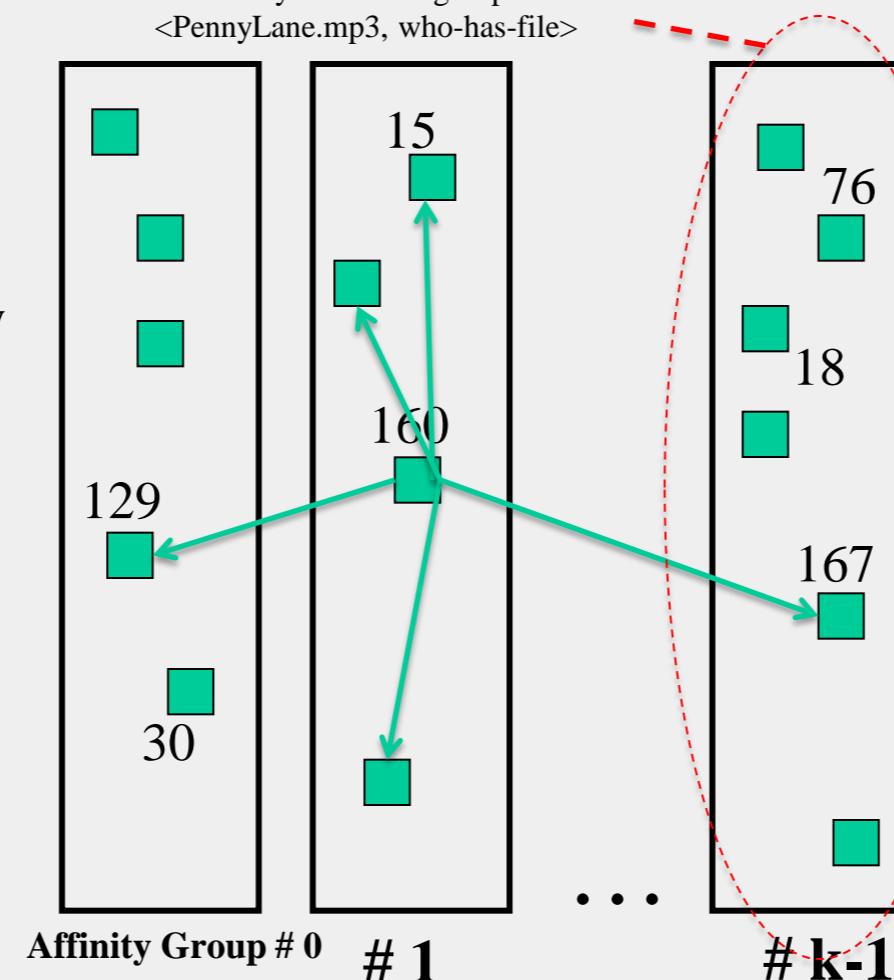
- Lookup
 - Find file affinity group
 - Go to your contact for the file affinity group
 - Failing that try another of your neighbors to find a contact
- Lookup = 1 hop (or a few)
 - Memory cost $O(\sqrt{N})$
 - 1.93 MB for 100K nodes, 10M files
 - Fits in RAM of most workstations/laptops today (COTS machines)



KELIPS SOFT STATE

- Membership lists
 - Gossip-based membership
 - Within each affinity group
 - And also across affinity groups
 - $O(\log(N))$ dissemination time
- File metadata
 - Needs to be periodically refreshed from source node
 - Times out

- PennyLane.mp3 hashes to $k-1$
- Everyone in this group stores $\langle \text{PennyLane.mp3}, \text{who-has-file} \rangle$



CHORD VS. PASTRY VS. KELIPS

- Range of tradeoffs available
 - Memory vs. lookup cost vs. background bandwidth (to keep neighbors fresh)

WHAT WE HAVE STUDIED

- Widely-deployed P2P systems
 1. Napster
 2. Gnutella
 3. Fasttrack (Kazaa, Kazaalite, Grokster)
 4. BitTorrent
- P2P systems with provable properties
 1. Chord
 2. Pastry
 3. Kelips

Distributed Systems

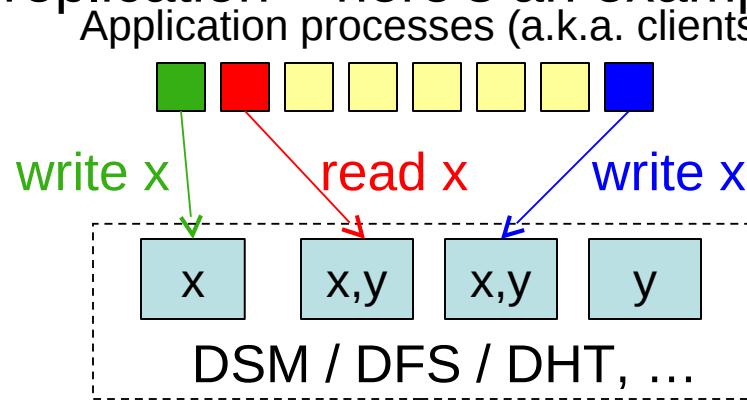
Lec 12: Consistency Models –
Sequential, Causal, and Eventual Consistency

Slide acks: Jinyang Li

(<http://www.news.cs.nyu.edu/~jinyang/fa10/notes/ds-eventual.ppt>)

Consistency (Reminder)

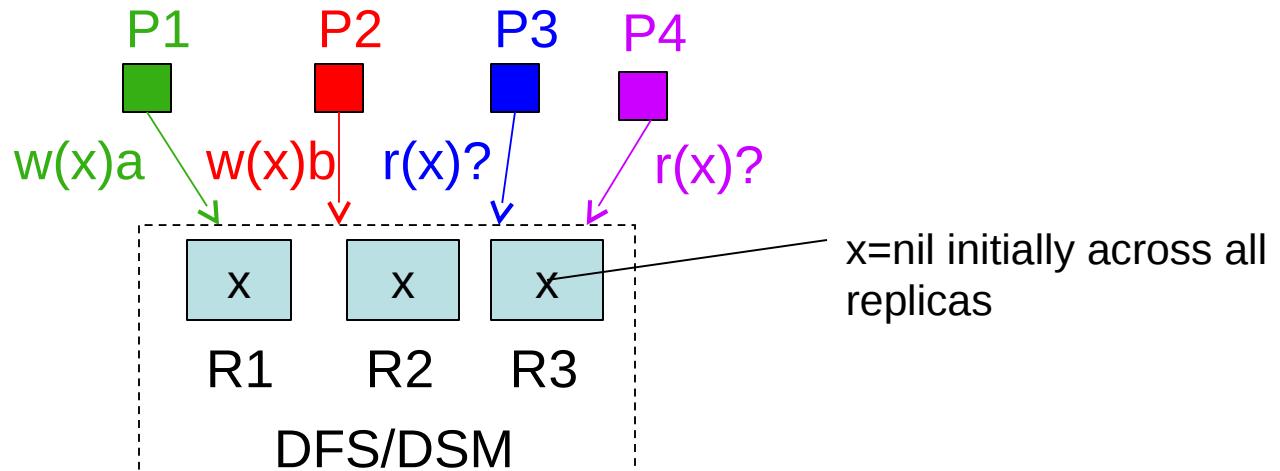
- What is consistency?
 - What processes can expect when RD/WR shared data concurrently
- When do consistency concerns arise?
 - With **replication** and **caching**
- Why are replication and caching needed?
 - For **performance**, **scalability**, **fault tolerance**, **disconnection**
- Let's focus on replication – here's an example:



Consistency (Reminder)

- What is a consistency model?
 - Contract between a distributed data system (e.g., DFS, DSM) and processes constituting its applications
 - E.g.: “If a process reads a certain piece of data, I (the DFS/DSM) pledge to return the value of the last write”
- What are some consistency models?
 - Strict consistency
 - Sequential consistency
 - Causal consistency
 - Eventual consistency
 - Less intuitive, harder to program
 - More feasible, scalable, efficient (traditionally)
- Variations boil down to:
 - The allowable staleness of reads
 - The ordering of writes across all replicas

Example



- Consistency model defines what values reads are admissible by the DFS/DSM

Time at which client process issues op

wall-clock time

P1: $w(x)a$

P2: $w(x)b$

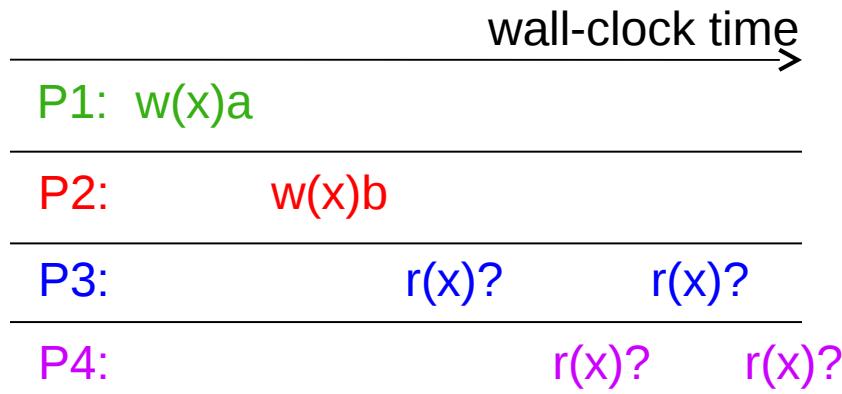
P3: $r(x)?$ $r(x)?$

P4: $r(x)?$ $r(x)?$

May differ from the time at which the op request gets to relevant replica!

Strict Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in order of **wall-clock time** at which they were issued
- Therefore:
 - Reads are never stale
 - All replicas enforce wall-clock ordering for all writes
- If DSM were strictly consistent, **what can these reads return?**



Strict Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in order of **wall-clock time** at which they were issued
- Therefore:
 - Reads are never stale
 - All replicas enforce wall-clock ordering for all writes
- If DSM were strictly consistent, **what can these reads return?**



P1: w(x)a

P2: w(x)b

P3: r(x)b r(x)b

P4: r(x)b r(x)b



P1: w(x)a

P2: w(x)b

P3: r(x)a r(x)b

P4: r(x)b r(x)b

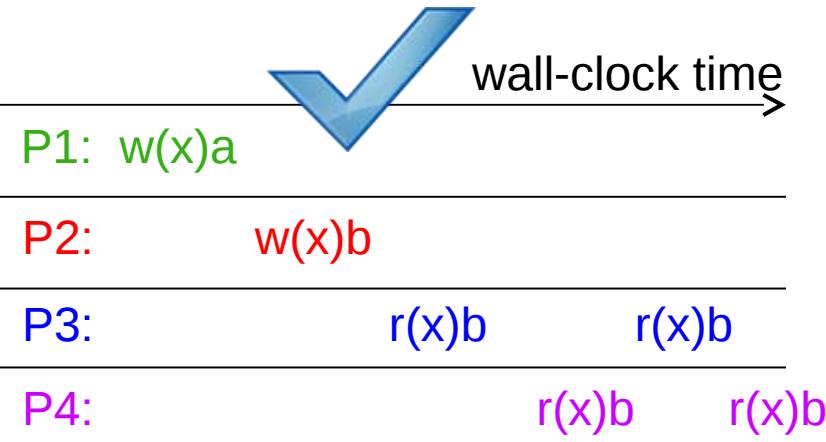
Sequential Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in **some global ordering**, and the ops of each client process appear in the order specified by its program
- Therefore:
 - Reads may be stale in terms of real time, but not in logical time
 - Writes are totally ordered according to logical time across all replicas
- If DSM were seq. consistent, **what can these reads return?**

	wall-clock time →	
P1:	w(x)a	
P2:		w(x)b
P3:		r(x)? r(x)?
P4:		r(x)? r(x)?

Sequential Consistency (Last Time)

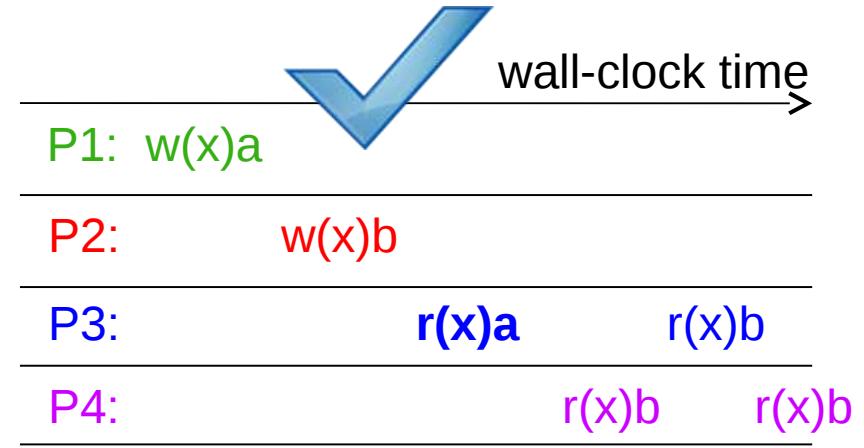
- Any execution is the same as if all read/write ops were executed in **some global ordering**, and the ops of each client process appear in the order specified by its program



What's a global sequential order
that can explain these results?

wall-clock ordering

This was also strictly
consistent



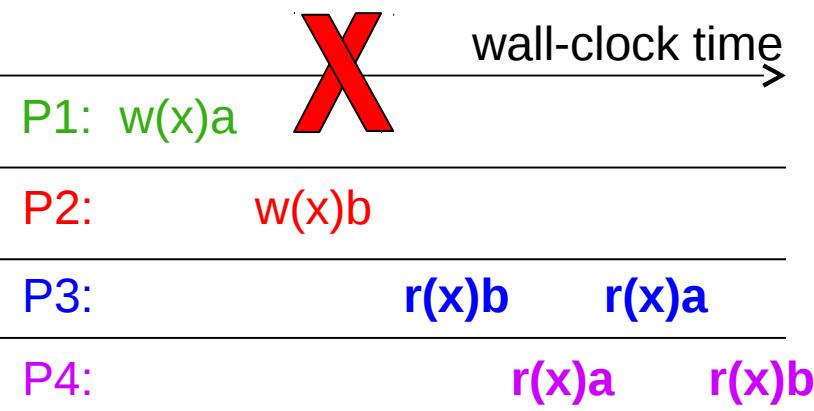
What's a global sequential order
that can explain these results?

w(x)a, r(x)a, w(x)b, r(x)b, ...

This wasn't strictly
consistent

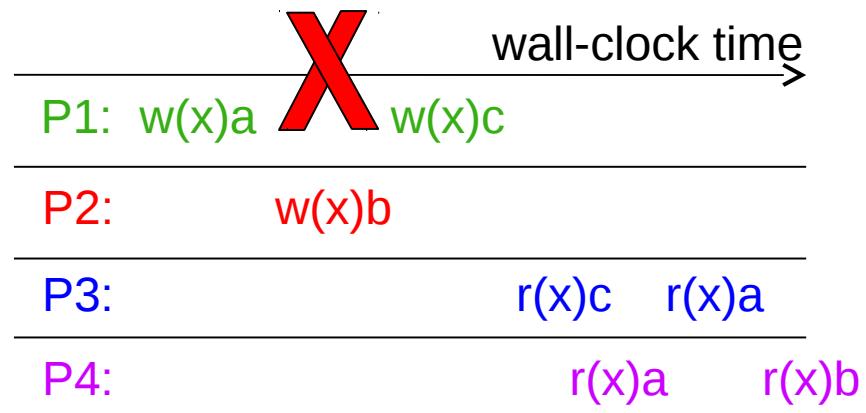
Sequential Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in **some global ordering**, and the ops of each client process appear in the order specified by its program



No global ordering can explain these results...

=> not seq. consistent



No *global sequential* global ordering can explain these results...

E.g.: the following global ordering doesn't preserve P1's ordering
w(x)c, r(x)c, w(x)a, r(x)a, w(x)b, ...

Today

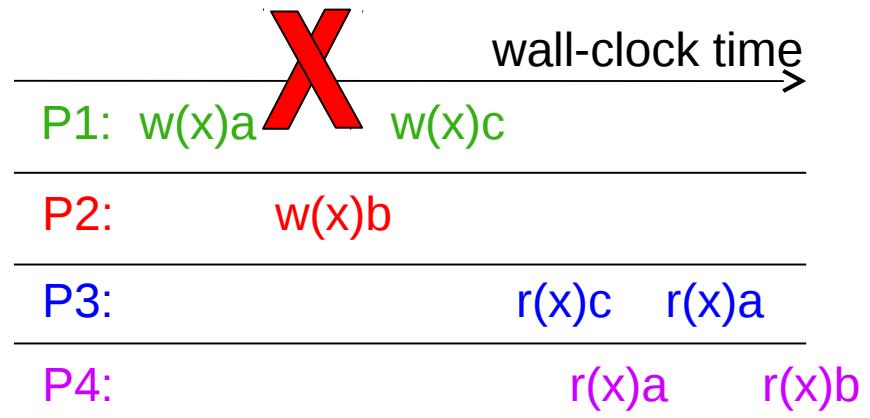
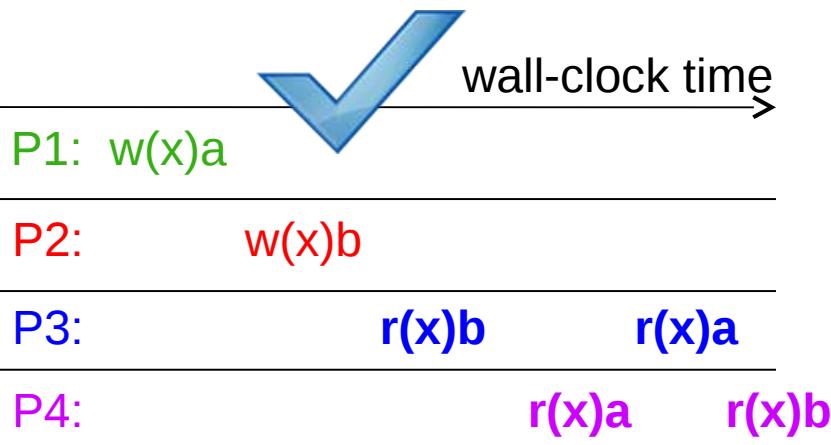
- Causal consistency
- Eventual consistency
- Implementing eventual consistency

Causal Consistency

- Remember causality notion from Lamport (logical) clocks?
 - That's what causal consistency enforces
- Causal consistency: Any execution is the same as if all **causally-related** read/write ops were executed in an **order that reflects their causality**
 - All **concurrent** ops may be seen in different orders
- Therefore:
 - Reads are fresh only w.r.t. the writes that they are causally dependent on
 - Only causally-related writes are ordered by all replicas in the same way, but concurrent writes may be committed in different orders by different replicas, and hence read in different orders by different applications

Causal Consistency: (Counter)Examples

- Any execution is the same as if all causally-related read/write ops were executed in an order that reflects their causality
 - All concurrent ops may be seen in different orders



Only per-process ordering restrictions:

$$w(x)b < r(x)b; r(x)b < r(x)a; \dots$$

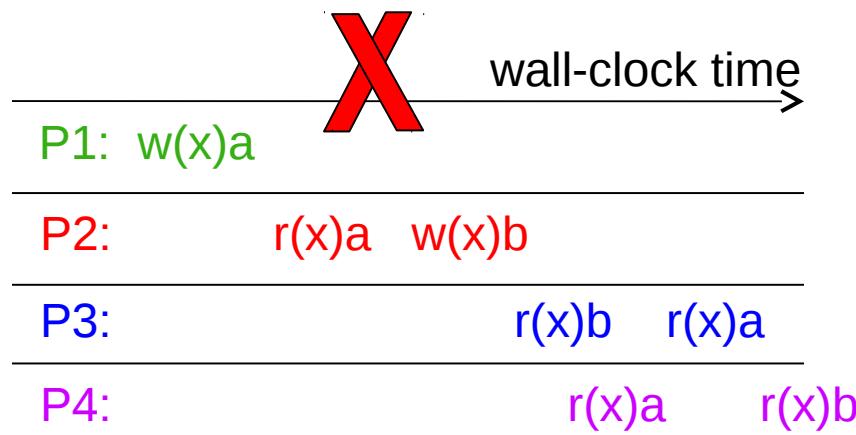
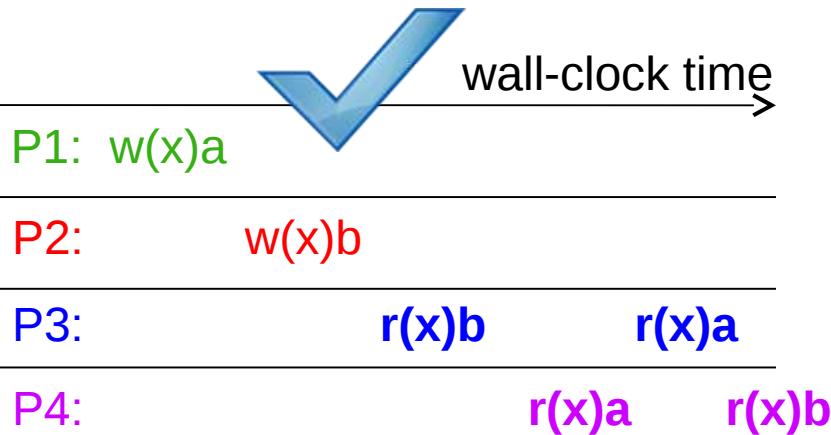
$w(x)a \parallel w(x)b$, hence they can be seen in \neq orders by \neq processes

This wasn't sequentially consistent

Having read **c** ($r(x)c$), P3 must continue to read **c** or some newer value (perhaps **b**), but can't go back to **a**, b/c $w(x)c$ was conditional upon $w(x)a$ having finished

Causal Consistency: (Counter)Examples

- Any execution is the same as if all causally-related read/write ops were executed in an order that reflects their causality
 - All concurrent ops may be seen in different orders



$w(x)b$ is causally-related on $r(x)a$, which is causally-related on $w(x)a$. Therefore, system must enforce $w(x)a < w(x)b$ ordering. But P3 violates that ordering, b/c it reads a after reading b .

Why Causal Consistency?

- Causal consistency is **strictly weaker** than sequential consistency and can give **weird results**, as you've seen
 - If system is sequentially consistent => it is also causally consistent
- BUT: it also offers more possibilities for **concurrency**
 - Concurrent operations (which are not causally-dependent) can be executed in different orders by different people
 - In contrast, with sequential consistency, you need to enforce a global ordering of all operations
 - Hence, one can get **better performance** than sequential
- From what I know, not very popular in industry
 - So, we're not gonna focus on it any more

Eventual Consistency (Overview)

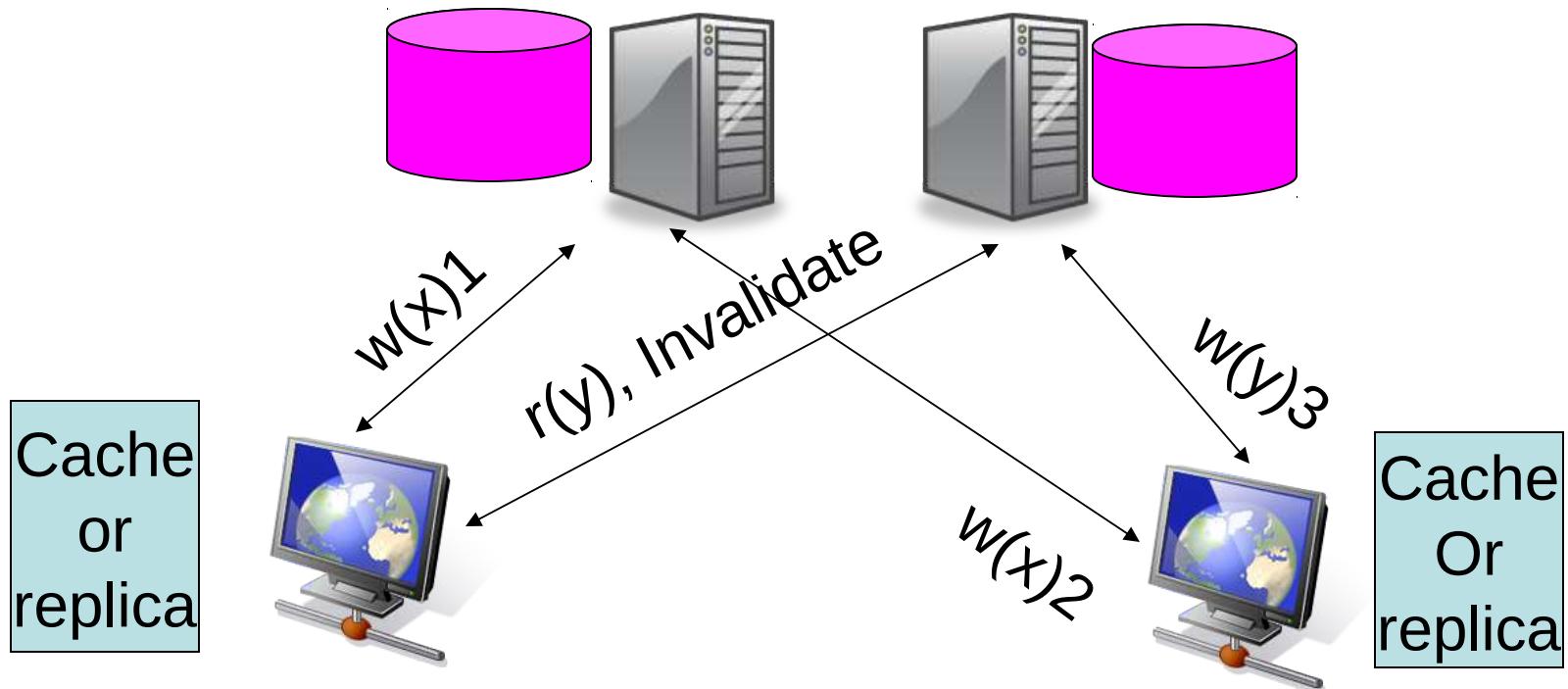
- Allow stale reads, but ensure that reads will eventually reflect previously written values
 - Even after very long times
- Doesn't order concurrent writes as they are executed, which might create conflicts later: which write was first?
- Used in Amazon's Dynamo, a key/value store
 - Plus a lot of academic systems
 - Plus file synchronization ← familiar example, we'll use this

Why Eventual Consistency?

- More concurrency opportunities than strict, sequential, or causal consistency
- Sequential consistency requires **highly available connections**
 - Lots of chatter between clients/servers
- Sequential consistency many be unsuitable for certain scenarios:
 - Disconnected clients (e.g. your laptop goes offline, but you still want to edit your shared document)
 - Network partitioning across datacenters
 - Apps might prefer potential inconsistency to loss of availability

Case-in-Point: Realizing Sequential Consistency

- All reads/writes to address X must be ordered by one memory/storage module responsible for X (see Ivy, Lec10)
- If you write data that others have, you must let them know
- Thus, everyone must be online all the time



Why (Not) Eventual Consistency?

- ✓ Support **disconnected** operations or network partitions
 - Better to read a stale value than nothing
 - Better to save writes somewhere than nothing
- ✓ Support for increased parallelism
 - But that's not what people have typically used this for
- ✗ Potentially **anomalous** application behavior
 - Stale reads and **conflicting writes**...

Sequential vs. Eventual Consistency

- Sequential: pessimistic concurrency handling
 - Decide on update order as they are executed
- Eventual: optimistic concurrency handling
 - Let updates happen, worry about deciding their order later
 - May raise conflicts
 - Think about when you code offline for a while – you may need to resolve conflicts with other teammembers when you commit
 - Resolving conflicts is not that difficult with code, but it's really hard in general (e.g., think about resolving conflicts when you've updated an image)

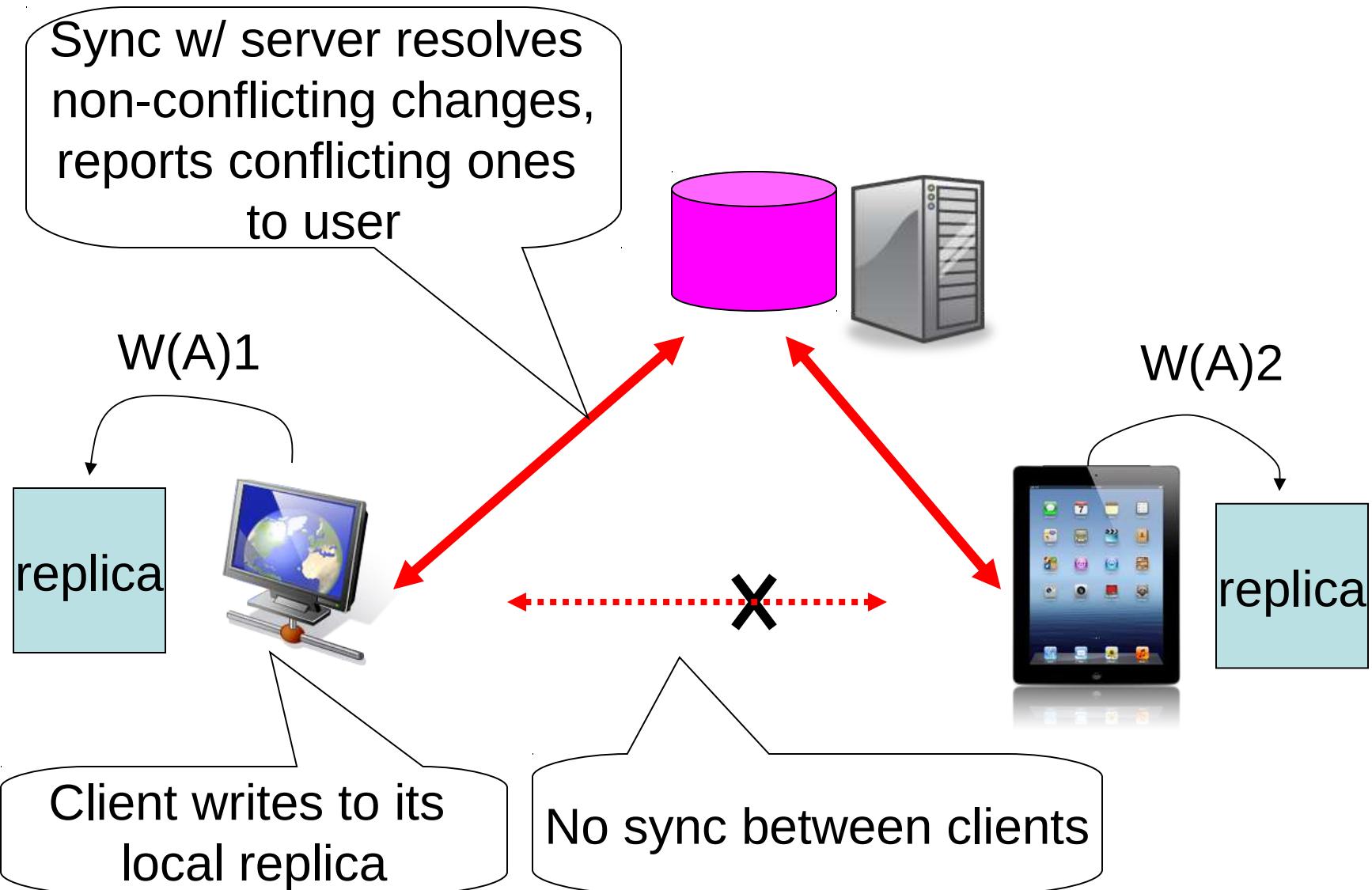
Example Usage: File Synchronizer

- One user, many gadgets, common files (e.g., contacts)

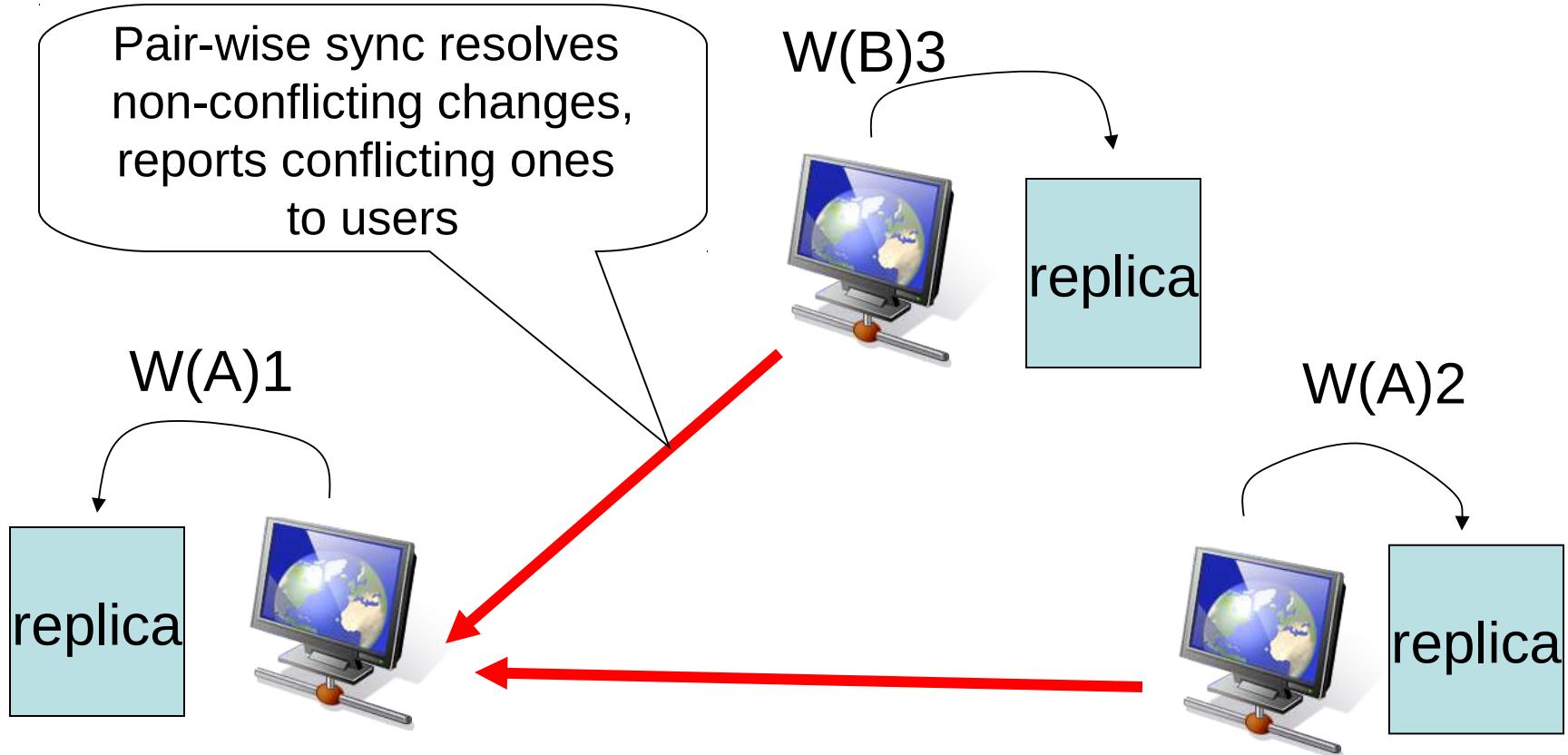


- Goal of file synchronization
 - 1. All replica contents eventually become identical
 - 2. No lost updates
 - Do not replace new version with old ones

Operating w/o Total Connectivity



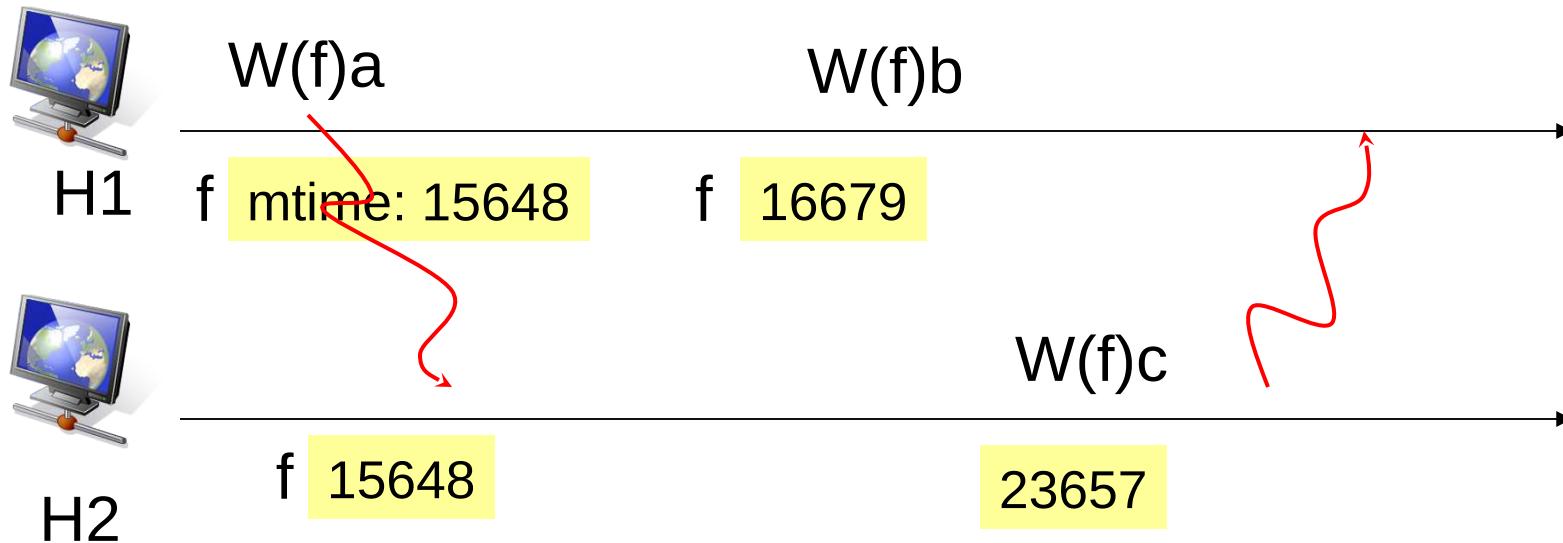
Pair-wise Synchronization



Prevent lost updates

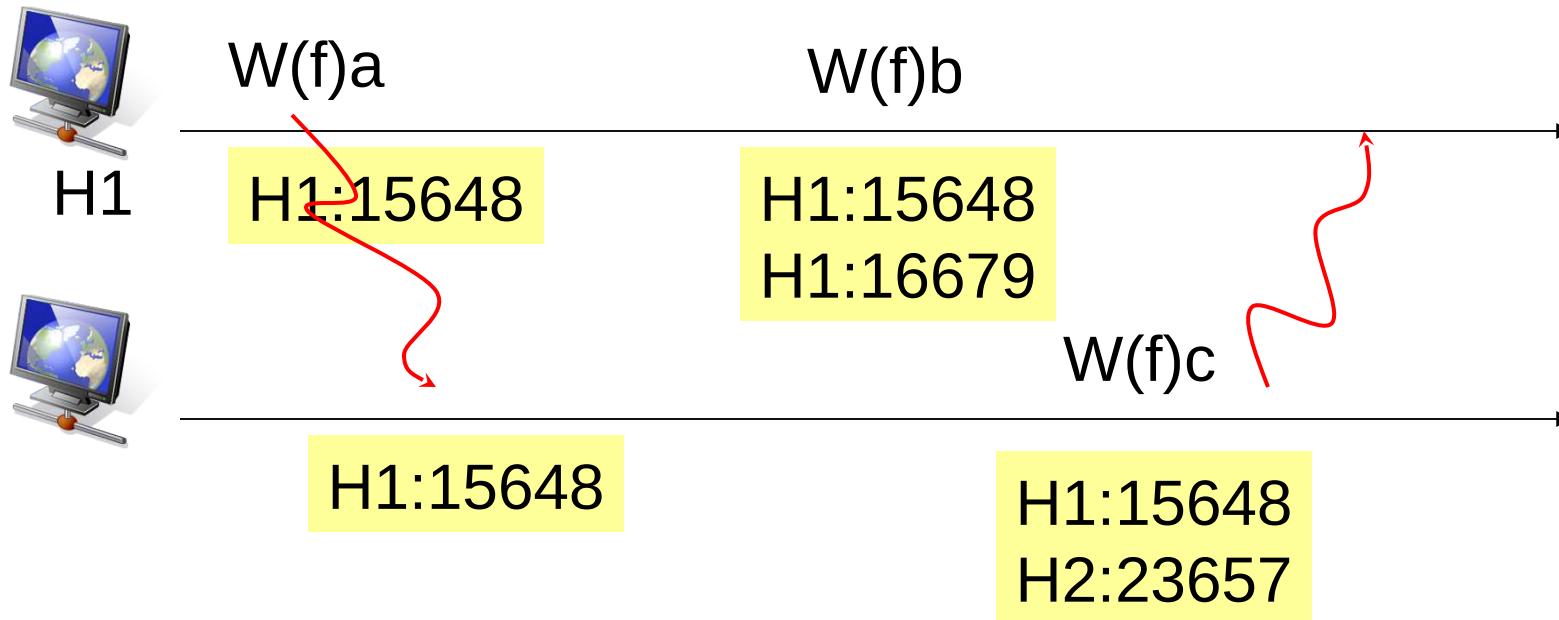
- Detect if updates were sequential
 - If so, replace old version with new one
 - If not, detect conflict

How to Prevent Lost Updates?



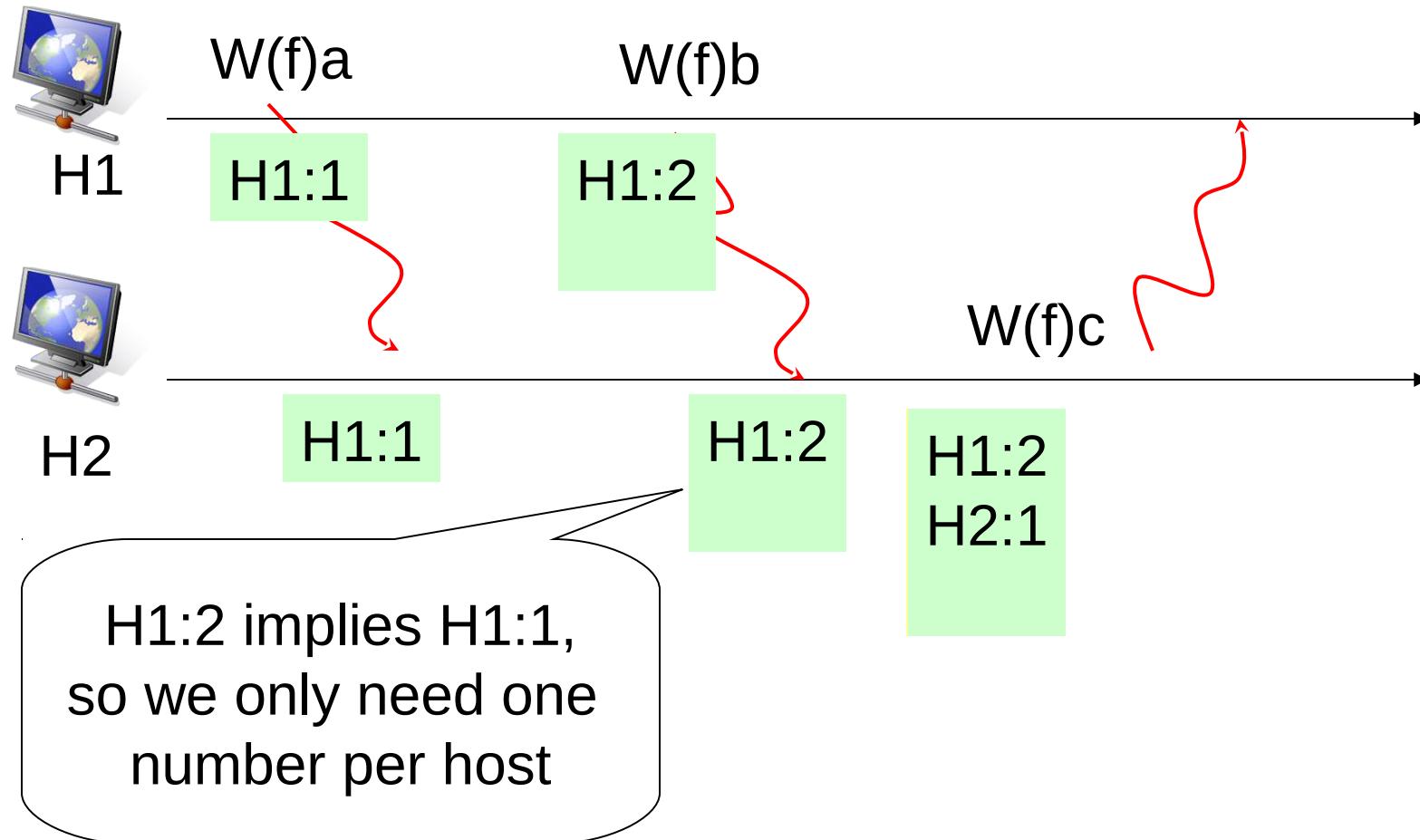
- Strawman: use mtime to decide which version should replace the other
- Problems?
 1. If **clocks are unsynchronized**: new data might have older timestamp than old data
 2. **Does not detect conflicts** => may lose some contacts...

Strawman Fix



- Carry the entire **modification history** (a log)
- If history X is a prefix of Y, Y is newer
- If it's not, then detect and potentially solve conflicts

Compress Version History



How to Deal w/ Conflicts?

- Easy: mailboxes w/ two different set of messages
- Medium: changes to different lines of a C source file
- Hard: changes to same line of a C source file
- After conflict resolution, add a new item to the history?

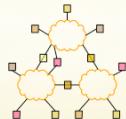
So, What's Used Where?

- Strict consistency
 - Google's just presented Spanner at last week's OSDI '12 conference, which looks similar to strict consistency (they call it "external consistency")
 - EXCITING: thus far thought of as impossible
- Sequential consistency
 - A number of both academic and industrial systems provide (at least) sequential consistency (some a bit stronger – linearizability)
 - Examples: Yale's IVY DSM, Microsoft's Niobe DFS, Cornell's chain replication, ...
- Causal consistency – dunno
- Eventual consistency
 - Very popular for a while both in industry and in academia
 - Examples: file synchronizers, Amazon's Dynamo, Bayou

Many Other Consistency Models Exist

- Other standard consistency models
 - Linearizability
 - Serializability
 - Monotonic reads
 - Monotonic writes
 - ... read Tanenbaum 7.3 if interested (these are not required for exam)
- In-house consistency models:
 - AFS's close-to-open
 - GFS's atomic at-most-once appends

15-446 Distributed Systems Spring 2009



L-10 Consistency

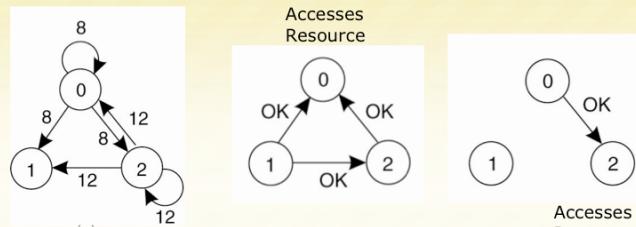
1

Important Lessons

- Lamport & vector clocks both give a logical timestamps
 - Total ordering vs. causal ordering
- Other issues in coordinating node activities
 - Exclusive access to resources/data
 - Choosing a single leader

2

A Distributed Algorithm (2)



- Two processes want to access a shared resource at the same moment.
- Process 0 has the lowest timestamp, so it wins
- When process 0 is done, it sends an OK also, so 2 can now go ahead.

3

Today's Lecture - Replication

- Motivation
 - Performance Enhancement
 - Enhanced availability
 - Fault tolerance
 - Scalability
 - [tradeoff between benefits of replication and work required to keep replicas consistent](#)
- Requirements
 - Consistency
 - Depends upon application
 - In many applications, we want that different clients making (read/write) requests to different replicas of the same logical data item should not obtain different results
 - Replica transparency
 - desirable for most applications

4

Outline

- Consistency Models
 - Data-centric
 - Client-centric
- Approaches for implementing Sequential Consistency
 - primary-backup approaches
 - active replication using multicast communication
 - quorum-based approaches

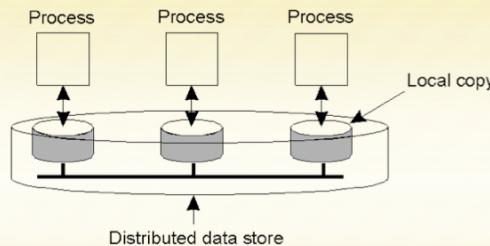
5

Consistency Models

- Consistency Model is a contract between processes and a data store
 - if processes follow certain rules, then store will work "correctly"
- Needed for understanding how concurrent reads and writes behave with respect to shared data
- Relevant for shared memory multiprocessors
 - cache coherence algorithms
- Shared databases, files
 - independent operations
 - our main focus in the rest of the lecture
 - transactions

6

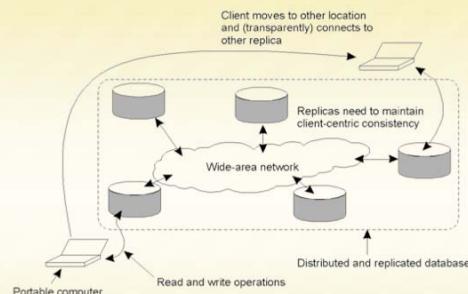
Data-Centric Consistency Models



- The general organization of a logical data store, physically distributed and replicated across multiple processes. Each process interacts with its local copy, which must be kept 'consistent' with the other copies.

7

Client-centric Consistency Models



- A mobile user may access different replicas of a distributed database at different times. This type of behavior implies the need for a view of consistency that provides guarantees for single client regarding accesses to the data store.

8

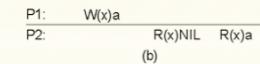
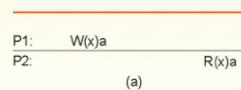
Data-centric Consistency Models

- Strict consistency
 - Sequential consistency
 - Linearizability
 - Causal consistency
 - FIFO consistency
 - Weak consistency
 - Release consistency
 - Entry consistency
- use explicit synchronization operations
- Notation:
- $W_i(x)a$ → process i writes value a to location x
 - $R_i(x)a$ → process i reads value a from location x

9

Strict Consistency

Any read on a data item x returns a value corresponding to the result of the *most recent write* on x. "All writes are instantaneously visible to all processes"



A strictly consistent store A store that is not strictly consistent.

Behavior of two processes, operating on the same data item.

The problem with strict consistency is that it relies on *absolute global time* and is impossible to implement in a distributed system.

10

Sequential Consistency - 1

Sequential consistency: the result of any execution is the same as if the read and write operations by all 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].

Note: Any valid interleaving is legal but all processes must see the same interleaving.

P1: W(x)a
P2: W(x)b
P3: R(x)b R(x)a
P4: R(x)b R(x)a

(a)

- A sequentially consistent data store.
- A data store that is not sequentially consistent.

P1: W(x)a
P2: W(x)b
P3: R(x)b R(x)a
P4: R(x)a R(x)b

(b)
P3 and P4 disagree on the order of the writes

11

Sequential Consistency - 2

Process P1	Process P2	Process P3
x = 1; print (y, z);	y = 1; print (x, z);	z = 1; print (x, y);
x = 1; print (y, z);	y = 1; print (x, z);	z = 1; print (x, z);
y = 1; print (x, z);	print (y, z);	print (x, y);
print (x, z);	print (y, z);	print (x, z);
z = 1;	z = 1;	x = 1;
print (x, y);	print (x, y);	print (y, z);
		print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111
(a)	(b)	(c)

(a)-(d) are all legal interleavings.

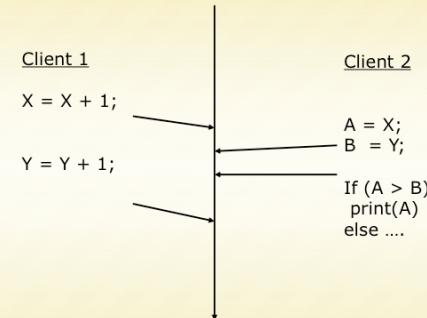
12

Linearizability / Atomic Consistency

- Definition of sequential consistency says nothing about time
 - there is no reference to the “most recent” write operation
- Linearizability
 - weaker than strict consistency, stronger than sequential consistency
 - operations are assumed to receive a timestamp with a global available clock that is loosely synchronized
 - “The result of any execution is the same as if the operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program. In addition, if $tstop1(x) < tstop2(y)$, then $OP1(x)$ should precede $OP2(y)$ in this sequence.” [Herlihy & Wing, 1991]

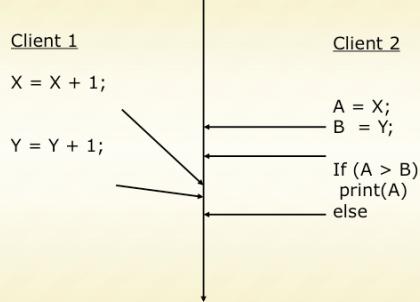
13

Linearizable



14

Not linearizable but sequentially consistent



15

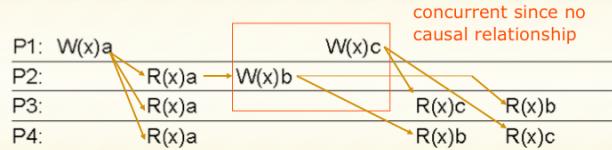
Sequential Consistency vs. Linearizability

- Linearizability has proven useful for reasoning about program correctness but has not typically been used otherwise.
- Sequential consistency is implementable and widely used but has poor performance.
- To get around performance problems, weaker models that have better performance have been developed.

16

Causal Consistency - 1

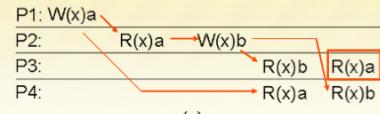
Necessary condition: Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.



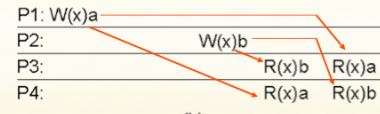
This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store. Can be implemented with vector clocks.

17

Causal Consistency - 2



(a)



(b)

- a) A violation of a causally-consistent store. The two writes are NOT concurrent because of the $R_2(x)a$.
- b) A correct sequence of events in a causally-consistent store ($W_1(x)a$ and $W_2(x)b$ are concurrent).

18

FIFO Consistency

Necessary Condition: 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.

P1:	W(x)a			
P2:	R(x)a	W(x)b	W(x)c	
P3:		R(x)b	R(x)a	R(x)c
P4:		R(x)a	R(x)b	R(x)c

A valid sequence of events of FIFO consistency. Only requirement in this example is that P2's writes are seen in the correct order. FIFO consistency is easy to implement.

19

Weak Consistency - 1

- Uses a synchronization variable with one operation synchronize(S), which causes all writes by process P to be propagated and all external writes propagated to P.
- Consistency is on groups of operations
- Properties:
 1. Accesses to synchronization variables associated with a data store are sequentially consistent (i.e. all processes see the synchronization calls in the same order).
 2. No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.
 3. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

20

Weak Consistency - 2

P2 and P3 have not synchronized, so no guarantee about what order they see.

P1: W(x)a	W(x)b	S	
P2:	R(x)a	R(x)b	S
P3:	R(x)b	R(x)a	S

(a)

P1: W(x)a	W(x)b	S	
P2:	S	R(x)a	

(b)

- a) A valid sequence of events for weak consistency.
- b) An invalid sequence for weak consistency.

21

Release Consistency

- Uses two different types of synchronization operations (*acquire* and *release*) to define a critical region around access to shared data.
- Rules:
 - Before a read or write operation on shared data is performed, all previous *acquires* done by the process must have completed successfully.
 - Before a *release* is allowed to be performed, all previous reads and writes by the process must have completed.
 - Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).

P1: Acq(L) W(x)a W(x)b Rel(L)

P2: Acq(L) R(x)b Rel(L)

P3: R(x)a

No guarantee since operations not used.

22

Entry Consistency

Associate locks with individual variables or small groups.
Conditions:

- An *acquire* access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
- Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
- After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

P1: Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)
P2:		Acq(Lx)	R(x)a	R(y)NIL	
P3:		Acq(Ly)	R(y)b		

No guarantees since y is not acquired.

23

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

- a) Consistency models not using synchronization operations.
- b) Models with synchronization operations.

24

Outline

- Consistency Models
 - Data-centric
 - Client-centric
- Approaches for implementing Sequential Consistency
 - primary-backup approaches
 - active replication using multicast communication
 - quorum-based approaches

25

Consistency Protocols

- Remember that a consistency model is a contract between the process and the data store. If the processes obey certain rules, the store promises to work correctly.
- A consistency protocol is an implementation that meets a consistency model.

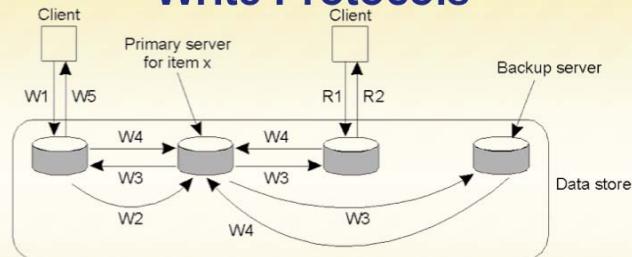
26

Mechanisms for Sequential Consistency

- Primary-based replication protocols
 - Each data item has associated primary responsible for coordination
 - Remote-write protocols
 - Local-write protocols
- Replicated-write protocols
 - Active replication using multicast communication
 - Quorum-based protocols

27

Primary-based: Remote-Write Protocols



W1. Write request
 W2. Forward request to primary
 W3. Tell backups to update
 W4. Acknowledge update
 W5. Acknowledge write completed

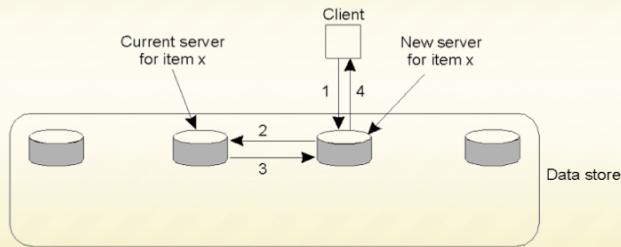
R1. Read request
 R2. Response to read

- The principle of primary-backup protocol.

28

Primary-based: Local-Write Protocols (1)

- Primary-based local-write protocol in which the single copy of the shared data is migrated between processes. One problem with approach is keeping track of current location of data.

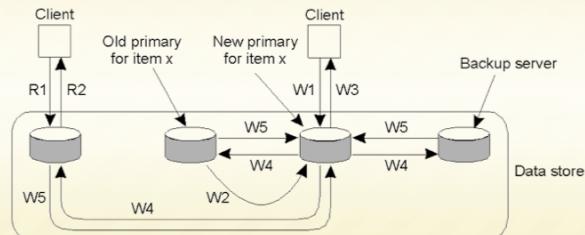


1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

29

Primary-based: Local-Write Protocols (2)

- Primary-backup protocol where replicas are kept but in which the role of primary migrates to the process wanting to perform an update. In this version, clients can read from non-primary copies.



- W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

- R1. Read request
R2. Response to read

30

Replica-based protocols

- Active replication: Updates are sent to all replicas
- Problem: updates need to be performed at all replicas in same order. Need a way to do totally-ordered multicast
- Problem: invocation replication

31

Implementing Ordered Multicast

- Incoming messages are held back in a queue until delivery guarantees can be met
- Coordination between all machines needed to determine delivery order
- FIFO-ordering
 - easy, use a separate sequence number for each process
- Total ordering
- Causal ordering
 - use vector timestamps

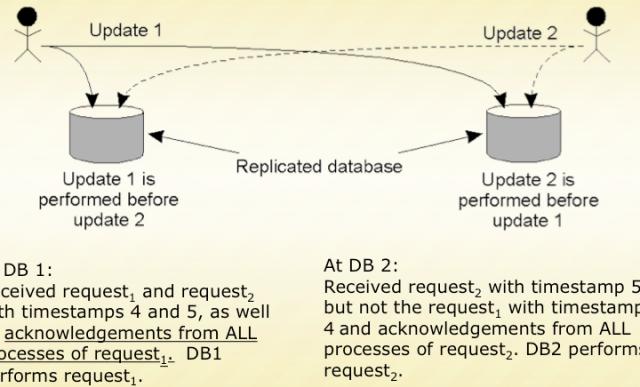
32

Totally Ordered Multicast

- Use Lamport timestamps
- Algorithm
 - Message is timestamped with sender's logical time
 - Message is multicast (including sender itself)
 - When message is received
 - It is put into local queue
 - Ordered according to timestamp
 - Multicast acknowledgement
 - Message is delivered to applications only when
 - It is at head of queue
 - It has been acknowledged by all involved processes
 - Lamport algorithm (extended) ensures total ordering of events

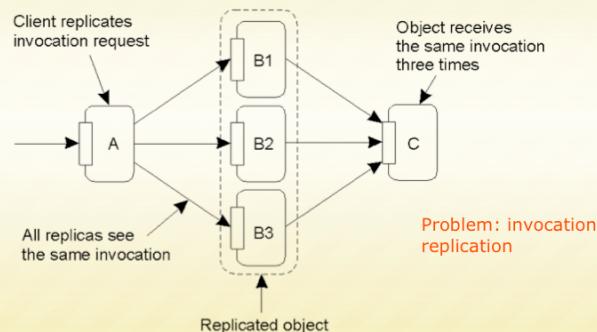
33

Totally-Ordered Multicasting



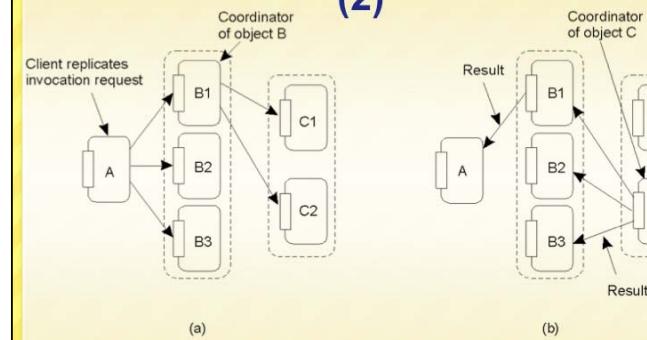
34

Replica-based: Active Replication (1)



35

Replica-based: Active Replication (2)



Assignment of a coordinator for the replicas can ensure that invocations are not replicated.

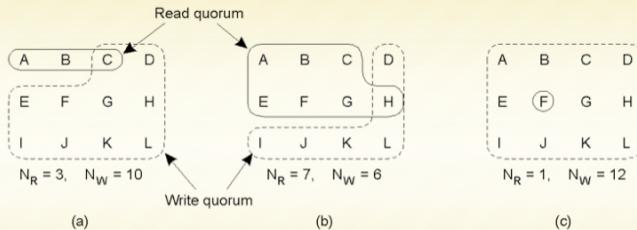
36

Quorum-based protocols - 1

- Assign a number of votes to each replica
- Let N be the total number of votes
- Define R = read quorum, W =write quorum
 - $R+W > N$
 - $W > N/2$
- Only one writer at a time can achieve write quorum
- Every reader sees at least one copy of the most recent read (takes one with most recent version number)

37

Quorum-based protocols - 2



Three examples of the voting algorithm:

- a) A correct choice of read and write set
- b) A choice that may lead to write-write conflicts
- c) A correct choice, known as ROWA (read one, write all)

38

Quorum-based protocols - 3

- ROWA: $R=1$, $W=N$
 - Fast reads, slow writes (and easily blocked)
- RAWO: $R=N$, $W=1$
 - Fast writes, slow reads (and easily blocked)
- Majority: $R=W=N/2+1$
 - Both moderately slow, but extremely high availability
- Weighted voting
 - give more votes to "better" replicas

39

Scaling

- None of the protocols for sequential consistency scale
- To read or write, you have to either
 - (a) contact a primary copy
 - (b) use reliable totally ordered multicast
 - (c) contact over half of the replicas
- All this complexity is to ensure sequential consistency
 - Note: even the protocols for causal consistency and FIFO consistency are difficult to scale if they use reliable multicast

40

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

KEY-VALUE STORES NoSQL

Lecture A

WHY KEY-VALUE/NoSQL?

THE KEY-VALUE ABSTRACTION

- (Business) Key → Value
- (twitter.com) Tweet id → information about tweet
- (amazon.com) Item number → information about it
- (kayak.com) Flight number → information about flight, e.g., availability
- (yourbank.com) Account number → information about it



THE KEY-VALUE ABSTRACTION (2)

- It's a dictionary datastructure.
 - Insert, lookup, and delete by key
 - E.g., hash table, binary tree
- But distributed
- Sound familiar? Remember distributed hash tables (DHT) in P2P systems?
- It's not surprising that key-value stores reuse many techniques from DHTs.



ISN'T THAT JUST A DATABASE?

- Yes, sort of
- Relational Database Management Systems (RDBMSs) have been around for ages
- MySQL is the most popular among them
- Data stored in tables
- Schema-based, i.e., structured tables
- Each row (data item) in a table has a primary key that is unique within that table
- Queried using SQL (Structured Query Language)
- Supports joins



RELATIONAL DATABASE EXAMPLE

users table

user_id	name	zipcode	blog_url	blog_id
101	Alice	12345	alice.net	1
422	Charlie	45783	charlie.com	3
555	Bob	99910	bob.blogspot.com	2

↑
Primary keys
↓

↑
Foreign keys

blog table

id	url	last_updated	num_posts
1	alice.net	5/2/14	332
2	bob.blogspot.com	4/2/13	10003
3	charlie.com	6/15/14	7

Example SQL queries

1. `SELECT zipcode
FROM users
WHERE name = "Bob"`
2. `SELECT url
FROM blog
WHERE id = 3`
3. `SELECT users.zipcode, blog.num_posts
FROM users JOIN blog
ON users.blog_url = blog.url`



MISMATCH WITH TODAY'S WORKLOADS

- Data: Large and unstructured
- Lots of random reads and writes
- Sometimes write-heavy
- Foreign keys rarely needed
- Joins infrequent



NEEDS OF TODAY'S WORKLOADS

- Speed
- Avoid Single Point of Failure (SPOF)
- Low TCO (Total cost of operation)
- Fewer system administrators
- Incremental scalability
- Scale out, not up
 - What?



SCALE OUT, NOT SCALE UP

- Scale up = grow your cluster capacity by replacing with more powerful machines
 - Traditional approach
 - Not cost-effective, as you're buying above the sweet spot on the price curve
 - And you need to replace machines often
- Scale out = incrementally grow your cluster capacity by adding more COTS machines (Components Off the Shelf)
 - Cheaper
 - Over a long duration, phase in a few newer (faster) machines as you phase out a few older machines
 - Used by most companies who run datacenters and clouds today



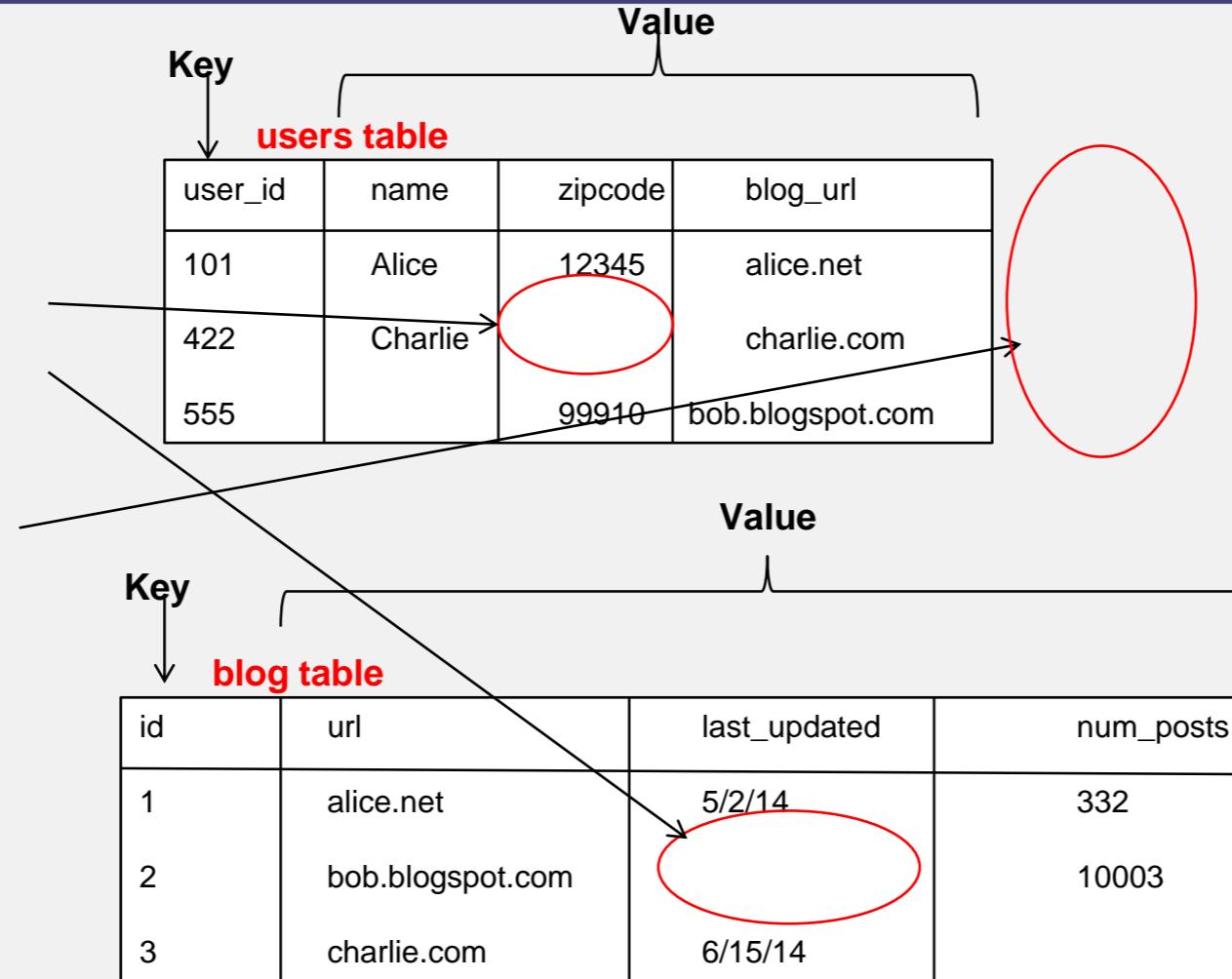
KEY-VALUE/NoSQL DATA MODEL

- NoSQL = “Not Only SQL”
- Necessary API operations: `get(key)` and `put(key, value)`
 - And some extended operations, e.g., “CQL” in Cassandra key-value store
- Tables
 - “Column families” in Cassandra, “Table” in HBase, “Collection” in MongoDB
 - Like RDBMS tables, but ...
 - May be unstructured: May not have schemas
 - Some columns may be missing from some rows
 - Don’t always support joins or have foreign keys
 - Can have index tables, just like RDBMSs



KEY-VALUE/NoSQL DATA MODEL

- Unstructured
- No schema imposed
- Columns missing from some rows
- No foreign keys, joins may not be supported



COLUMN-ORIENTED STORAGE

NoSQL systems often use column-oriented storage

- RDBMSs store an entire row together (on disk or at a server)
- NoSQL systems typically store a column together (or a group of columns).
 - Entries within a column are indexed and easy to locate, given a key (and vice-versa)
- Why useful?
 - Range searches within a column are fast since you don't need to fetch the entire database
 - E.g., get me all the blog_ids from the blog table that were updated within the past month
 - Search in the last_updated column, fetch corresponding blog_id column
 - Don't need to fetch the other columns



NEXT

Design of a real key-value store, Cassandra.



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

KEY-VALUE STORES NoSQL

Lecture B

CASSANDRA

CASSANDRA

- A distributed key-value store
- Intended to run in a datacenter (and also across DCs)
- Originally designed at Facebook
- Open-sourced later, today an Apache project
- Some of the companies that use Cassandra in their production clusters
 - IBM, Adobe, HP, eBay, Ericsson, Symantec
 - Twitter, Spotify
 - PBS Kids
 - Netflix: uses Cassandra to keep track of your current position in the video you're watching



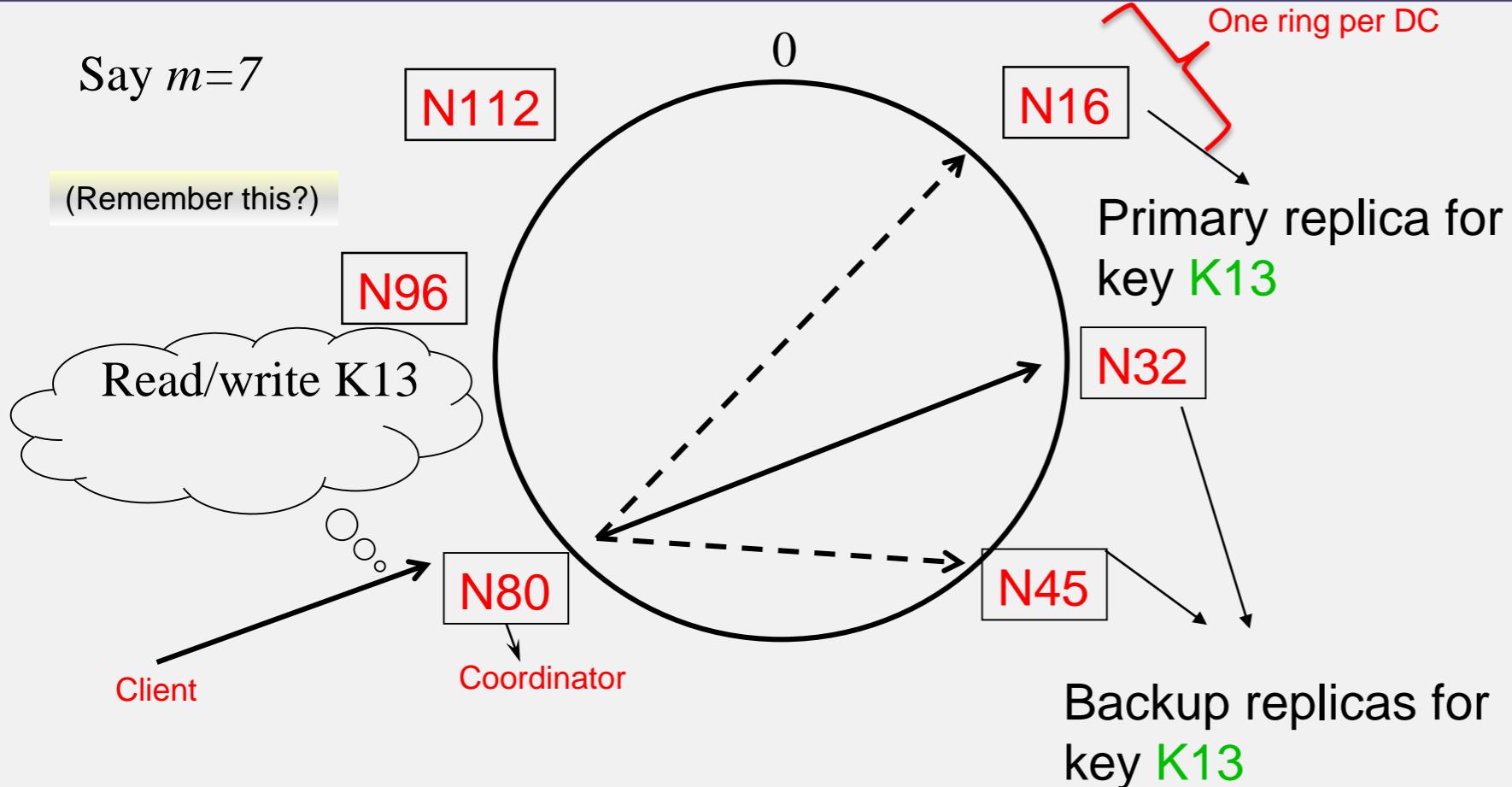
LET'S GO INSIDE CASSANDRA:

KEY -> SERVER MAPPING

- How do you decide which server(s) a key-value resides on?



Say $m=7$



Cassandra uses a ring-based DHT but without finger tables or routing
Key → server mapping is the “Partitioner”



DATA PLACEMENT STRATEGIES

- Replication Strategy: two options:
 1. *SimpleStrategy*
 2. *NetworkTopologyStrategy*
- 1. SimpleStrategy: uses the Partitioner, of which there are two kinds
 1. *RandomPartitioner*: Chord-like hash partitioning
 2. *ByteOrderedPartitioner*: Assigns ranges of keys to servers.
 - Easier for *range queries* (e.g., get me all twitter users starting with [a-b])
- 2. NetworkTopologyStrategy: for multi-DC deployments
 - Two replicas per DC
 - Three replicas per DC
 - Per DC
 - First replica placed according to Partitioner
 - Then go clockwise around ring until you hit a different rack



SNITCHES

- Maps: IPs to racks and DCs. Configured in `cassandra.yaml` config file
- Some options:
 - SimpleSnitch: Unaware of Topology (Rack-unaware)
 - RackInferring: Assumes topology of network by octet of server's IP address
 - $101.201.301.401 = x.<\text{DC octet}>.<\text{rack octet}>.<\text{node octet}>$
 - PropertyFileSnitch: uses a config file
 - EC2Snitch: uses EC2
 - EC2 Region = DC
 - Availability zone = rack
- Other snitch options available



WRITES

- Need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one coordinator node in Cassandra cluster
 - Coordinator may be per-key, per-client, or per-query
 - Per-key Coordinator ensures writes for the key are serialized
- Coordinator uses Partitioner to send query to all replica nodes responsible for key
- When X replicas respond, coordinator returns an acknowledgement to the client
 - X? We'll see later.



WRITES (2)

- Always writable: Hinted Handoff mechanism
 - If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until down replica comes back up.
 - When all replicas are down, the Coordinator (front end) buffers writes (for up to a few hours).
- One ring per datacenter
 - Per-DC coordinator elected to coordinate with other DCs
 - Election done via Zookeeper, which runs a Paxos (consensus) variant
 - Paxos: elsewhere in this course



WRITES AT A REPLICA NODE

On receiving a write

1. Log it in disk commit log (for failure recovery)
2. Make changes to appropriate memtables
 - **Memtable** = In-memory representation of multiple key-value pairs
 - Cache that can be searched by key
 - Write-back cache as opposed to write-through

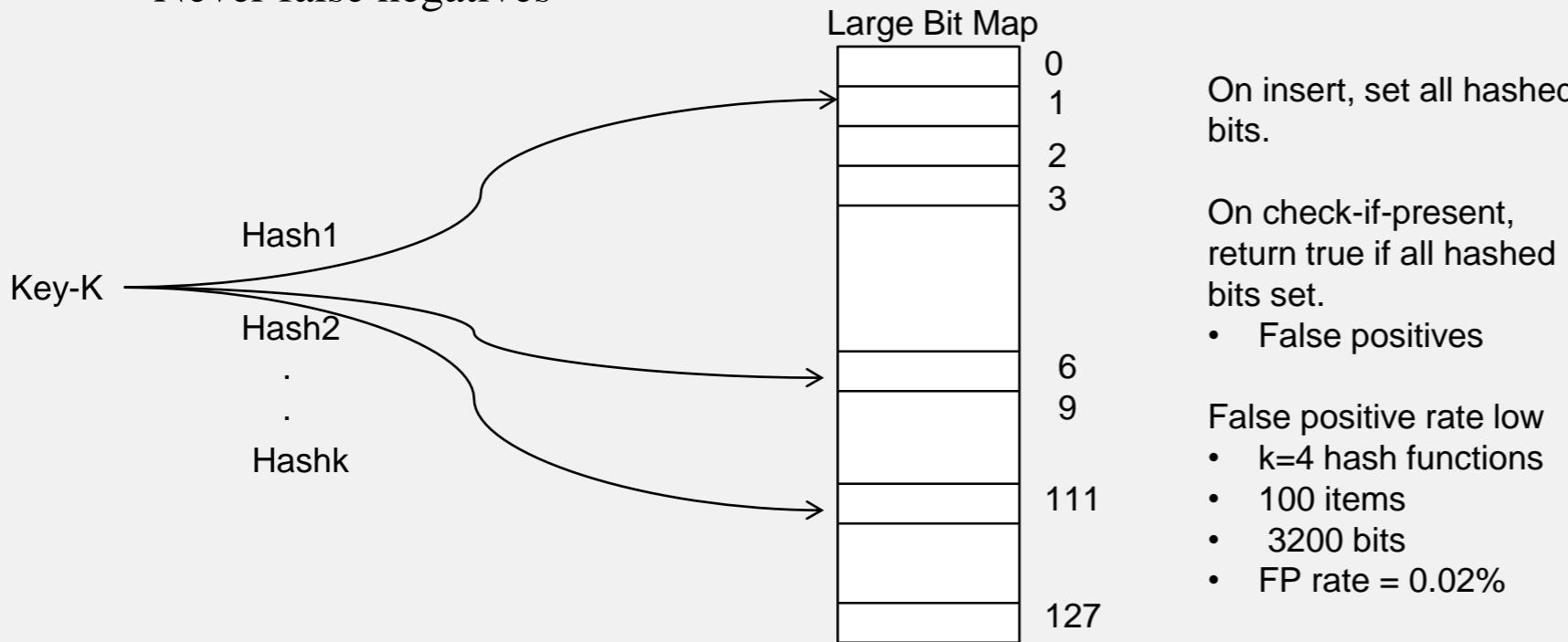
Later, when memtable is full or old, flush to disk

- Data file: An **SSTable** (Sorted String Table) – list of key-value pairs, sorted by key
- Index file: An SSTable of (key, position in data sstable) pairs
- And a Bloom filter (for efficient search) – next slide



BLOOM FILTER

- Compact way of representing a set of items
- Checking for existence in set is cheap
- Some probability of false positives: an item not in set may check true as being in set
- Never false negatives



On insert, set all hashed bits.

On check-if-present,
return true if all hashed
bits set.

- False positives

False positive rate low

- $k=4$ hash functions
- 100 items
- 3200 bits
- FP rate = 0.02%

COMPACTI

Data updates accumulate over time and SSTables and logs need to be compacted

- The process of compaction merges SSTables, i.e., by merging updates for a key
- Run periodically and locally at each server



DELETES

Delete: don't delete item right away

- Add a **tombstone** to the log
- Eventually, when compaction encounters tombstone it will delete item



READS

Read: Similar to writes, except

- Coordinator can contact X replicas (e.g., in same rack)
 - Coordinator sends read to replicas that have responded quickest in past
 - When X replicas respond, coordinator returns the latest-timestamped value from among those X
 - (X? We'll see later.)
- Coordinator also fetches value from other replicas
 - Checks consistency in the background, initiating a **read repair** if any two values are different
 - This mechanism seeks to eventually bring all replicas up to date
- A row may be split across multiple SSTables => reads need to touch multiple SSTables => reads slower than writes (but still fast)



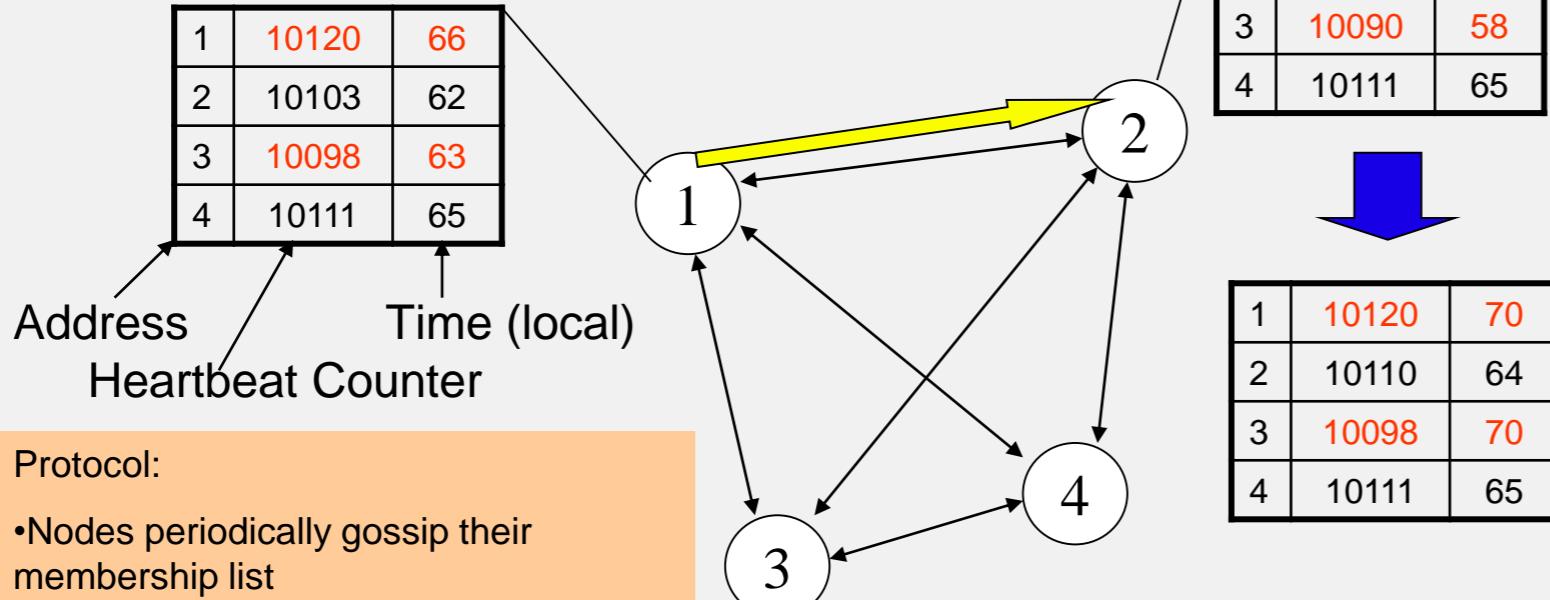
MEMBERSHIP

- Any server in cluster could be the coordinator
- So every server needs to maintain a list of all the other servers that are currently in the server
- List needs to be updated automatically as servers join, leave, and fail



CLUSTER MEMBERSHIP – GOSSIP-STYLE

Cassandra uses gossip-based cluster membership



(Remember this?)



SUSPICION MECHANISMS IN CASSANDRA

- Suspicion mechanisms to adaptively set the timeout based on underlying network and failure behavior
- Accrual detector: Failure detector outputs a value (PHI) representing suspicion
- Apps set an appropriate threshold
- PHI calculation for a member
 - Inter-arrival times for gossip messages
 - $\text{PHI}(t) = -\log(\text{CDF or Probability}(t_{\text{now}} - t_{\text{last}})) / \log 10$
 - PHI basically determines the detection timeout, but takes into account historical inter-arrival time variations for gossiped heartbeats
- In practice, $\text{PHI} = 5 \Rightarrow 10-15 \text{ sec detection time}$



CASSANDRA Vs. RDBMS

- MySQL is one of the most popular (and has been for a while)
- On > 50 GB data
- MySQL
 - Writes 300 ms avg
 - Reads 350 ms avg
- Cassandra
 - Writes 0.12 ms avg
 - Reads 15 ms avg
- Orders of magnitude faster
- What's the catch? What did we lose?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

KEY-VALUE STORES NoSQL

Lecture C

THE MYSTERY OF X
THE CAP THEOREM

CAP THEOREM

- Proposed by Eric Brewer (Berkeley)
- Subsequently proved by Gilbert and Lynch (NUS and MIT)
- In a distributed system you can satisfy at most 2 out of the 3 guarantees:
 1. **Consistency**: all nodes see same data at any time, or reads return latest written value by any client
 2. **Availability**: the system allows operations all the time, and operations return quickly
 3. **Partition-tolerance**: the system continues to work in spite of network partitions



WHY IS AVAILABILITY IMPORTANT?

- Availability = Reads/writes complete reliably and quickly.
- Measurements have shown that a 500 ms increase in latency for operations at Amazon.com or at Google.com can cause a 20% drop in revenue.
- At Amazon, each added millisecond of latency implies a \$6M yearly loss.
- SLAs (Service Level Agreements) written by providers predominantly deal with latencies faced by clients.



WHY IS CONSISTENCY IMPORTANT?

- Consistency = all nodes see same data at any time, or reads return latest written value by any client.
- When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.
- When thousands of customers are looking to book a flight, all updates from any client (e.g., book a flight) should be accessible by other clients.



WHY IS PARTITION-TOLERANCE IMPORTANT?

- Partitions can happen across datacenters when the Internet gets disconnected
 - Internet router outages
 - Under-sea cables cut
 - DNS not working
- Partitions can also occur within a datacenter, e.g., a rack switch outage
- Still desire system to continue functioning normally under this scenario



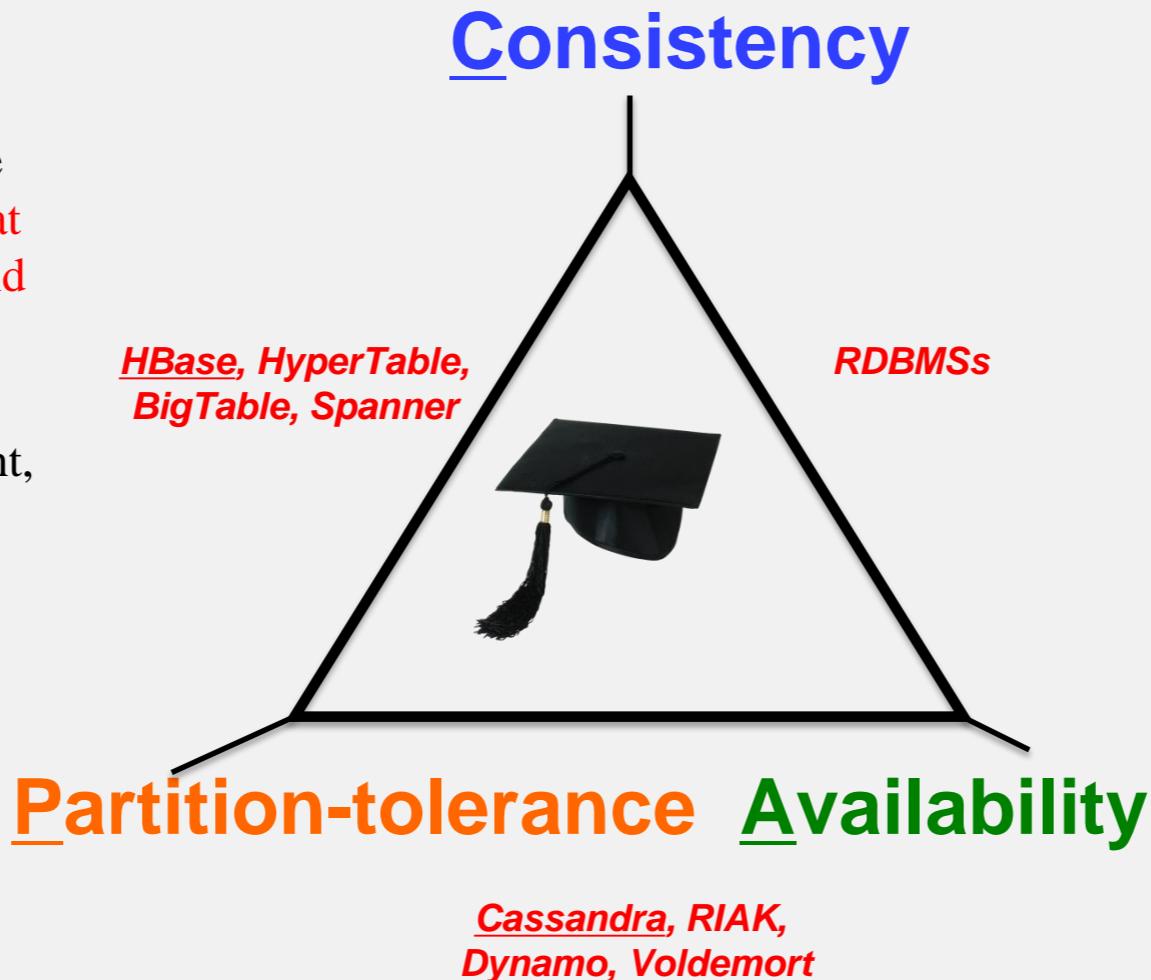
CAP THEOREM FALLOUT

- Since partition-tolerance is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability
- Cassandra
 - Eventual (weak) consistency, availability, partition-tolerance
- Traditional RDBMSs
 - Strong consistency over availability under a partition



CAP TRADEOFF

- Starting point for NoSQL Revolution
- A distributed storage system can achieve at most two of C, A, and P.
- When partition-tolerance is important, you have to choose between consistency and availability



EVENTUAL CONSISTENCY

- If all writes stop (to a key), then all its values (replicas) will converge eventually.
- If writes continue, then system always tries to keep converging.
 - Moving “wave” of updated values lagging behind the latest values sent by clients, but always trying to catch up.
- May still return stale values to clients (e.g., if many back-to-back writes).
- But works well when there a few periods of low writes – system converges quickly.



RDBMS vs. KEY-VALUE STORES

- While RDBMS provide **ACID**
 - Atomicity
 - Consistency
 - Isolation
 - Durability
- Key-value stores like Cassandra provide **BASE**
 - Basically Available Soft-state Eventual consistency
 - Prefers availability over consistency



BACK TO CASSANDRA: MYSTERY OF X

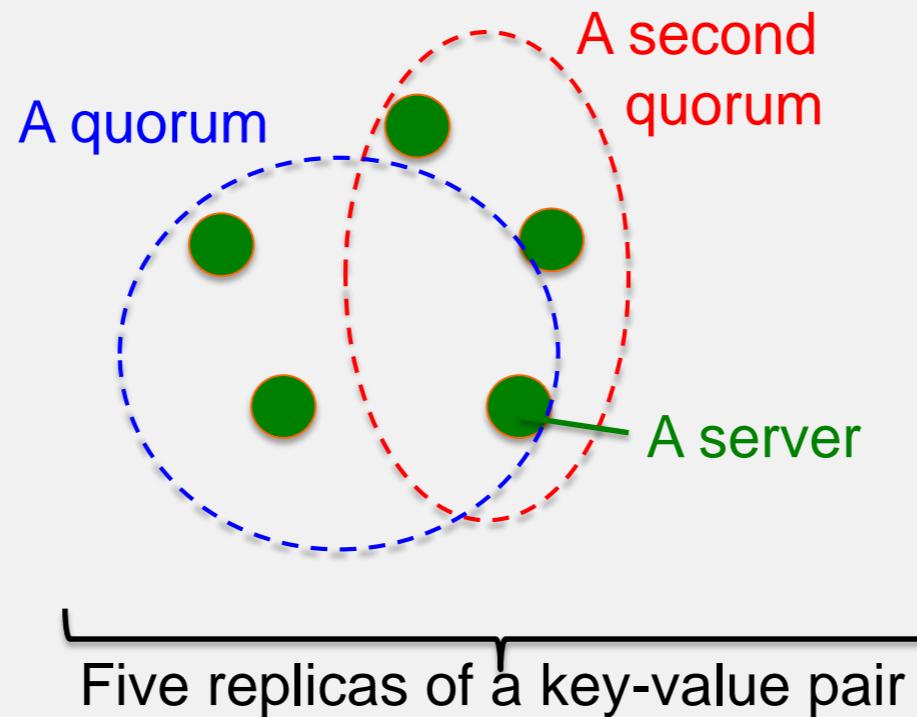
- Cassandra has **consistency levels**
- Client is allowed to choose a consistency level for each operation (read/write)
 - ANY: any server (may not be replica)
 - Fastest: coordinator caches write and replies quickly to client
 - ALL: all replicas
 - Ensures strong consistency, but slowest
 - ONE: at least one replica
 - Faster than ALL, but cannot tolerate a failure
 - QUORUM: quorum across all replicas in all datacenters (DCs)
 - What?



QUORUMS?

In a nutshell:

- Quorum = majority
 - $> 50\%$
- Any two quorums intersect
 - Client 1 does a write in red quorum
 - Then client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums faster than ALL, but still ensure strong consistency



QUORUMS IN DETAIL

- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.
- Reads
 - Client specifies value of **R** ($\leq N$ = total number of replicas of that key).
 - R = read consistency level.
 - Coordinator waits for R replicas to respond before sending result to client.
 - In background, coordinator checks for consistency of remaining $(N-R)$ replicas, and initiates read repair if needed.



QUORUMS IN DETAIL (CONTD.)

- Writes come in two flavors
 - Client specifies W ($\leq N$)
 - W = write consistency level.
 - Client writes new value to W replicas and returns. Two flavors:
 - Coordinator blocks until quorum is reached.
 - Asynchronous: Just write and return.



QUORUMS IN DETAIL (CONTD.)

- R = read replica count, W = write replica count
- Two necessary conditions:
 1. $W+R > N$
 2. $W > N/2$
- Select values based on application
 - ($W=1, R=1$): very few writes and reads
 - ($W=N, R=1$): great for read-heavy workloads
 - ($W=N/2+1, R=N/2+1$): great for write-heavy workloads
 - ($W=1, R=N$): great for write-heavy workloads with mostly one client writing per key



CASSANDRA CONSISTENCY LEVELS (CONTD.)

- Client is allowed to choose a consistency level for each operation (read/write)
 - ANY: any server (may not be replica)
 - Fastest: coordinator may cache write and reply quickly to client
 - ALL: all replicas
 - Slowest, but ensures strong consistency
 - ONE: at least one replica
 - Faster than ALL, and ensures durability without failures
- QUORUM: quorum across all replicas in all datacenters (DCs)
 - Global consistency, but still fast
- LOCAL_QUORUM: quorum in coordinator's DC
 - Faster: only waits for quorum in first DC client contacts
- EACH_QUORUM: quorum in every DC
 - Lets each DC do its own quorum: supports hierarchical replies



TYPES OF CONSISTENCY

- Cassandra offers eventual consistency
- Are there other types of weak consistency models?



CLOUD COMPUTING CONCEPTS

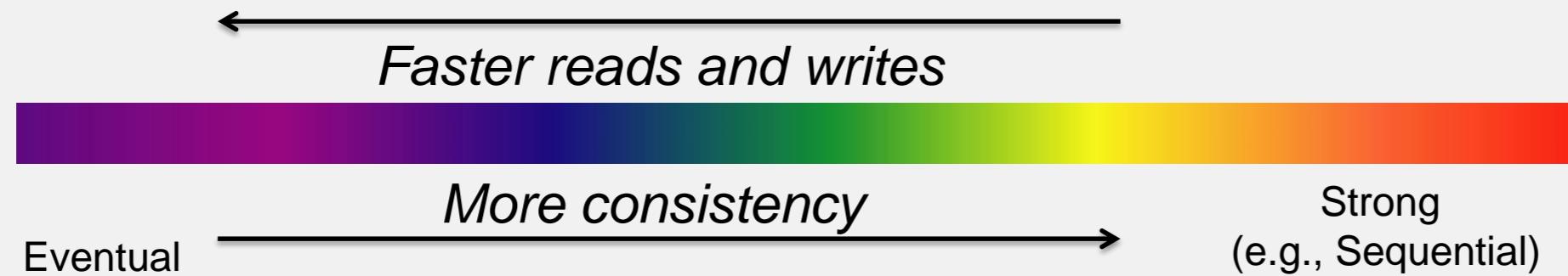
with Indranil Gupta (Indy)

KEY-VALUE STORES NoSQL

Lecture D

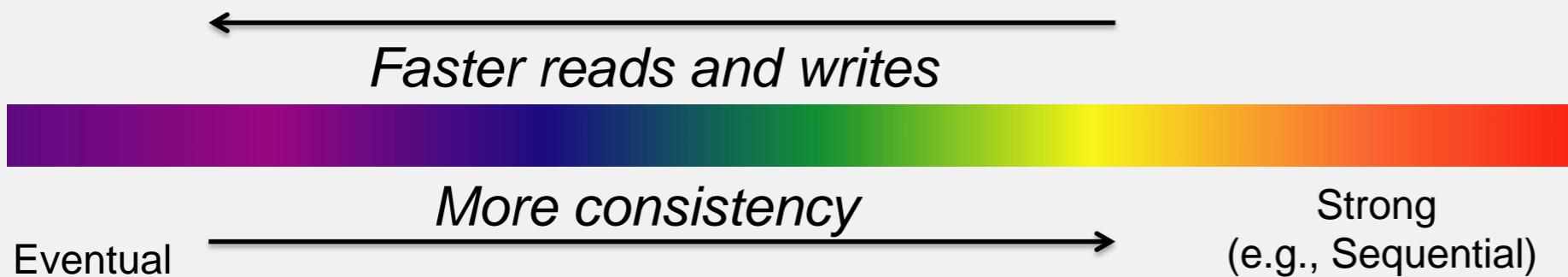
THE CONSISTENCY SPECTRUM

CONSISTENCY SPECTRUM



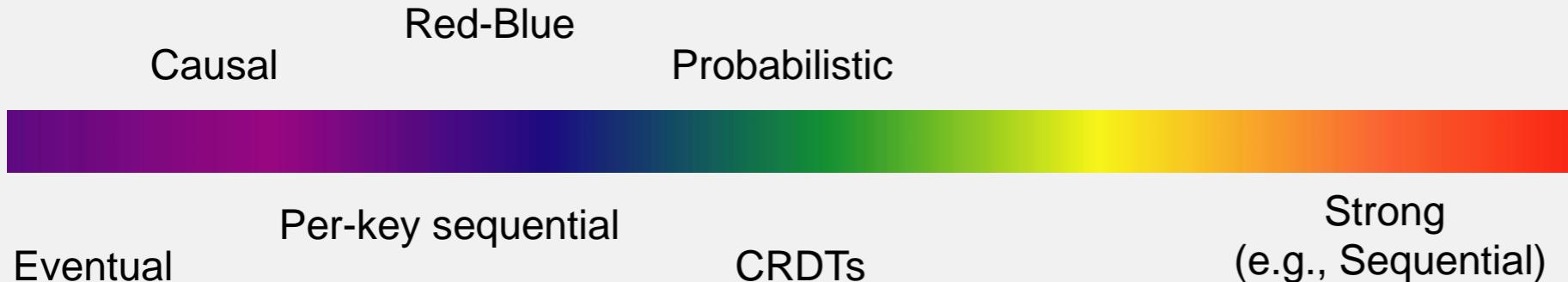
CONSISTENCY SPECTRUM

- Cassandra offers **eventual consistency**
 - If writes to a key stop, all replicas of key will converge
 - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems



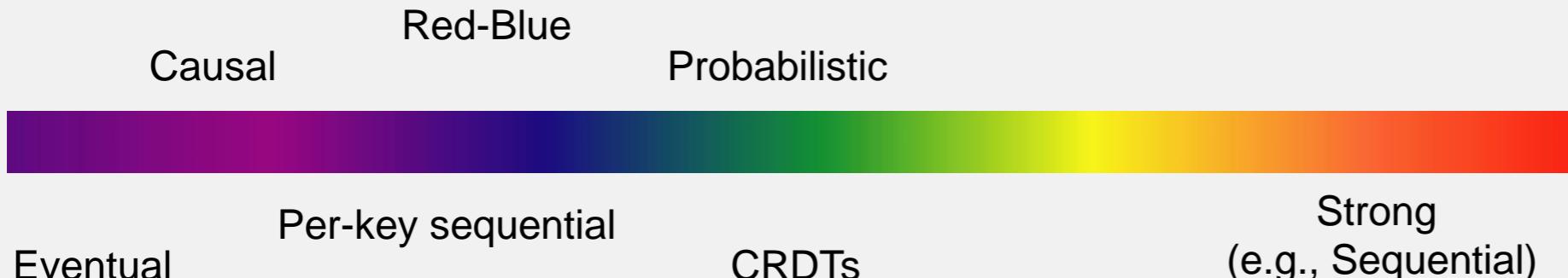
NEWER CONSISTENCY MODELS

- Striving towards strong consistency
- While still trying to maintain high availability and partition-tolerance



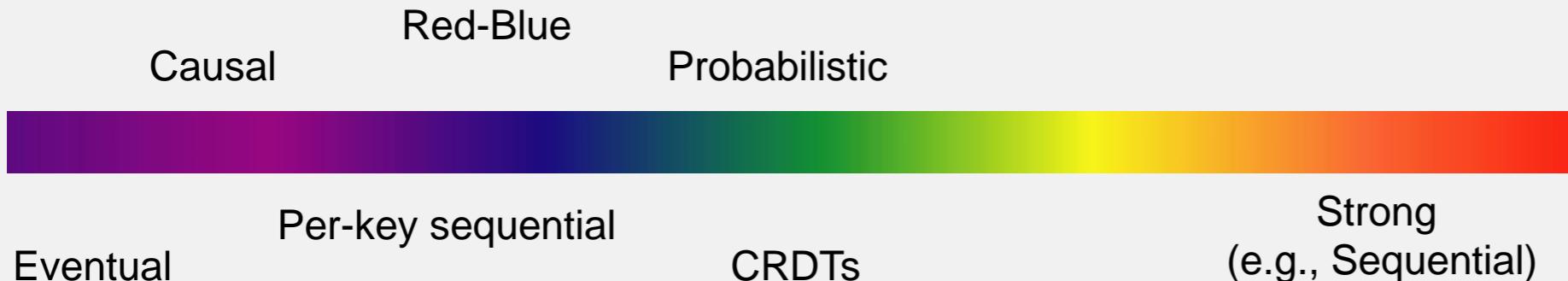
NEWER CONSISTENCY MODELS (CONTD.)

- **Per-key sequential:** Per key, all operations have a global order
- **CRDTs** (Commutative Replicated Data Types): Data structures for which commutated writes give same result [INRIA, France]
 - E.g., value == int, and only op allowed is +1
 - Effectively, servers don't need to worry about consistency



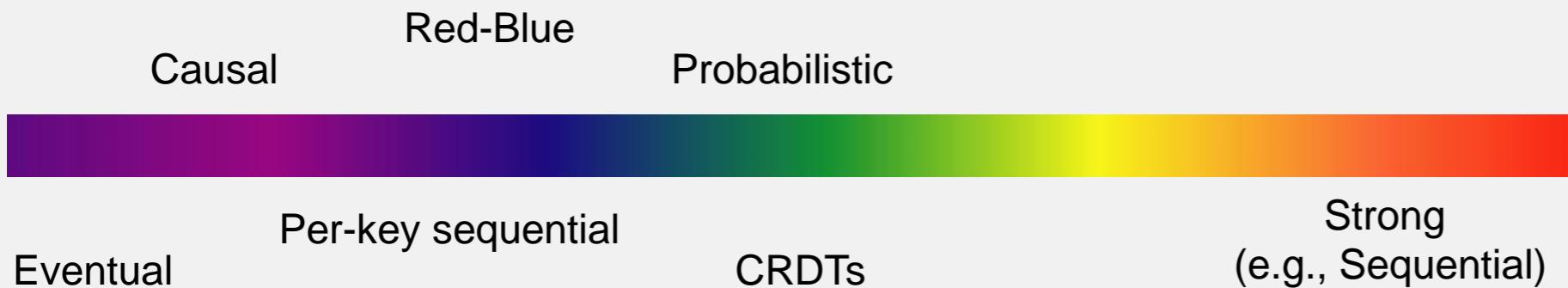
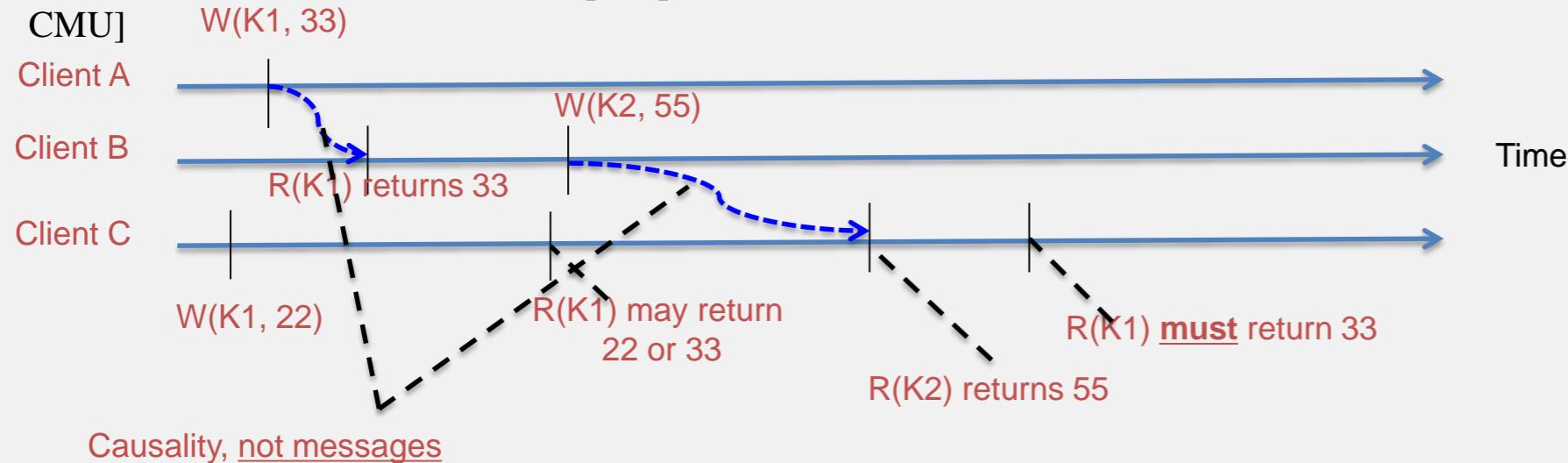
NEWER CONSISTENCY MODELS (CONTD.)

- **Red-blue consistency:** Rewrite client transactions to separate ops into red ops vs. blue ops [MPI-SWS Germany]
 - Blue ops can be executed (commutated) in any order across DCs
 - Red ops need to be executed in the same order at each DC



NEWER CONSISTENCY MODELS (CONTD.)

Causal Consistency: Reads must respect partial order based on information flow [Princeton, CMU]



STRONG CONSISTENCY MODELS

- **Linearizability:** Each operation by a client is visible (or available) instantaneously to all other clients
 - Instantaneously in real time
- **Sequential Consistency** [Lamport]:
 - *... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
 - After the fact, find a “reasonable” ordering of the operations (can re-order operations) that obeys sanity (consistency) at all clients, and across clients.
- Transaction ACID properties, e.g., newer key-value/NoSQL stores (sometimes called “NewSQL”)
 - Hyperdex [Cornell]
 - Spanner [Google]
 - Transaction chains [Microsoft Research]



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

KEY-VALUE STORES NoSQL

Lecture E

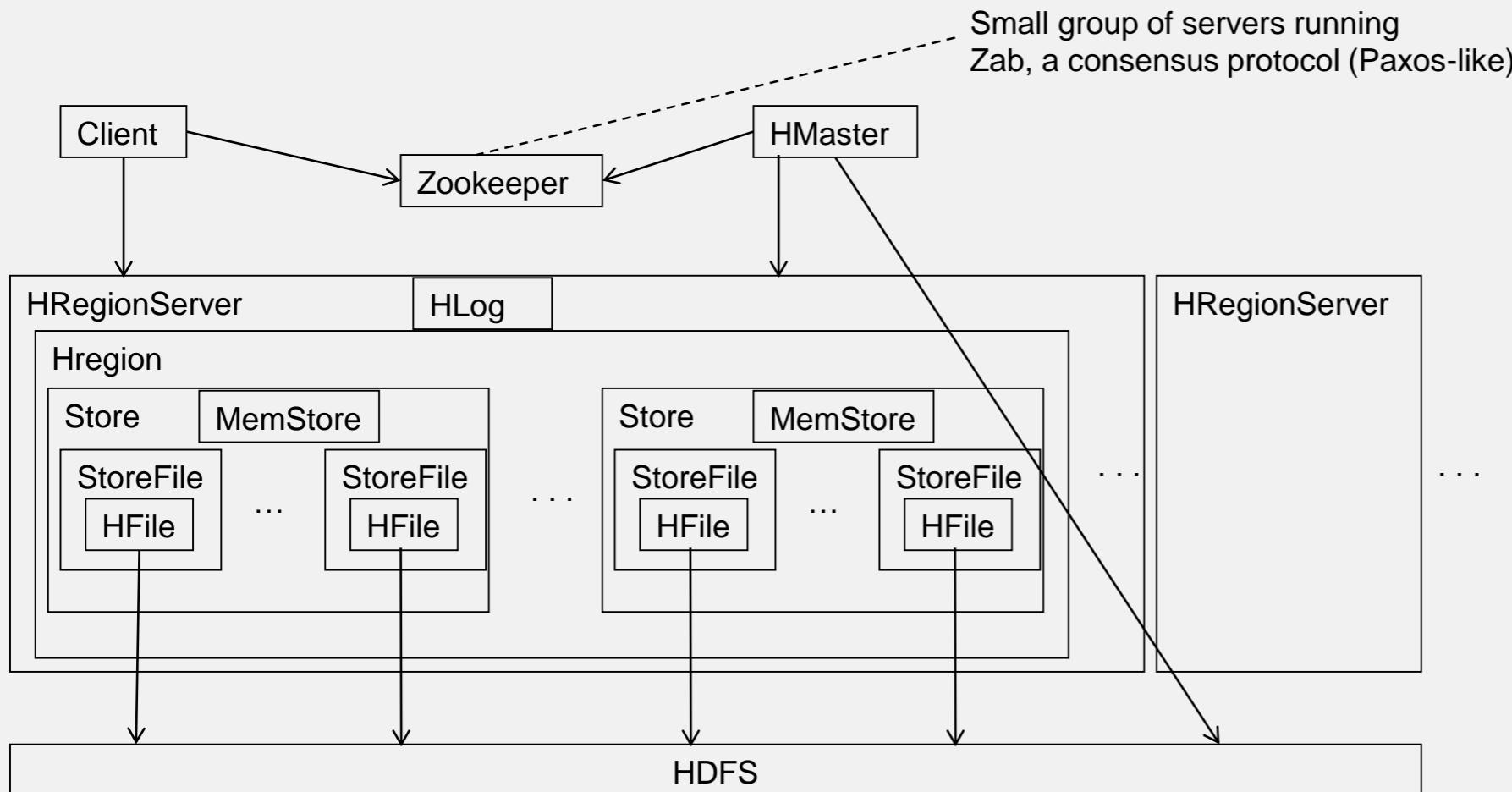
HBASE

HBASE

- Google's BigTable was first “blob-based” storage system
- Yahoo! Open-sourced it → HBase
- Major Apache project today
- Facebook uses HBase internally
- API functions
 - Get/Put(row)
 - Scan(row range, filter) – range queries
 - MultiPut
- Unlike Cassandra, HBase prefers consistency (over availability)



HBASE ARCHITECTURE

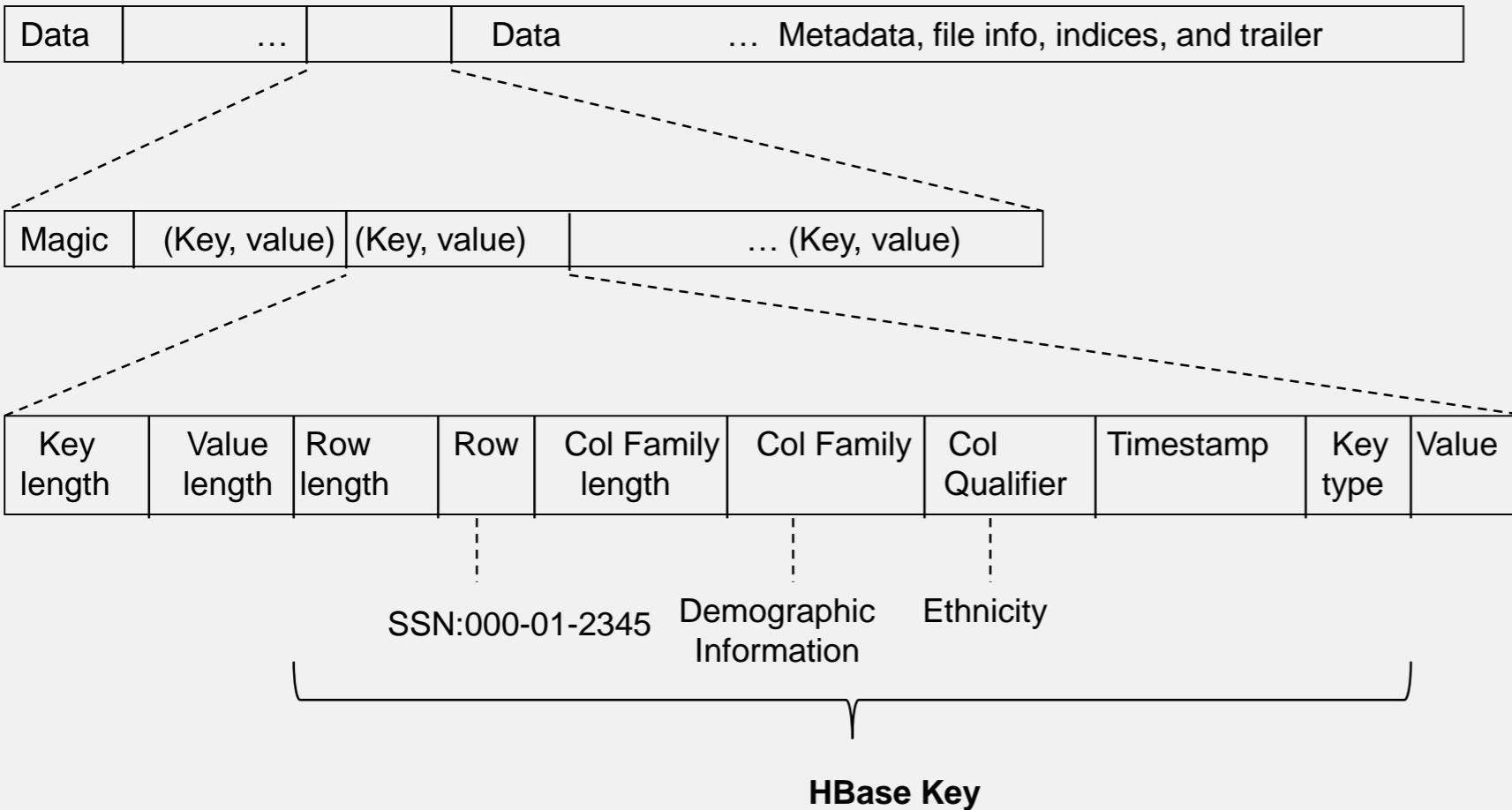


HBASE STORAGE HIERARCHY

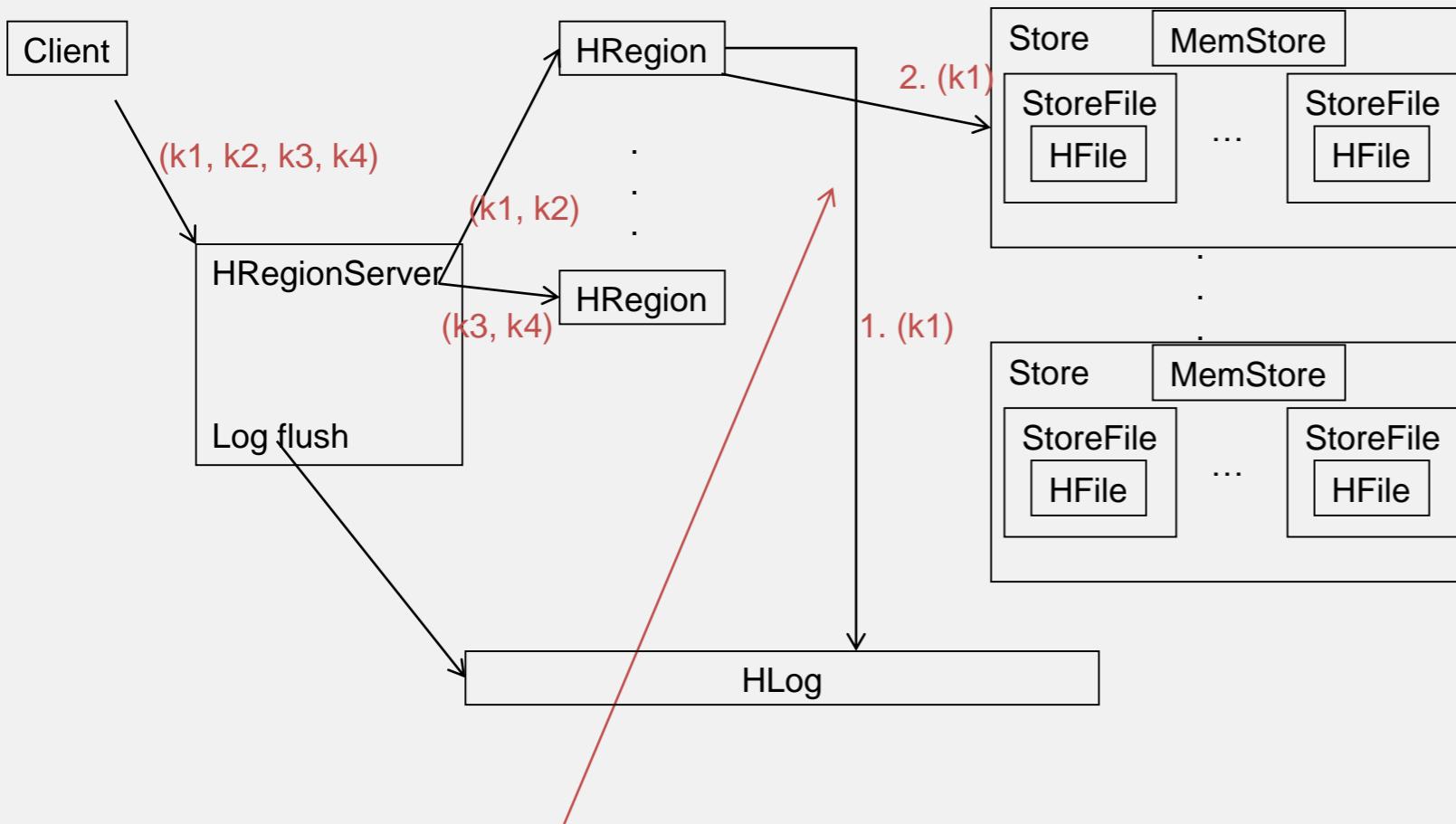
- HBase Table
 - Split it into multiple [regions](#): replicated across servers
 - ColumnFamily = subset of columns with similar query patterns
 - One [Store](#) per combination of ColumnFamily + region
 - [Memstore](#) for each store: in-memory updates to store; flushed to disk when full
 - [StoreFiles](#) for each store for each region: where the data lives
 - [HFile](#)
- HFile
 - SSTable from Google's BigTable



HFILE



STRONG CONSISTENCY: HBASE WRITE-AHEAD LOG



Write to HLog before writing to MemStore
Helps recover from failure by replaying Hlog.



LOG REPLAY

- After recovery from failure, or upon bootup (HRegionServer/HMaster)
 - Replay any stale logs (use timestamps to find out where the database is w.r.t. the logs)
 - Replay: add edits to the MemStore



CROSS-DATACENTER REPLICATION

- Single “Master” cluster
- Other “Slave” clusters replicate the same tables
- Master cluster synchronously sends HLogs over to slave clusters
- Coordination among clusters is via Zookeeper
- Zookeeper can be used like a file system to store control information

1. */hbasereplication/state*
2. */hbasereplication/peers/<peer cluster number>*
3. */hbasereplication/rs/<hlog>*



SUMMARY

- Traditional databases (RDBMSs) work with strong consistency and offer ACID
- Modern workloads don't need such strong guarantees but do need fast response times (availability)
- Unfortunately, CAP theorem
- Key-value/NoSQL systems offer BASE
 - Eventual consistency, and a variety of other consistency models striving towards strong consistency
- We discussed design of
 - Cassandra
 - HBase





CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

TIME AND ORDERING

Lecture A

INTRODUCTION AND BASICS

WHY SYNCHRONIZATION?

- You want to catch a bus at 6:05 pm, but your watch is off by 15 minutes.
 - What if your watch is late by 15 minutes?
 - You'll miss the bus!
 - What if your watch is fast by 15 minutes?
 - You'll end up unfairly waiting for a longer time than you intended.
- Time synchronization is required for both
 - Correctness
 - Fairness

SYNCHRONIZATION IN THE CLOUD

- Cloud airline reservation system
- Server A receives a client request to purchase last ticket on flight ABC 123.
- Server A timestamps purchase using local clock **9h:15m:32.45s**, and logs it. Replies ok to client.
- That was the last seat. Server A sends message to Server B saying “flight full.”
- B enters “Flight ABC 123 full” + its own local clock value (which reads **9h:10m:10.11s**) into its log.
- Server C queries A’s and B’s logs. Is confused that a client purchased a ticket at A after the flight became full at B.
- **This may lead to further incorrect actions by C**

WHY IS IT CHALLENGING?

- **End hosts in Internet-based systems (like clouds)**
 - Each have their own clocks
 - Unlike processors (CPUs) within one server or workstation which share a system clock
- **Processes in Internet-based systems follow an *asynchronous* system model**
 - No bounds on
 - Message delays
 - Processing delays
 - Unlike multi-processor (or parallel) systems which follow a *synchronous* system model

SOME DEFINITIONS

- An asynchronous distributed system consists of a number of processes
- Each process has a state (values of variables).
- Each process takes actions to change its state, which may be an instruction or a communication action (send, receive).
- An event is the occurrence of an action.
- Each process has a local clock – events *within* a process can be assigned timestamps, and thus ordered linearly.
- But – in a distributed system, we also need to know the time order of events across different processes.

CLOCK SKEW VS. CLOCK DRIFT

- Each process (running at some end host) has its own clock.
- When comparing two clocks at two processes:
 - Clock **Skew** = Relative Difference in clock *values* of two processes
 - Like distance between two vehicles on a road
 - Clock **Drift** = Relative Difference in clock *frequencies (rates)* of two processes
 - Like difference in speeds of two vehicles on the road
- A non-zero clock skew implies clocks are not synchronized.
- A non-zero clock drift causes skew to increase (eventually).
 - If faster vehicle is ahead, it will drift away
 - If faster vehicle is behind, it will catch up and then drift away

HOW OFTEN TO SYNCHRONIZE?

- Maximum Drift Rate (**MDR**) of a clock
- Absolute MDR is defined relative to Coordinated Universal Time (UTC). UTC is the “correct” time at any point of time.
 - MDR of a process depends on the environment.
- Max drift rate between two clocks with similar MDR is **$2 * MDR$**
- Given a maximum acceptable skew M between any pair of clocks, need to synchronize at least once every: $M / (2 * MDR)$ time units
 - Since time = distance/speed

EXTERNAL VS INTERNAL SYNCHRONIZATION

- Consider a group of processes
- External Synchronization
 - Each process $C(i)$'s clock is within a bound D of a well-known clock S external to the group
 - $|C(i) - S| < D$ at all times
 - External clock may be connected to UTC (Universal Coordinated Time) or an atomic clock
 - E.g., Cristian's algorithm, NTP
- Internal Synchronization
 - Every pair of processes in group have clocks within bound D
 - $|C(i) - C(j)| < D$ at all times and for all processes i, j
 - E.g., Berkeley algorithm

EXTERNAL VS INTERNAL SYNCHRONIZATION (2)

- **External Synchronization with D => Internal Synchronization with 2*D**
- **Internal synchronization does not imply external synchronization**
 - In fact, the entire system may drift away from the external clock S!

NEXT

- Algorithms for clock synchronization



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

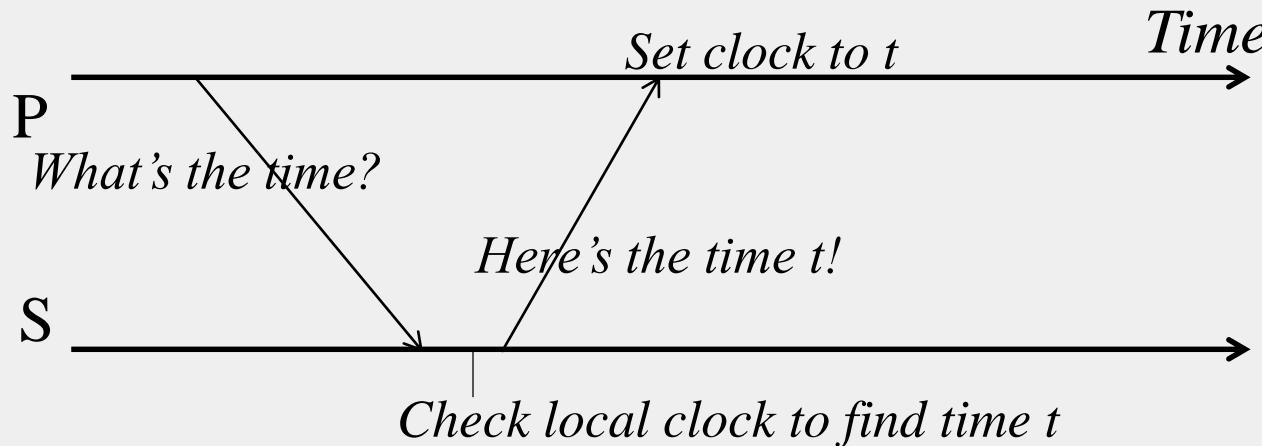
TIME AND ORDERING

Lecture B

CRISTIAN'S ALGORITHM

BASICS

- External time synchronization
- All processes P synchronize with a time server S

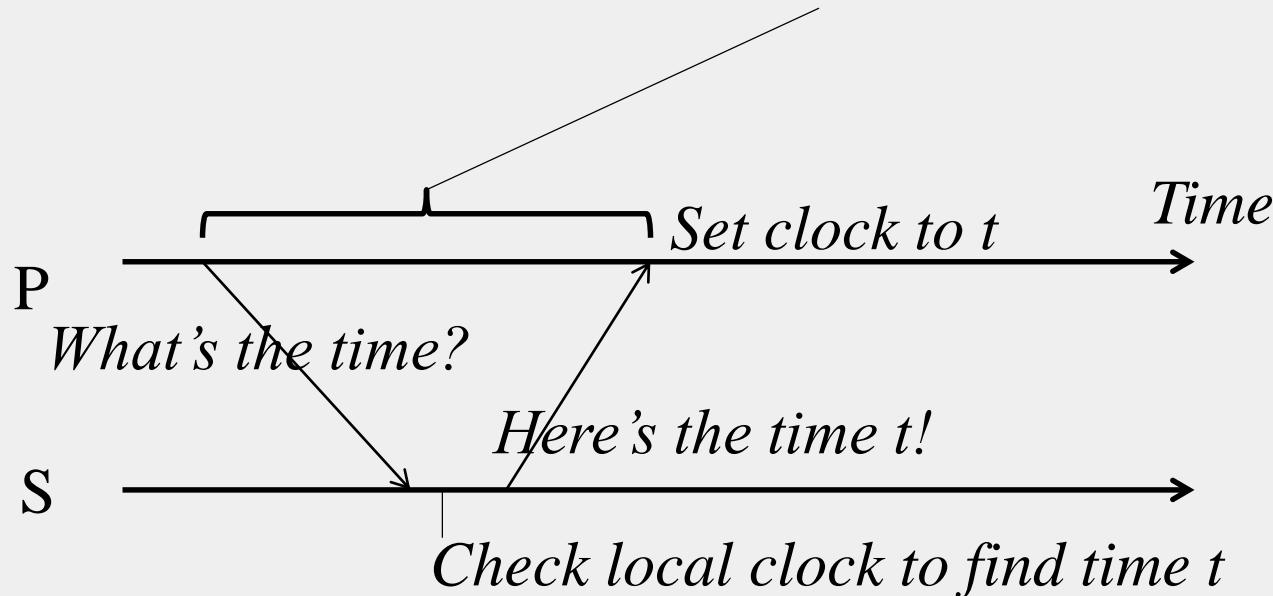


WHAT'S WRONG

- By the time response message is received at P,
time has moved on
- P's time set to t is inaccurate!
- Inaccuracy a function of message latencies
- Since latencies unbounded in an asynchronous
system, the inaccuracy cannot be bounded

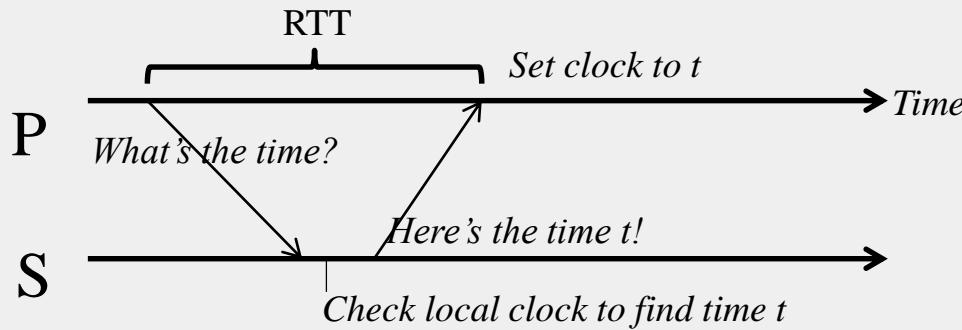
CRISTIAN'S ALGORITHM

- P measures the round-trip-time RTT of message exchange



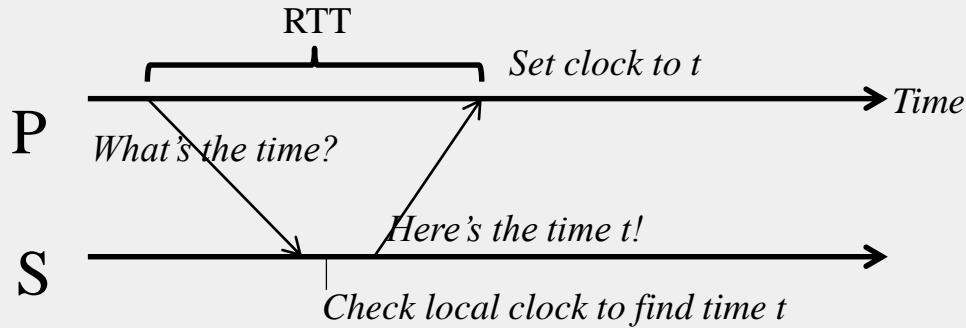
CRISTIAN'S ALGORITHM (2)

- P measures the round-trip-time RTT of message exchange
- Suppose we know the minimum P \rightarrow S latency min1
- And the minimum S \rightarrow P latency min2
 - min1 and min2 depend on operating system overhead to buffer messages, TCP time to queue messages, etc.



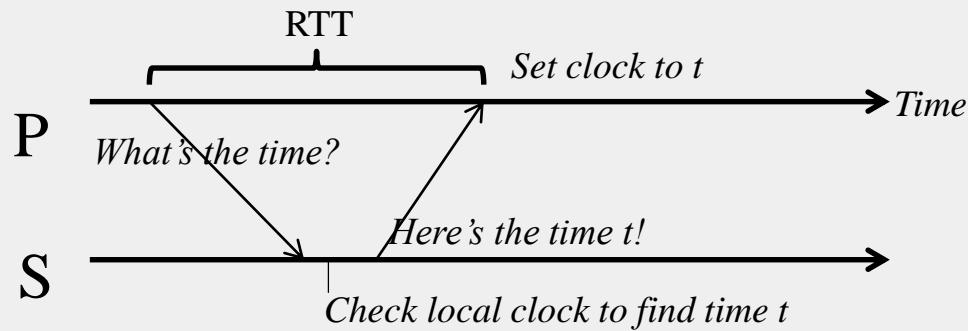
CRISTIAN'S ALGORITHM (3)

- P measures the round-trip-time RTT of message exchange
- Suppose we know the minimum $P \rightarrow S$ latency min1
- And the minimum $S \rightarrow P$ latency min2
 - min1 and min2 depend on Operating system overhead to buffer messages, TCP time to queue messages, etc.
- The actual time at P when it receives response is between $[t+min2, t+RTT-min1]$



CRISTIAN'S ALGORITHM (4)

- The actual time at P when it receives response is between $[t+\text{min2}, t+\text{RTT}-\text{min1}]$
- P sets its time to halfway through this interval
 - To: $t + (\text{RTT}+\text{min2}-\text{min1})/2$
- Error is at most $(\text{RTT}-\text{min2}-\text{min1})/2$
 - Bounded!



GOTCHAS

- **Allowed to increase clock value but should never decrease clock value**
 - May violate ordering of events within the same process
- **Allowed to increase or decrease speed of clock**
- **If error is too high, take multiple readings and average them**



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

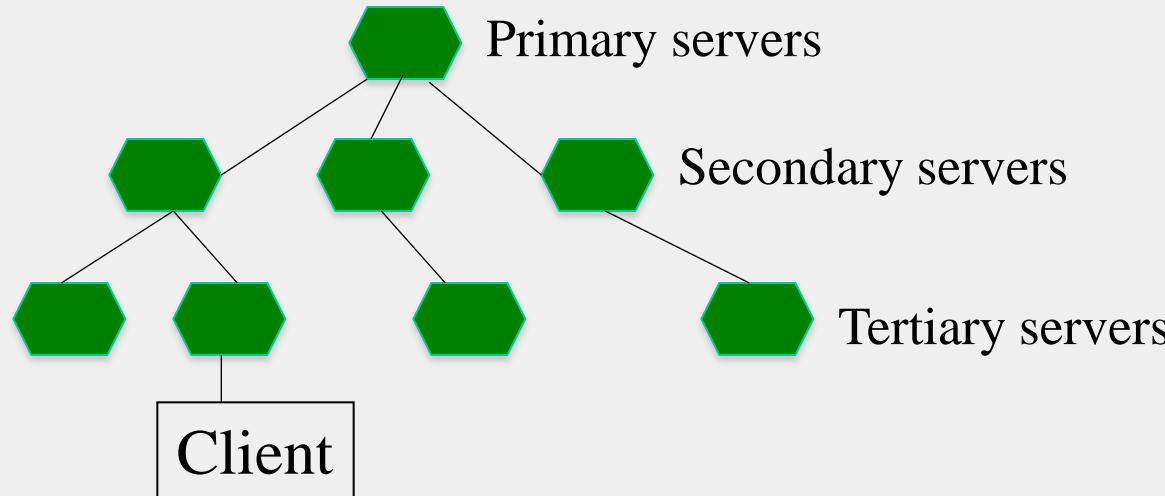
TIME AND ORDERING

Lecture C

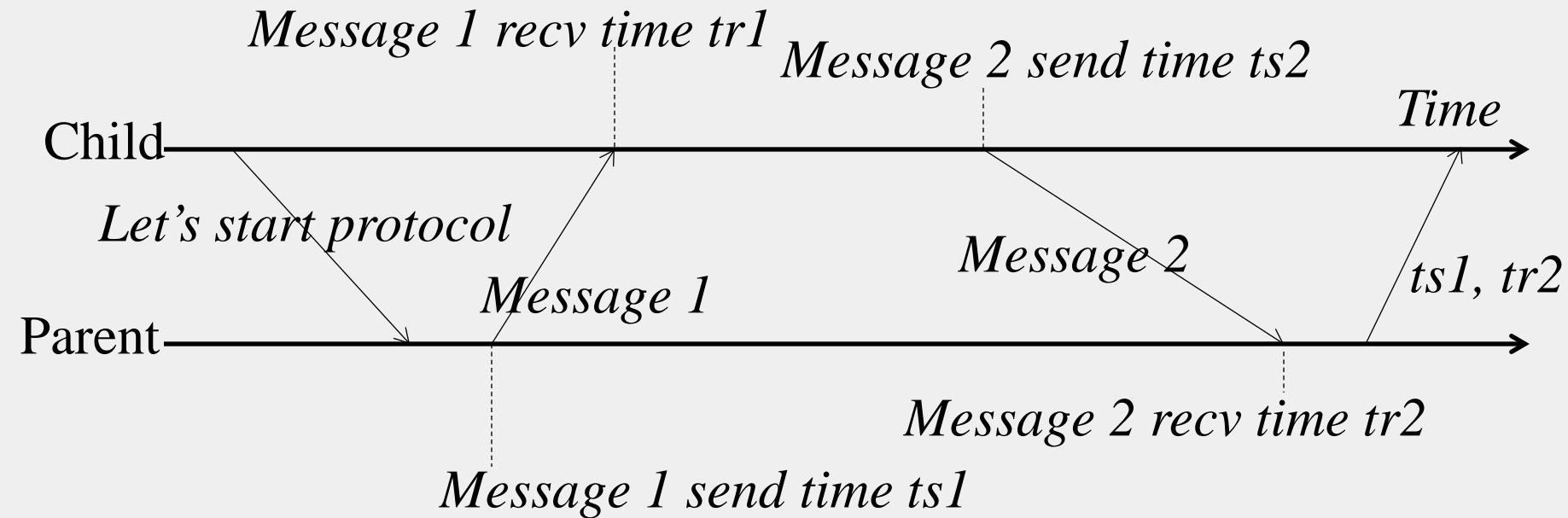
NTP

NTP = Network Time Protocol

- NTP servers organized in a tree
- Each client = a leaf of tree
- Each node synchronizes with its tree parent



NTP Protocol



What the Child Does

- Child calculates *offset* between its clock and parent's clock
- Uses $ts1, tr1, ts2, tr2$
- Offset is calculated as

$$o = (tr1 - tr2 + ts2 - ts1)/2$$

Why $o = (tr1 - tr2 + ts2 - ts1)/2$?

- Offset $o = (tr1 - tr2 + ts2 - ts1)/2$
- Let's calculate the error
- Suppose real offset is $oreal$
 - Child is ahead of parent by $oreal$
 - Parent is ahead of child by $-oreal$
- Suppose one-way latency of Message 1 is $L1$ ($L2$ for Message 2)
- No one knows $L1$ or $L2$!
- Then

$$tr1 = ts1 + L1 + oreal$$

$$tr2 = ts2 + L2 - oreal$$

Why $o = (tr1 - tr2 + ts2 - ts1)/2$?

- Then

$$tr1 = ts1 + L1 + oreal$$

$$tr2 = ts2 + L2 - oreal$$

- Subtracting second equation from the first

$$oreal = (tr1 - tr2 + ts2 - ts1)/2 + (L2 - L1)/2$$

$$\Rightarrow oreal = o + (L2 - L1)/2$$

$$\Rightarrow |oreal - o| < |(L2 - L1)/2| < |(L2 + L1)/2|$$

- Thus, the error is bounded by the round-trip-time

And yet...

- **We still have a non-zero error!**
- **We just can't seem to get rid of error**
 - Can't, as long as message latencies are non-zero
- **Can we avoid synchronizing clocks altogether and still be able to order events?**



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

TIME AND ORDERING

Lecture D

LAMPORT TIMESTAMPS

ORDERING EVENTS IN A DISTRIBUTED SYSTEM

- To order events across processes, trying to sync clocks is one approach.
- What if we instead assigned timestamps to events that were not *absolute* time?
- As long as these timestamps obey *causality*, that would work.

If an event A causally happens before another event B, then $\text{timestamp}(A) < \text{timestamp}(B)$.

Humans use causality all the time.

E.g., I enter a house only after I unlock it.

E.g., you receive a letter only after I send it.

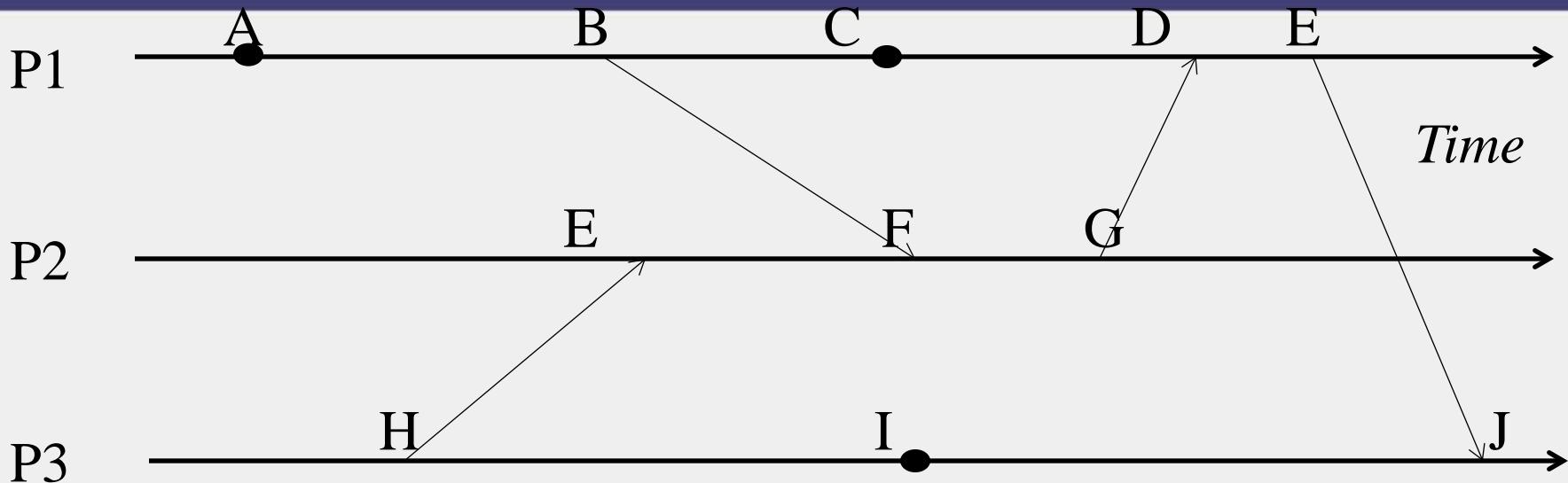
LOGICAL (OR LAMPORT) ORDERING

- Proposed by Leslie Lamport in the 1970s
- Used in almost all distributed systems since then
- Almost all cloud computing systems use some form of logical ordering of events

LOGICAL (OR LAMPORT) ORDERING(2)

- Define a logical relation *Happens-Before* among pairs of events
- Happens-Before denoted as \rightarrow
- Three rules
 1. On the same process: $a \rightarrow b$, if $time(a) < time(b)$ (using the local clock)
 2. If p1 sends m to p2: $send(m) \rightarrow receive(m)$
 3. (Transitivity) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
- Creates a *partial order* among events
 - Not all events related to each other via \rightarrow

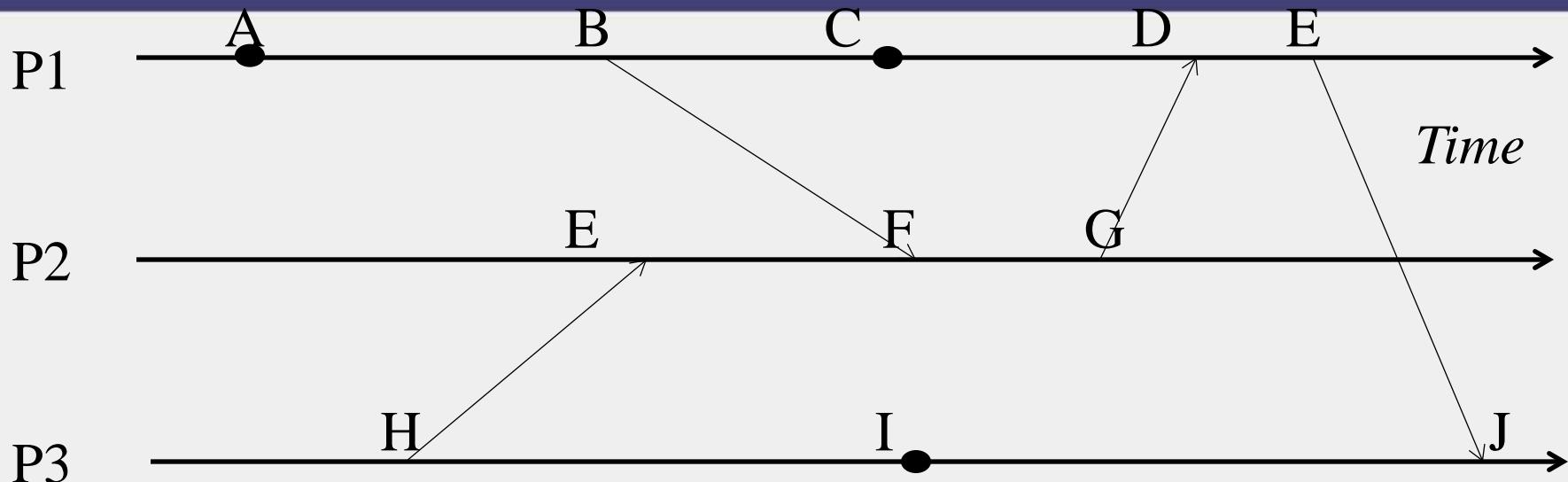
EXAMPLE



While P1 and P3 each have an event labeled E, these are different events as they occur at different processes

● *Instruction or step*
→ *Message*

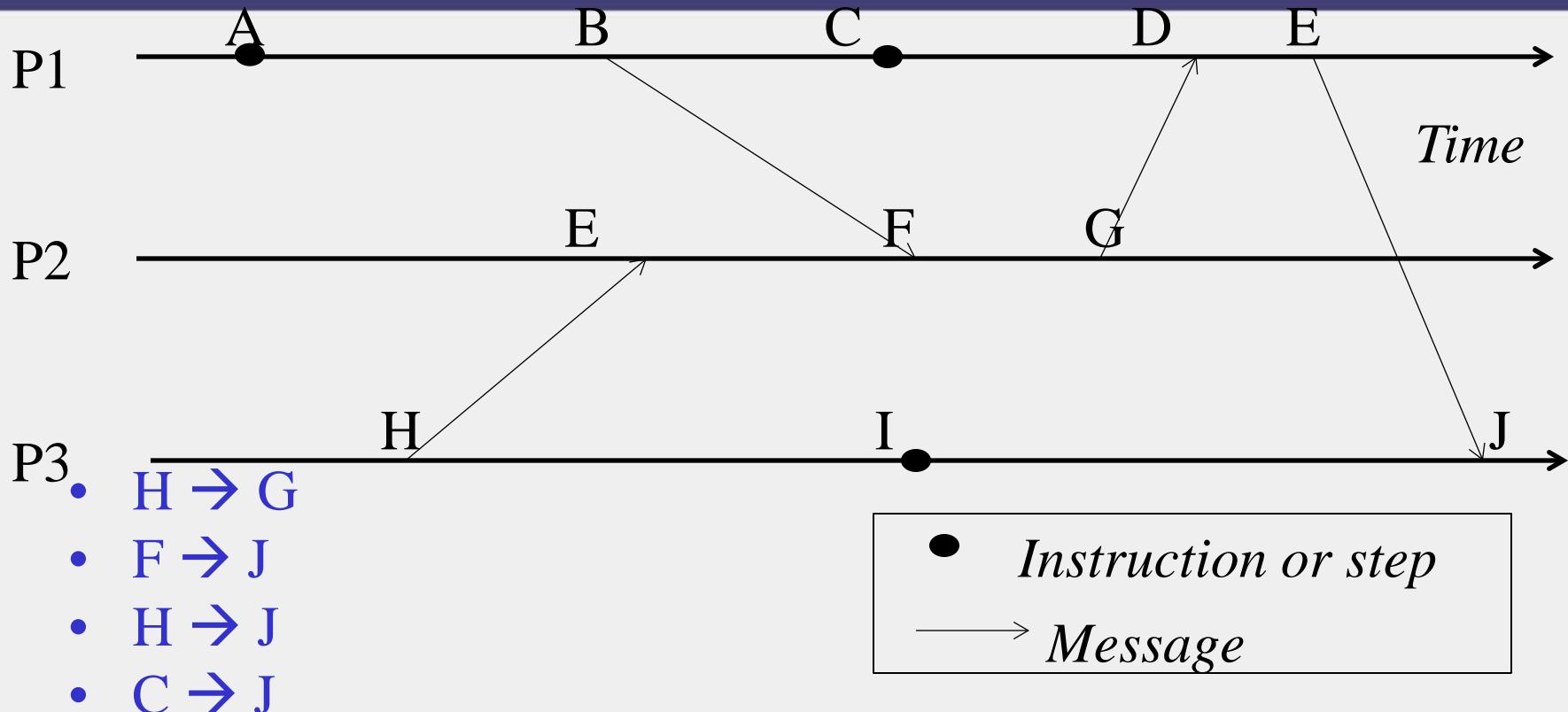
HAPPENS-BEFORE



- $A \rightarrow B$
- $B \rightarrow F$
- $A \rightarrow F$

● *Instruction or step*
→ *Message*

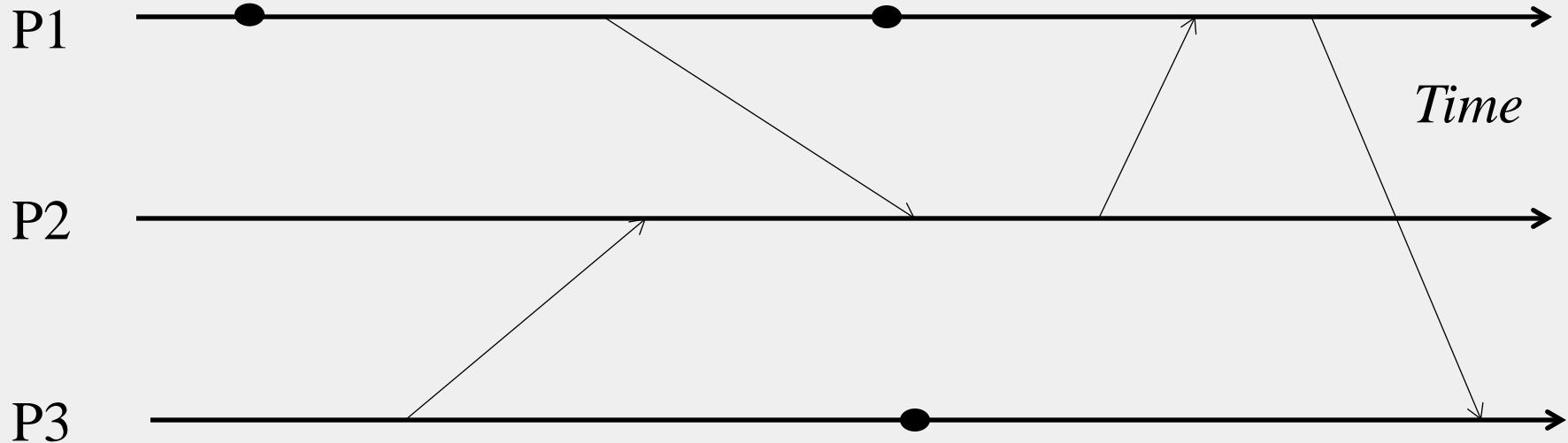
HAPPENS-BEFORE (2)



IN PRACTICE: LAMPORT TIMESTAMPS

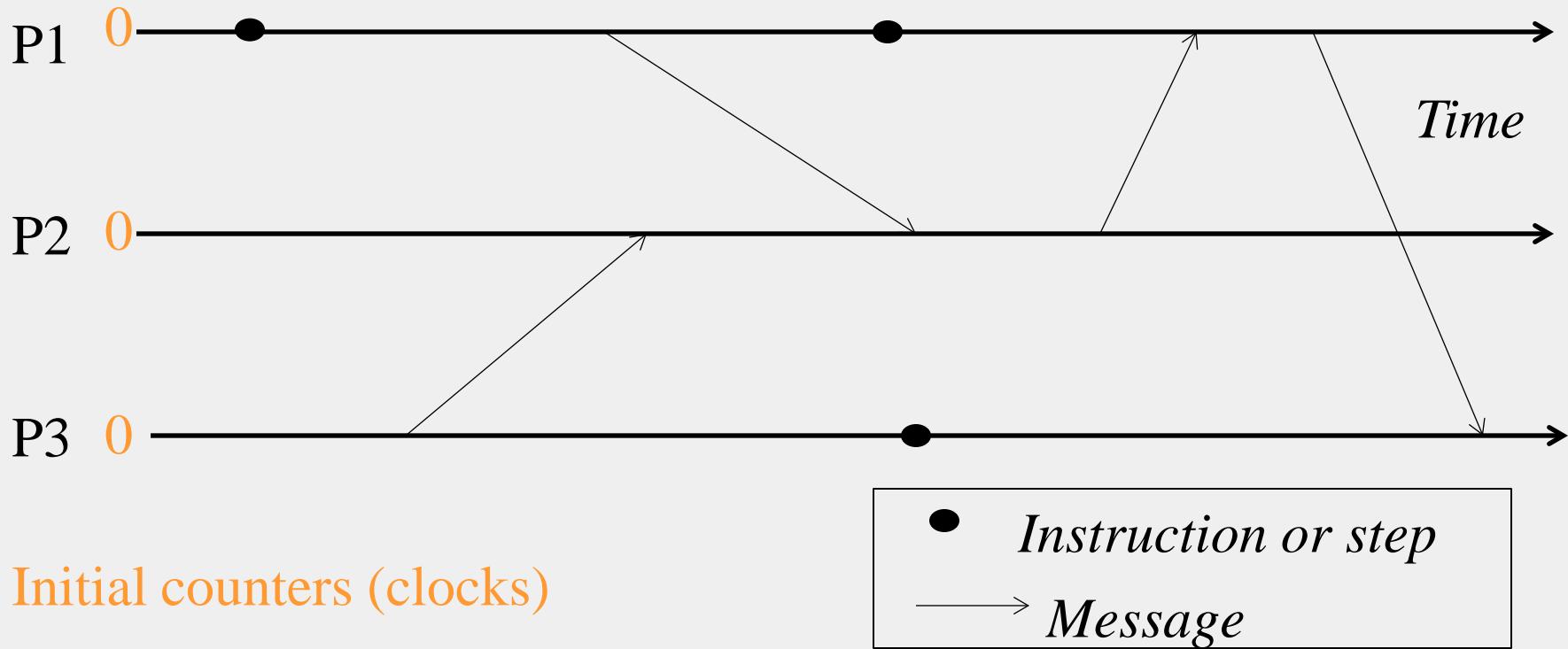
- **Goal:** Assign logical (Lamport) timestamp to each event
- **Timestamps obey causality**
- **Rules**
 - Each process uses a local counter (clock) which is an integer
 - Initial value of counter is zero
 - A process increments its counter when a **send** or an **instruction** happens at it. The counter is assigned to the event as its timestamp.
 - A **send (message)** event carries its timestamp
 - For a **receive (message)** event the counter is updated by
$$\max(\text{local clock}, \text{message timestamp}) + 1$$

EXAMPLE

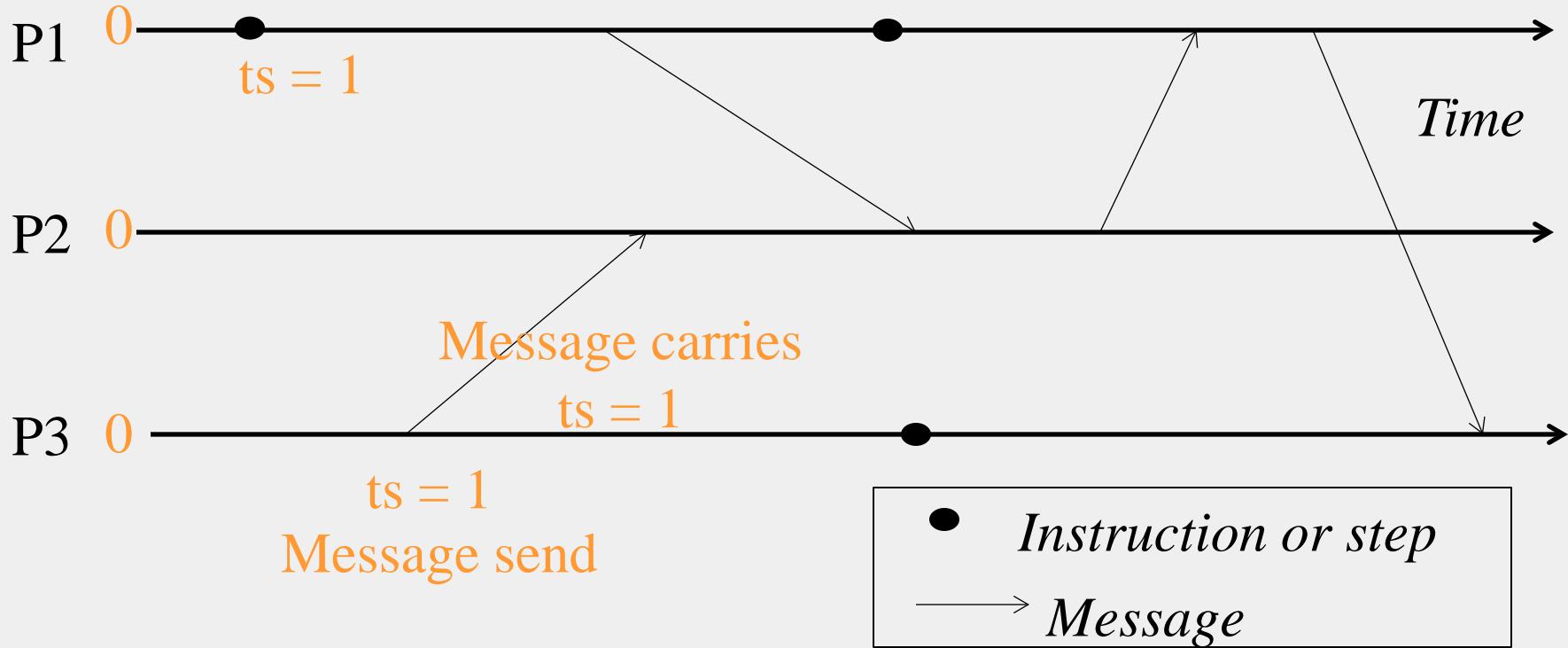


● *Instruction or step*
→ *Message*

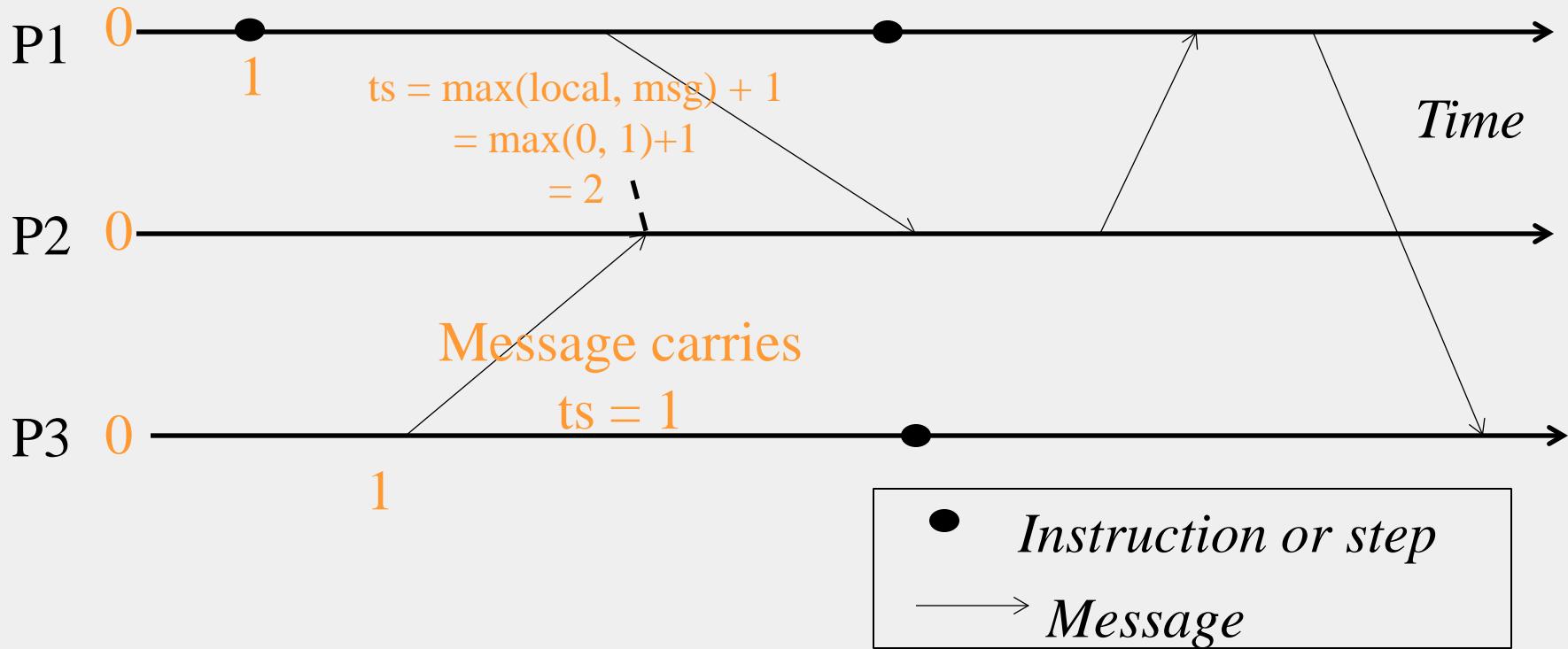
LAMPORT TIMESTAMPS



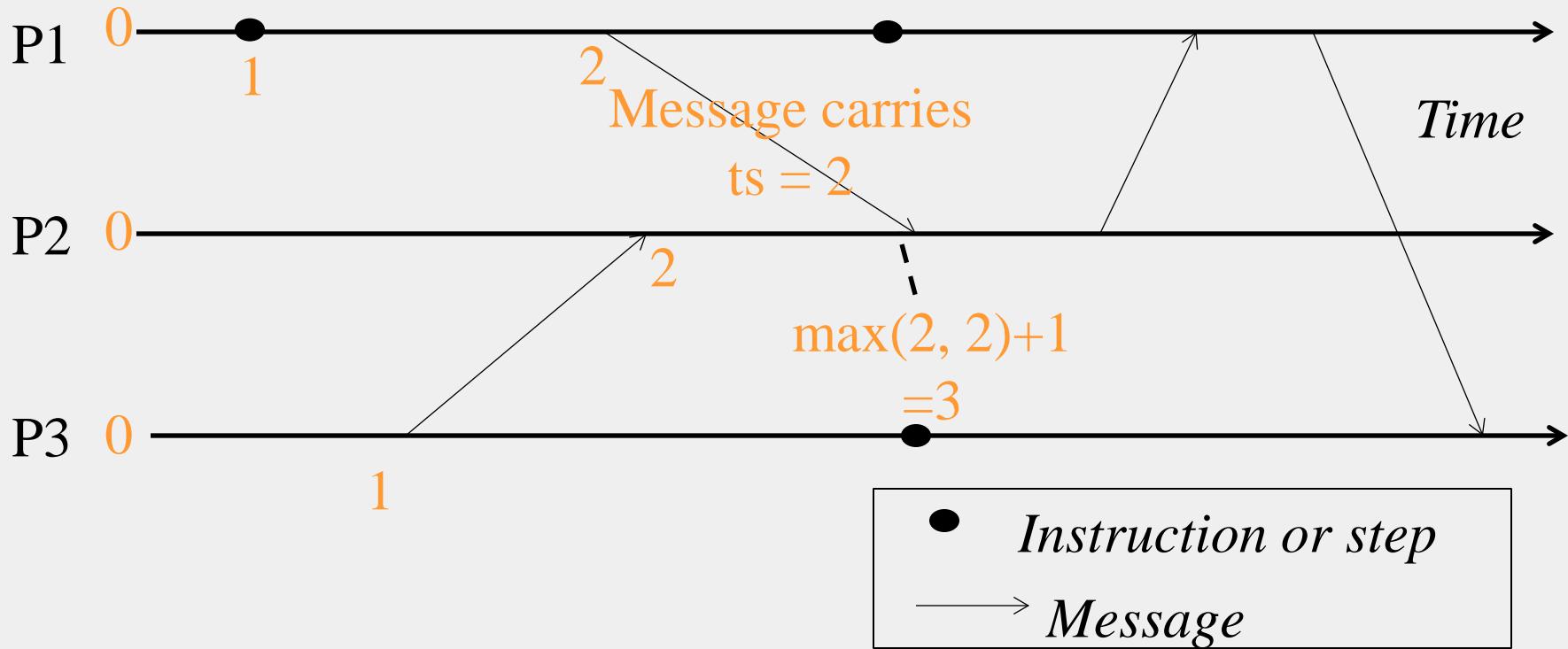
LAMPORT TIMESTAMPS



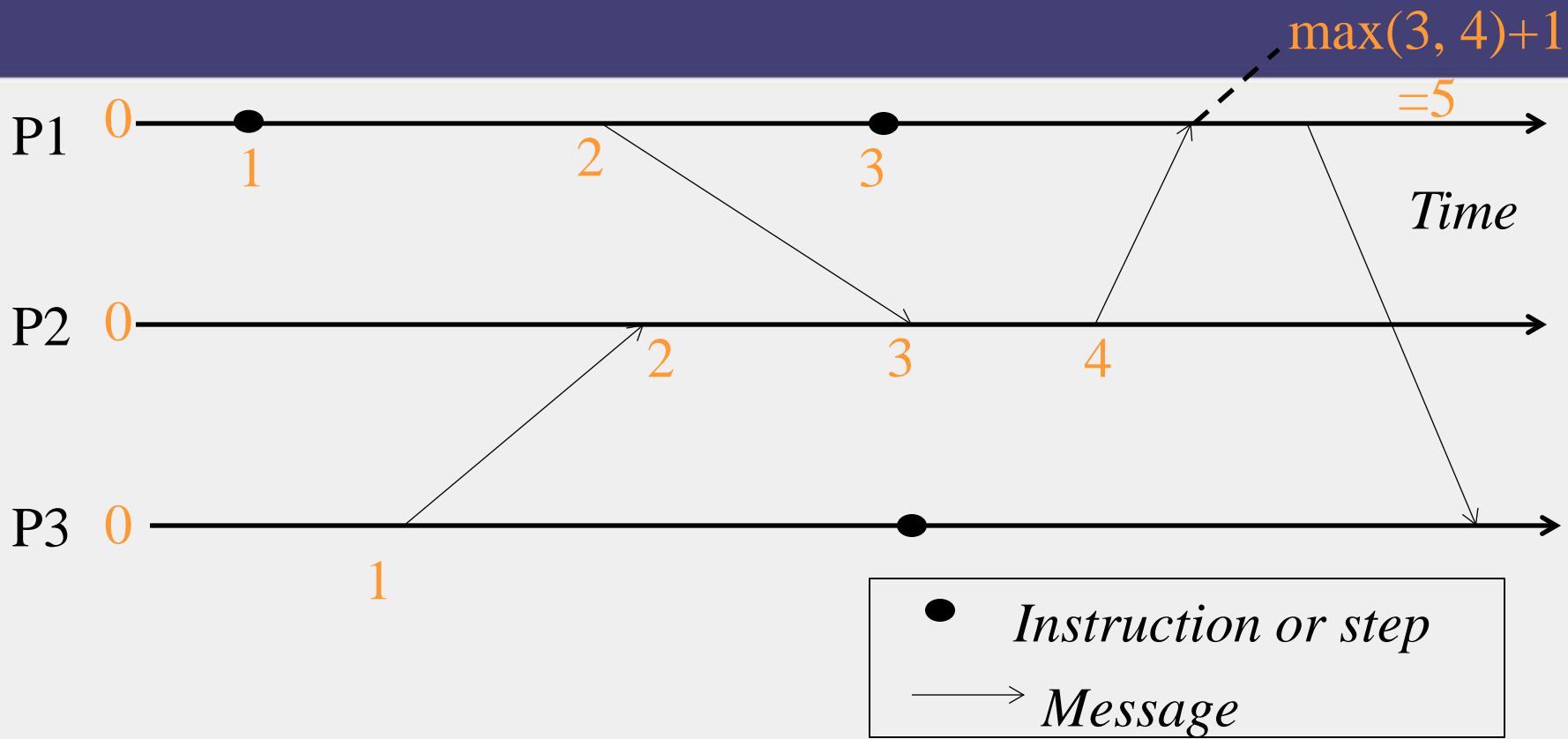
LAMPORT TIMESTAMPS



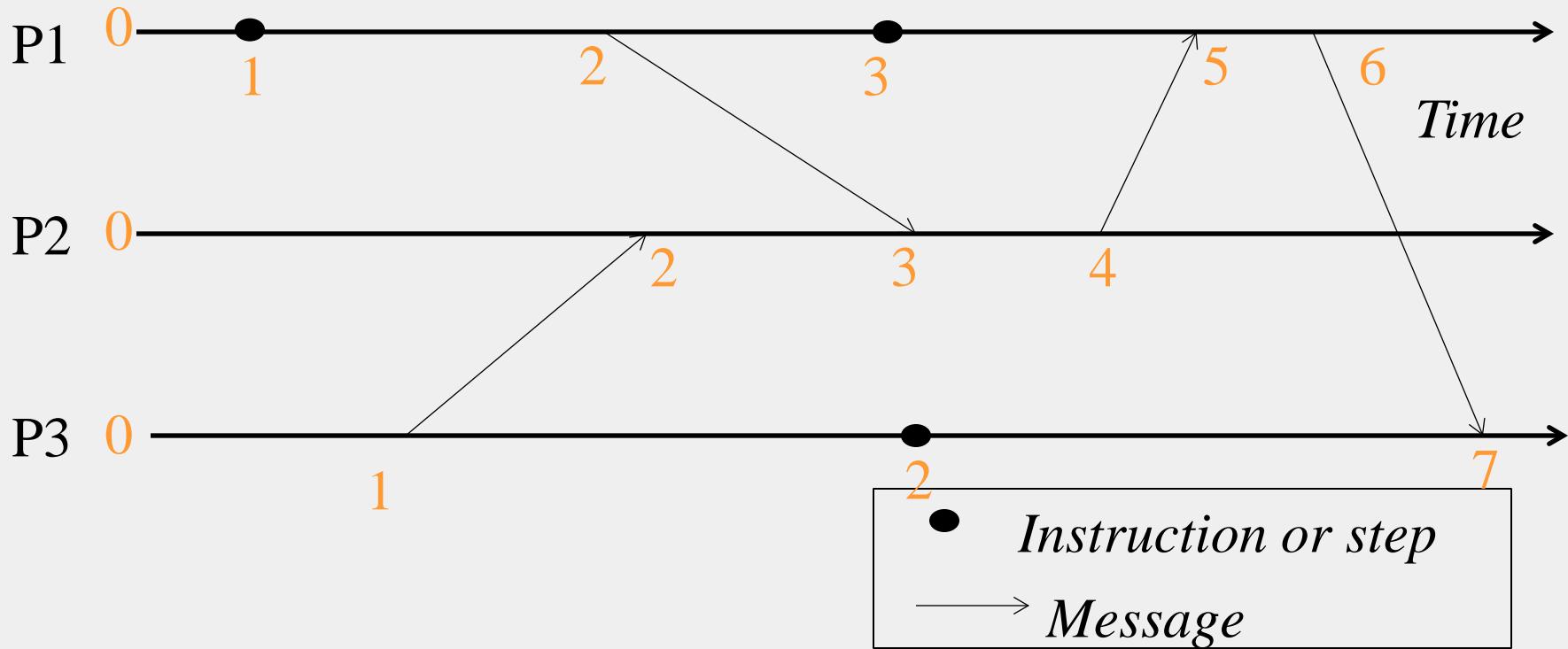
LAMPORT TIMESTAMPS



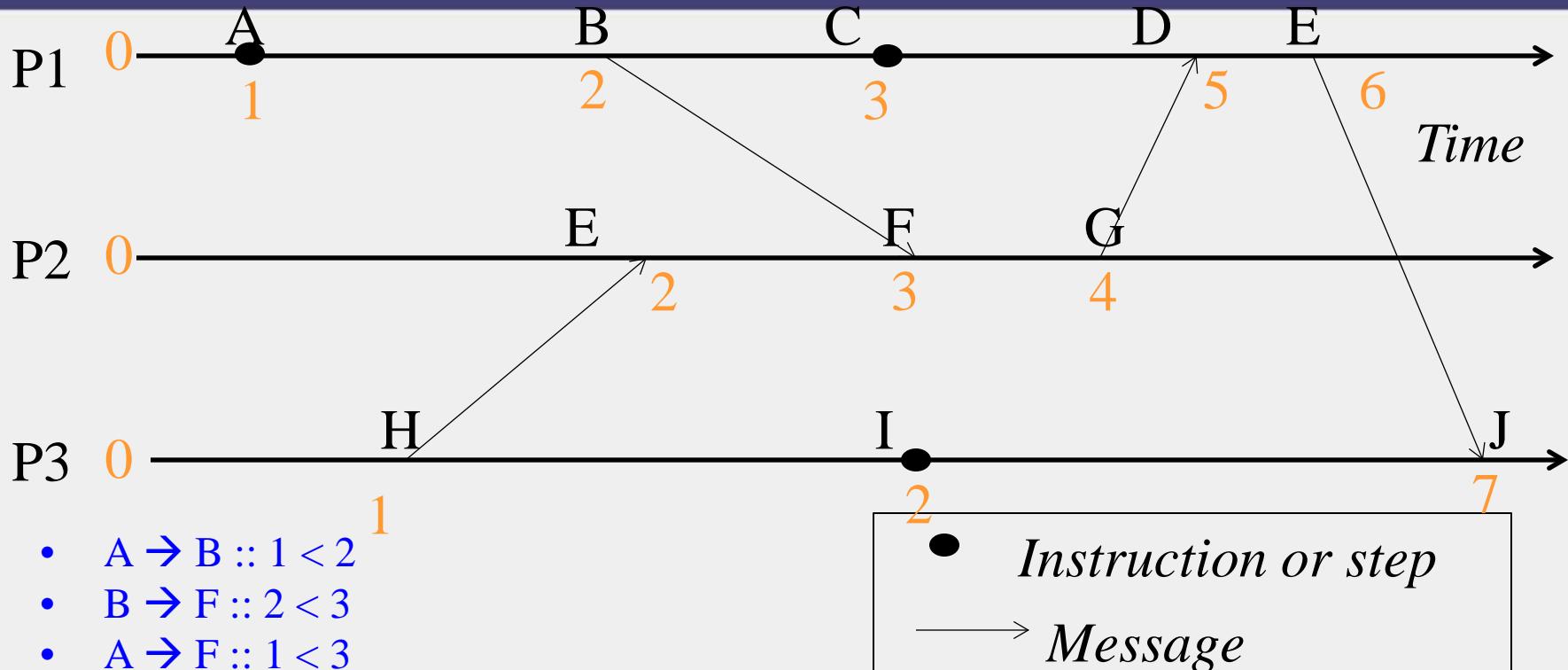
LAMPORT TIMESTAMPS



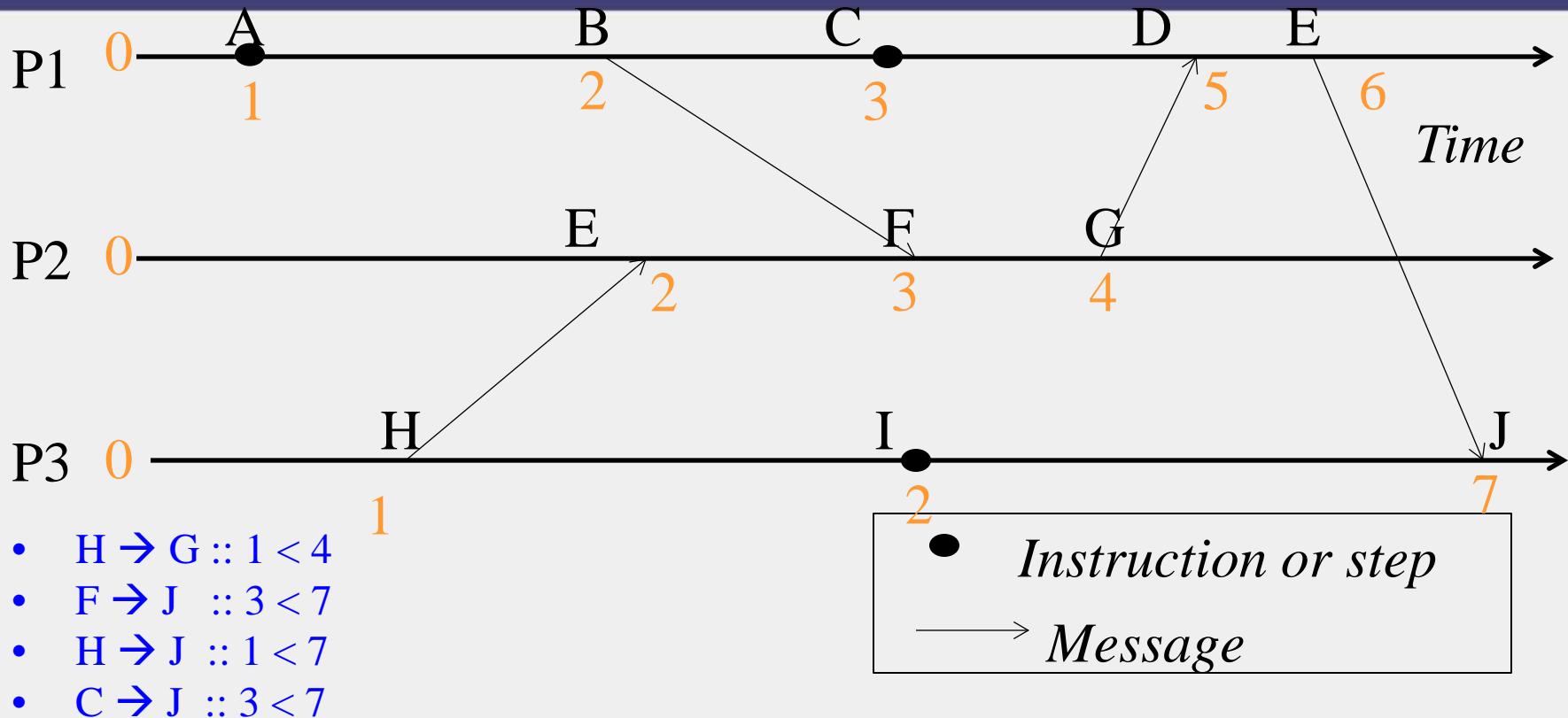
LAMPORT TIMESTAMPS



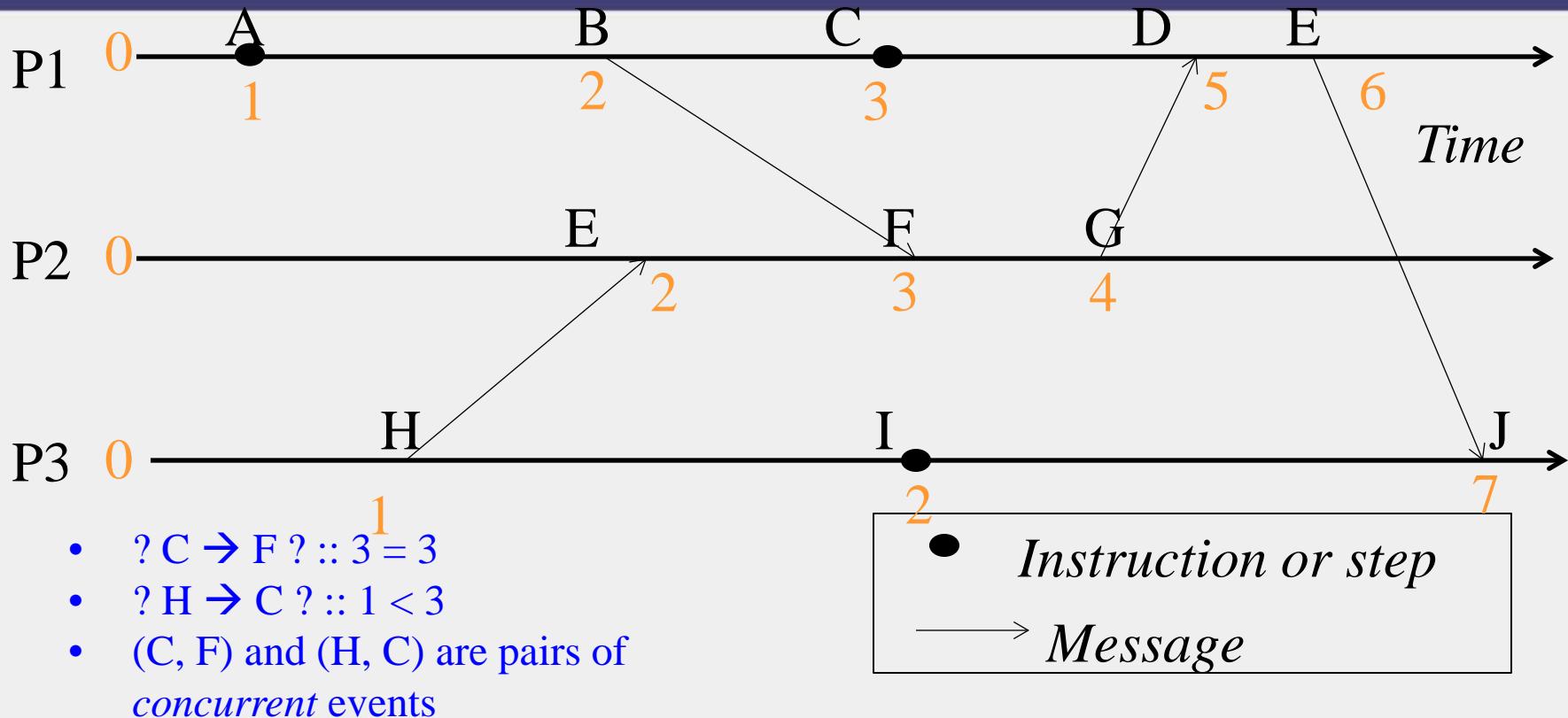
OBEYING CAUSALITY



OBEYING CAUSALITY (2)



NOT ALWAYS IMPLYING CAUSALITY



CONCURRENT EVENTS

- A pair of concurrent events doesn't have a causal path from one event to another (either way, in the pair)
- Lamport timestamps not guaranteed to be ordered or unequal for concurrent events
- Ok, since concurrent events are not causality related!
- Remember
 - E1 → E2 ⇒ timestamp(E1) < timestamp (E2), BUT
 - timestamp(E1) < timestamp (E2) ⇒
 - {E1 → E2} OR {E1 and E2 concurrent}

NEXT

- Can we have causal or logical timestamps from which we can tell if two events are concurrent or causally related?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

TIME AND ORDERING

Lecture E

VECTOR CLOCKS

VECTOR TIMESTAMPS

- Used in key-value stores like Riak
- Each process uses a vector of integer clocks
- Suppose there are N processes in the group 1...N
- Each vector has N elements
- Process i maintains vector $\mathbf{V}_i[1\dots N]$
- j th element of vector clock at process i , $V_i[j]$, is i 's knowledge of latest events at process j

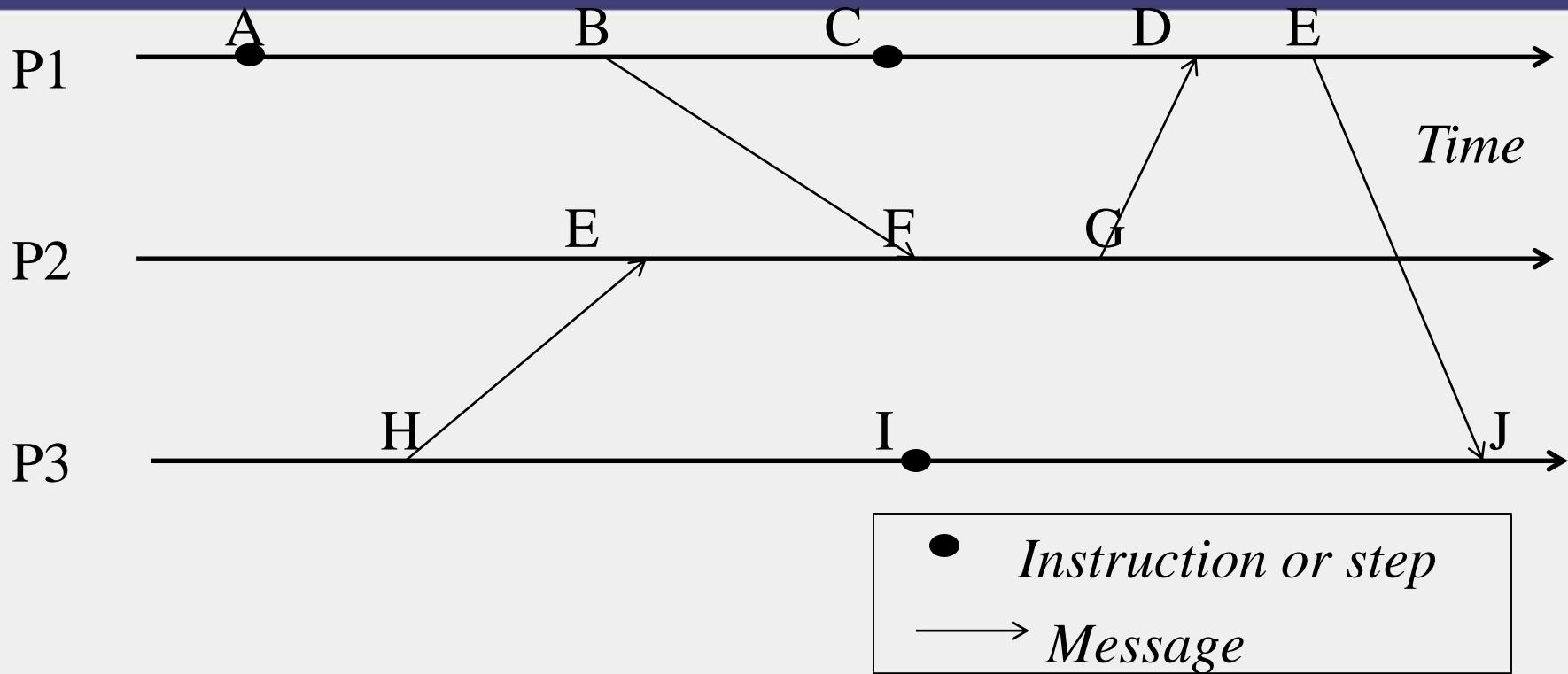
ASSIGNING VECTOR TIMESTAMPS

- Incrementing vector clocks
 1. On an instruction or send event at process i , it increments only its i th element of its vector clock
 2. Each message carries the send-event's vector timestamp $V_{\text{message}}[1 \dots N]$
 3. On receiving a message at process i :

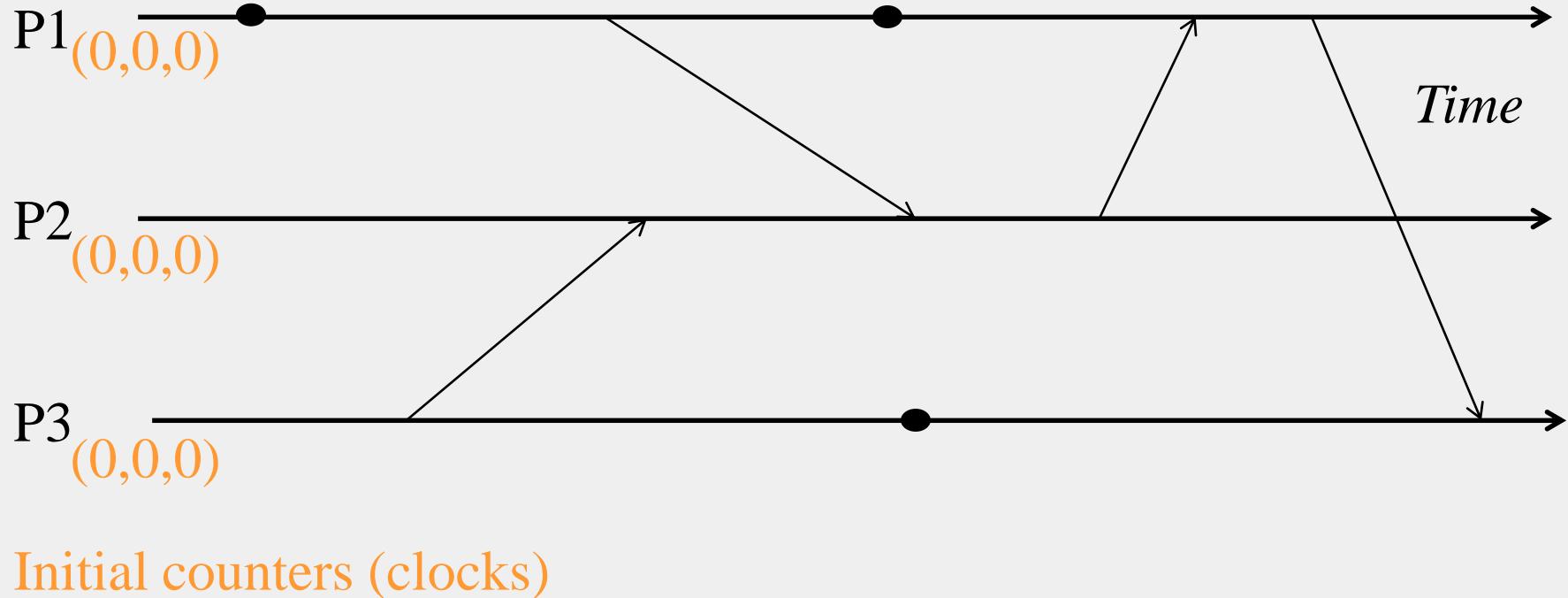
$$V_i[i] = V_i[i] + 1$$

$$V_i[j] = \max(V_{\text{message}}[j], V_i[j]) \text{ for } j \neq i$$

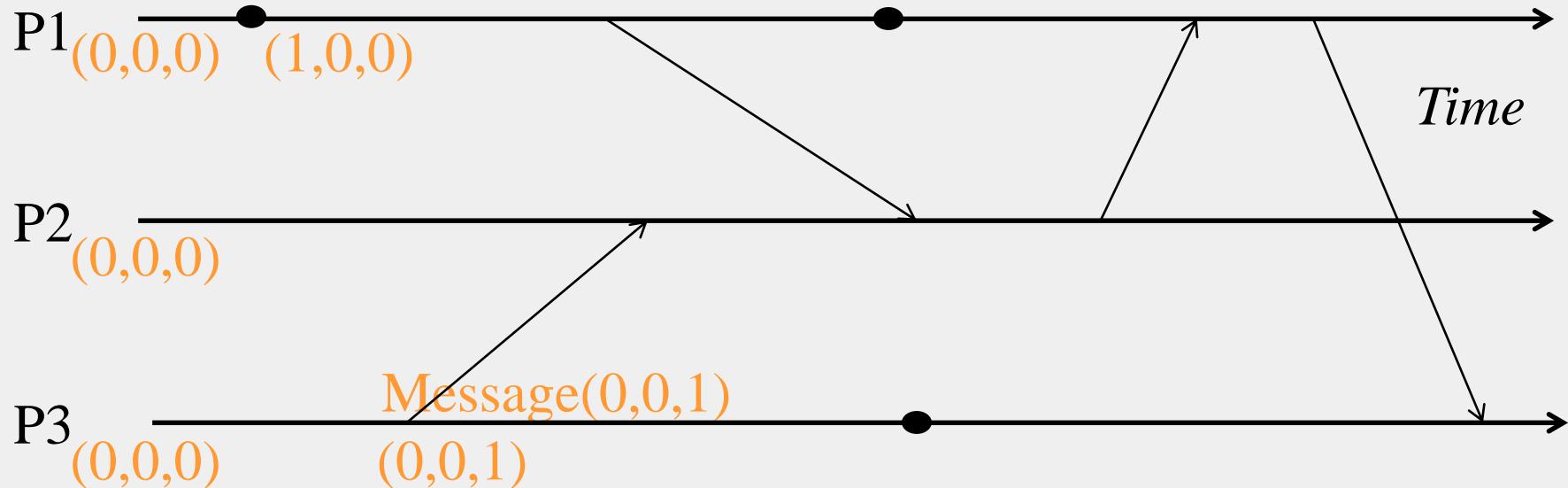
EXAMPLE



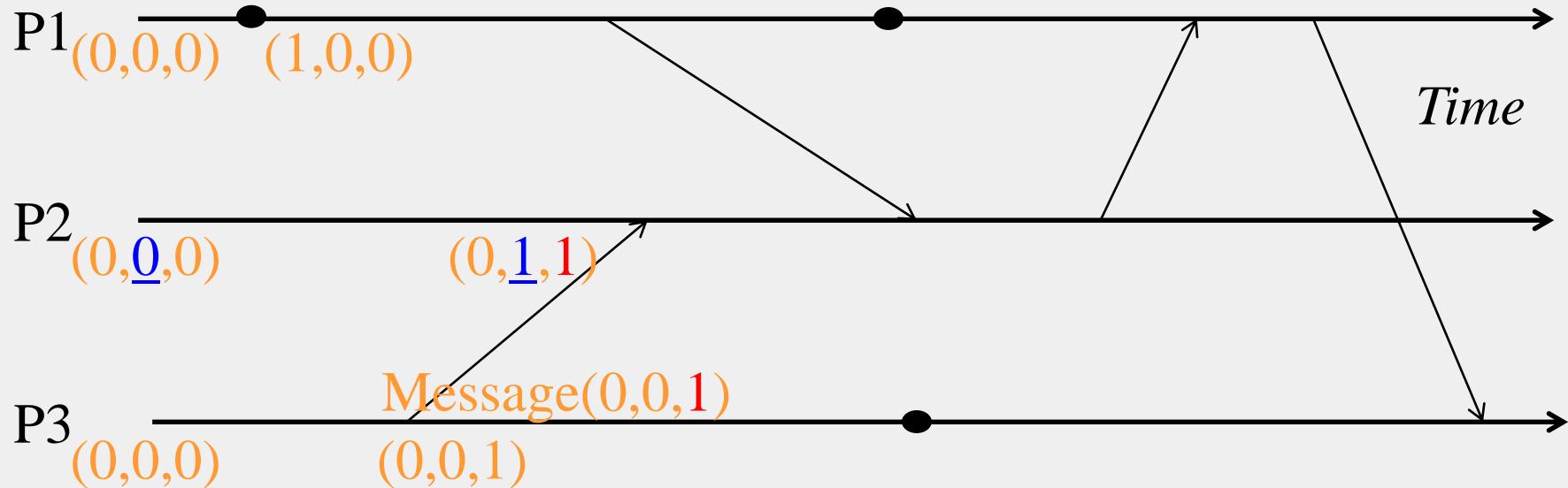
VECTOR TIMESTAMPPS



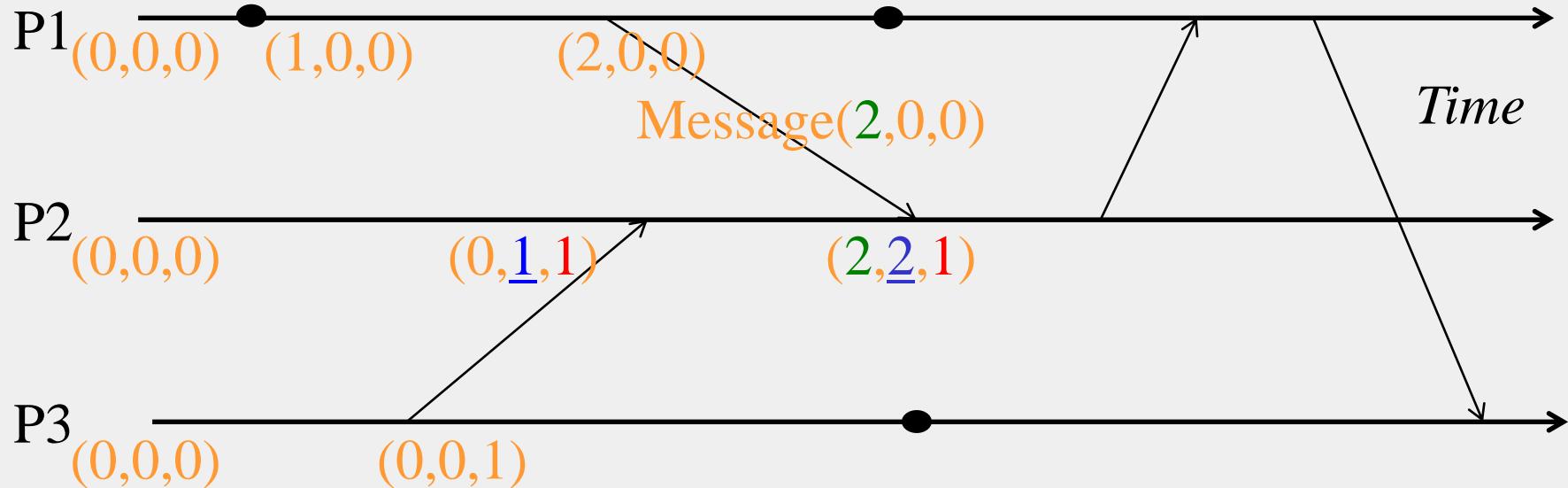
VECTOR TIMESTAMPS



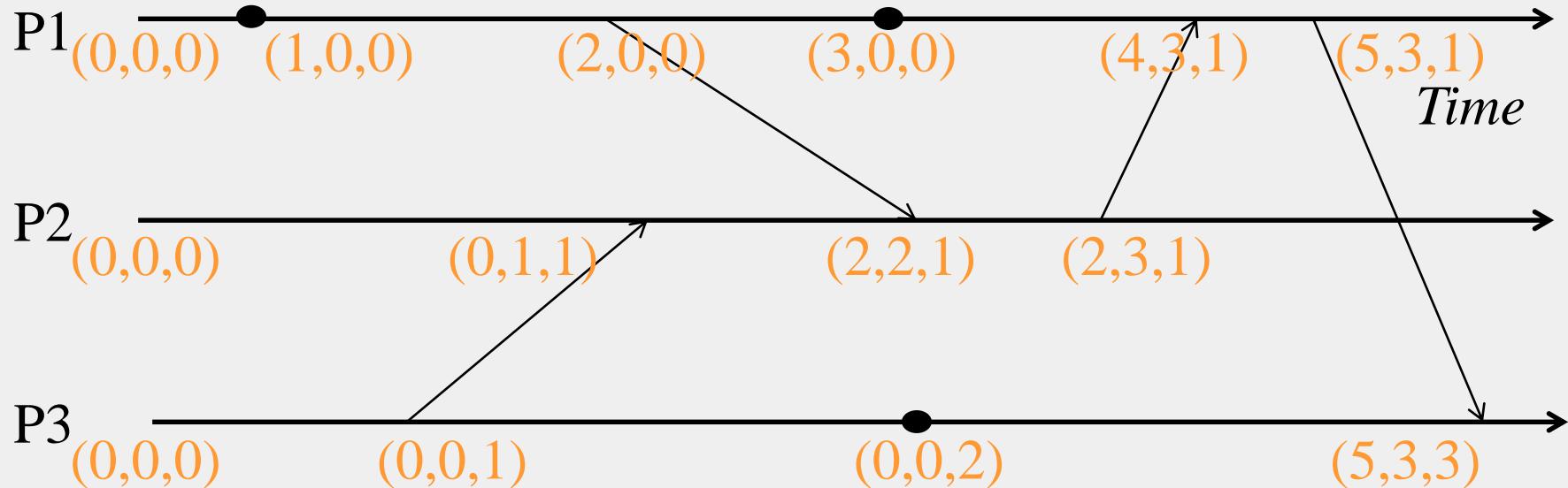
VECTOR TIMESTAMPS



VECTOR TIMESTAMPPS



VECTOR TIMESTAMPPS



CAUSALLY-RELATED ...

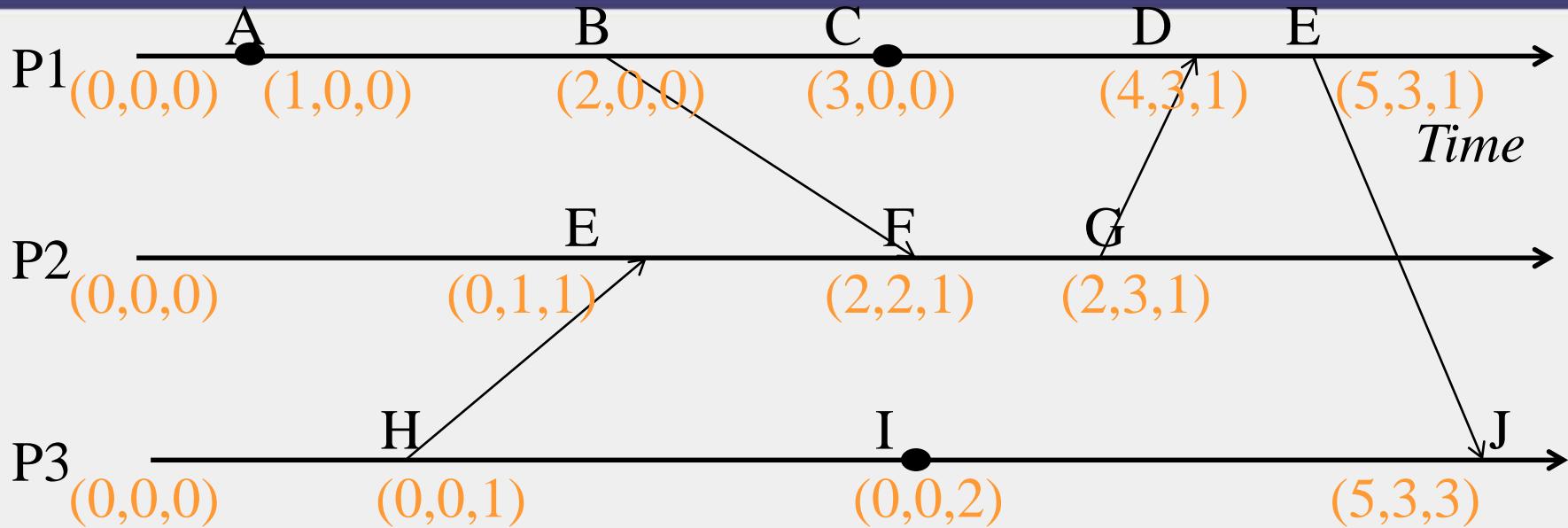
- $\text{VT}_1 = \text{VT}_2$,
iff (if and only if)
 $\text{VT}_1[i] = \text{VT}_2[i]$, for all $i = 1, \dots, N$
- $\text{VT}_1 \leq \text{VT}_2$,
iff $\text{VT}_1[i] \leq \text{VT}_2[i]$, for all $i = 1, \dots, N$
- Two events are **causally related** *iff*
 $\text{VT}_1 < \text{VT}_2$, i.e.,
iff $\text{VT}_1 \leq \text{VT}_2$ &
there exists j such that
 $1 \leq j \leq N \text{ & } \text{VT}_1[j] < \text{VT}_2[j]$

... OR NOT CAUSALLY-RELATED

- Two events VT_1 and VT_2 are **concurrent**
iff
 $\text{NOT } (VT_1 \leq VT_2) \text{ AND NOT } (VT_2 \leq VT_1)$

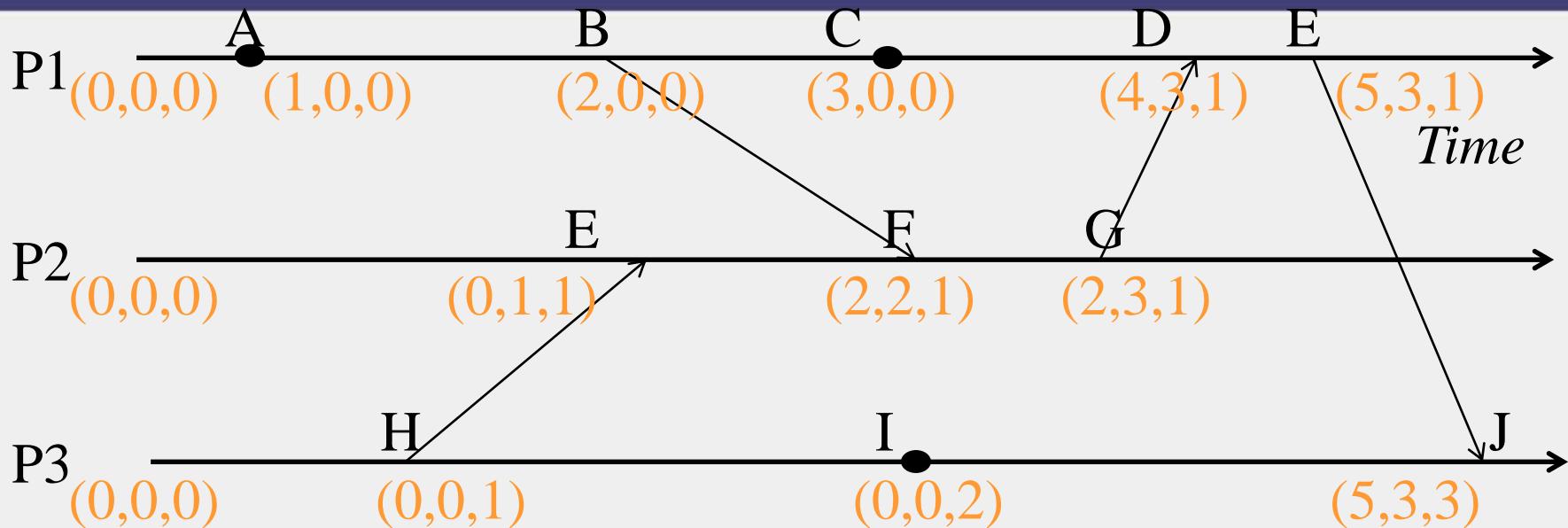
We'll denote this as $VT_2 \parallel\!\!\parallel VT_1$

OBEYING CAUSALITY



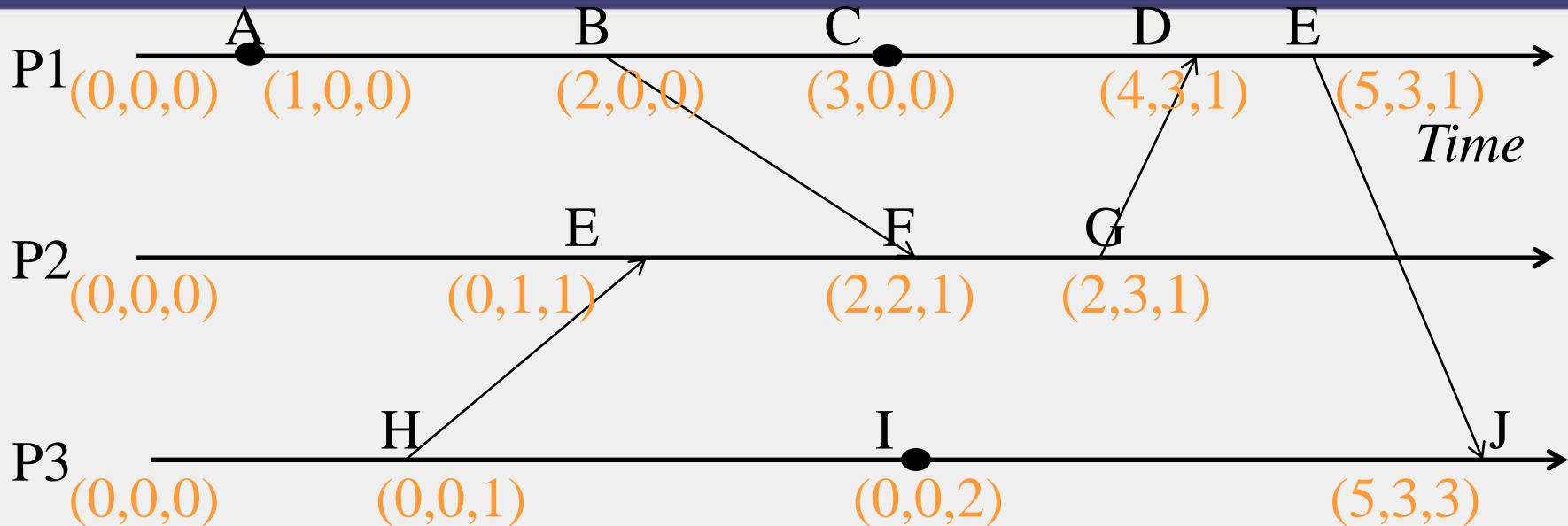
- $A \rightarrow B :: (1,0,0) < (2,0,0)$
- $B \rightarrow F :: (2,0,0) < (2,2,1)$
- $A \rightarrow F :: (1,0,0) < (2,2,1)$

OBEYING CAUSALITY (2)



- $H \rightarrow G :: (0,0,1) < (2,3,1)$
- $F \rightarrow J :: (2,2,1) < (5,3,3)$
- $H \rightarrow J :: (0,0,1) < (5,3,3)$
- $C \rightarrow J :: (3,0,0) < (5,3,3)$

IDENTIFYING CONCURRENT EVENTS



- C & F :: $(\underline{3},0,0) \parallel (2,2,1)$
- H & C :: $(0,0,\underline{1}) \parallel (\underline{3},0,0)$
- (C, F) and (H, C) are pairs of concurrent events

LOGICAL TIMESTAMPS: SUMMARY

- **Lamport timestamps**
 - Integer clocks assigned to events
 - Obey causality
 - Cannot distinguish concurrent events
- **Vector timestamps**
 - Obey causality
 - By using more space, can also identify concurrent events

TIME AND ORDERING: SUMMARY

- **Clocks are unsynchronized in an asynchronous distributed system**
- **But need to order events, across processes!**
- **Time synchronization**
 - Cristian's algorithm
 - NTP
 - Berkeley algorithm
 - But error a function of round-trip-time
- **Can avoid time sync altogether by instead assigning logical timestamps to events**



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

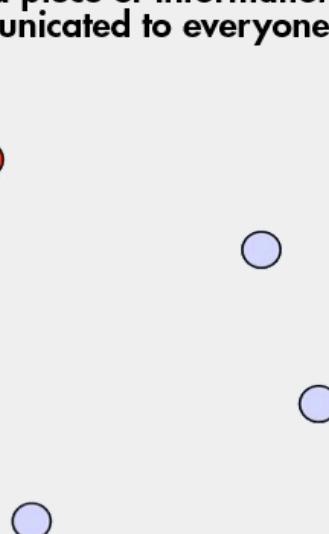
MULTICAST

Lecture A

MULTICAST ORDERING

MULTICAST PROBLEM

**Node with a piece of information
to be communicated to everyone**



**Distributed Group
of "Nodes" =
Processes at
Internet-based host**

OTHER COMMUNICATION FORMS

- **Multicast** → message sent to a group of processes
- **Broadcast** → message sent to all processes (anywhere)
- **Unicast** → message sent from one sender process to one receiver process

Who USES MULTICAST?

- A widely-used abstraction by almost all cloud systems
- Storage systems like Cassandra or a database
 - Replica servers for a key: Writes/reads to the key are multicast within the replica group
 - All servers: membership information (e.g., heartbeats) is multicast across all servers in cluster
- Online scoreboards (ESPN, French Open, FIFA World Cup)
 - Multicast to group of clients interested in the scores
- Stock exchanges
 - Group is the set of broker computers
 - Groups of computers for high-frequency trading
- Air traffic control system
 - All controllers need to receive the same updates in the same order

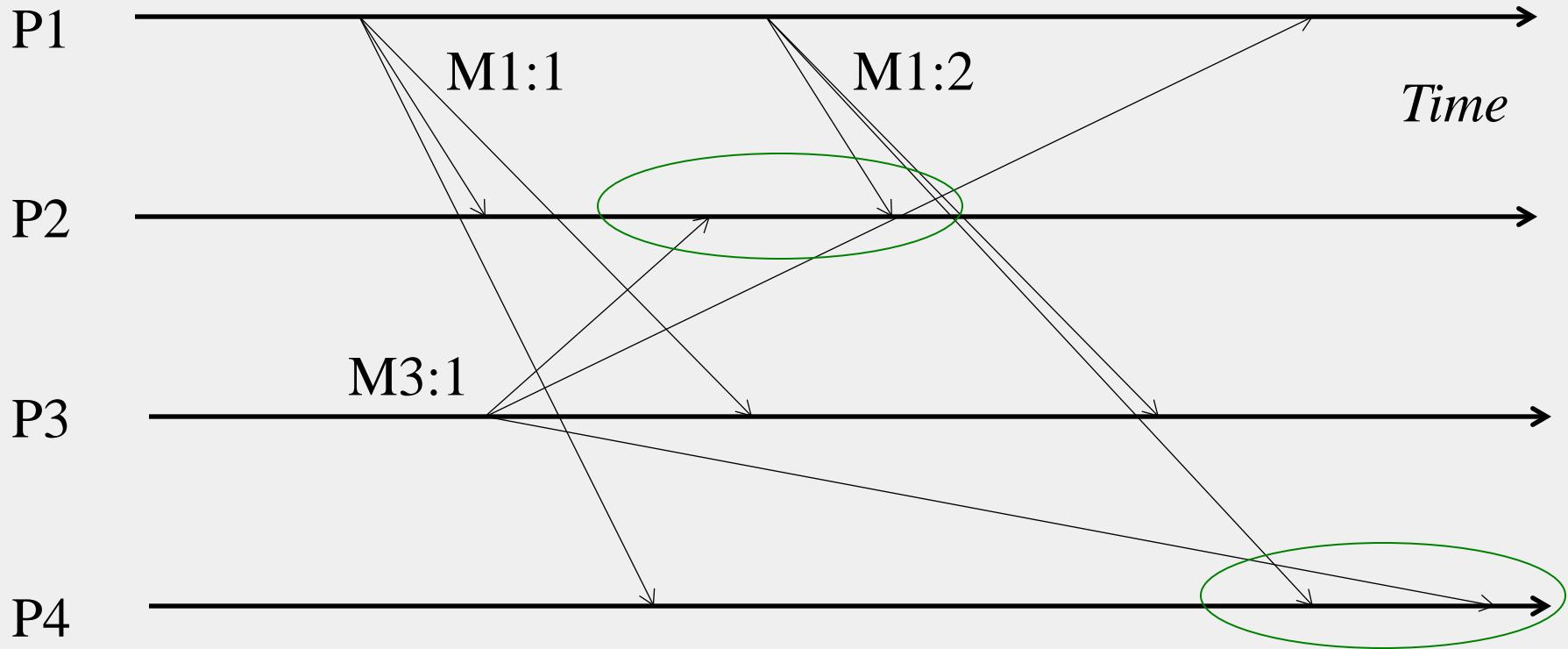
MULTICAST ORDERING

- Determines the meaning of “same order” of multicast delivery at different processes in the group
- Three popular flavors implemented by several multicast protocols
 1. FIFO ordering
 2. Causal ordering
 3. Total ordering

1. FIFO ORDERING

- Multicasts from each sender are received in the order they are sent, at all receivers
- Don't worry about multicasts from different senders
- More formally
 - *If a correct process issues (sends) $\text{multicast}(g,m)$ to group g and then $\text{multicast}(g,m')$, then every correct process that delivers m' would already have delivered m .*

FIFO Ordering: Example



M1:1 and M1:2 should be received in that order at each receiver

Order of delivery of M3:1 and M1:2 could be different at different receivers

2. CAUSAL ORDERING

- Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers
- Formally
 - *If $\text{multicast}(g,m) \rightarrow \text{multicast}(g,m')$ then any correct process that delivers m' would already have delivered m .*
 - (\rightarrow is Lamport's happens-before)

Causal Ordering: Example



$M3:1 \rightarrow M3:2$, and so should be received in that order at each receiver

$M1:1 \rightarrow M3:1$, and so should be received in that order at each receiver

M3:1 and M2:1 are concurrent and thus ok to be received in different orders at different receivers

CAUSAL VS. FIFO

- Causal Ordering => FIFO Ordering
- Why?
 - If two multicasts M and M' are sent by the same process P , and M was sent before M' , then $M \rightarrow M'$
 - Then a multicast protocol that implements causal ordering will obey FIFO ordering since $M \rightarrow M'$
- Reverse is not true! FIFO ordering does not imply causal ordering.

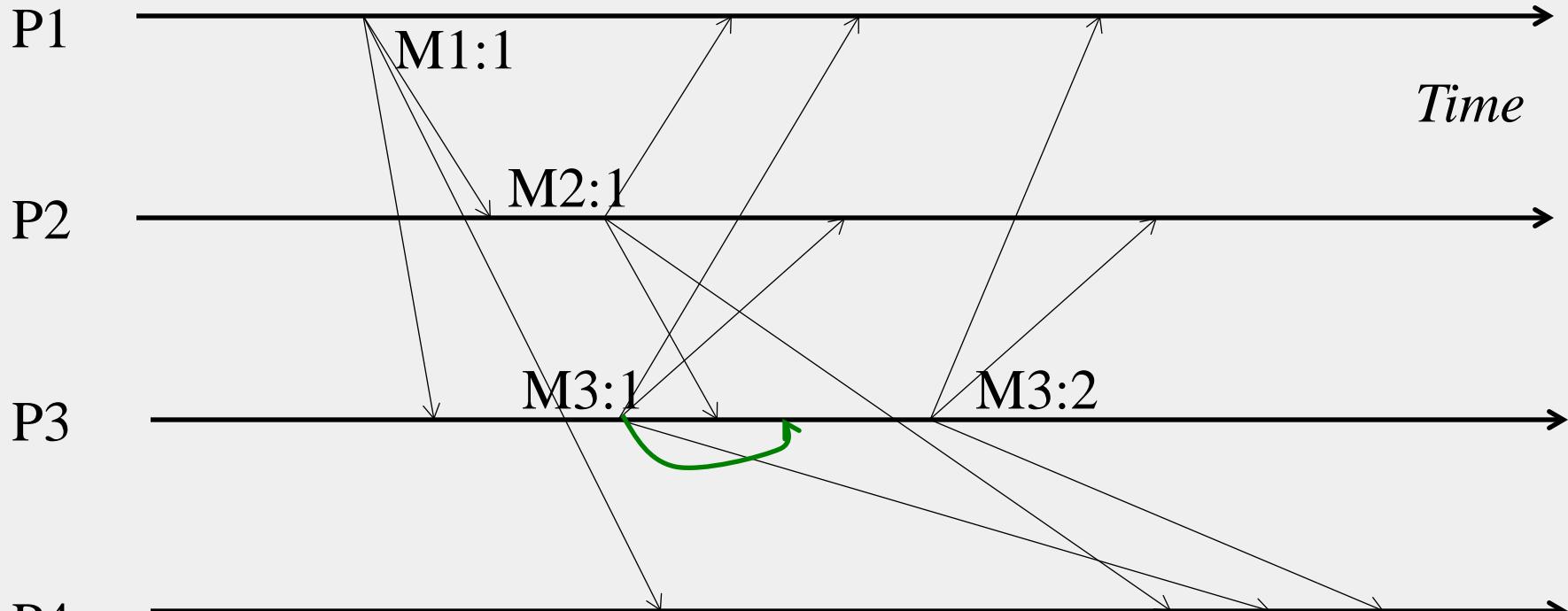
WHY CAUSAL AT ALL?

- Group = set of your friends on a social network
- A friend sees your message m , and she posts a response (comment) m' to it
 - If friends receive m' before m , it wouldn't make sense
 - But if two friends post messages m'' and n'' concurrently, then they can be seen in any order at receivers
- A variety of systems implement causal ordering: Social networks, bulletin boards, comments on websites, etc.

3. TOTAL ORDERING

- Also known as “Atomic Broadcast”
- Unlike FIFO and causal, this does not pay attention to order of multicast sending
- Ensures all receivers receive all multicasts in the same order
- Formally
 - *If a correct process P delivers message m before m' (independent of the senders), then any other correct process P' that delivers m' would already have delivered m .*

Total Ordering: Example



The order of receipt of multicasts is the same at all processes.

M1:1, then M2:1, then M3:1, then M3:2

May need to delay delivery of some messages

HYBRID VARIANTS

- Since FIFO/Causal are orthogonal to Total, can have hybrid ordering protocols too
 - FIFO-total hybrid protocol satisfies both FIFO and total orders
 - Causal-total hybrid protocol satisfies both Causal and total orders

IMPLEMENTATION?

- That was *what* ordering is
- But *how* do we implement each of these orderings?
- Next lecture



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MULTICAST

Lecture B

IMPLEMENTING MULTICAST
ORDERING 1

MULTICAST ORDERING

- How do we implement each of the ordering schemes we've seen
 1. FIFO ordering (this lecture)
 2. Causal ordering (next lecture)
 3. Total ordering (this lecture)

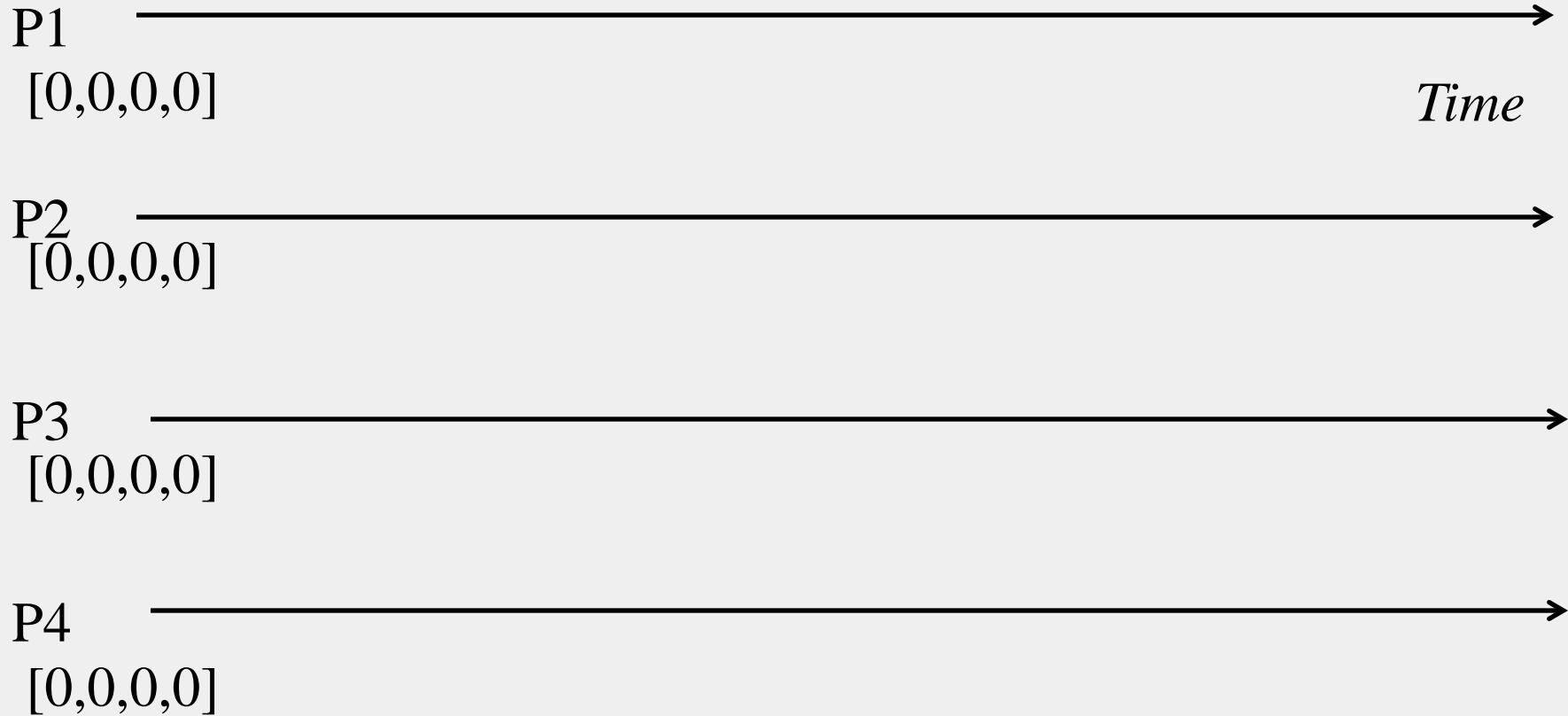
FIFO MULTICAST: DATA STRUCTURES

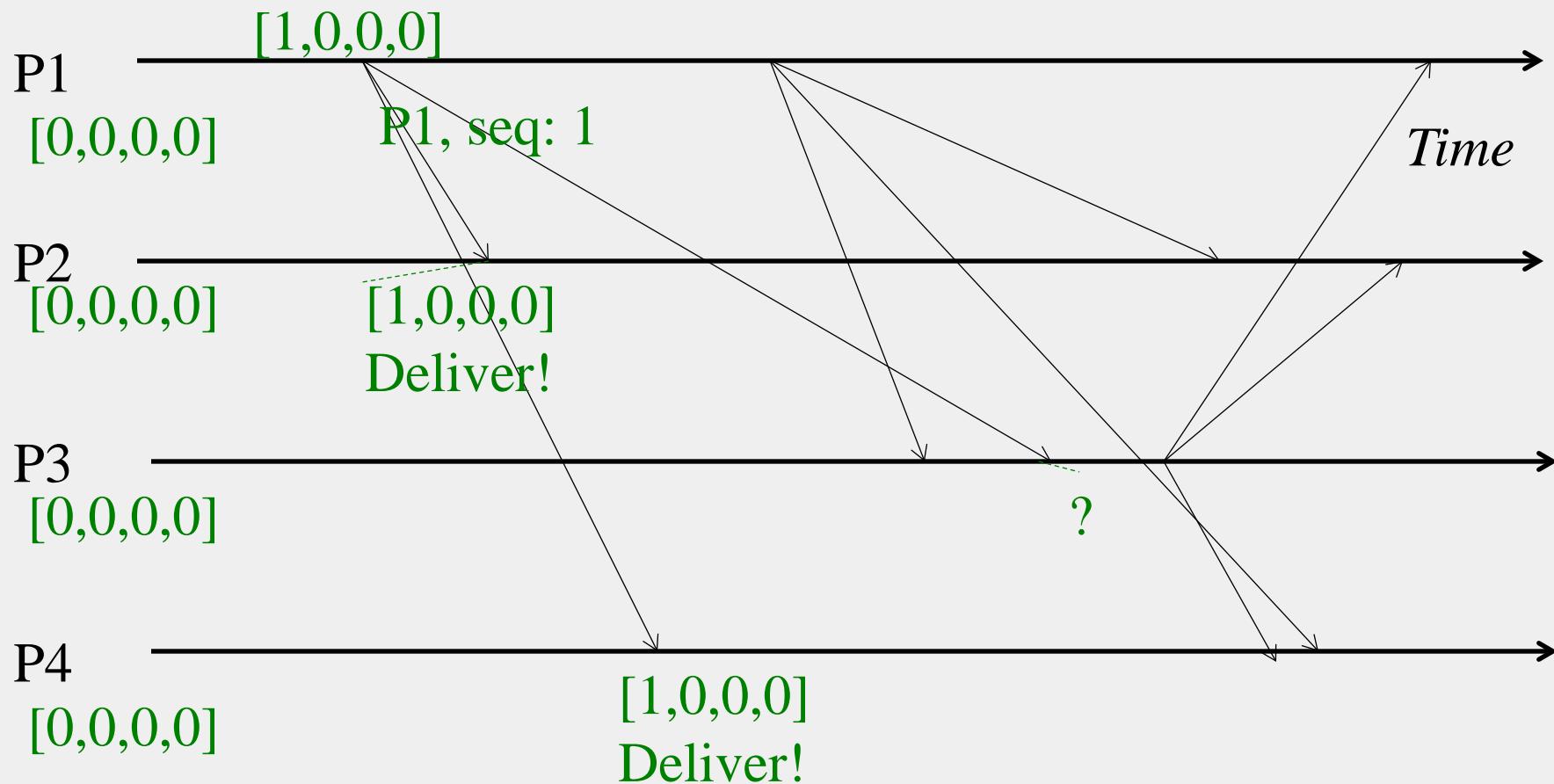
- Each receiver maintains a per-sender sequence number (integers)
 - Processes P_1 through P_N
 - P_i maintains a vector of sequence numbers $P_i[1 \dots N]$ (initially all zeroes)
 - $P_i[j]$ is the latest sequence number P_i has received from P_j

FIFO MULTICAST: UPDATING RULES

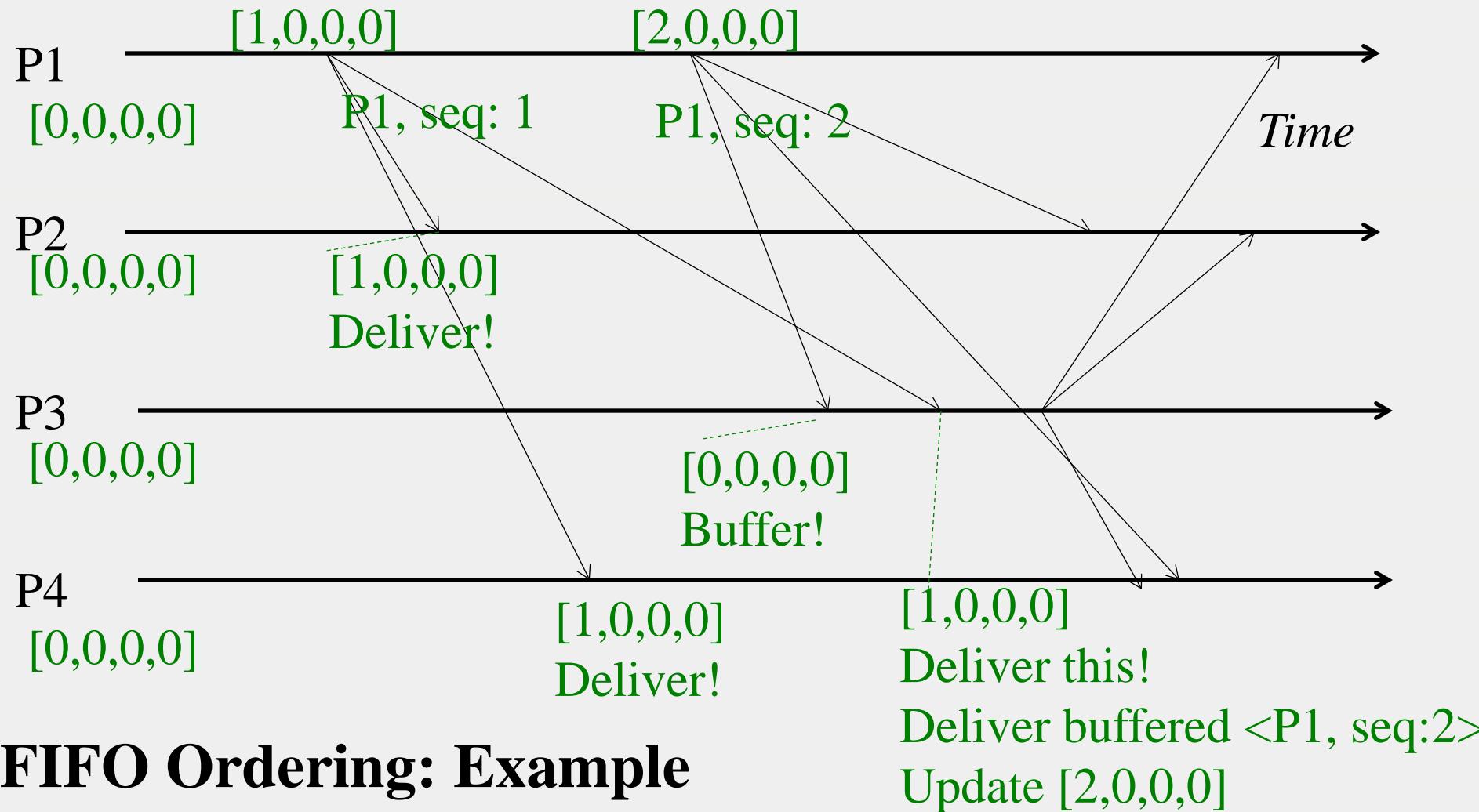
- Send multicast at process P_j :
 - Set $P_j[j] = P_j[j] + 1$
 - Include new $P_j[j]$ in multicast message as its sequence number
- Receive multicast: If P_i receives a multicast from P_j with sequence number S in message
 - if $(S == P_i[j] + 1)$ then
 - deliver message to application
 - Set $P_i[j] = P_i[j] + 1$
 - else buffer this multicast until above condition is true

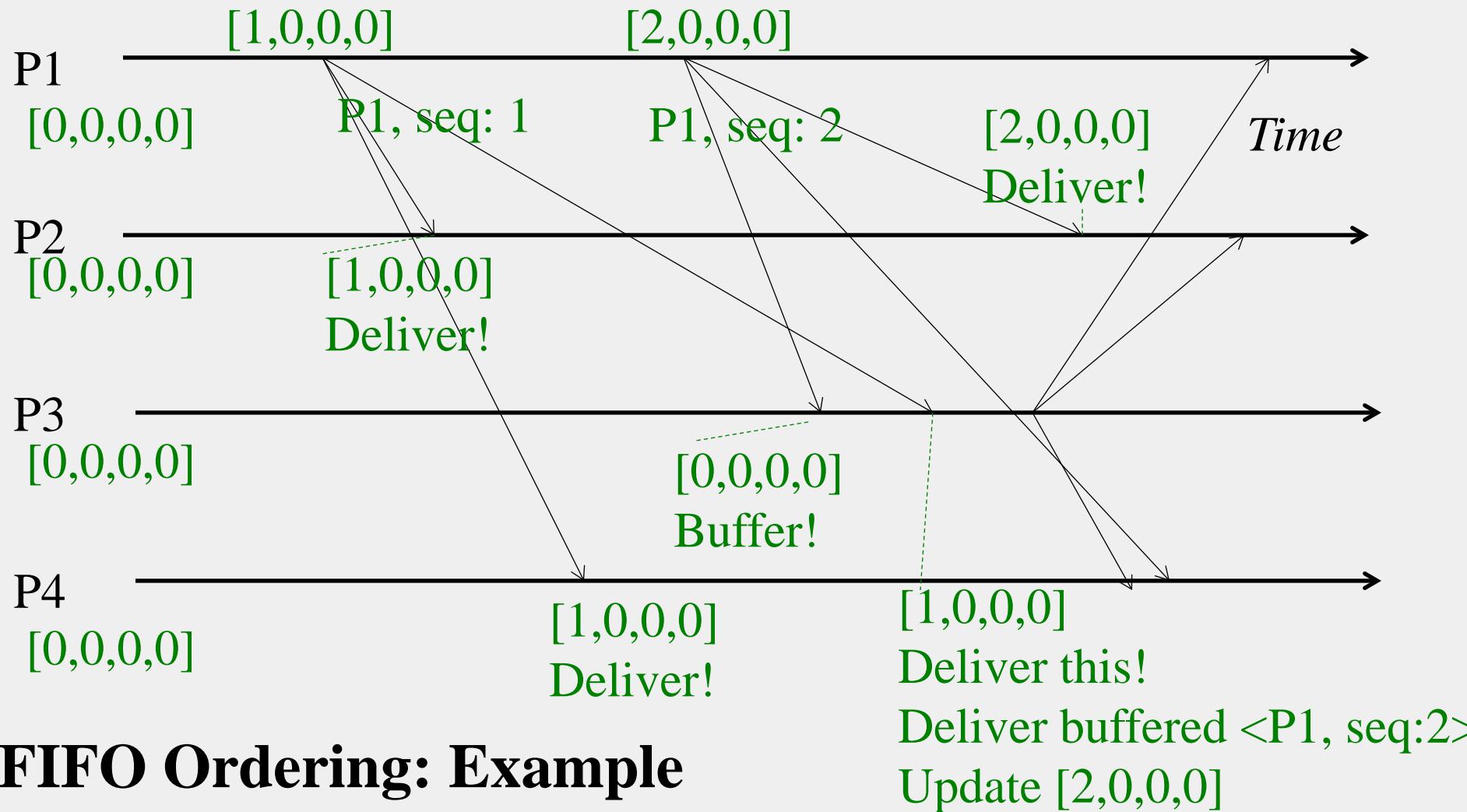
FIFO Ordering: Example

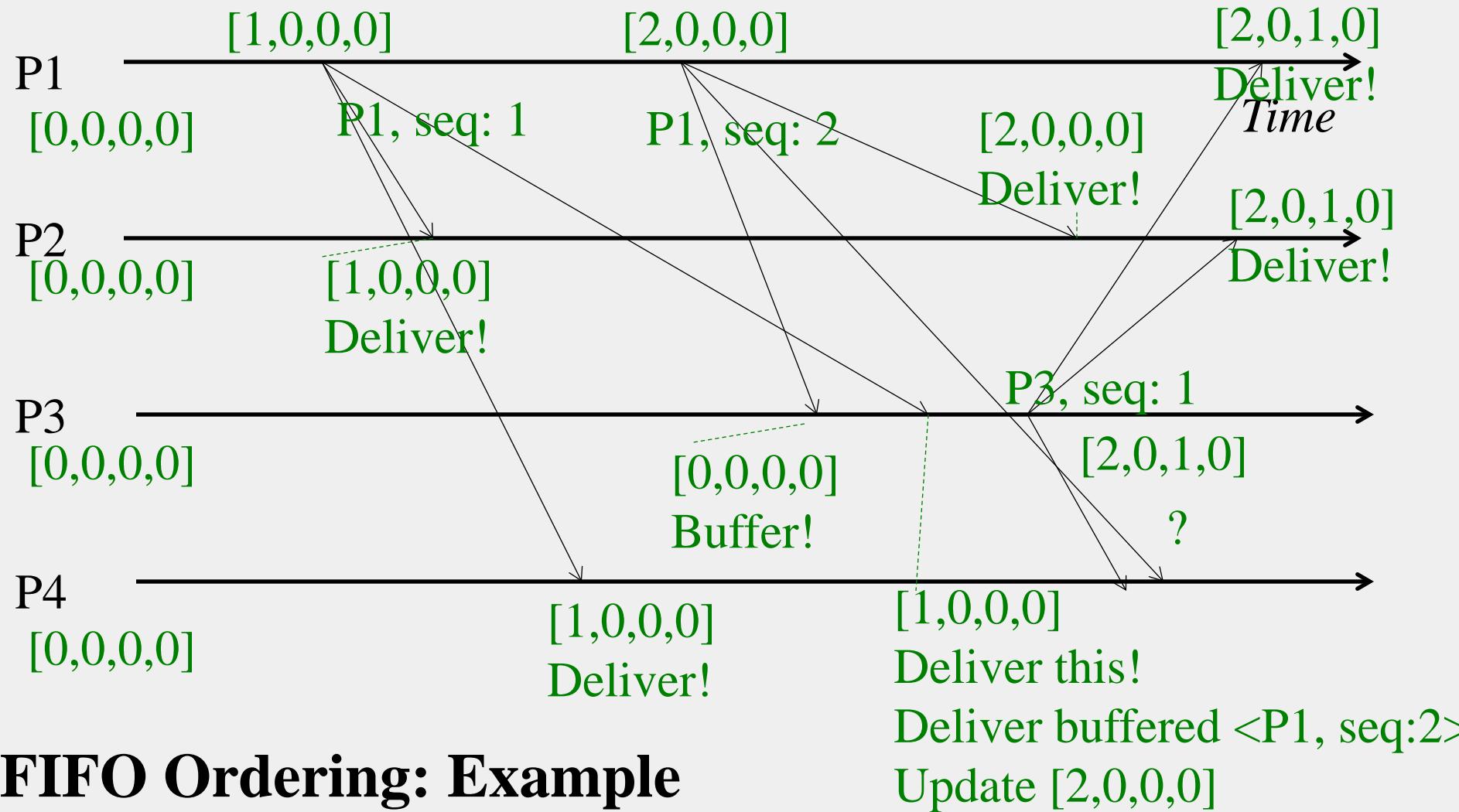


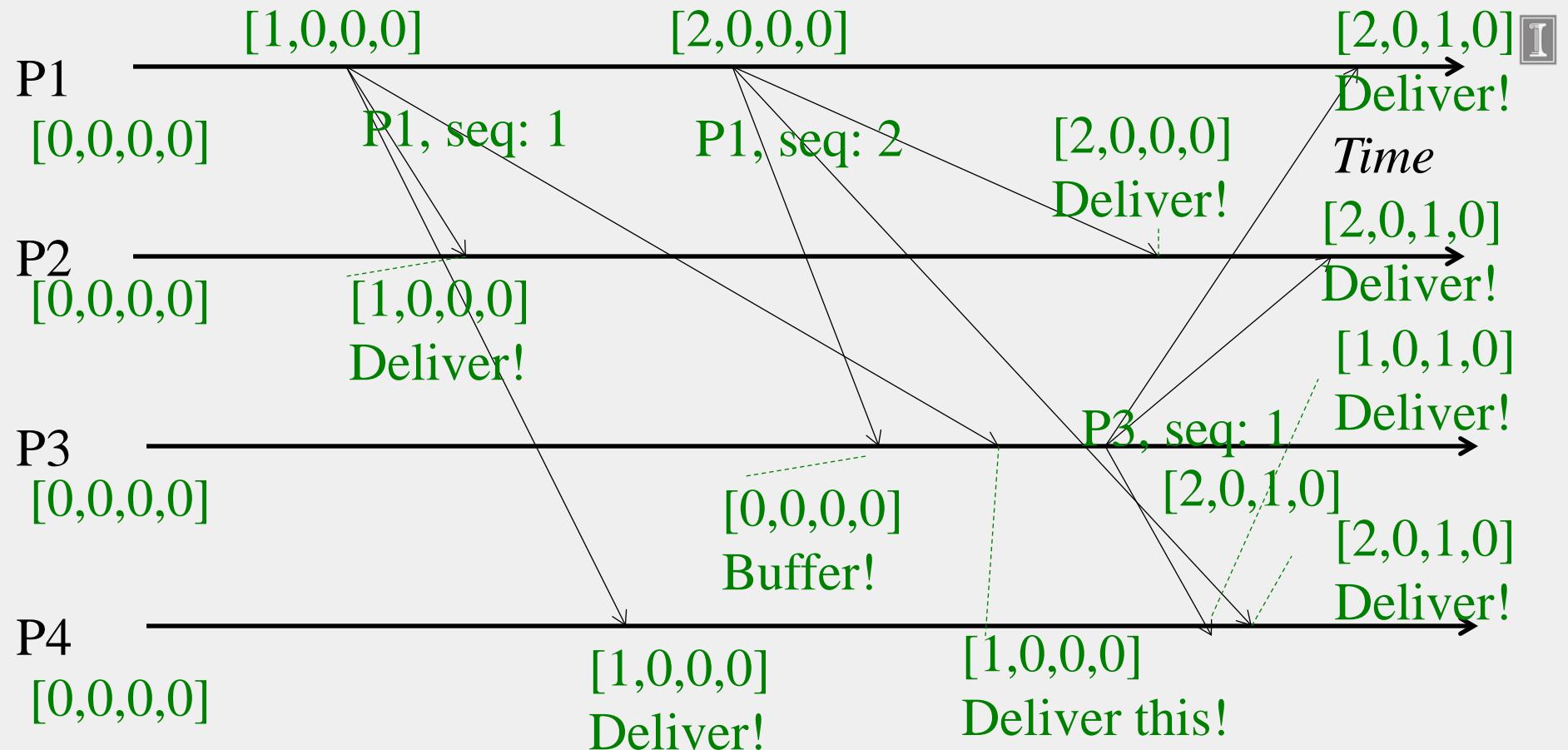


FIFO Ordering: Example









FIFO Ordering: Example

TOTAL ORDERING

- Ensures all receivers receive all multicasts in the same order
- Formally
 - *If a correct process P delivers message m before m' (independent of the senders), then any other correct process P' that delivers m' would already have delivered m .*

SEQUENCER-BASED APPROACH

- Special process elected as leader or sequencer
- Send multicast at process P_i :
 - Send multicast message M to group and sequencer
- Sequencer:
 - Maintains a global sequence number S (initially 0)
 - When it receives a multicast message M , it sets $S = S + 1$, and multicasts $\langle M, S \rangle$
- Receive multicast at process P_i :
 - P_i maintains a local received global sequence number S_i (initially 0)
 - If P_i receives a multicast M from P_j , it buffers it until it both
 1. P_i receives $\langle M, S(M) \rangle$ from sequencer, and
 2. $S_i + 1 = S(M)$ - Then deliver it message to application and set $S_i = S_i + 1$

NEXT

- Implementing causal ordering



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MULTICAST

Lecture C

IMPLEMENTING MULTICAST
ORDERING 2

CAUSAL ORDERING

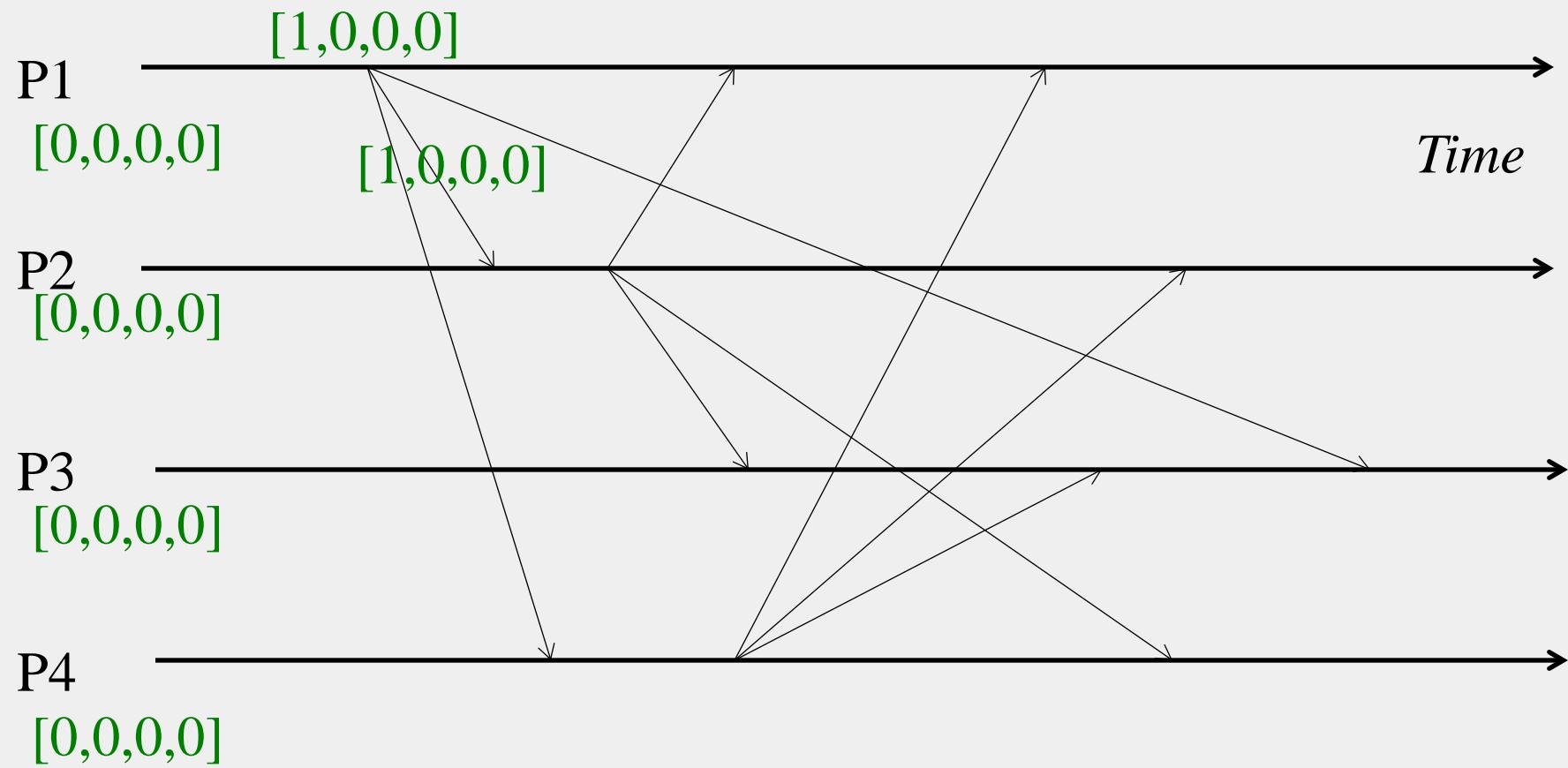
- Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers
- Formally
 - *If $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$ then any correct process that delivers m' would already have delivered m .*
 - (\rightarrow is Lamport's happens-before)

CAUSAL MULTICAST: DATASTRUCTURES

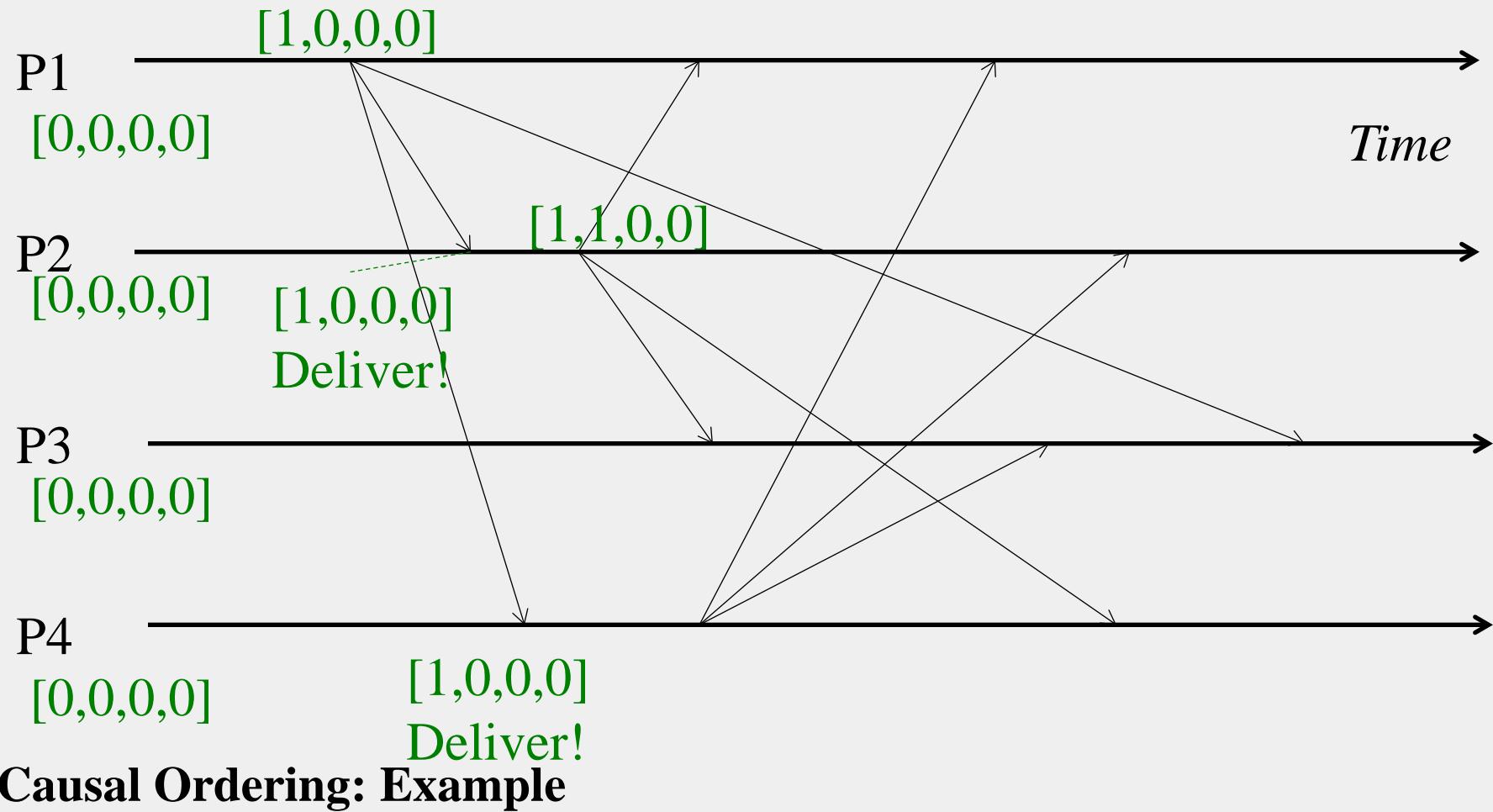
- Each receiver maintains a vector of per-sender sequence numbers (integers)
 - Similar to FIFO Multicast, but updating rules are different
 - Processes P_1 through P_N
 - P_i maintains a vector $P_i[1\dots N]$ (initially all zeroes)
 - $P_i[j]$ is the latest sequence number P_i has received from P_j

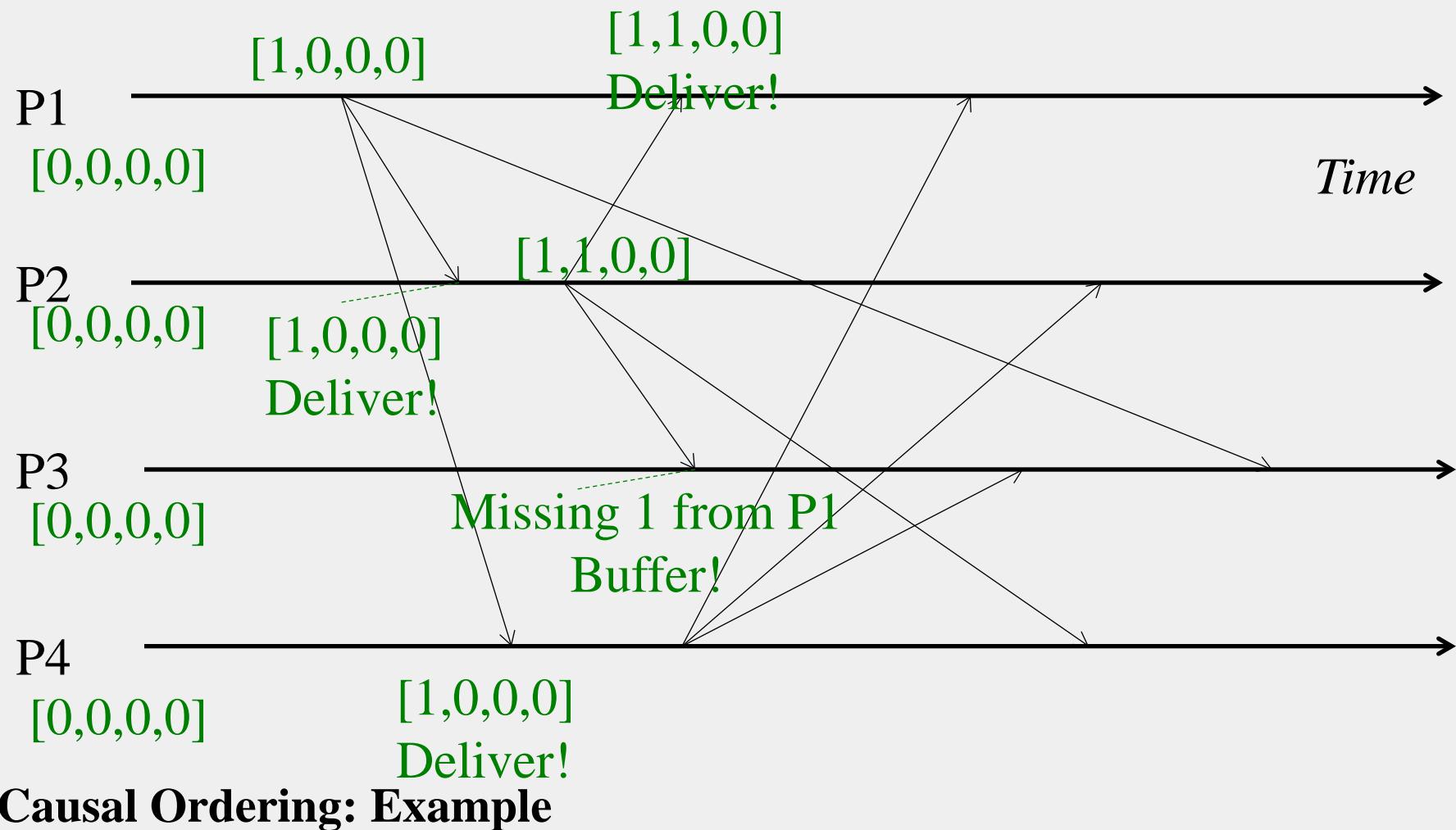
CAUSAL MULTICAST: UPDATING RULES

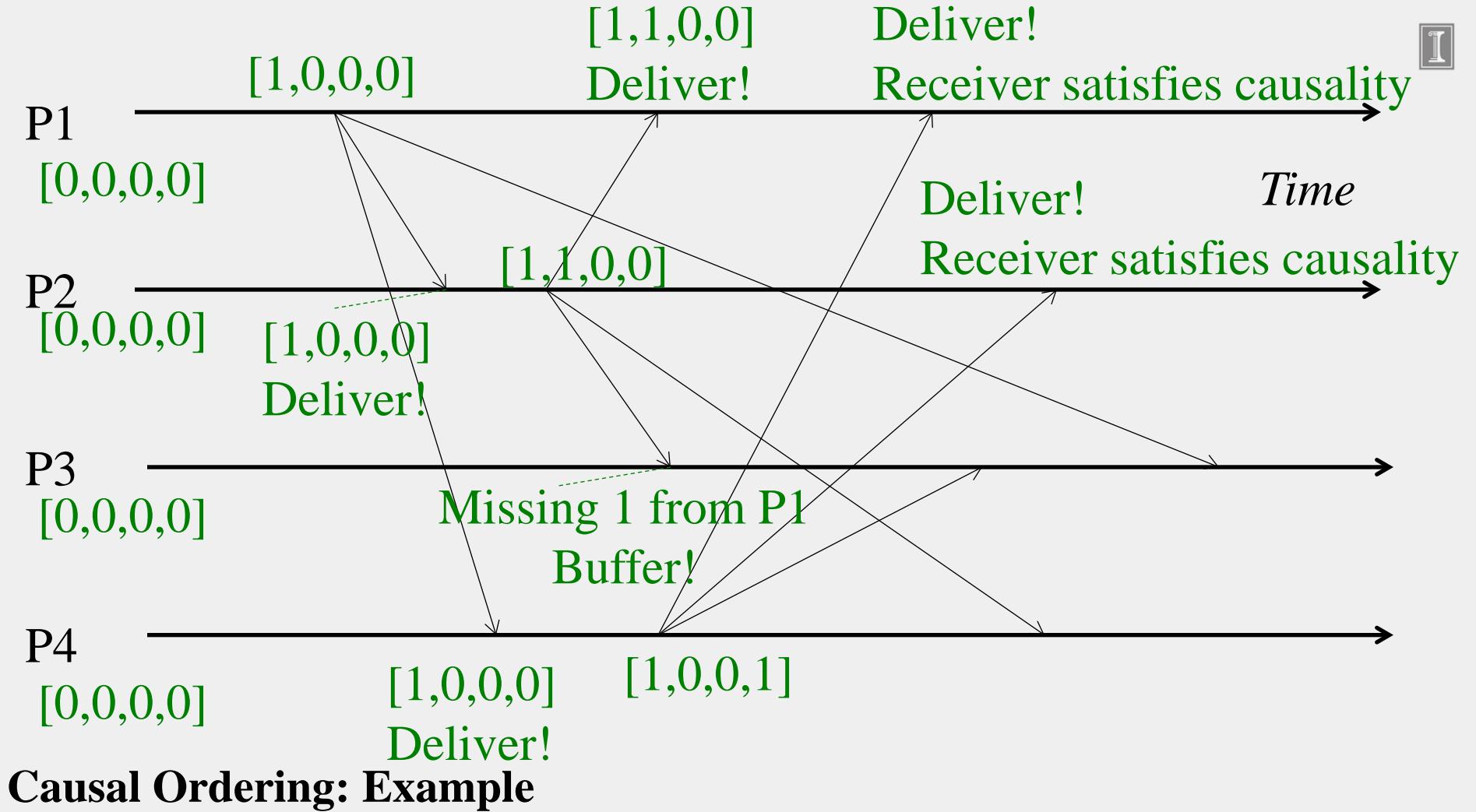
- Send multicast at process P_j :
 - Set $P_j[j] = P_j[j] + 1$
 - Include new entire vector $P_j[1\dots N]$ in multicast message as its sequence number
- Receive multicast: If P_i receives a multicast from P_j with vector $M[1\dots N]$ ($= P_j[1\dots N]$) in message, buffer it until both:
 1. This message is the next one P_i is expecting from P_j , i.e.,
 - $M[j] = P_i[j] + 1$
 2. All multicasts, anywhere in the group, which happened-before M have been received at P_i , i.e.,
 - For all $k \neq j$: $M[k] \leq P_i[k]$
 - i.e., ***Receiver satisfies causality***
 3. When above two conditions satisfied, deliver M to application and set $P_i[j] = M[j]$

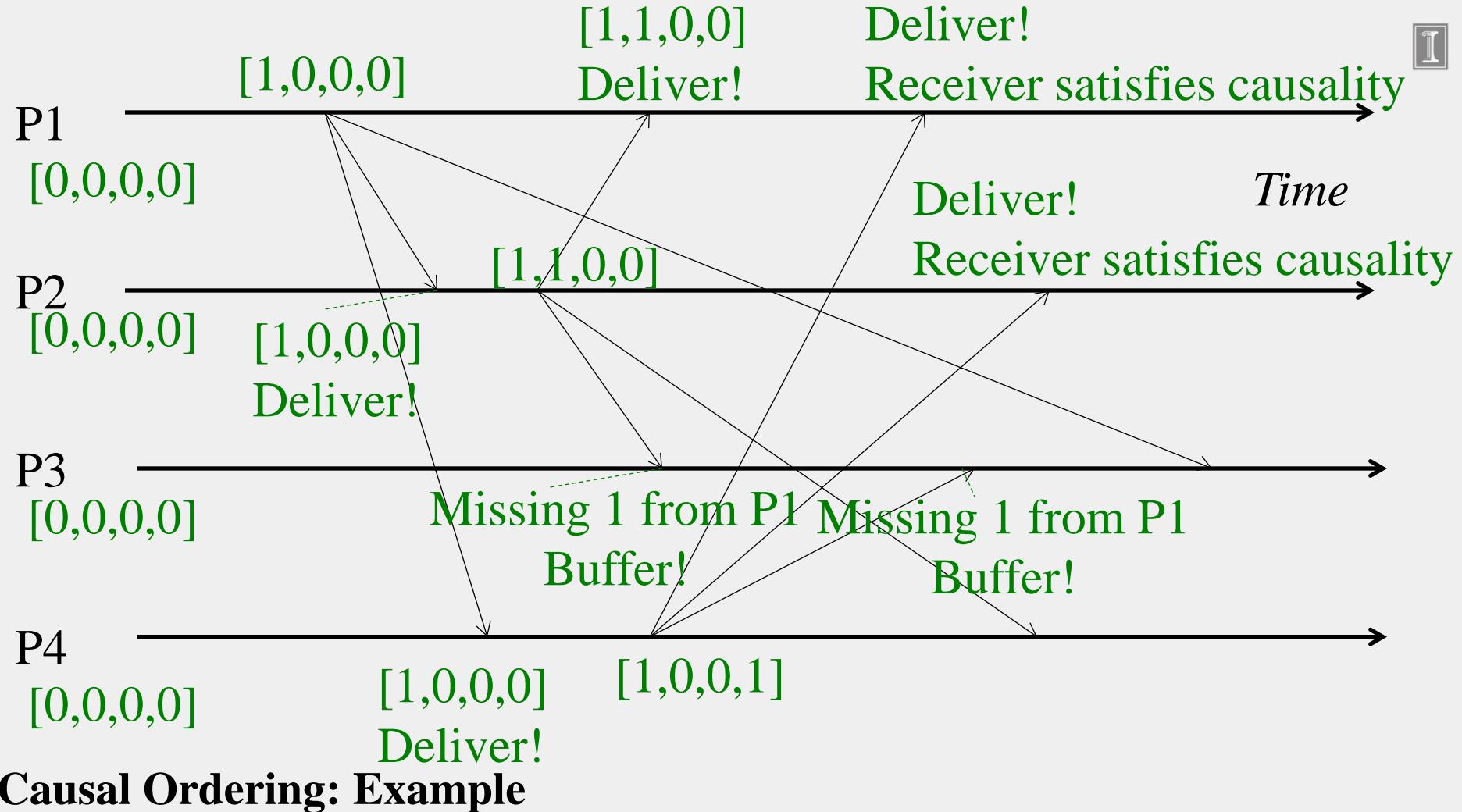


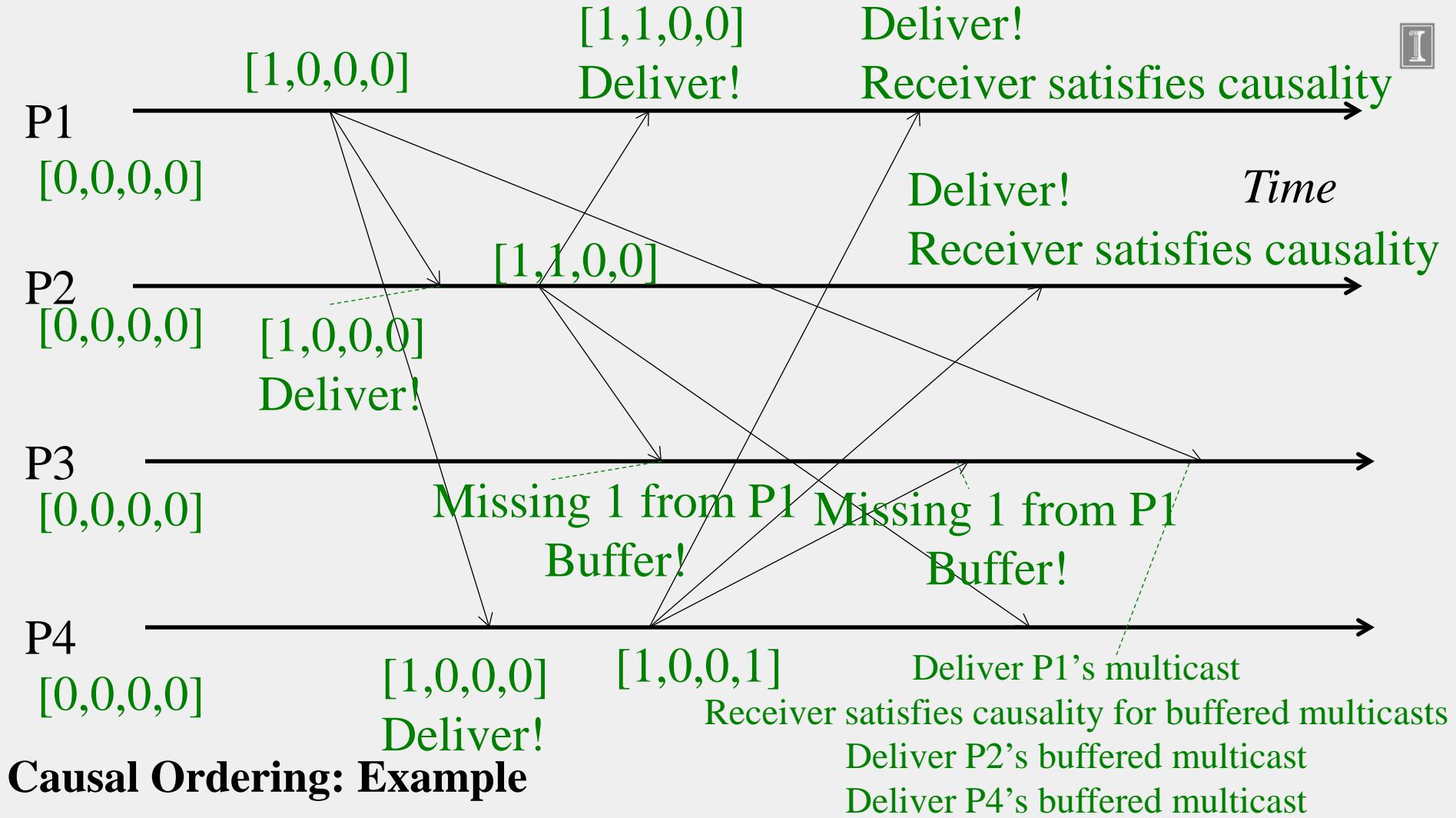
Causal Ordering: Example

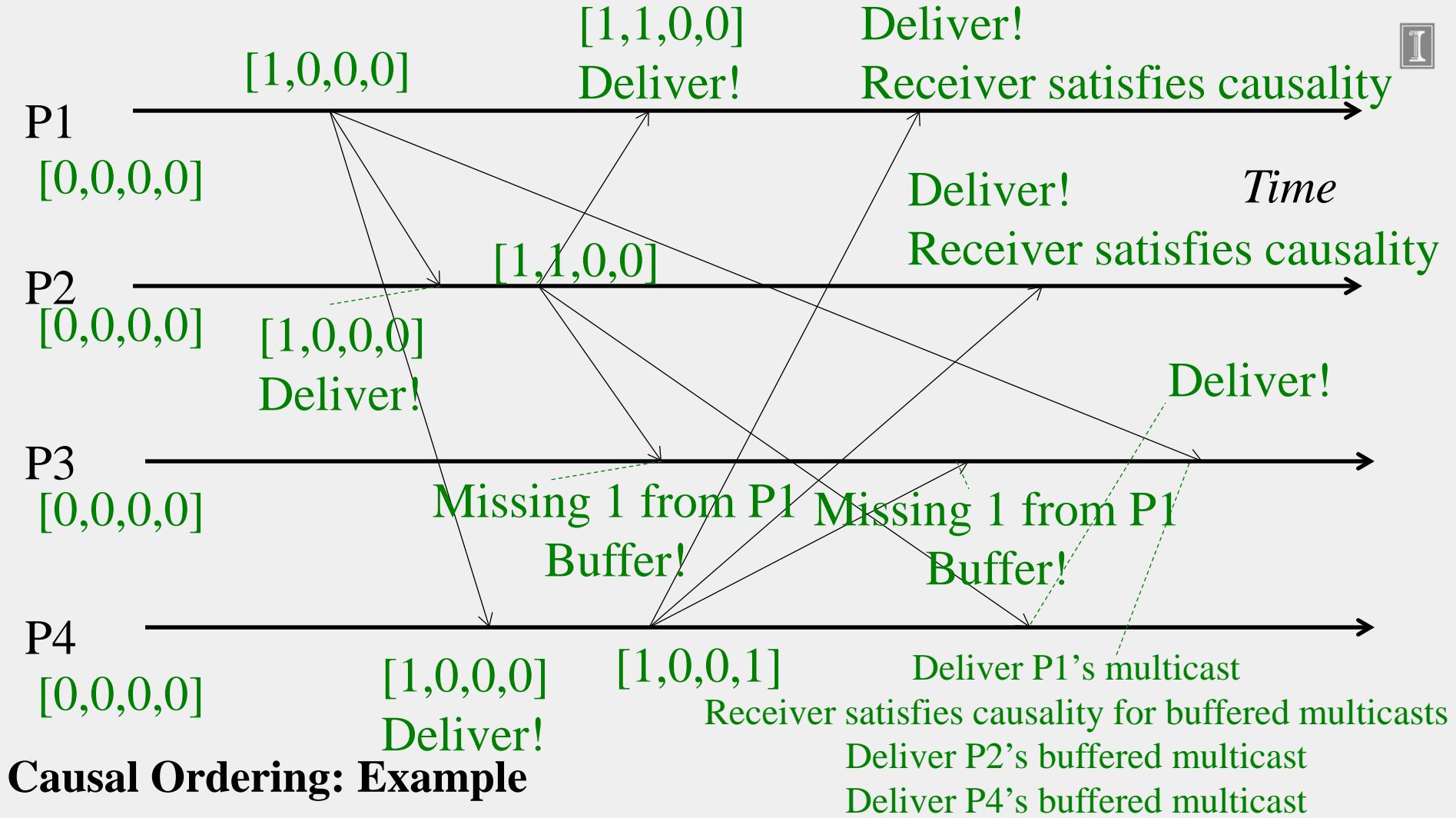












SUMMARY: MULTICAST ORDERING

- Ordering of multicasts affects correctness of distributed systems using multicasts
- Three popular ways of implementing ordering
 - FIFO, Causal, Total
- And their implementations
- What about reliability of multicasts?
- What about failures?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MULTICAST

Lecture D

RELIABLE MULTICAST

RELIABLE MULTICAST

- Reliable multicast loosely says that every process in the group receives all multicasts
 - Reliability is orthogonal to ordering
 - Can implement Reliable-FIFO, or Reliable-Causal, or Reliable-Total, or Reliable-Hybrid protocols
- What about process failures?
- Definition becomes vague

RELIABLE MULTICAST (UNDER FAILURES)

- Need all *correct* (i.e., non-faulty) processes to receive the same set of multicasts as all other correct processes
 - Faulty processes are unpredictable, so we won't worry about them

IMPLEMENTING RELIABLE MULTICAST

- Let's assume we have reliable unicast (e.g., TCP) available to us
- First-cut: Sender process (of each multicast M) sequentially sends a reliable unicast message to all group recipients
- First-cut protocol does not satisfy reliability
 - If sender fails, some correct processes might receive multicast M, while other correct processes might not receive M

REALLY IMPLEMENTING RELIABLE MULTICAST

- Trick: Have receivers help the sender
 1. Sender process (of each multicast M) sequentially sends a reliable unicast message to all group recipients
 2. When a receiver receives multicast M, it also sequentially sends M to all the group's processes

ANALYSIS

- Not the most efficient multicast protocol, but reliable
- Proof is by contradiction
- Assumption two correct processes P_i and P_j are so that P_i received a multicast M and P_j did not receive that multicast M
 - Then P_i would have sequentially sent the multicast M to all group members, including P_j , and P_j would have received M
 - A contradiction
 - Hence our initial assumption must be false
 - Hence protocol preserves reliability

NEXT

- Combining fault-tolerance and multicast



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MULTICAST

Lecture E

VIRTUAL SYNCHRONY

VIRTUAL SYNCHRONY OR VIEW SYNCHRONY

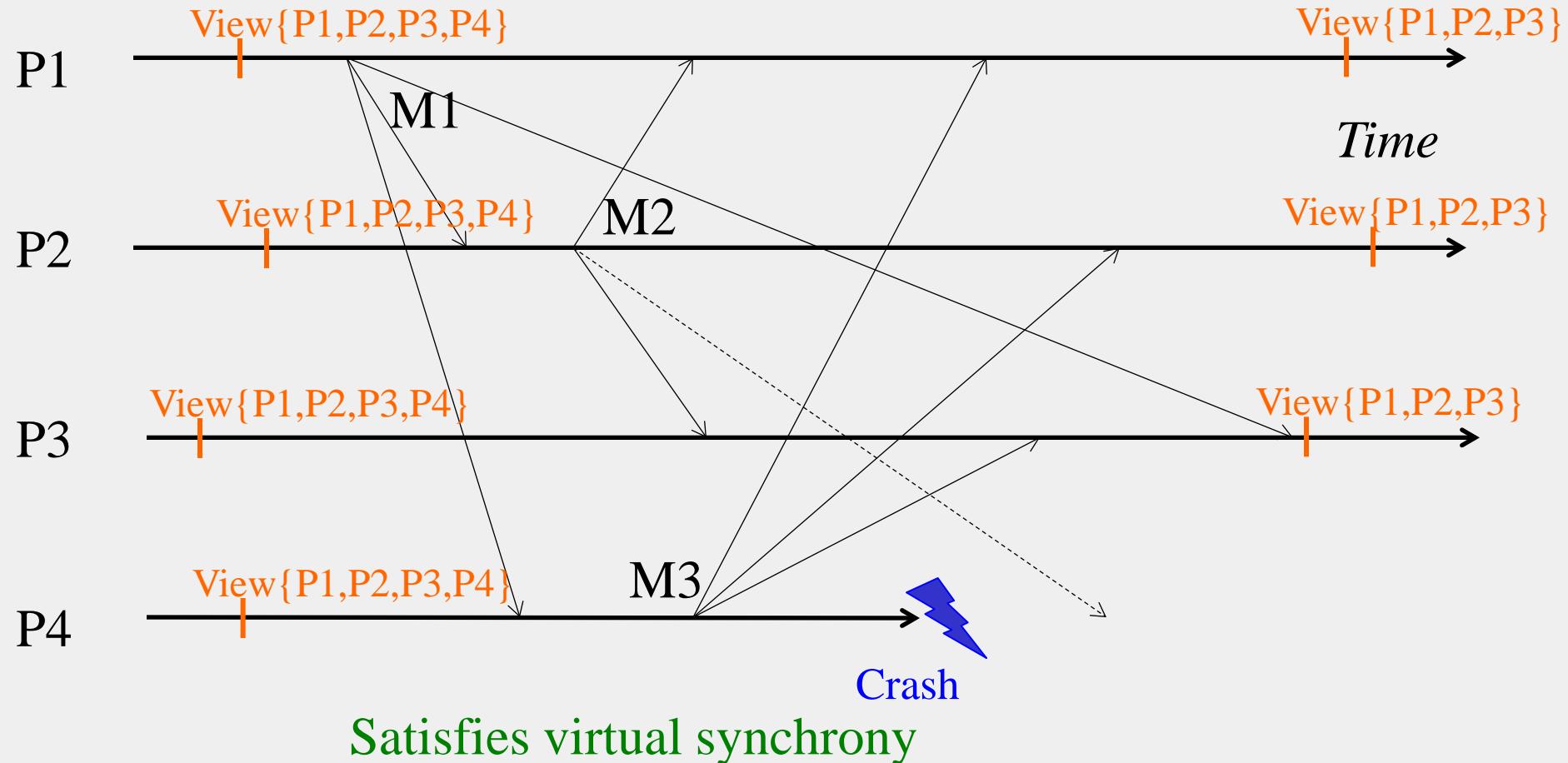
- Attempts to preserve multicast ordering and reliability in spite of failures
- Combines a membership protocol with a multicast protocol
- Systems that implemented it (like Isis) have been used in NYSE, French Air Traffic Control System, Swiss Stock Exchange

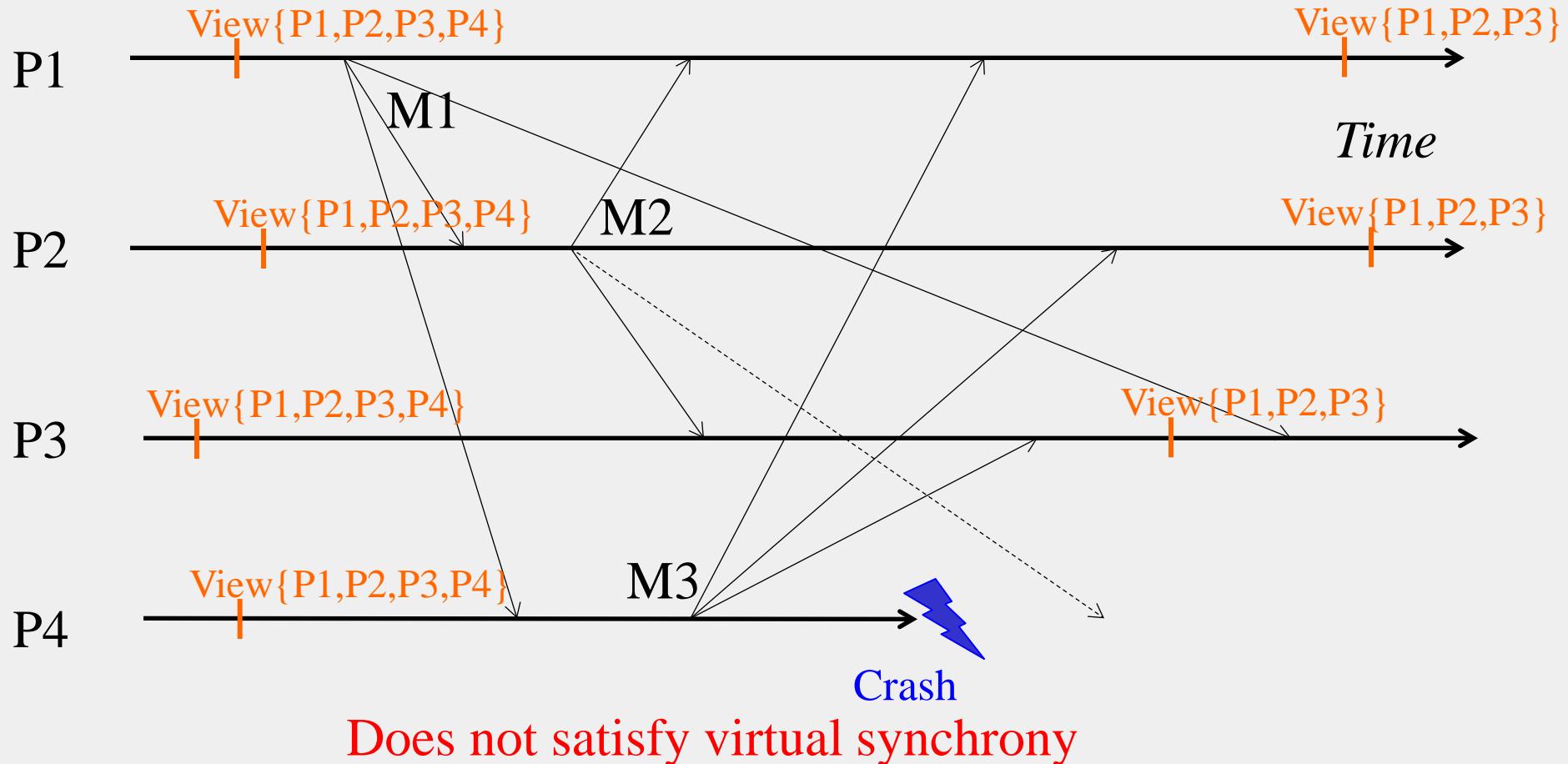
VIEWS

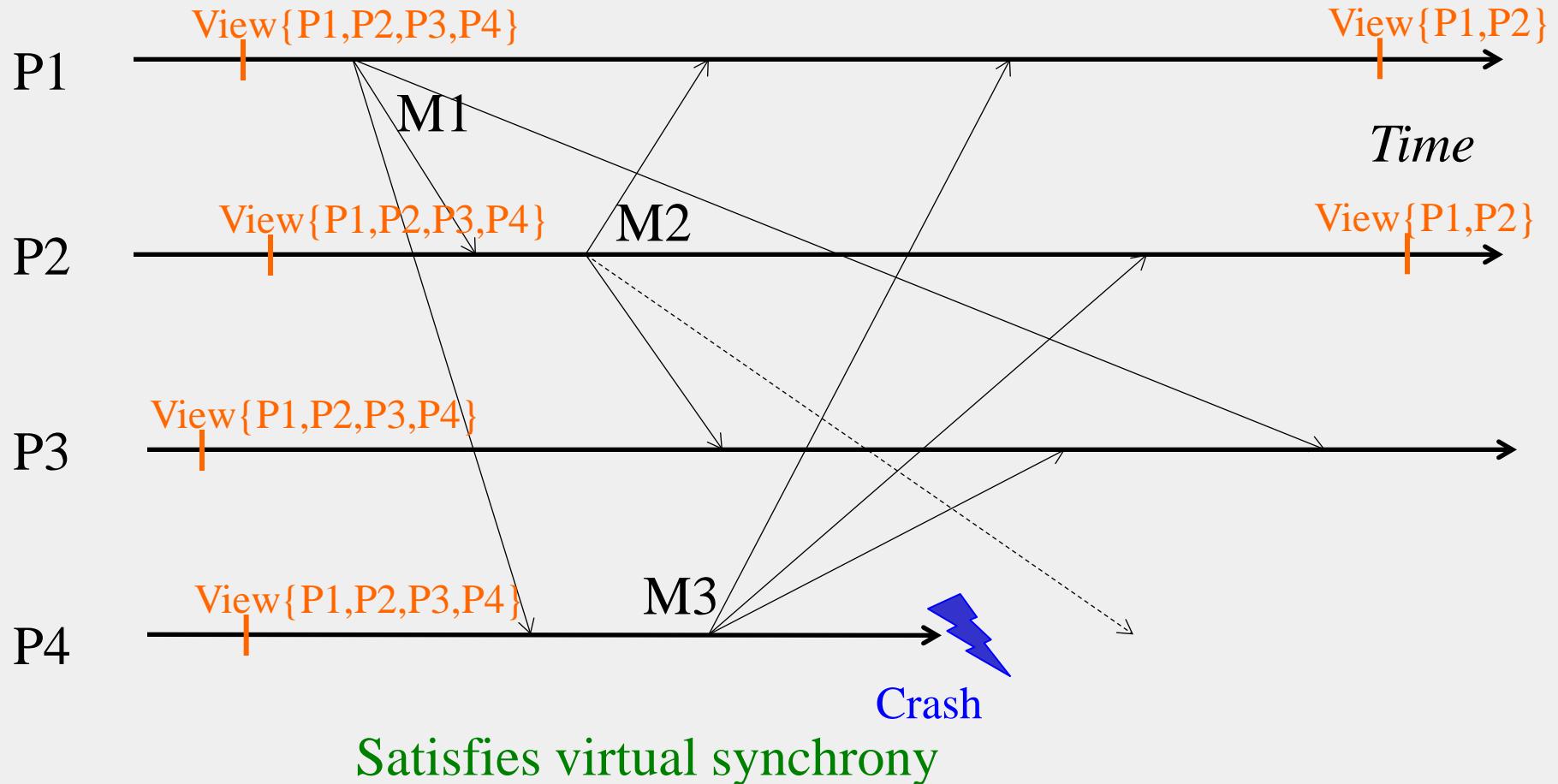
- Each process maintains a membership list
- The membership list is called a *View*
- An update to the membership list is called a *View Change*
 - Process join, leave, or failure
- Virtual synchrony guarantees that all **view changes are delivered in the same order at all correct processes**
 - If a correct P1 process receives views, say $\{P1\}$, $\{P1, P2, P3\}$, $\{P1, P2\}$, $\{P1, P2, P4\}$ then
 - Any other correct process receives the *same sequence* of view changes (after it joins the group)
 - P2 receives views $\{P1, P2, P3\}$, $\{P1, P2\}$, $\{P1, P2, P4\}$
 - Views may be delivered at different physical times at processes, but they are delivered in the same order

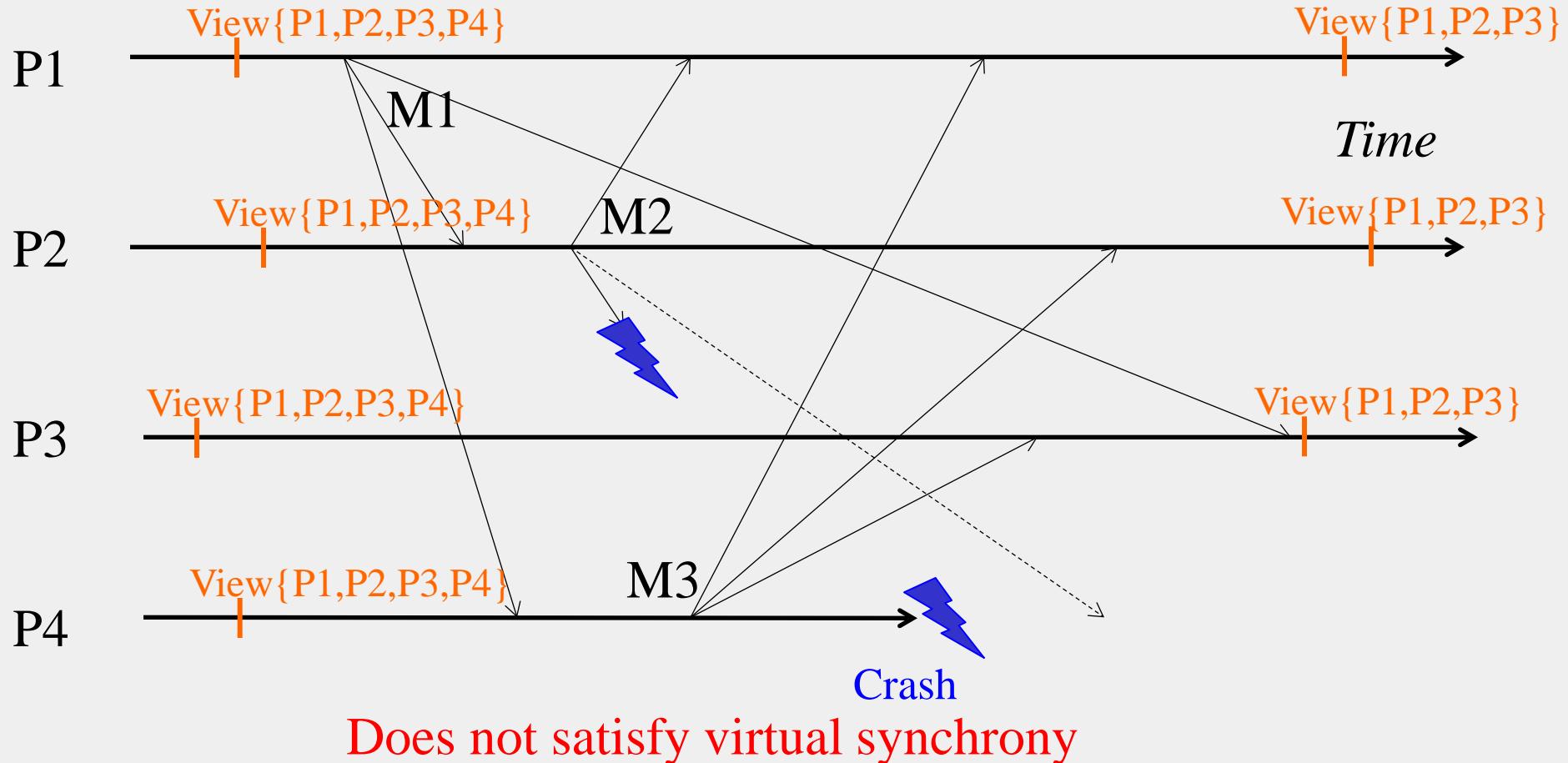
VSYNC MULTICASTS

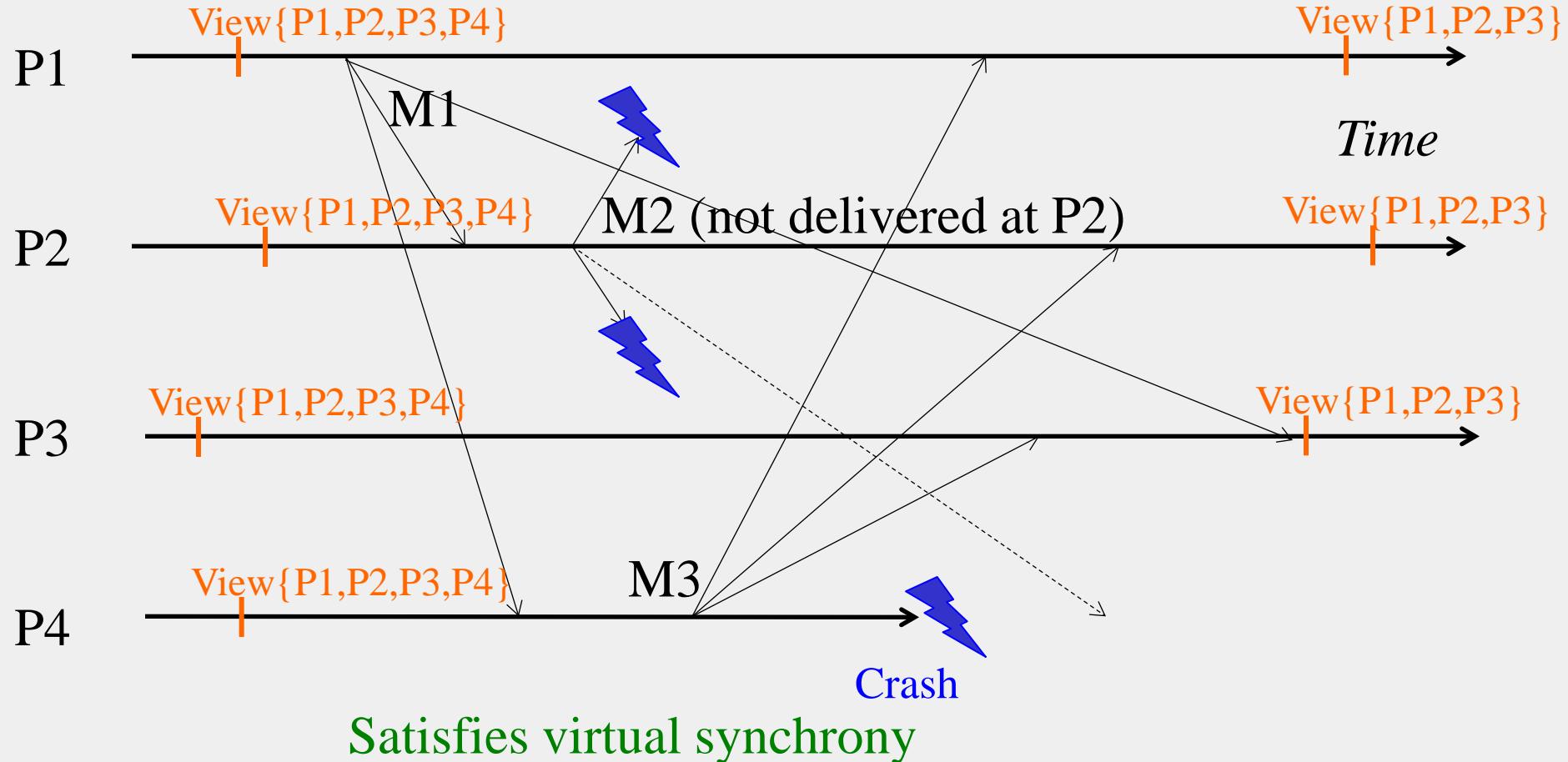
- A multicast M is said to be “delivered in a view V at process Pi” if
 - P_i receives view V, and then sometime before P_i receives the next view it delivers multicast M
- Virtual synchrony ensures that
 1. **The set of multicasts delivered in a given view is the same set at all correct processes that were in that view**
 - *What happens in a View, stays in that View*
 2. The sender of the multicast message also belongs to that view
 3. If a process P_i does not deliver a multicast M in view V while other processes in the view V delivered M in V, then P_i will be *forcibly removed* from the next view delivered after V at the other processes

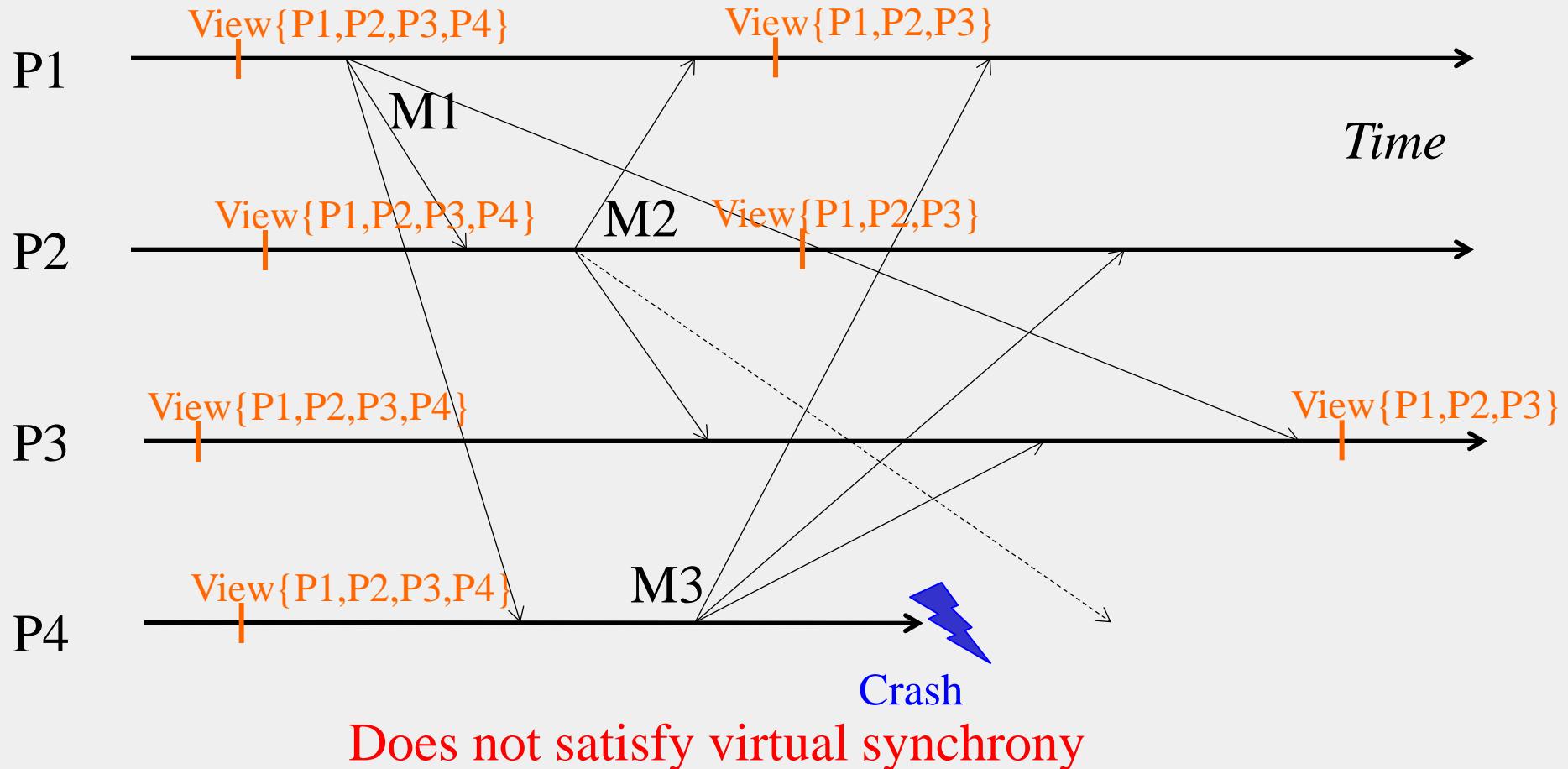


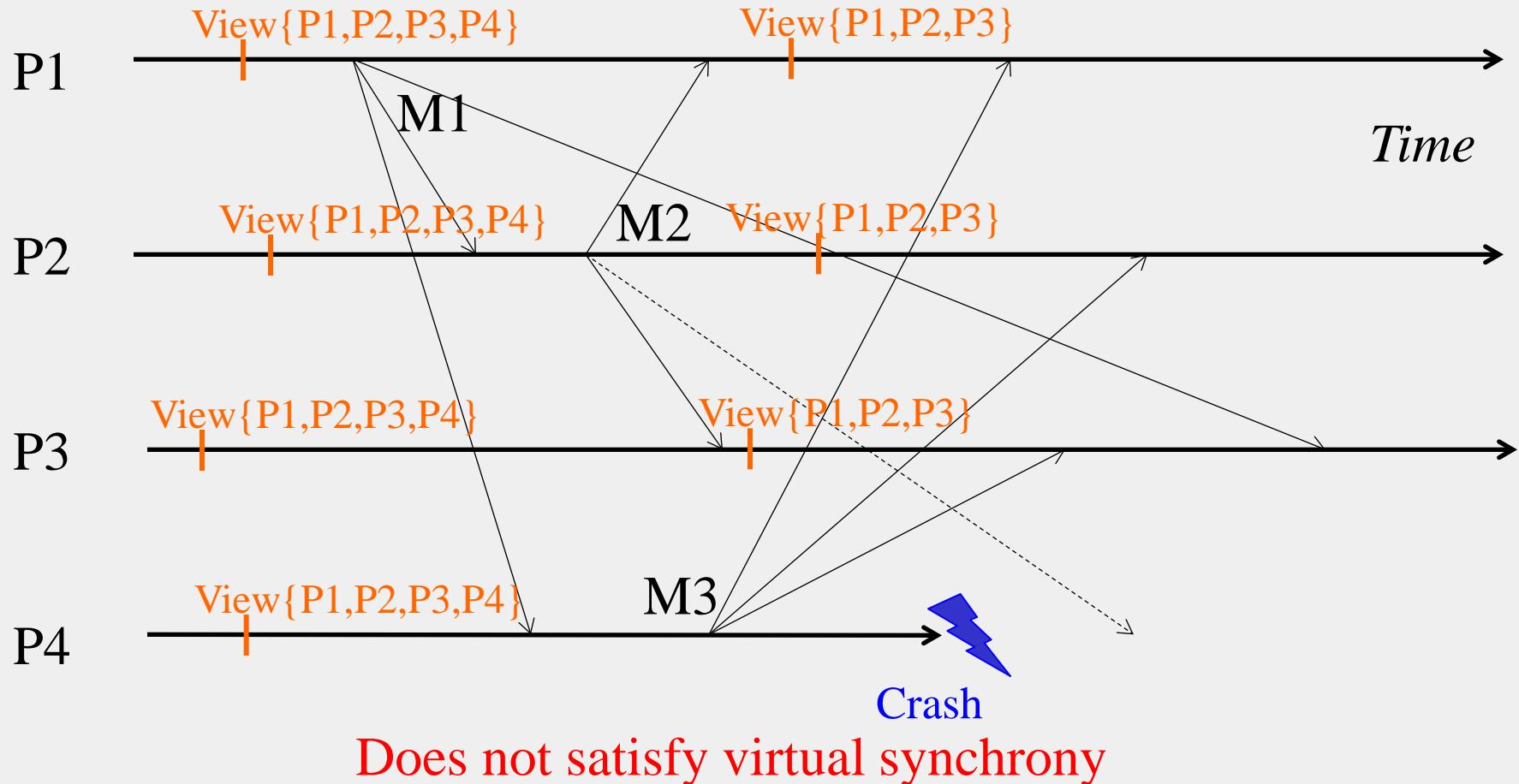


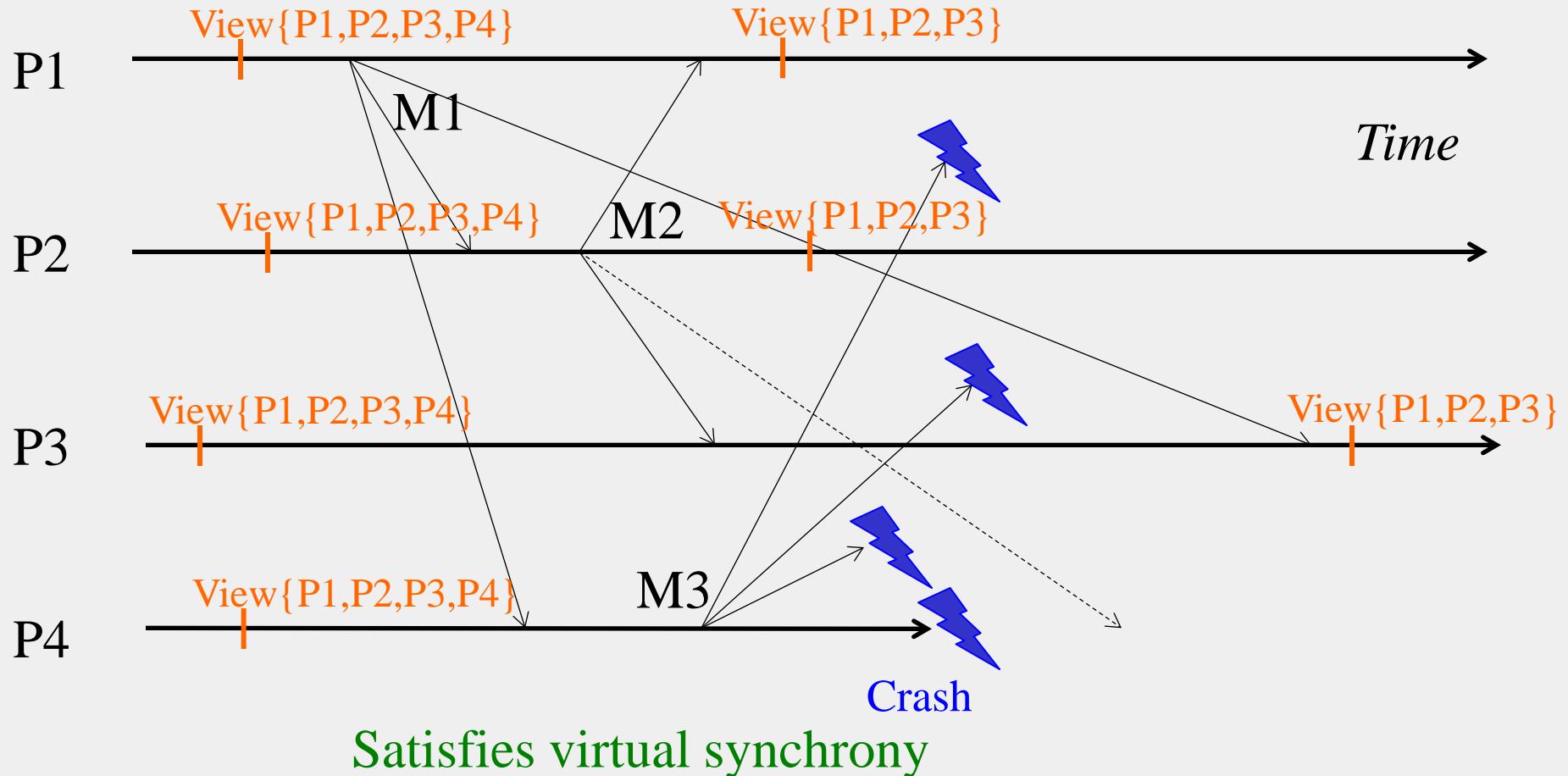










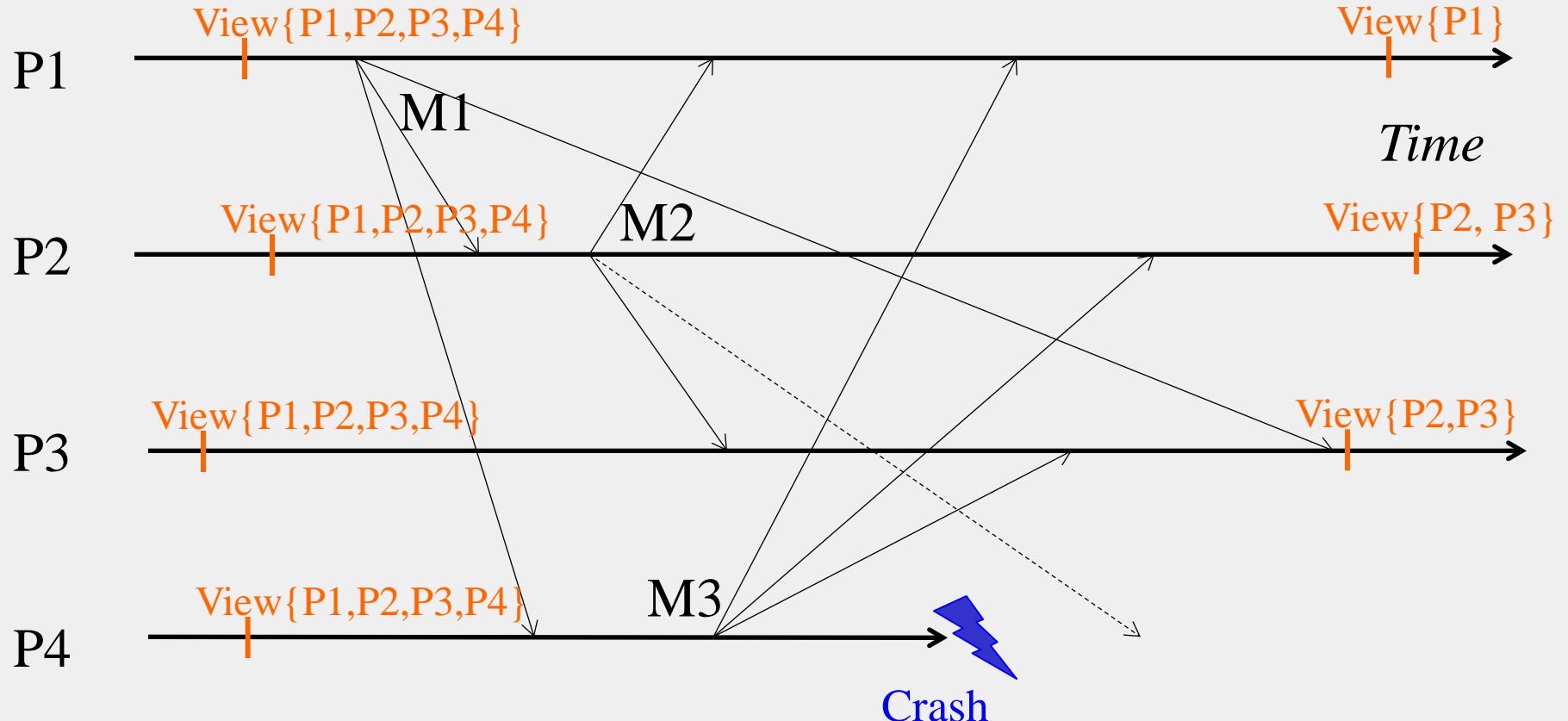


WHAT ABOUT MULTICAST ORDERING?

- Again, orthogonal to virtual synchrony
- The set of multicasts delivered in a view can be ordered either
 - FIFO
 - Or Causally
 - Or Totally
 - Or using a hybrid scheme

ABOUT THAT NAME

- Called “virtual synchrony” since in spite of running on an asynchronous network, it gives the appearance of a synchronous network underneath that obeys the same ordering at all processes
- So can this virtually synchronous system be used to implement consensus?
- No! VSync groups susceptible to partitioning
 - E.g., due to inaccurate failure detections



Partitioning in View synchronous systems

SUMMARY

- Multicast an important building block for cloud computing systems
- Depending on application need, can implement
 - Ordering
 - Reliability
 - Virtual synchrony



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

PAXOS

Lecture A

THE CONSENSUS PROBLEM

GIVE IT A THOUGHT

Have you ever wondered why distributed server vendors always only offer solutions that promise five-9's reliability, seven-9's reliability, but never 100% reliable?

The fault does not lie with the companies themselves, or the worthlessness of humanity.

The fault lies in the impossibility of consensus.

WHAT IS COMMON TO ALL OF THESE?

A group of servers attempting:

- To make sure that all of them receive the same updates in the same order as each other
- To keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously
- To elect a leader among them, and let everyone in the group know about it
- To ensure mutually exclusive (one process at a time only) access to a critical resource like a file

WHAT IS COMMON TO ALL OF THESE?

A group of servers attempting:

- Make sure that all of them receive the same updates in the same order as each other [Reliable Multicast]
- To keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously [Membership/Failure Detection]
- Elect a leader among them, and let everyone in the group know about it [Leader Election]
- To ensure mutually exclusive (one process at a time only) access to a critical resource like a file [Mutual Exclusion]

So WHAT IS COMMON?

- Let's call each server a “process” (think of the daemon at each server)
- All of these were groups of processes attempting to *coordinate* with each other and reach *agreement* on the value of something
 - The ordering of messages
 - The up/down status of a suspected failed process
 - Who the leader is
 - Who has access to the critical resource
- All of these are related to the *Consensus* problem

WHAT IS CONSENSUS?

Formal problem statement

- N processes
- Each process p has

input variable x_p : initially either 0 or 1

output variable y_p : initially b (can be changed only once)

- Consensus problem: design a protocol so that at the end, either:

1. All processes set their output variables to 0 (all-0's)
2. Or all processes set their output variables to 1 (all-1's)

WHAT IS CONSENSUS? (2)

- Every process contributes a value
- *Goal is to have all processes decide same (some) value*
 - Decision once made can't be changed
- There might be other constraints
 - Validity = if everyone proposes same value, then that's what's decided
 - Integrity = decided value must have been proposed by some process
 - Non-triviality = there is at least one initial system state that leads to each of the all-0's or all-1's outcomes

WHY IS IT IMPORTANT?

- Many problems in distributed systems are equivalent to (*or harder than*) consensus!
 - Perfect Failure Detection
 - Leader election (select exactly one leader, and every alive process knows about it)
 - Agreement (harder than consensus)
- So consensus is a very important problem, and solving it would be really useful!
- So, is there a solution to Consensus?

Two DIFFERENT MODELS OF DISTRIBUTED SYSTEMS

- Synchronous System Model and Asynchronous System Model
- Synchronous Distributed System
 - Each message is received within bounded time
 - Drift of each process' local clock has a known bound
 - Each step in a process takes $lb < \text{time} < ub$

E.g., A collection of processors connected by a communication bus, e.g., a Cray supercomputer or a multicore machine

ASYNCHRONOUS SYSTEM MODEL

- **Asynchronous** Distributed System
 - No bounds on process execution
 - The drift rate of a clock is arbitrary
 - No bounds on message transmission delays

E.g., The Internet is an asynchronous distributed system, so are ad-hoc and sensor networks

- *This is a more general (and thus challenging) model than the synchronous system model. A protocol for an asynchronous system will also work for a synchronous system (but not vice-versa)*

POSSIBLE OR NOT

- In the synchronous system model
 - Consensus is solvable
- In the asynchronous system model
 - Consensus is impossible to solve
 - Whatever protocol/algorithm you suggest, there is always a worst-case possible execution (with failures and message delays) that prevents the system from reaching consensus
 - Powerful result (see the FLP proof in the optional lecture of this series)
 - Subsequently, safe or probabilistic solutions have become quite popular to consensus or related problems.

So WHAT NEXT?

- Next lecture: Let's just solve consensus!



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

PAXOS

Lecture B

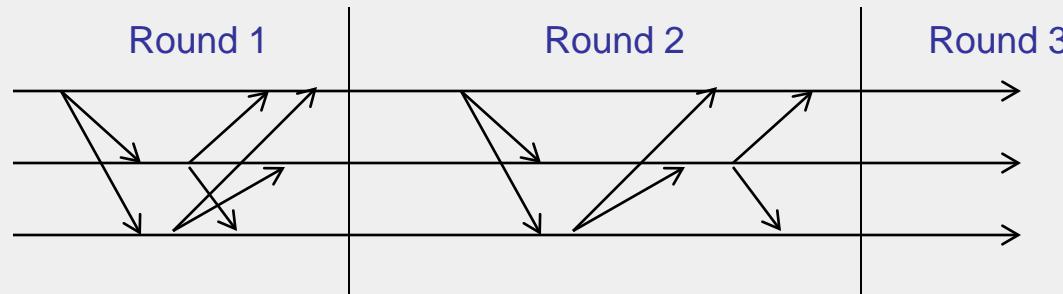
CONSENSUS IN
SYNCHRONOUS SYSTEMS

LET'S TRY TO SOLVE CONSENSUS!

- Uh, what's the **system model?**
(assumptions!)
- **Synchronous system:** bounds on
 - Message delays
 - Upper bound on clock drift rates
 - Max time for each process stepe.g., multiprocessor (common clock across processors)
- Processes can fail by stopping (crash-stop or crash failures)

CONSENSUS IN SYNCHRONOUS SYSTEMS

- For a system with at most f processes crashing
 - All processes are synchronized and operate in “rounds” of time
 - the algorithm proceeds in $f+1$ rounds (with timeout), using reliable communication to all members
 - $Values^r_i$: the set of proposed values known to p_i at the beginning of round r .



CONSENSUS IN SYNCHRONOUS SYSTEM

Possible to achieve!

- For a system with at most f processes crashing
 - All processes are synchronized and operate in “rounds” of time
 - The algorithm proceeds in $f+1$ rounds (with timeout), using reliable communication to all members
 - $Values^r_i$: the set of proposed values known to p_i at the beginning of round r .
- Initially $Values^0_i = \{\}$; $Values^1_i = \{v_i\}$
 - for round = 1 to $f+1$ do
 - multicast** ($Values^r_i - Values^{r-1}_i$) // iterate through processes, send each a message
 - $Values^{r+1}_i \leftarrow Values^r_i$
 - for each V_j received
 - $Values^{r+1}_i = Values^{r+1}_i \cup V_j$
 - end
 - end

$d_i = \text{minimum}(Values^{f+1}_i)$

WHY DOES THE ALGORITHM WORK?

- After $f+1$ rounds, all non-faulty processes would have received the same set of values. Proof by contradiction.
- Assume that two non-faulty processes, say p_i and p_j , differ in their final set of values (i.e., after $f+1$ rounds)
- Assume that p_i possesses a value v that p_j does not possess.
 - p_i must have received v in the **very last** round
 - Else, p_i would have sent v to p_j in that last round
 - So, in the last round: a third process, p_k , must have sent v to p_i , but then crashed before sending v to p_j .
 - Similarly, a fourth process sending v in the **last-but-one round** must have crashed; otherwise, both p_k and p_j should have received v .
 - Proceeding in this way, we infer at least one (unique) crash in each of the preceding rounds.
 - This means a total of $f+1$ crashes, while we have assumed at most f crashes can occur => contradiction.

NEXT

- Let's be braver and solve Consensus in the Asynchronous System Model



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

PAXOS

Lecture C

PAXOS, SIMPLY

CONSENSUS PROBLEM

- Consensus **impossible** to solve in asynchronous systems (FLP Proof)
 - Key to the Proof: It is impossible to distinguish a failed process from one that is just very very (very) slow. Hence the rest of the alive processes may stay ambivalent (forever) when it comes to deciding.
- But Consensus important since it maps to many important distributed computing problems
- Um, can't we just solve consensus?

YES WE CAN!

- Paxos algorithm

- Most popular “consensus-solving” algorithm
- Does not solve consensus problem (which would be impossible, because we already proved that)
- But provides safety and eventual liveness
- A lot of systems use it
 - Zookeeper (Yahoo!), Google Chubby, and many other companies

- Paxos invented by? (take a guess)

YES WE CAN!

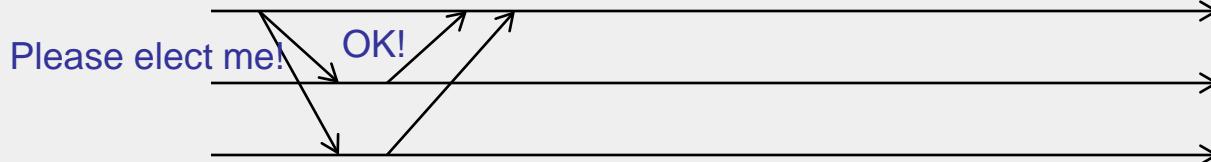
- Paxos invented by Leslie Lamport
- Paxos provides safety and eventual liveness
 - Safety: Consensus is not violated
 - Eventual Liveness: If things go well sometime in the future (messages, failures, etc.), there is a good chance consensus will be reached. But there is no guarantee.
- FLP result still applies: Paxos is not *guaranteed* to reach Consensus (ever, or within any bounded time)

POLITICAL SCIENCE 101, I.E., PAXOS GROKED

- Paxos has **rounds**; each round has a unique ballot id
- Rounds are asynchronous
 - Time synchronization not required
 - If you're in round j and hear a message from round $j+1$, abort everything and move over to round $j+1$
 - Use timeouts; may be pessimistic
- Each round itself broken into phases (which are also asynchronous)
 - Phase 1: A leader is elected (**Election**)
 - Phase 2: Leader proposes a value, processes ack (**Bill**)
 - Phase 3: Leader multicasts final value (**Law**)

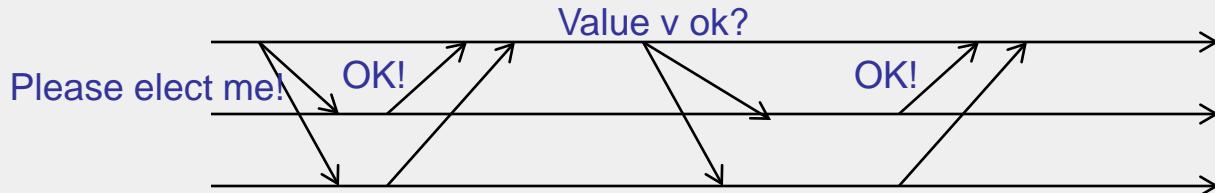
PHASE 1 - ELECTION

- Potential leader chooses a unique ballot id, higher than anything seen so far
- Sends to all processes
- Processes wait, respond once to highest ballot id
 - If potential leader sees a higher ballot id, it can't be a leader
 - Paxos tolerant to multiple leaders, but we'll only discuss 1 leader case
 - Processes also **log** received ballot ID on disk
- If a process has in a previous round decided on a value v' , it includes value v' in its response
- If majority (i.e., quorum) respond OK then you are the leader
 - If no one has majority, start new round
- (If things go right) A round cannot have two leaders (why?)



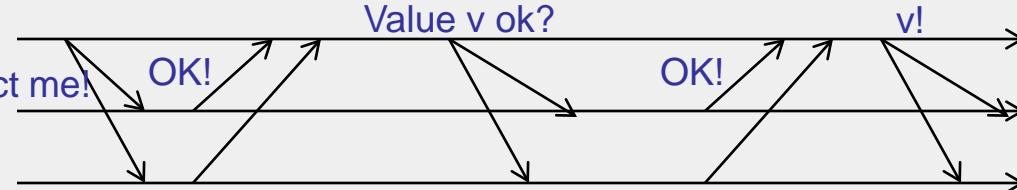
PHASE 2 – PROPOSAL (BILL)

- Leader sends proposed value v to all
 - use $v=v'$ if some process already decided in a previous round and sent you its decided value v'
- Recipient logs on disk; responds OK



PHASE 3 - DECISION (LAW)

- If leader hears a majority of OKs, it lets everyone know of the decision
- Recipients receive decision, log it on disk



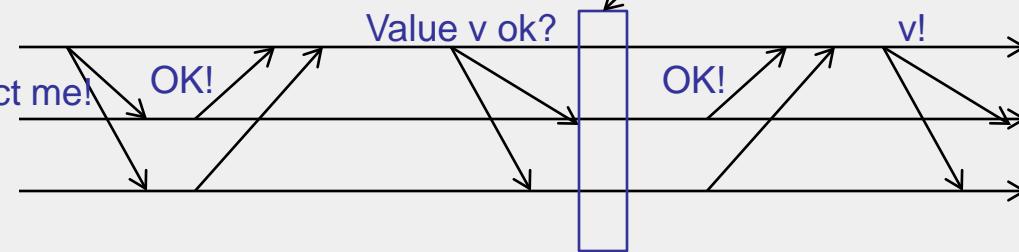
WHICH IS THE POINT OF No-RETURN?

- That is, when is consensus reached in the system



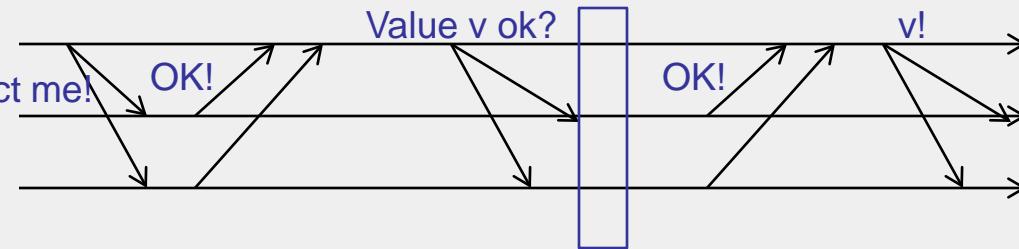
WHICH IS THE POINT OF No-RETURN?

- If/when a majority of processes hear proposed value and accept it (i.e., are about to/have respond(ed) with an OK!)
- Processes *may not know it yet*, but a decision has been made for the group
 - Even leader does not know it yet
- What if leader fails after that?
 - Keep having rounds until some round completes



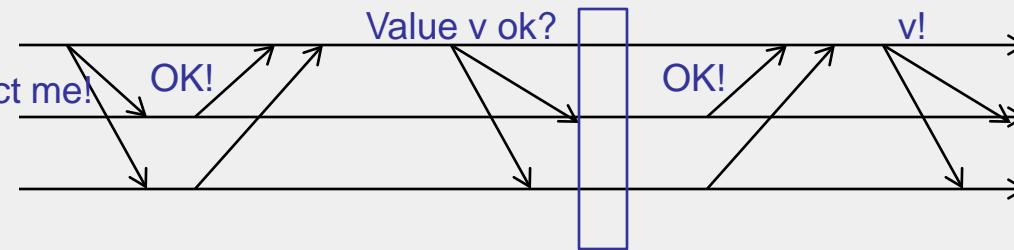
SAFETY

- If some round has a majority (i.e., quorum) hearing proposed value v' and accepting it (middle of Phase 2), then subsequently at each round either: 1) the round chooses v' as decision or 2) the round fails
- Proof:
 - Potential leader waits for majority of OKs in Phase 1
 - At least one will contain v' (because two majorities or quorums always intersect)
 - It will choose to send out v' in Phase 2
- Success requires a majority, and any two majority sets intersect



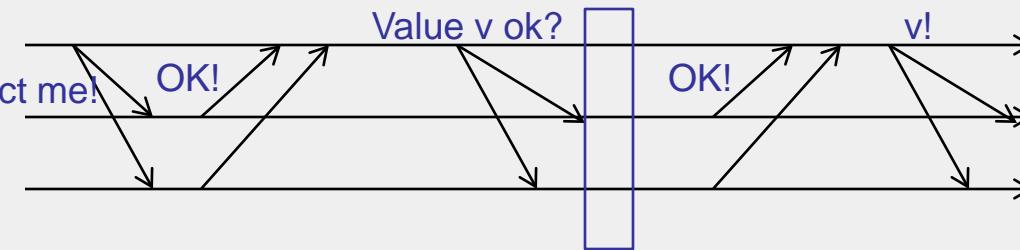
WHAT COULD GO WRONG?

- Process fails
 - Majority does not include it
 - When process restarts, it uses log to retrieve a past decision (if any) and past-seen ballot ids. Tries to know of past decisions.
- Leader fails
 - Start another round
- Messages dropped
 - If too flaky, just start another round
- Note that anyone can start a round any time
- Protocol may never end – tough luck, buddy!
 - Impossibility result not violated
 - If things go well sometime in the future, consensus reached



WHAT COULD GO WRONG?

- A lot more!
- This is a highly simplified view of Paxos.
- See Lamport's original paper:
<http://research.microsoft.com/en-us/um/people/lamport/pubs/paxosimple.pdf>



SUMMARY

- Consensus is a very important problem
 - Equivalent to many important distributed computing problems that have to do with *reliability*
- Consensus is possible to solve in a synchronous system where message delays and processing delays are bounded
- Consensus is impossible to solve in an asynchronous system where these delays are unbounded
- Paxos protocol: widely used implementation of a safe, eventually-live consensus protocol for asynchronous systems
 - Paxos (or variants) used in Apache Zookeeper, Google's Chubby system, Active Disk Paxos, and many other cloud computing systems

NEXT

- For the brave among you: the proof of Impossibility of Consensus (FLP Proof)



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

PAXOS

Lecture D

THE FLP PROOF

CONSENSUS IN AN ASYNCHRONOUS SYSTEM

- Impossible to achieve!
- Proved in a now-famous result by Fischer, Lynch, and Patterson, 1983 (FLP)
 - Stopped many distributed system designers dead in their tracks
 - A lot of claims of “reliability” vanished overnight

RECALL

Asynchronous system: All message delays and processing delays can be arbitrarily long or short.

Consensus:

- Each process p has a **state**

- Program counter, registers, stack, local variables
- Input register x_p : initially either 0 or 1
- Output register y_p : initially b (undecided)

- Consensus Problem: design a protocol so that either

- All processes set their output variables to 0 (all-0's)
- Or all processes set their output variables to 1 (all-1's)
- Non-triviality: at least one initial system state leads to each of the above two outcomes

PROOF SETUP

- For impossibility proof, OK to consider
 1. More restrictive system model, and
 2. Easier problem
 - Why is this ok?

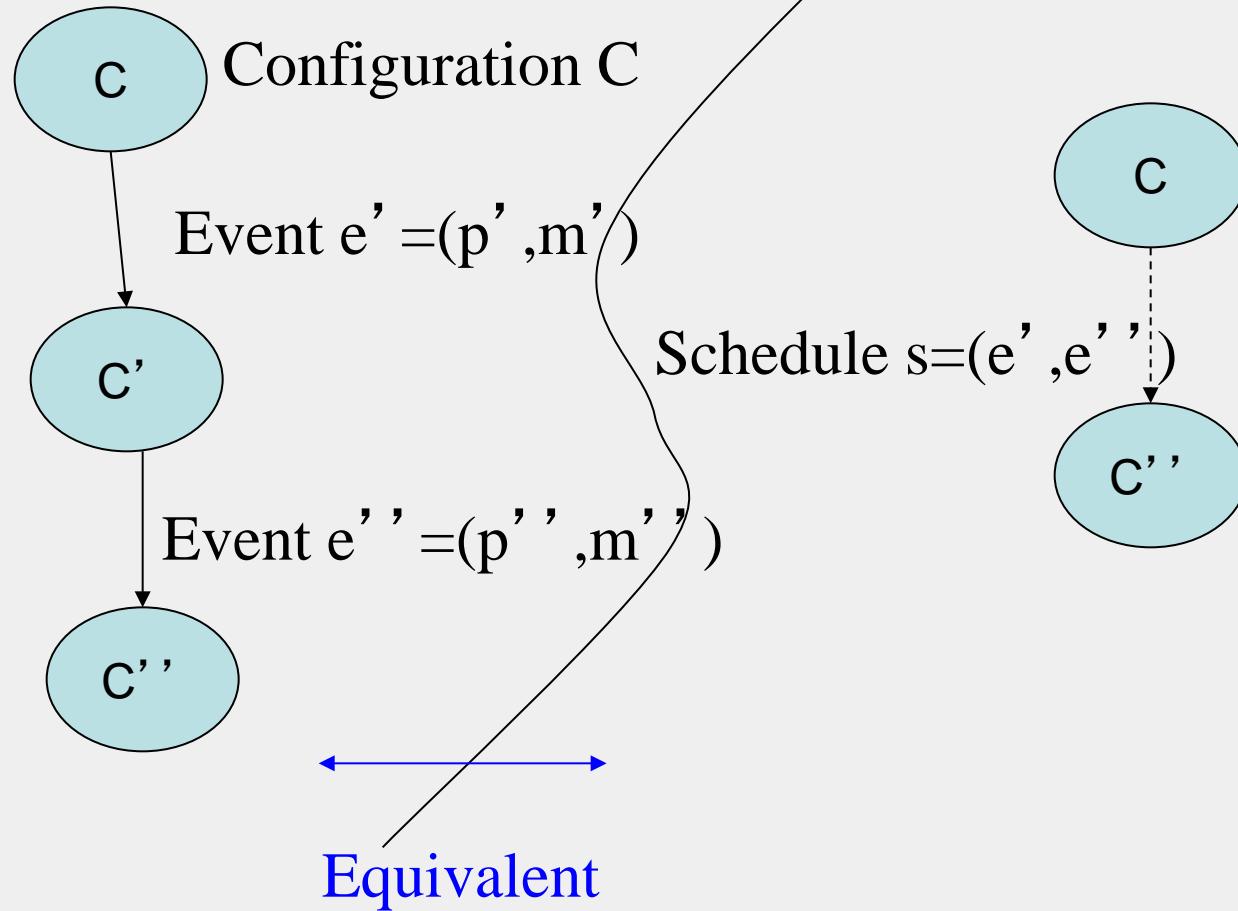
NETWORK



“Network”

STATES

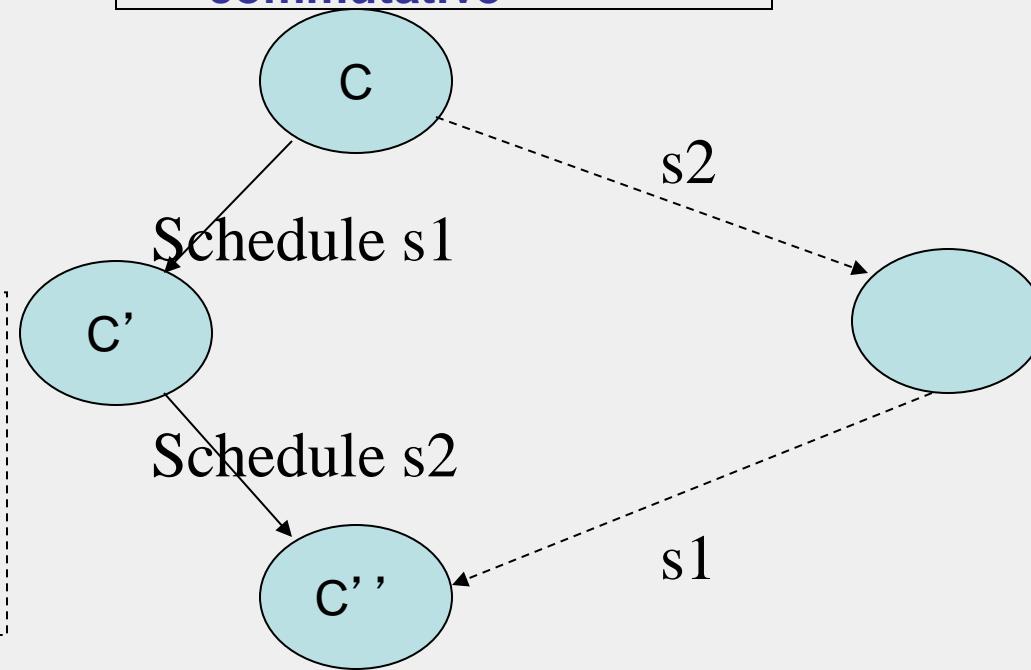
- State of a process
- **Configuration=global state.** Collection of states, one for each process; alongside state of the global buffer.
- Each **event** (different from Lamport events)
 - Receipt of a message by a process (say p)
 - Processing of message (may change recipient's state)
 - Sending out of all necessary messages by p
- **Schedule:** sequence of events



LEMMA 1

Disjoint schedules are commutative

s1 and s2 involve disjoint sets of receiving processes, and are each applicable on C



EASIER CONSENSUS PROBLEM

Easier Consensus Problem:

some process eventually
sets y_p to be 0 or 1

Only one process crashes –

we're free to choose
which one

EASIER CONSENSUS PROBLEM

- Let config. C have a set of decision values V reachable from it
 - If $|V| = 2$, config. C is bivalent
 - If $|V| = 1$, config. C is 0-valent or 1-valent, as is the case
- **Bivalent means outcome is unpredictable**

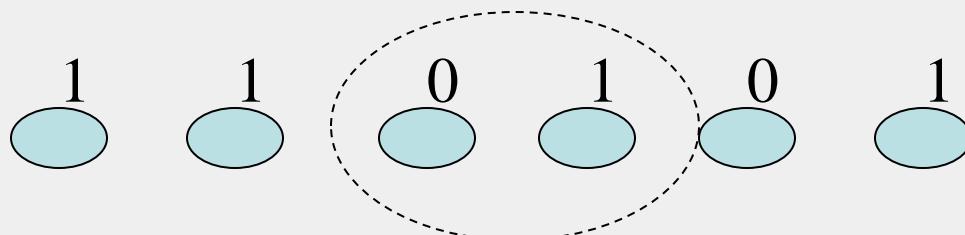
WHAT THE FLP PROOF SHOWS

1. There exists an initial configuration that is bivalent
2. Starting from a bivalent config., there is always another bivalent config. that is reachable

LEMMA 2

Some initial configuration is bivalent

- Suppose all initial configurations were either 0-valent or 1-valent.
- If there are N processes, there are 2^N possible initial configurations
- Place all configurations side-by-side (in a lattice), where adjacent configurations differ in initial x_p value for exactly one process.

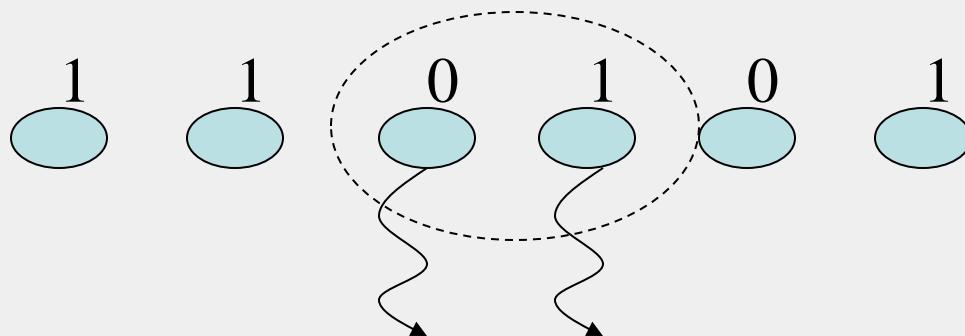


- There has to be **some** adjacent pair of 1-valent and 0-valent configs.

LEMMA 2

Some initial configuration is bivalent

- There has to be **some** adjacent pair of 1-valent and 0-valent configs.
- Let the process p, that has a different state across these two configs., be the process that has crashed (i.e., is silent throughout)



Both initial configs. will lead to the same config. for the same sequence of events

Therefore, both these initial configs. are bivalent when there is such a failure

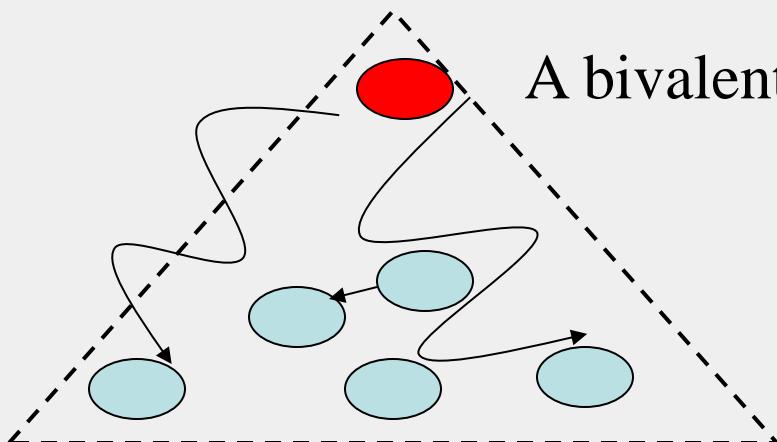
WHAT WE'LL SHOW

1. There exists an initial configuration that is bivalent
2. Starting from a bivalent config., there is always another bivalent config. that is reachable

LEMMA 3

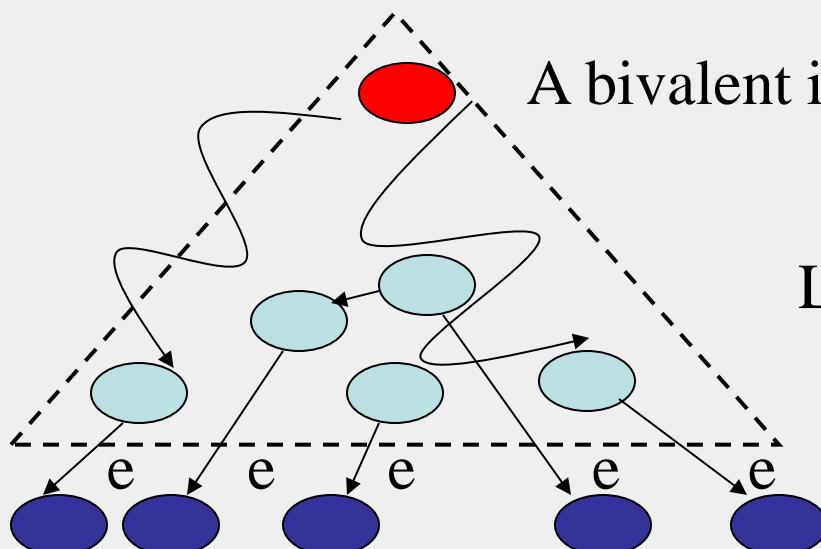
Starting from a bivalent config., there is always another bivalent config. that is reachable

LEMMA 3



let $e=(p,m)$ be some event applicable to the initial config.
Let \mathcal{C} be the set of configs. reachable **without** applying e

LEMMA 3



A bivalent initial config.

let $e=(p,m)$ be some event

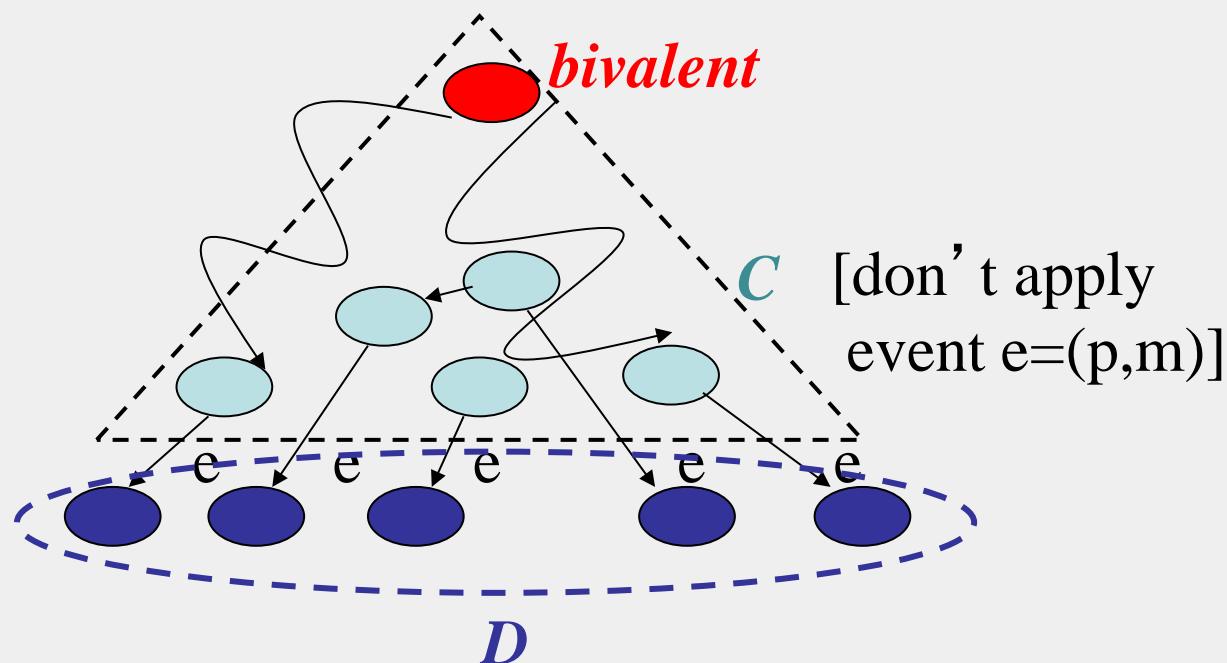
applicable to the initial config.

Let C be the set of configs. reachable
without applying e

Let D be the set of configs.

obtained by **applying** e to some
config. in C

LEMMA 3



Claim. Set D contains a bivalent config.

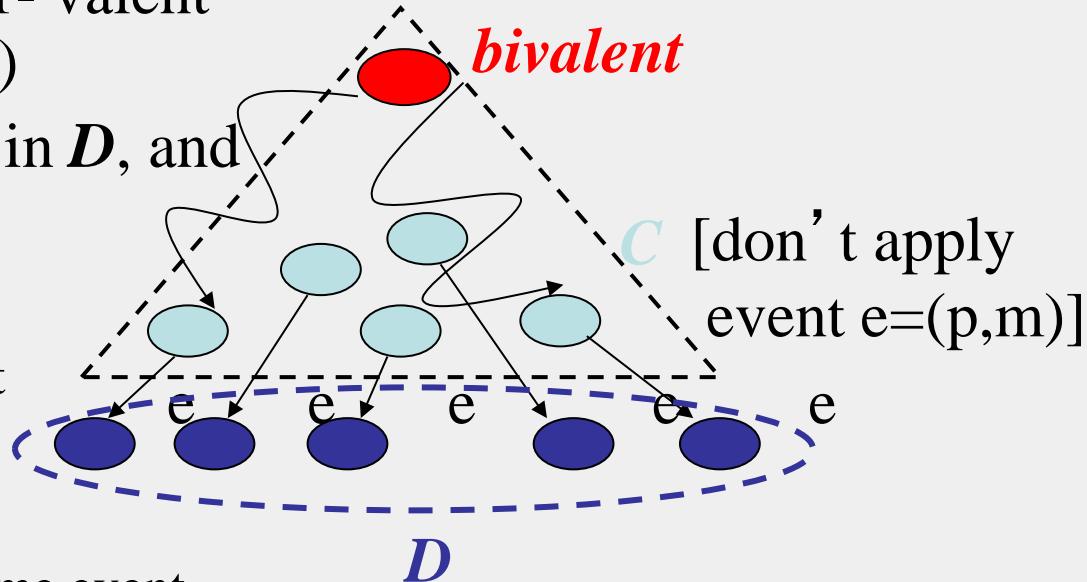
Proof. By contradiction. That is,

suppose D has only 0- and 1- valent states (and no bivalent ones)

- There are states D_0 and D_1 in D , and C_0 and C_1 in C such that

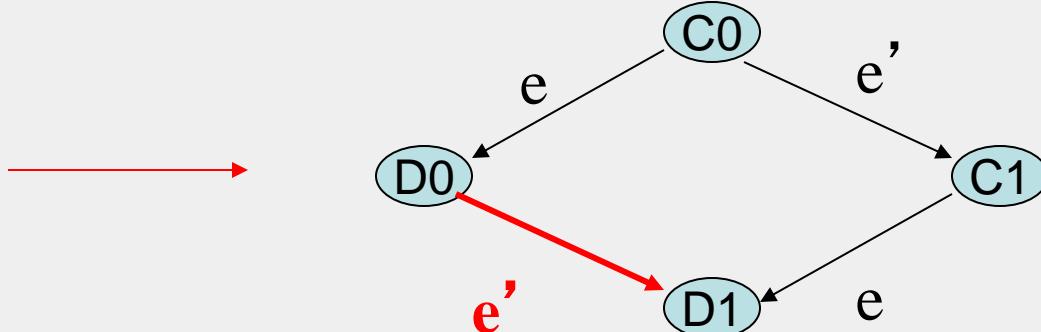
- D_0 is 0-valent, D_1 is 1-valent
- $D_0=C_0$ foll. by $e=(p,m)$
- $D_1=C_1$ foll. by $e=(p,m)$
- And $C_1 = C_0$ followed by some event $e'=(p',m')$

(why?)

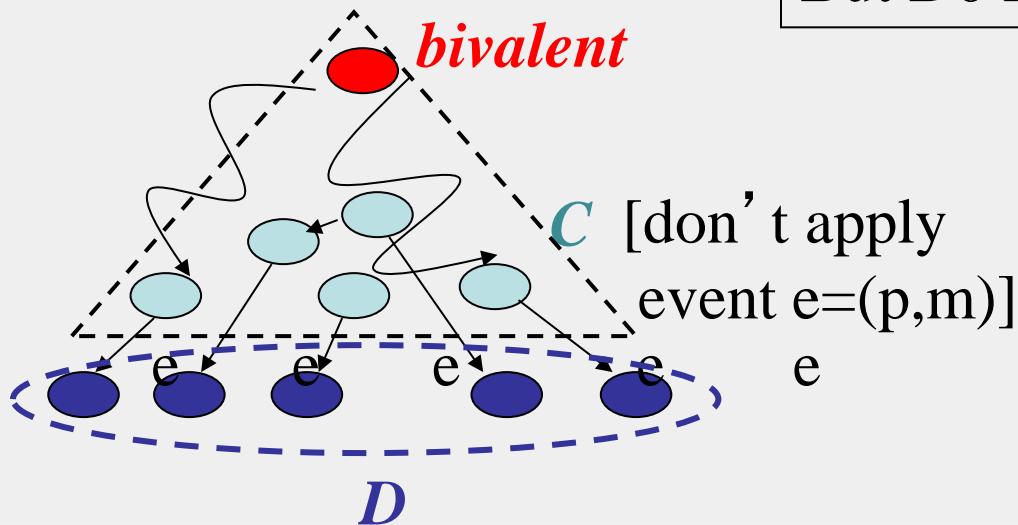


Proof. (contd.)

- Case I: p' is not p
- Case II: p' same as p

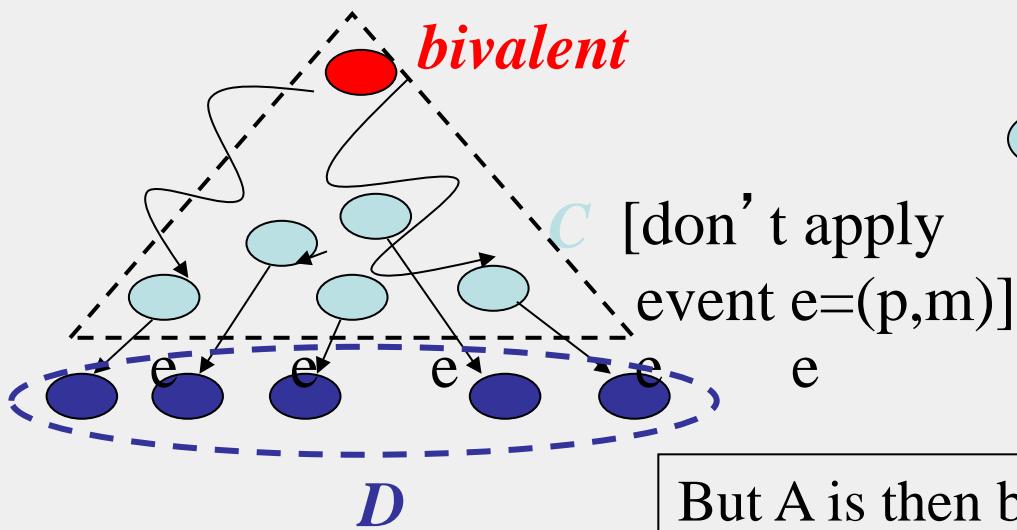
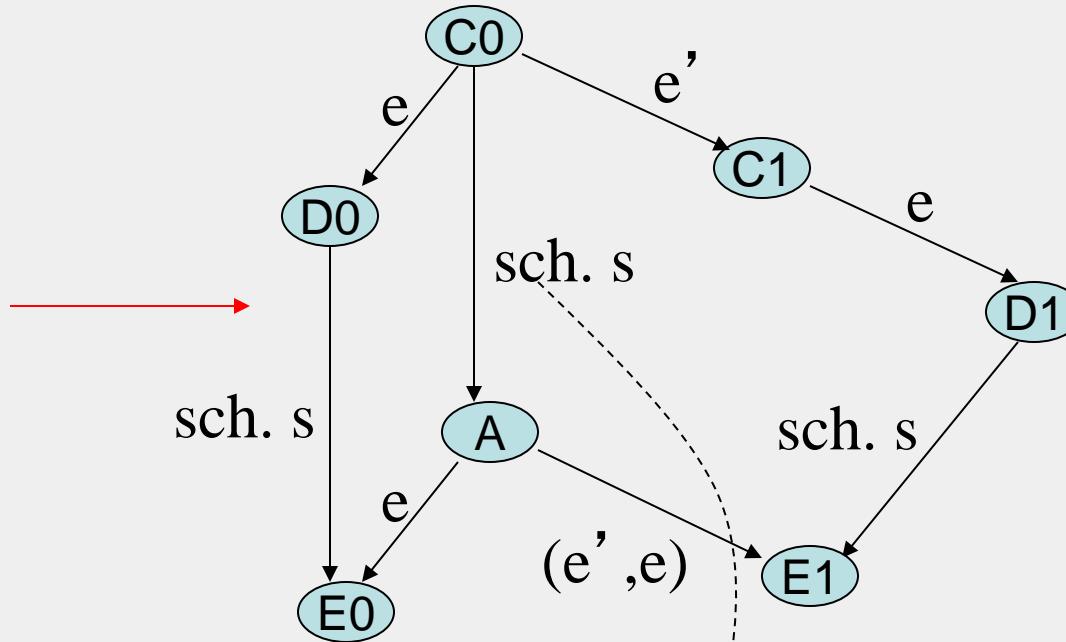


Why? (Lemma 1)
But D_0 is then bivalent!



Proof. (contd.)

- Case I: p' is not p
- Case II: p' same as p



But A is then bivalent!

- finite
- **deciding run from C_0**
- p takes no steps

LEMMA 3

Starting from a bivalent config., there is always another bivalent config. that is reachable

PUTTING IT ALL TOGETHER

- Lemma 2: There exists an initial configuration that is bivalent
- Lemma 3: Starting from a bivalent config., there is always another bivalent config. that is reachable
- Theorem (Impossibility of Consensus): **There is always a run of events in an asynchronous distributed system such that the group of processes never reaches consensus (i.e., stays bivalent all the time)**

SUMMARY

- Consensus problem
 - Agreement in distributed systems
 - Solution exists in synchronous system model (e.g., supercomputer)
 - Impossible to solve in an asynchronous system (e.g., Internet, Web)
 - Key idea: with even one (adversarial) crash-stop process failure, there are always sequences of events for the system to decide any which way
 - Holds true regardless of whatever algorithm you choose!
 - FLP impossibility proof
- One of the most fundamental results in distributed systems

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SNAPSHOTS

Lecture A

WHAT IS GLOBAL SNAPSHOT

HERE'S A SNAPSHOT



DISTRIBUTED SNAPSHOT

- More often, each country's representative is sitting in their respective capital and sending messages to each other (say emails).
- How do you calculate a “global snapshot” in that distributed system?
- What does a “global snapshot” even mean?

IN THE CLOUD

- **In a cloud: each application or service is running on multiple servers**
- **Servers handling concurrent events and interacting with each other**
- **The ability to obtain a “global photograph” of the system is important**
- **Some uses of having a global picture of the system**
 - *Checkpointing*: can restart distributed application on failure
 - *Garbage collection* of objects: objects at servers that don't have any other objects (at any servers) with pointers to them
 - Deadlock detection: Useful in database transaction systems
 - Termination of computation: Useful in batch computing systems like Folding@Home, SETI@Home

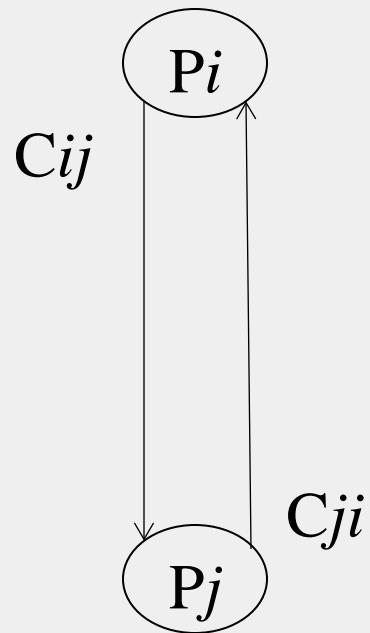
WHAT'S A GLOBAL SNAPSHOT?

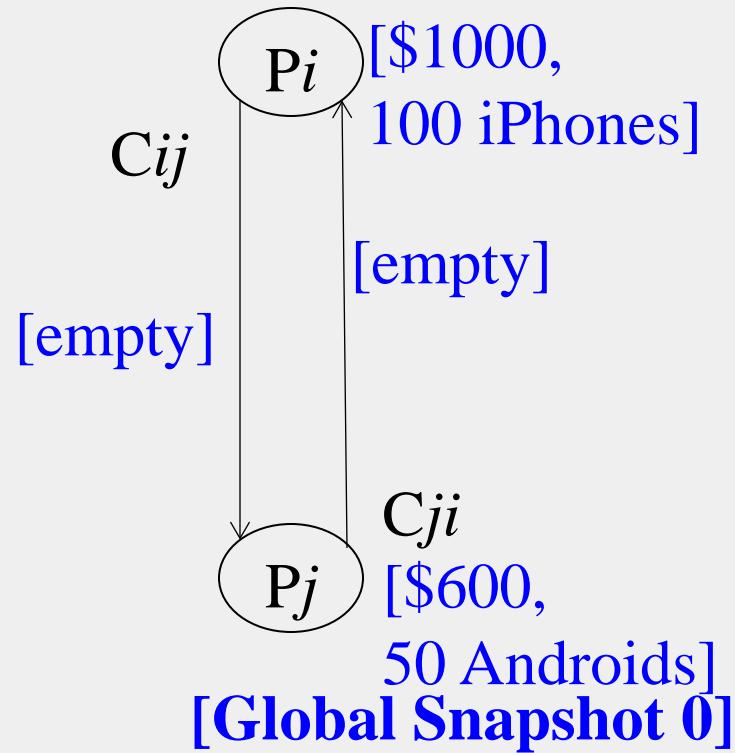
- **Global Snapshot = Global State =**
 - Individual state of each process in the distributed system
 - +
 - Individual state of each communication channel in the distributed system
- Capture the instantaneous state of each process
- And the instantaneous state of each communication channel, i.e., *messages* in transit on the channels

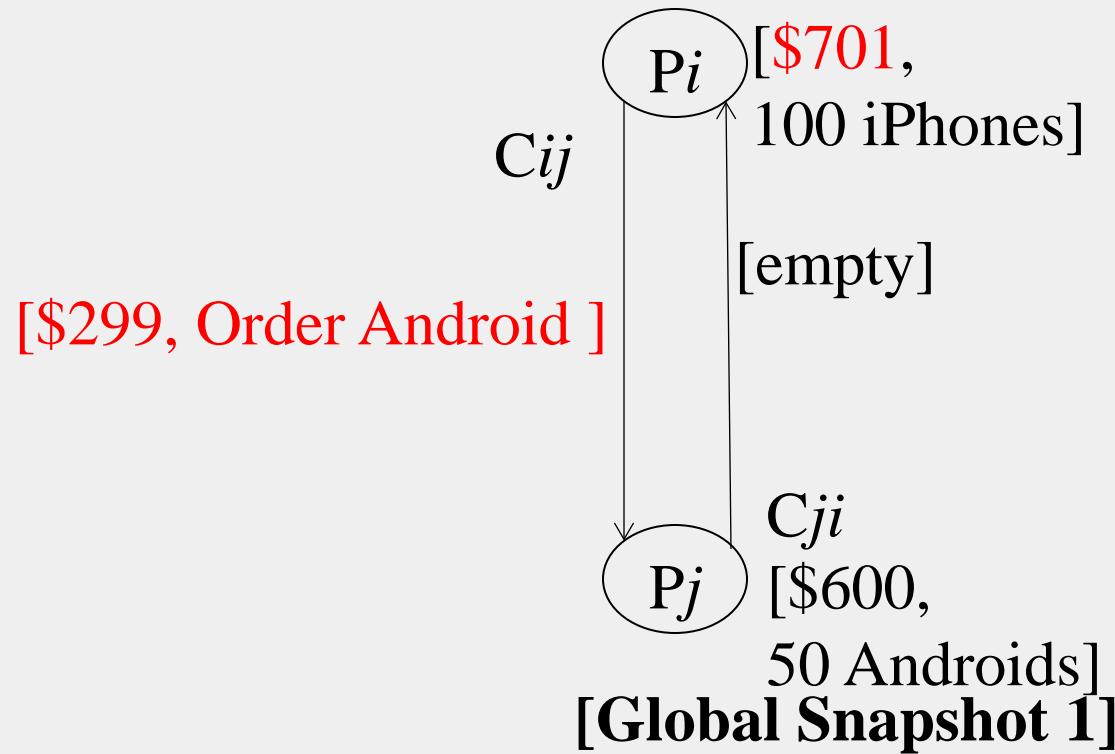
OBVIOUS FIRST SOLUTION

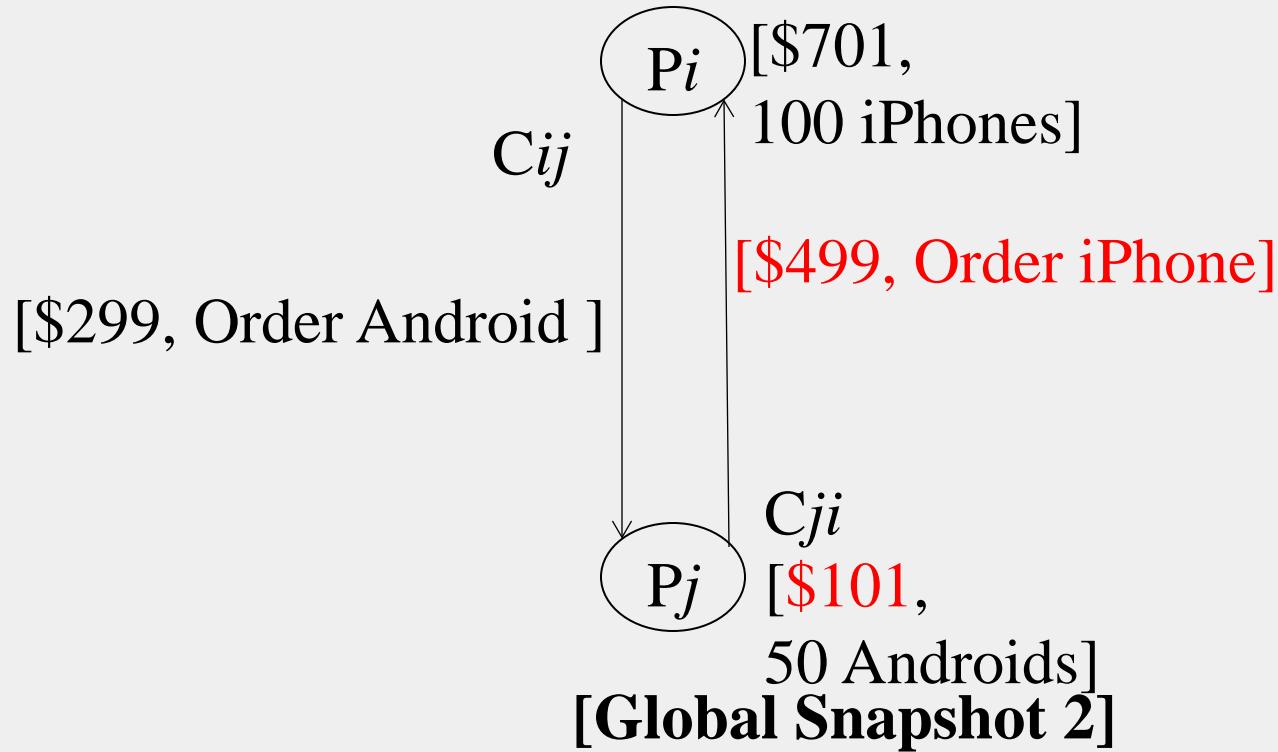
- Synchronize clocks of all processes
- Ask all processes to record their states at known time t
- Problems?
 - Time synchronization always has error
 - Your bank might inform you, “We lost the state of our distributed cluster due to a 1 ms clock skew in our snapshot algorithm.”
 - Also, does not record the state of messages in the channels
- Again: synchronization not required – causality is enough!

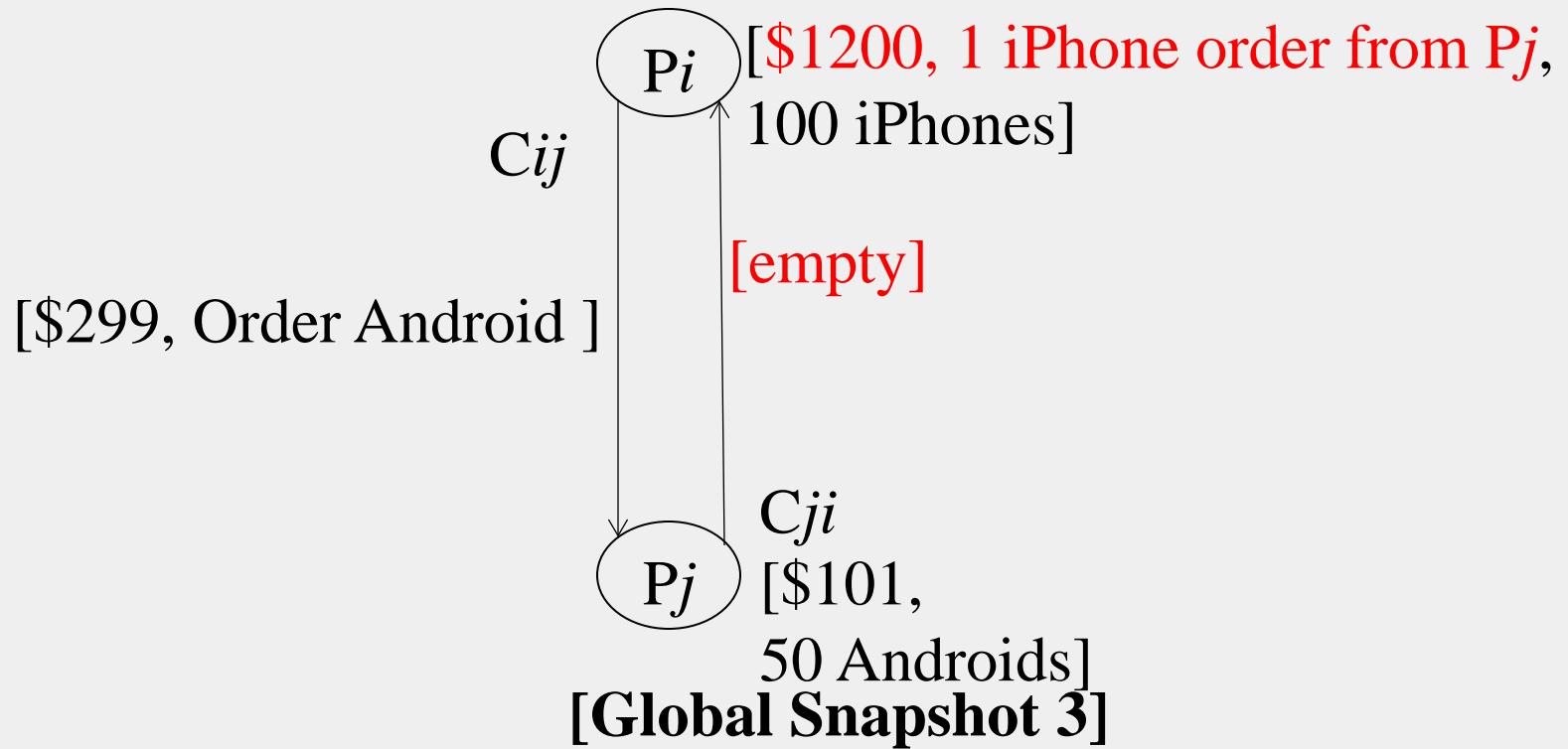
EXAMPLE

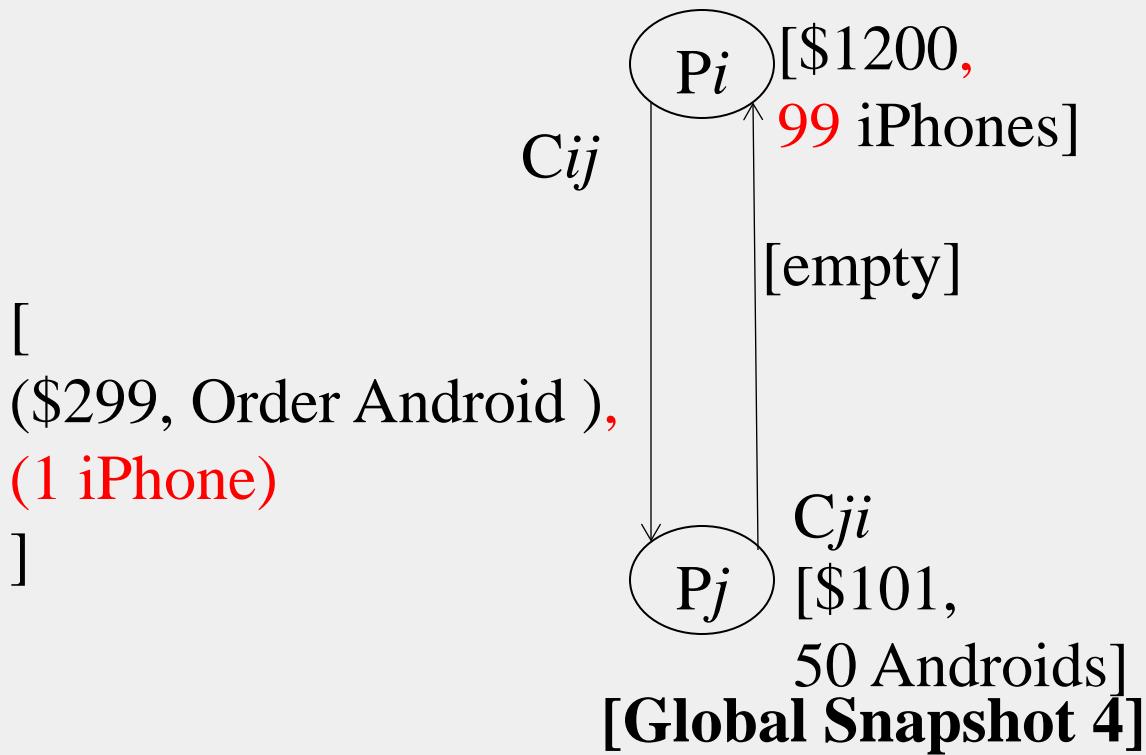




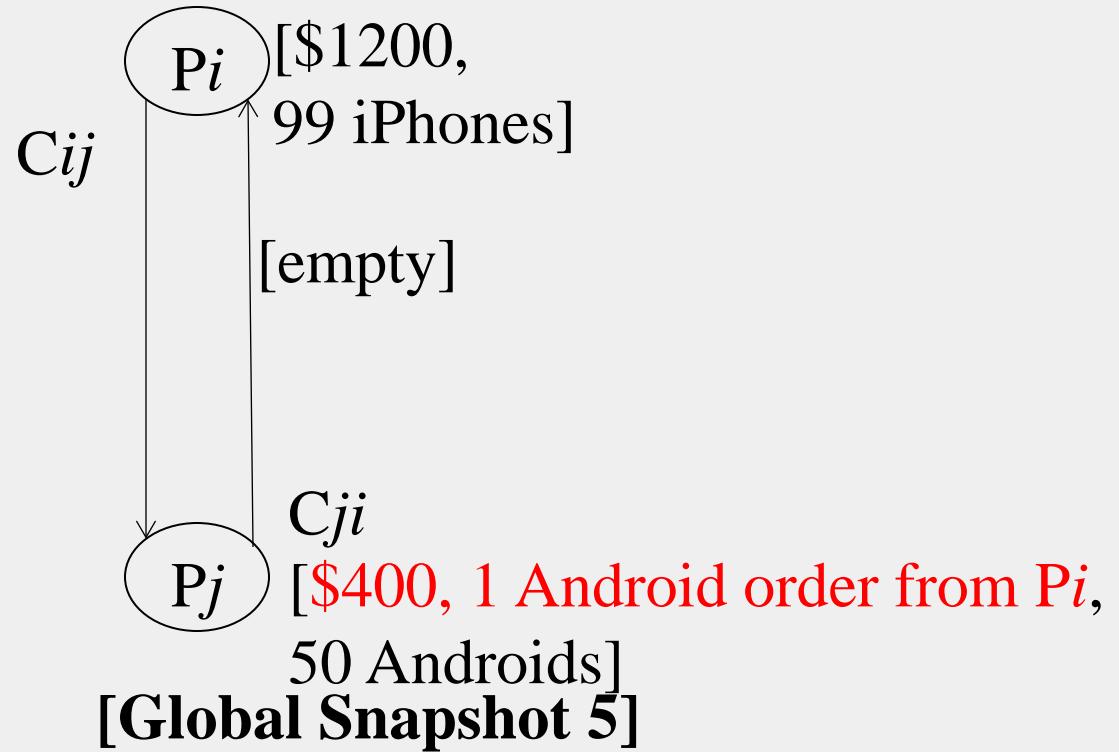


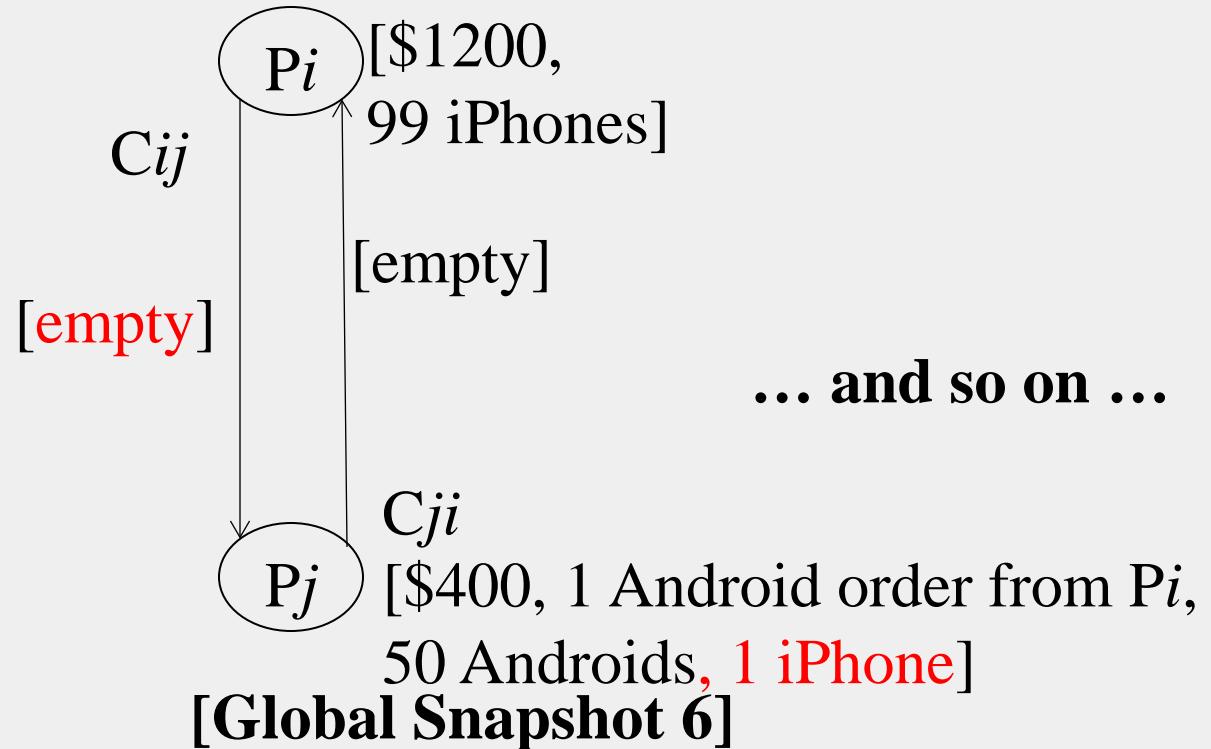






[
(1 iPhone)
]





MOVING FROM STATE TO STATE

- Whenever an event happens anywhere in the system, the global state changes
 - Process receives message
 - Process sends message
 - Process takes a step
- State to state movement obeys causality
 - Next: Causal algorithm for Global Snapshot calculation



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SNAPSHOTS

Lecture B

GLOBAL SNAPSHOT ALGORITHM

SYSTEM MODEL

- **Problem:** Record a global snapshot (state for each process, and state for each channel)
- *System Model:*
 - N processes in the system
 - There are two uni-directional communication channels between each ordered process pair : $P_j \rightarrow P_i$ and $P_i \rightarrow P_j$
 - Communication channels are FIFO-ordered
 - First in, first out
 - No failure
 - All messages arrive intact and are not duplicated
 - Other papers later relaxed some of these assumptions

REQUIREMENTS

- **Snapshot should not interfere with normal application actions, and it should not require application to stop sending messages**
- **Each process is able to record its own state**
 - Process state: Application-defined state or, in the worst case:
 - Its heap, registers, program counter, code, etc. (essentially the coredump)
- **Global state is collected in a distributed manner**
- **Any process may initiate the snapshot**
 - We'll assume just one snapshot run for now

CHANDY-LAMPORT GLOBAL SNAPSHOT ALGORITHM

- First, Initiator P_i records its own state
- Initiator process creates special messages called “Marker” messages
 - Not an application message, does not interfere with application messages
- for $j=1$ to N except i
 - P_i sends out a Marker message on outgoing channel C_{ij}
 - ($N-1$) channels
- Starts recording the incoming messages on each of the incoming channels at P_i : C_{ji} (for $j=1$ to N except i)

CHANDY-LAMPORT GLOBAL SNAPSHOT ALGORITHM (2)

Whenever a process P_i receives a Marker message on an incoming channel C_{ki}

- **if** (this is the first Marker P_i is seeing)
 - P_i records its own state first
 - Marks the state of channel C_{ki} as “empty”
 - For $j=1$ to N except i
 - P_i sends out a Marker message on outgoing channel C_{ij}
 - Starts recording the incoming messages on each of the incoming channels at P_i : C_{ji} (for $j=1$ to N except i and k)
- **else // already seen a Marker message**
 - Mark the state of channel C_{ki} as all the messages that have arrived on it since recording was turned on for C_{ki}

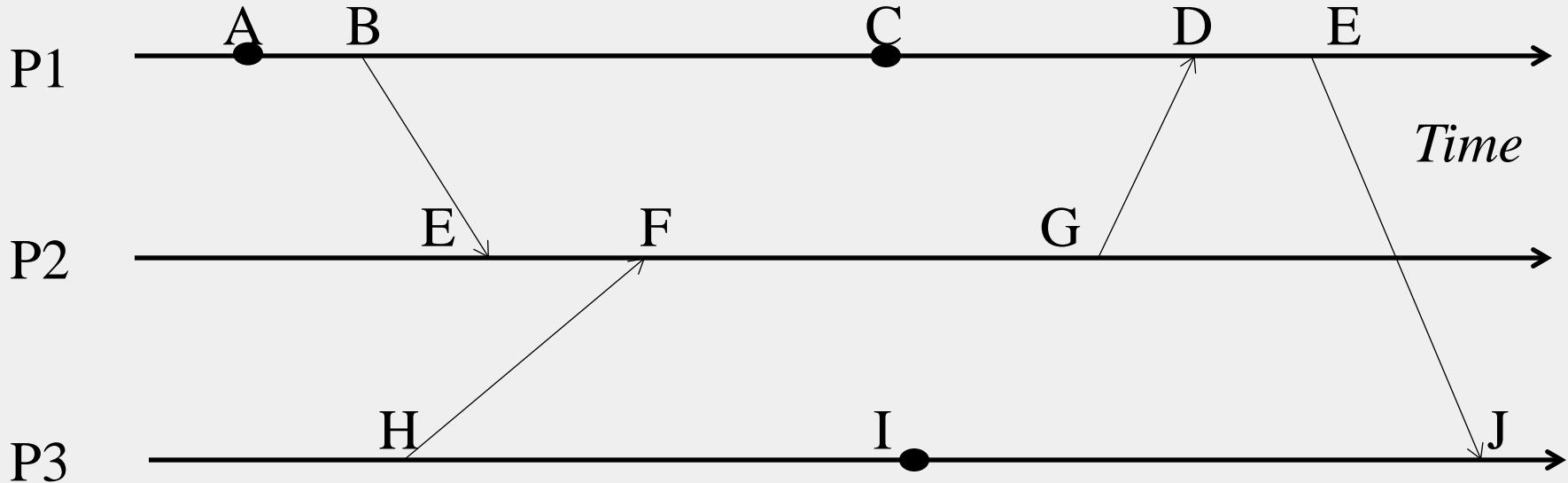
CHANDY-LAMPORT GLOBAL SNAPSHOT ALGORITHM (3)

The algorithm terminates when

- All processes have received a Marker
 - To record their own state
- All processes have received a Marker on all the ($N-1$) incoming channels at each
 - To record the state of all channels

Then, (if needed), a central server collects all these partial state pieces to obtain the full global snapshot

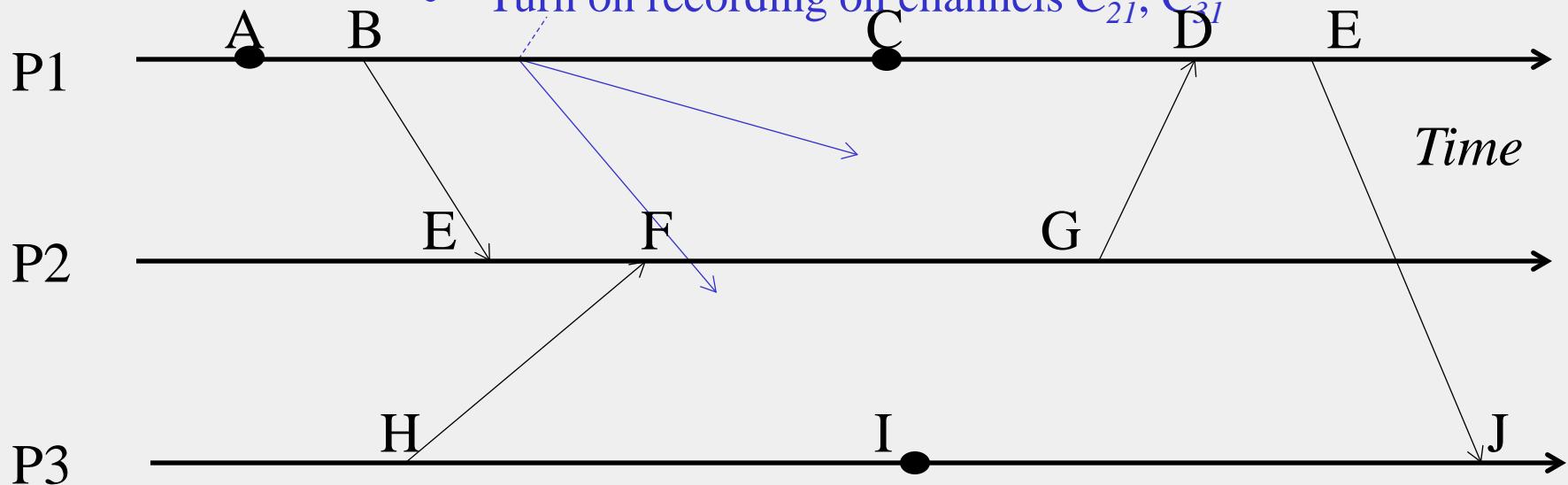
EXAMPLE

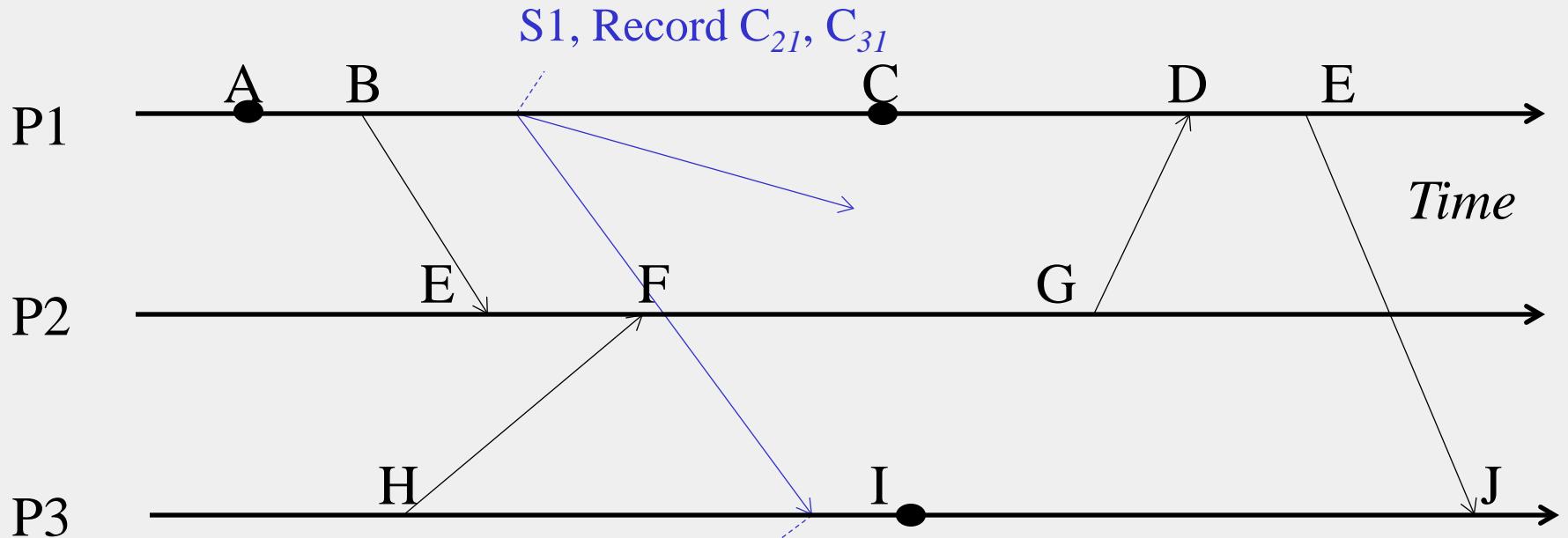


● *Instruction or Step*
→ *Message*

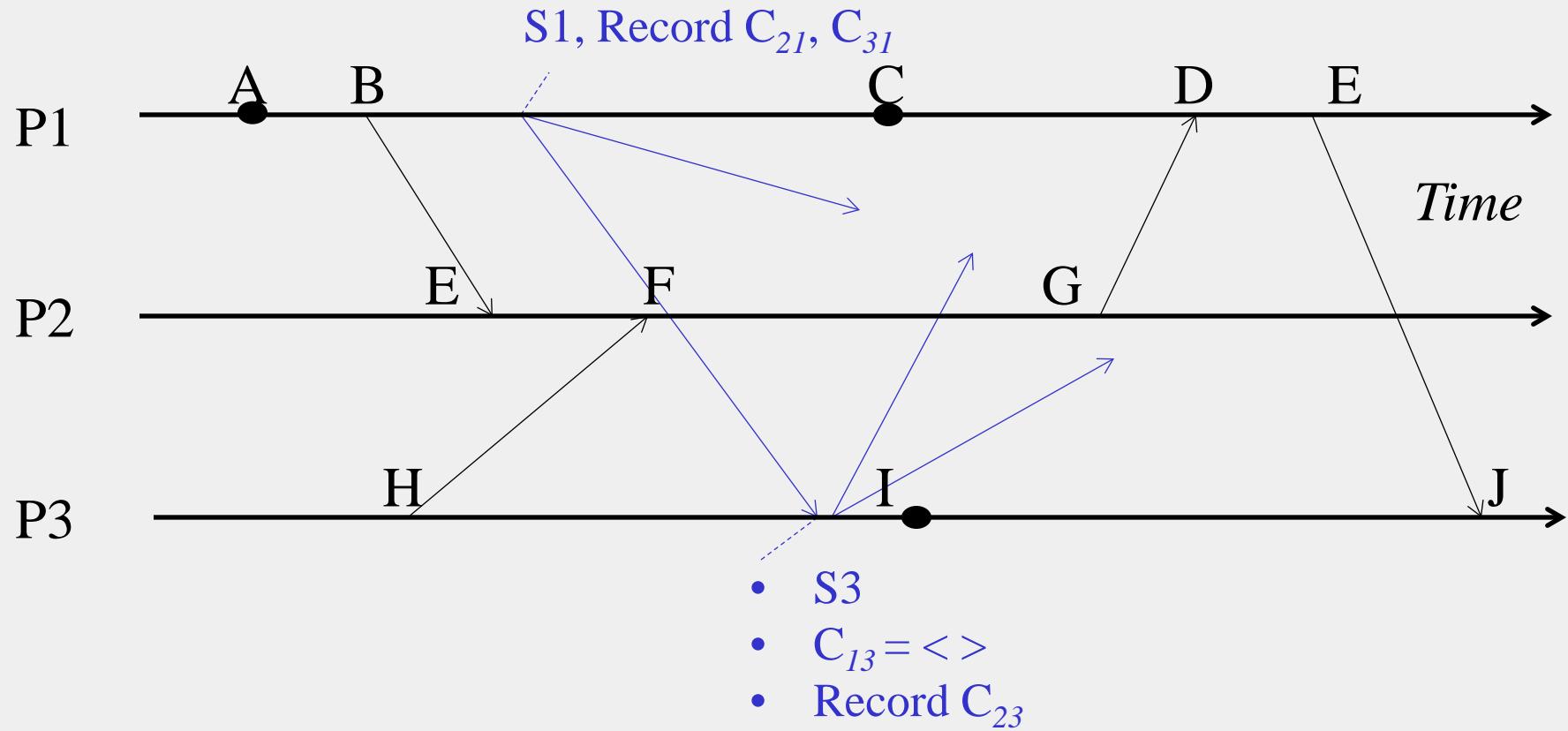
P1 is Initiator:

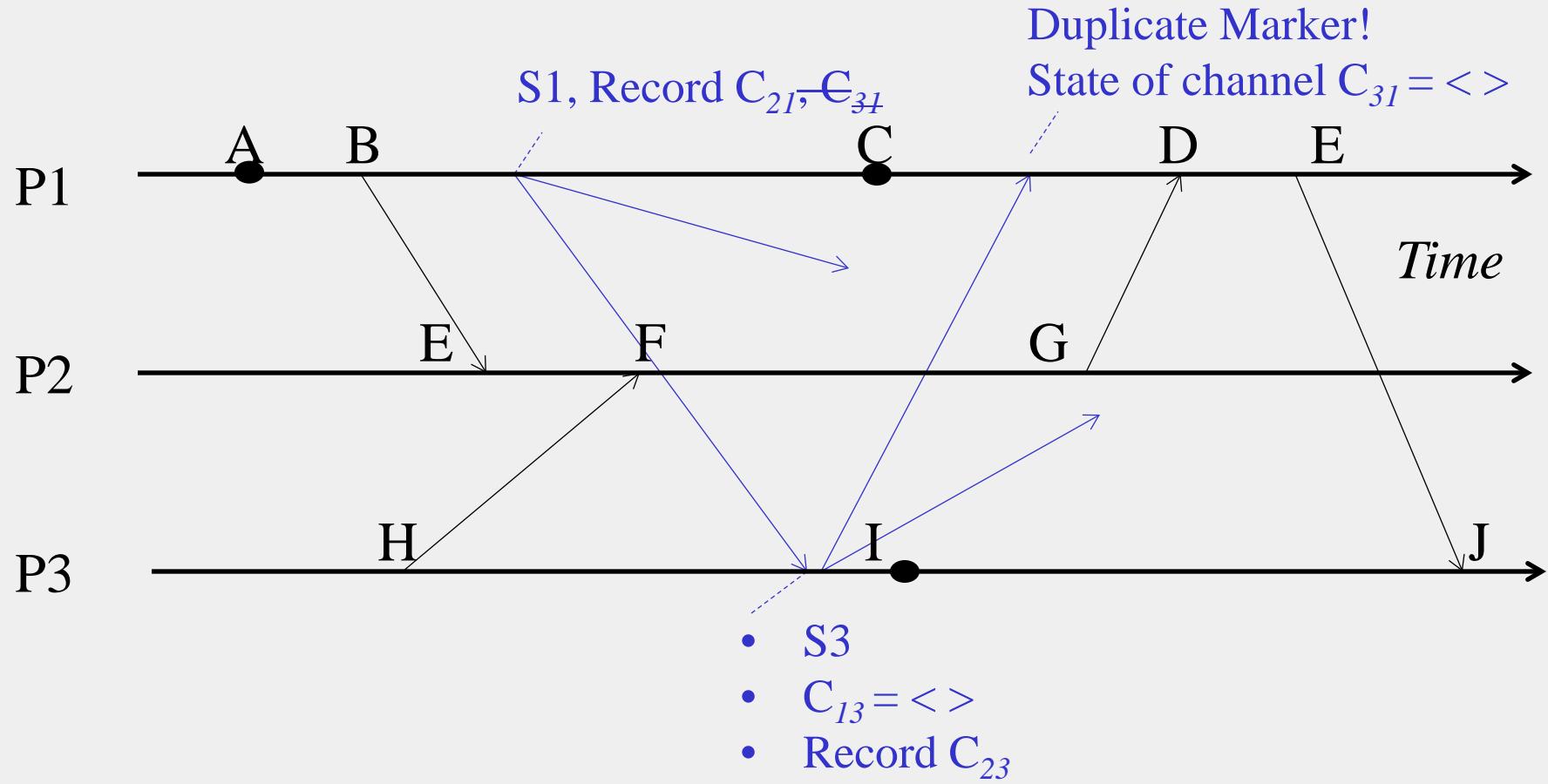
- Record local state S1
- Send out markers
- Turn on recording on channels C_{21}, C_{31}

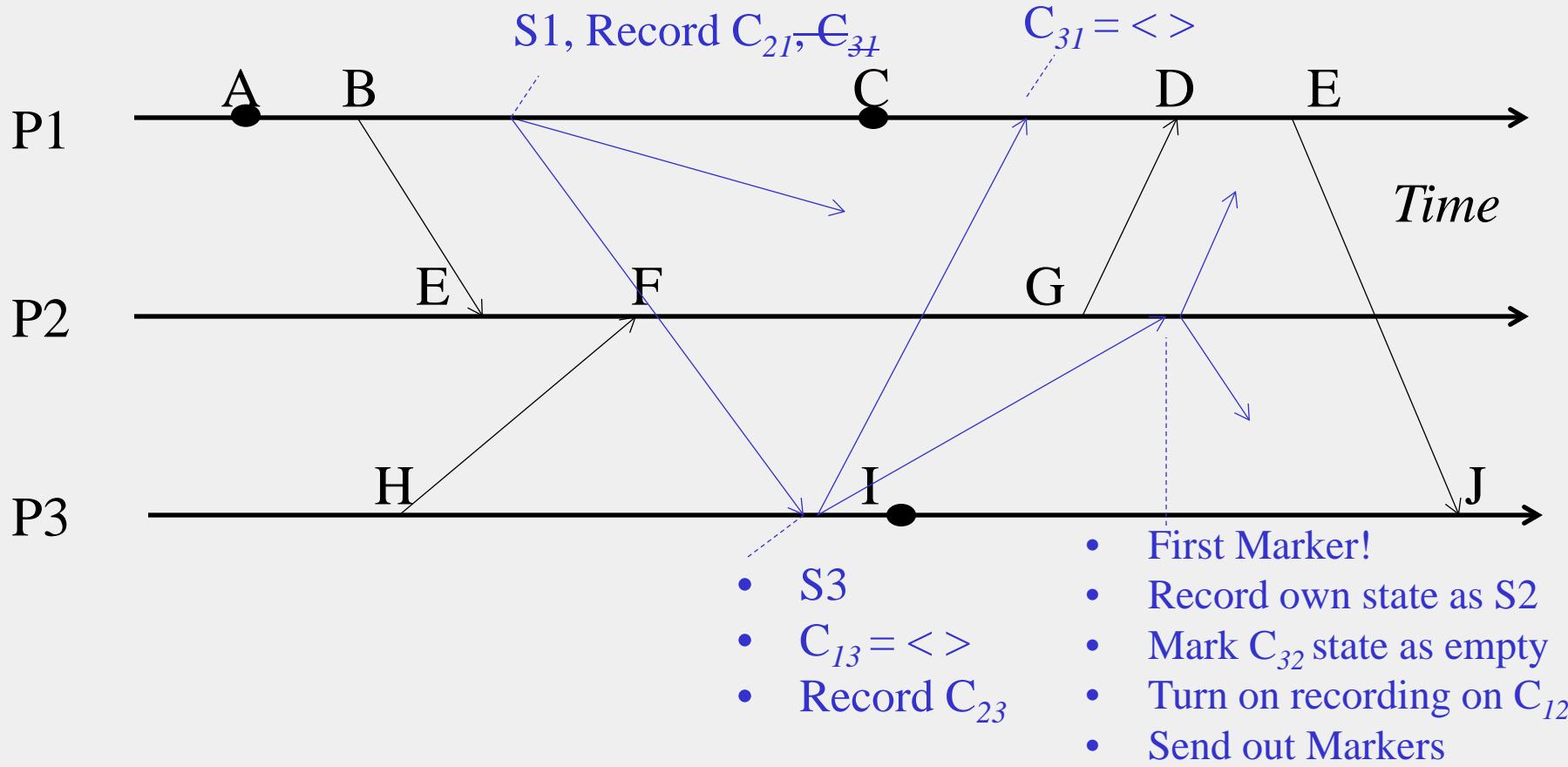


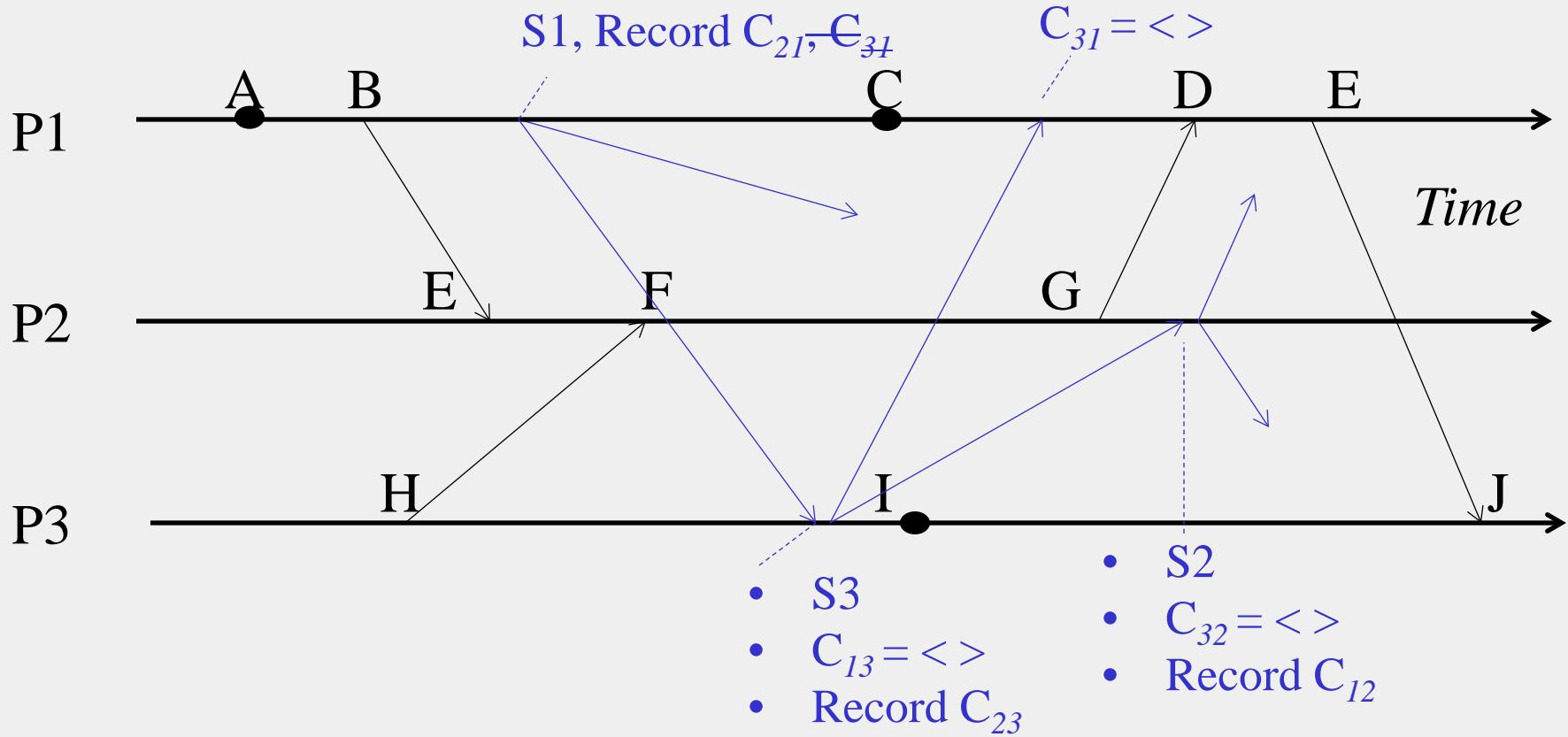


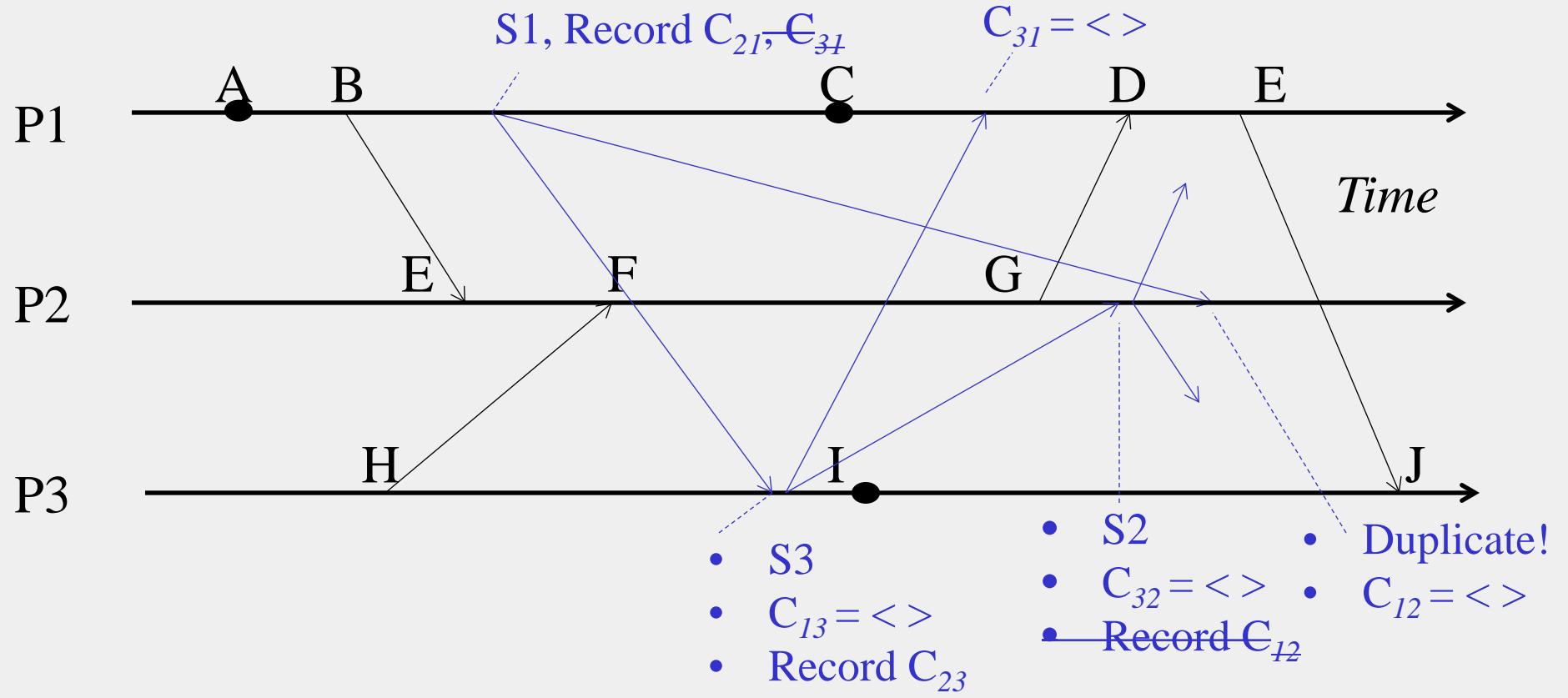
- First Marker!
- Record own state as S3
- Mark C_{13} state as empty
- Turn on recording on other incoming C_{23}
- Send out Markers

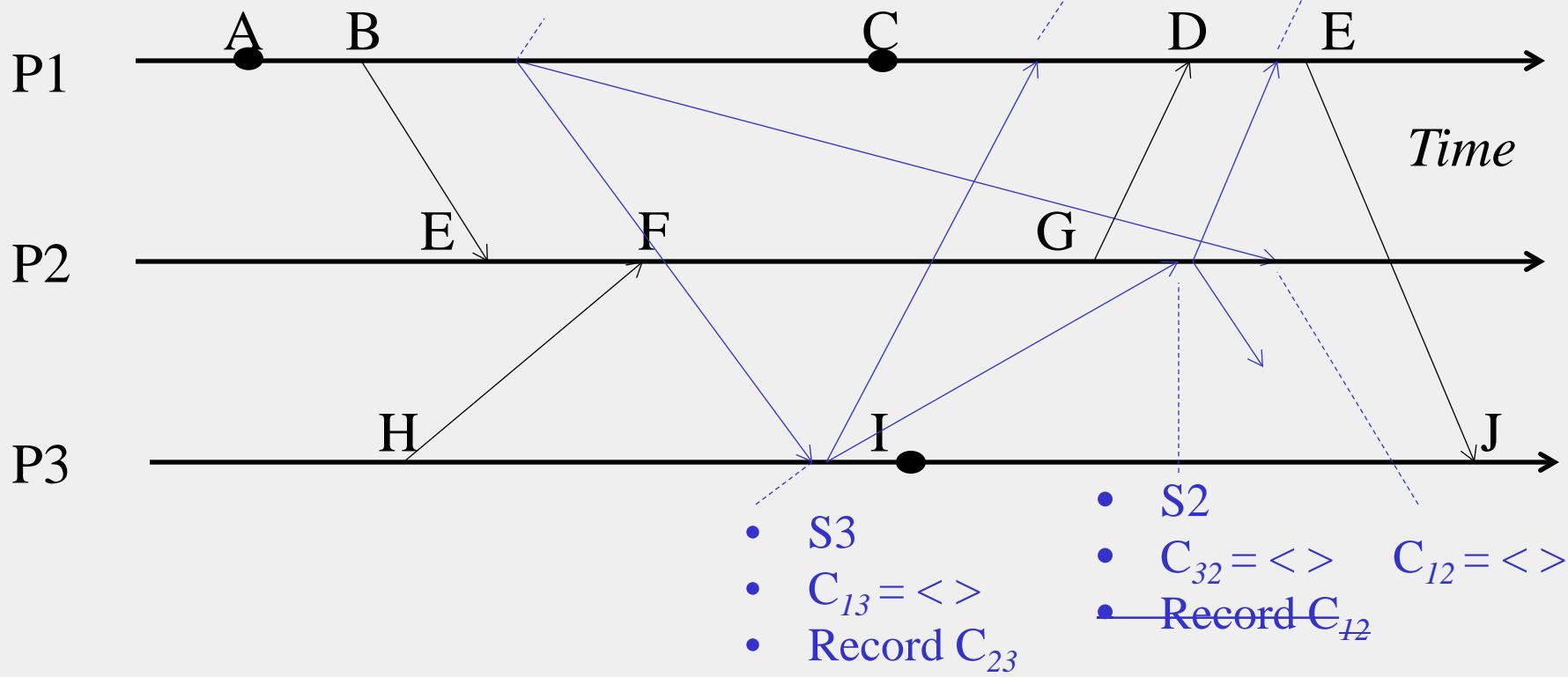


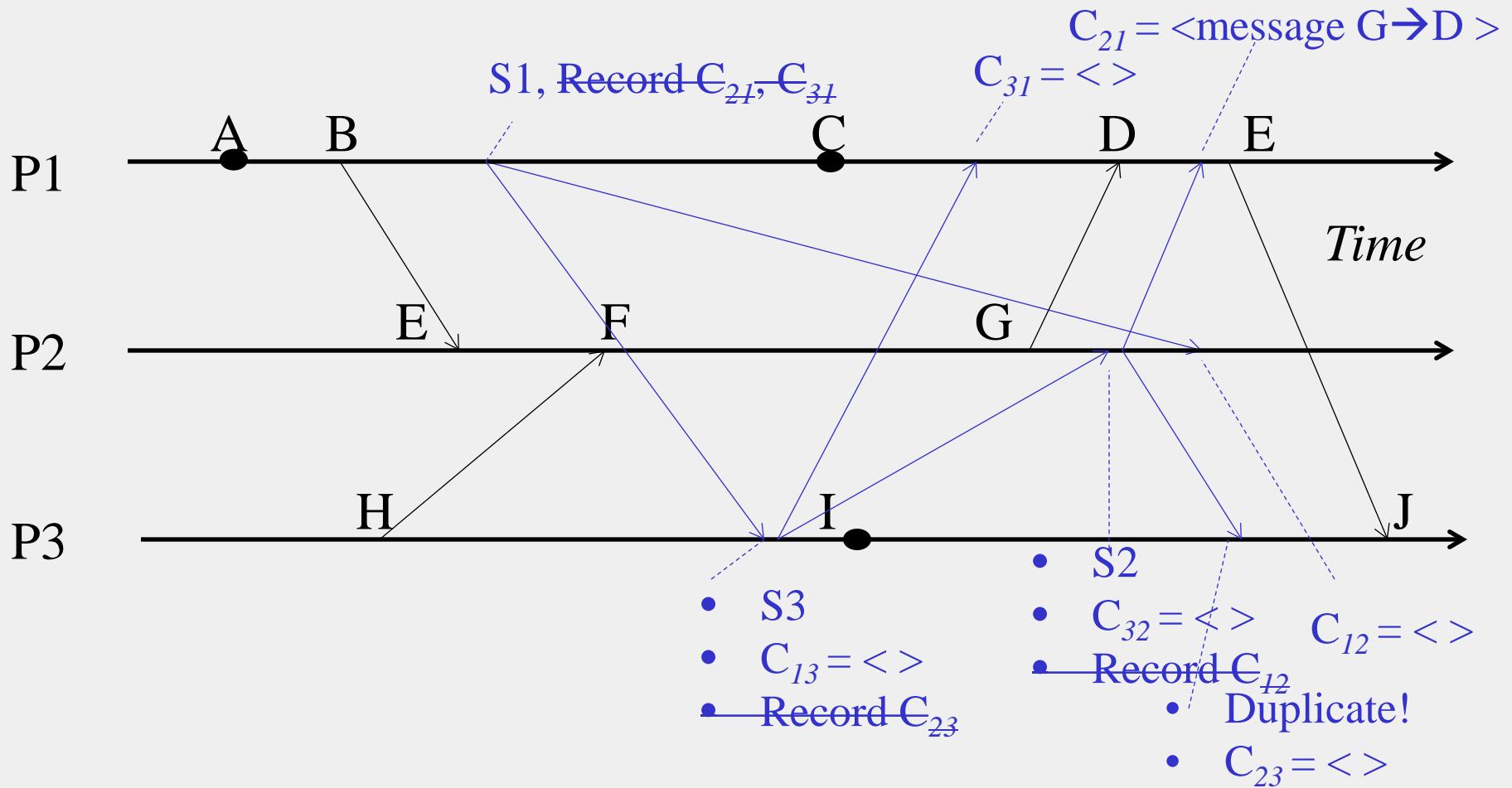




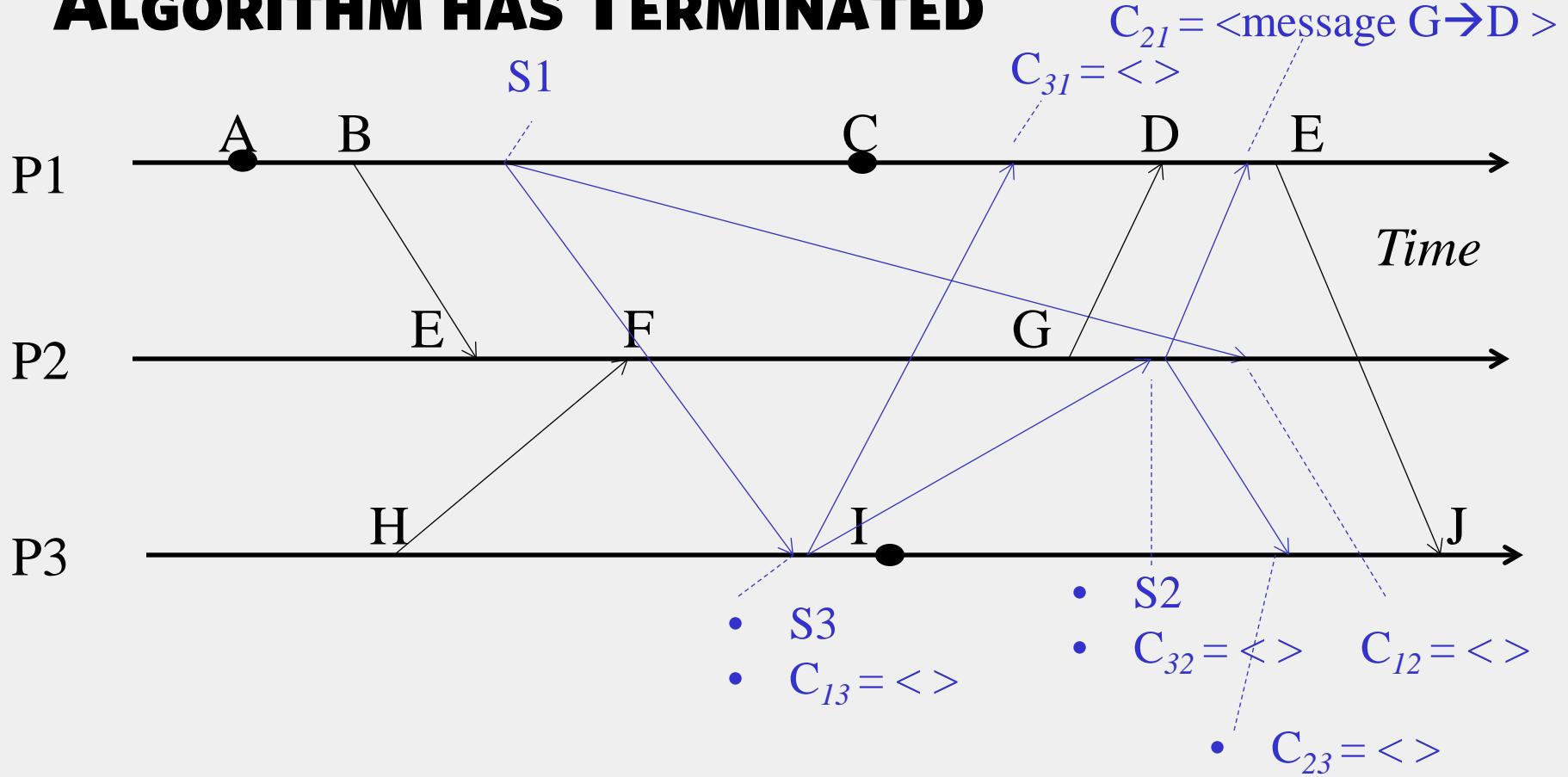




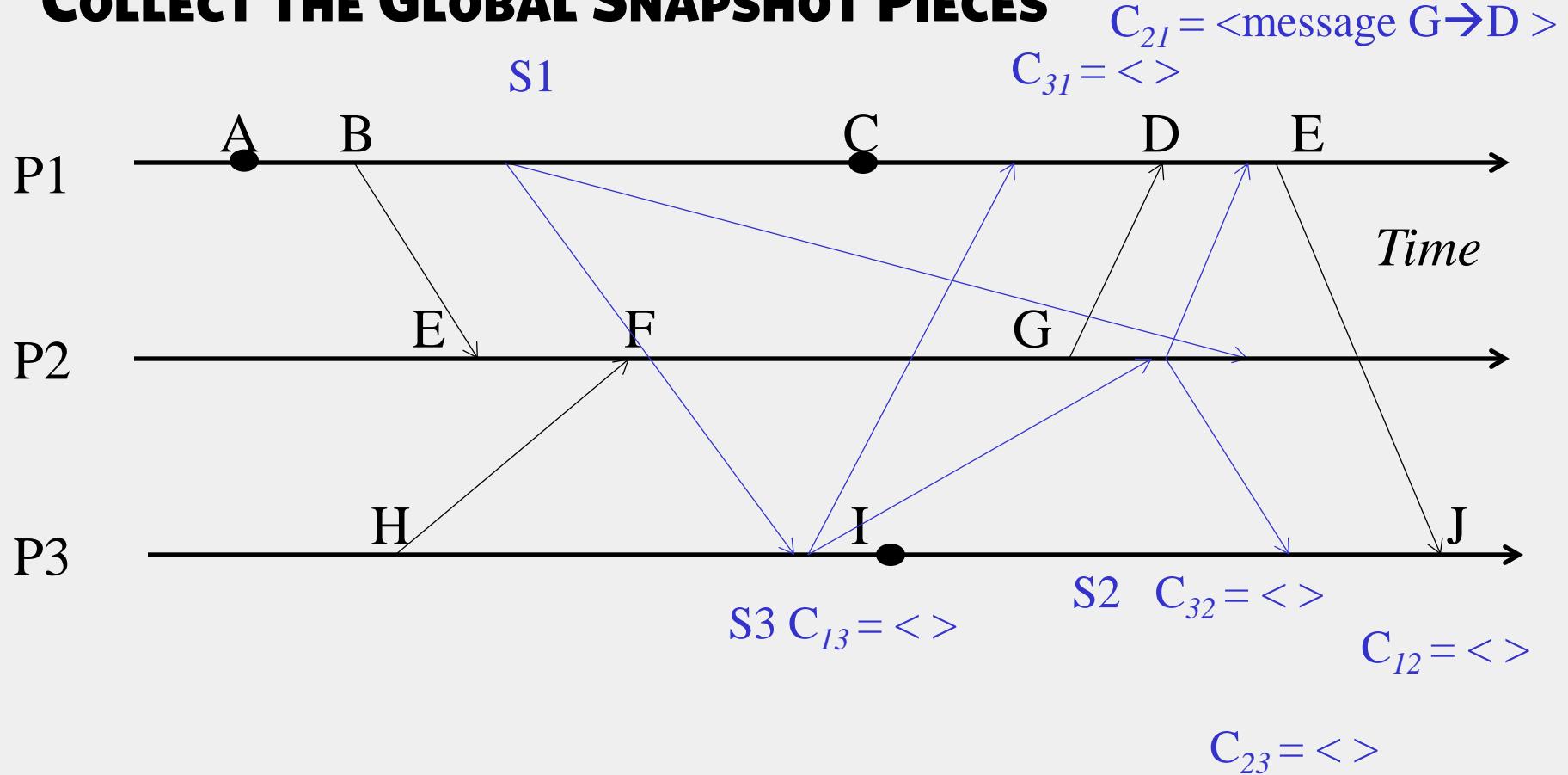




ALGORITHM HAS TERMINATED



COLLECT THE GLOBAL SNAPSHOT PIECES



NEXT

- Global Snapshot calculated by Chandy-Lamport algorithm is causally correct
 - What?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SNAPSHOTS

Lecture C

CONSISTENT CUTS

CUTS

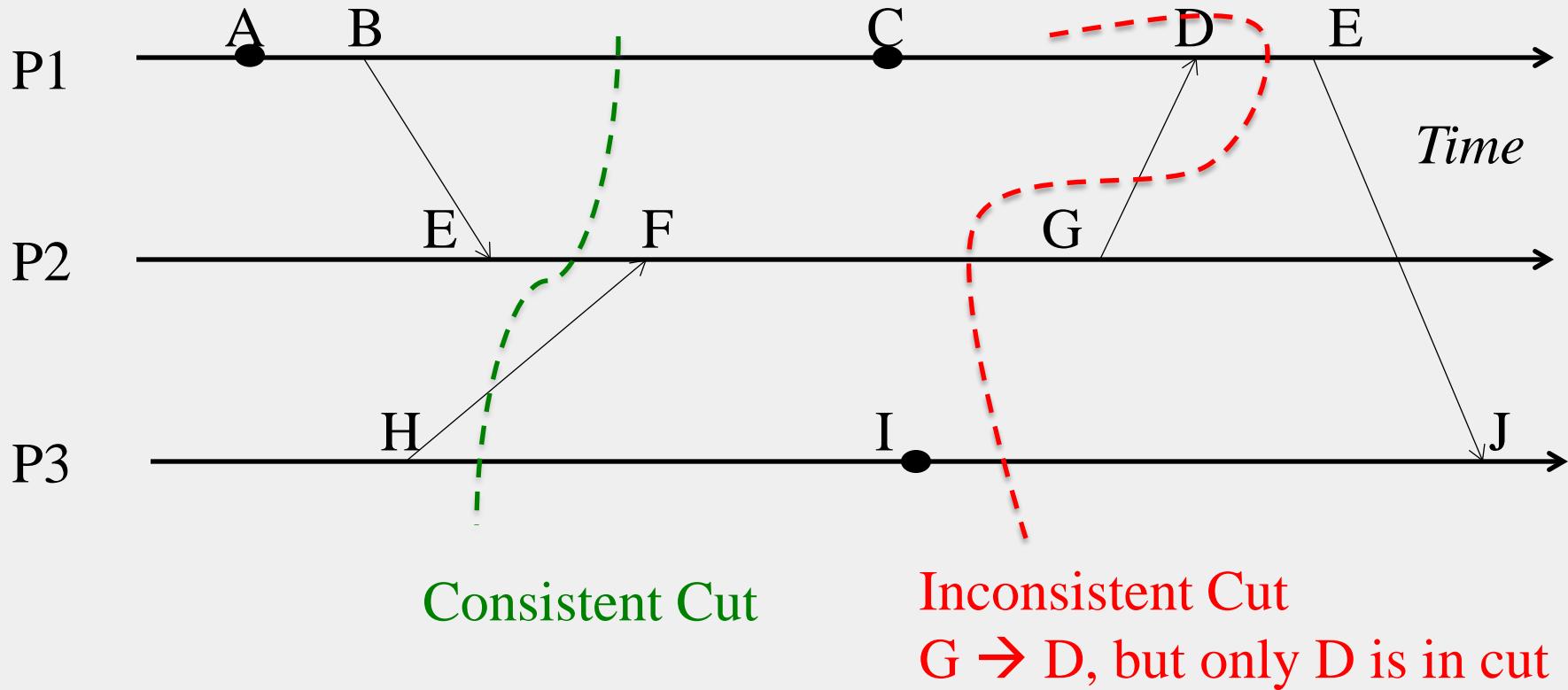
- **Cut** = time frontier at each process and at each channel
- Events at the process/channel that happen before the cut are “in the cut”
 - And happening after the cut are “out of the cut”

CONSISTENT CUTS

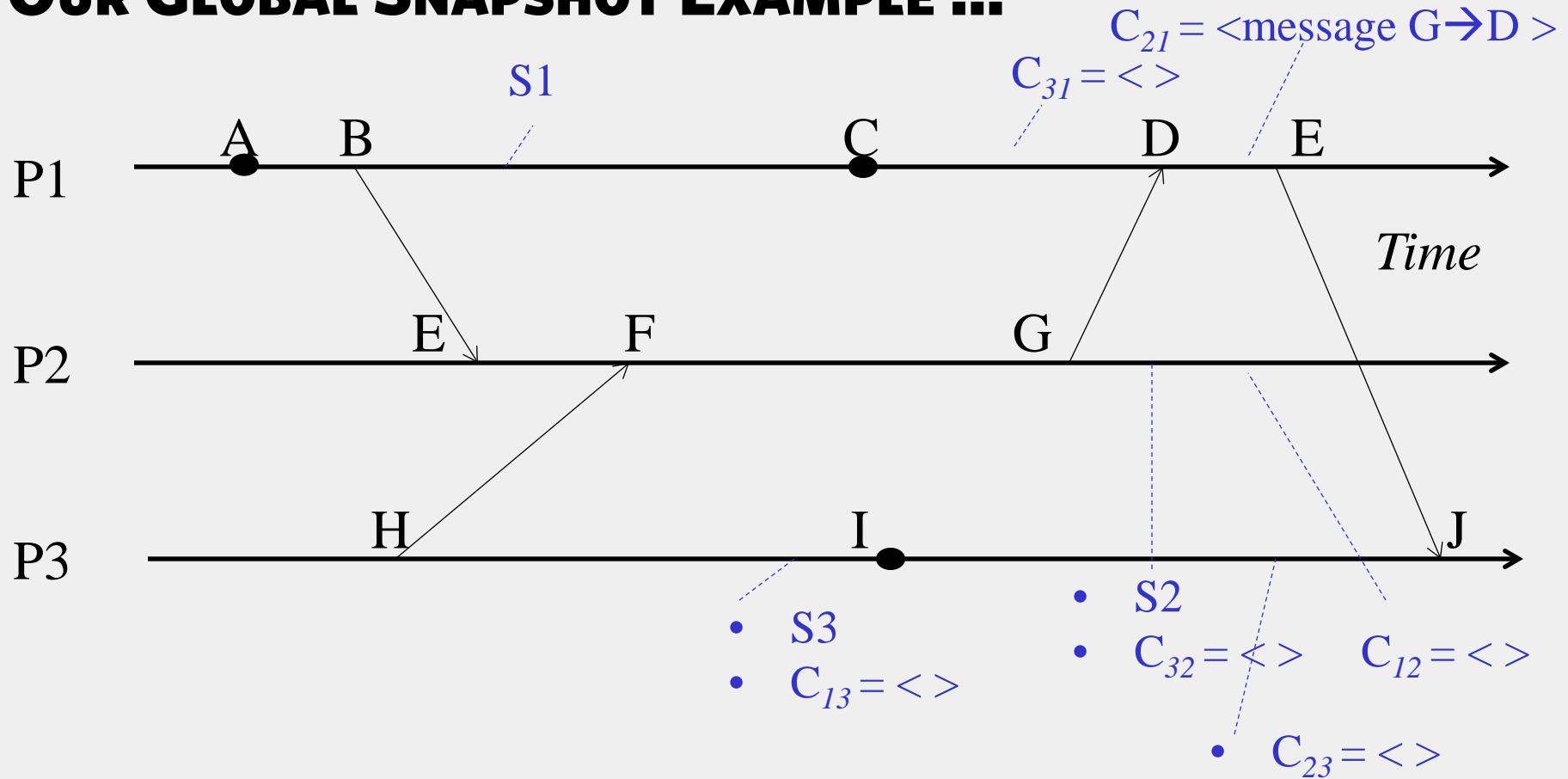
Consistent Cut: a cut that obeys causality

- A cut C is a consistent cut if and only if:
for (each pair of events e, f in the system)
 - Such that event e is in the cut C, and if $f \rightarrow e$ (f happens-before e)
 - Then: Event f is also in the cut C

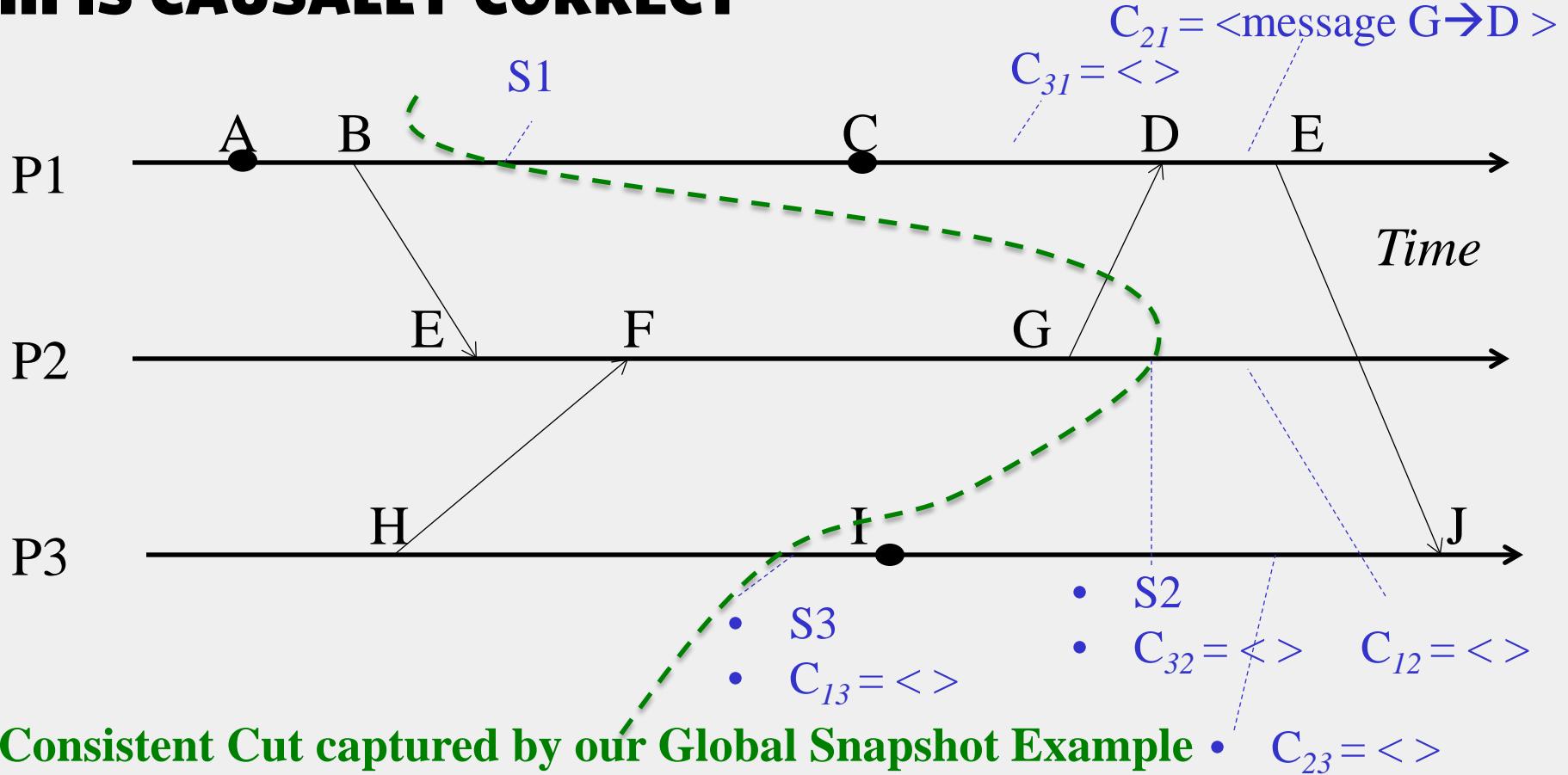
EXAMPLE



OUR GLOBAL SNAPSHOT EXAMPLE ...



... IS CAUSALLY CORRECT



IN FACT...

- Any run of the Chandy-Lamport Global Snapshot algorithm creates a consistent cut

CHANDY-LAMPORT GLOBAL SNAPSHOT ALGORITHM CREATES A CONSISTENT CUT

Let's quickly look at the proof

- Let e_i and e_j be events occurring at P_i and P_j , respectively, such that
 - $e_i \rightarrow e_j$ (e_i happens before e_j)
- The snapshot algorithm ensures that
 - if e_j is in the cut then e_i is also in the cut.
- That is: if $e_j \rightarrow \langle P_j \text{ records its state} \rangle$, then
 - It must be true that $e_i \rightarrow \langle P_i \text{ records its state} \rangle$.

CHANDY-LAMPORT GLOBAL SNAPSHOT

ALGORITHM CREATES A CONSISTENT CUT

- If $e_j \rightarrow \langle P_j \text{ records its state} \rangle$, then it must be true that $e_i \rightarrow \langle P_i \text{ records its state} \rangle$.
 - By contradiction, suppose $e_j \rightarrow \langle P_j \text{ records its state} \rangle$ and $\langle P_i \text{ records its state} \rangle \rightarrow e_i$
 - Consider the path of app messages (through other processes) that go from $e_i \rightarrow e_j$
 - Due to FIFO ordering, markers on each link in above path will precede regular app messages
 - Thus, since $\langle P_i \text{ records its state} \rangle \rightarrow e_i$, it must be true that P_j received a marker before e_j
 - Thus e_j is not in the cut \Rightarrow contradiction

NEXT

- What is the Chandy-Lamport algorithm used for?

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

SNAPSHOTS

Lecture D

SAFETY AND LIVENESS

"CORRECTNESS" IN DISTRIBUTED SYSTEMS

- Can be seen in two ways
- Liveness and Safety
- Often confused – it's important to distinguish from each other

LIVENESS

- Liveness = guarantee that something good will happen, eventually
 - Eventually == does not imply a time bound, but if you let the system run long enough, then ...

LIVENESS: EXAMPLES

- **Liveness = guarantee that something good will happen, eventually**
 - Eventually == does not imply a time bound, but if you let the system run long enough, then ...
- **Examples in Real World**
 - Guarantee that “at least one of the athletes in the 100m final will win gold” is liveness
 - A criminal will eventually be jailed
- **Examples in a Distributed System**
 - Distributed computation: Guarantee that it will terminate
 - “Completeness” in failure detectors: every failure is eventually detected by some non-faulty process
 - In Consensus: All processes eventually decide on a value

SAFETY

- Safety = guarantee that something **bad** will **never** happen

SAFETY: EXAMPLES

- **Safety** = guarantee that something **bad** will **never** happen
- **Examples in Real World**
 - A peace treaty between two nations provides safety
 - War will never happen
 - An innocent person will never be jailed
- **Examples in a Distributed System**
 - There is no deadlock in a distributed transaction system
 - No object is orphaned in a distributed object system
 - “Accuracy” in failure detectors
 - In Consensus: No two processes decide on different values

CAN'T WE GUARANTEE BOTH?

- **Can be difficult to satisfy both liveness and safety in an asynchronous distributed system!**
 - Failure Detector: Completeness (Liveness) and Accuracy (Safety) cannot both be guaranteed by a failure detector in an asynchronous distributed system
 - Consensus: Decisions (Liveness) and correct decisions (Safety) cannot both be guaranteed by any consensus protocol in an asynchronous distributed system
 - Very difficult for legal systems (anywhere in the world) to guarantee that all criminals are jailed (Liveness) and no innocents are jailed (Safety)

IN THE LANGUAGE OF GLOBAL STATES

- Recall that a distributed system moves from one global state to another global state, via causal steps
- Liveness w.r.t. a property Pr in a given state S means
 - S satisfies Pr , or there is **some** causal path of global states from S to S' where S' satisfies Pr
- Safety w.r.t. a property Pr in a given state S means
 - S satisfies Pr , and **all** global states S' reachable from S also satisfy Pr

USING GLOBAL SNAPSHOT ALGORITHM

- Chandy-Lamport algorithm can be used to detect global properties that are **stable**
 - Stable = once true, stays true forever afterwards
- **Stable Liveness examples**
 - Computation has terminated
- **Stable Non-Safety examples**
 - There is a deadlock
 - An object is orphaned (no pointers point to it)
- All stable global properties can be detected using the Chandy-Lamport algorithm
 - Due to its causal correctness

SUMMARY

- The ability to calculate global snapshots in a distributed system is very important
- But don't want to interrupt running distributed application
- Chandy-Lamport algorithm calculates global snapshot
- Obeys causality (creates a consistent cut)
- Can be used to detect stable global properties
- Safety vs. Liveness



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

LEADER ELECTION

Lecture A

THE ELECTION PROBLEM

Why Election?

- Example 1: Your Bank account details are replicated at a few servers, but one of these servers is responsible for receiving all reads and writes, i.e., it is the **leader** among the replicas

- What if there are two leaders per customer?
- What if servers disagree about who the leader is?
- What if the leader crashes?

Each of the above scenarios leads to Inconsistency

More motivating examples

- Example 2: (A few lectures ago) In the sequencer-based algorithm for total ordering of multicasts, the “sequencer” = leader
- Example 3: Group of NTP servers: who is the root server?
- Other systems that need leader election: Apache Zookeeper, Google’s Chubby
- Leader is useful for coordination among distributed servers

Leader Election Problem

- In a group of processes, elect a *Leader* to undertake special tasks
 - And *let everyone know* in the group about this Leader
- What happens when a leader fails (crashes)
 - Some process detects this (using a Failure Detector!)
 - Then what?
- Focus of this lecture: *Election algorithm*. Its goal:
 1. Elect one leader only among the non-faulty processes
 2. All non-faulty processes agree on who is the leader

System Model

- N processes.
- Each process has a unique id.
- Messages are eventually delivered.
- Failures may occur during the election protocol.

Calling for an Election

- Any process can call for an election.
- A process can call for at most one election at a time.
- Multiple processes are allowed to call an election simultaneously.
 - All of them together must yield only a single leader
- The result of an election should not depend on which process calls for it.

Election Problem, Formally

- A run of the election algorithm must always guarantee at the end:
 - **Safety**: For all non-faulty processes p : (p 's elected = (q: a particular non-faulty process with the best attribute value) or Null)
 - **Liveness**: For all election runs: (election run terminates)
& for all non-faulty processes p : p 's elected is not Null
- At the end of the election protocol, the non-faulty process with the best (highest) election attribute value is elected.
 - Common attribute : leader has highest id
 - Other attribute examples: leader has highest IP address, or fastest cpu, or most disk space, or most number of files, etc.

Next

- A classical leader election algorithm.

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

LEADER ELECTION

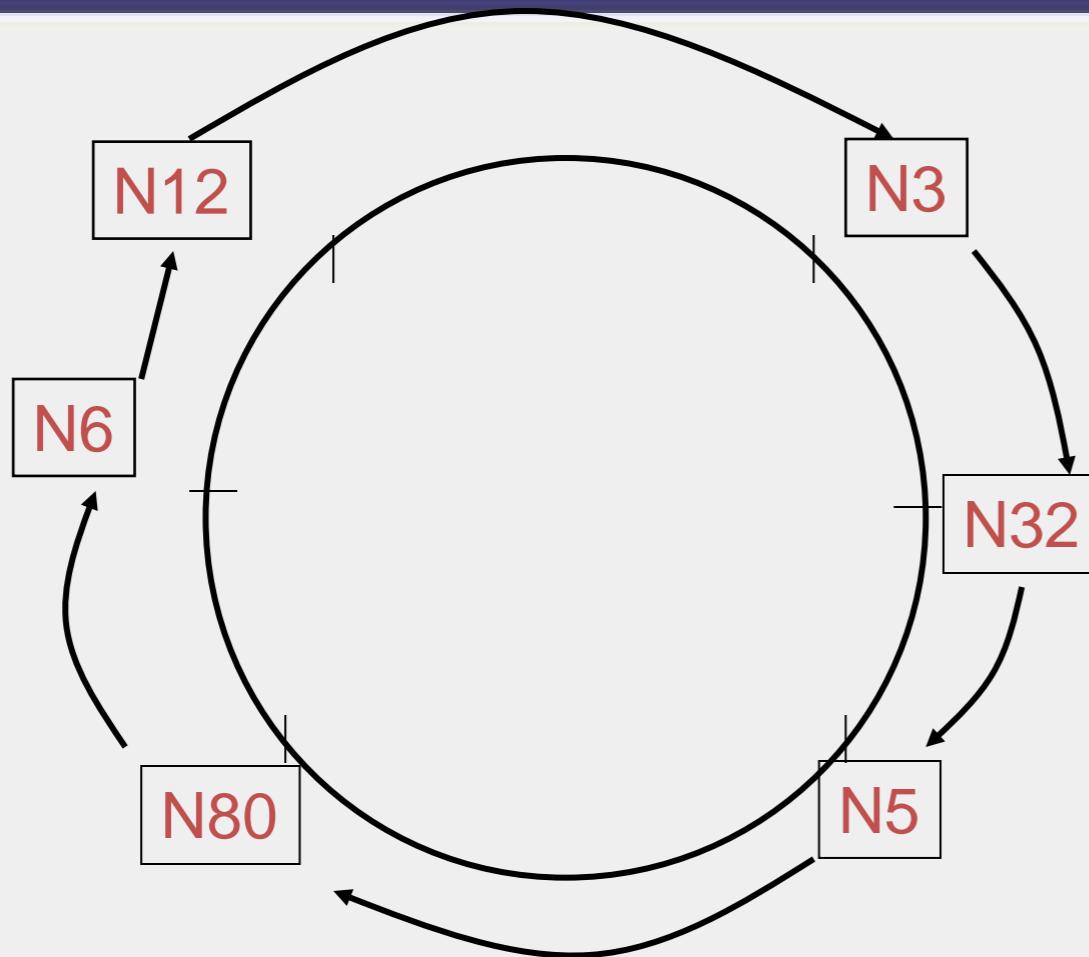
Lecture B

RING LEADER ELECTION

The Ring

- N processes are organized in a logical ring
 - Similar to ring in Chord p2p system
 - i -th process p_i has a communication channel to $p_{(i+1) \bmod N}$
 - All messages are sent clockwise around the ring.

The Ring



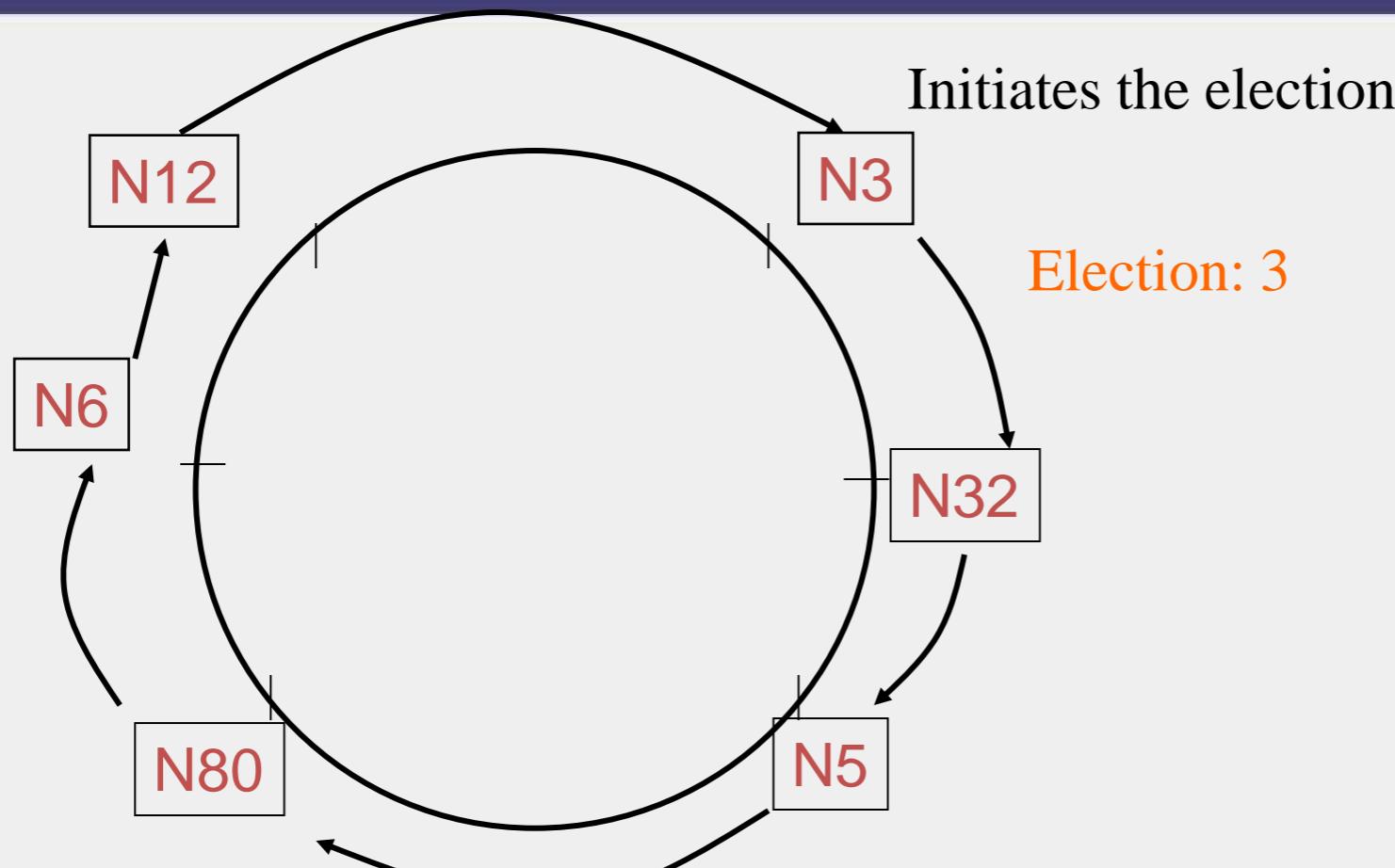
The Ring Election Protocol

- Any process p_i that discovers the old coordinator has failed initiates an “**Election**” message that contains p_i ’s own id:attr. This is the *initiator* of the election.
- When a process p_i receives an “Election” message, it compares the attr in the message with its own attr.
 - If the arrived attr is greater, p_i forwards the message.
 - If the arrived attr is smaller and p_i has not forwarded an election message earlier, it overwrites the message with its own id:attr, and forwards it.
 - If the arrived id:attr matches that of p_i , then p_i ’s attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an “**Elected**” message to its neighbor with its id, announcing the election result.

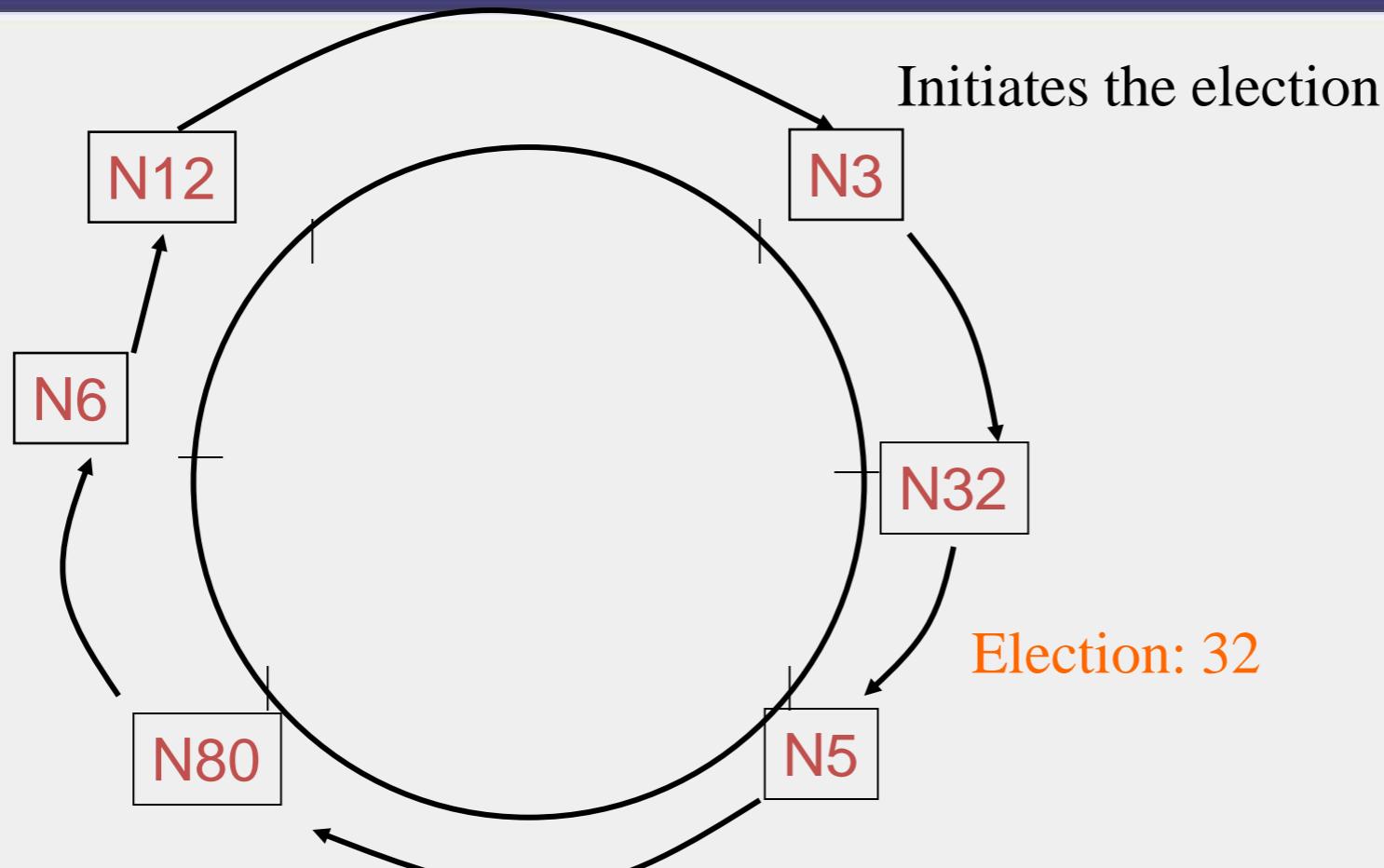
The Ring Election Protocol (2)

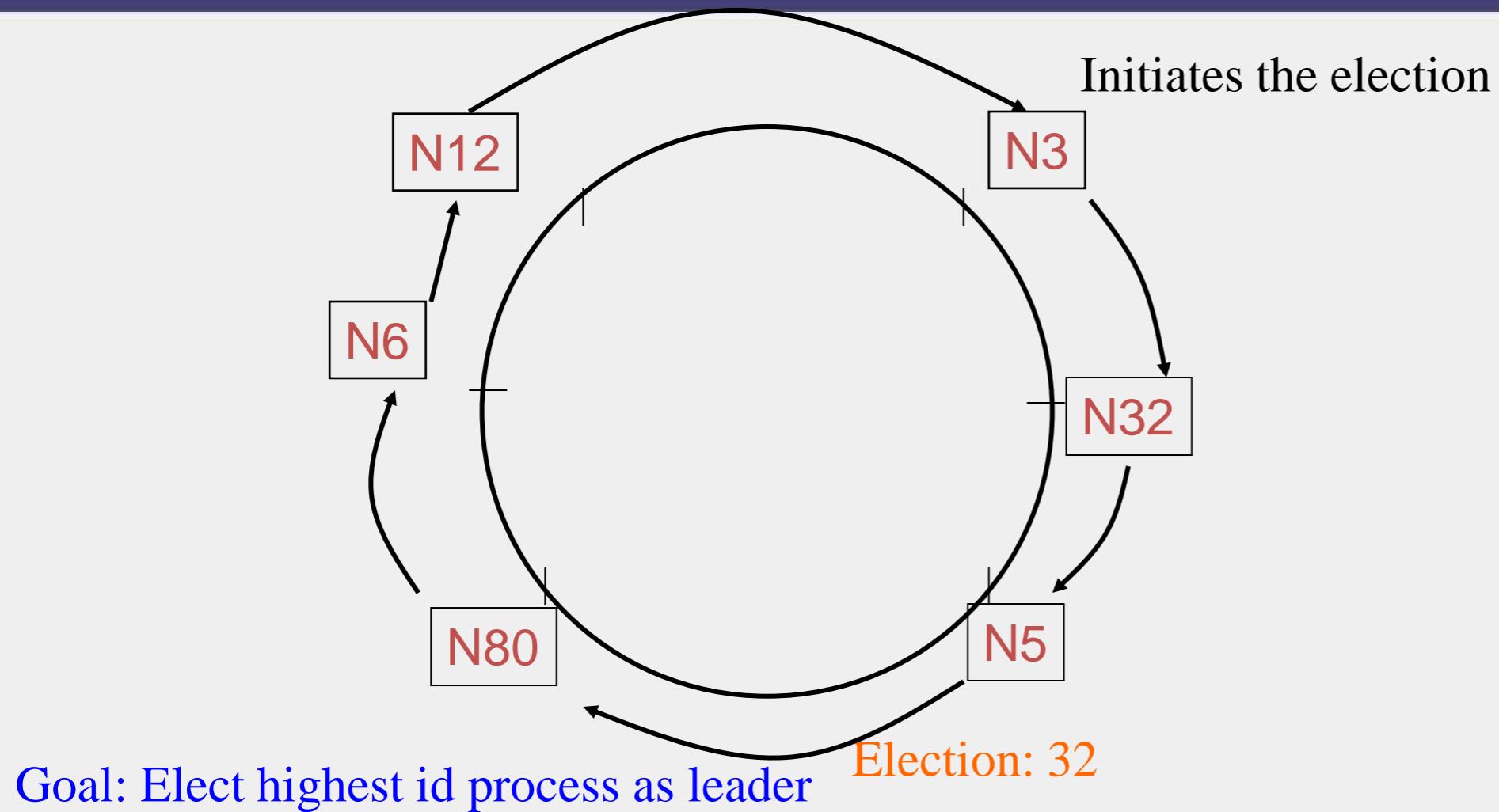
- When a process p_i receives an “Elected” message, it
 - sets its variable $elected_i \leftarrow id$ of the message.
 - forwards the message unless it is the new coordinator.

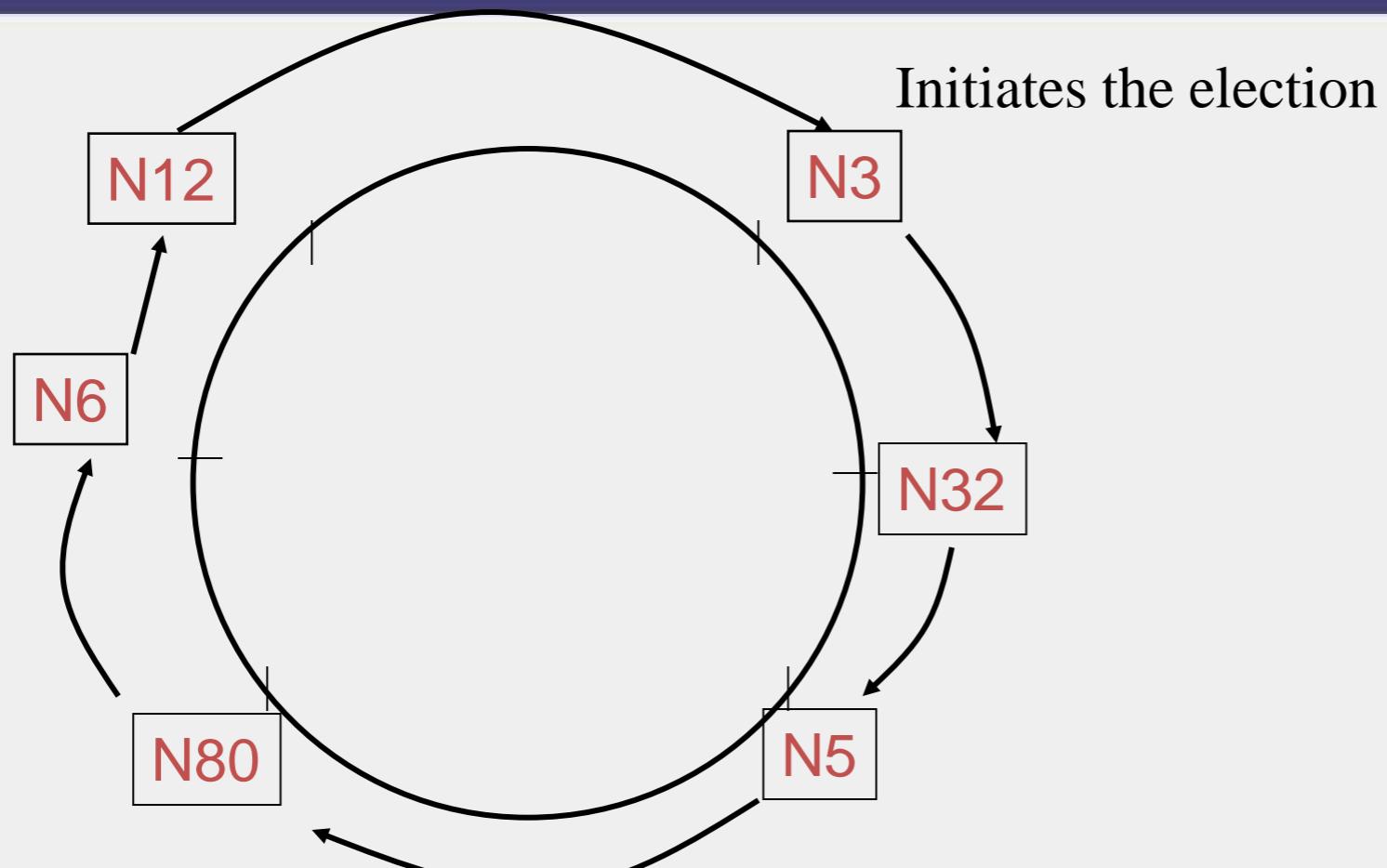
Ring Election: Example



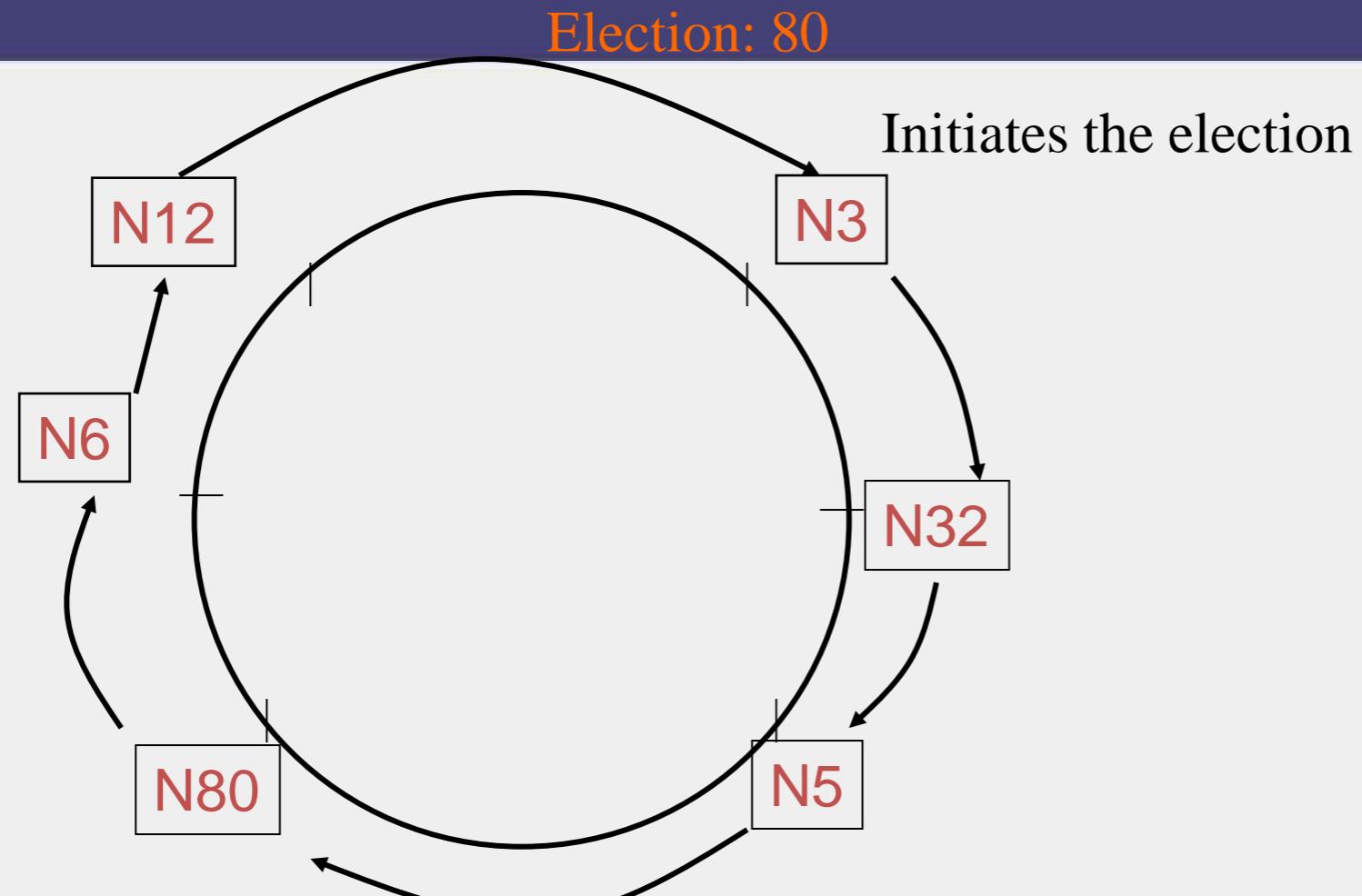
Goal: Elect highest id process as leader



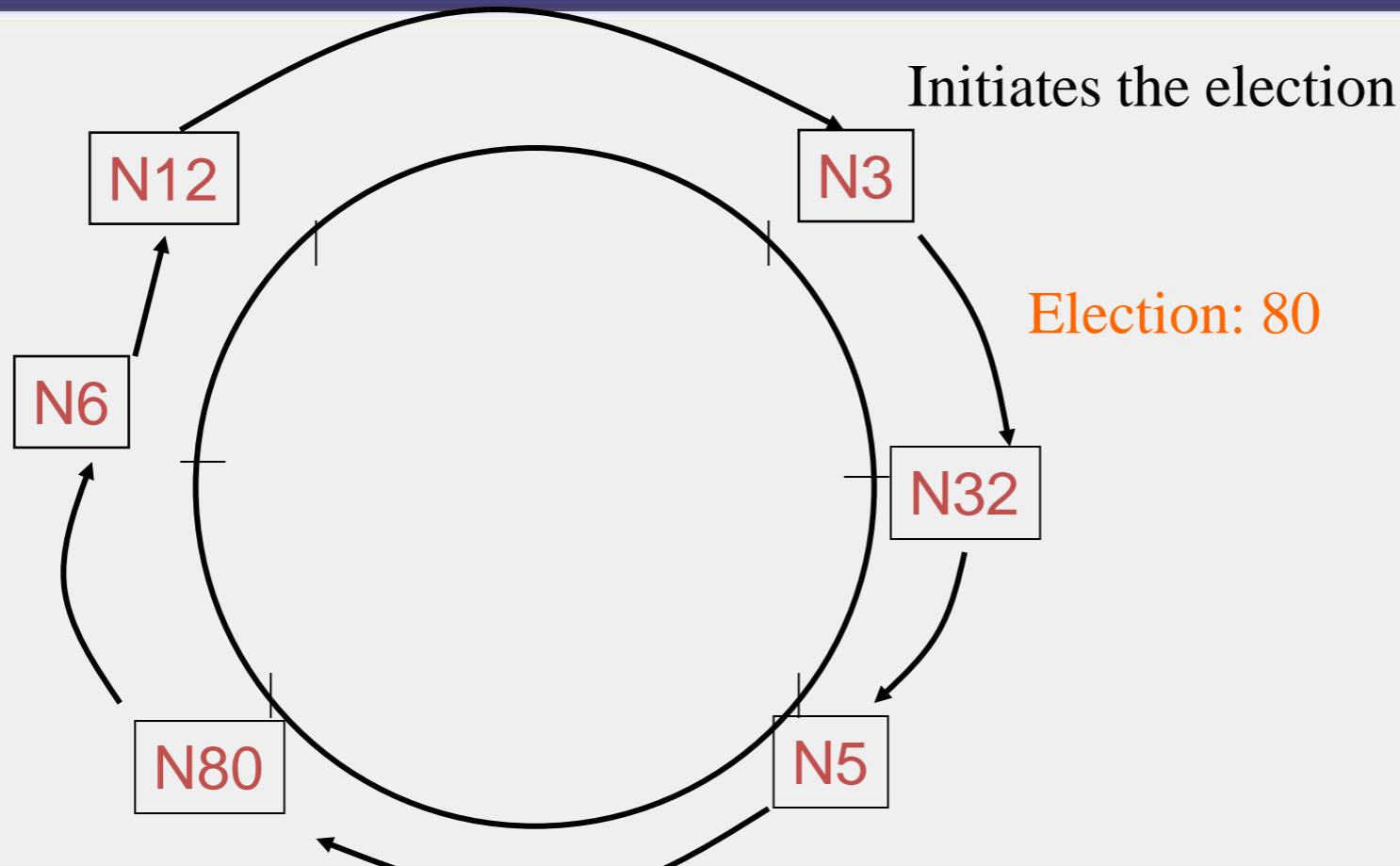




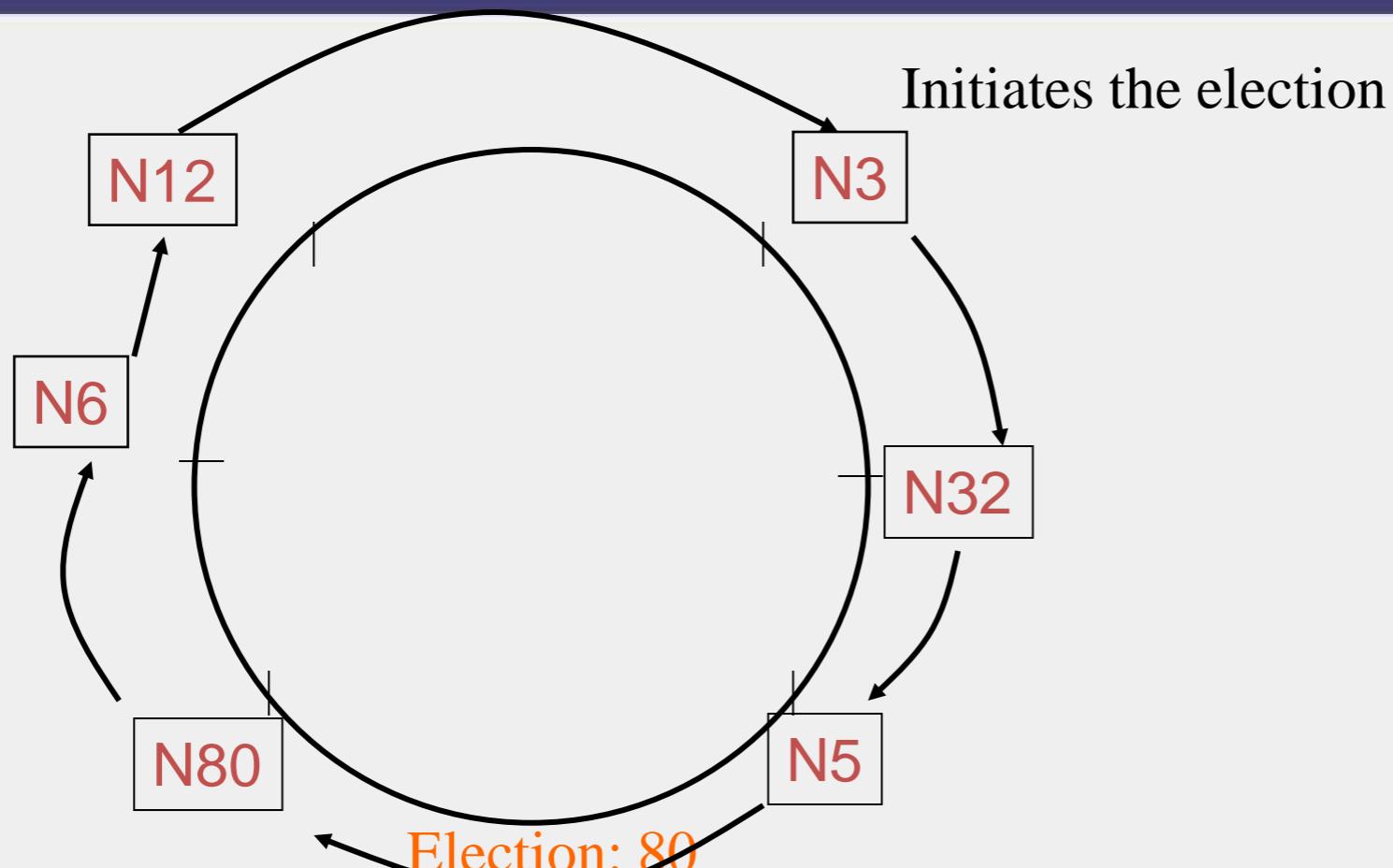
Goal: Elect highest id process as leader



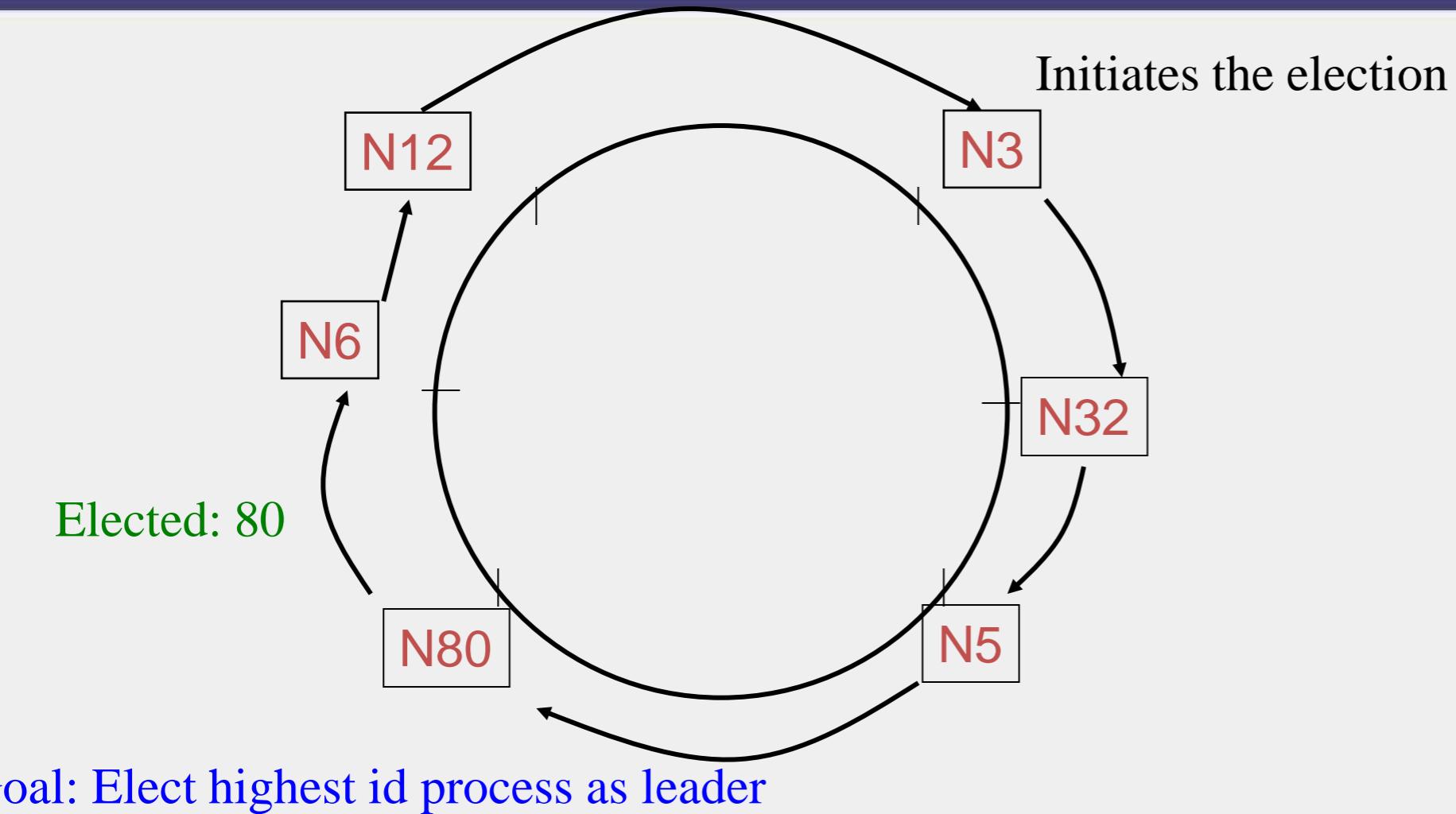
Goal: Elect highest id process as leader

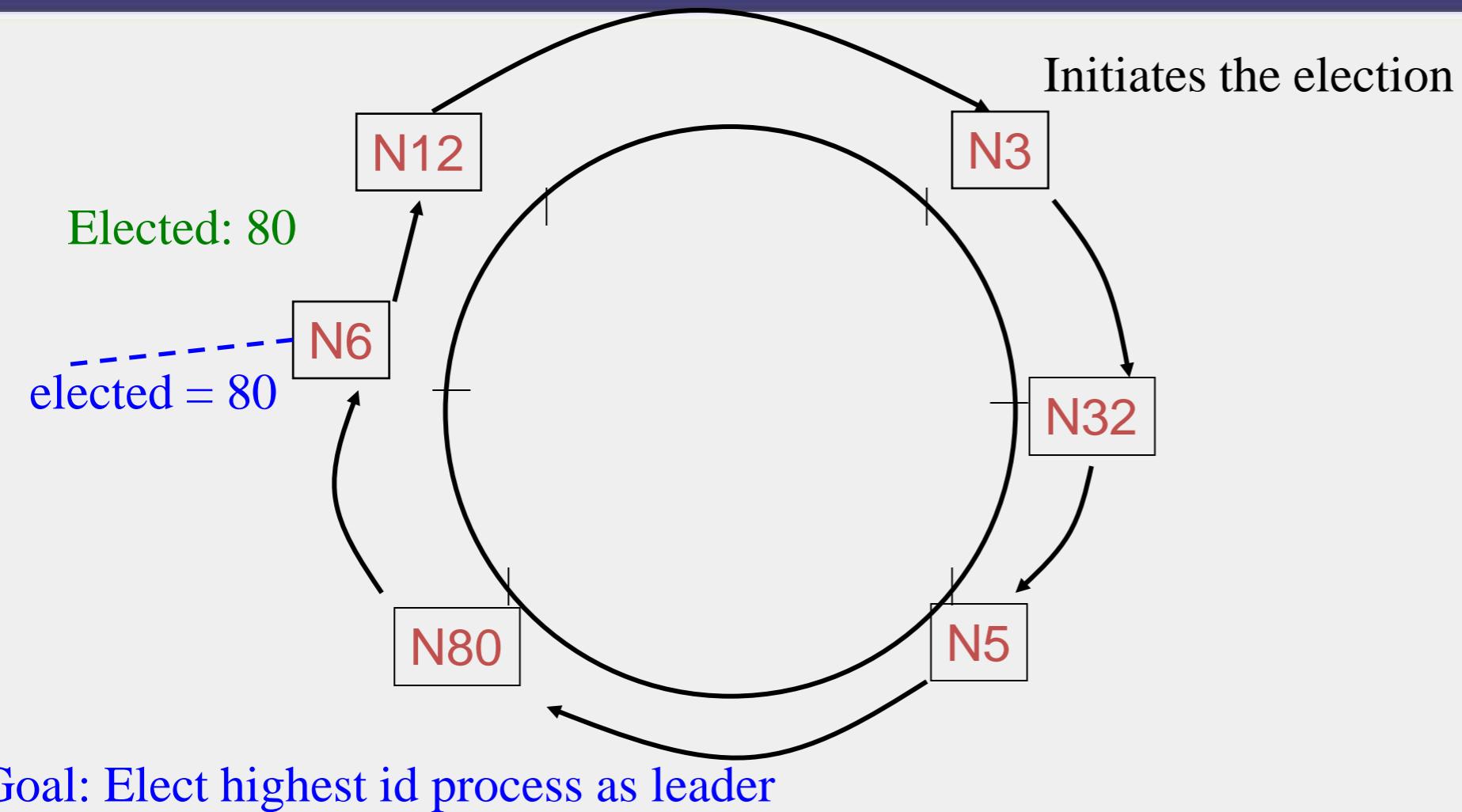


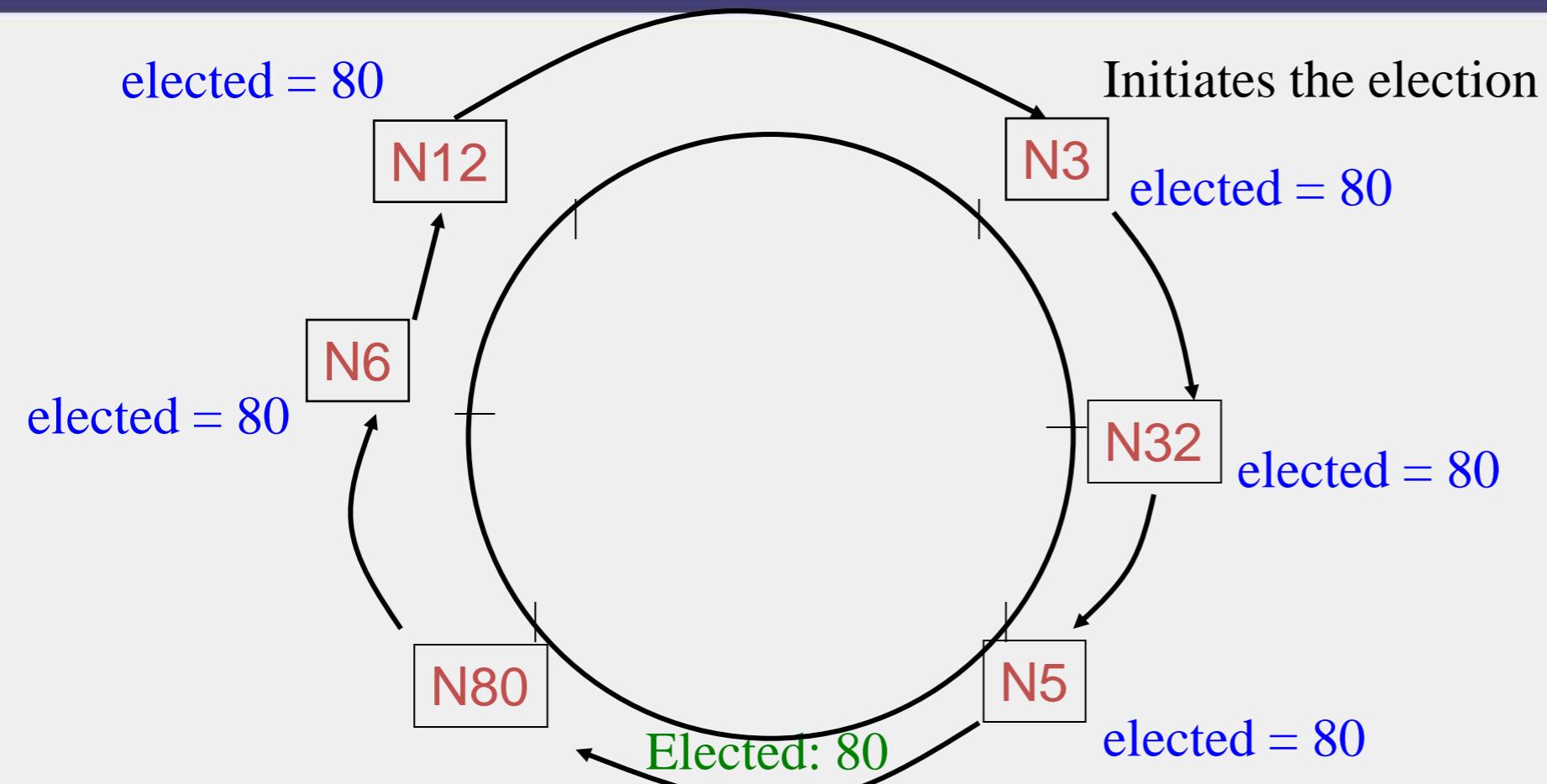
Goal: Elect highest id process as leader



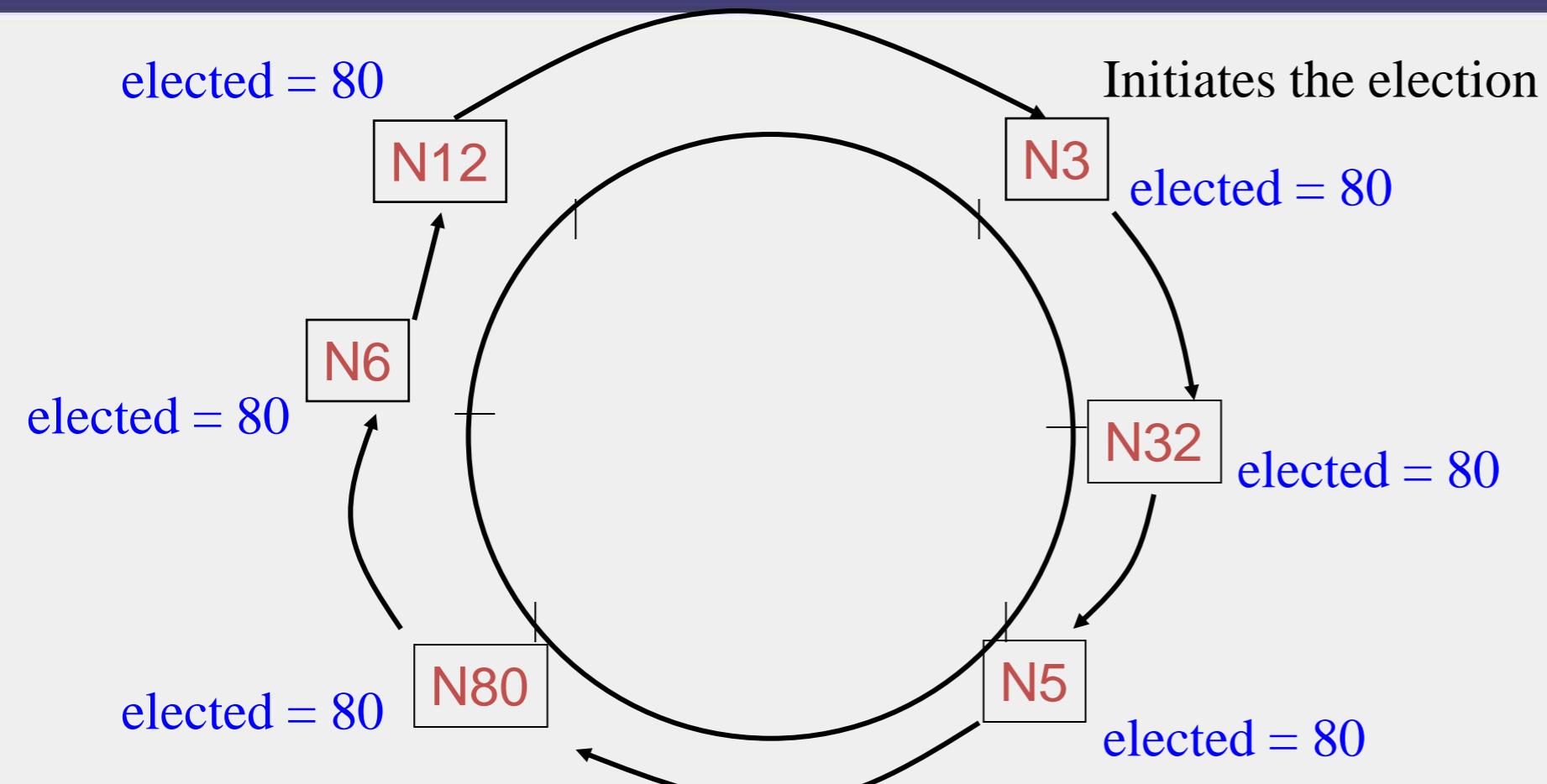
Goal: Elect highest id process as leader







Goal: Elect highest id process as leader

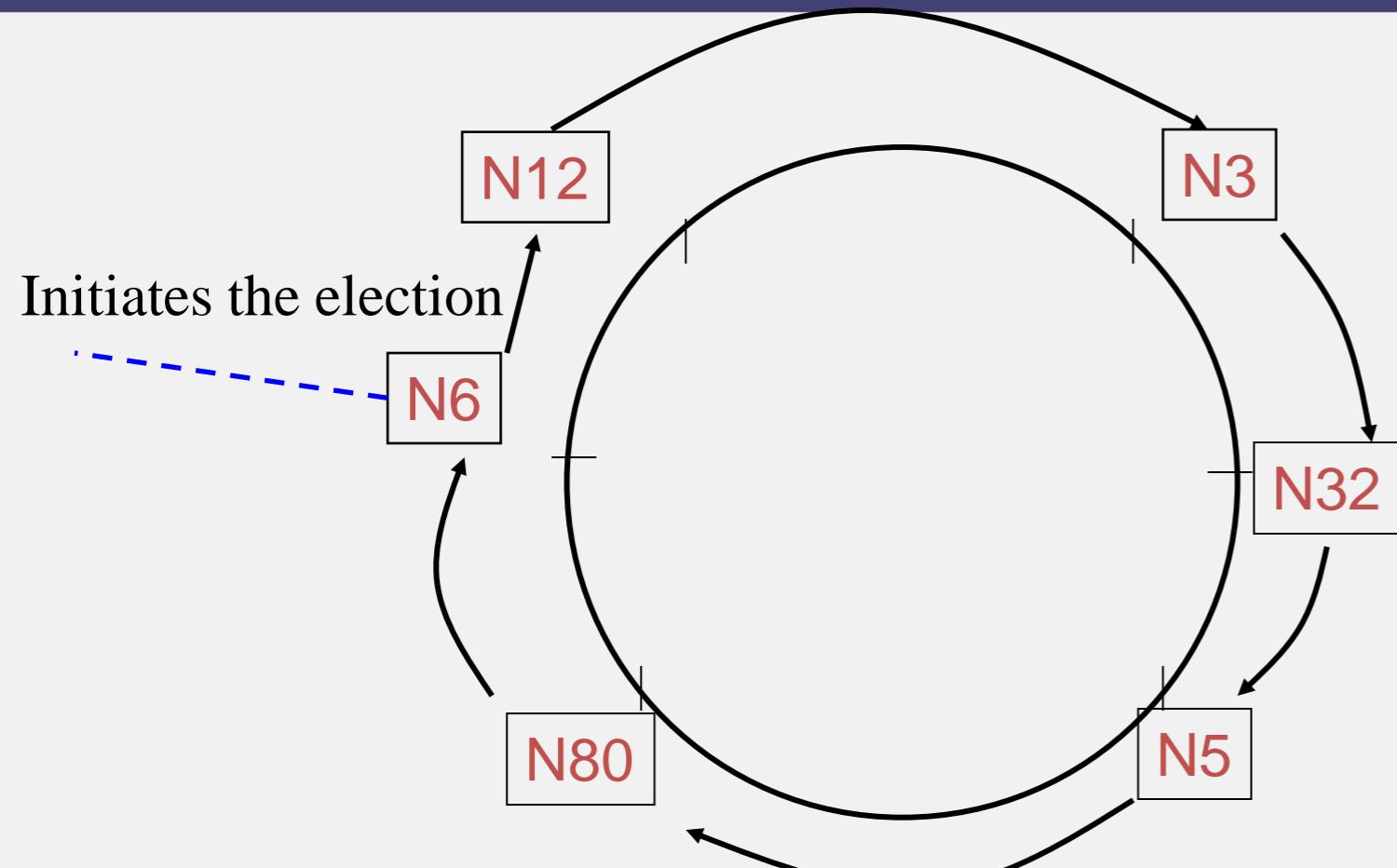


Goal: Elect highest id process as leader

Analysis

- Let's assume no failures occur during the election protocol itself, and there are N processes
- How many messages?
- Worst case occurs when the initiator is the ring successor of the would-be leader

Worst-case



Worst-case Analysis

- $(N-1)$ messages for Election message to get from Initiator (N6) to would-be coordinator (N80)
- N messages for Election message to circulate around ring without message being changed
- N messages for Elected message to circulate around the ring
- Message complexity: $(3N-1)$ messages
- Completion time: $(3N-1)$ message transmission times
- Thus, if there are no failures, election terminates (liveness) and everyone knows about highest-attribute process as leader (safety)

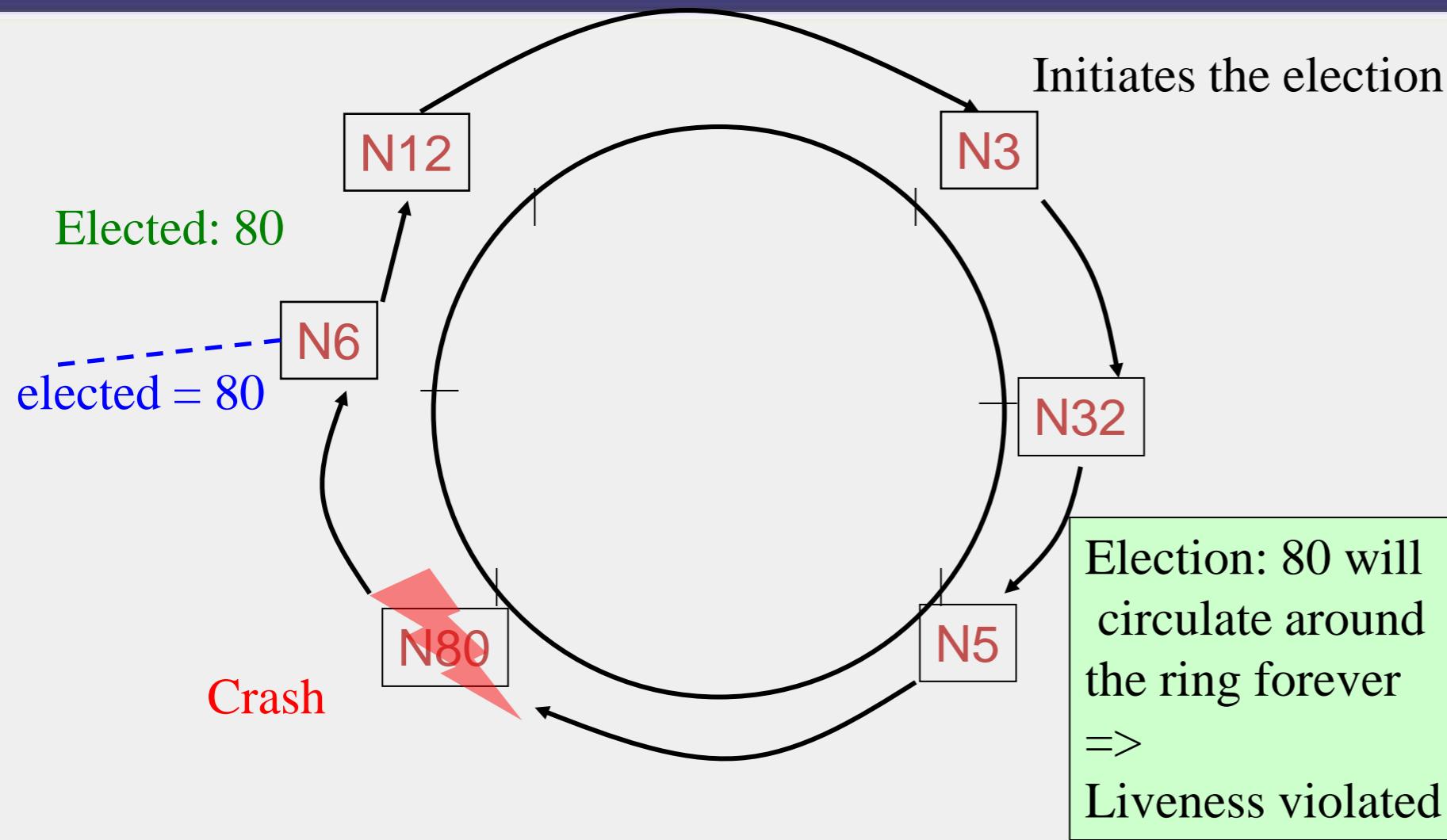
Best Case?

- Initiator is the would-be leader, i.e., N80 is the initiator
- Message complexity: $2N$ messages
- Completion time: $2N$ message transmission times

Multiple Initiators?

- Each process remembers in cache the initiator of each Election/Elected message it receives
- (All the time) Each process suppresses Election/Elected messages of any lower-id initiators
- Updates cache if receives higher-id initiator's Election/Elected message
- Result is that only the highest-id initiator's election run completes

Effect of Failures



Fixing for failures

- One option: have predecessor (or successor) of would-be leader N80 detect failure and start a new election run
 - May re-initiate election if
 - Receives an Election message but times out waiting for an Elected message
 - Or after receiving the Elected:80 message
 - But what if predecessor also fails?
 - And its predecessor also fails? (and so on)

Fixing for failures (2)

- Second option: use the failure detector
- Any process, after receiving Election:80 message, can detect failure of N80 via its own local failure detector
 - If so, start a new run of leader election
- But failure detectors may not be both complete and accurate
 - Incompleteness in FD \Rightarrow N80's failure might be missed \Rightarrow Violation of Safety
 - Inaccuracy in FD \Rightarrow N80 mistakenly detected as failed
 - \Rightarrow new election runs initiated forever
 - \Rightarrow Violation of Liveness

Why is Election so Hard?

- Because it is related to the consensus problem!
- If we could solve election, then we could solve consensus!
 - Elect a process, use its id's last bit as the consensus decision
- But since consensus is impossible in asynchronous systems, so is election!
- Next lecture: Can't we use Paxos?

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

LEADER ELECTION

Lecture C

ELECTION IN
CHUBBY AND ZOOKEEPER

Can use Consensus to solve Election

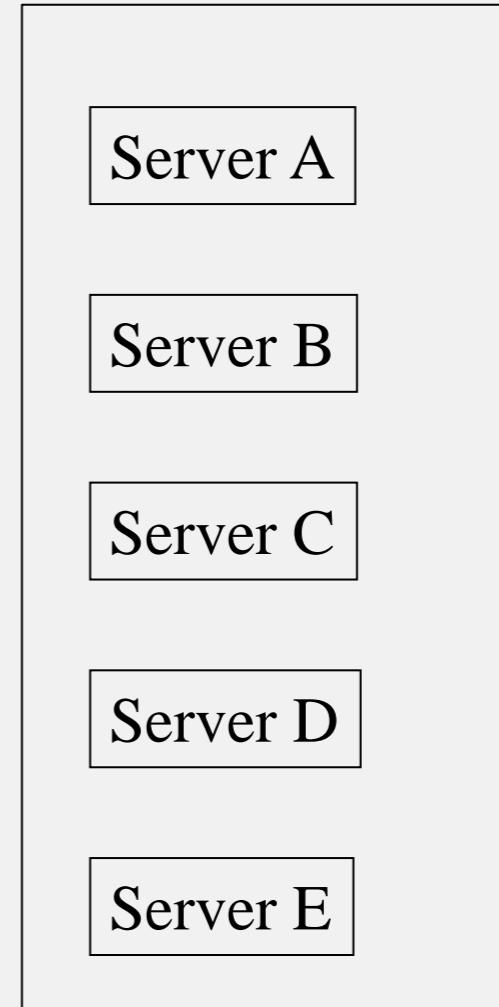
- One approach
 - Each process proposes a value
 - Everyone in group reaches consensus on some process P_i 's value
 - That lucky P_i is the new leader!

Election in Industry

- Several systems in industry use Paxos-like approaches for election
 - Paxos is a consensus protocol (safe, but eventually live): elsewhere in this course
- Google's Chubby system
- Apache Zookeeper

Election in Google Chubby

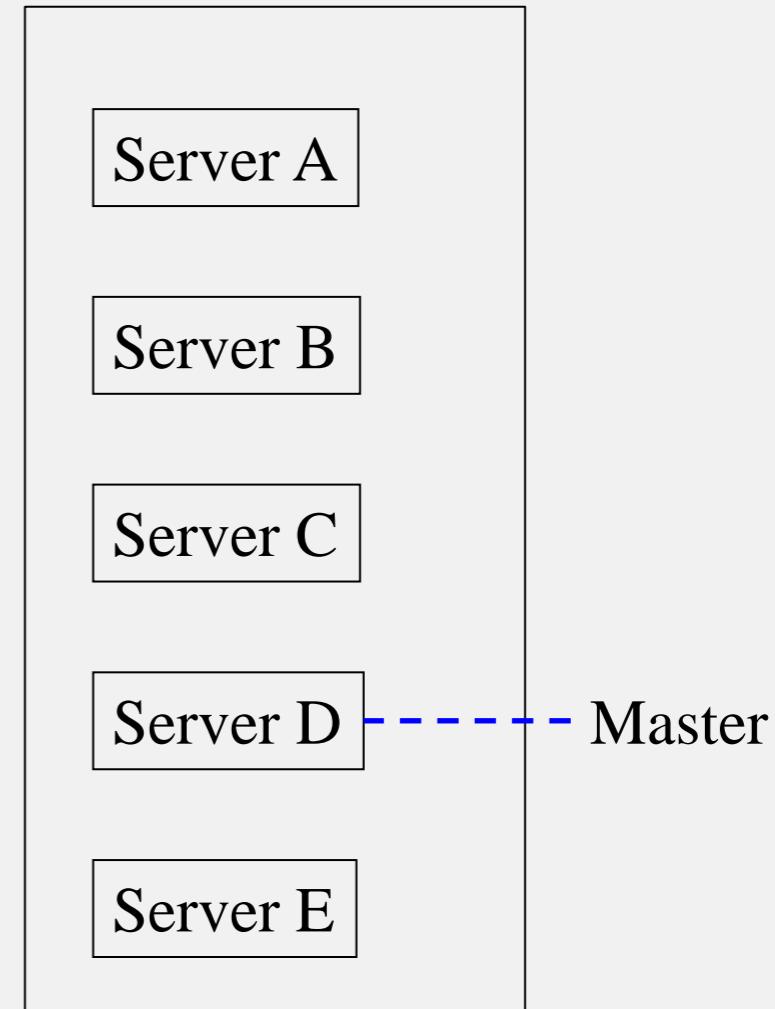
- A system for locking
- Essential part of Google's stack
 - Many of Google's internal systems rely on Chubby
 - BigTable, Megastore, etc.
- Group of replicas
 - Need to have a master server elected at all times



Reference: <http://research.google.com/archive/chubby.html>

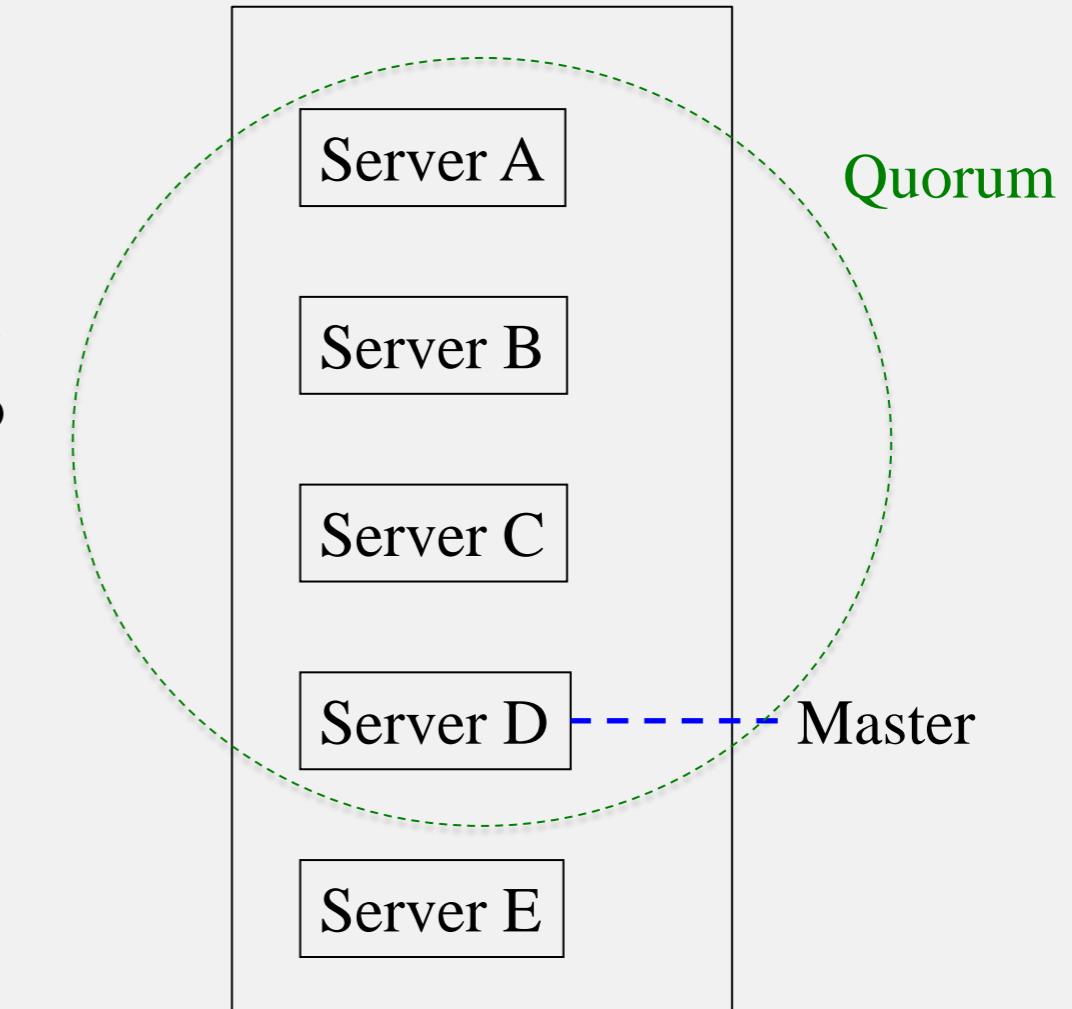
Election in Google Chubby (2)

- Group of replicas
 - Need to have a master (i.e., leader)
- Election protocol
 - Potential leader tries to get votes from other servers
 - Each server votes for at most one leader
 - Server with *majority* of votes becomes new leader, informs everyone



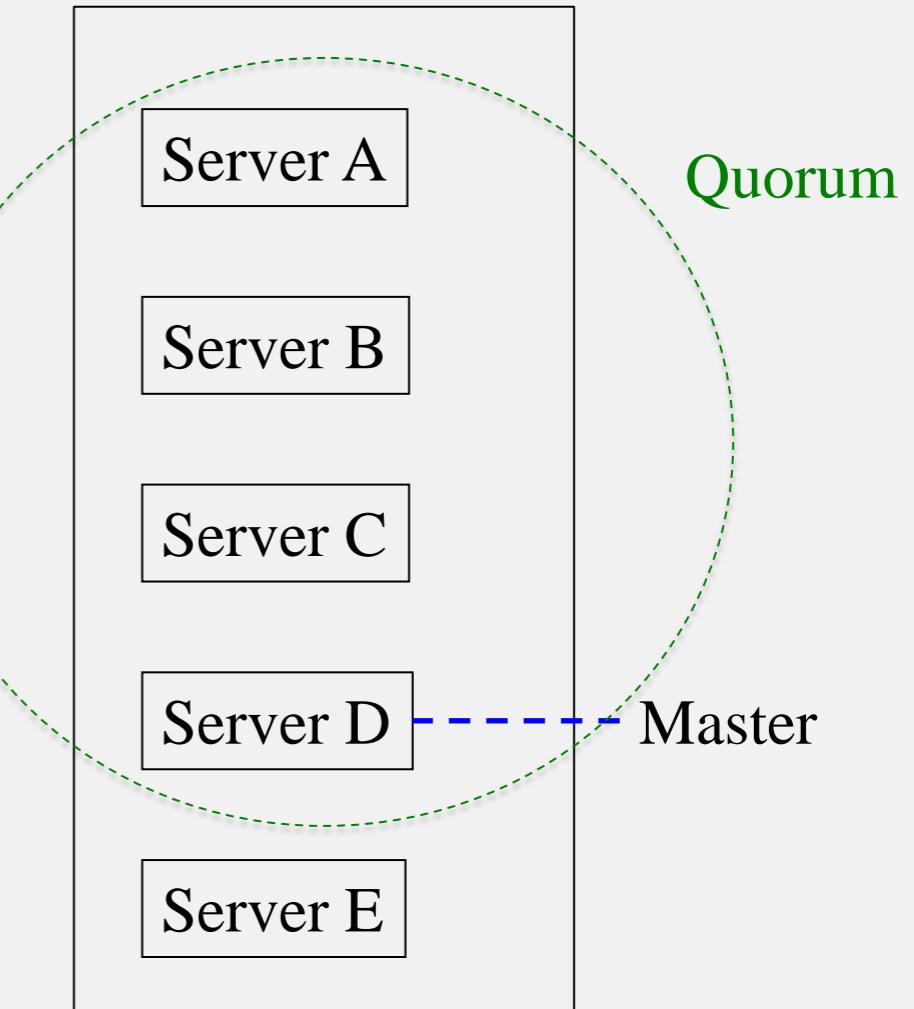
Election in Google Chubby (3)

- Why **safe**?
 - Essentially, each potential leader tries to reach a *quorum* (should sound familiar!)
 - Since any two quorums intersect, and each server votes at most once, cannot have two leaders elected simultaneously
- Why **live**?
 - Only eventually live! Failures may keep happening so that no leader is ever elected
 - In practice: elections take a few seconds.
Worst-case noticed by Google: 30 s



Election in Google Chubby (4)

- After election finishes, other servers promise not to run election again for “a while”
 - “While” = time duration called “Master lease”
 - Set to a few seconds
- Master lease can be renewed by the master as long as it continues to win a majority each time
- Lease technique ensures automatic re-election on master failure



Election in Zookeeper

- Centralized service for maintaining configuration information
- Uses a variant of Paxos called Zab (Zookeeper Atomic Broadcast)
- Needs to keep a leader elected at all times
- <http://zookeeper.apache.org/>

Election in Zookeeper (2)

- Each server creates a new *sequence number* for itself
 - Let's say the sequence numbers are **ids**
 - Gets highest id so far (from ZK file system), creates next-higher id, writes it into ZK file system
- Elect the highest-id server as leader

N12

N3

N6

N32

Master

N80

N5

Election in Zookeeper (3)

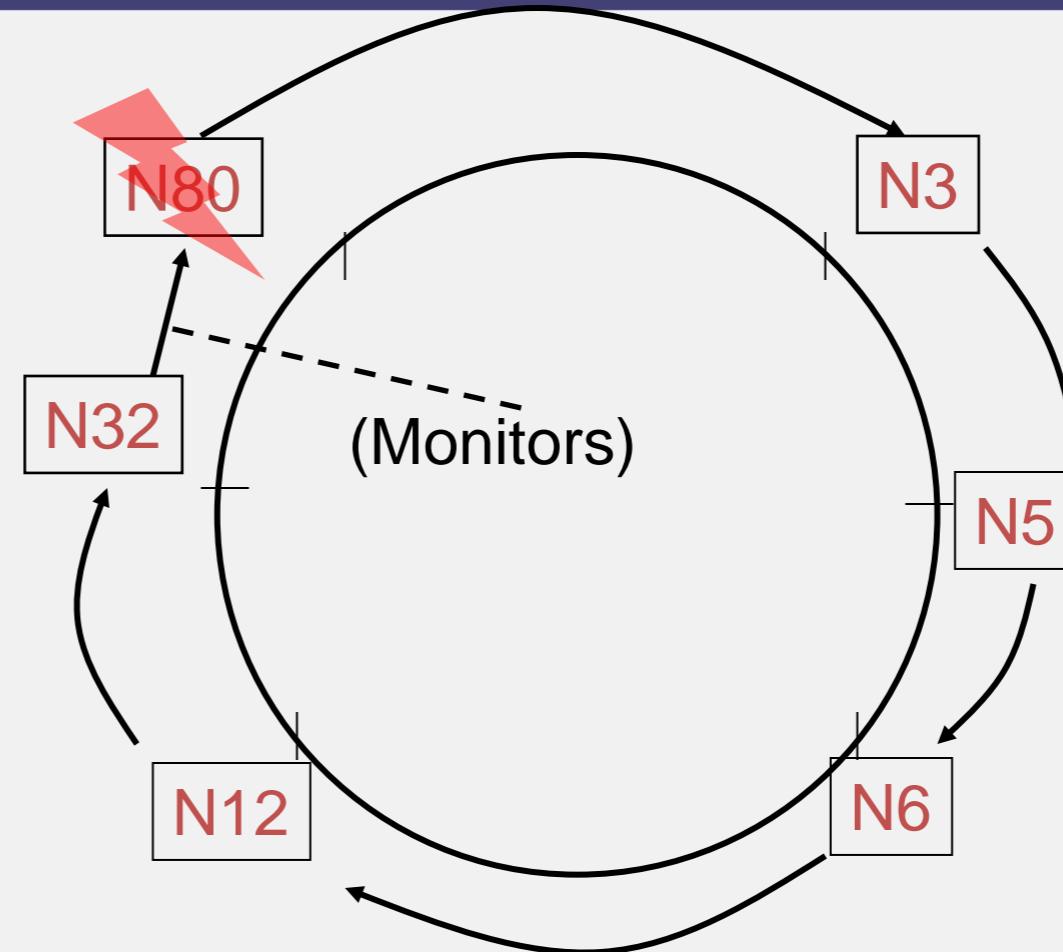
Failures:

- One option: everyone monitors current master (directly or via a failure detector)
 - On failure, initiate election
 - Leads to a flood of elections
 - Too many messages



Election in Zookeeper (4)

- Second option (implemented in Zookeeper)
 - Each process monitors its next-higher id process
 - if that successor was the leader and it has failed
 - Become the new leader
 - else
 - wait for a timeout, and check your successor again



Election in Zookeeper (5)

- What about id conflicts? What if leader fails during election?
- To address this, Zookeeper uses a *two-phase commit* (run after the sequence/id) protocol to commit the leader
 - Leader sends NEW_LEADER message to all
 - Each process responds with ACK to at most one leader, i.e., one with highest process id
 - Leader waits for a majority of ACKs, and then sends COMMIT to all
 - On receiving COMMIT, process updates its leader variable
- Ensures that safety is still maintained

Next

- Another classical algorithm: Bully algorithm

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

LEADER ELECTION

Lecture D

BULLY ALGORITHM

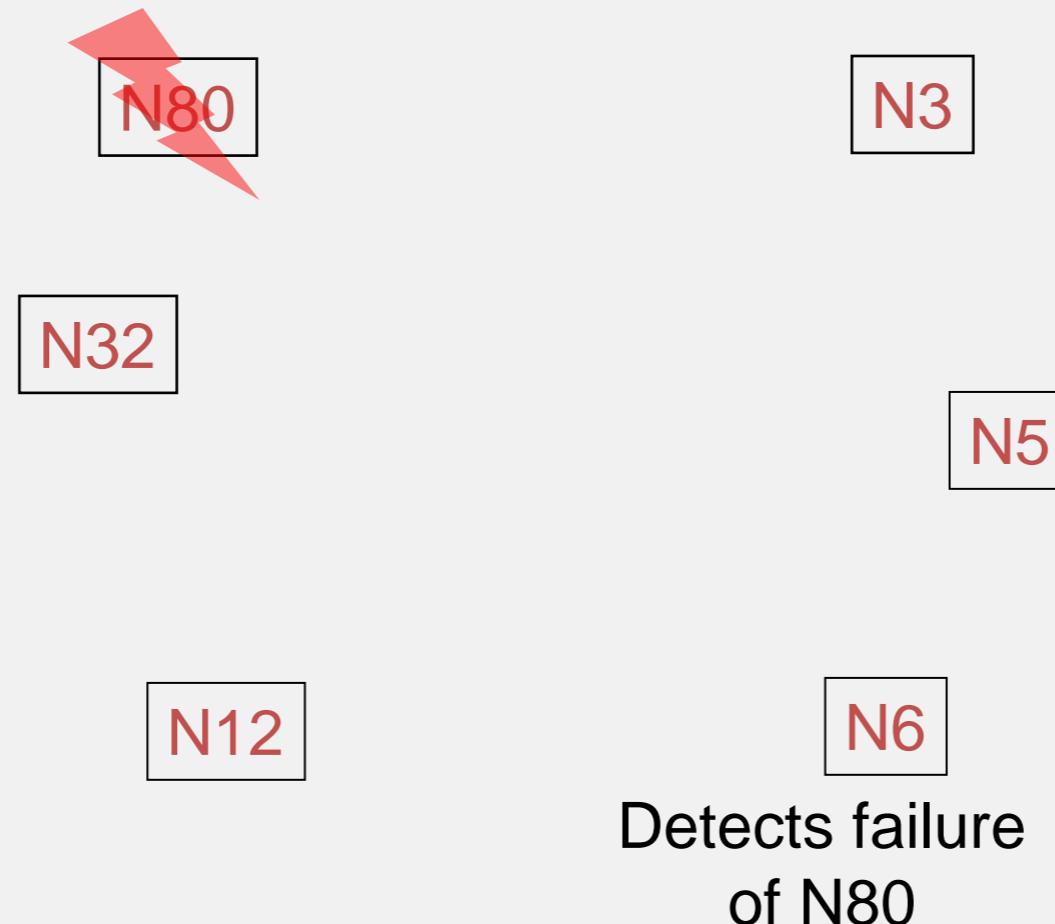
Bully Algorithm

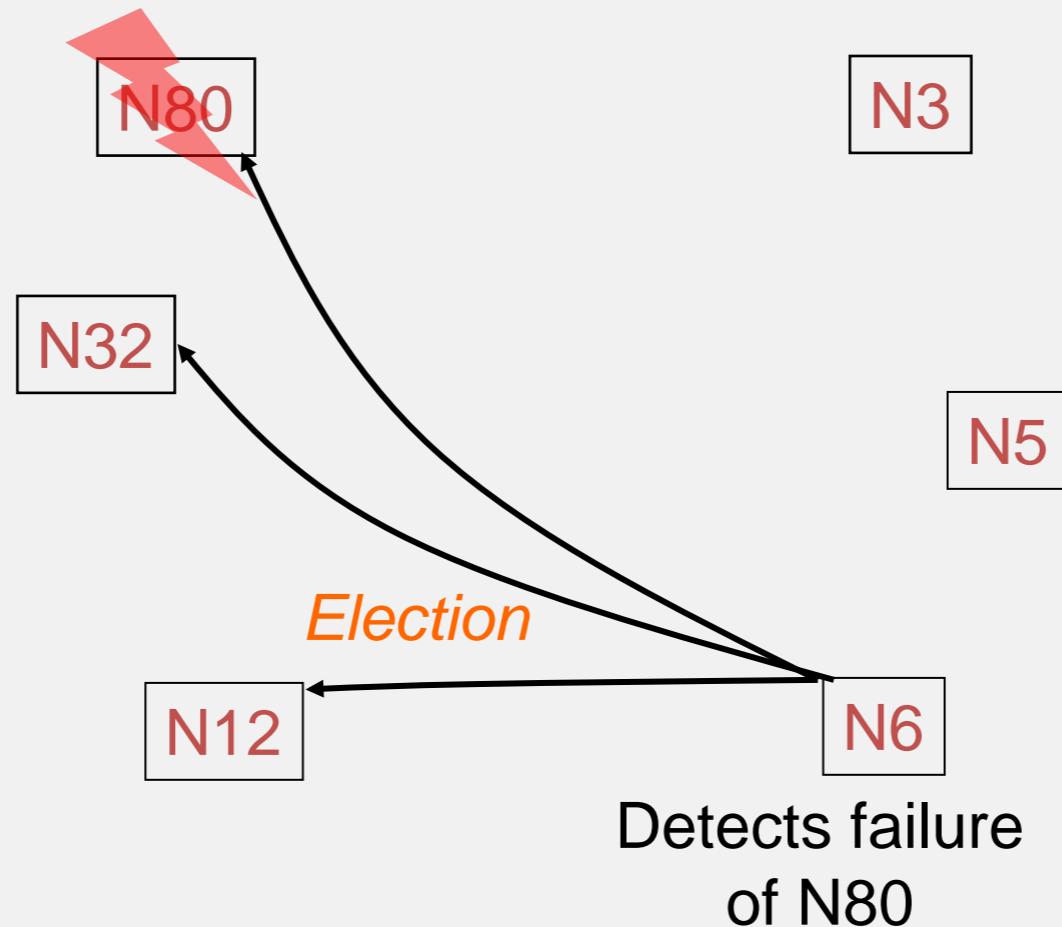
- All processes know other process' ids
- When a process finds the coordinator has failed (via the failure detector):
 - if it knows its id is the highest, it elects itself as coordinator, then sends a *Coordinator* message to all processes with lower identifiers. Election is completed.
 - else it initiates an election by sending an *Election* message
 - (contd.)

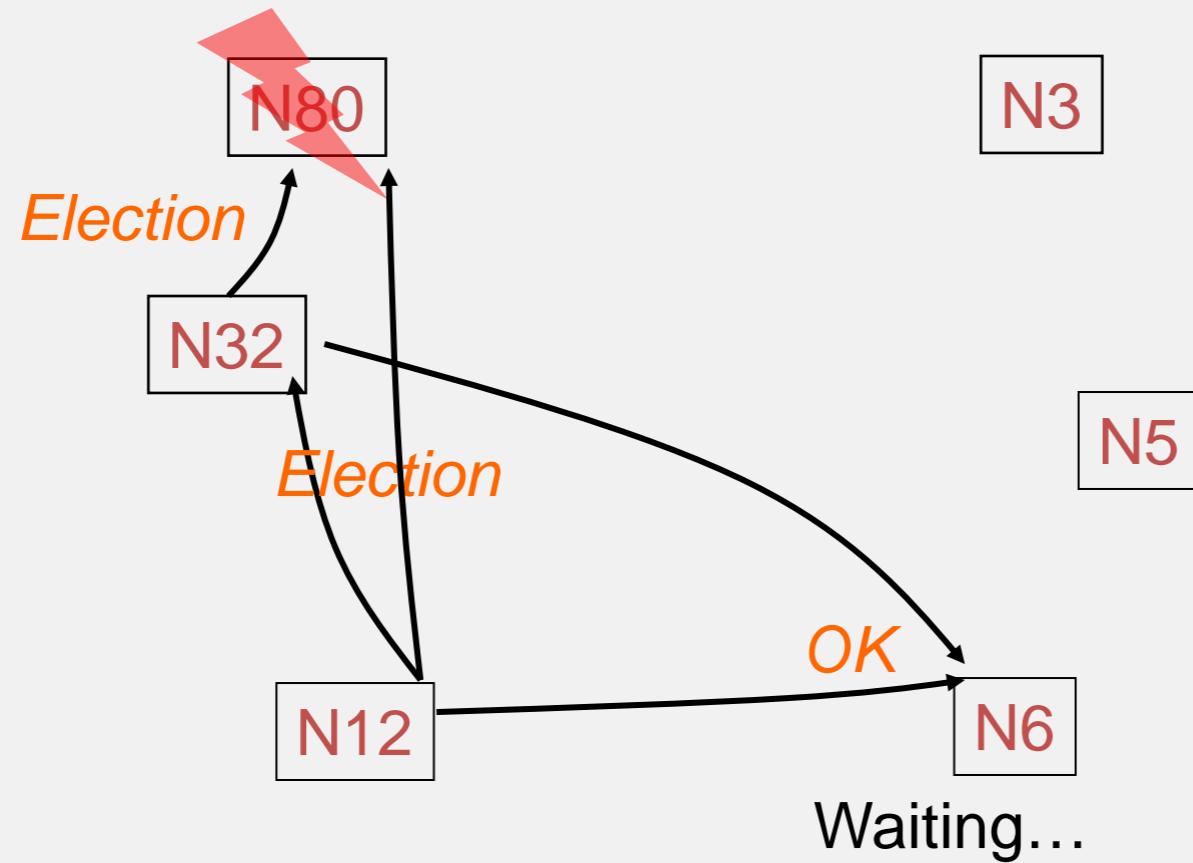
Bully Algorithm (2)

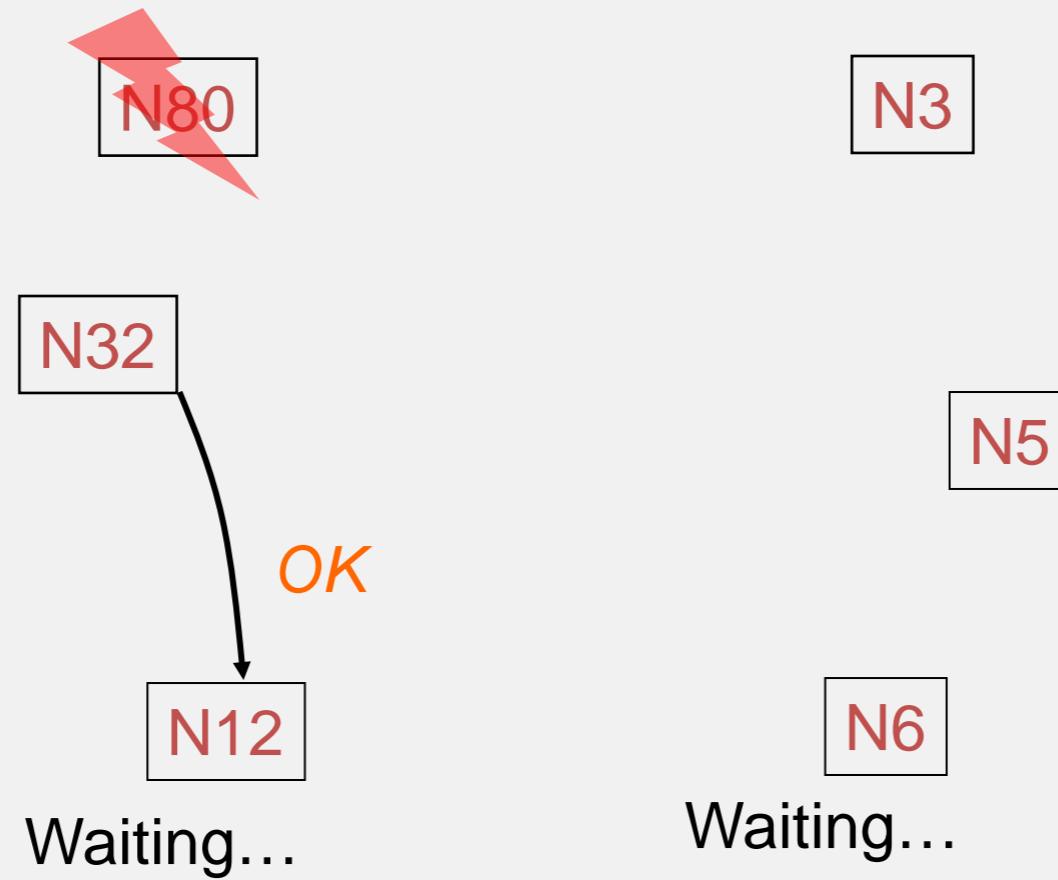
- **else** it initiates an election by sending an *Election* message
 - Sends it to only processes that have a *higher id than itself*.
 - **if** receives no answer within timeout, calls itself leader and sends *Coordinator* message to all lower id processes. Election completed.
 - **if** an answer received however, then there is some non-faulty higher process => so, wait for coordinator message. If none received after another timeout, start a new election run.
- A process that receives an *Election* message replies with *OK* message, and starts its own leader election protocol (unless it has already done so)

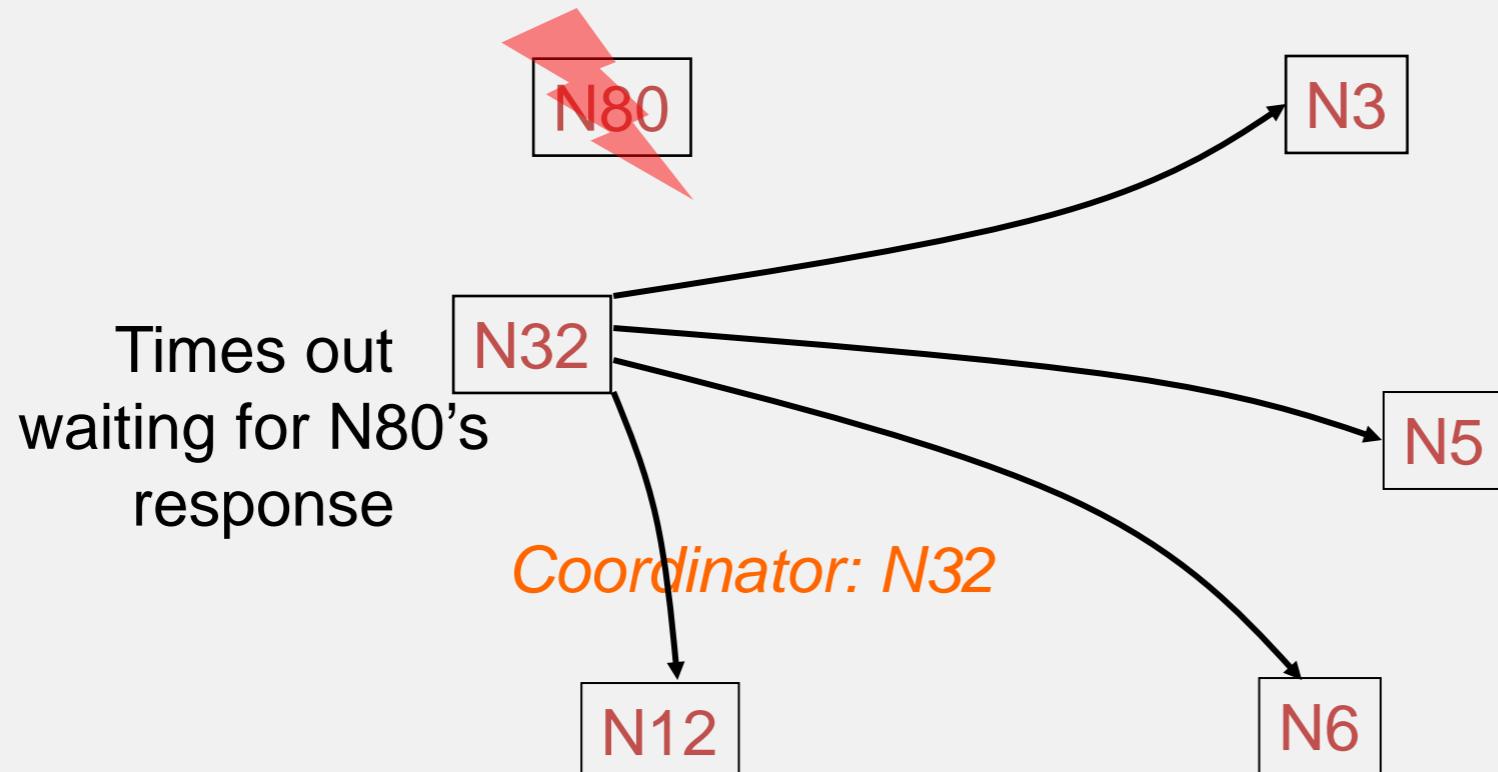
Bully Algorithm: Example







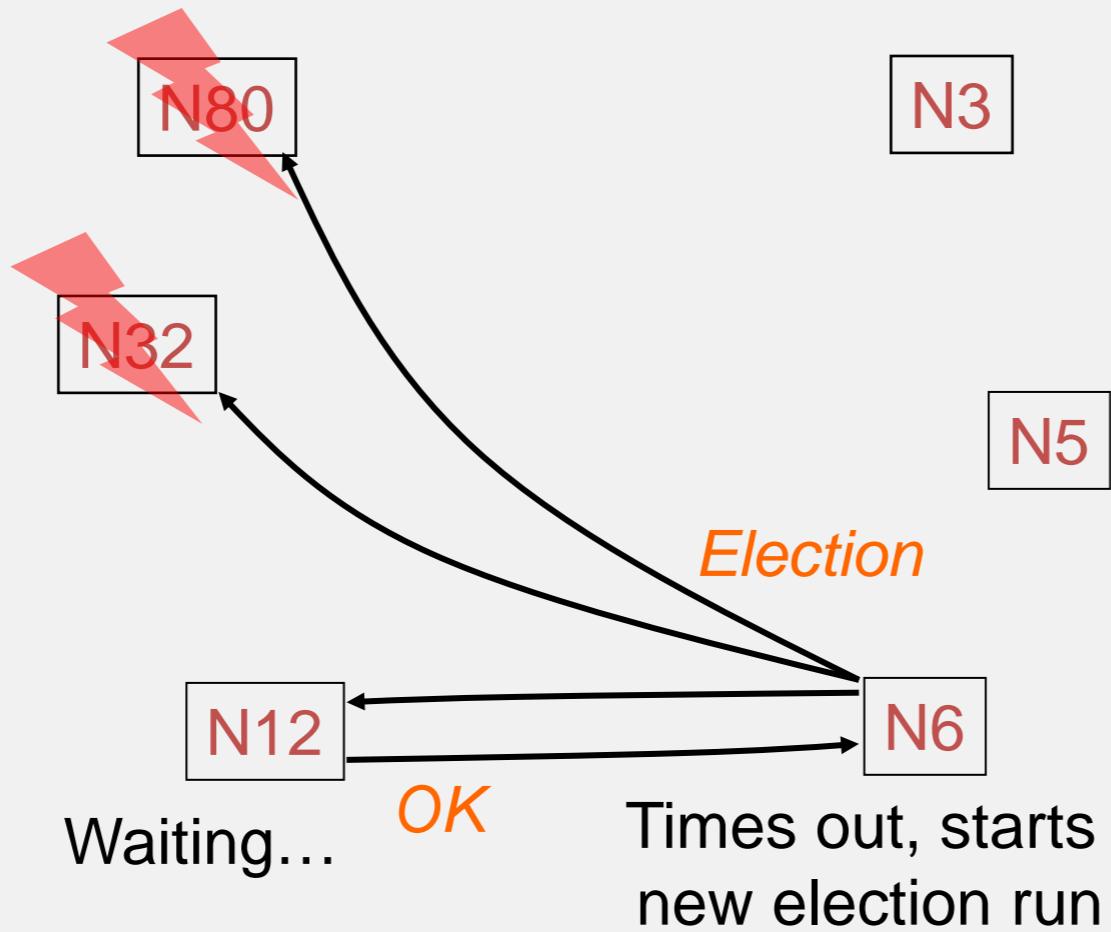


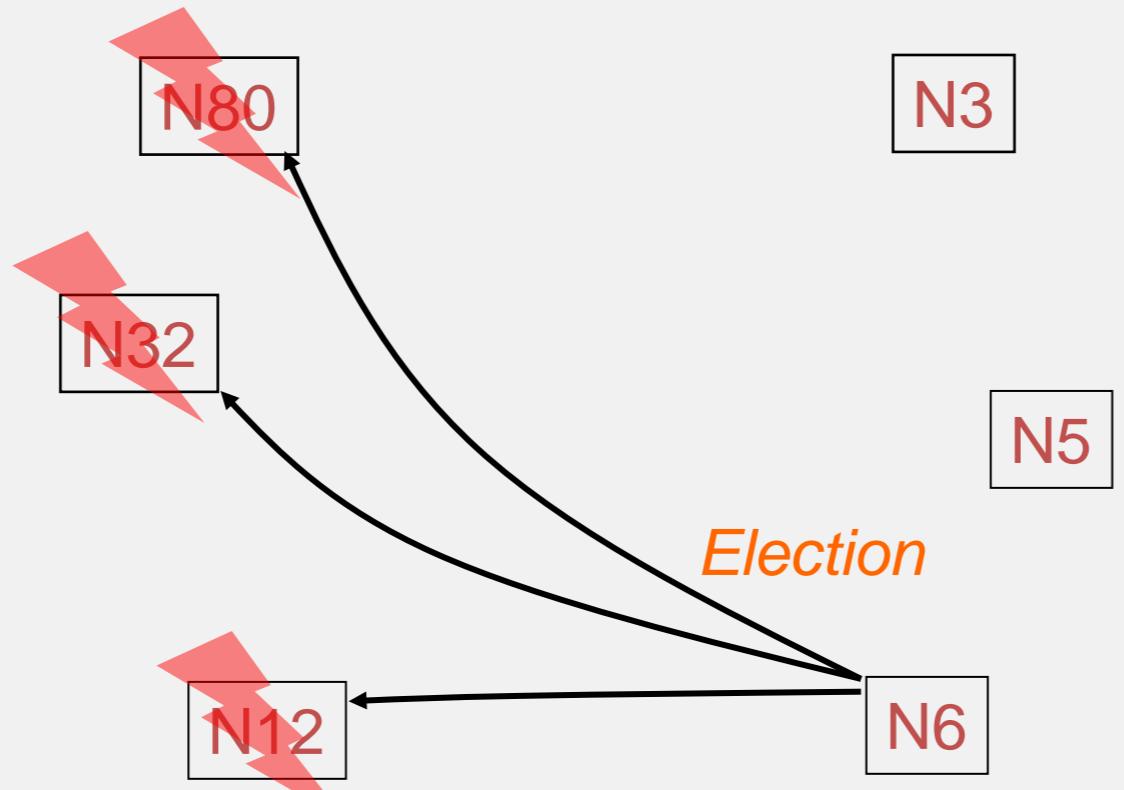


Election is completed

Failures during Election Run







Times out, starts
another new election run

Failures and Timeouts

- If failures stop, eventually will elect a leader
- How do you set the timeouts?
- Based on Worst-case time to complete election
 - 5 message transmission times if there are no failures during the run:
 1. Election from lowest id server in group
 2. Answer to lowest id server from 2nd highest id process
 3. Election from 2nd highest id server to highest id
 4. Timeout for answers @ 2nd highest id server
 5. Coordinator from 2nd highest id server

Analysis

- **Worst-case** completion time: 5 message transmission times
 - When the process with the lowest id in the system detects the failure.
 - $(N-1)$ processes altogether begin elections, each sending messages to processes with higher ids.
 - i -th highest id process sends $(i-1)$ election messages
 - Number of Election messages
$$= N-1 + N-2 + \dots + 1 = (N-1)*N/2 = O(N^2)$$
- **Best-case**
 - Second-highest id detects leader failure
 - Sends $(N-2)$ Coordinator messages
 - Completion time: 1 message transmission time

Impossibility?

- Since timeouts built into protocol, in asynchronous system model:
 - Protocol may never terminate => Liveness not guaranteed
- But satisfies liveness in synchronous system model where
 - Worst-case one-way latency can be calculated = worst-case process time + worst-case message latency

Summary

- Leader election an important component of many cloud computing systems
- Classical leader election protocols
 - Ring-based
 - Bully
- But failure-prone
 - Paxos-like protocols used by Google Chubby, Apache Zookeeper



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MUTUAL EXCLUSION

Lecture A

INTRODUCTION AND BASICS

Why Mutual Exclusion?

- **Bank's Servers in the Cloud:** Two of your customers make simultaneous deposits of \$10,000 into your bank account, each from a separate ATM.
 - Both ATMs read initial amount of \$1000 concurrently from the bank's cloud server
 - Both ATMs add \$10,000 to this amount (locally at the ATM)
 - Both write the final amount to the server
 - **What's wrong?**

Why Mutual Exclusion?

- **Bank's Servers in the Cloud:** Two of your customers make simultaneous deposits of \$10,000 into your bank account, each from a separate ATM.
 - Both ATMs read initial amount of \$1000 concurrently from the bank's cloud server
 - Both ATMs add \$10,000 to this amount (locally at the ATM)
 - Both write the final amount to the server
 - You lost \$10,000!
- The ATMs need *mutually exclusive* access to your account entry at the server
 - or, mutually exclusive access to executing the code that modifies the account entry

More Uses of Mutual Exclusion

- **Distributed File systems**
 - Locking of files and directories
- **Accessing objects** in a safe and consistent way
 - Ensure at most one server has access to object at any point of time
- **Server coordination**
 - Work partitioned across servers
 - Servers coordinate using locks
- **In industry**
 - Chubby is Google's locking service
 - Many cloud stacks use Apache Zookeeper for coordination among servers

Problem Statement for Mutual Exclusion

- **Critical Section** Problem: Piece of code (at all processes) for which we need to ensure there is at most one process executing it at any point of time.
- Each process can call three functions
 - `enter()` to enter the critical section (CS)
 - `AccessResource()` to run the critical section code
 - `exit()` to exit the critical section

Our Bank Example

ATM1:

```
enter(S);  
// AccessResource()  
obtain bank amount;  
add in deposit;  
update bank amount;  
// AccessResource() end  
exit(S); // exit
```

ATM2:

```
enter(S);  
// AccessResource()  
obtain bank amount;  
add in deposit;  
update bank amount;  
// AccessResource() end  
exit(S); // exit
```

Approaches to Solve Mutual Exclusion

- Single OS:
 - If all processes are running in one OS on a machine (or VM), then
 - Semaphores, mutexes, condition variables, monitors, etc.

Approaches to Solve Mutual Exclusion (2)

- Distributed system:
 - Processes communicating by passing messages

Need to guarantee 3 properties:

- **Safety** (essential) – At most one process executes in CS (CriticalSection) at any time
- **Liveness** (essential) – Every request for a CS is granted eventually
- **Ordering** (desirable) – Requests are granted in the order they were made

Processes Sharing an OS: Semaphores

- Semaphore == an integer that can only be accessed via two special functions
- Semaphore S=1; // Max number of allowed accessors

1. **wait(S)** (or **P(S)** or **down(S)**):

enter()

```
while(1) { // each execution of the while loop is atomic
    if (S > 0) {
        S--;
        break;
    }
}
```

Each while loop execution and S++ are each **atomic** operations – supported via hardware instructions such as compare-and-swap, test-and-set, etc.

exit()

2. **signal(S)** (or **V(S)** or **up(s)**):

S++; // atomic

Our Bank Example Using Semaphores

```
Semaphore S=1; // shared  
ATM1:  
    wait(S);  
    // AccessResource()  
    obtain bank amount;  
    add in deposit;  
    update bank amount;  
    // AccessResource() end  
    signal(S); // exit
```

```
Semaphore S=1; // shared  
ATM2:  
    wait(S);  
    // AccessResource()  
    obtain bank amount;  
    add in deposit;  
    update bank amount;  
    // AccessResource() end  
    signal(S); // exit
```

Next

- In a distributed system, cannot share variables like semaphores
- So how do we support mutual exclusion in a distributed system?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MUTUAL EXCLUSION

Lecture B

DISTRIBUTED MUTUAL EXCLUSION

System Model

- Before solving any problem, specify its System Model:
 - Each pair of processes is connected by reliable channels (such as TCP).
 - Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
 - Processes do not fail.
 - Fault-tolerant variants exist in literature.

Central Solution

- Elect a central master (or leader)
 - Use one of our election algorithms!
- Master keeps
 - A **queue** of waiting requests from processes who wish to access the CS
 - A special **token** which allows its holder to access CS
- Actions of any process in group:
 - **enter()**
 - Send a request to master
 - Wait for token from master
 - **exit()**
 - Send back token to master

Central Solution

- Master Actions:
 - On receiving a request from process P_i
if (master has token)
 Send token to P_i
else
 Add P_i to queue
 - On receiving a token from process P_i
if (queue is not empty)
 Dequeue head of queue (say P_j), send that
 process the token
else
 Retain token

Analysis of Central Algorithm

- Safety – at most one process in CS
 - Exactly one token
- Liveness – every request for CS granted eventually
 - With N processes in system, queue has at most N processes
 - If each process exits CS eventually and no failures, liveness guaranteed
- FIFO Ordering is guaranteed, in order of requests received at master

Analyzing Performance

Efficient mutual exclusion algorithms use fewer messages, and make processes wait for shorter durations to access resources. Three metrics:

- **Bandwidth**: the total number of messages sent in each *enter* and *exit* operation.
- **Client delay**: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)

(We will prefer mostly the enter operation.)

- **Synchronization delay**: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)

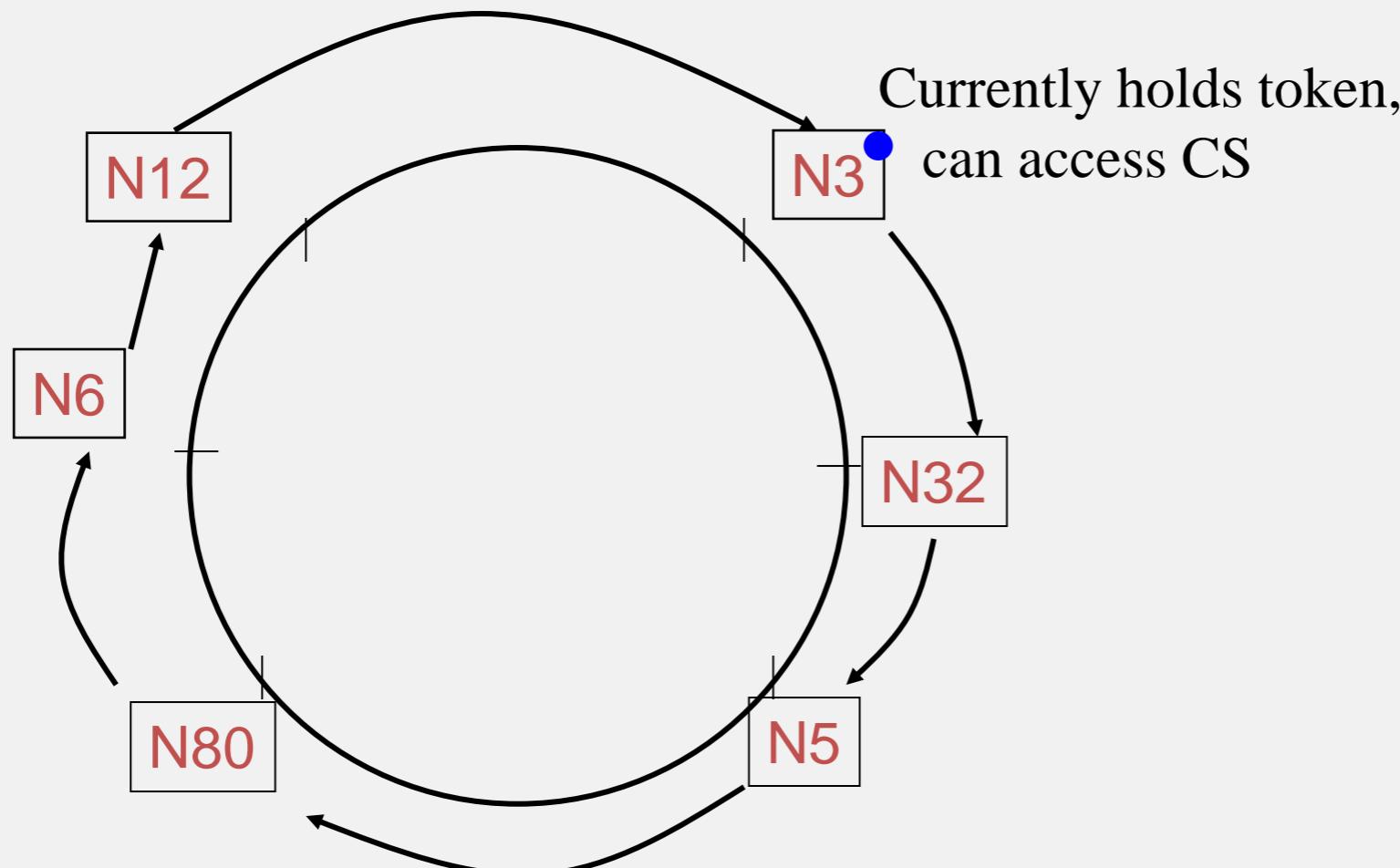
Analysis of Central Algorithm

- **Bandwidth**: the total number of messages sent in each *enter* and *exit* operation.
 - 2 messages for enter
 - 1 message for exit
- **Client delay**: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)
 - 2 message latencies (request + grant)
- **Synchronization delay**: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)
 - 2 message latencies (release + grant)

But...

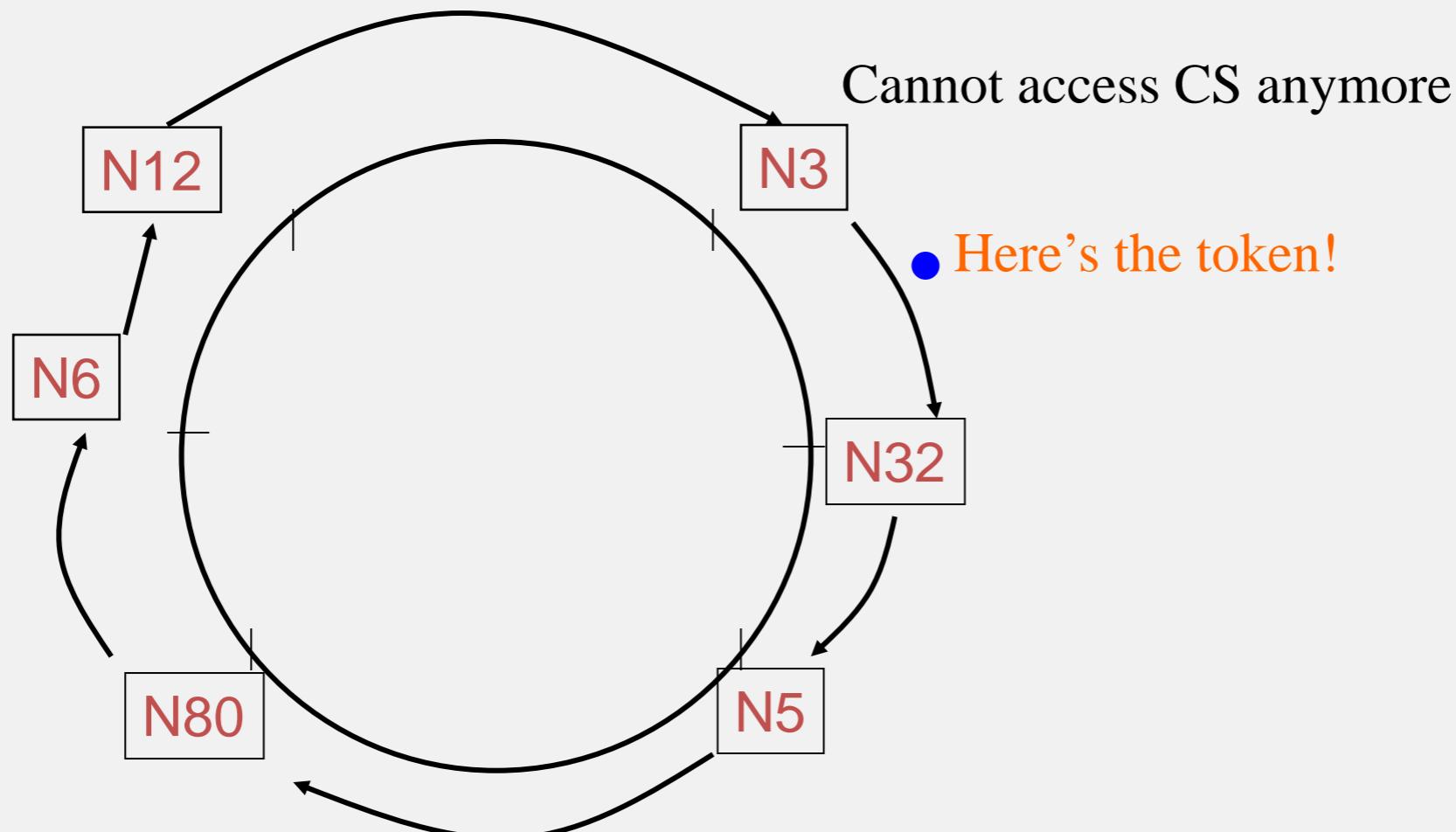
- The master is the performance bottleneck and SPoF (single point of failure)

Ring-based Mutual Exclusion



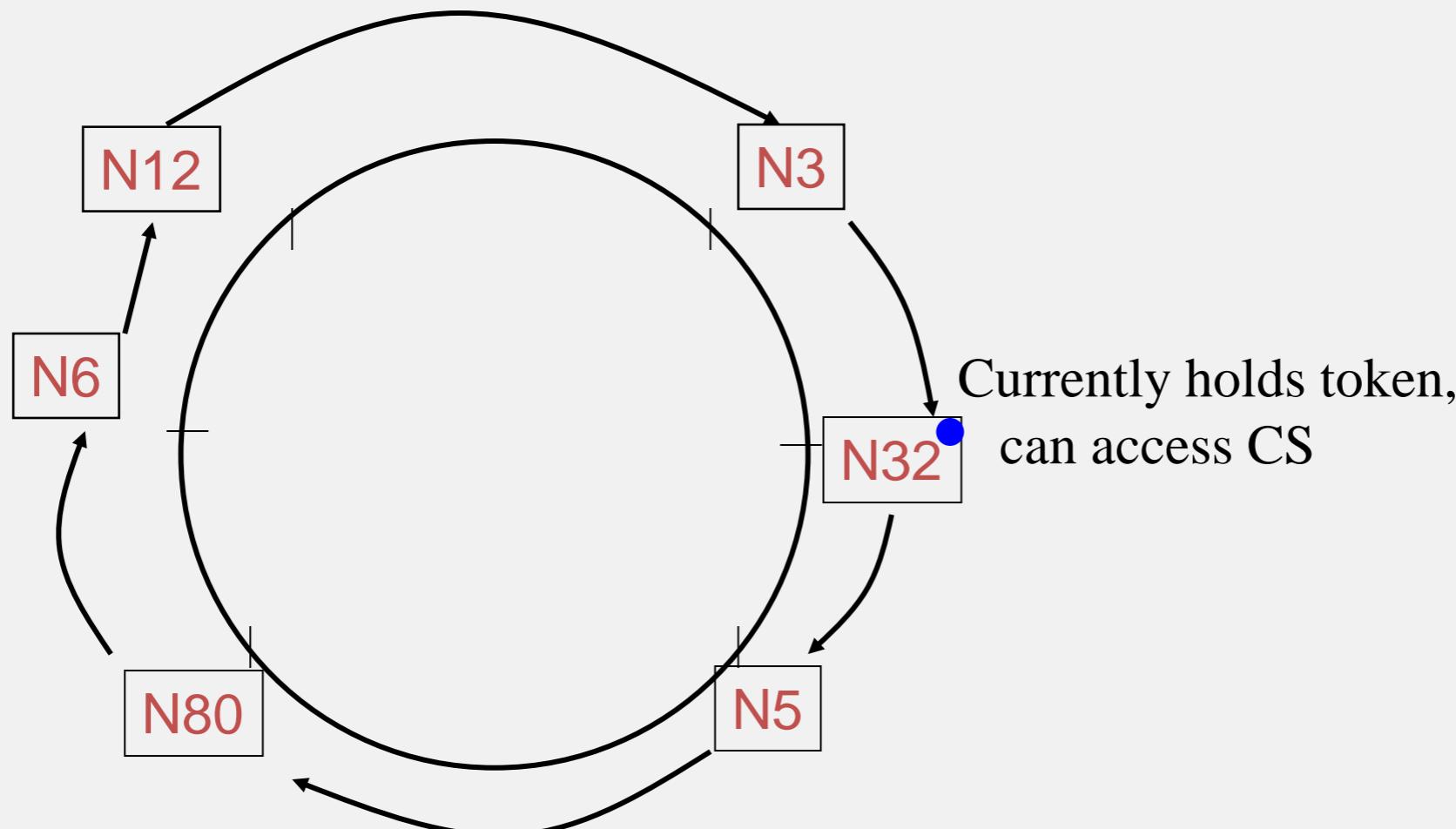
Token: ●

Ring-based Mutual Exclusion



Token: ●

Ring-based Mutual Exclusion



Token: ●

Ring-based Mutual Exclusion

- N Processes organized in a virtual ring
- Each process can send message to its successor in ring
- Exactly 1 token
- enter()
 - Wait until you get token
- exit() // already have token
 - Pass on token to ring successor
- If receive token, and not currently in enter(), just pass on token to ring successor

Analysis of Ring-based Mutual Exclusion

- Safety
 - Exactly one token
- Liveness
 - Token eventually loops around ring and reaches requesting process (no failures)
- Bandwidth
 - Per enter(), 1 message by requesting process but N messages throughout system
 - 1 message sent per exit()

Analysis of Ring-Based Mutual Exclusion (2)

- Client delay: 0 to N message transmissions after entering `enter()`
 - Best case: already have token
 - Worst case: just sent token to neighbor
- Synchronization delay between one process' `exit()` from the CS and the next process' `enter()`:
 - Between 1 and $(N-1)$ message transmissions.
 - Best case: process in `enter()` is successor of process in `exit()`
 - Worst case: process in `enter()` is predecessor of process in `exit()`

Next

- Client/Synchronization delay to access CS still $O(N)$ in Ring-Based approach.
- Can we lower this?



CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MUTUAL EXCLUSION

Lecture C

RICART-AGRAWALA'S ALGORITHM

System Model

- Before solving any problem, specify its System Model:
 - Each pair of processes is connected by reliable channels (such as TCP).
 - Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
 - Processes do not fail.

Ricart-Agrawala's Algorithm

- Classical algorithm from 1981
- Invented by Glenn Ricart (NIH) and Ashok Agrawala (U. Maryland)
- No token
- Uses the notion of causality and multicast
- Has lower waiting time to enter CS than Ring-Based approach

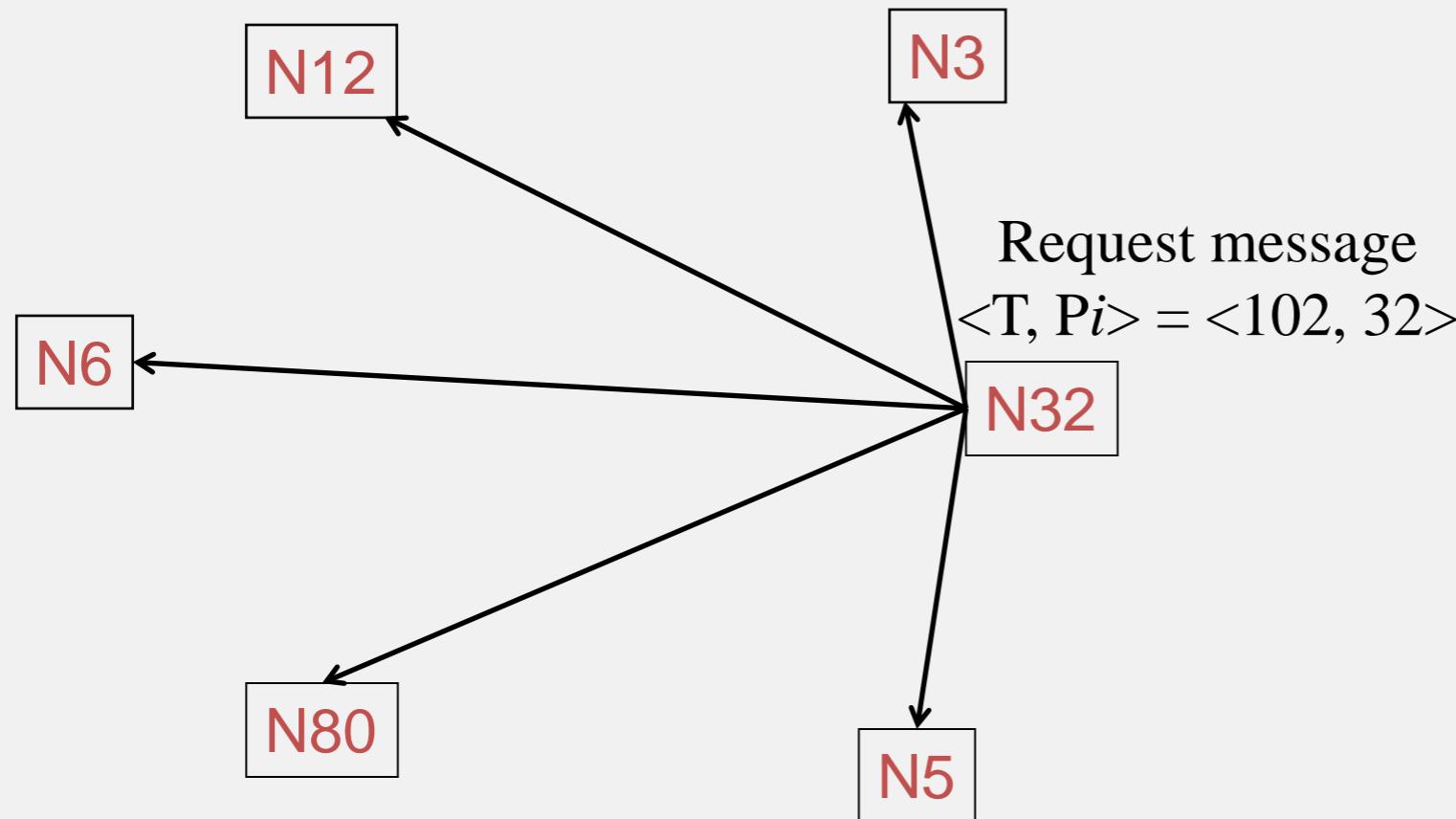
Key Idea: Ricart-Agrawala Algorithm

- enter() at process P_i
 - multicast a request to all processes
 - Request: $\langle T, P_i \rangle$, where $T =$ current Lamport timestamp at P_i
 - Wait until *all* other processes have responded positively to request
- Requests are granted in order of causality
- P_i in request $\langle T, P_i \rangle$ is used to break ties (since Lamport timestamps are not unique for concurrent events)

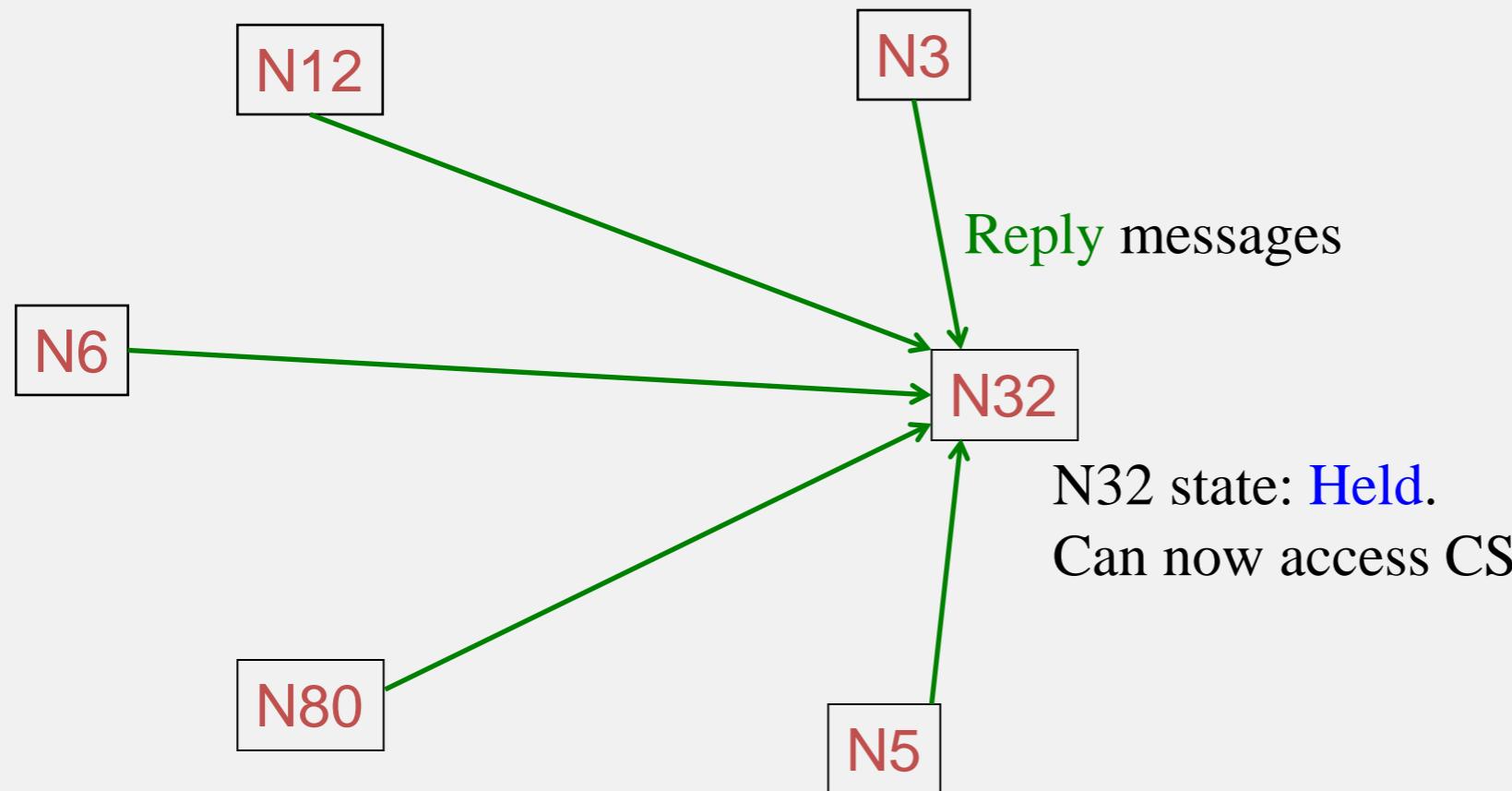
Messages in RA Algorithm

- enter() at process P_i
 - set state to Wanted
 - multicast “Request” $\langle T_i, P_i \rangle$ to all processes, where $T_i =$ current Lamport timestamp at P_i
 - wait until all processes send back “Reply”
 - change state to Held and enter the CS
- On receipt of a Request $\langle T_j, P_j \rangle$ at $P_i (i \neq j)$:
 - if (state = Held) or (state = Wanted & $(T_i, i) < (T_j, j)$)
// lexicographic ordering in (T_j, P_j)
add request to local queue (of waiting requests)
else send “Reply” to P_j
- exit() at process P_i
 - change state to Released and “Reply” to all queued requests.

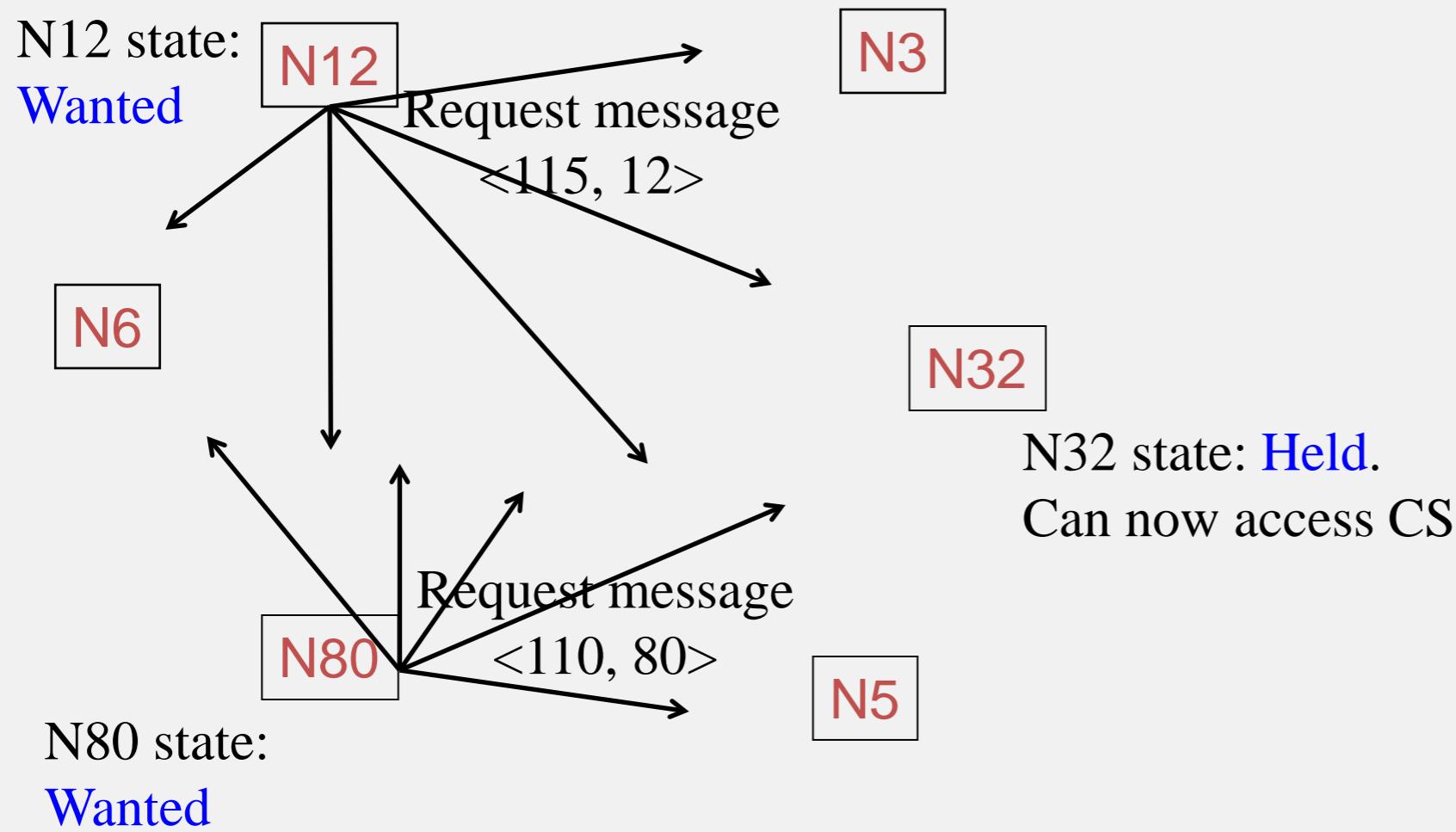
Example: Ricart-Agrawala Algorithm



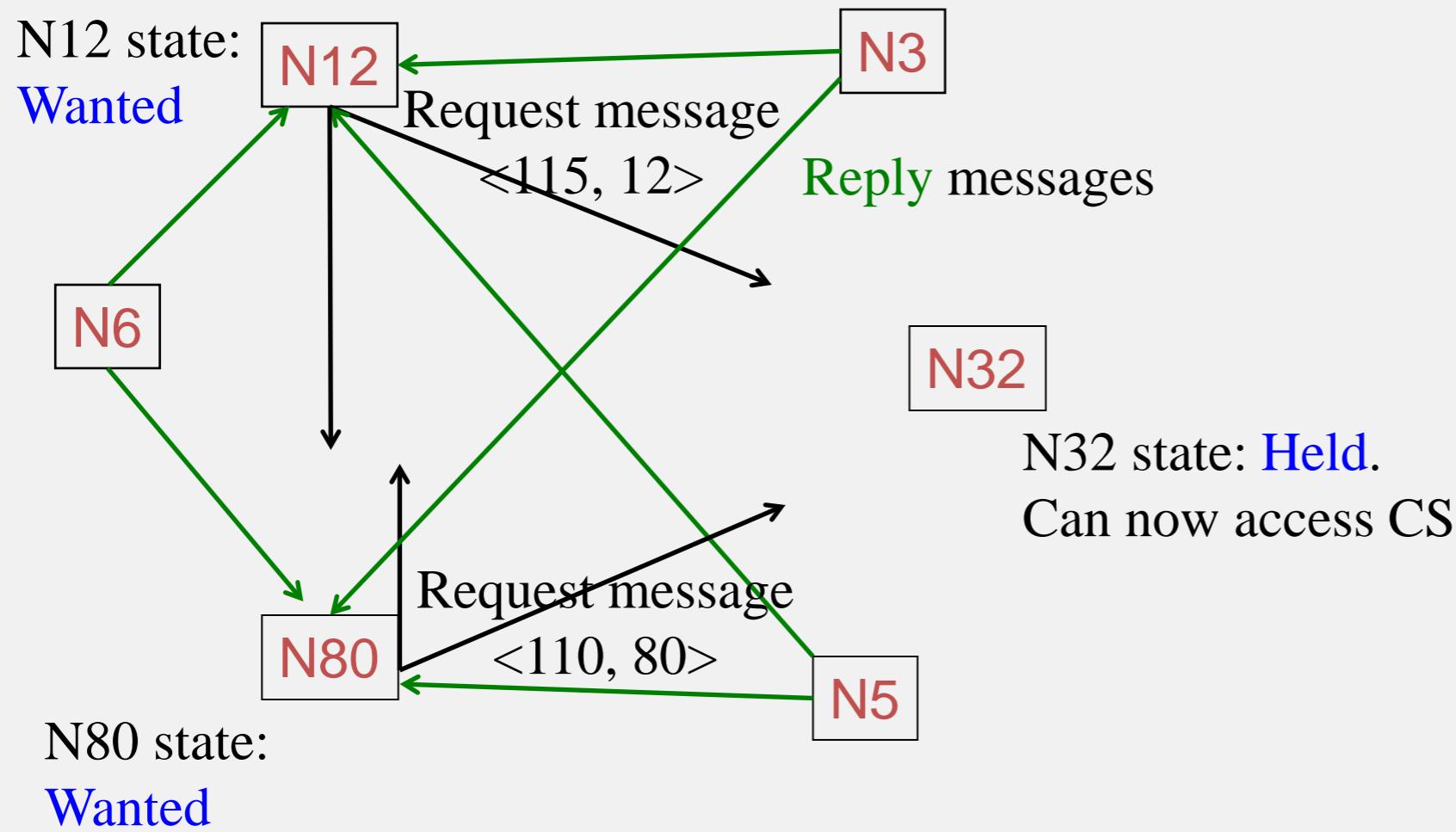
Example: Ricart-Agrawala Algorithm



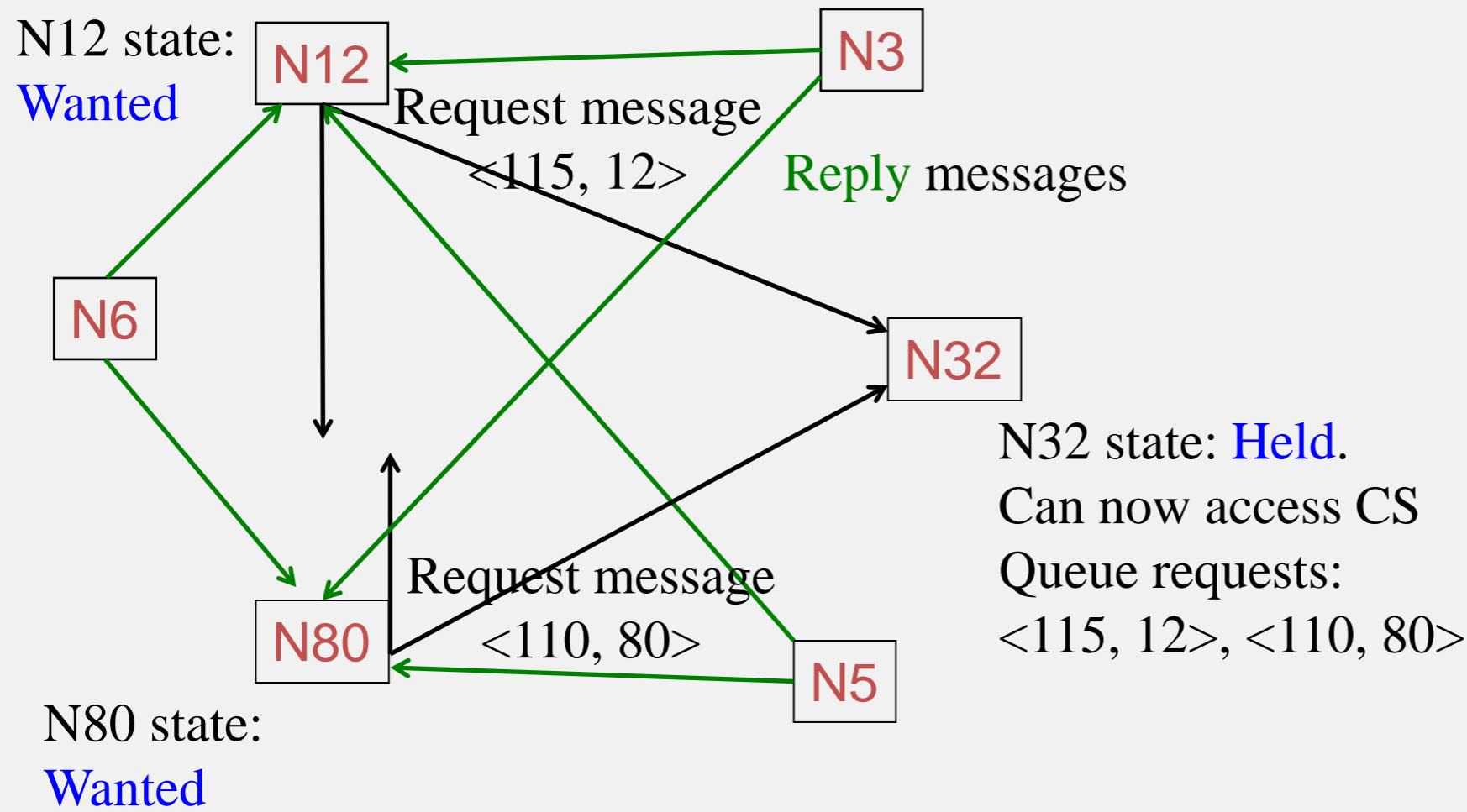
Example: Ricart-Agrawala Algorithm



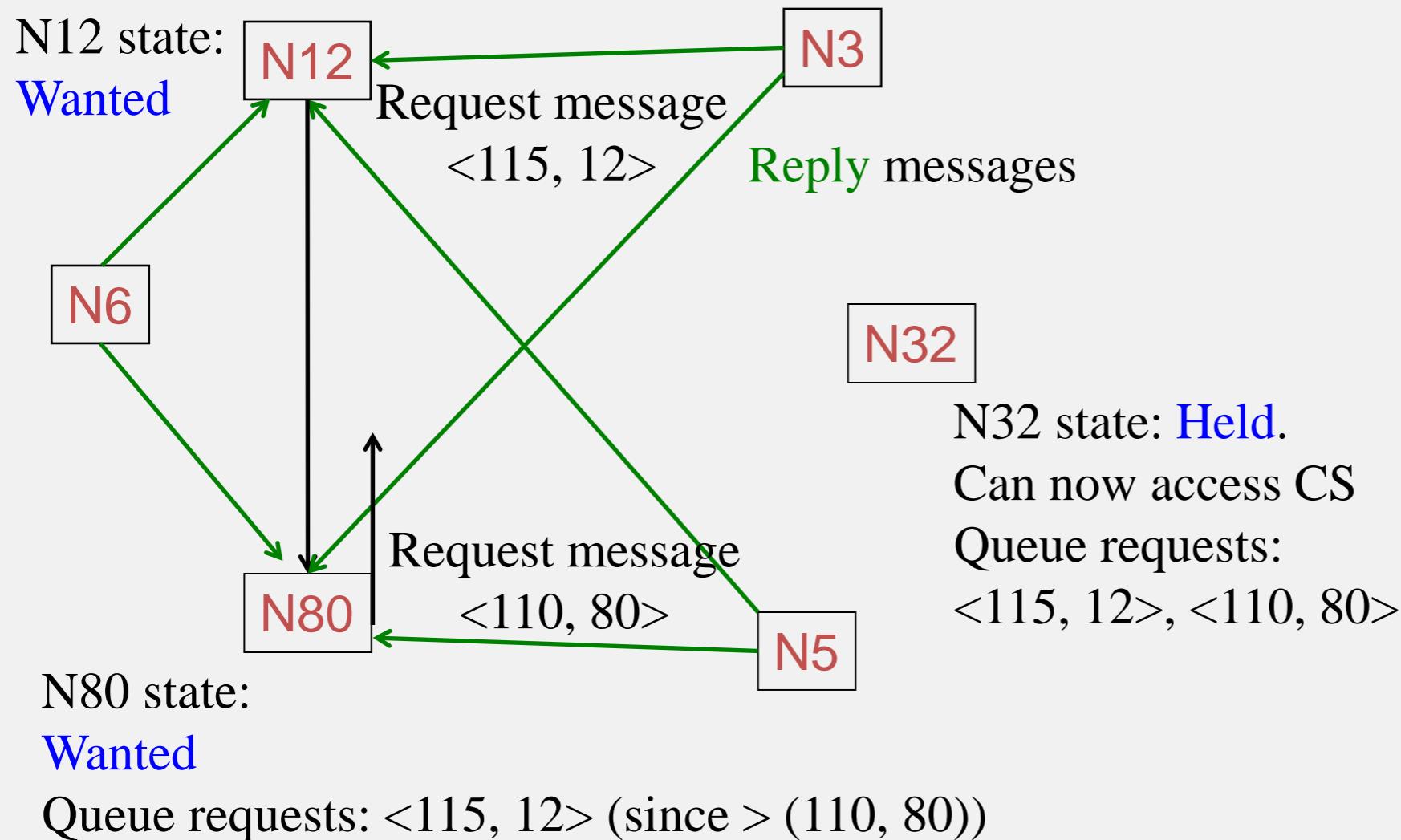
Example: Ricart-Agrawala Algorithm



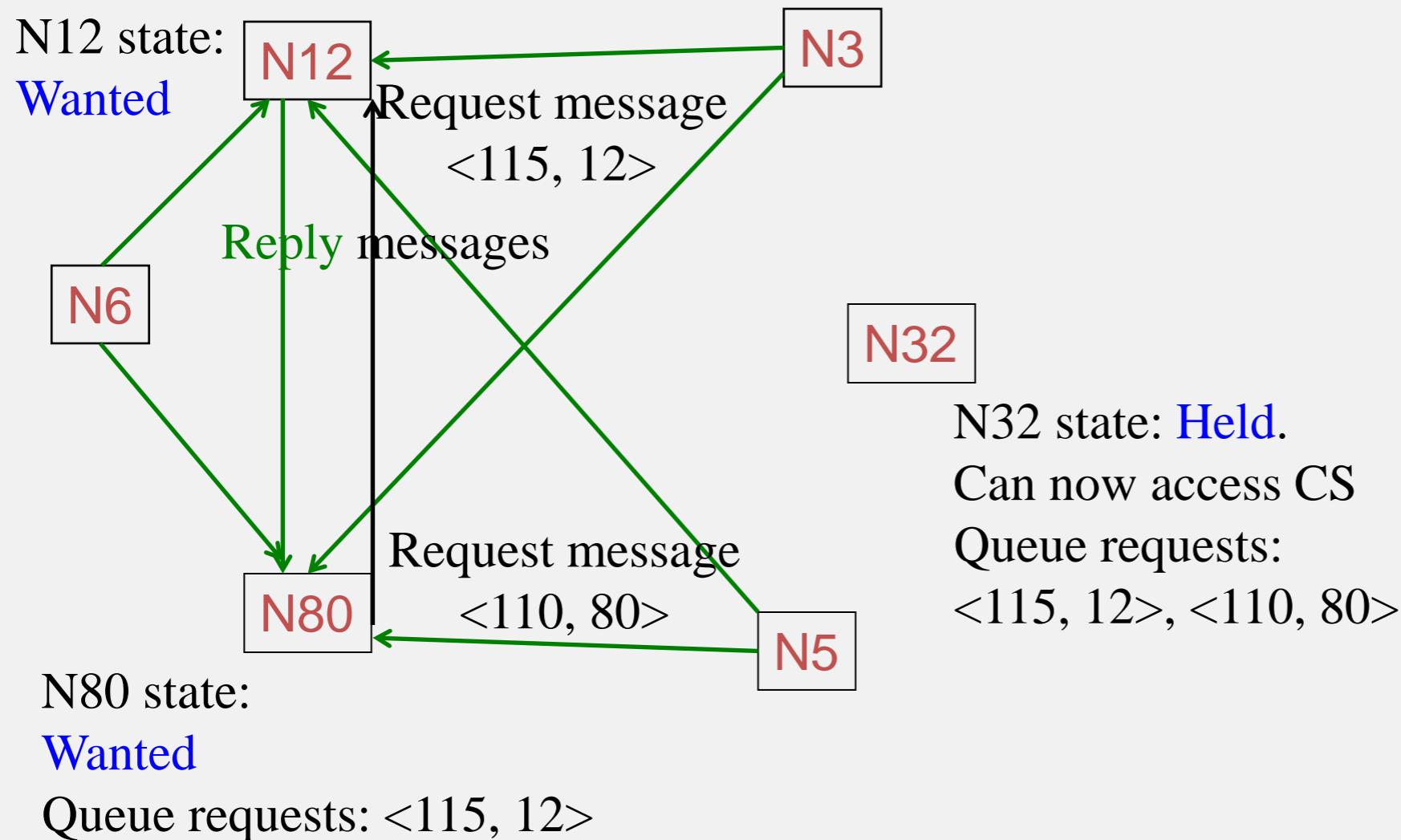
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm



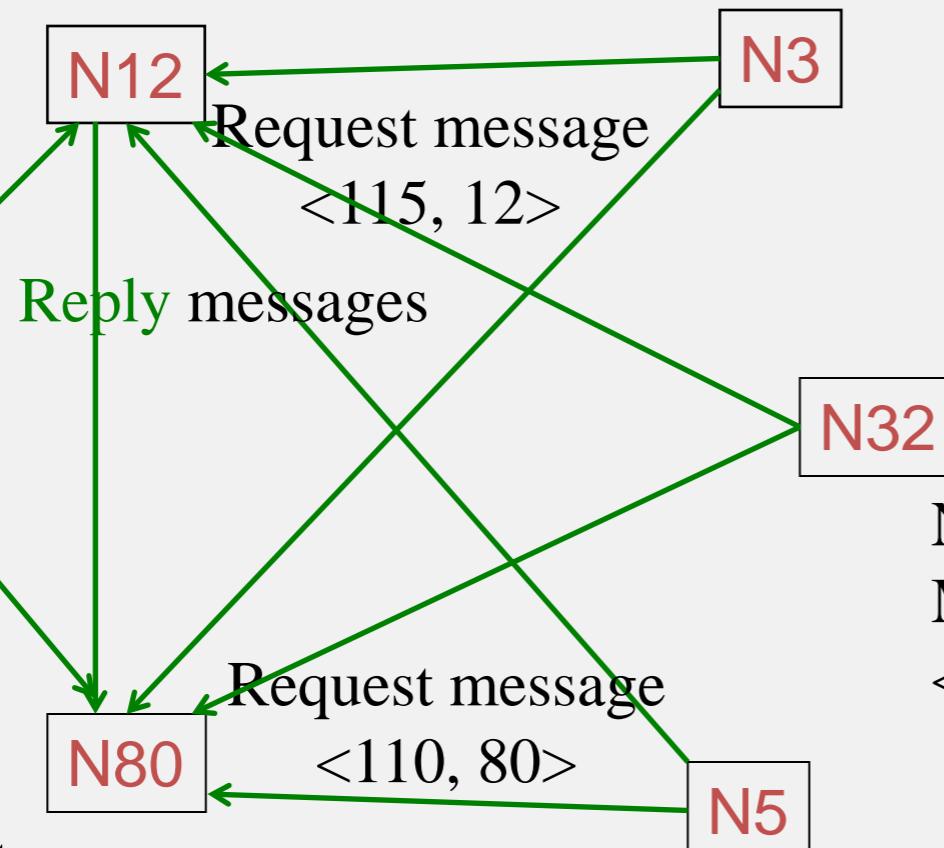
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm

N12 state:
Wanted
(waiting for
N80's
reply)

N6



N80 state:
Held. Can now access CS.
Queue requests: $\langle 115, 12 \rangle$

N32 state: **Released.**
Multicast Reply to
 $\langle 115, 12 \rangle, \langle 110, 80 \rangle$

Analysis: Ricart-Agrawala's Algorithm

- Safety
 - Two processes P_i and P_j cannot both have access to CS
 - If they did, then both would have sent Reply to each other
 - Thus, $(T_i, i) < (T_j, j)$ and $(T_j, j) < (T_i, i)$, which are together not possible
 - What if $(T_i, i) < (T_j, j)$ and P_i replied to P_j 's request before it created its own request?
 - Then it seems like both P_i and P_j would approve each others' requests
 - But then, causality and Lamport timestamps at P_i implies that $T_i > T_j$, which is a contradiction
 - So this situation cannot arise

Analysis: Ricart-Agrawala's Algorithm (2)

- Liveness
 - Worst-case: wait for all other ($N-1$) processes to send Reply
- Ordering
 - Requests with lower Lamport timestamps are granted earlier

Performance: Ricart-Agrawala's Algorithm

- Bandwidth: $2*(N-1)$ messages per enter() operation
 - $N-1$ unicasts for the multicast request + $N-1$ replies
 - N messages if the underlying network supports multicast
 - $N-1$ unicast messages per exit operation
 - 1 multicast if the underlying network supports multicast
- Client delay: one round-trip time
- Synchronization delay: one message transmission time

Ok, but ...

- Compared to Ring-Based approach, in Ricart-Agrawala approach
 - Client/synchronization delay has now gone down to $O(1)$
 - But bandwidth has gone up to $O(N)$
- Can we get *both* down?

CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

MUTUAL EXCLUSION

Lecture D

MAEKAWA'S ALGORITHM
AND WRAP-UP

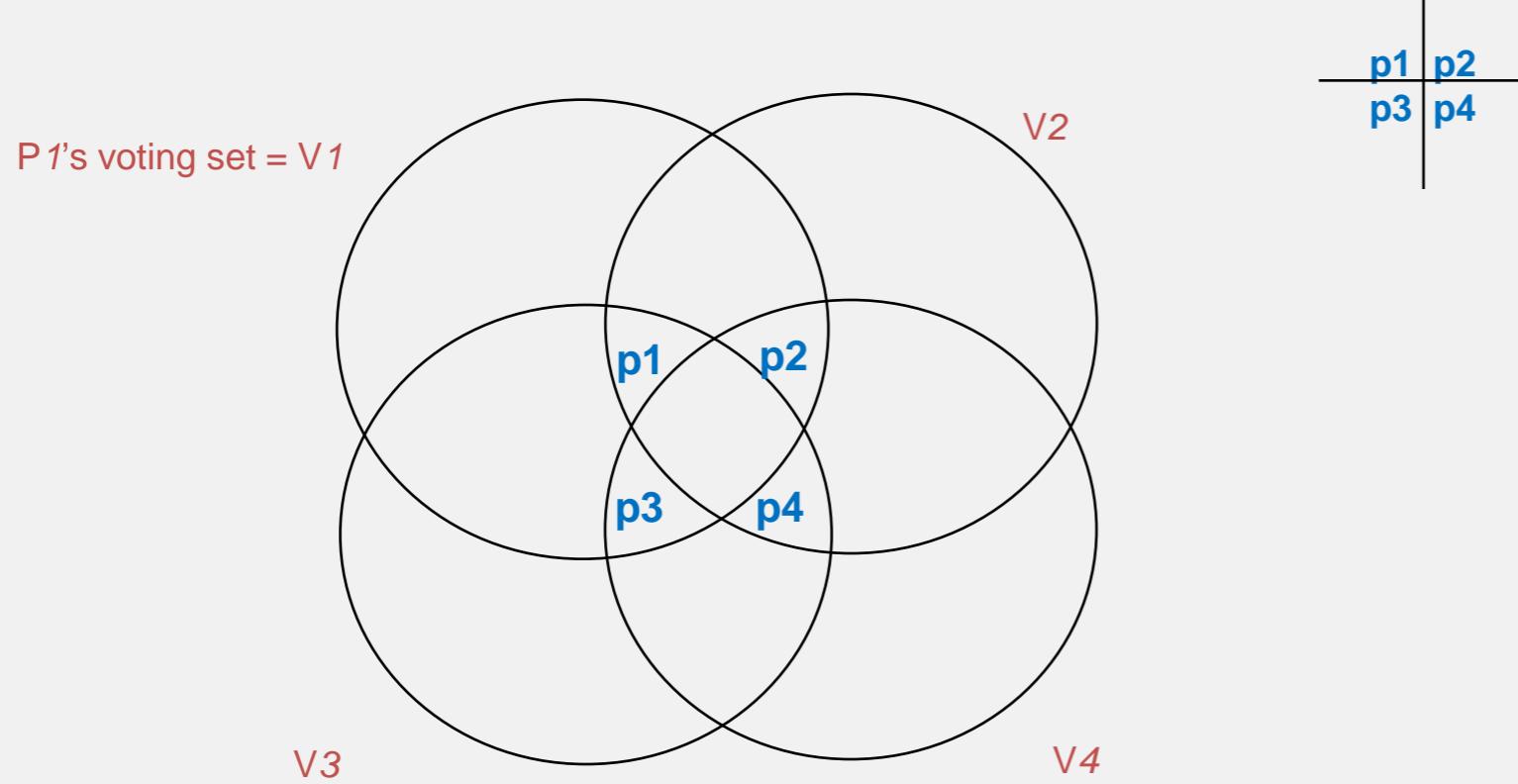
Key Idea

- Ricart-Agrawala requires replies from *all* processes in group
- Instead, get replies from only *some* processes in group
- But ensure that only one process is given access to CS (CriticalSection) at a time

Maekawa's Voting Sets

- Each process P_i is associated with a voting set V_i (of processes)
- Each process belongs to its own voting set
- *The intersection of any two voting sets must be non-empty*
 - *Same concept as Quorums!*
- Each voting set is of size K
- Each process belongs to M other voting sets
- Maekawa showed that $K=M=\sqrt{N}$ works best
- One way of doing this is to put N processes in a \sqrt{N} by \sqrt{N} matrix and for each P_i , its voting set $V_i = \text{row containing } P_i + \text{column containing } P_i$. Size of voting set = $2*\sqrt{N}-1$

Example: Voting Sets with N=4



p1	p2
p3	p4

Maekawa: Key Differences From Ricart-Agrawala

- Each process requests permission from only its voting set members
 - Not from all
- Each process (in a voting set) gives permission to at most one process at a time
 - Not to all

Actions

- state = Released, voted = false
- enter() at process P_i :
 - state = Wanted
 - Multicast **Request** message to all processes in V_i
 - Wait for **Reply (vote)** messages from all processes in V_i (including vote from self)
 - state = Held
- exit() at process P_i :
 - state = Released
 - Multicast **Reply** to all processes in V_i

Actions (2)

- When P_i receives a request from P_j :
if (state == Held OR voted = true)
 queue request
else
 send **Reply** to P_j and set voted = true
- When P_i receives a Reply from P_j :
if (queue empty)
 voted = false
else
 dequeue head of queue, say P_k
 Send **Reply** *only* to P_k
 voted = true

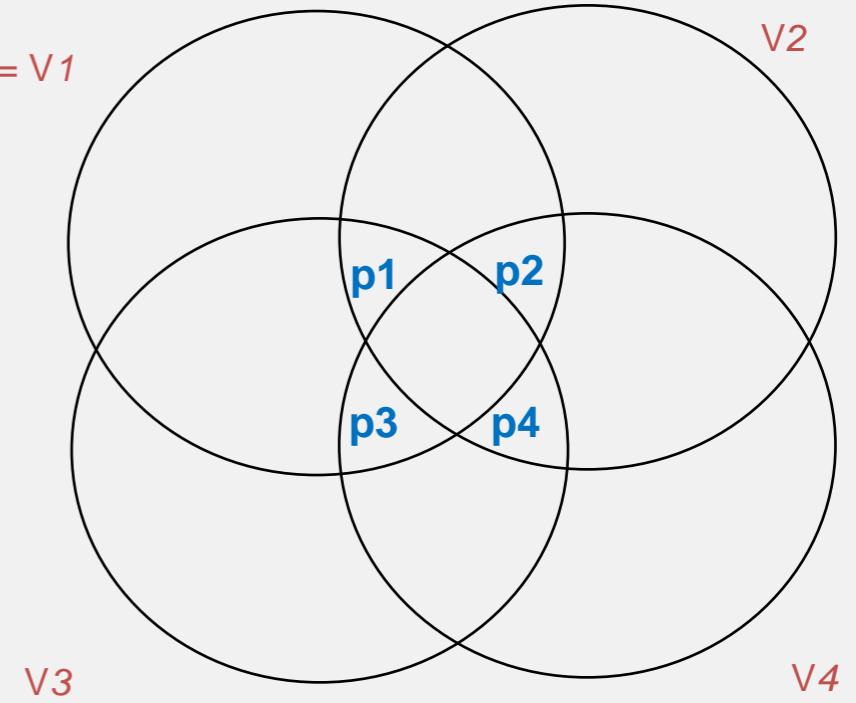
Safety

- When a process P_i receives replies from all its voting set V_i members, no other process P_j could have received replies from all its voting set members V_j
 - V_i and V_j intersect in at least one process say P_k
 - But P_k sends only one Reply (vote) at a time, so it could not have voted for both P_i and P_j

Liveness

- A process needs to wait for at most ($N-1$) other processes to finish CS
- But does not guarantee liveness
- Since can have a *deadlock*
- Example: all 4 processes need access
 - P1 is waiting for P3
 - P3 is waiting for P4
 - P4 is waiting for P2
 - P2 is waiting for P1
 - No progress in the system!
- There are deadlock-free versions

P1's voting set = V1



Performance

- Bandwidth
 - $2\sqrt{N}$ messages per enter()
 - \sqrt{N} messages per exit()
 - Better than Ricart and Agrawala's ($2*(N-I)$ and $N-I$ messages)
 - \sqrt{N} quite small. $N \sim 1$ million $\Rightarrow \sqrt{N} = 1K$
- Client delay: One round trip time
- Synchronization delay: 2 message transmission times

Why \sqrt{N} ?

- Each voting set is of size K
- Each process belongs to M other voting sets
- Total number of voting set members (processes may be repeated) = $K*N$
- But since each process is in M voting sets
 - $K*N/M = N \Rightarrow K = M$ (1)
- Consider a process P_i
 - Total number of voting sets = members present in P_i 's voting set and all their voting sets = $(M-1)*K + 1$
 - This must equal the number of processes
 - To minimize the overhead at each process (K), need each of the above members to be unique, i.e.,
 - $N = (M-1)*K + 1$
 - $N = (K-1)*K + 1$ (due to (1))
 - $K \sim \sqrt{N}$

Failures?

- There are fault-tolerant versions of the algorithms we've discussed
 - E.g., Maekawa
- One other way to handle failures: Use Paxos-like approaches!

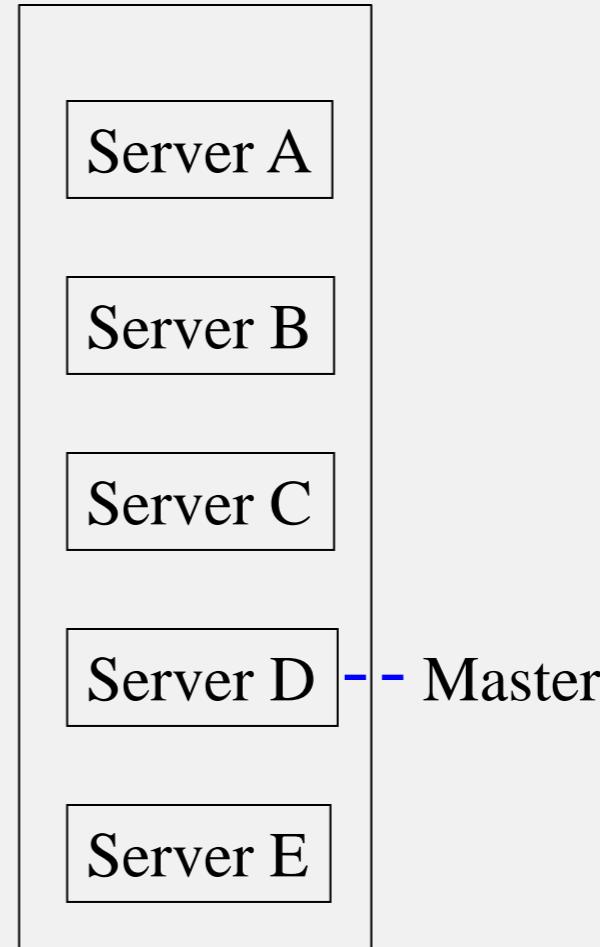
Chubby

- Google's system for locking
- Used underneath Google's systems like BigTable, Megastore, etc.
- Not open-sourced but published
- Chubby provides *Advisory* locks only
 - Doesn't guarantee mutual exclusion unless every client checks lock before accessing resource

Reference: <http://research.google.com/archive/chubby.html>

Chubby (2)

- Can use not only for locking but also writing small configuration files
- Relies on Paxos
- Group of servers with one elected as Master
 - All servers replicate same information
- Clients send read requests to Master, which serves it locally
- Clients send write requests to Master, which sends it to all servers, gets majority (quorum) among servers, and then responds to client
- On master failure, run election protocol
- On replica failure, just replace it and have it catch up



Summary

- Mutual exclusion important problem in cloud computing systems
- Classical algorithms
 - Central
 - Ring-based
 - Ricart-Agrawala
 - Maekawa
- Industry systems
 - Chubby: a coordination service
 - Similarly, Apache Zookeeper for coordination