

EECE 411: Design of Distributed Software Applications
(or Distributed Systems 101)

Matei Ripeanu

<http://www.ece.ubc.ca/~matei>

Today's Objectives

- Class mechanics
 - <http://www.ece.ubc.ca/~matei/EECE411/>
- Understand the sources of complexity for system design

What this class IS about

Today's large-scale computing systems today are built using a common set of *core techniques, algorithms, and design philosophies*— all centered around distributed systems.

Goals

- Understand how these systems are designed
 - and become familiar with most widely-used building blocks
- Get your hands dirty using these concepts
 - Large hands-on component.

What this class IS about

Today's computing systems are built using a common set of *core techniques, algorithms, and design philosophies* –all centered around distributed systems.

Topics include:

- key-value stores, map-reduce, clouds
- Classical precursors: algorithms, P2P, shared filesystems
- Scalability
- Trending areas
- And more!

What I'll assume you know

- You are familiar with basic notions of network protocol design, system-level programming
 - Basic notions of parallelism, synchronization, TCP/UDP
- You are familiar and productive writing Java code
 - IDE, code management
 - Working in a group
- You know (or you are able to quickly learn) a scripting language (e.g., bash, C shell, Python) and command line interaction with UNIX-like systems

If there are things that don't make sense, ask!

What this class is NOT about

An analogy: if this course were about cars,

- You'll learn about the physics relating to the internals of the car (friction, transmission), basics about the internals of the car (carburetor, engine), and how successful designs have assembled things together.
- You'll NOT learn how to drive a car, accident statistics, or about how roads are built.

You will NOT learn how to use cloud computing systems or about networking or Big Data.

Course Organization/Syllabus/etc.

Back to class mechanics

Course page: <http://www.ece.ubc.ca/~matei/EECE411/>

Email: matei @ ece.ubc.ca; Office: KAIS 4033

Office hours: after each class OR stop by (see ‘open door policy’ below)
OR by appointment (email me)

Open door policy



Schedule

(Note: the dates are tentative)

- **Class schedule:** MWF 3-4pm
 - Note: Fri: different location
- **Likely travel:** February 13th, March 13th
 - Make-up time: We'll schedule project delivery in the exam period
- **Midterm(s):** February 11th, March 27th
- **Labs:** Wed 6-8pm in MCLD 348
- Department ‘distinguished lecture’ series: Mondays 4pm

Administravia: Grading 50/50

- Exams: final, midterm: 50%
- Assignments: project, class assignments: 50%
- Bonus points: class participation, mailing list etc

Project

Design, build, and operate a large-scale distributed application.

- Issues to worry about: scalability, reliability, efficient use of resources, easy to operate, reuse,

Context: Large-scale deployment platform.
(PlanetLab)

Context: Limited handheld

Groups: up to 4 students.

- TO DO: Start thinking
about your group.



Project details

Distributed key/value store.

- put(key, value); get (value) interface.
- Widely adopted building block: think [memcached](#)

Want to do your own project?

- Develop evaluation infrastructure for memcached project!
 - Adversarial learning ☺
- Others

Project steps

- *P1. Warmup:* Develop a minimal client application, implement client protocol
- *P2. Warmup:* Familiarize yourself with PlanetLab, setup the environment, develop a monitoring service
- *P3. Minimal service.* Develop and test the service at minimal scale
- *P4-... Adding features (e.g., high reliability, scalability, throughput) and consistency model*

2014 project [description](#)

- Options: Single node vs. distributed; Blocking vs. non-blocking IO; etc

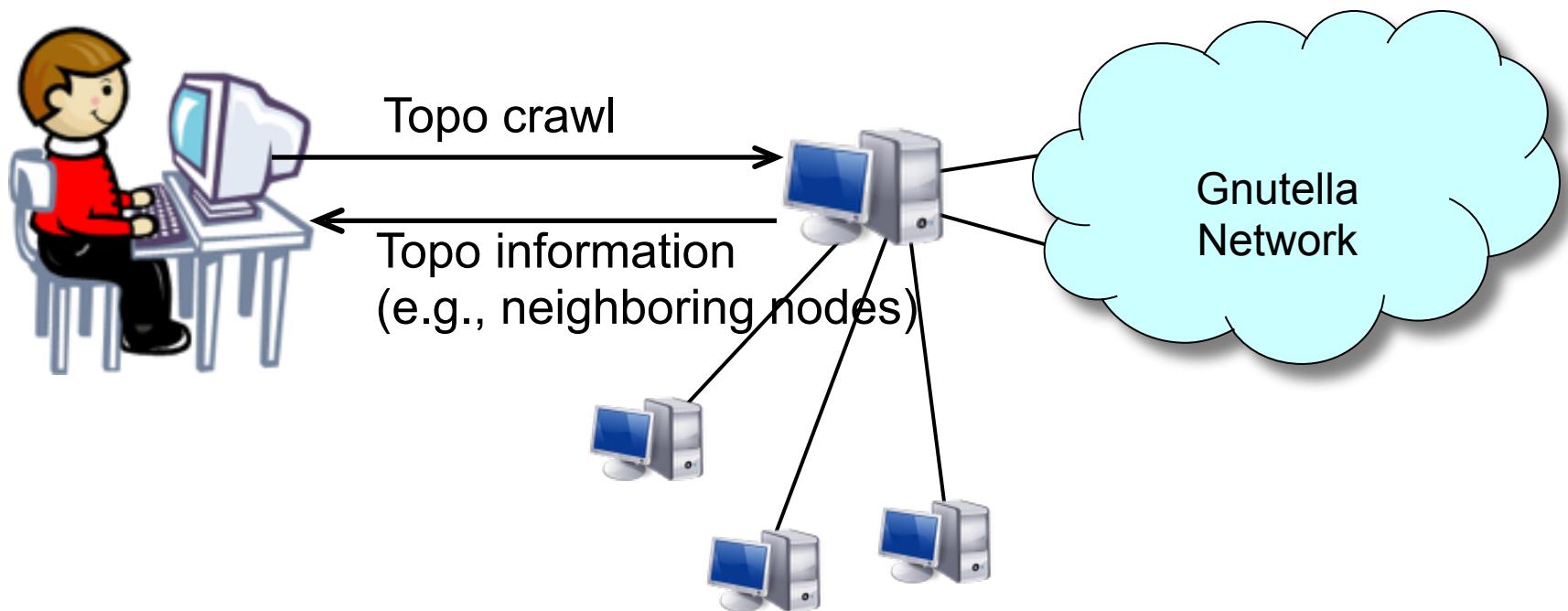
Alternatives: Your own project

Goal: Design, build, and operate a large-scale distributed application

Some ideas

- *Crawl and analyze data from other p2p or social networks:*
 - e.g., Twitter, Skype, YouTube
 - *Hard*: closed protocols (e.g., Skype)
 - *Cool*: no (or few) independent measurements
- *Characterize Amazon service performance*: e.g., S3
 - Performance: latency, throughput, consistency
 - multiple vantage points (migration?)
 - *Hard*: limited budget (!), black box
 - *Cool*: real, well engineered service, huge scale
- *Others*:
 - location services,
 - Network health monitoring

One past idea: recursively crawl the entire Gnutella network



Support provided: libraries, bootstrap node

Today's Objectives

- Class mechanics

Your TO DOs

- Subscribe to Piazza (see info on [webpage](#))
- Start thinking about project groups

QUESTIONS?

Next: Sources of complexity for system design

Constraints for Computer System Designs

Not like those for “real” engineers:
Weight, physics, etc.

Complexity of what we can understand

Challenge: managing complexity to efficiently produce *reliable, scalable, secure* [add your attribute here] software systems.

Challenge: coping w. complexity

- Hard to define; symptoms:
 - Large # of components
 - Large # of connections
 - Irregular
 - No short description
 - Many people required to design/maintain
- Technology rarely the limit!
 - Indeed tech opportunity can be a problem
 - Limit is usually designers' understanding

Problem Types in Complex Systems

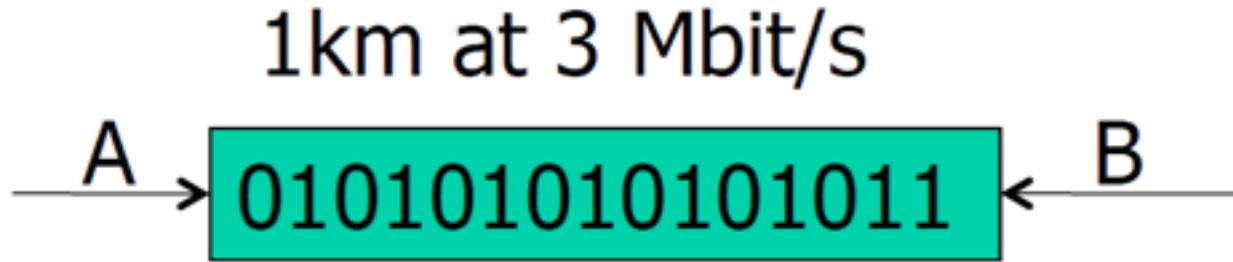
- Emergent properties
 - Surprises
- Propagation of effects
 - Small change -> big effect
- [Incommensurate] scaling
 - Design for small model may not scale
- Tradeoffs

Note: These problems show up in non-computer systems as well

Emergent Property Example: Ethernet

- All computers share single cable
- Goal is reliable delivery
- Listen before send to avoid collisions

Will listen-while-send detect collisions?



- 1 km at 60% speed of light = 5 microseconds
 - A can send 15 bits before bit 1 arrives at B
- A must keep sending for $2 * 5$ microseconds
 - To detect collision if B sends when bit 1 arrives
- Minimum packet size is $5 * 2 * 3 = 30$ bits

3 Mbit/s -> 10 Mbit/s

- Experimental Ethernet design: 3Mbit/s
 - Default header is: 5 bytes = 40 bits
 - No problem with detecting collisions
- First Ethernet standard: 10 Mbit/s
 - Must send for $2 \times 20 \mu\text{seconds} = 400 \text{ bits} = 50 \text{ bytes}$
 - But header is 14 bytes
- Need to pad packets to at least 50 bytes
 - Minimum packet size!

Propagation of Effects Example (L. Cole 1969)

- Attempt to control malaria in North Borneo
 - ...
- Sprayed villages with DDT
- Wiped out mosquitoes, but
- Roaches collected DDT in tissue
- Lizards ate roaches and became slower
- Easy target for cats
- Cats didn't deal with DDT well and died
- Forest rats moved into villages
- Rats carried the bacillus for the plague
- ... Leads to replacing malaria with the plague

Incommensurate scaling example

Mouse -> Elephant (Haldane 1928)

- Mouse has a particular skeleton design
 - Can one scale it to something big?
- Scaling mouse to size of an elephant
 - Weight ~ Volume ~ $O(n^3)$
 - Bone strength ~ cross section ~ $O(n^2)$
- Mouse design will collapse
- Elephant needs different design than mouse!

Incommensurate scaling example

- Scaling Ethernet's bit-rate
 - 10 mbit/s: min packet 64 bytes, max cable 2.5 km
 - 100: 64 bytes, 0.25 km
 - 1,000: 512 bytes, 0.25 km
 - 10,000: no shared cable

Sources of Complexity

- Many goals/requirements
 - Interaction of features
- Requirements for Performance/High utilization

Example: More goals → more complexity

1975 Unix kernel:

10,500 lines of code

2008 Linux 2.6.24 line counts:

85,000 processes

430,000 sound drivers

490,000 network protocols

710,000 file systems

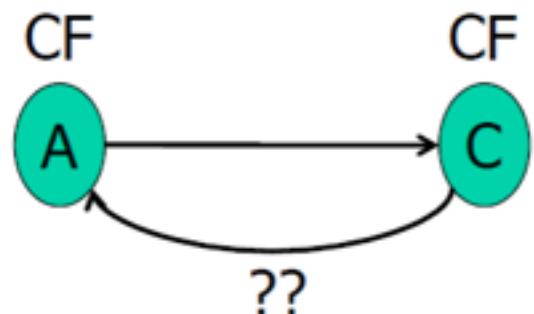
1,000,000 different CPU architectures

4,000,000 drivers

7,800,000 Total

Example: Interacting features → more complexity

- Call Forwarding



- Call Number Delivery Blocking
- Automatic Call Back
- Itemized Billing

CNDB



ACB + IB



- A calls B, B is busy
- Once B is done, B calls A
- A's number on appears on B's bill

Interacting Features

- The point is not that these interactions can't be fixed ...
- ... but that there are so many interactions that have to be considered: they are a huge source of complexity.
 - Perhaps more than n^2 interactions,
- Cost of thinking about / fixing interaction gradually grows to dominate s/w costs.
- Complexity is super-linear

Example: More performance -> more complexity

- One track in a narrow canyon
- Base design: alternate trains
 - Low throughput, high delay, low utilization
 - Worse than two-track, cheaper than blasting
- Improved design: Increase utilization w/ a siding and two trains
 - Precise schedule
 - Risk of collision / signal lights
 - *Siding limits train length (a global effect!)*

Cost of increased ‘performance’ is supra-linear

Summary of examples

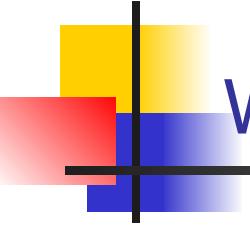
- Expect surprises
- There is no small change
- 10x increase \Rightarrow perhaps re-design
- Just one more feature!
- Performance cost is super-linear

Coping with Complexity

- Simplifying insights / experience / principles
 - E.g., “Avoid excessive generality”
- Modularity
 - Split up system, consider separately
- Abstraction
 - Interfaces/hiding,
 - Helps avoid propagation of effects
- Hierarchy
 - Reduce connections
 - Divide-and-conquer
- Layering
 - Gradually build up capabilities
 - Reduce connections

Summary

- Class mechanics
 - <http://www.ece.ubc.ca/~matei/EECE411/>
- Understand the sources of complexity for system design



What is a Distributed System?

- You know when you have one ...
... when the failure of a computer you've never heard of stops you from getting any work done

(L.Lamport, '84)

Developing di

- The network is perfect
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- The network is secure
- The topology does not change
- There is one and only one path
- The network is reliable

Typical first year for a new cluster:

- ~1 **network rewiring** (rolling ~5% of machines down over 2-day span)
- ~20 **rack failures** (40-80 machines instantly disappear, 1-6 hours to get back)
- ~5 **racks go wonky** (40-80 machines see 50% packetloss)
- ~8 **network maintenances** (4 might cause ~30-minute random connectivity losses)
- ~12 **router reloads** (takes out DNS and external vips for a couple minutes)
- ~3 **router failures** (have to immediately pull traffic for an hour)
- ~dozens of minor **30-second blips for dns**
- ~1000 **individual machine failures**
- ~thousands of **hard drive failures**
- slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.



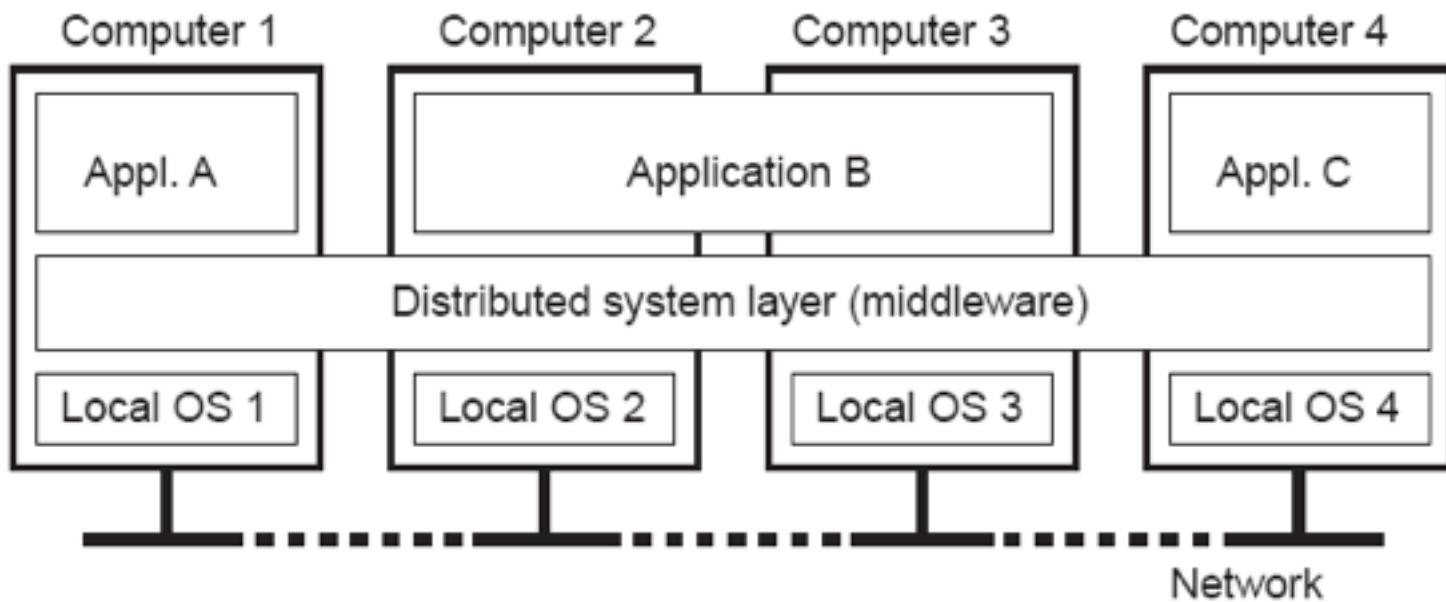
**Your network link: 10Gbps, 50ms
Transport protocol: TCP**

**How long it will take to send a
100KB message?**

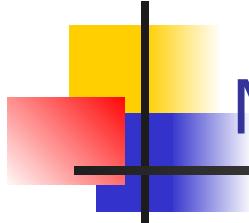
[aka Peter Deutsch's]

What is a Distributed System?

A collection of independent computers that appears to its users as a **single coherent system**

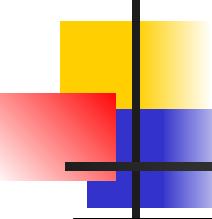


- Independent hardware installations
- Uniform software layer (**middleware**)



Main goals of a distributed system

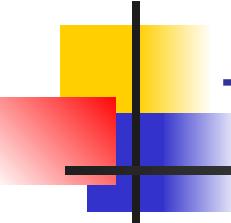
- Connect users and resources
- Distribution transparency
- Openness
- Scalability



Goal II: Transparency

Transparency	Description
Access	Hide differences in data representation and resource access
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may migrate while in use
Replication	Hide that a resource may have multiple copies
Concurrency	Hide that a resource may be shared by several competing users
Failure	Hide the failure and recovery of a resource

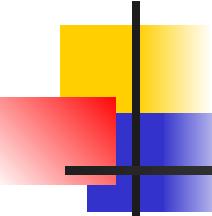
Note: transparency may be set as a goal, but achieving it is a different story.



Transparency – discussion

Observation: Aiming at full transparency may be too much:

- Full transparency will **cost performance**
 - Keeping Web caches exactly up-to-date with the master copy
 - Immediately flushing write operations to disk for fault tolerance
- Completely hiding failures of networks and nodes may be **impossible** (depending on assumptions and/or requirements)
 - You cannot distinguish a slow computer from a failed one
- Sometimes full transparency is not desirable from an application perspective
 - Users may be located in different continents; distribution is apparent and not something you want to hide



Goal III: Openness

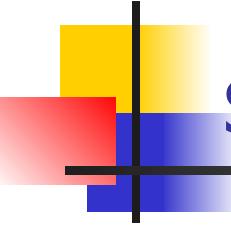
Openness ability to interact with or integrate services/components from other open systems, irrespective of the underlying environment

Achieving openness: Standard rules (protocols/interfaces) to describe services/components

- Interface definitions should be:
 - Complete and neutral

At least make the distributed system independent from **heterogeneity** of the underlying environment:

- Hardware
- Platforms
- Languages



Separating policy and mechanism

Managing complexity: split the systems in smaller components.

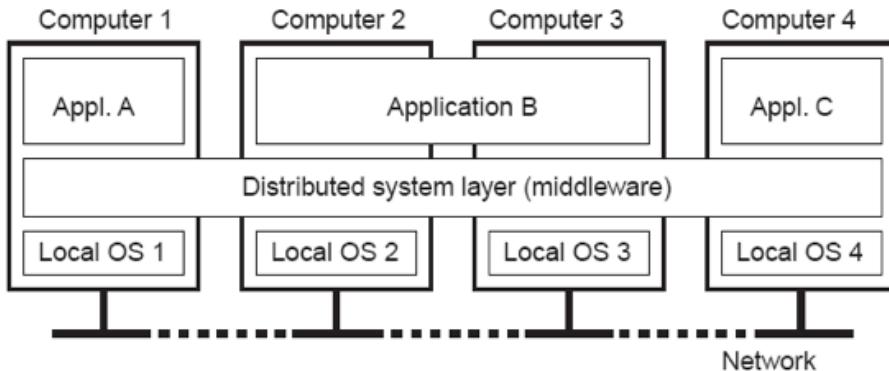
- Components controlled by **policies** specified by applications/users
- Example – web browser caching;
 - Mechanism: caching infrastructure
 - Policy: what to cache, how large the cache is, cache replacement algorithms

Other examples:

- What operations do we allow downloaded code to perform?
- What level of secrecy do we require for communication?

Achieving openness: Ideally, the (distributed) system provides only the **mechanisms** (and a way to specify policies)

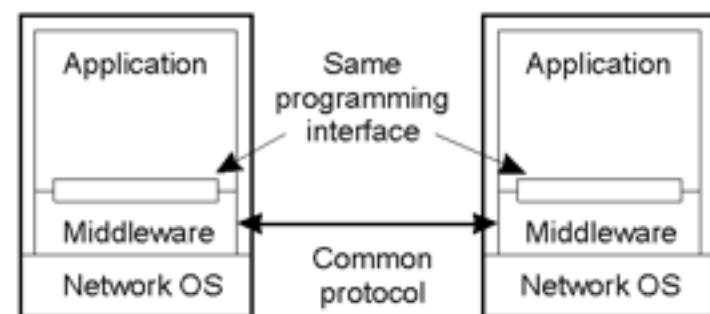
Middleware and Interoperability

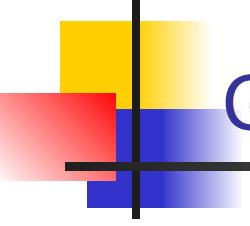


- Independent hardware installations
- Uniform software layer (middleware)

Interoperability provided by:

- **Protocols** used by each middleware layer
- **Interfaces** offered to applications

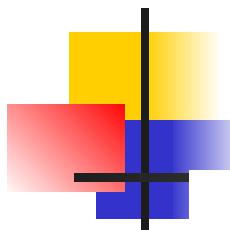




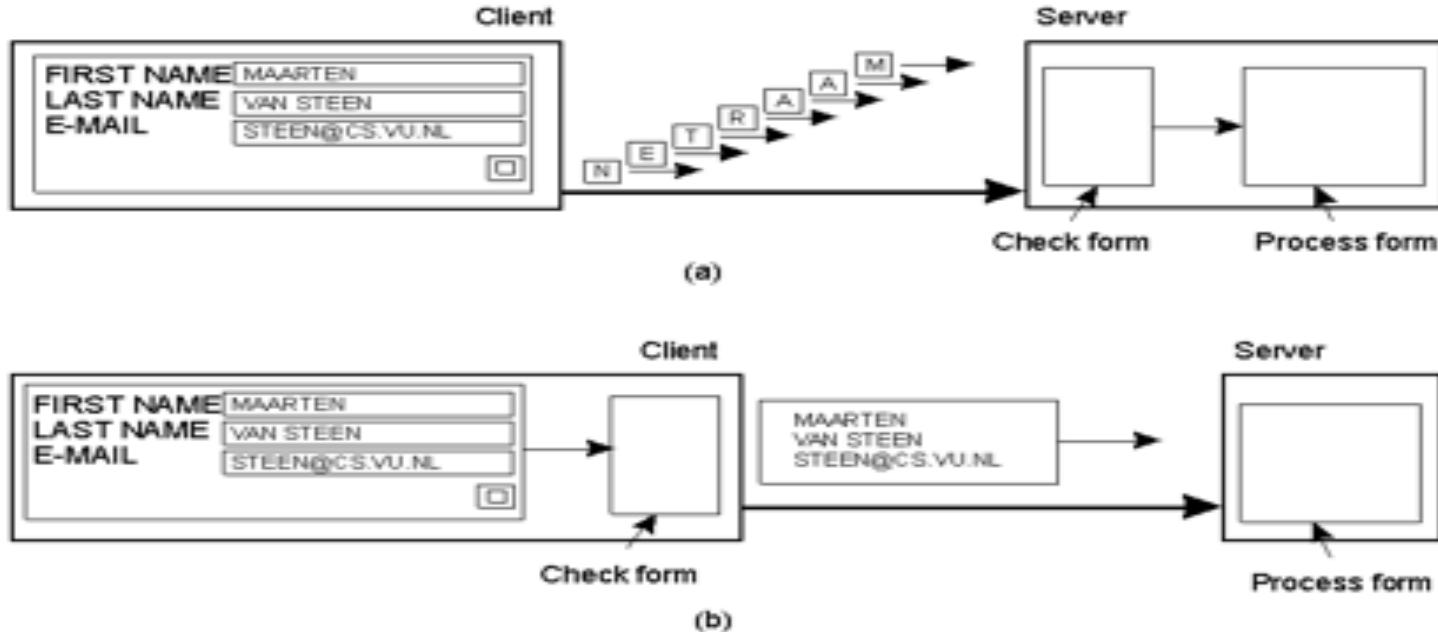
Goal IV: Scalability

Observation: Many developers easily use the adjective “scalable” without making clear **why** their system actually scales.

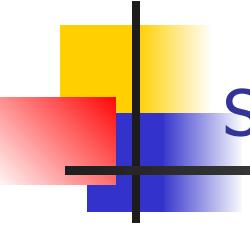
- System should be able to grow over multiple axes:
 - size (#user, #resources), geographical distribution, maintainability



Scaling Techniques (1): Offload work to clients



Technique: Offload work to clients



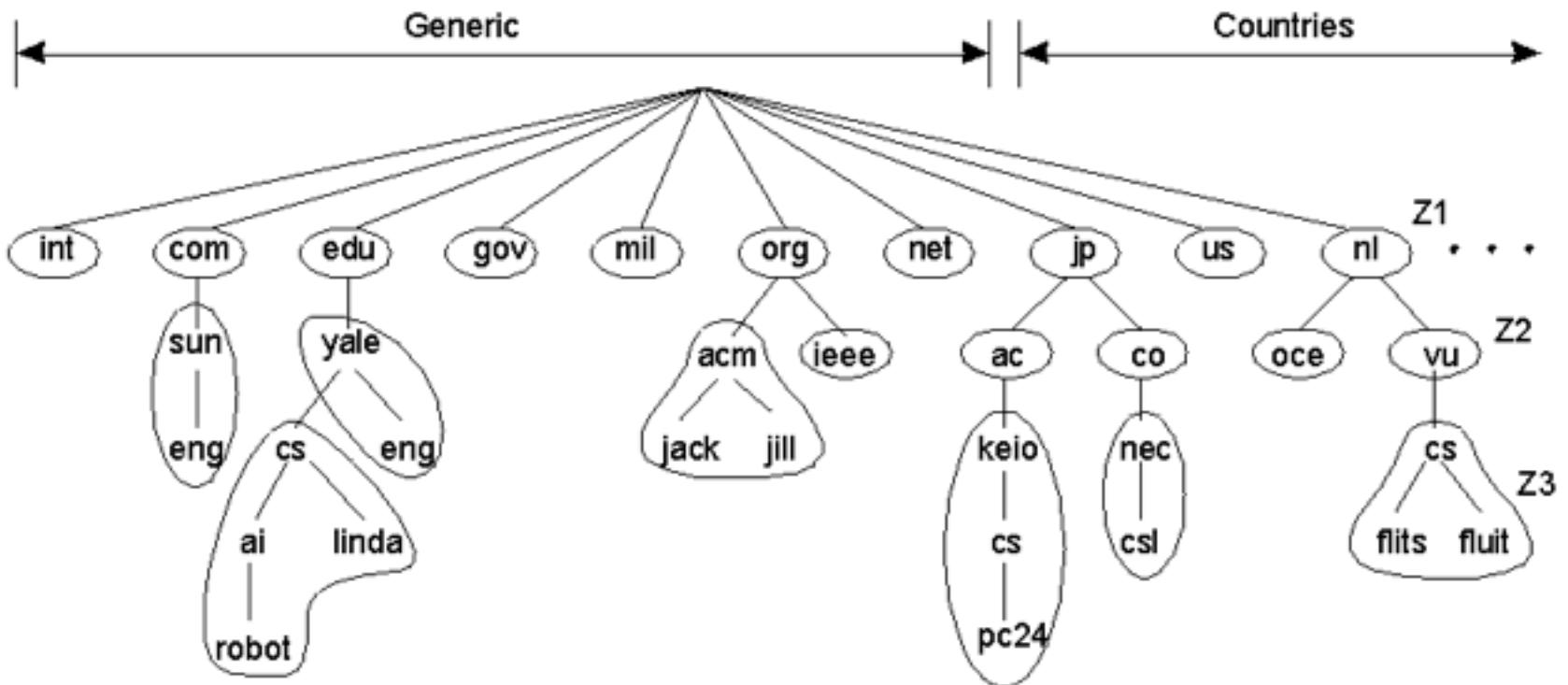
Scaling Techniques (2): Hide communication latency

Technique: Hide communication latency

- Make use of **asynchronous** communication
- Have separate handler for incoming response

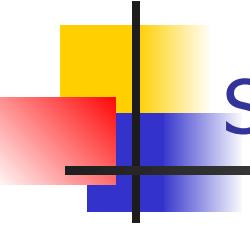
Problem: not every application fits this model

Scaling Techniques (3): Divide the problem space.



Technique: Divide the problem space.

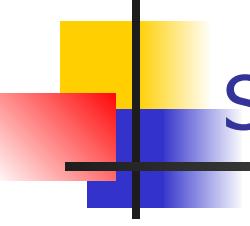
- example: the way DNS divides the name space into zones.



Scaling Techniques (4): Replication/caching

Replication/caching: Make copies of data available at different machines:

- Replicated file servers and databases
- Mirrored Web sites
- Web caches (in browsers and proxies)
- File caching (at server and client)



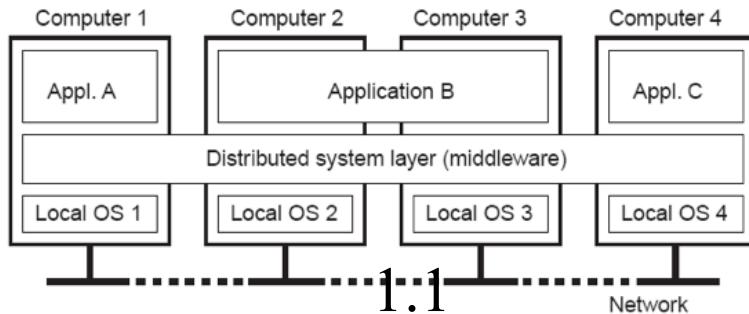
Scaling: The problem

Applying scaling techniques is easy, except for one thing:

- Having multiple copies (cached or replicated), leads to **inconsistencies**: modifying one copy makes that copy different from the rest.
- Always keeping copies consistent (and in a generic way) requires **global synchronization** on each modification.
 - This precludes large-scale solutions.

Last time: What is a Distributed System?

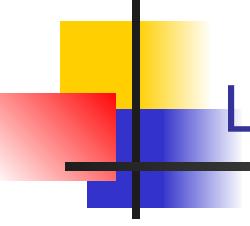
Distributed System: A collection of independent computers that appears to its users as a single coherent system



- Independent hardware installations
- Uniform software layer (**middleware**)

Key goals

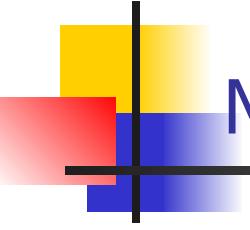
- Connect users and resources
- Distribution transparency
- Openness
- Scalability



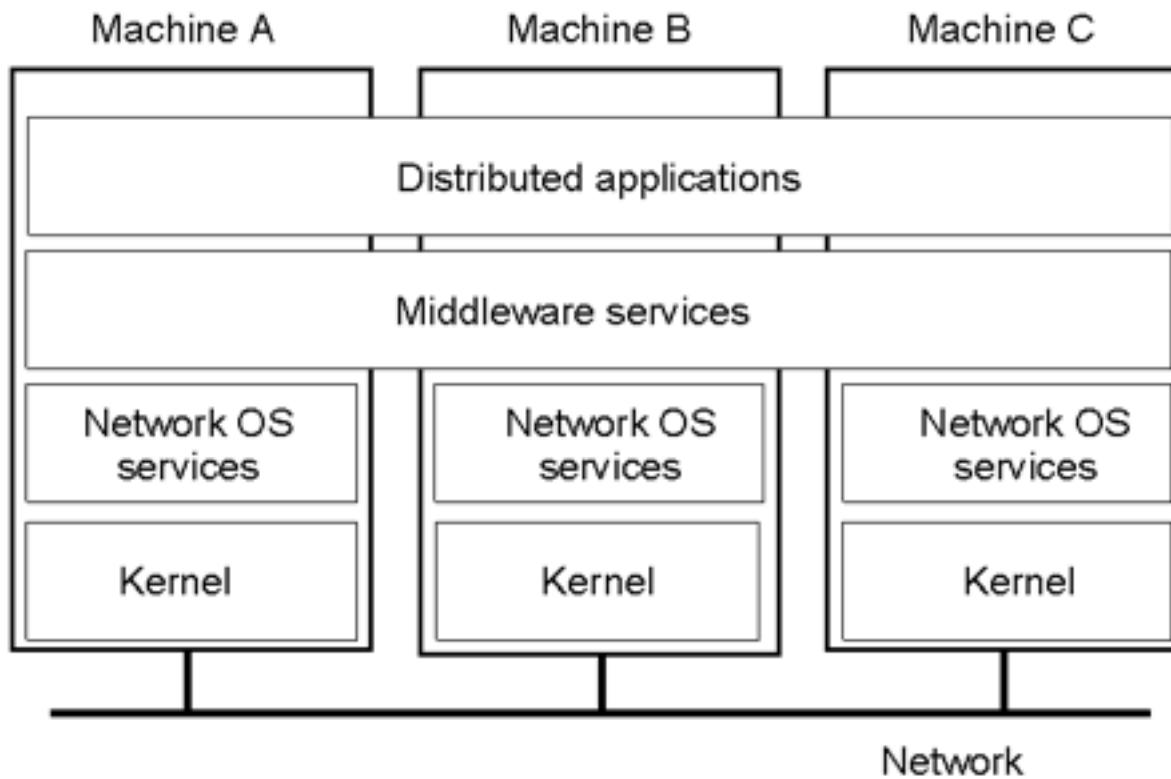
Last time: Pitfalls when developing distributed systems

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- The network is secure
- The topology does not change
- There is one administrator
- The network is homogeneous

[aka Peter Deutsch's "8 fallacies"] [[link](#)]



Middleware



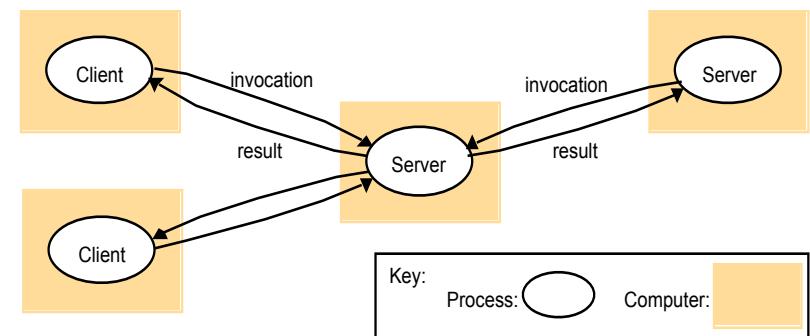
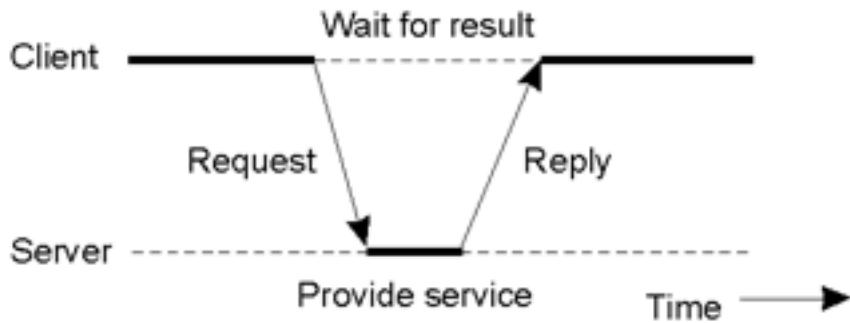
Architecture styles: Client/server

Model:

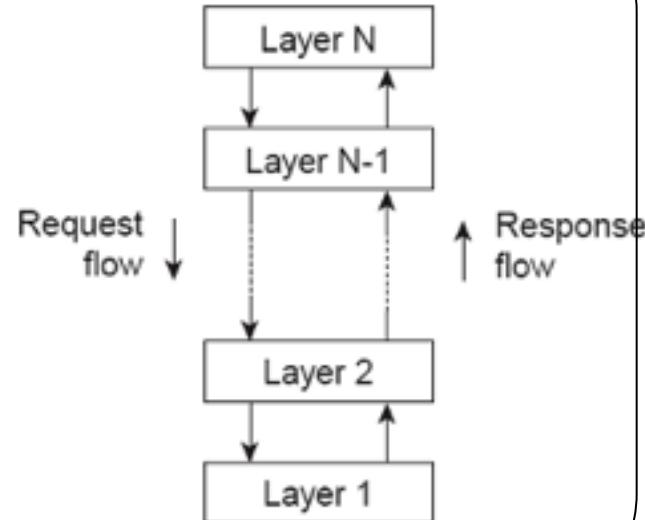
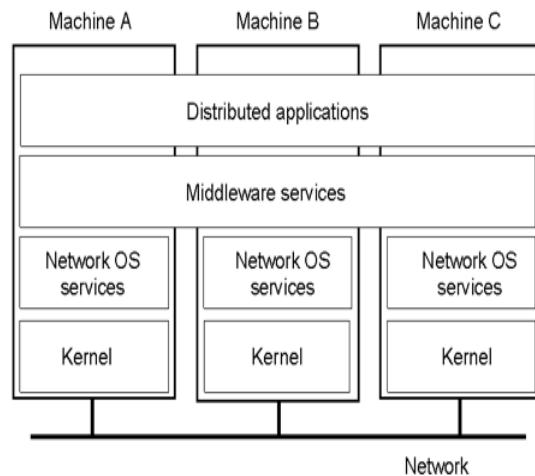
- Server: process implementing a certain service
- Client: uses the service by sending a request and waiting for the reply

Main problem to deal with: unreliable communication

Note: often both roles simultaneously for different services

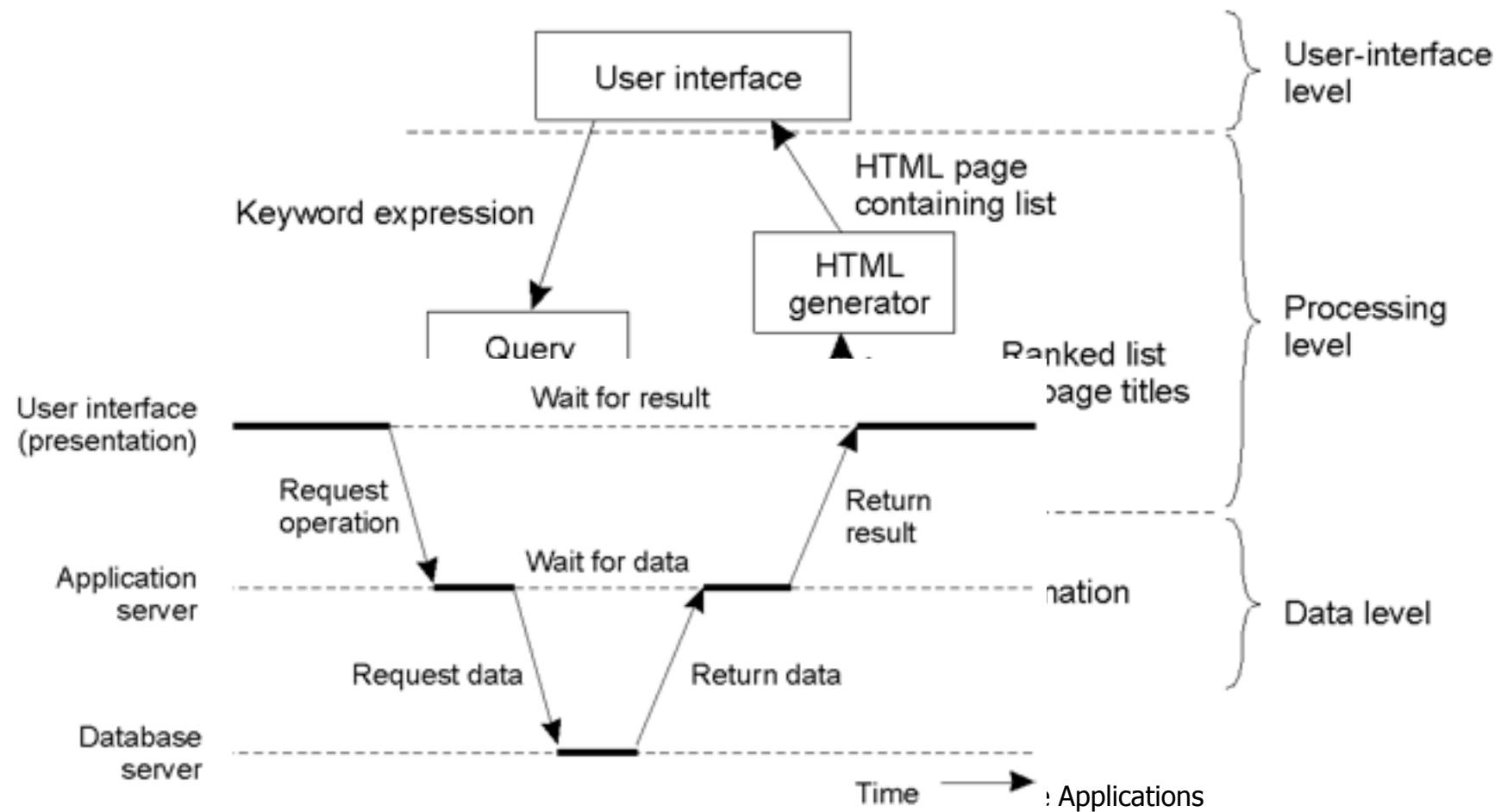


Architectural styles: Layered style



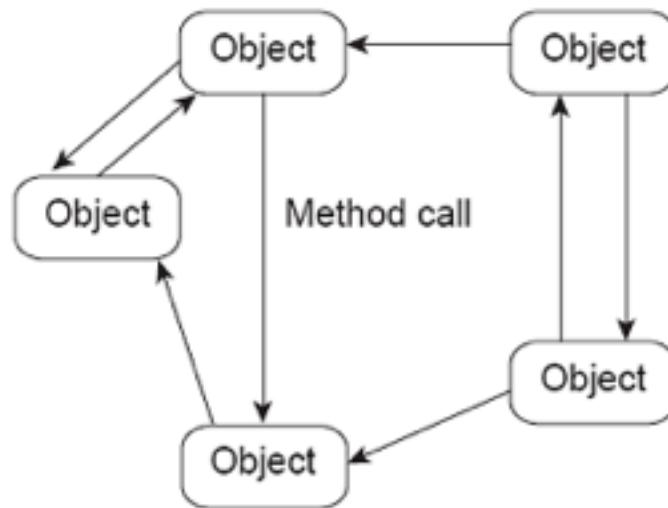
Architectural styles: Layered style example

- **Layered style:** three-layer (tier) architecture commonly used in many internet based applications today



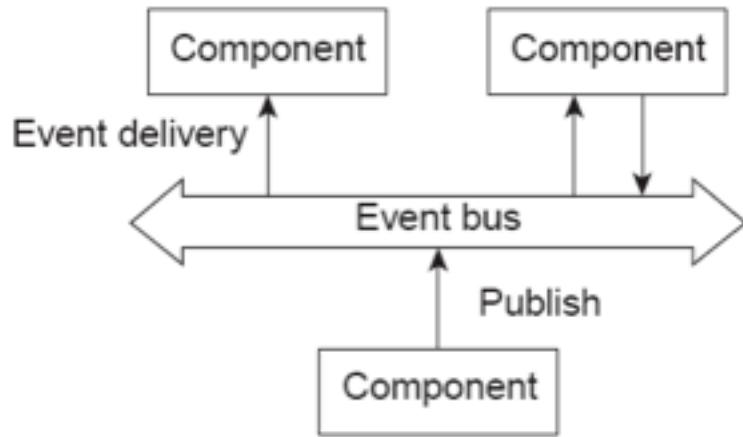
Architectural styles: Object based

Idea: Organize into logically different components, and subsequently distribute those components over the various machines.

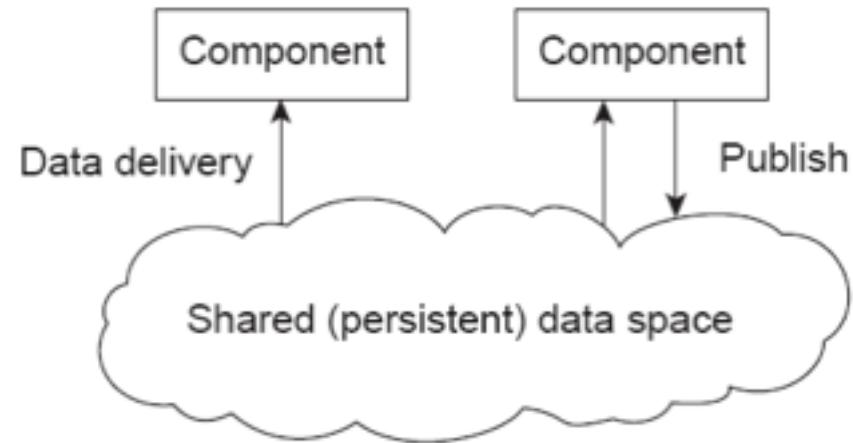


More architectural styles

Alternatives: Decouple processes in space (“anonymous”) and/or time (“asynchronous”)



Event-based

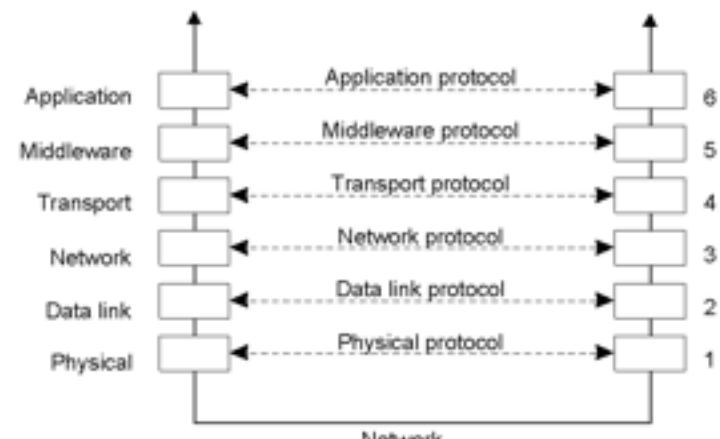


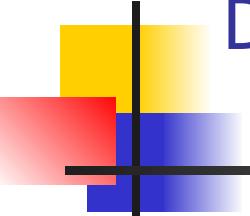
Data-centered architectures

Summary so far:

- Definition of distributed systems
 - collection of independent components that appears to its users as a single coherent system
- Goals, pitfalls, scalability techniques,
- Architectural styles

One time-tested design guideline: end-to-end argument ...

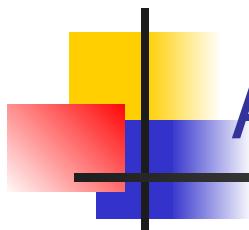




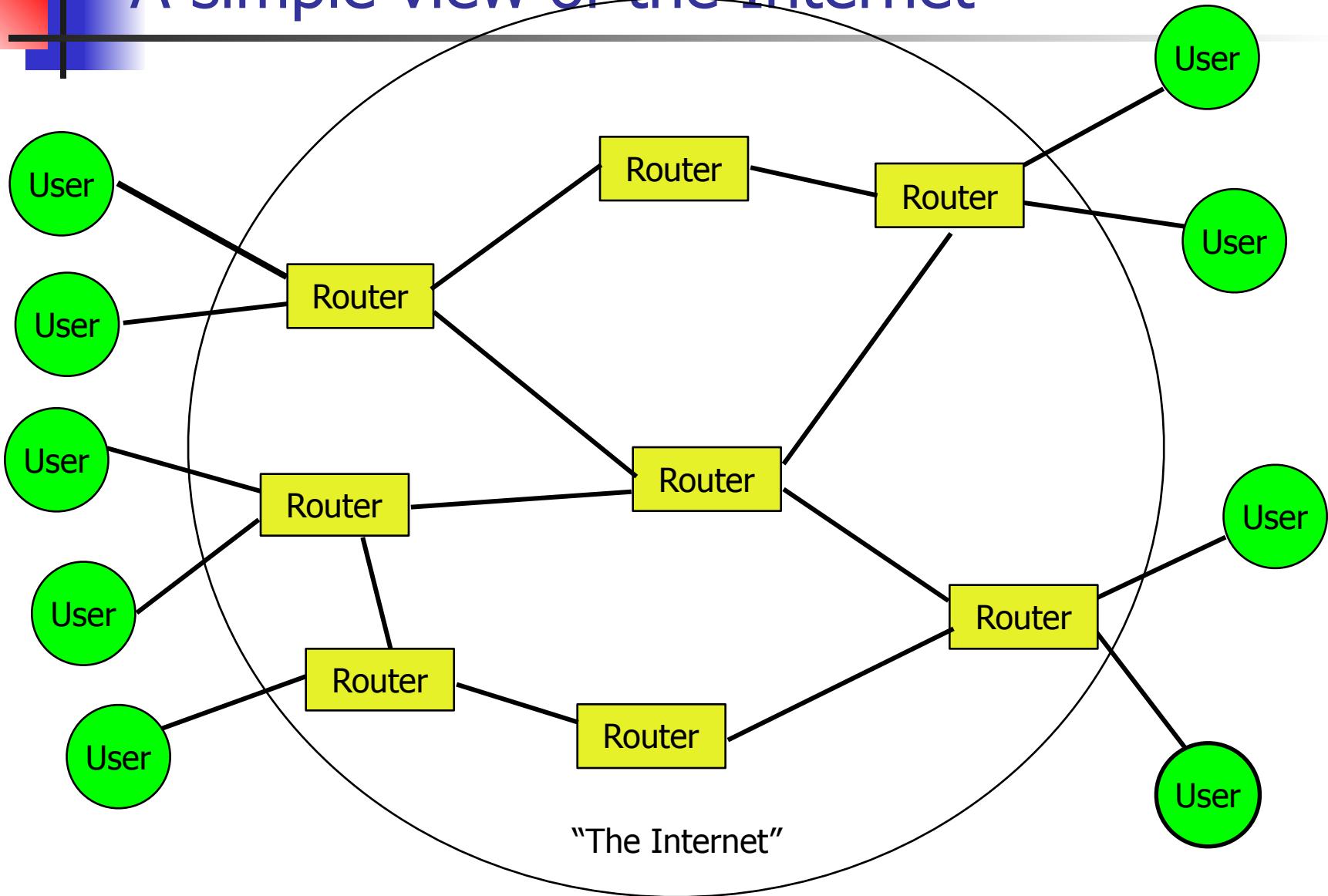
Design principle for the Internet

The end-to-end argument

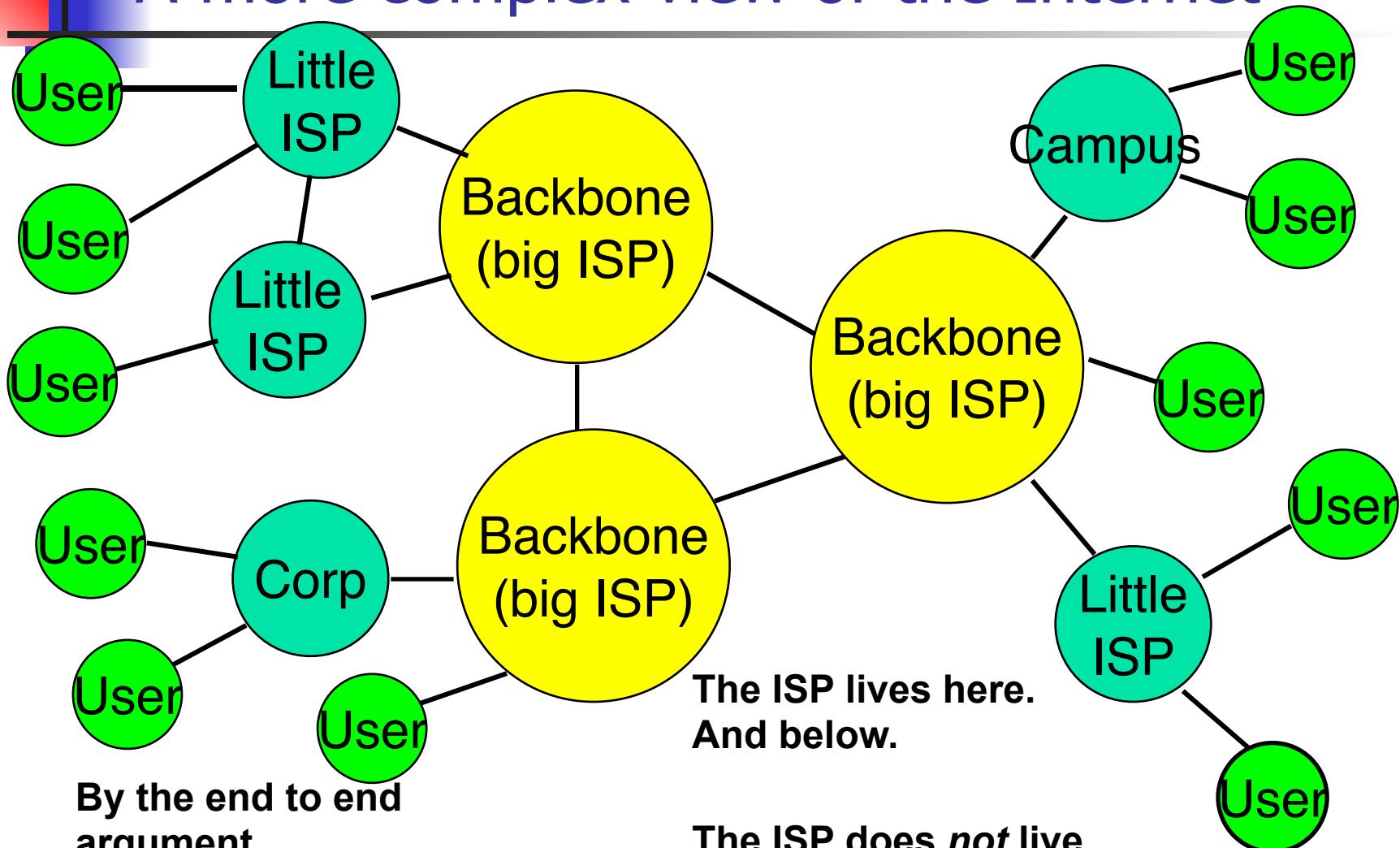
- The lower layers of the network are not the right place to implement application-specific functions.
 - Lower layers of the network should implement basic and general functions,
 - the applications should be built “above” these functions, at the edges.
 - **Caveat:** unless performance critical
- Consequence: move functions “up and out”.
 - the result is function migration to the end-node.
- The network should be “as transparent as technology permits”



A simple view of the Internet



A more complex view of the Internet



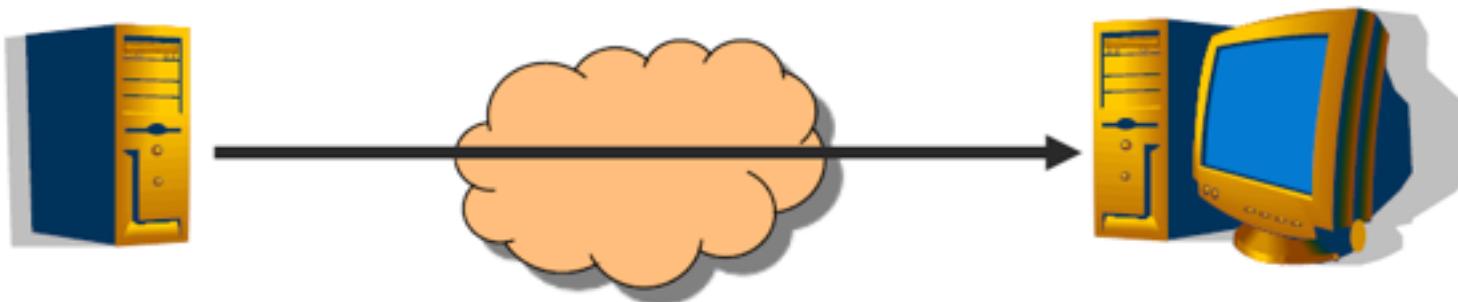
By the end to end argument,
applications live at the edge.

The ISP lives here.
And below.

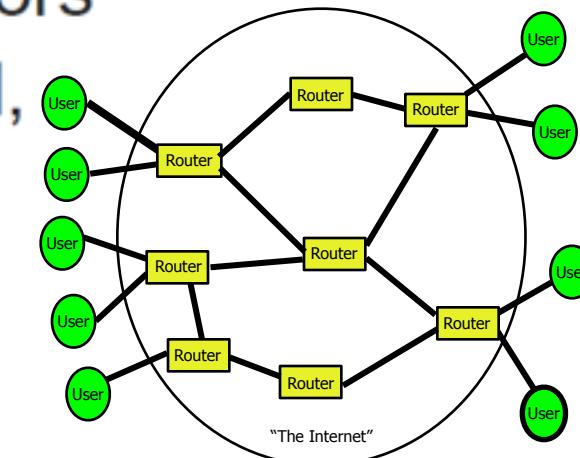
The ISP does *not* live
at the end-points.

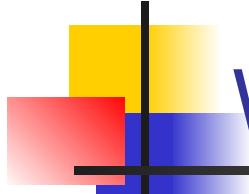
(They can try...)

Example: RELIABLE file transfer for the

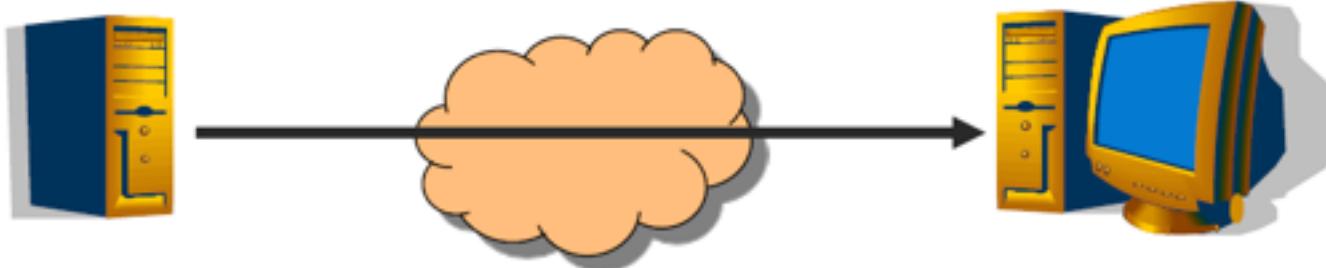


- What can go wrong?
 - Disk can introduce bit errors
 - Host I/O can introduce bit errors
 - Packets can get lost, delayed, misordered, corrupted

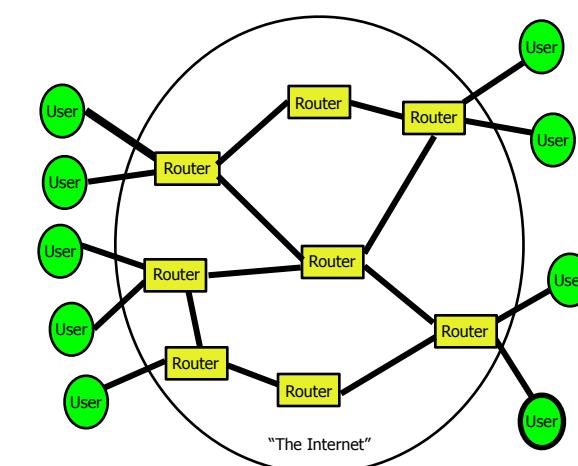




Which one?



- Solution I: hop-by-hop reliable transfer
- Solution II: end-to-end check and retry

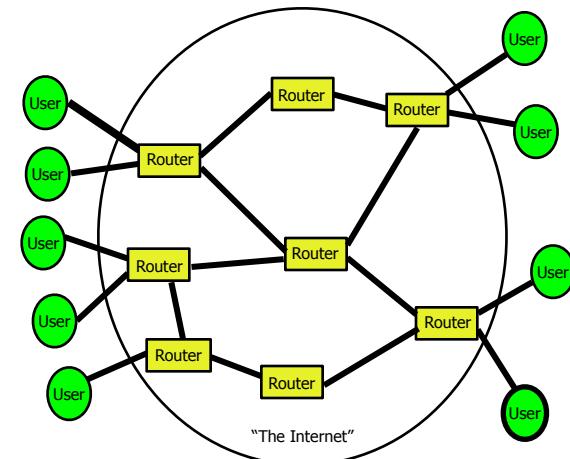


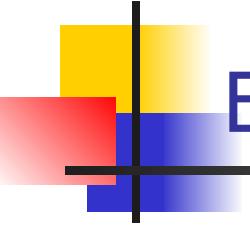
What belongs IN (the network), what OUT?

Questions:

- Does routing belong in the “dumb, minimal” net?
- How about:
 - Secure (encrypted) transmission
 - Duplication suppression
 - What if the application creates duplicates?
 - In-order delivery
 - Do we need hop-by-hop ordering?
 - Multicast (i.e., one-to-many communication)?
- Compare to telephone network!

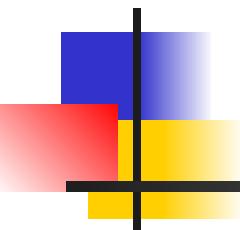
Is the E2E principle constraining innovation?





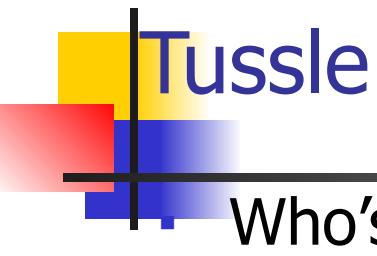
End to end argument operates at two levels

- At the “network” level:
 - Avoid putting constraining per-application functions into the core of the network.
 - Build general purpose network services.
- At the “application” level:
 - Build applications in a way that makes them robust, easy to use, reliable, etc.
 - Simple approach: push software to the edge.
 - Perhaps more realistic: reason carefully about role of servers, trusted third parties, etc.



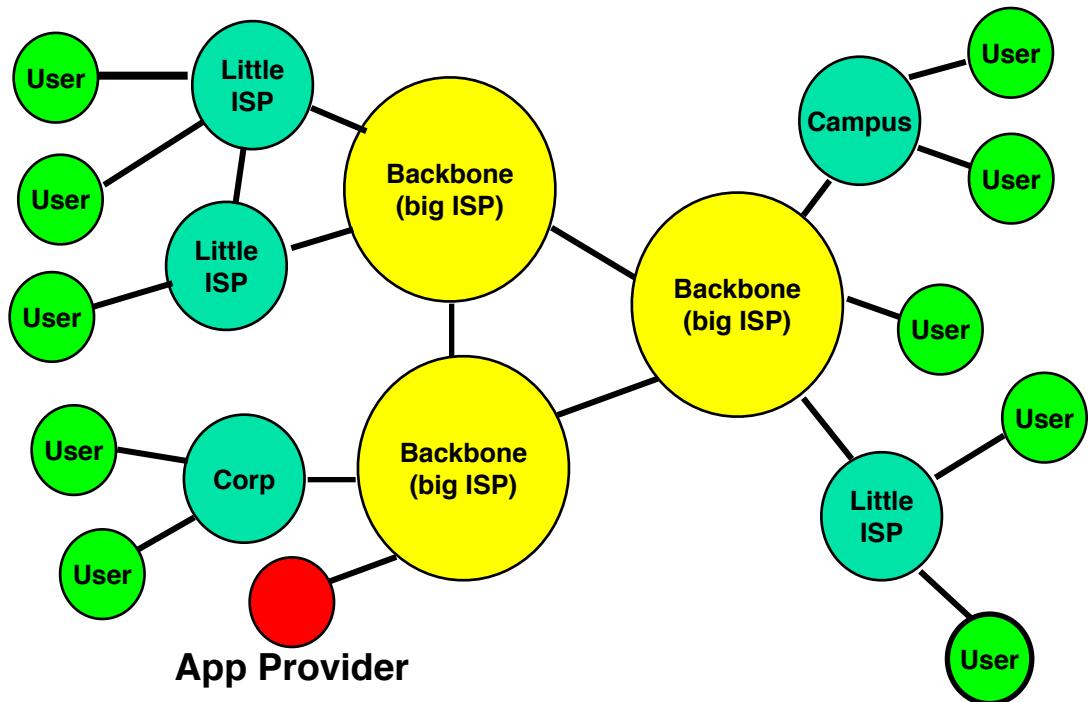
Net Neutrality

.



Who's battling?

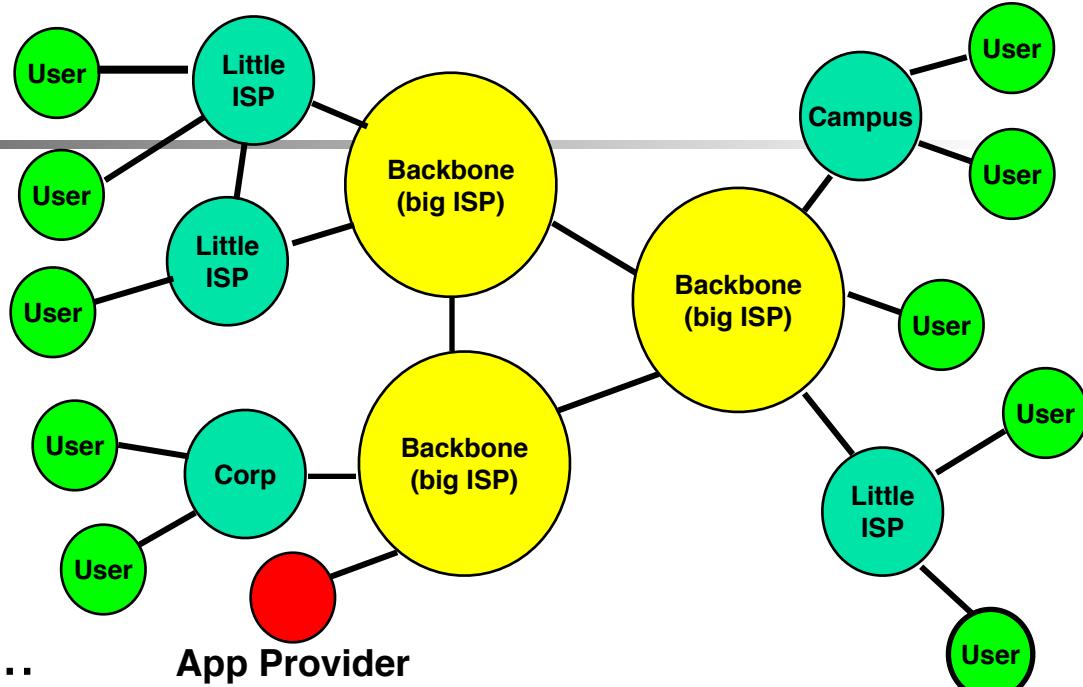
- What's at issue?

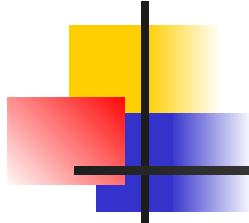


Discrimination?

Harmful?

- Examples?
- Beneficial?
- Examples?
- Unpredictable?
- Examples?
- How to differentiate...
 - Tests?
- Who should differentiate?
 - Market
 - Regulators



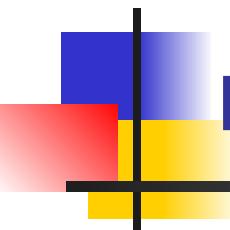


Summary so far:

- Definition of distributed systems
 - collection of independent components that appears to its users as a single coherent system
- Goals, pitfalls, scalability techniques, architectural styles

Key requirement: Components need to communicate

- ⇒ Shared memory
- ⇒ Message exchange
- ⇒ need to agree on many things
 - ⇒ **Protocols:** data formats, exception handling, naming, ...



Lecture 3: OSI Stack, Sockets, Protocols

Academic honesty is essential to the continued functioning of the University [...]. All UBC students are expected to behave as honest and responsible members of an academic community. Breach of those expectations [...] may result in disciplinary action.

It is the student's obligation to inform himself or herself of the applicable standards for academic honesty.

Students must be aware that standards at the University of British Columbia may be different from those in secondary schools or at other institutions. If a student is in any doubt as to the standard of academic honesty in a particular course or assignment, then the student must consult with the instructor as soon as possible, and in no case should a student submit an assignment if the student is not clear on the relevant standard of academic honesty.

If an allegation is made against a student ...

As members of this enterprise, all students are expected to know, understand, and follow the codes of conduct regarding academic integrity. **At the most basic level, this means submitting only original work done by you and acknowledging all sources of information or ideas and attributing them to others as required. This also means you should not cheat, copy, or mislead others about what is your work.**

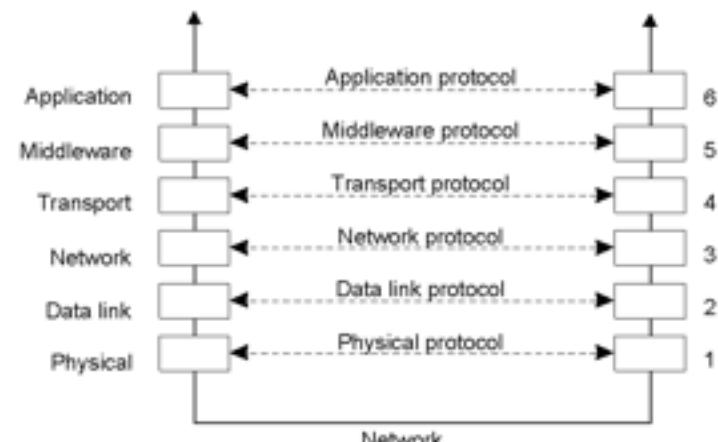
Violations of academic integrity (i.e., misconduct) lead to the breakdown of the academic enterprise, and therefore serious consequences arise and harsh sanctions are imposed. [...]

Summary so far:

- Definition of distributed systems
 - collection of independent components that appears to its users as a single coherent system
- Goals, pitfalls, scalability techniques, architecture styles

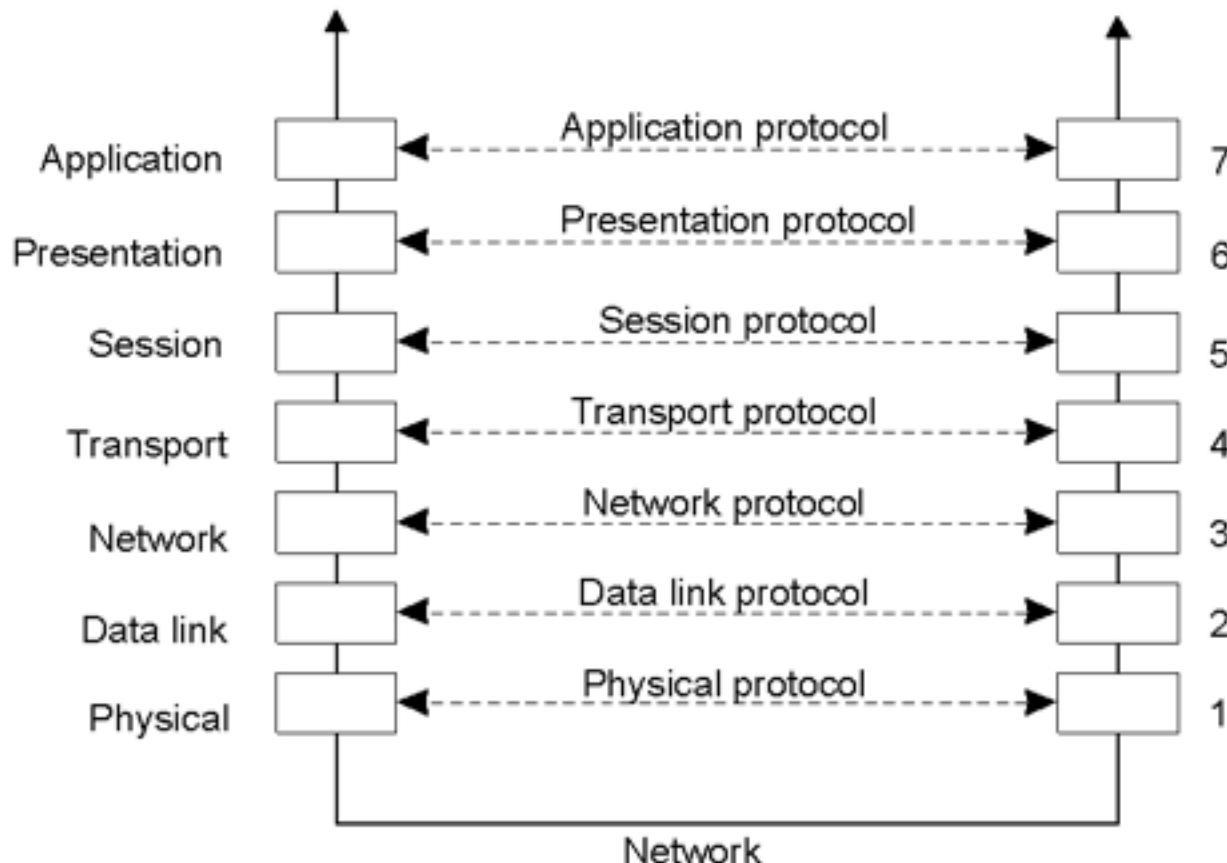
Requirement: Components need to communicate

- ⇒ Shared memory
- ⇒ Message exchange
- ⇒ need to agree on many things
 - ⇒ **Protocols**: data formats, exception handling, naming



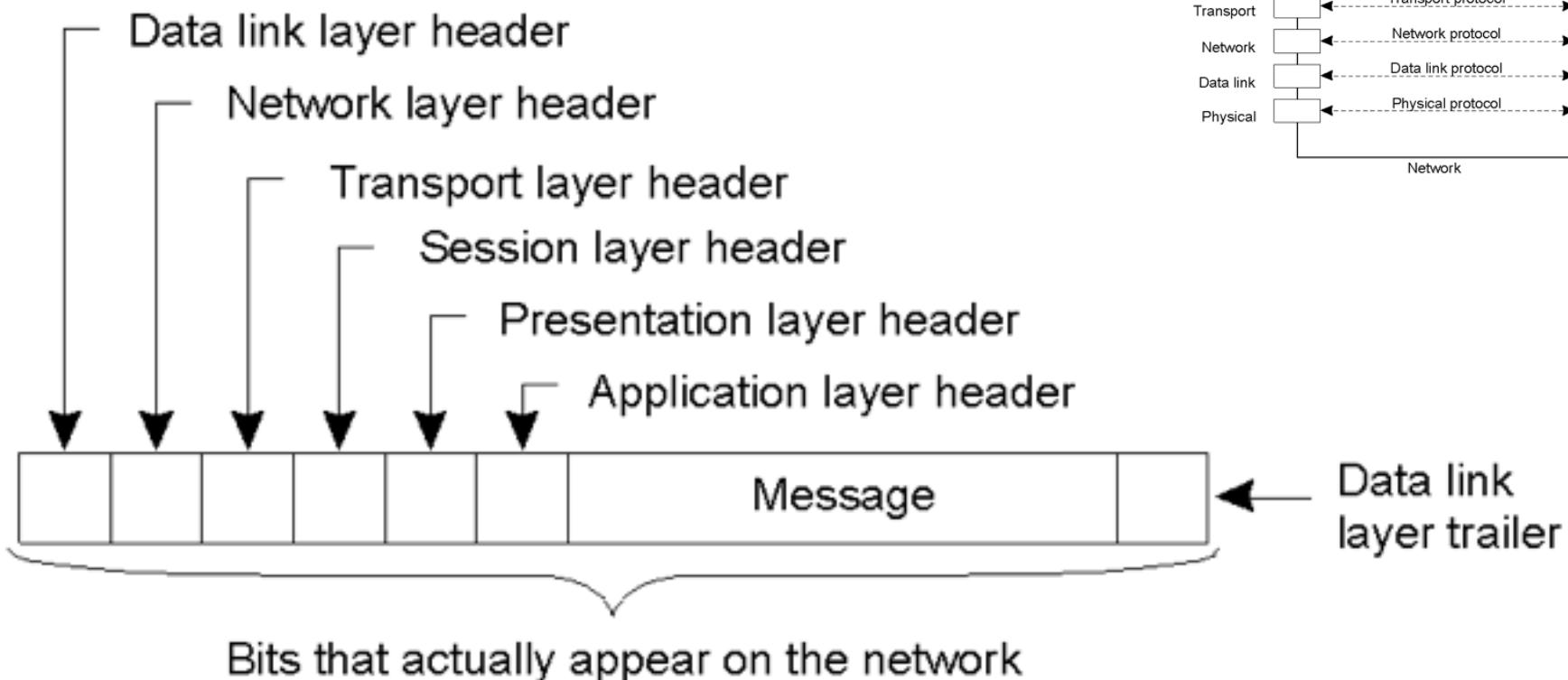
Layered Protocols

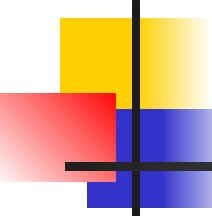
- Layers, interfaces, and protocols in the OSI model.



Layered Protocols (2)

- A typical message as it appears on the network.





Low-level layers

Physical layer: contains the specification and implementation of bits, and their transmission between sender and receiver

Data link layer: prescribes the transmission of a series of bits into a frame to allow for error and flow control

Network layer: describes how packets in a network of computers are to be routed.

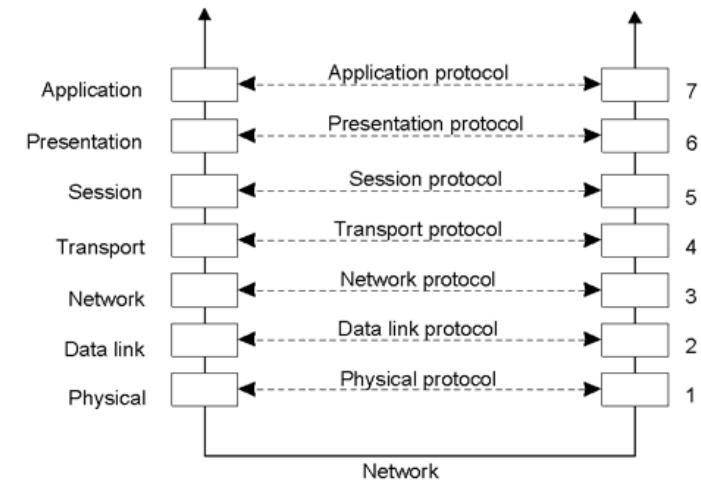
Note: for most distributed systems, the lowest level interface is that of the **network layer**.

Transport Layer

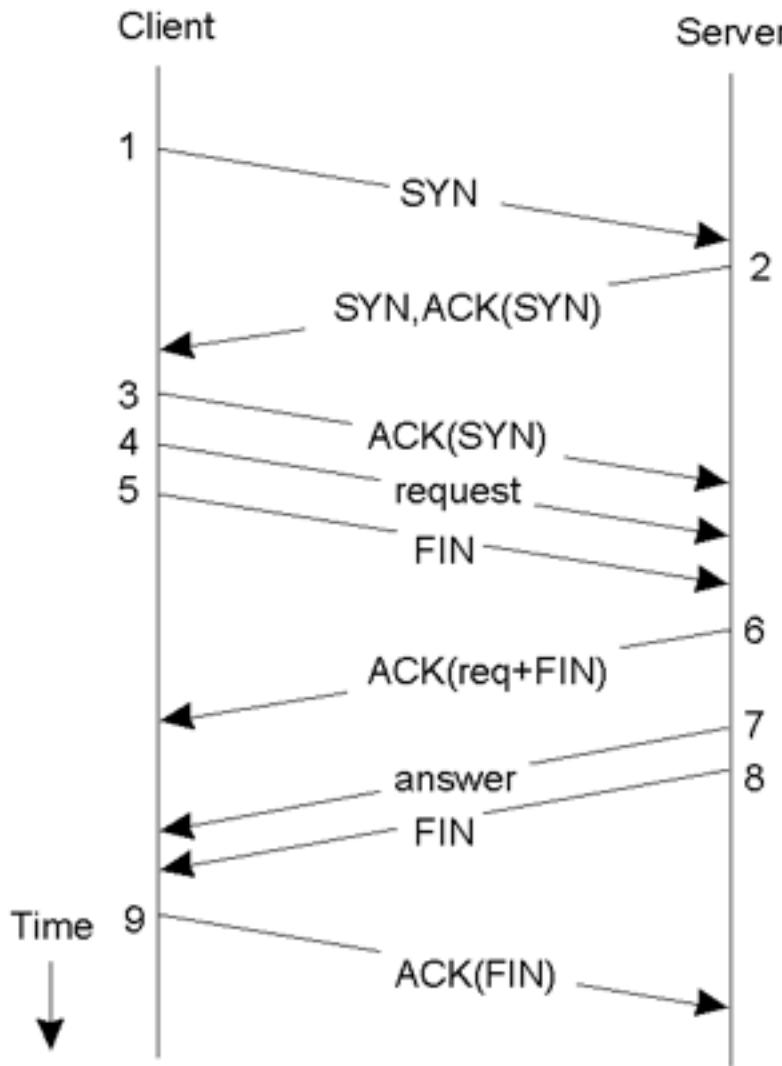
The transport layer provides the actual communication facilities for most distributed systems.

Standard Internet protocols

- TCP: connection-oriented, reliable, stream-oriented communication
- UDP: unreliable (best-effort) datagram communication



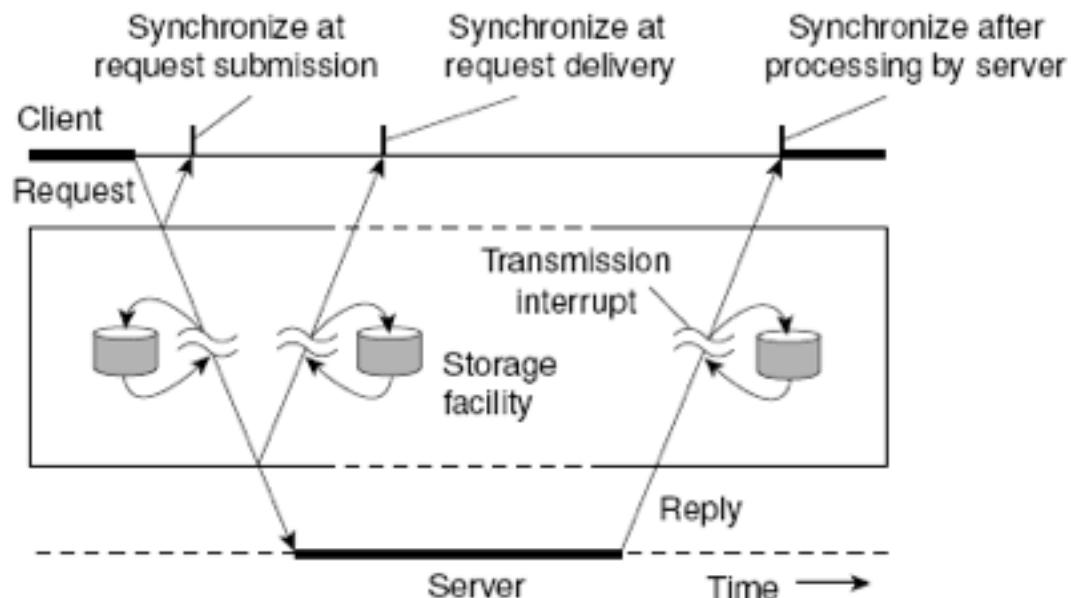
Customization can bring efficiency gains: TCP modified for client server communication



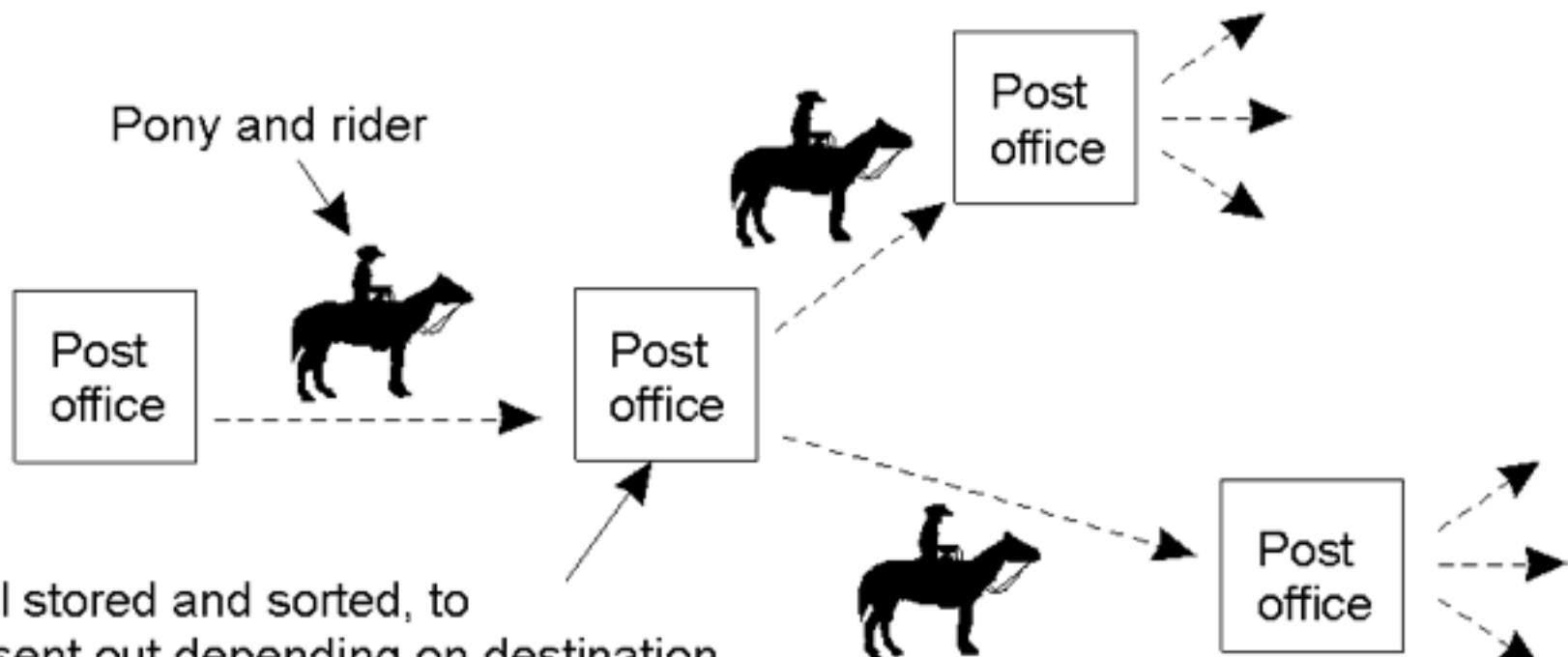
(a)

Types of communication protocols: Persistence and Synchronicity

- **Synchronicity:** Does the sender wait for a reply or acknowledgement?
 - Yes → synchronous communication
 - No → asynchronous communication
- **Persistence:** Do the sender and the receiver have to be both online for the message exchange to happen?
 - No → persistent communication
 - Yes → non-persistent communication

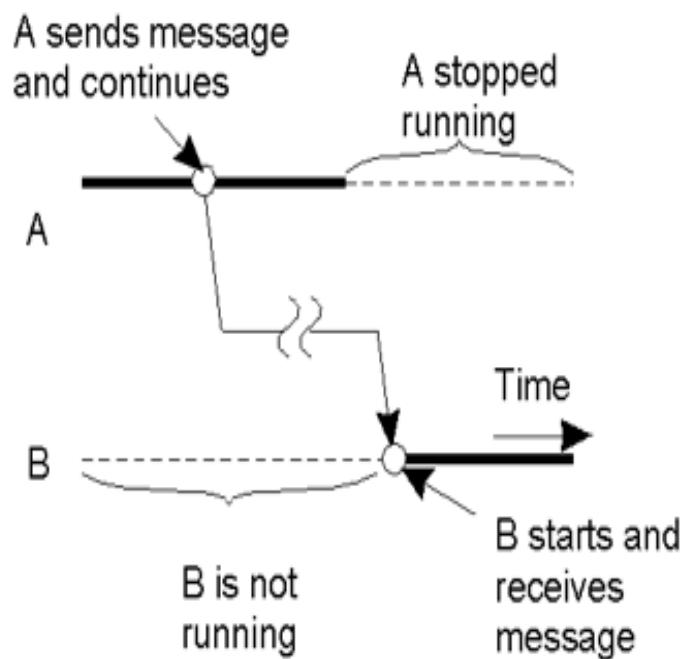


Persistence and Synchronicity ... (II)

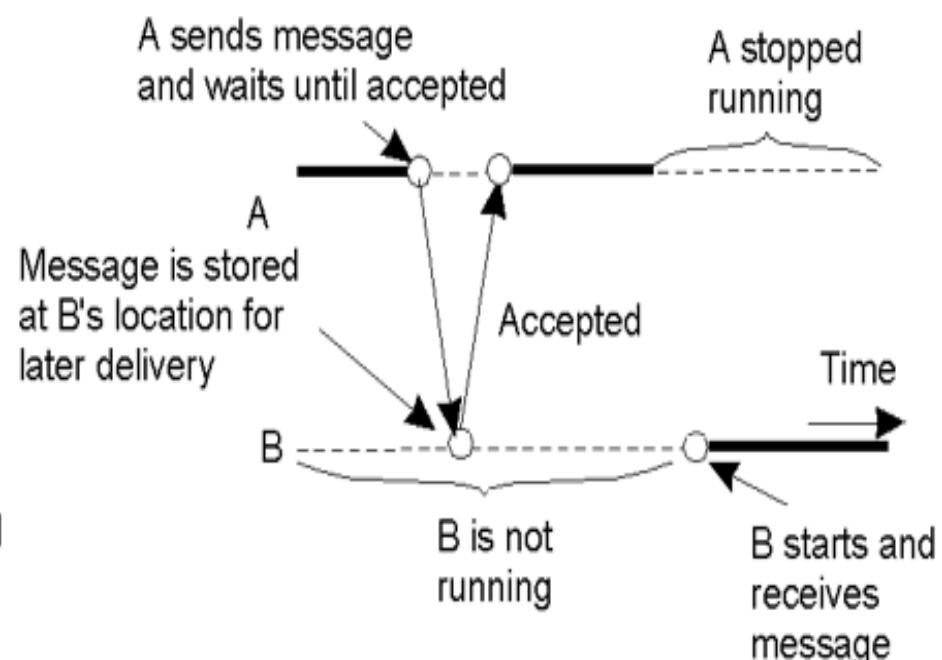


- **Asynchronous, Persistent**

Persistence and Synchronicity ... (III)

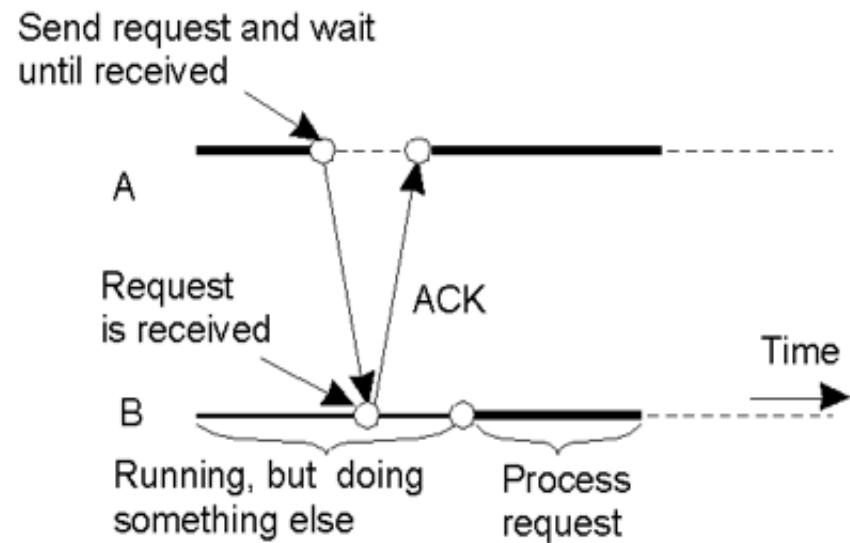
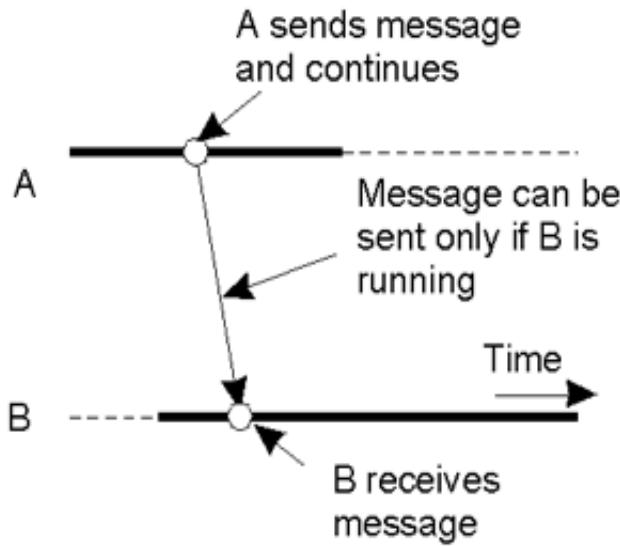


Persistent asynchronous communication



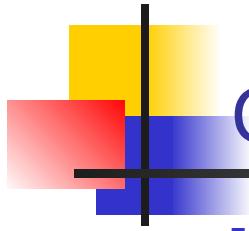
Persistent synchronous communication

Persistence and Synchronicity ... (IV)



Transient asynchronous communication

Receipt-based transient synchronous communication



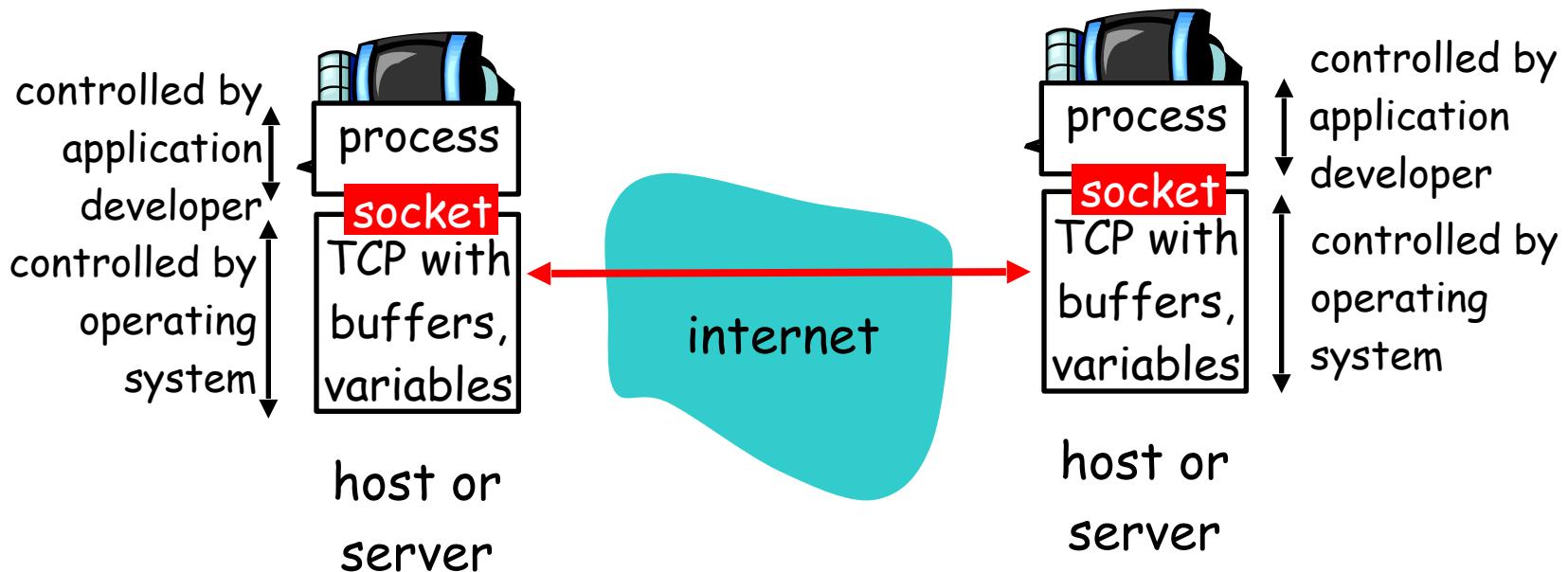
Getting more hands on

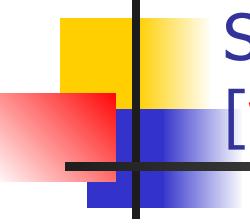
- Crash course on socket programming

Socket-programming using TCP

Socket: a 'door' between an application process and an end-end-transport protocol (UCP or TCP)

TCP: reliable transfer of **bytes** from one process to another





Socket programming [with TCP]

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Client must contact server

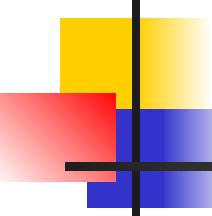
- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

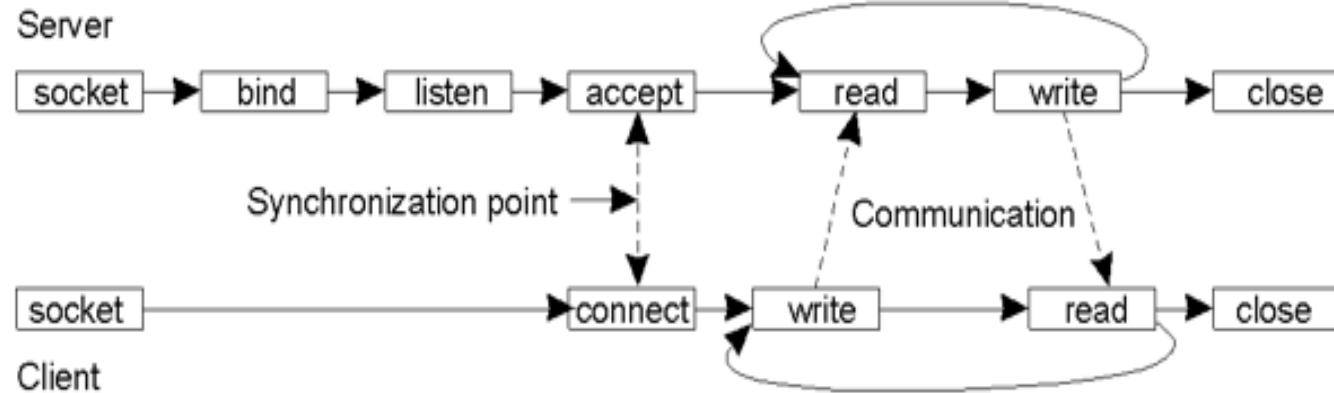
- creating client-local TCP socket
- specifying IP address & port of server process
- client establishes connection to server TCP

Server-side

- When contacted by client, **server creates new socket** for server process to communicate with client
 - allows server to talk with multiple clients
 - source port numbers used to distinguish between clients



State machine



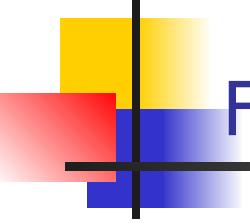
Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Example: echo – client

```
public class TCPEchoClient {  
    public static void main(String[] args) throws IOException {  
        String server = args[0];          // Server name or IP address  
        byte[] byteBuffer = args[1].getBytes();  
        int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;  
  
        // Create socket that is connected to server on specified port  
        Socket socket = new Socket(server, servPort);  
        System.out.println("Connected to server...sending echo string");  
  
        InputStream in = socket.getInputStream();  
        OutputStream out = socket.getOutputStream();  
  
        out.write(byteBuffer); // Send the encoded string to the server  
  
        // Receive the same string back from the server  
        int totalBytesRcvd = 0; // Total bytes received so far  
        int bytesRcvd;         // Bytes received in last read  
        while (totalBytesRcvd < byteBuffer.length) {  
            if ((bytesRcvd = in.read(byteBuffer, totalBytesRcvd,  
                byteBuffer.length - totalBytesRcvd)) == -1)  
                throw new SocketException("Connection close prematurely");  
            totalBytesRcvd += bytesRcvd;  
        }  
        socket.close(); // Close the socket and its streams  
    }  
}
```

Example: echo – server

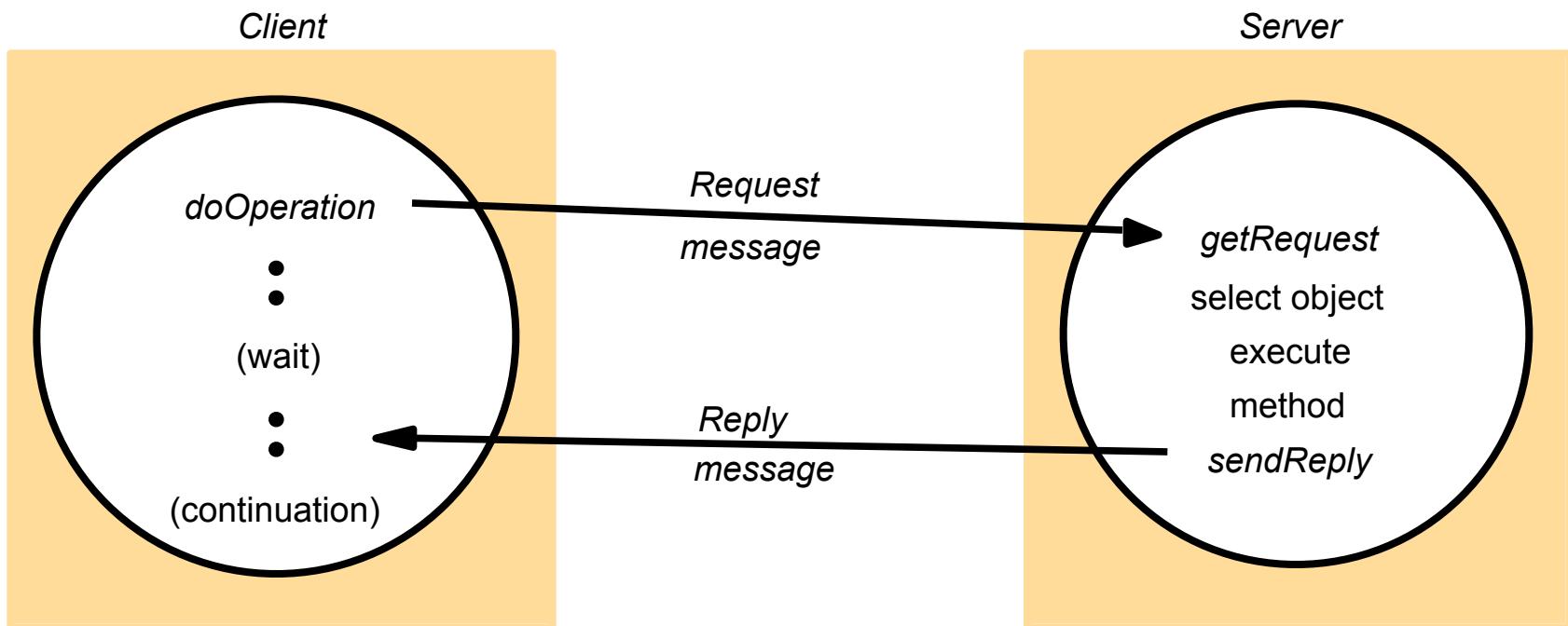
```
public class TCPEchoServer {  
    private static final int BUFSIZE = 32;      // Size of receive buffer  
    public static void main(String[] args) throws IOException {  
        int servPort = Integer.parseInt(args[0]);  
  
        // Create a server socket to accept client connection requests  
        ServerSocket servSock = new ServerSocket(servPort);  
  
        int recvMsgSize;    // Size of received message  
        byte[] byteBuffer = new byte[BUFSIZE];    // Receive buffer  
  
        for (;;) { // Run forever, accepting and servicing connections  
            Socket clntSock = servSock.accept();      // Get client connection  
  
            System.out.println("Handling client at " +  
                clntSock.getInetAddress().getHostAddress());  
  
            InputStream in = clntSock.getInputStream();  
            OutputStream out = clntSock.getOutputStream();  
  
            // Receive until client closes connection, indicated by -1 return  
            while ((recvMsgSize = in.read(byteBuffer)) != -1)  
                out.write(byteBuffer, 0, recvMsgSize);  
  
            clntSock.close();    // Close the socket.  We are done with this client!  
        }  
    }  
}
```

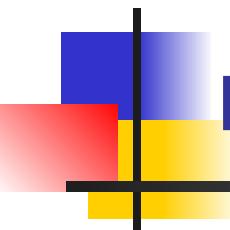


First assignment

- **UDP**
- Client for an “encoding server”.
- Protocol
 - Request-reply: define message format and client/server behavior.
 - Application level:
- Submit individually:
 - Your client code
 - The ‘secret’ encoding you’ve got by successfully running your client

Request-reply communication





Lecture 4-5-6: Remote Procedure Calls

Academic honesty is essential to the continued functioning of the University [...]. All UBC students are expected to behave as honest and responsible members of an academic community. Breach of those expectations [...] may result in disciplinary action.

It is the student's obligation to inform himself or herself of the applicable standards for academic honesty.

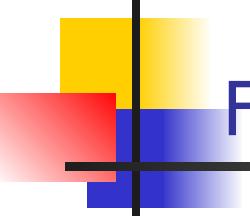
Students must be aware that standards at the University of British Columbia may be different from those in secondary schools or at other institutions. If a student is in any doubt as to the standard of academic honesty in a particular course or assignment, then the student must consult with the instructor as soon as possible, and in no case should a student submit an assignment if the student is not clear on the relevant standard of academic honesty.

If an allegation is made against a student ...

As members of this enterprise, all students are expected to know, understand, and follow the codes of conduct regarding academic integrity. **At the most basic level, this means submitting only original work done by you and acknowledging all sources of information or ideas and attributing them to others as required. This also means you should not cheat, copy, or mislead others about what is your work.**

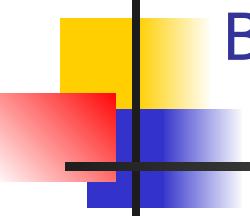
Violations of academic integrity (i.e., misconduct) lead to the breakdown of the academic enterprise, and therefore serious consequences arise and harsh sanctions are imposed. [...]

More info? Follow this [link](#)



First assignment

- Story: client for an “encoding server”.
- Protocol.
 - Request-reply layer: define message format and client/server behavior. UDP based.
 - **Note: you'll reuse this part. Make it robust! We'll try to evaluate this!**
 - Application level: defines payload format
- Submit individually:
 - Your client code; The ‘secret’ encoding you’ve got by successfully running your client
- Test server available
- Firewalls
- Utility: nc –u,
- uniqueID



Building Distributed Applications:

Two Paradigms

RPC offers support here!

Communication oriented design

- Start with communication protocol;
 - Design message format and syntax
- Design components (clients, servers) by specifying how they react to incoming messages

Problems

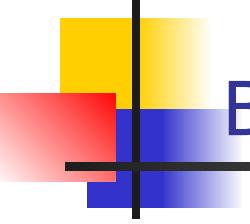
- Protocol design problems
- Specify components as finite state machines
- Focus on communication instead of on application

Application-oriented design

- Start with application
 - Design, build, test conventional (single-box) application
- Partition program

Problems

- Preserving semantics when using partitioning program (using remote resources)
- Masking failures
- Concurrency



Building distributed applications

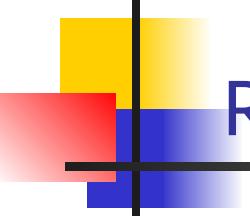
RPC goal: minimize the difference between a single box and a distributed deployment environment

Observations:

- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There are few reasons not to execute procedures on separate machines

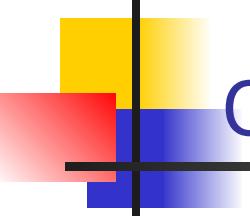
Idea: communication between caller & callee can be hidden by using procedure-call mechanism.

- (local program calls procedures hosted remotely in a similar way to local procedures)



Remote Procedure Calls (RPC)

- **Idea:** local program calls procedure hosted remotely in a similar way to a local procedure
 - Information is passed in procedure arguments, results
- **Key issue:** Provide transparency
 - (access transparency) mask differences in data representation
 - (location, failure transparency) handle failures
 - (location transparency) handle different address spaces (?)
 - (provide migration transparency , replication transparency)
 - Security!
- **Context:** constrained by the programming language

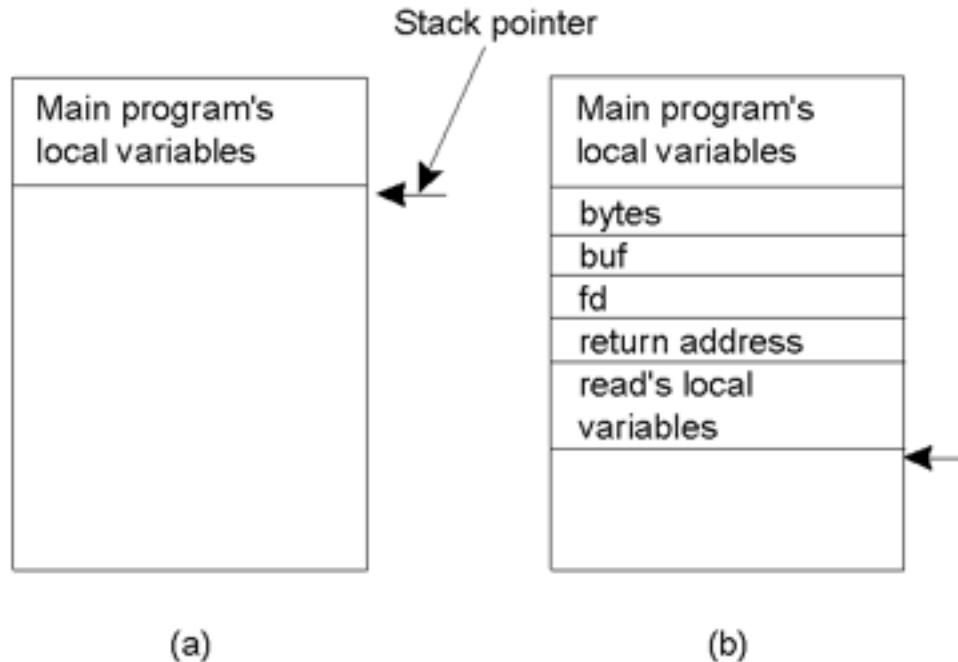


Outline

- Mechanics.
 - How does it actually work ...
 - ... and limitations
- RPC in practice.
- Case study: The Network File System
- Discussion:
 - Reliable RPC
 - Asynchronous RPC

Conventional (local) Procedure Call

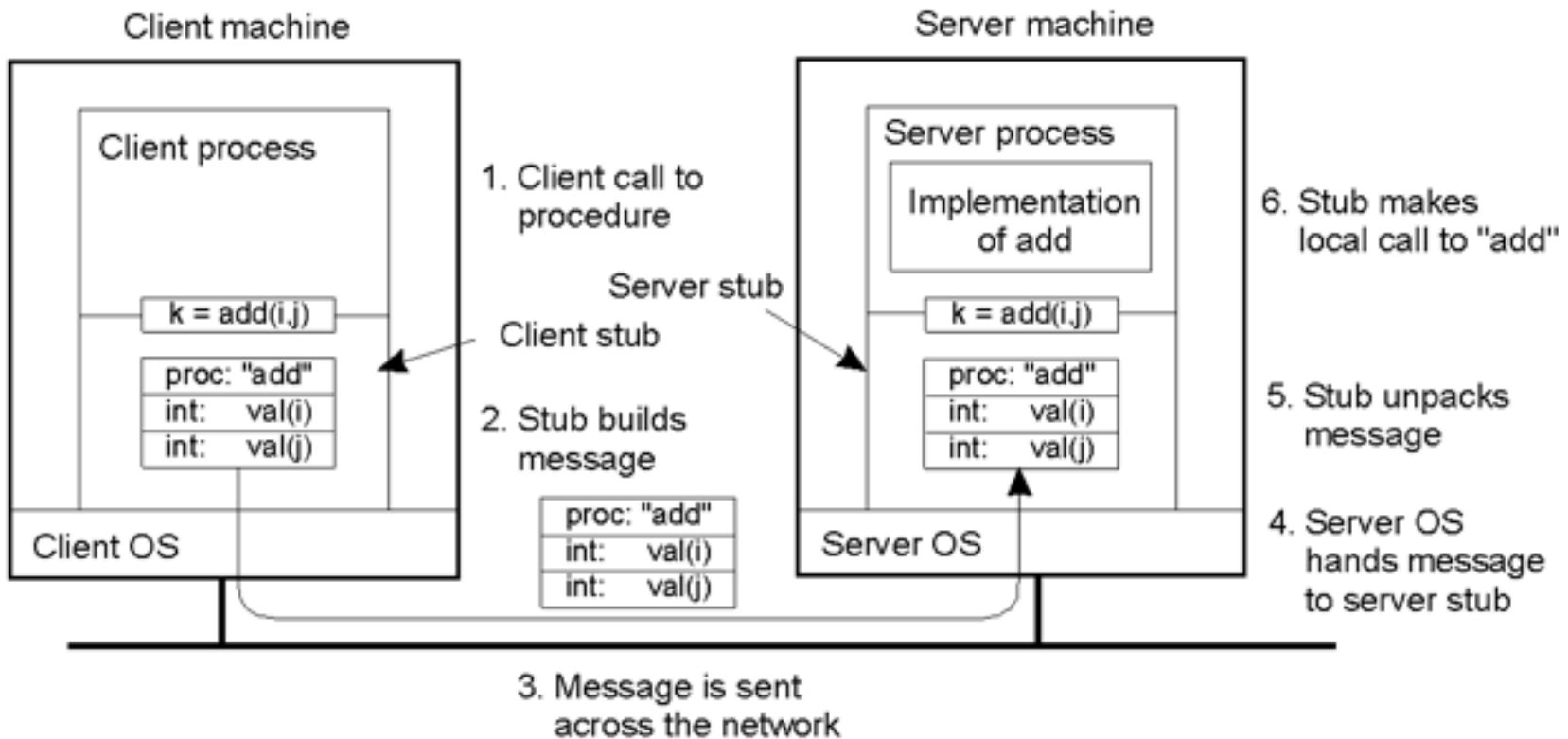
count = read(fd, buf, bytes)



Parameter passing in a local procedure call – the stack before & after the call
(passing by value)

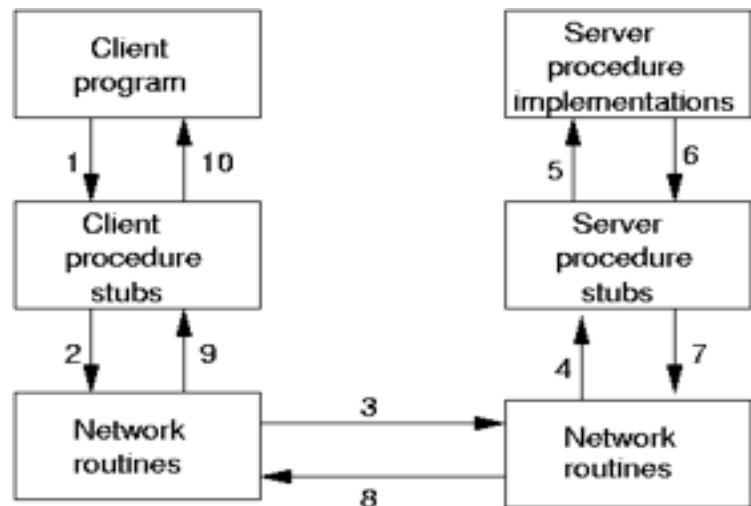
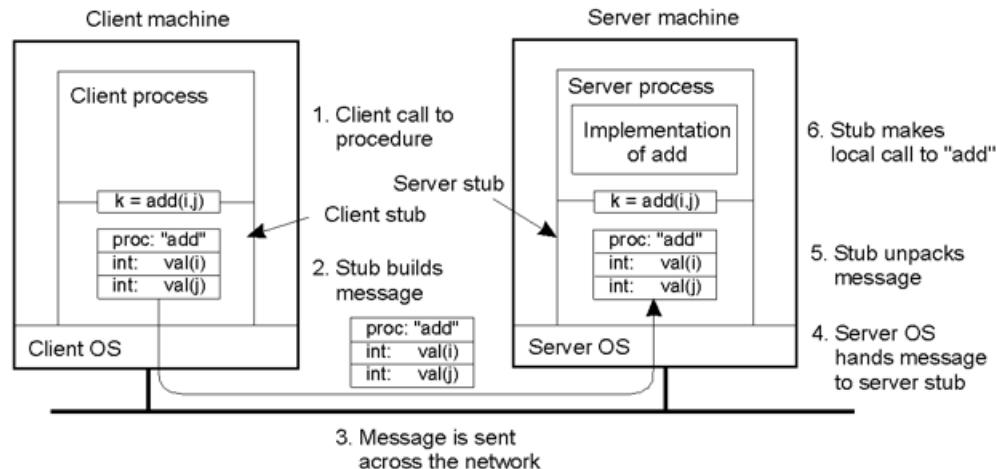
Passing Value Parameters

- Steps involved in doing remote computation through RPC



Under the hoods: Steps of a Remote Procedure Call. Stubs

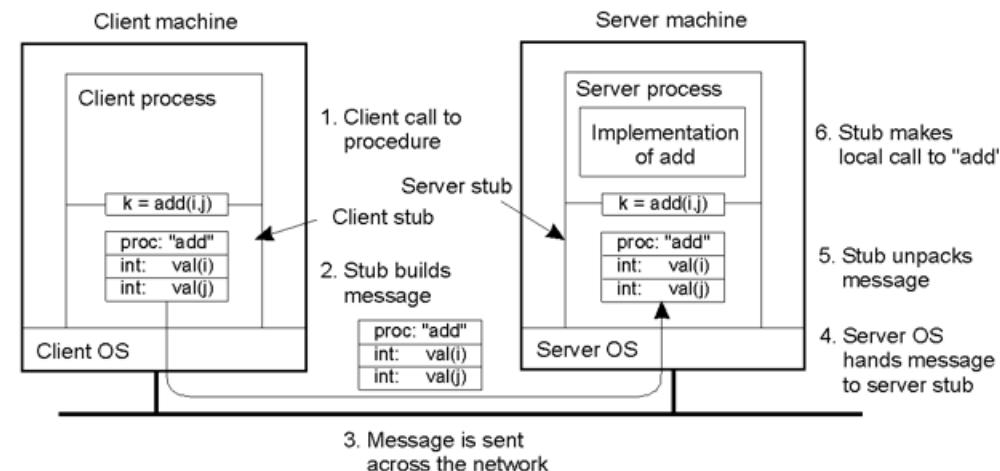
1. Client procedure calls client stub in normal way
2. Client stub builds message calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

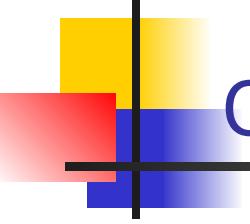


Under the hoods: Parameter marshaling

More than just wrapping parameters into a message:

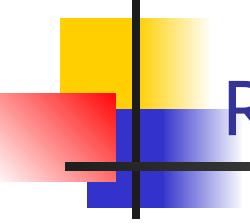
- Client and server machines may have **different data representations** (e.g., size, byte ordering)
- Client and server have to agree on the same **encoding**:
 - How are basic data values represented (integers, floats, characters)
 - How are complex data values represented (arrays, structures)





Outline

- Mechanics. How does it actually work ...
 - ... and limitations
- RPC in practice.
- Case study: The Network File System
- Discussion:
 - Reliable RPC
 - Asynchronous RPC



RPC in mechanics practice

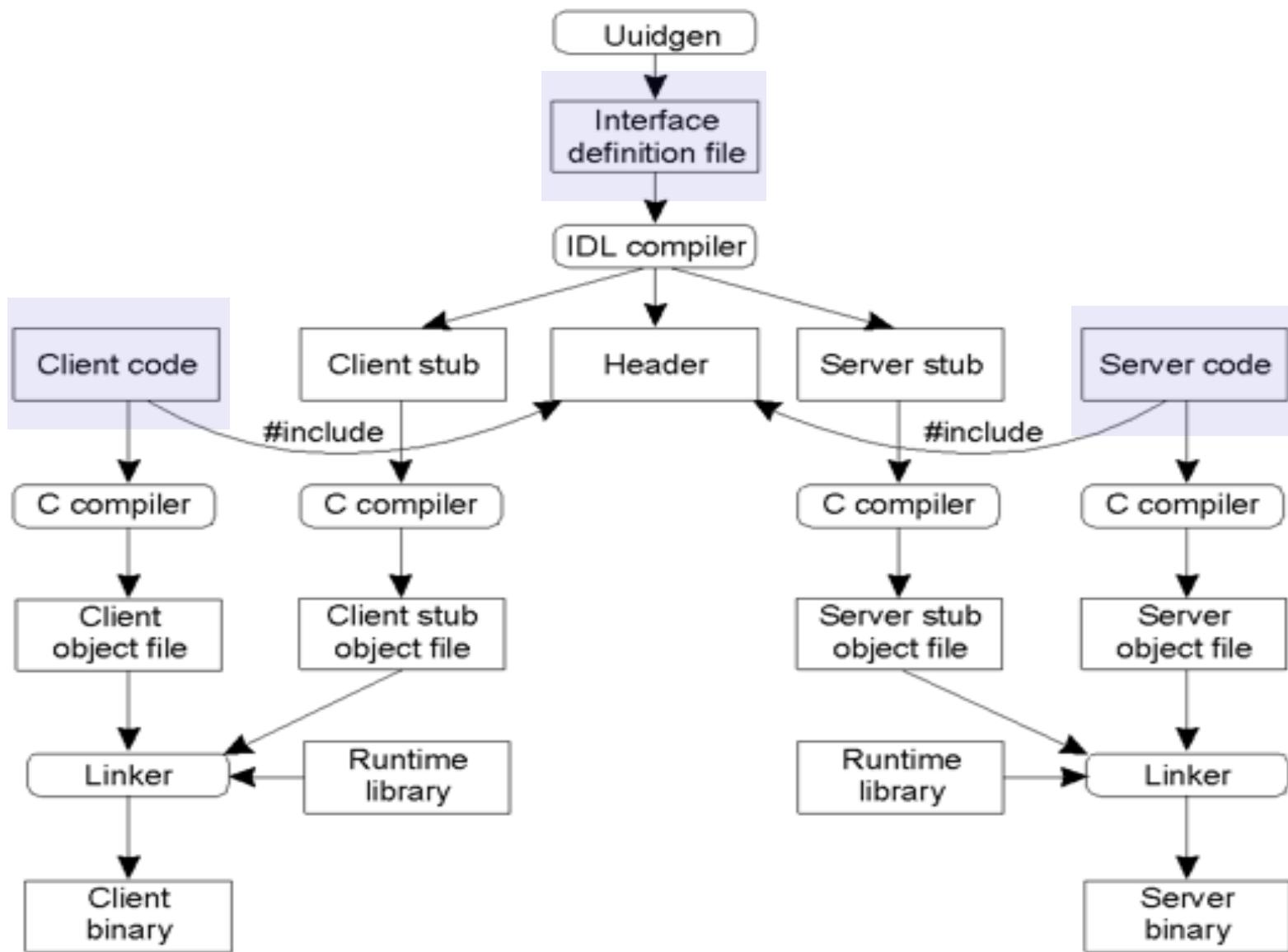
Objective:

- let the developer concentrate on only the client- and server-specific code;
- the RPC system (generators and libraries) do the rest.

What components does an RPC system consist of?

- **Standards** for wire format of RPC msgs and data types. (e.g. Sun's XDR)
- **Library** of routines to marshal / unmarshal data.
- **Stub generator** or "RPC compiler", to produce "stubs".
 - For client: marshal arguments, call, wait, unmarshal reply.
 - For server: unmarshal arguments, call real fn, marshal reply.
- **Server framework**: Dispatch each call message to correct server stub.
- **Client framework**: Give each reply to correct waiting thread / callback.
- **Binding mechanisms**: how does client find the right server?

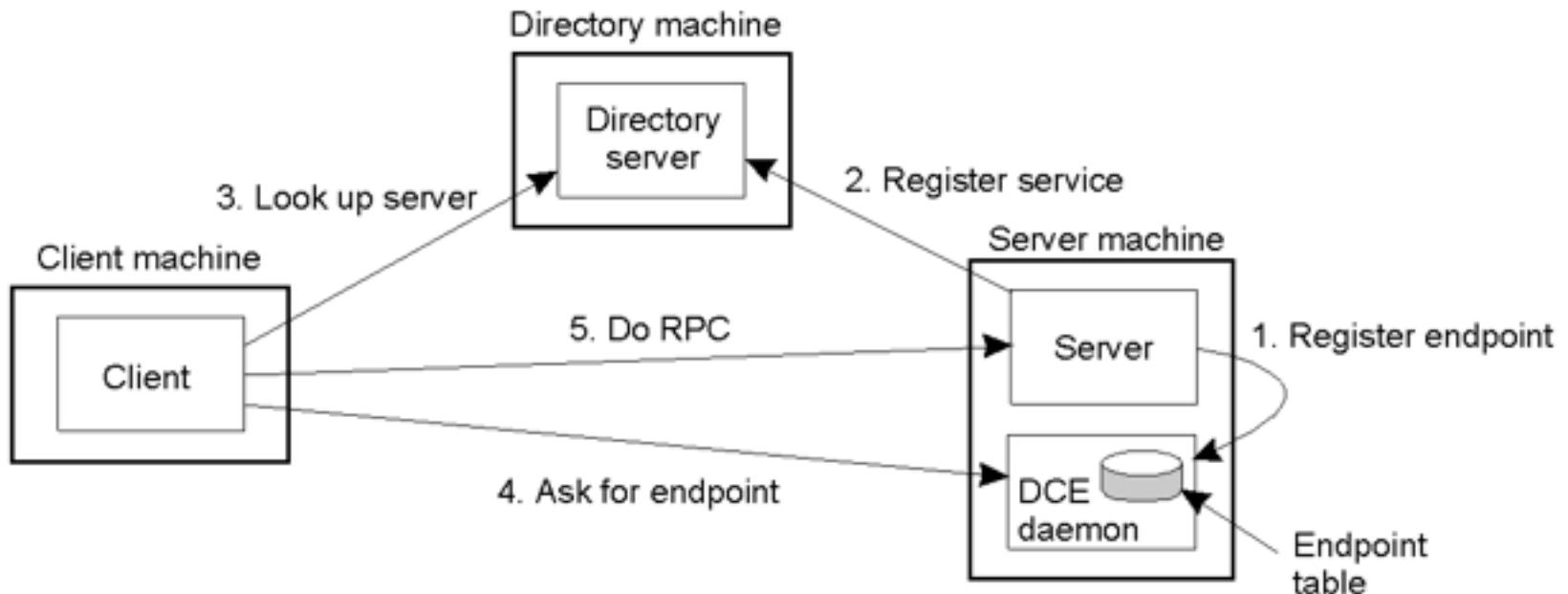
RPC in practice: Writing a Client and a Server

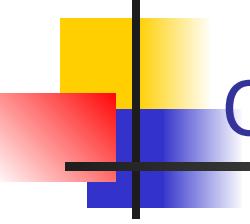


Practicalities: Binding a Client to a Server

Issues:

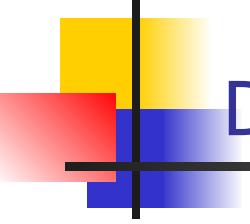
- Client must locate the server machine
- Client must locate the server (port)





Outline

- Mechanics. How does it actually work ...
 - ... and limitations
- RPC mechanics in practice.
- Discussion: does one achieve transparency
 - Designing an RPC framework
 - Case study: The Network File System



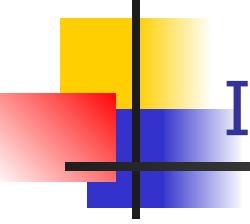
Discussion: Do we achieve transparency?

Yes:

- Hides wire formats and marshal / unmarshal.
- Hides details of send / receive APIs
- Hides details of transport protocol (TCP vs. UDP)
- Hides who the client is.

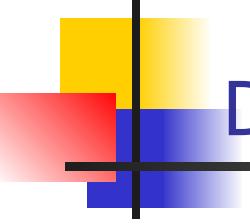
No:

- Latency, performance
- Depending on the language to support ...
 - Parameter passing, typing info needed.
 - Global variables, pointers
- Concurrency
- Failures



Issue (I): Should the IDL break transparency?

- Original take (Sun RPC): attempt to provide total transparency
- Today: same interface but force programmer to handle exceptions for remote calls (e.g.,)



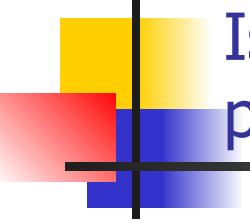
Discussion: Do we achieve transparency?

Yes:

- Hides wire formats and marshal / unmarshal.
- Hides details of send / receive APIs
- Hides details of transport protocol (TCP vs. UDP)
- Hides who the client is.

No:

- Latency, performance
- Depending on the language to support ...
 - Parameter passing, typing info needed.
 - Global variables, pointers
- Concurrency
- Failures

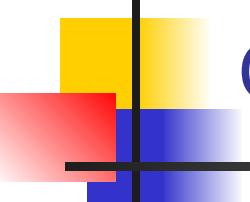


Issue (II): Different languages have different parameter passing semantics

Traditional parameter-passing possibilities:

- By value
 - Parameter value is copied on the stack and/or sent on the wire
- By reference
 - Reference/Identifier is passed
 - How does this work for remote calls?
- Copy in/copy out
 - A copy of the referenced object is copied in/out
 - While procedure is executed, nothing can be assumed about parameter values

Do they lead to different results after a procedure call?



Quiz sample question: Argument passing

```
Procedure Foo (integer X, integer Y) {
```

```
    X = 10
```

```
    Y = X + 20
```

```
}
```

```
.....
```

```
integer a=0
```

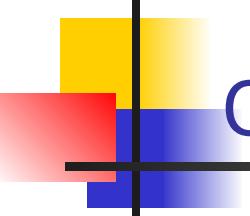
```
integer b=0
```

```
Foo (a, a)
```

```
Print (a,b)
```

What is printed if parameters are passed

- **By value?**
- **By reference?**
- **By copy in/copy out**



Quiz sample question

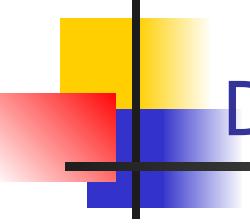
You are to implement the RPC support for C. Can your RPC implementation support 'unions'? Explain your answer.

The C language has a construct called union where the same memory location can hold one of several alternative data types.

Example: The following piece of code declares a new union type `union_def`. Variables of this type can hold either one of an integer, a float or a character. Then in the variable is initialized with an float then with an integer.

```
union union_def { int a; float b; char c; } ; // define the type
union union_def union_var; // define the variable
union_var.b=99.99; or // initialize the variable
union_var.a=34;
```

Key issue: static analysis can not guarantee how a union passed as an argument will be used inside a procedure.



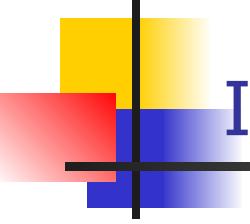
Discussion: Do we achieve transparency?

Yes:

- Hides wire formats and marshal / unmarshal.
- Hides details of send / receive APIs
- Hides details of transport protocol (TCP vs. UDP)
- Hides who the client is.

No:

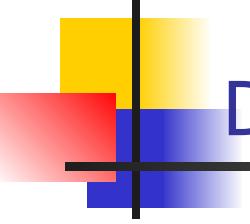
- Latency
- Depending on the language to support ...
 - Parameter passing, typing info needed.
 - Global variables, pointers
- Concurrency
- Failures



Issue III: Dealing with failures

Failures: crash, omission, timing, arbitrary

- some hidden by underlying network layers
(e.g., TCP)



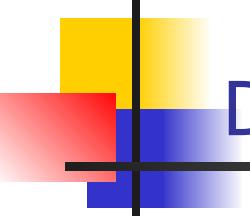
Dealing with failures

Failures: crash, omission, timing, arbitrary

- some hidden by underlying network layers
(e.g., TCP)

What can go wrong:

1. Client cannot locate server
2. Client request is lost
3. Server crashes
4. Server response is lost
5. Client crashes

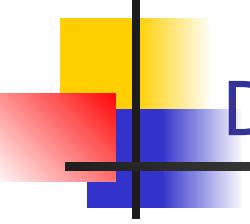


Dealing with failures (1/5)

What can go wrong with RPC:

1. Client cannot locate server
2. Client request is lost
3. Server crashes
4. Server response is lost
5. Client crashes

[1:] **Client cannot locate server.** Relatively simple
→ just report back to client application
(but transparency is lost!)



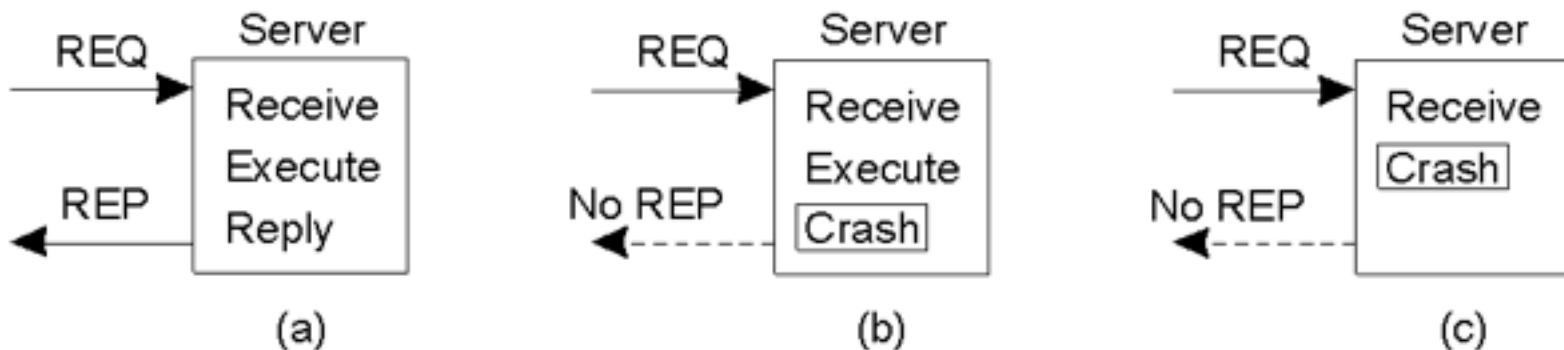
Dealing with failures (2/5)

[2:] Client request lost. Just resend message
(and use message ID to uniquely identify messages)

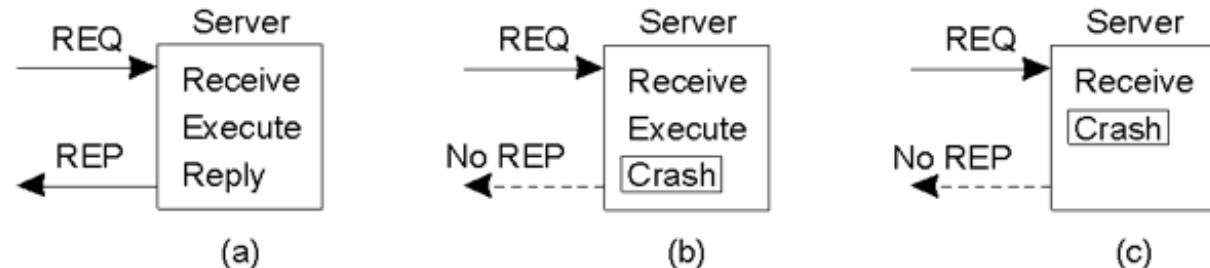
- But: now one has to deal with state at the server.
 - New question: for how long to maintain this state

Dealing with failures RPC (3/5)

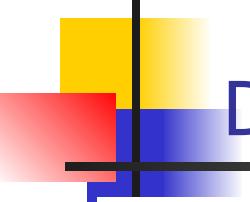
[3] **Server crashes** → harder as you don't know what the server has already done:



[3] Server crashes → you don't know what the server has already done



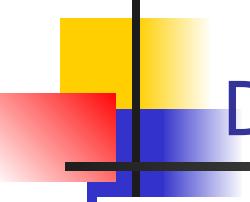
- Solution: None that is general!
- Possible avenues
 - A.] (works sometimes) [At the application level] make your operations **idempotent**: repeatable without any harm done if it happened to be carried out before.
 - B.] Add sequence numbers so that you can repeat invocation
 - Decide on what to expect from the system:
 - **At-least-once-semantics**: The server guarantees it will carry out the operation at least once, no matter what.
 - **At-most-once-semantics**: The server guarantees it will carry out an operation at most once.



Dealing with failures RPC (4/5)

[4:] **Lost replies** → Detection hard: because it can also be that the server is just slow. You don't know whether the server has carried out the operation

Solution: Do not attempt to diagnose, resend the request.

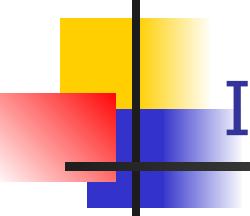


Dealing with failures RPC (5/5)

[5:] **Client crashes** → Issue: The server is doing work and holding resources for nothing (**orphan** computation).

Possible solutions:

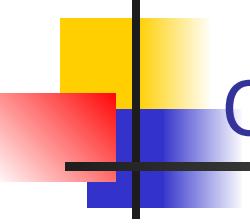
- [orphan extermination] Orphan is killed by client when it reboots
 - But expensive to log all calls,
 - and it may never work (grand-orphans, partitions)
- [reincarnation] Broadcast new epoch number when recovering → servers kill orphans
- [expiration] Require computations to complete in a T time units. Old ones are simply removed.



Issue III: Dealing with failures (summary)

Resulting RPC semantics

<i>Fault tolerance measures</i>			<i>Resulting RPC call semantics</i>
<i>Retransmit request</i>	<i>Duplicate filtering</i>	<i>Re-execute procedure or retransmit reply</i>	
No	Not applicable	Not applicable	<i>Maybe</i>
Yes	No	Re-execute proc	<i>At-least-once</i>
Yes	Yes	Retransmit reply	<i>At-most-once</i>



Outline

- Mechanics. How does it actually work ...
 - ... and limitations
- RPC mechanics in practice.
- Discussion: does one achieve transparency
 - Designing an RPC framework
 - Case study: The Network File System

Case study: Network File System (NFS)

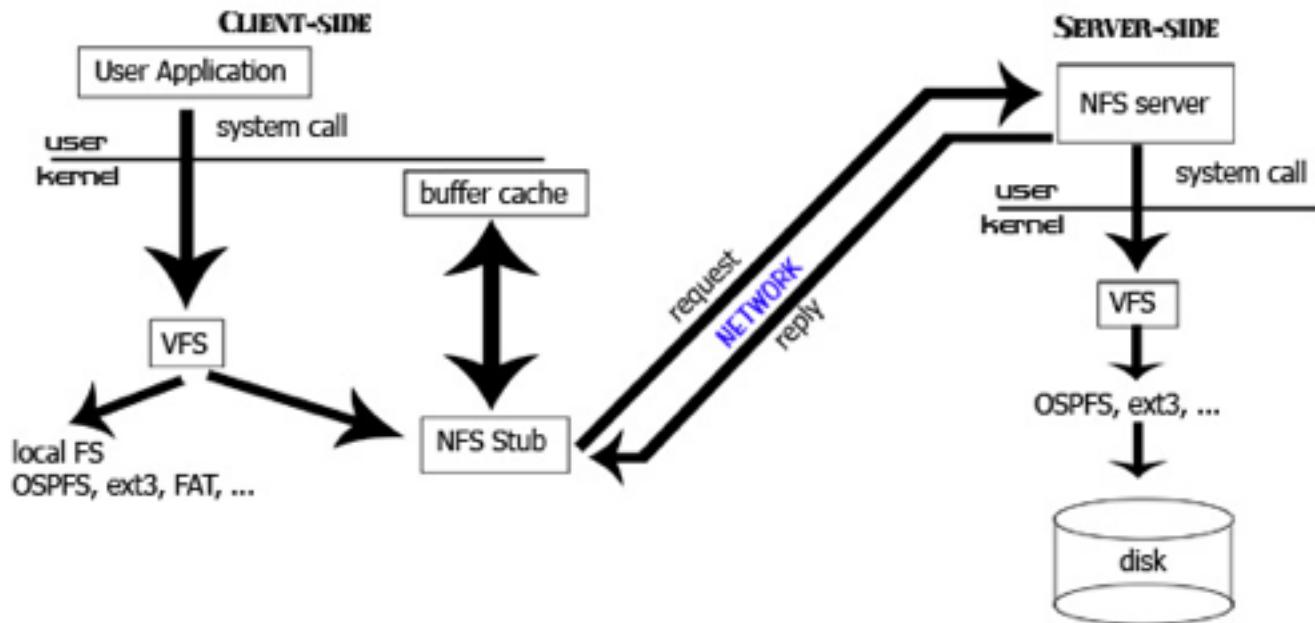
- What does the RPC split up in this case? App calls, syscalls, kernel file system, local disk
 - In kernel, just below syscall interface. "vnode" layer.

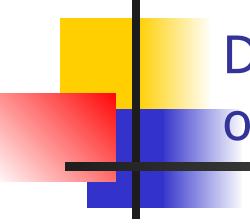
Is transparent

- Syntax
- Semantics

Not enough
must map

- Other
be



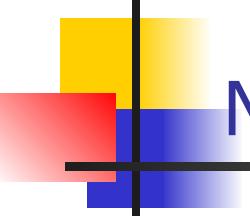


Does NFS preserve the **semantics**
of file system operations?

New semantics: Open() system call:

- Originally, open() only failed if file didn't exist.
- Now open (and all others) can fail if server has died.
 - Obs: Apps have to know to retry or fail gracefully.
 - Obs: Think of process coordination through FS
- *even worse* open() could hang forever,
 - This was never the case before.
 - Apps have to know to set their own timeouts if they don't want to hang.

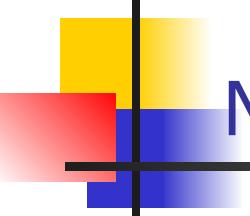
This is fundamental, not an NFS quirk.



NSF: New Semantics ... (II)

New semantics: close() system call

- Originally client only waits for disk in write()
 - close() never returned an error for local file system.
- Now: might fail if server disk out of space.
 - So apps have to check close() for out-of-space, as well as write().
 - This is caused by NFS trying to hide latency by batching.
 - Side effect of async write RPCs in client, for efficiency.
 - They could have made write() synchronous (and much slower!).

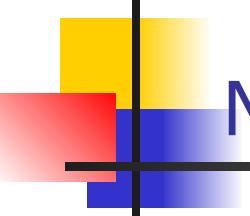


NSF: New Semantics ... (III)

New semantics: deletion of open files

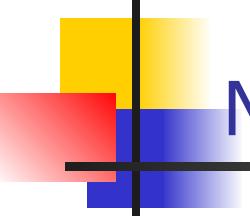
Scenario: I open a file for reading. Some other client deletes it while I have it open.

- Old behavior: my reads still work.
- New behavior: my reads fail.
 - Side-effect of NFS's **statelessness**.
 - NFS server never remembers all operations it has performed.
- How would one fix this?



NSF: New Semantics examples ... (IV)

- Scenario:
 - `rename("a", "b")` on an NFS file.
 - Suppose server performs rename, crashes before sending reply.
- NFS client re-sends `rename()`. But now "a" doesn't exist, so produces an error. This never used to happen.
- Another side-effect of NFS's **statelessness**.
- Hard to fix: hard to keep that state consistent across crashes. Update state first? Or perform operation first?



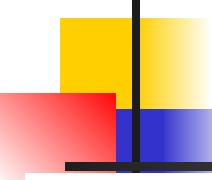
NFS: security

Security is totally different

- On local system: UNIX enforces read/write protections:
Can't read my files w/o my password
- On NFS: Server believes whatever UID appears in NFS request
 - Anyone on the Internet can put whatever they like in the request
 - (Or you (on your own workstation) can su to root, then su to me[2nd su requires no password])

Why aren't NFS servers ridiculously vulnerable?

- Hard to guess correct file handles.
- This is fixable (SFS, AFS, even some NFS variants do it)
 - Require clients to authenticate themselves cryptographically.
 - Hard to reconcile with statelessness.



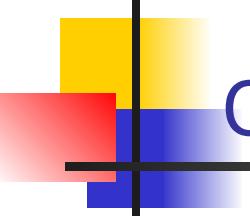
NFS case-study summary

Areas of RPC non-transparency

- 1. Partial failure, network failure
- 2. Latency
- 3. Efficiency/semantics tradeoff
- 4. Security. You can rarely deal with it transparently.
- 5. Pointers. Write-sharing.
- 6. Concurrency (if multiple clients)

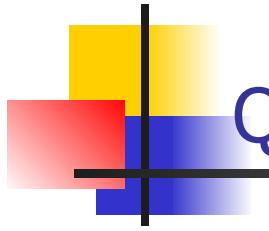
However, it turns out none of these issues prevented NFS from being useful.

- People fix their programs to handle new semantics.
- ... install firewalls for security.
- And get most advantages of transparent client/server.

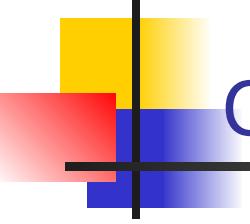


Outline

- Mechanics. How does it actually work ...
 - ... and limitations
- RPC mechanics in practice.
- Discussion: does one achieve transparency
 - Designing an RPC framework
 - Case study: The Network File System
- One more usecase: Java RMI

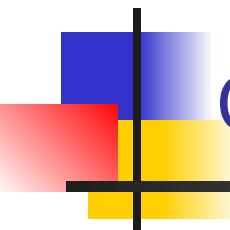


Qs

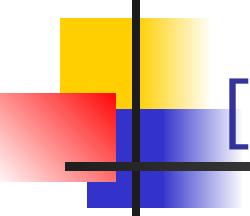


Concepts to remember

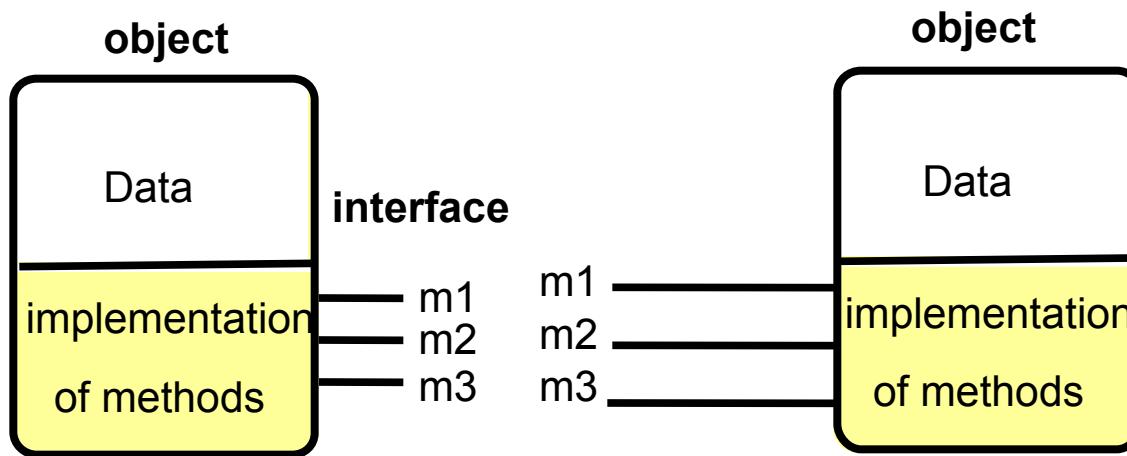
- Design choices
 - RPC call semantics
 - At-most-once / at-least-once semantics
 - Idempotent calls
 - Statefull/Stateless servers



One more case-study: Java RMI



[Traditional] Objects



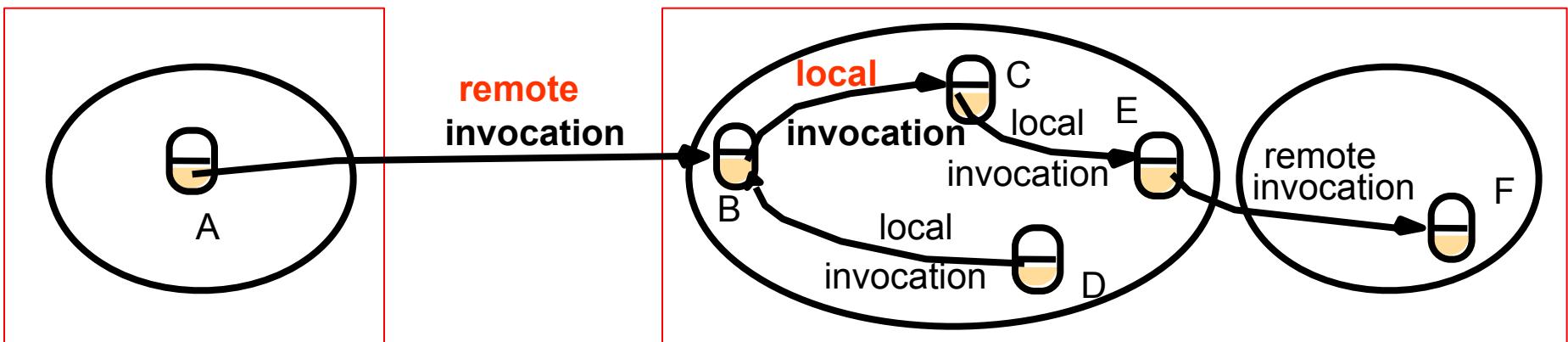
- **Object** = data + methods
 - logical and physical **encapsulation**
 - **Identified** by means of **references**
 - first class **citizens**, i.e., can be passed as arguments
- Interaction defined via **interfaces**
 - define types of arguments and exceptions for methods

The Distributed Object Model: Idea

Programs are (**logically and physically**) partitioned into objects

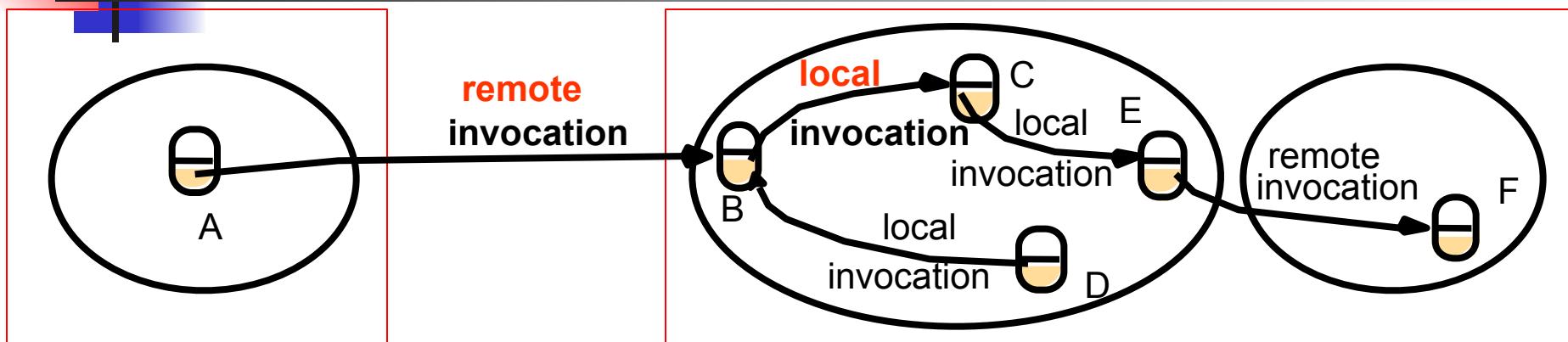
[Idea] **Distribute objects across memory spaces**

- Client-server relationship at the object level

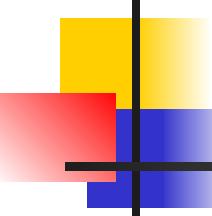


The Distributed Object Model:

What's needed?



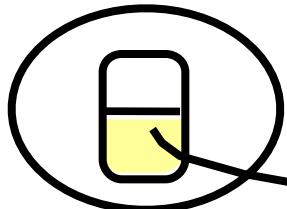
- Extend the object model with:
 - **Remote** object references
 - **Remote** method invocation (and new exceptions!)
 - **Remote** interfaces (to differentiate local access from remote one)
- Extend runtime
 - Distributed garbage collection



Remote Object References

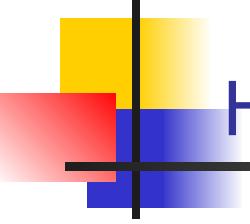
- [Traditional] Object **references**
 - used to identify/access objects which live in **processes**
 - can be passed as **arguments** and **results**
 - can be stored in **variables**
- **Remote object references**
 - object **identifiers** in a distributed system
 - must be **unique** across space and time
 - error returned if accessing a deleted object
 - may support **relocation** (e.g., in CORBA)

<i>32 bits</i>	<i>32 bits</i>	<i>32 bits</i>	<i>32 bits</i>	
Internet address	port number	time	object number	interface of remote object



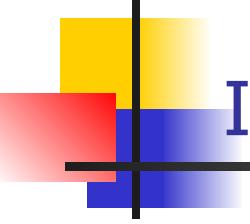
Argument passing (for JavaRMI)

- Specify **extern**
 - no direct reference
 - local interface
- Need to decide
 - Arguments passing (call by value / reference / copy in/out)
 - Retry policy (at-least-once / at-most-once)
 - **At-most-once invocation**
- If argument is **primitive type**
 - **Pass by value**
- If the object implements **Remote interface**
 - **Pass by reference**
- If object implements **Serializable interface**
 - **Pass by copy out**
 - Note: **deep copy!**
- If **none of the above**
 - **Exception is raised**



Handling Remote Objects

- New exceptions
 - raised in remote invocation
 - clients / servers need to handle exceptions
 - **timeouts** in case server crashed or too busy
- Garbage collection
 - **distributed** garbage collection may be necessary
 - combined local and distributed garbage collector
 - multiple solutions are possible

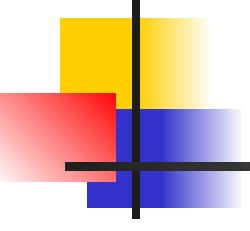


Issues: Granularity & Concurrency

- [generally] an RMI method invocation results in :
 - A new TCP connection to the remote server
 - Creation of a new thread on the remote server

Resulting issues

- RMI calls should be used for coarse-grain computation.
- Concurrency to be managed by application



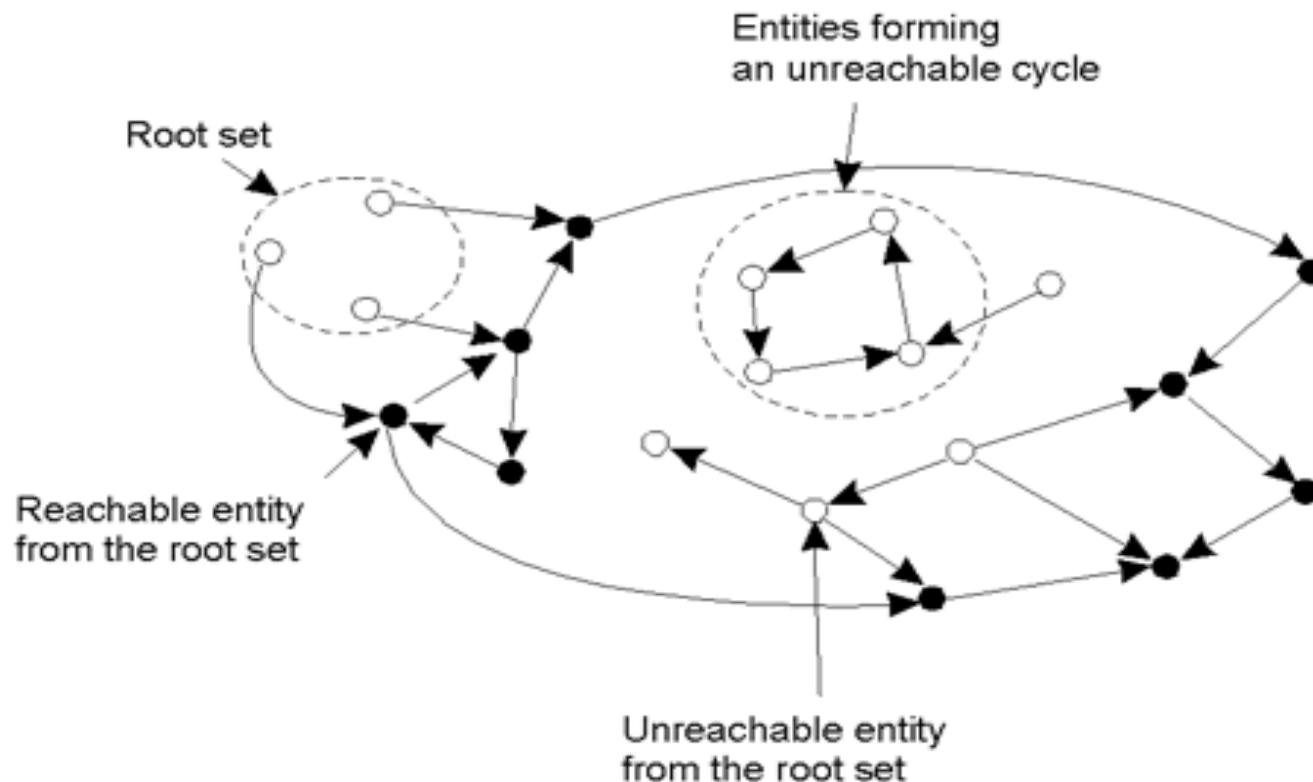
Mechanics and internals

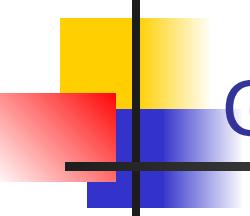
- Reference counting

Garbage collection in single box systems

Solutions

- Reference counting
- Tracing based solutions (mark and sweep)





Garbage collection in distributed systems

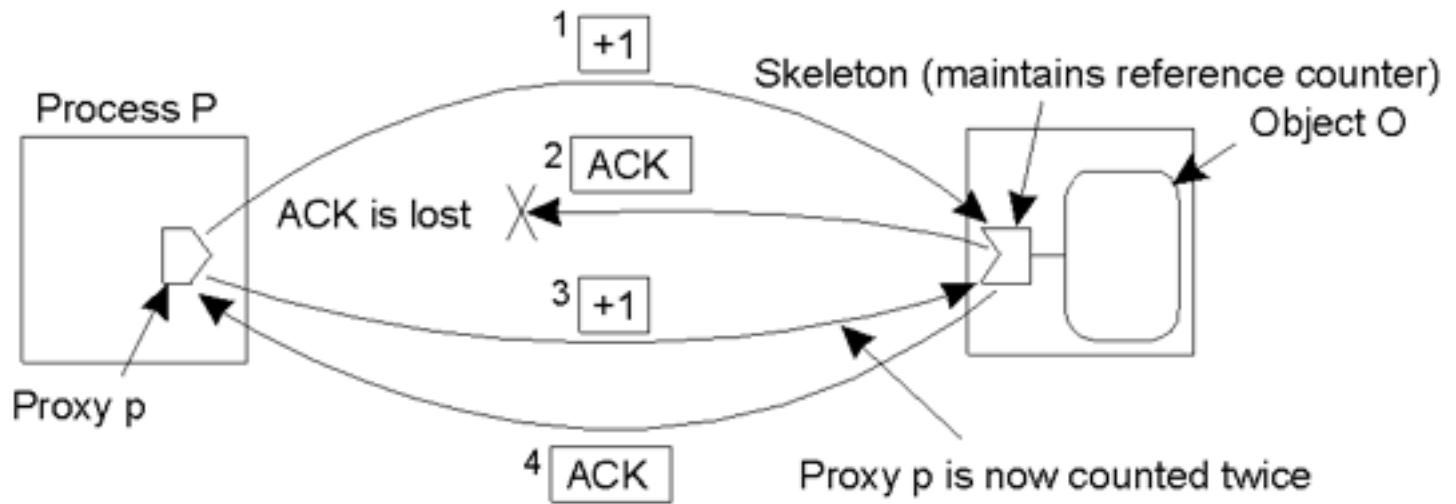
- Why is it different?
 - References distributed across multiple address spaces
- Why a solution may be hard to design:
 - Unreliable communication
 - Unannounced node failures
 - Overheads

Which one to start from for a distributed garbage collection substrate?

- Reference counting
- ~~Tracing based solutions (mark and sweep)~~

Reference Counting

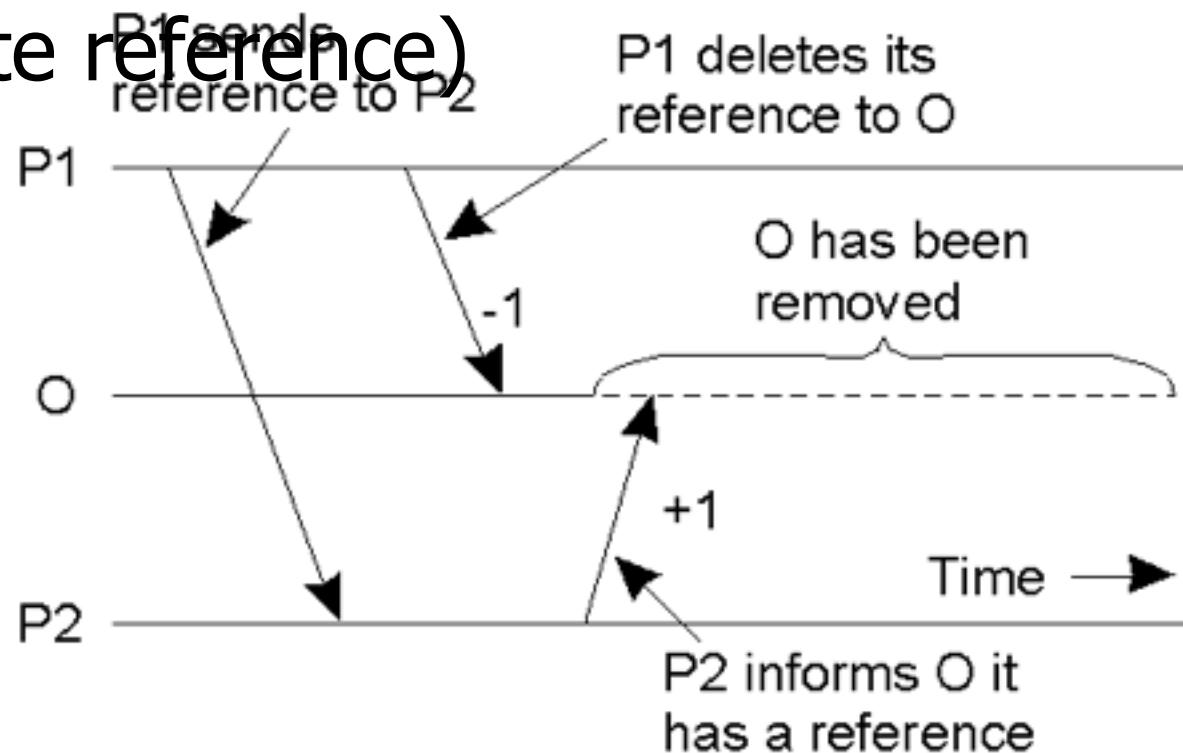
- The problem: maintain a proper reference count in the presence of unreliable communication.

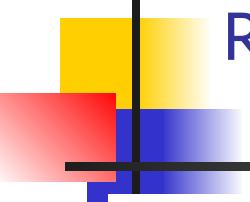


- Reliability problems: Similar to RPC

Reference Counting (cont)

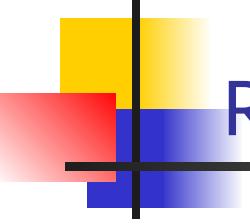
- How will creating remote references work?
- How will passing remote references work?
 - P¹ attempts to pass to P² a remote reference to O
- What happens if the client crashes (while holding a remote reference)





Reference Listing (Java RMI's solution)

- Solution
 - Object 'stub' maintains a list of client proxies
 - Each remote reference regularly reminds skeleton on its existence
- Creating a remote reference
 - Assume P creates remote reference to object O
 - P sends its identification to O 'stub'
 - O acknowledges and stores P identity
 - P creates the remote reference
- Copying a remote reference
 - i.e: P¹ attempts to pass to P² a remote reference to O
 - What are the steps?
- Pros/Cons?



Reference Listing (Java RMI's solution)

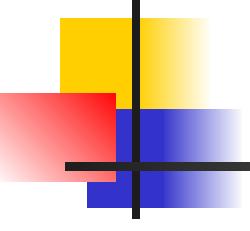
- Advantages:
 - add/delete are **idempotent**
 - i.e. duplicate operations have no effect
 - no reliable communication required
- Drawback
 - overheads/scalability – the list of proxies can grow large
 - unannounced client failures may lead to temporary resource leak

A larger category: Soft-state mechanisms (Leases)

- **Mechanism:**
 - Producer regularly (re)sends state to receiver(s) over a (lossy) channel.
 - Receivers keep state and associated timeouts. Deletes if no update.

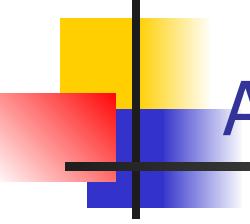


- **Advantages:**
 - Decouples state producer and consumer: no explicit failure detection and state removal messages
 - 'Eventual' state
 - Easy to tune
- **Works well in practice:** RSVP, RIP, tons of other systems.

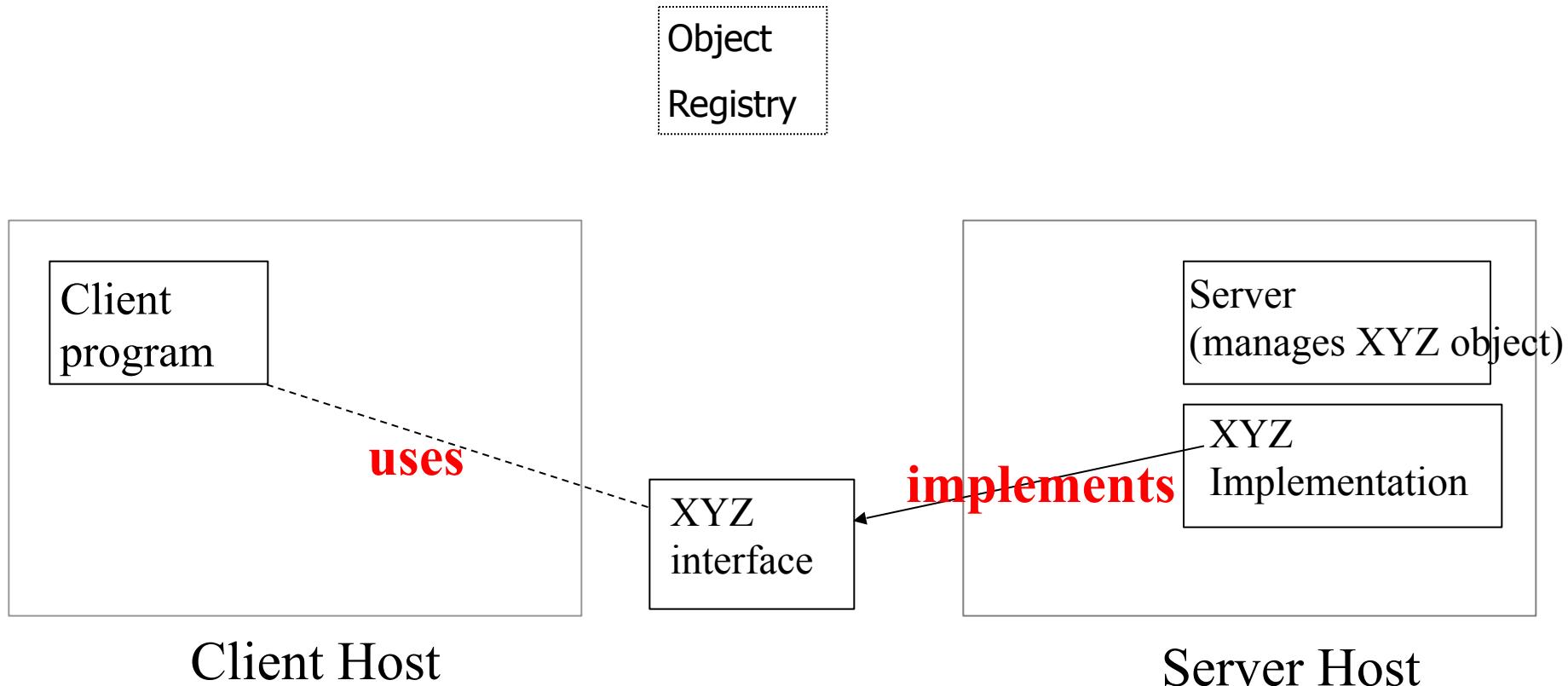


Mechanics and internals

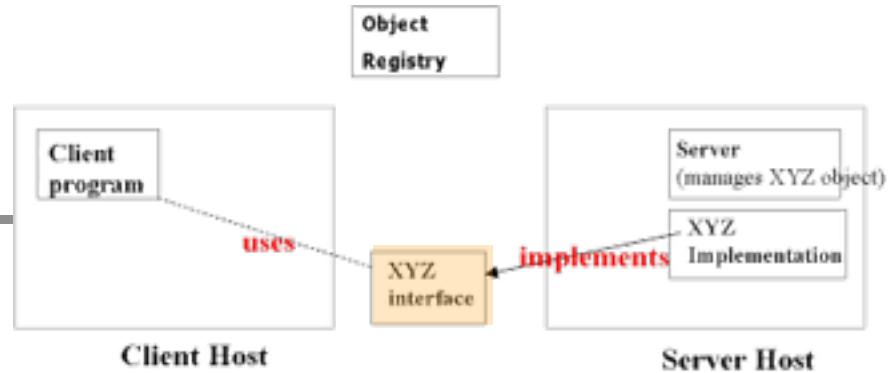
- Reference counting
- **Minimal code intro**



A programmer's view



Hello World: Remote Interface



```
import java.rmi.*;
```

```
public interface HelloInterface extends Remote {
```

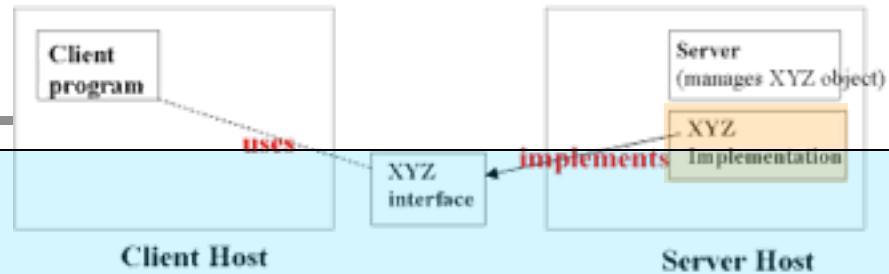
```
/*
 * Remotely invocable method,
 * returns a message from the remote object,
 * throws a RemoteException
 * if the remote invocation fails
 */
```

```
public String say() throws RemoteException;
```

```
}
```

Hello World: The Remote Object

Object
Registry

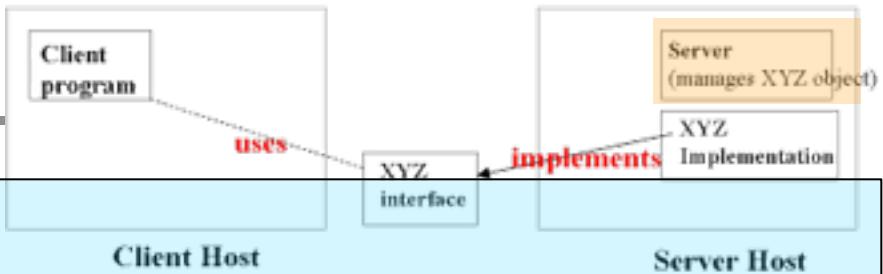


```
import java.rmi.*;  
import java.rmi.server.*;
```

```
public class HelloImpl extends UnicastRemoteObject  
    implements HelloInterface {  
  
    private String message;  
    /* Constructor for a remote object  
     * Throws a RemoteException if the object handle  
     * cannot be constructed  
     */  
    public HelloImpl(String msg) throws RemoteException{  
        message = msg;  
    }  
    /* Implementation of the remotely invocable method  
     */  
    public String say() throws RemoteException {  
        return message;  
    }  
}
```

Hello World: The 'Server'

Object
Registry

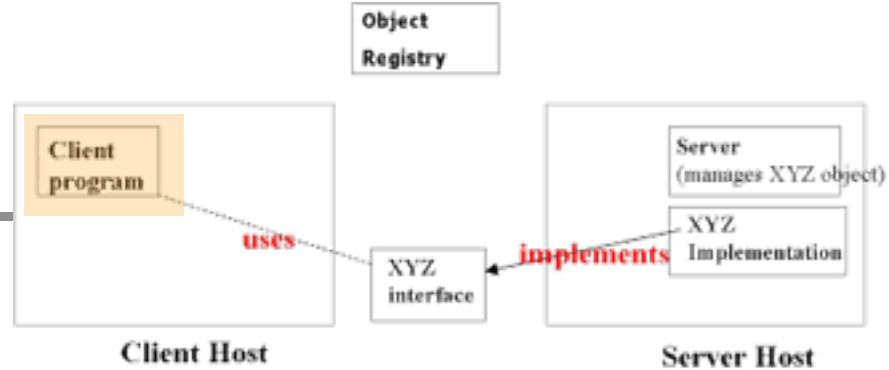


```
import java.io.*;  
import java.rmi.*;
```

```
public class HelloServer{  
    /*  
     * Server program for the "Hello, world!" example.  
     */  
    public static void main (String[] args) {  
        try {  
            Naming.rebind ("SHello",  
                           new HelloImpl ("Hello, world!"));  
            System.out.println ("HelloServer is ready.");  
        } catch (Exception e) {  
            System.out.println ("HelloServer failed: " + e);  
        }  
    }  
}
```

Hello World: The Client

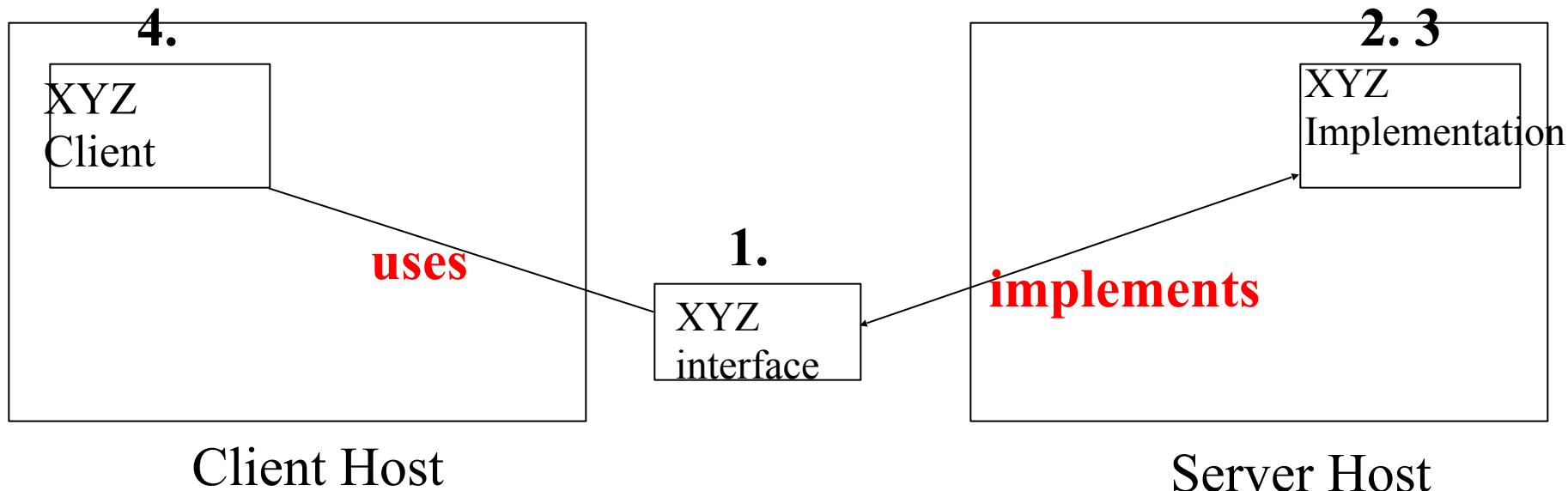
```
import java.io.*;  
import java.rmi.*;
```



```
public class HelloClient{  
    /*  
     * Client program for the "Hello, world!" example  
     */  
    public static void main (String[] args) {  
        try {  
            HelloInterface hello = (HelloInterface)  
                Naming.lookup ("//matei.ece.ubc.ca/SHello");  
  
            /* ... Now remote calls on hello can be used ...  
             System.out.println (hello.say());  
            }  
            catch (Exception e) {  
                System.out.println ("HelloClient failed: " + e);  
            }  
        }
```

Recap: Steps in developing an RMI application

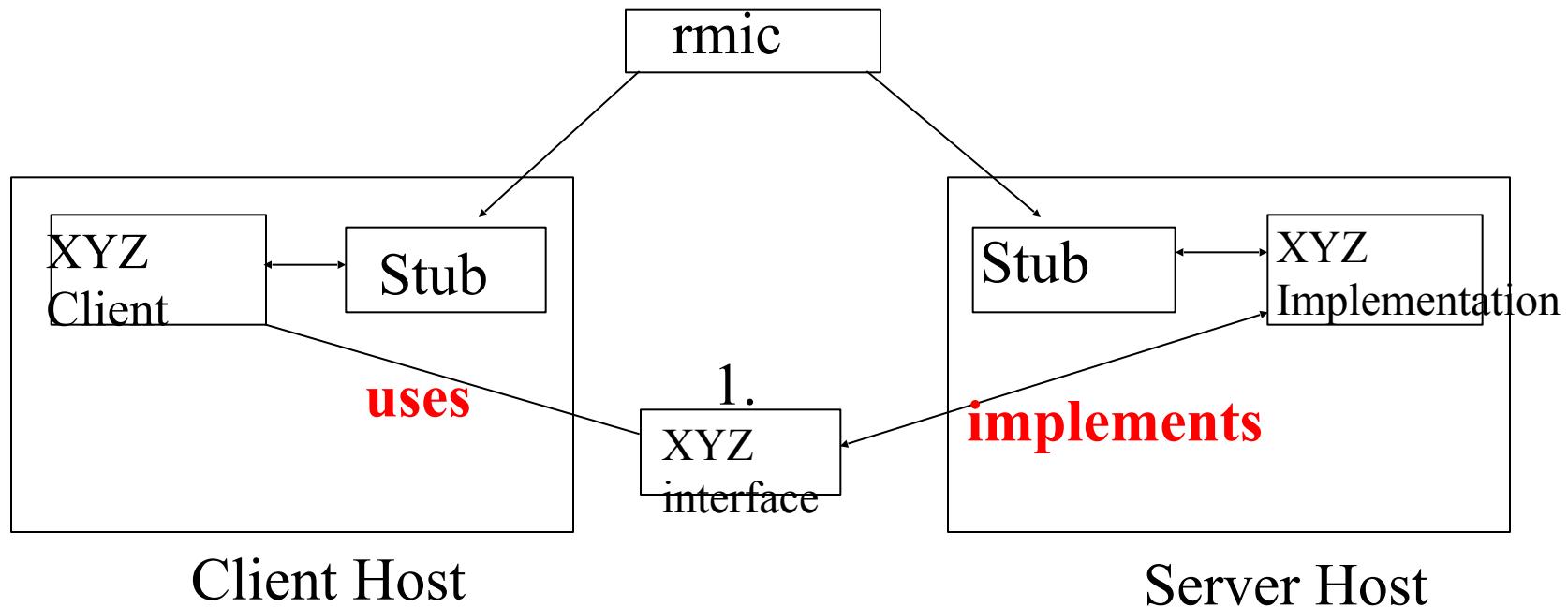
1. Design the **remote interface** for the object.
2. Implement the object specified in the interface.
3. Implement some server side program to bootstrap
4. Implement client/application



RMI – Steps: Compile, Deploy and Use

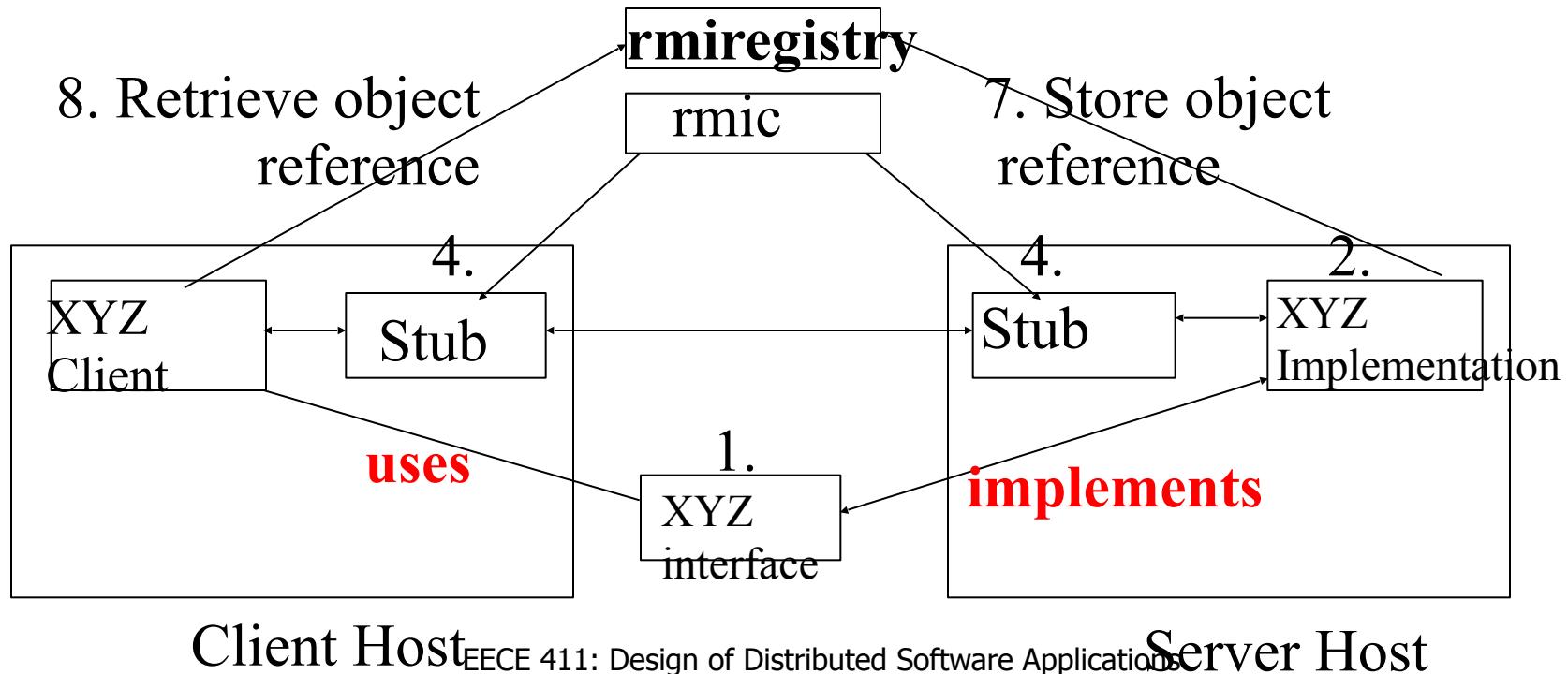
5. Compile.

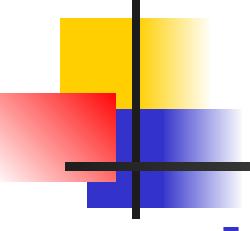
- Compile (javac): interface, server, client
- Compile interface (rmic): Generate the stubs



RMI – Steps: Compile, Deploy and Use

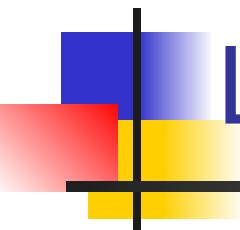
6. Start rmiregistry
7. Start server, instantiate remote object & register it
8. Start client – locate remote object reference
9. Invoke method on remote object





▪ What to remember?

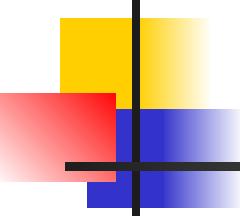
- JavaRMI: one more design data-point for a remote invocation framework
- Soft-state mechanism as an efficient design tool
- Distributed garbage collection design issues



Lecture 7

Data distribution

- Multicast
- Epidemic protocols



Roadmap

Distributed system: collection of independent components that appears to its users as a single coherent system

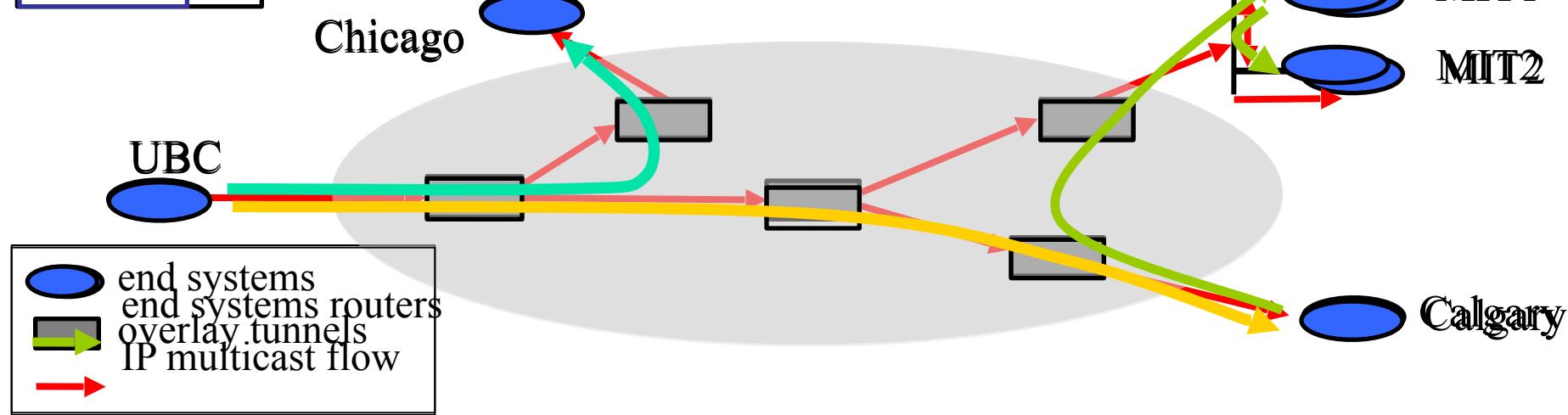
- ⇒ Components need to communicate
 - ⇒ Shared memory
 - ⇒ Message exchange

So far we talked about point-to-point, (generally synchronous, non-persistent) communication

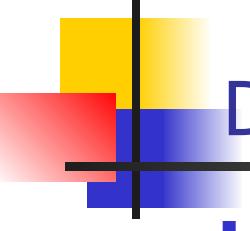
- Socket programming:
 - Message based, generally synchronous, non-persistent
- Client-server infrastructures
 - RPC, RMI
- **Data distribution:**
 - **Multicast**
 - **Epidemic algorithms**

Multicast Communication

Overcast
Multicast



- Two categories of solutions:
 - Based on support from the network: IP-multicast
 - Without network support: application-layer multicast

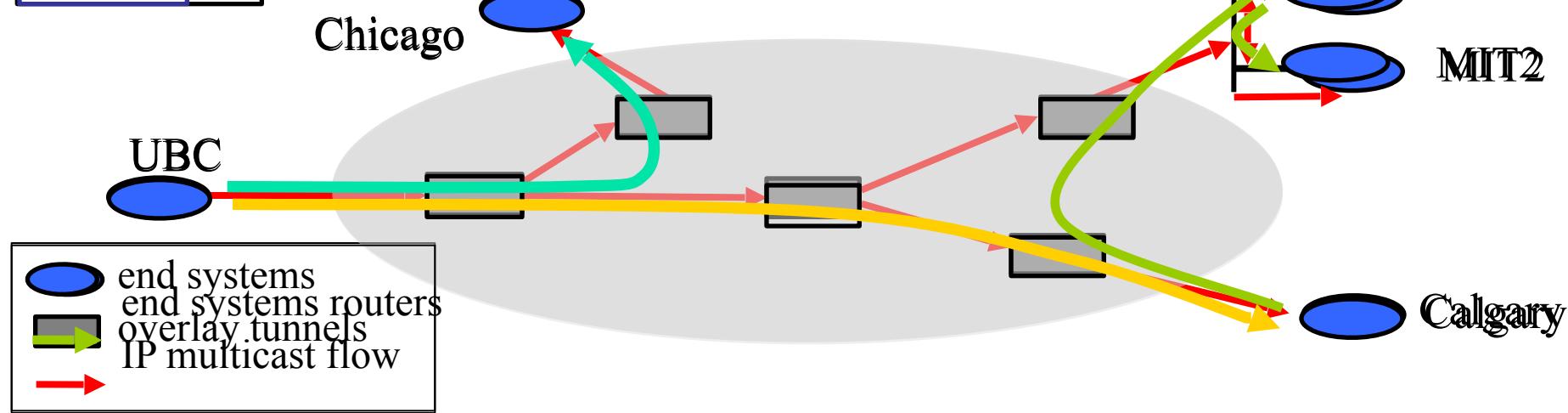


Discussion

- Deployment of IP-multicast is limited. Why?

Application Layer Multicast

Overcast
Multicast

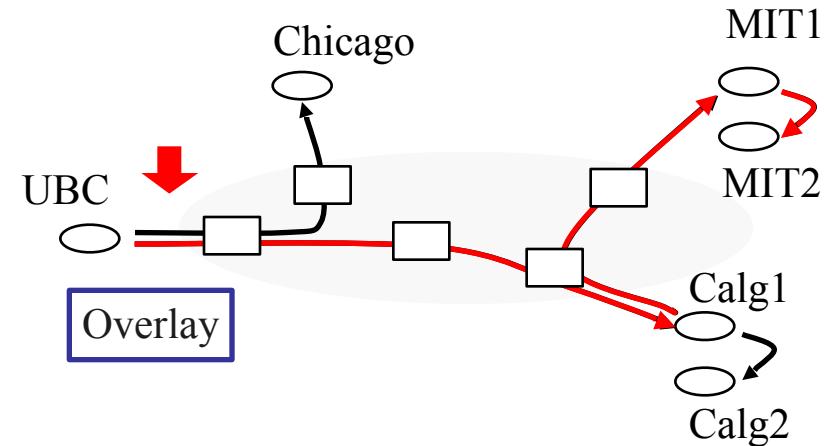
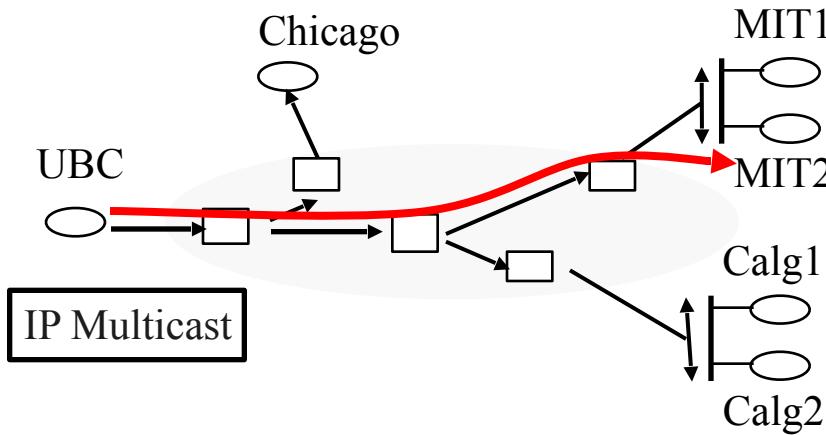


- What should be the success metrics?

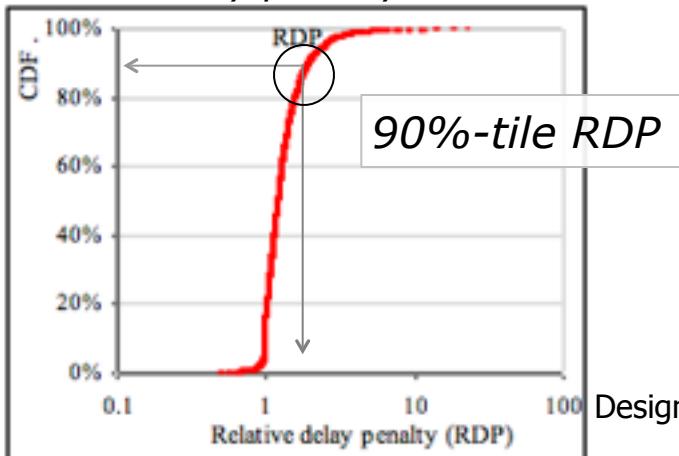
Application-level multicast success metrics: Relative Delay Penalty and Link Stress

Overheads compared to IP multicast

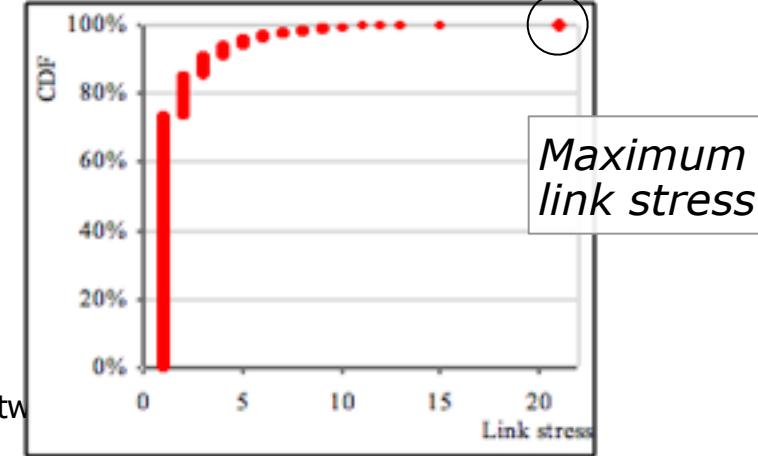
- **Relative Delay Penalty (RDP):** Overlay-delay vs. IP-delay
- **Stress:** number of duplicate packets on each physical link

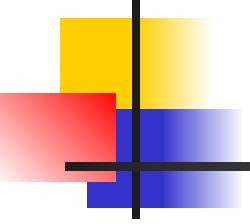


Relative delay penalty distribution



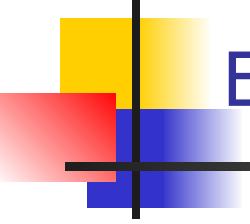
Link stress distribution





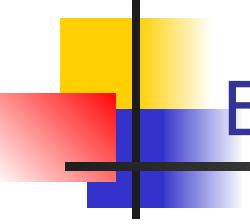
Roadmap ...

- Data distribution:
 - Multicast
 - **Epidemic algorithms**



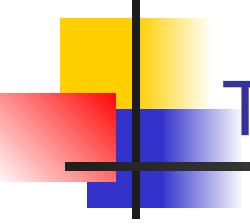
Epidemic algorithms: Context

- **Goal:**
 - Propagate from one initial data source (node) to all participants
 - [Assumption: if multiple sources, there are no write–write conflicts]
- **Context:**
 - Key element: frequent node failures
 - Consequence: message loss, no coordinator
 - Update propagation may be lazy,
 - i.e., solution does not require ‘immediate’ propagation



Epidemic algorithms: Basic Idea

- **Idea**
 - Update operations are initially performed at one node
 - Node passes its updated state to a limited number of 'peers' (often chosen randomly) ...
 - ... which, in-turn, pass the update to other peers
 - Eventually, each update will reach every node

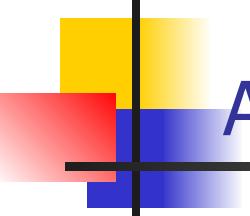


Two main categories:

- **Anti-entropy**:
 - Algorithm works in ‘phases’
 - Each node regularly chooses another node at random
 - they exchange state differences (single direction or both directions)
 - **Issue**: When to stop?

[Variation] Gossiping:

- A node which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).
- **Issue**: When to stop?



Advantages of epidemic techniques

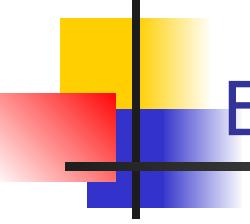
- Asynchronous communication pattern.

Operate in a 'fire-and -forget' mode, where, even if the initial sender fails, surviving nodes will receive the update.
- Autonomous actions.

Enable nodes to take actions based on the data received without the need for additional communication to reach agreement with partners; nodes can take decisions autonomously.
- Robust with respect to message loss & node failures.

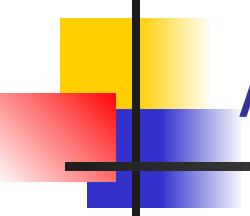
Once a message has been received by at least one of your peers it is almost impossible to prevent the spread of the information through the system.
- Probabilistic model yet Rigorous mathematical underpinnings.

Good framework for reasoning about the spread of information through a system over time.



Epidemic algorithms: Assumptions

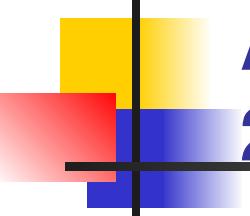
- Idea
 - Update operations are initially performed at one node
 - Node passes its updated state to a limited number of 'peers' (often chosen randomly) ...
 - ... which, in-turn, pass the update to other peers
 - Eventually, each update will reach every node
- Remember our starting assumptions.
 - Frequent node failures
 - Update propagation may be lazy, i.e., not immediate
 - There are no write–write conflicts
- What if I drop each of them?



Amazon S3 incident Sunday, July 20th, 2008

S3 service: Provides a simple web services interface to store and retrieve any amount of data.

- Intent: highly scalable, reliable, fast, and inexpensive data storage infrastructure...
- Scale:
 - Large number of customers, amazon itself uses S3 to run its own global network of web sites.
 - Lots of objects stored
 - 4billion Q4'06 → 40B Q4'08 → 100B Q2'10 → **2 trillion Q2'13**
- SLA guarantees 99.9% monthly uptime
 - Less than 43 minutes of downtime per month



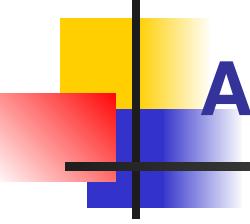
Amazon S3 incident on Sunday, July 20th, 2008

8:40am PDT: error rates began to quickly climb

10 min: error rates significantly elevated and very few requests complete successfully

15 min: Multiple engineers investigating the issue. Alarms pointed at problems within the systems and across multiple data centers.

- Trying to restore system health by reducing system load in several stages. No impact.



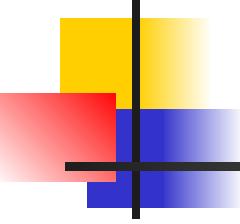
Amazon S3 incident on Sunday, July 20th, 2008

1h01min: engineers detect that servers within Amazon S3 service have problems communicating with each other

- Amazon S3 uses **an epidemic protocol** to spread servers' state info in order to quickly route around failed or unreachable servers
- After, engineers determine that a large number of servers were spending almost all of their time gossiping (i.e., in the epidemic protocol)

1h52min: unable to determine and solve the problem, they decide to shut down all components, clear the system's state, and then reactivate the request processing components.

Restart the system!



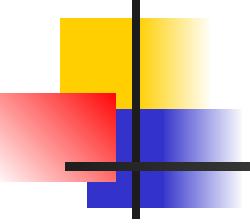
Amazon S3 incident on Sunday, July 20th, 2008

2h29min: the system's state cleared

5h49min: internal communication restored and began reactivating request processing components in the US and EU.

7h37min: EU was ok and US location began to process requests successfully.

8h33min: Request rates and error rates had returned to normal in US.



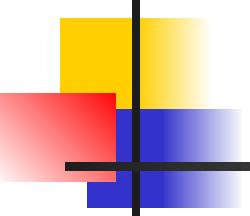
Post-event investigation

Message corruption was the cause of the server-to-server communication problems

Many messages on Sunday morning had a **single bit corrupted**

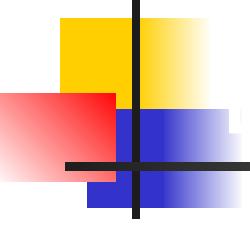
MD5 checksums are generally used in the system, but Amazon did **not** apply them to detect errors in this particular internal state

The **corruption spread** wrong states throughout the system and **increased** the system load



Lessons: Preventing a similar incident

- Verify message and state correctness – all kind of corruption errors may occur
 - Add checksums to detect corruption of system state and messages
 - Verify invariants before processing state
- Engineer protocols to control the amount of messages they generate.
 - Add rate limiters.
 - Put additional monitoring and alarming for gossip rates and failures
- An emergency procedure to restore clear state in your system may be the solution of last resort. Make it work quickly.

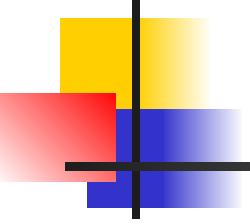


Lessons

You get a big hammer ... use it wisely!

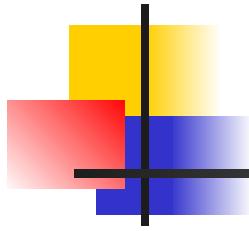
Verify message and state correctness – all kind of corruption errors may occur

An emergency procedure to **restore clear state** in your system may be the solution of last resort. Make it work **quickly**!

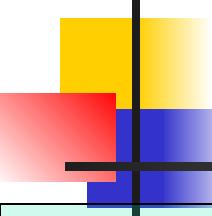


Amazon's the report for the incident
[http://status.aws.amazon.com/
s3-20080720.html](http://status.aws.amazon.com/s3-20080720.html)

Current status for Amazon services [http://
status.aws.amazon.com/](http://status.aws.amazon.com/)



Back to epidemic communication

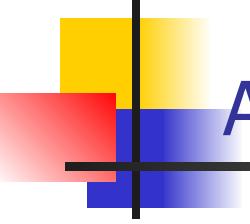


Two main categories:

- **Anti-entropy:**
 - Algorithm works in 'phases'
 - Each node regularly chooses another node at random
 - they exchange state differences (single direction or both directions)
 - **Issue:** When to stop?

[Variation] Gossiping:

- A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).
- Repeat until ...
- **What are the advantages?**



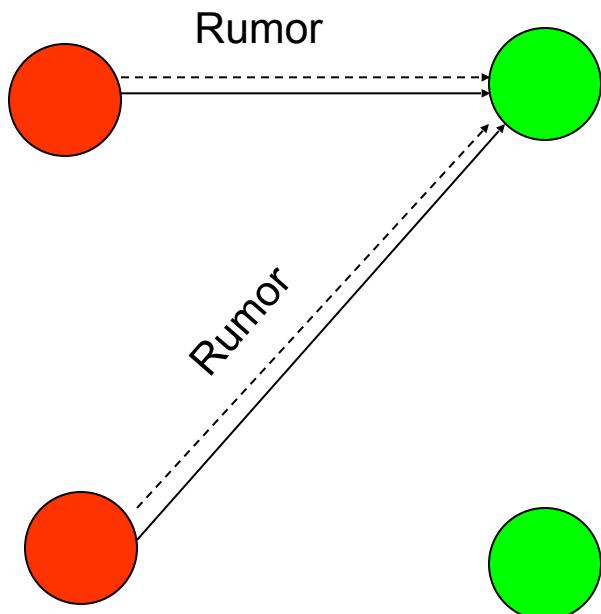
Anti-Entropy Protocols

- A round: each node P selects another node Q at random.

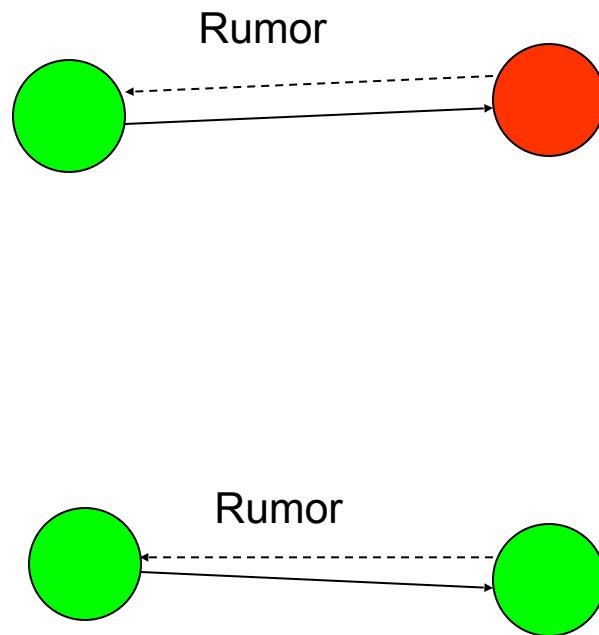
(three variants)
 - **Push**: P only sends its updates to Q
 - **Pull**: P only retrieves updates from Q
 - **Push-Pull**: P and Q exchange mutual updates (after which they hold the same information).

Anti-entropy – Push and Pull

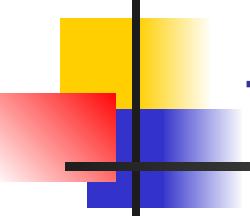
Push



Pull

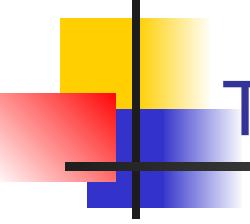


- Susceptible (clean) node
- Infected node



Termination

- A round: each node takes the initiative to start one exchange
 - **Push-Pull:** P and Q exchange mutual updates
- **Termination:** for push-pull it takes $O(\log(N))$ rounds to disseminate updates to all N nodes
- Main properties:
 - **Reliability:** node failures do not impact the protocol
 - **Scalability:** dissemination time & effort (at each individual node) scale well with the number of nodes
 - How much traffic gets generated: Per node? In the network core?

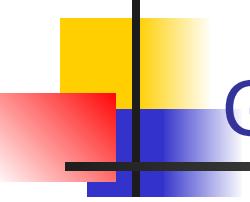


Two main categories:

- **Anti-entropy:**
 - Each node regularly chooses another node at random, and exchanges state differences, leading to identical states at both afterwards
 - Repeat until ...

[Variation] Gossiping:

- A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).
- Repeat until ...



Gossiping

Basic model:

- A node **S** that is 'infected' (i.e., having an update to report), contacts other randomly chosen nodes and 'infects' them
- Newly infected nodes proceed similarly

Gossiping: Termination decision

Basic model: Node **S** that is 'infected', contacts randomly chosen nodes and 'infects' them. Newly infected nodes proceed similarly

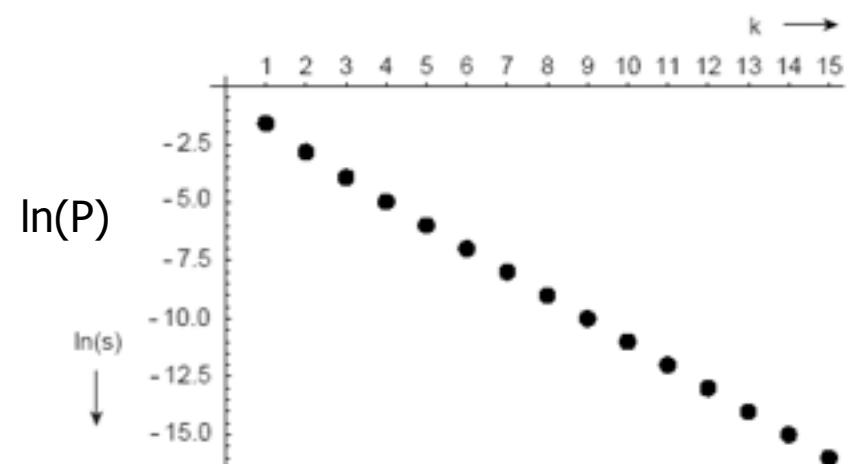
Termination decision:

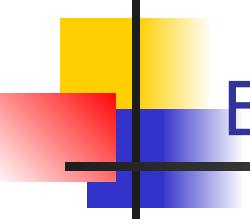
While (true):

- **If** the contacted node already has a already been infected
 - **Then** break (i.e., stop contacting others) with probability $1/k$.

P the share of nodes that
have not been reached
 $P = e^{-(k+1)(1-p)}$

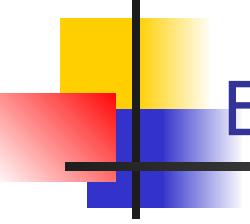
K	P
1	20.0%
2	6.0%
4	0.7%





Example applications (I)

- Updating replicas – distributing updates:
 - E.g., disconnected replicated address book maintenance – Demers et al., Epidemic algorithms for replicated database maintenance. SOSP'87
- Membership protocols:
 - e.g., Amazon Dynamo service: Dynamo: Amazon's Highly Available Key-value Store, SOSP'07
 - Various p2p networks (e.g., Tribler)



Example applications (II)

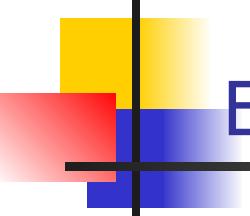
The problem: compute the max value for a large set of sensors

The Solution

- Let every node i maintain a variable x_i . When two nodes engage in the protocol, they each reset their variable to

$$x_i, x_k \leftarrow \max(x_i, x_k)$$

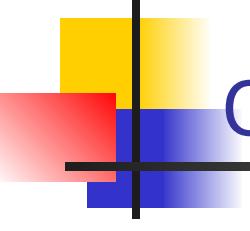
- Repeat for $\log(N)$ rounds
- Result: in the end each node will have the max



Example applications (III)

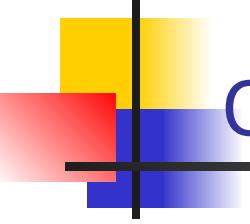
Data aggregation: avg; min; max

The problem: compute the average value for a large set of sensors



Quiz-like questions

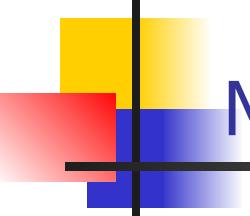
Design an epidemic style protocol to calculate the number of sensors in a sensor network.



Quiz-like questions

Discuss the tradeoffs between a multicast overlay and an epidemic protocol.

- Give motivated examples where one or the other solution may be more appropriate



More quiz-like questions

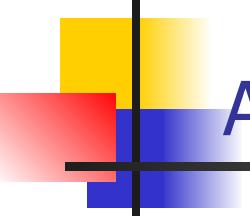
A distributed application operates on a large cluster. The application has one component running on each node.

Cluster nodes might be shut down for maintenance, they might simply fail, or (new) nodes might come (back) online.

To function correctly each application component needs an accurate list of all other nodes/components that are active.

TO DO: Design a mechanism that provides this list

- Describe the mechanism in natural language
- Provide the pseudocode.
- Evaluate overheads



Advantages of epidemic techniques

- Probabilistic model. Rigorous mathematical underpinnings.
Good framework for reasoning about the spread of information through a system over time.
- Asynchronous communication pattern.
Operate in a 'fire-and -forget' mode, where, even if the initial sender fails, surviving nodes will receive the update.
- Autonomous actions.
Enable nodes to take actions based on the data received without the need for additional communication to reach agreement with partners; nodes can take decisions autonomously.
- Robust with respect to message loss & node failures.
Once a message has been received by at least one of your peers it is almost impossible to prevent the spread of the information through the system.



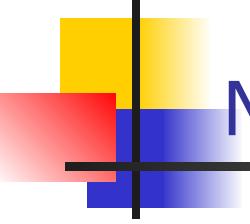
Naming services

A distributed system is:

- a collection of **independent computers** that appears to its users as a **single coherent system**

Components need to:

- Communicate
 - point-to-point communication
 - Sockets, RPC, RMI
 - **point-to-multipoint / data distribution [last time]**
 - **Multicast**
 - **Epidemic communication**
- Cooperate:
 - Naming, Synchronization



Naming / naming services

Names: used to denote **entities** in a (distributed) system.

- To operate on an entity, we need to access it at an access point (**address**).

Note: A **location-independent** name for an entity E, is independent from the addresses of the access points used to access E.

Naming Service **functionality**

- Map: names → access points (addresses)

Names are valuable!

TECHNOLOGY

Sale: Web Address, Unused, Not Cheap

By SABRA CHARTRAND

The Internet address www.flu.com is for sale and its sellers think it is worth \$1.4 million.

The address does not lead to a World Wide Web site, and it never has. Its estimated value seems to arise mostly because such an easy-to-remember destination is available for sale at all.

Because flu.com and virtually all of the other catchy Web addresses ending in ".com" have already been claimed, the value of such unused names has risen sharply. Into this speculative field have come a growing number of companies that have built businesses just by helping owners resell the names they no longer want or need.

But for the best generic and catchy domain names, prices are soaring. Early this year several domain names sold for astronomical amounts — www.business.com for \$7.5 million, www.loan.com for \$3 million and www.beauty.com for \$1 million.

That dynamic is the rise of resellers like GreatDomains.com and Afternic.com, which buy and sell domain names on commission. At GreatDomains.com, where 25 domain names are listed for sale at prices of more than \$1 million each, the company takes 7 to 10 percent of the sales.

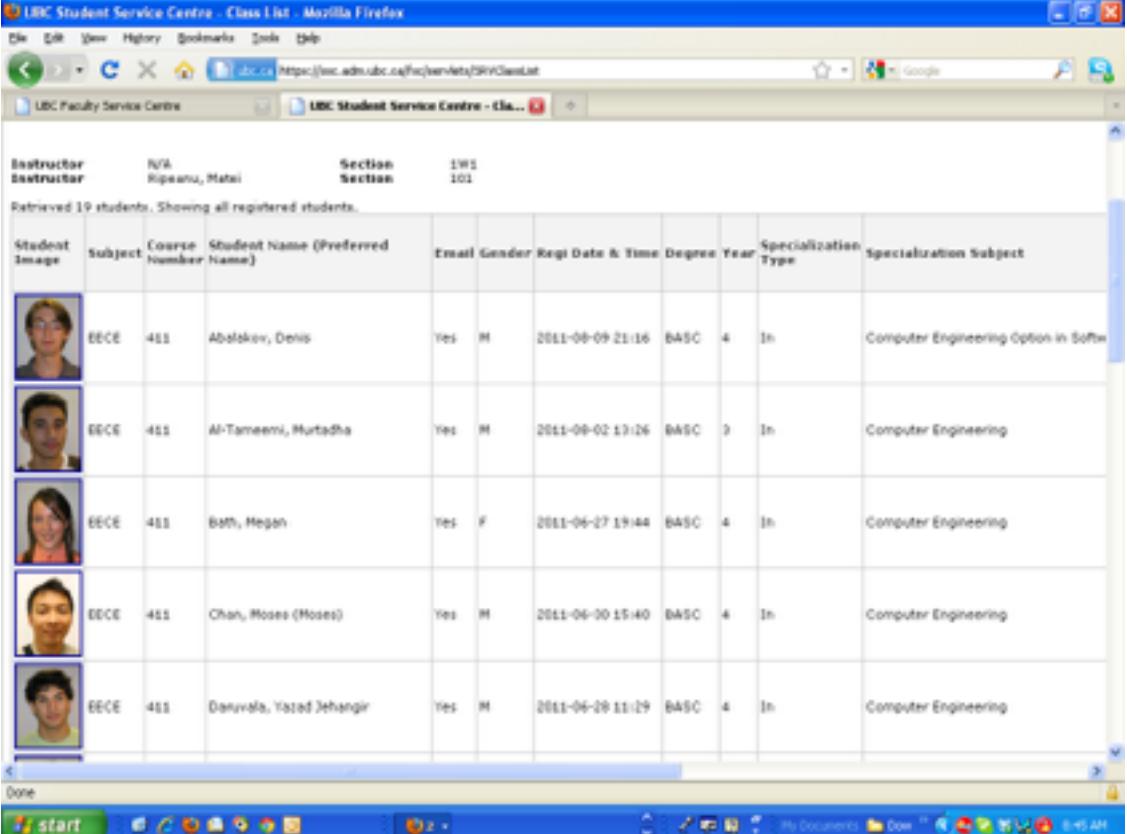
Afternic charges a flat fee of \$1,000 per buyer. GreatDomains says its auction brings in \$10,000 to \$15,000 a day, while the company's closer to \$3,000.



NYT, August'00

War stories (1): Pic database

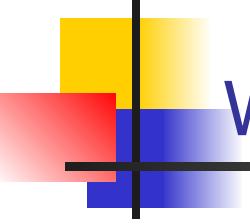
- Saving pics in a student registration database



The screenshot shows a Mozilla Firefox browser window with the title "UBC Student Service Centre - Class List - Mozilla Firefox". The address bar displays the URL <https://sec.adm.ubc.ca/fac/services/SWSClientList>. The main content area shows a table titled "Class List" with the following data:

Instructor	Office	Section	Section								
Ripanu, Matei		IWS	101								
Retrieved 19 students. Showing all registered students.											
Student Image	Subject	Course Number	Student Name (Preferred Name)	Email	Gender	Regi Date & Time	Degree	Year	Specialization Type	Specialization Subject	
	EECE	481	Abduakov, Denis	Denis.Abdakov@ubc.ca	Yes	M	2011-09-09 21:16	BASC	4	In	Computer Engineering Option in Software Engineering
	EECE	481	Al-Tameemi, Murtadha	Murtadha.Al-Tameemi@ubc.ca	Yes	M	2011-09-02 13:26	BASC	3	In	Computer Engineering
	EECE	481	Borth, Megan	Megan.Borth@ubc.ca	Yes	F	2011-06-27 19:44	BASC	4	In	Computer Engineering
	EECE	481	Chan, Moses (Moses)	Moses.Chan@ubc.ca	Yes	M	2011-06-30 15:40	BASC	4	In	Computer Engineering
	EECE	481	Danuvala, Yazeed Jehangir	Yazeed.Danuvala@ubc.ca	Yes	M	2011-06-29 11:29	BASC	4	In	Computer Engineering

- Issues:
 - Naming conflict
 - System vs. user chosen names

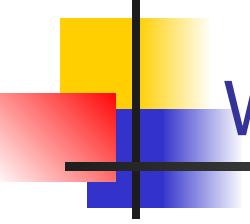


War stories (2): ISPs merge!

Story

- Bob's email is bob@superman.com
- But the ISP running superman.com is bought by superwoman.com and they want to translate all email addresses to their new domain
 - bob@superwoman.com ☺
- Issues:
 - Overloading (address is not location independent!)
 - bob@superman.com → bob99@superwoman.com

Solution: indirection service



War stories (3): ZIP codes – more overloading

US zip code structure: routing built into name.

1st digit: zone (e.g., New England, NW)

2nd-3rd digit: 'section'

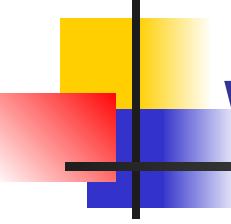
4-5th digit: post-office

Story: Congestion at Boston section (021).

- Solution adopted: split in two (021 and 024)
- Result?

Issue:

- Overloading, address not location independent



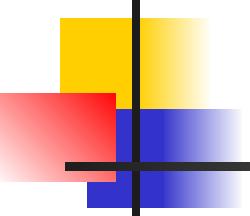
War stories (4): Running out of phone numbers

Phone numbers

- 10 digit: area(3)+switch(3)+identifier(4)

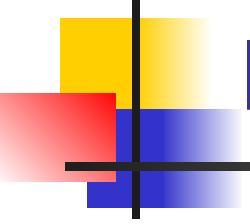
Story: Running out of numbers

Issue: Splitting vs. overlay



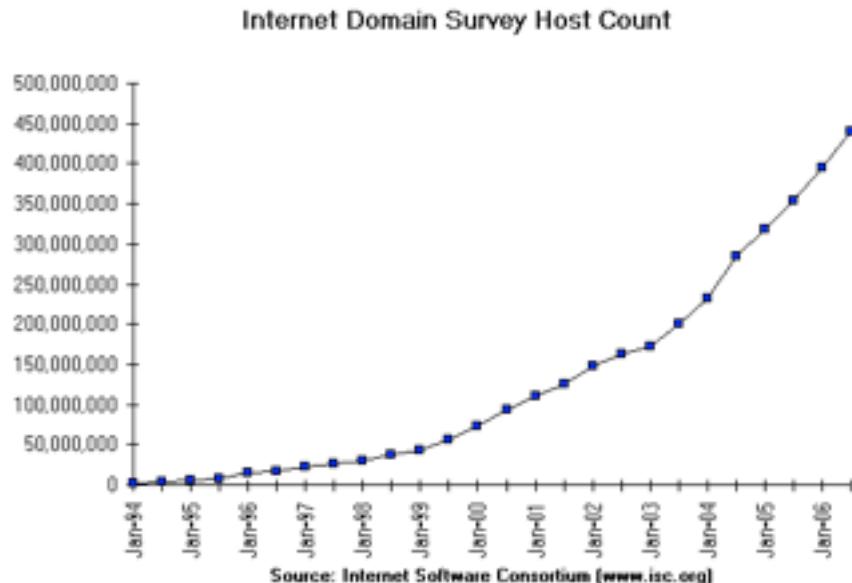
More terminology (and design choices!)

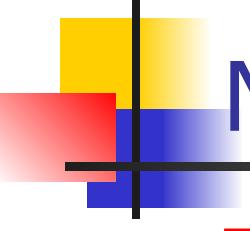
- Names vs. identifiers
 - Identifiers have three properties
 - refer to at most one entry
 - each entity is referred by at most one identifier
 - always refers to the same entity
- Human friendly vs. arbitrary names (random strings)
- Namespace
 - Flat (names have no structure), vs.
 - Hierarchical (names have structure)
 - Structure can be used for partitioning / routing



Naming system implementation

- Functionality
 - Map: names → access points (addresses)
- Key challenge: **scaling**
 - #of names,
 - #clients,
 - geographical distribution,
 - Management!





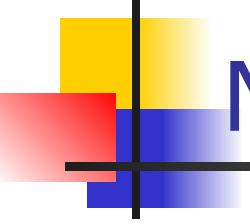
Naming system implementation

Functionality

- Map: names → access points (addresses)

Strawman #1: Why not centralize?

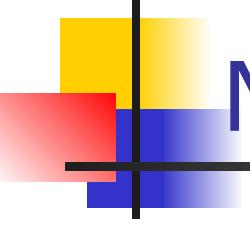
- Single point of failure
- High latency
 - Distant centralized database
- Scalability bottleneck:
 - Traffic volume
 - Management: Single point of update



Naming system implementation

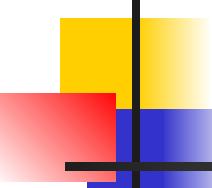
Strawman #2: Why not use a replicated database (old /etc/hosts)?

- Original Name-to-Address Mapping service
 - Flat namespace
 - /etc/hosts
 - SRI kept master copy
 - Everyone downloaded regularly
- Count of hosts was increasing: as Internet moved from machine per domain → machine per user
 - Many more downloads
 - Many more updates
- A scalability bottleneck



Naming system implementation

Strawman #3:.... ?



Naming system implementation

Strawman N+1 Broadcast based solution:

- Each participant keeps its own (name, address) mapping produced locally.
- All participants to listen to incoming requests
- Search function: Simply broadcast the name
 - Requests the entity that holds it to return the address.

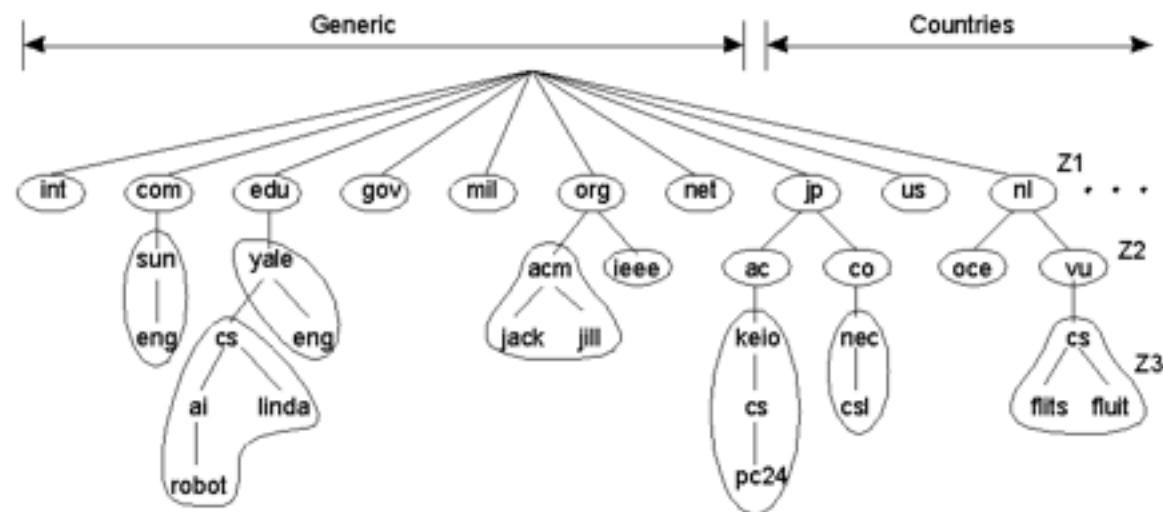
Properties:

- Flat namespace
- Does not scale beyond local-area networks
 - (think of ARP/RARP)

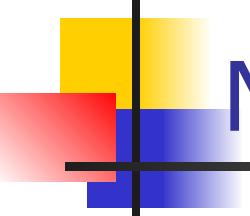
Scaling naming system implementation

Idea: partition the namespace

- Hierarchical namespace (e.g., DNS)
 - Implies some name structure



What if I want to keep the namespace flat?



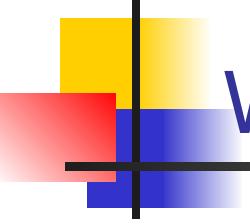
Naming system implementation

Problem: Design a system that is given an essentially **unstructured name** and is able to locate its associated **address**?

Note: This is similar to managing a hash-table:

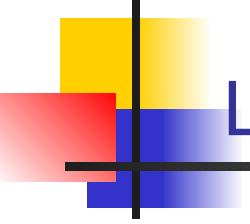
- manage list of (name, access point) pairs
- API: put (key, value), lookup (key) → value

Issue: scaling

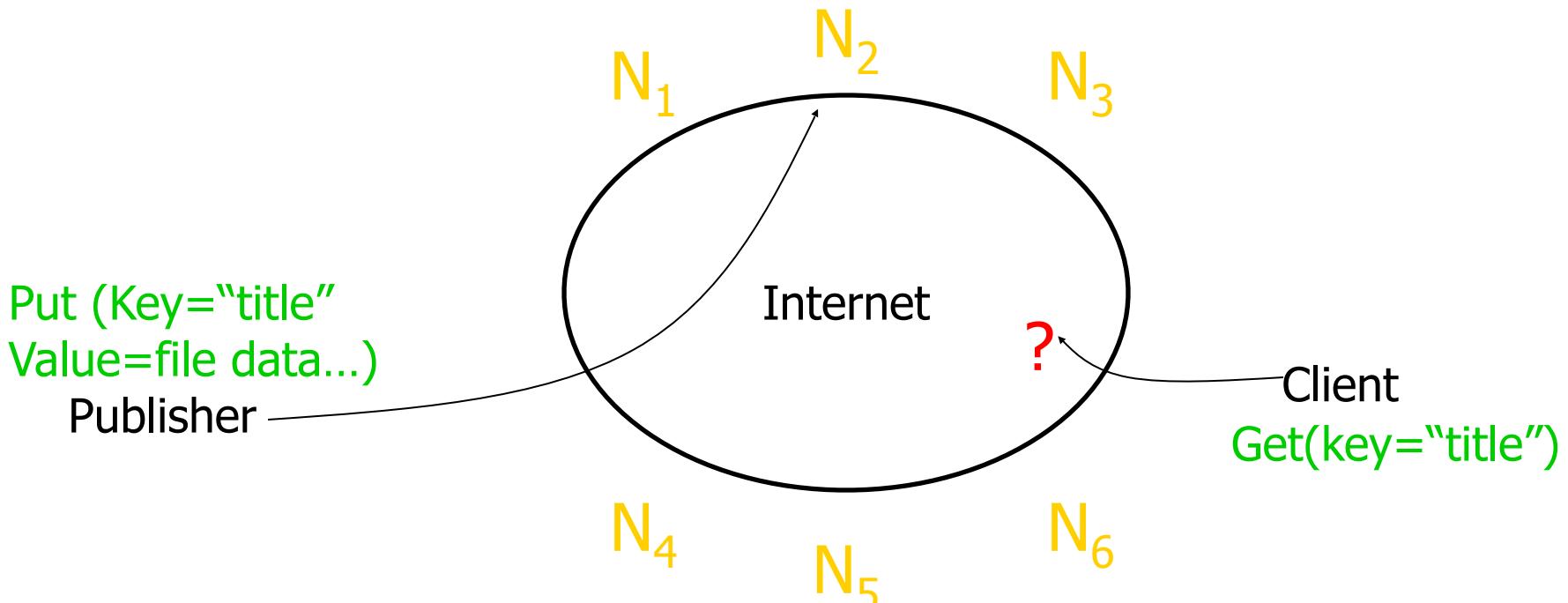


Why the put()/get() interface?

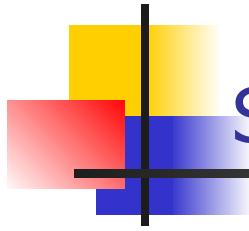
- API supports a wide range of applications
 - imposes no structure/meaning on keys
- key/value pairs are persistent and global
 - Can store keys in other values (**indirection**)
 - and thus build complex data structures



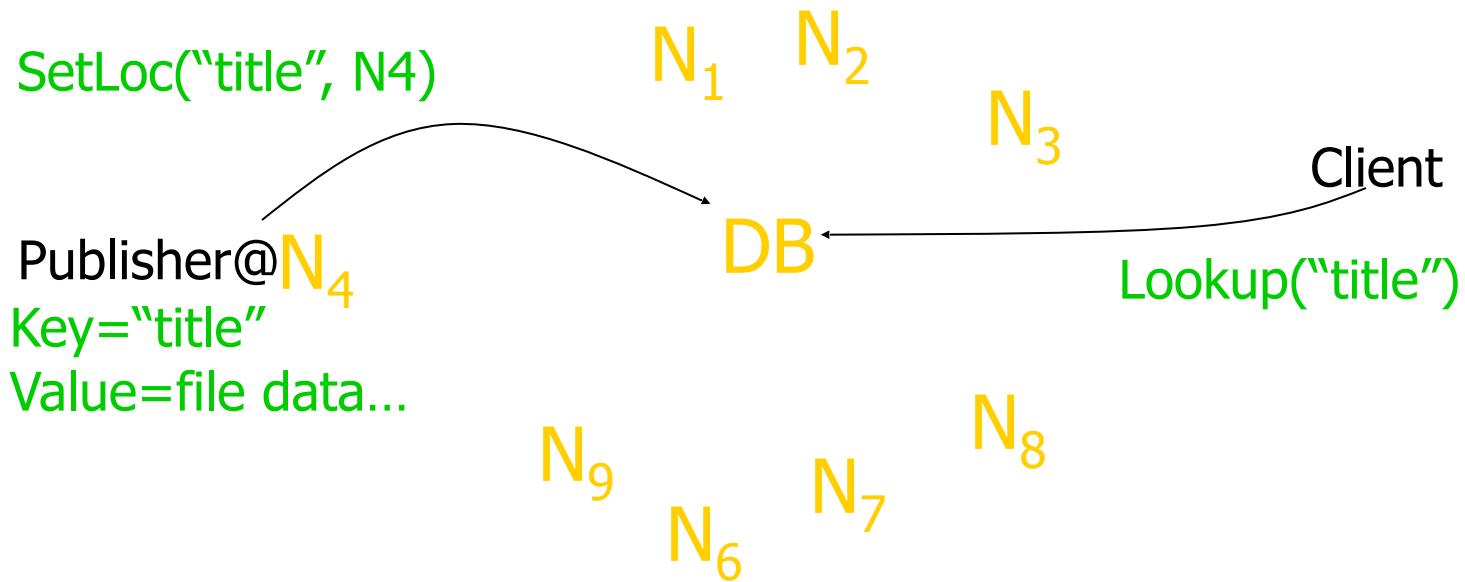
Let's recap: The Lookup Problem



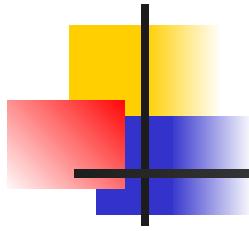
- At the heart of all these services



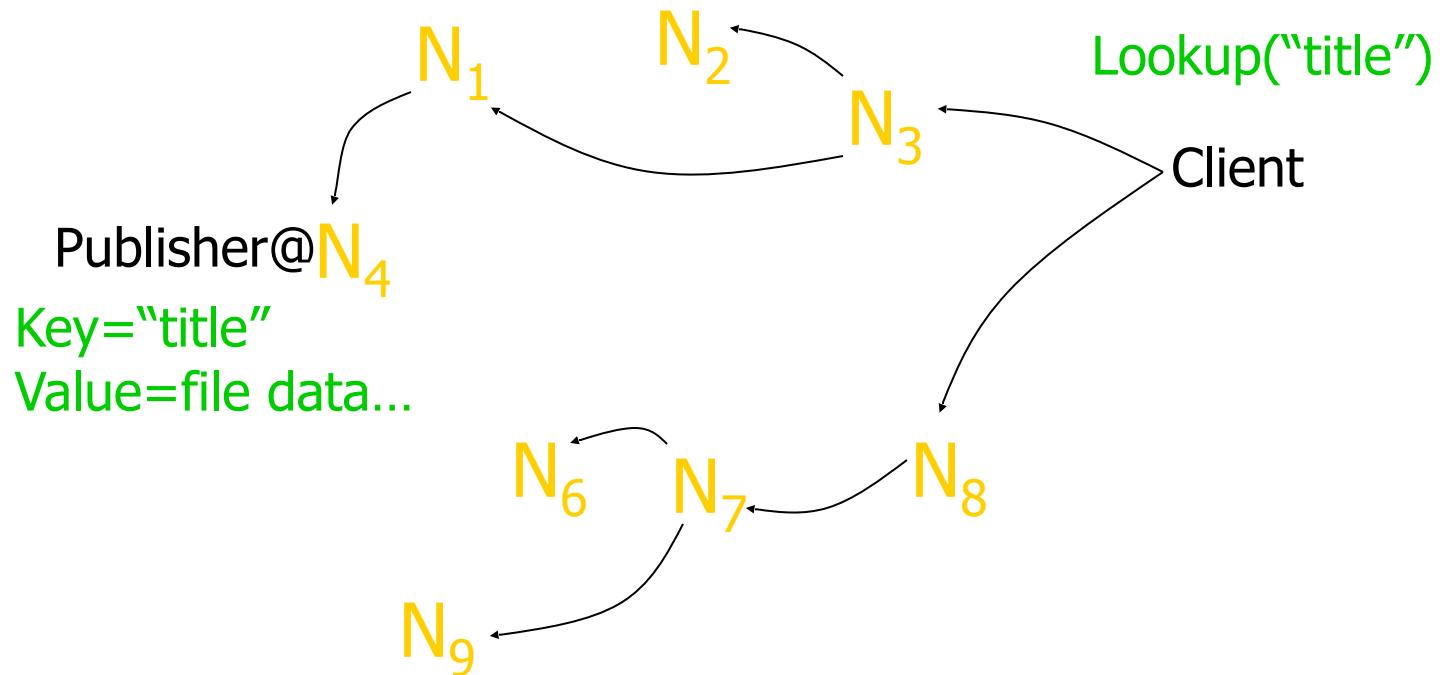
S1: Centralized Lookup (Napster)



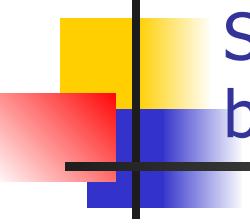
Simple, but $O(N)$ state and a single point of failure



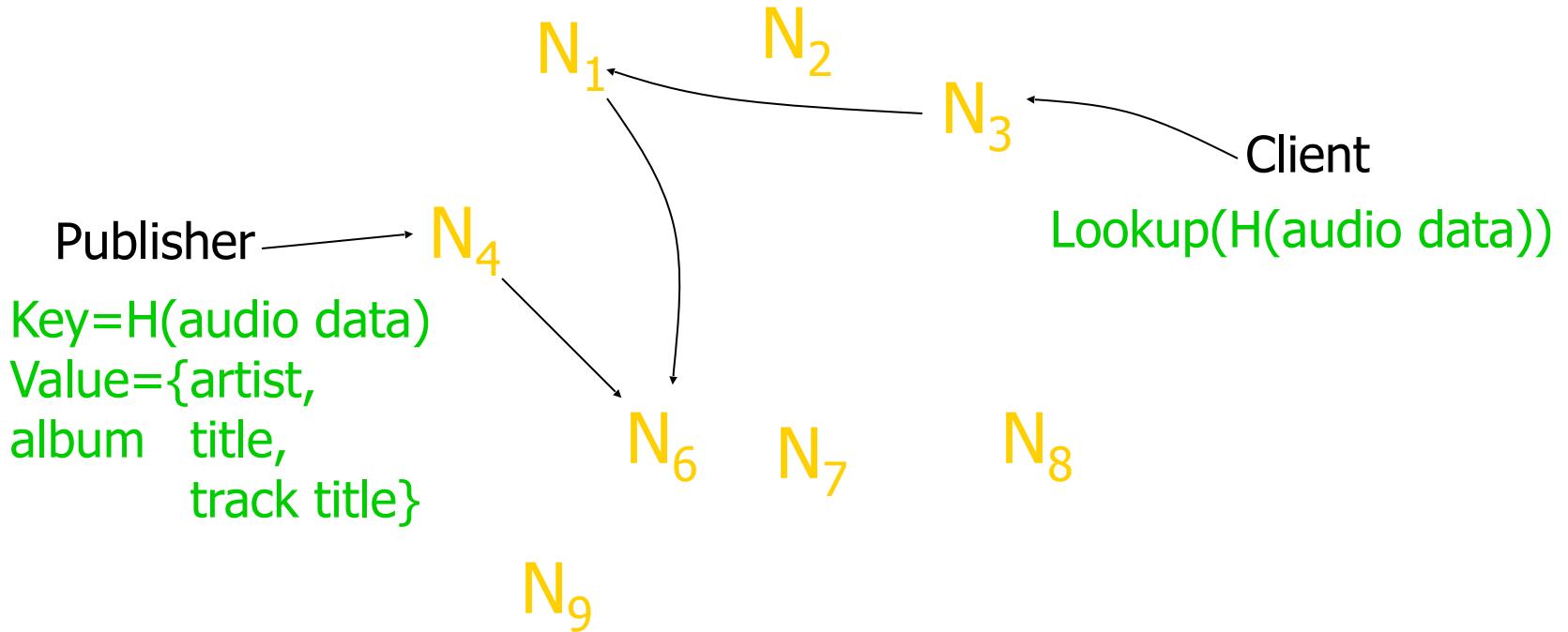
S2: Flooded Queries (Gnutella)



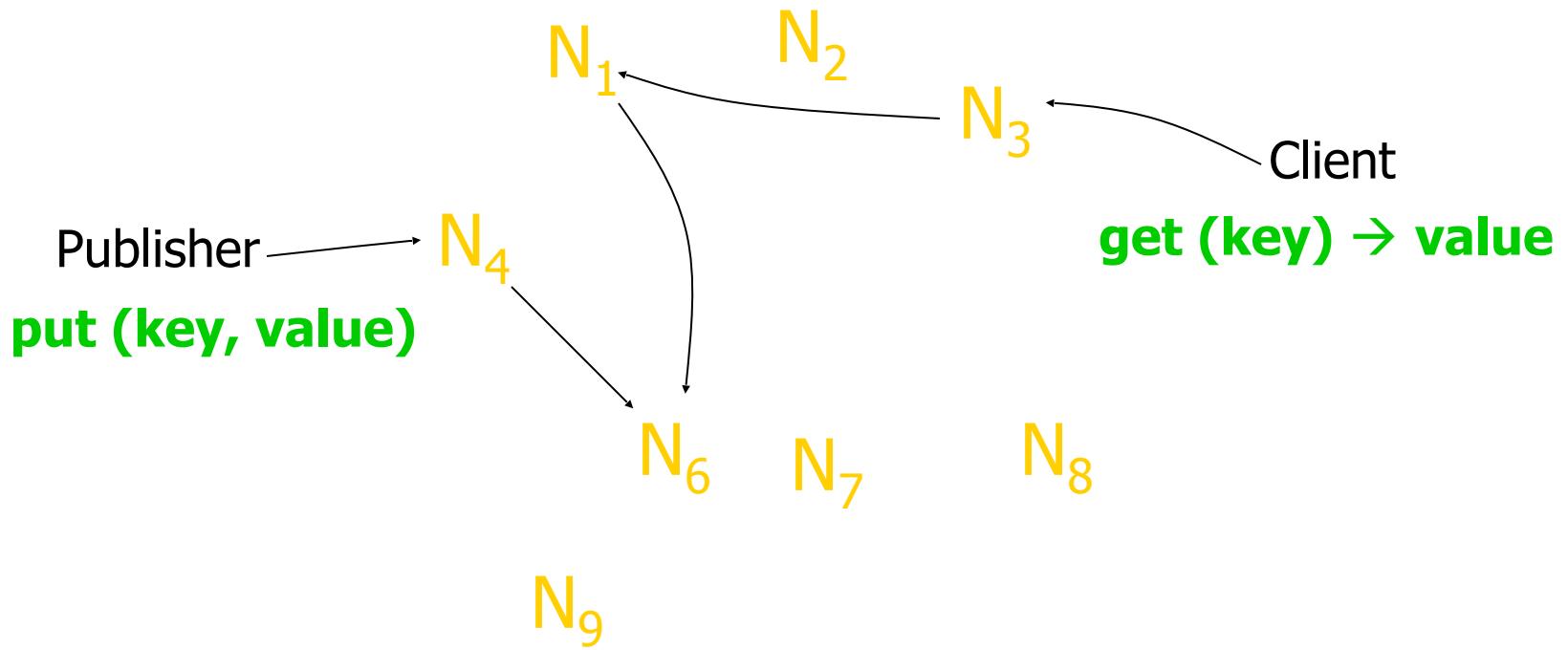
Robust, but worst case $O(N)$ messages per lookup

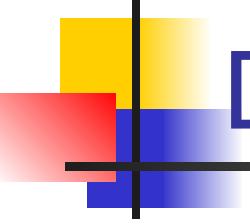


S3: Rendez-vous points between publisher and client



Challenges: Make it robust. Small state. Actually find stuff in a changing system.





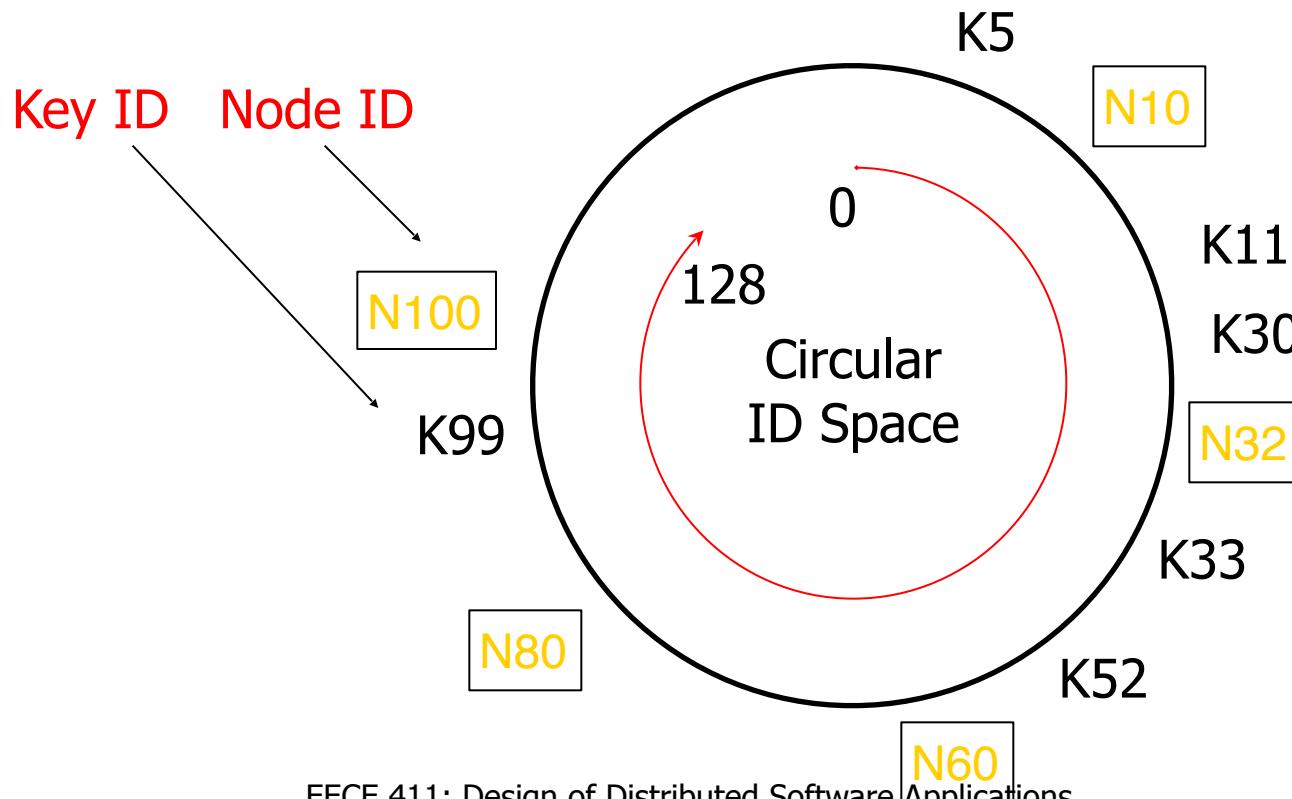
Desirable Properties

- Decentralized: no central authority
- Scalable: multiple axes
 - network traffic overhead, state at nodes, routing cost, ...
- Incremental scalability
- Efficient: find items quickly (latency)
- Dynamic: deals with node failure, join
- General-purpose: flat naming

- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node/join failures?
- Issue 4: How to deal with node heterogeneity
- Issue 5: ...

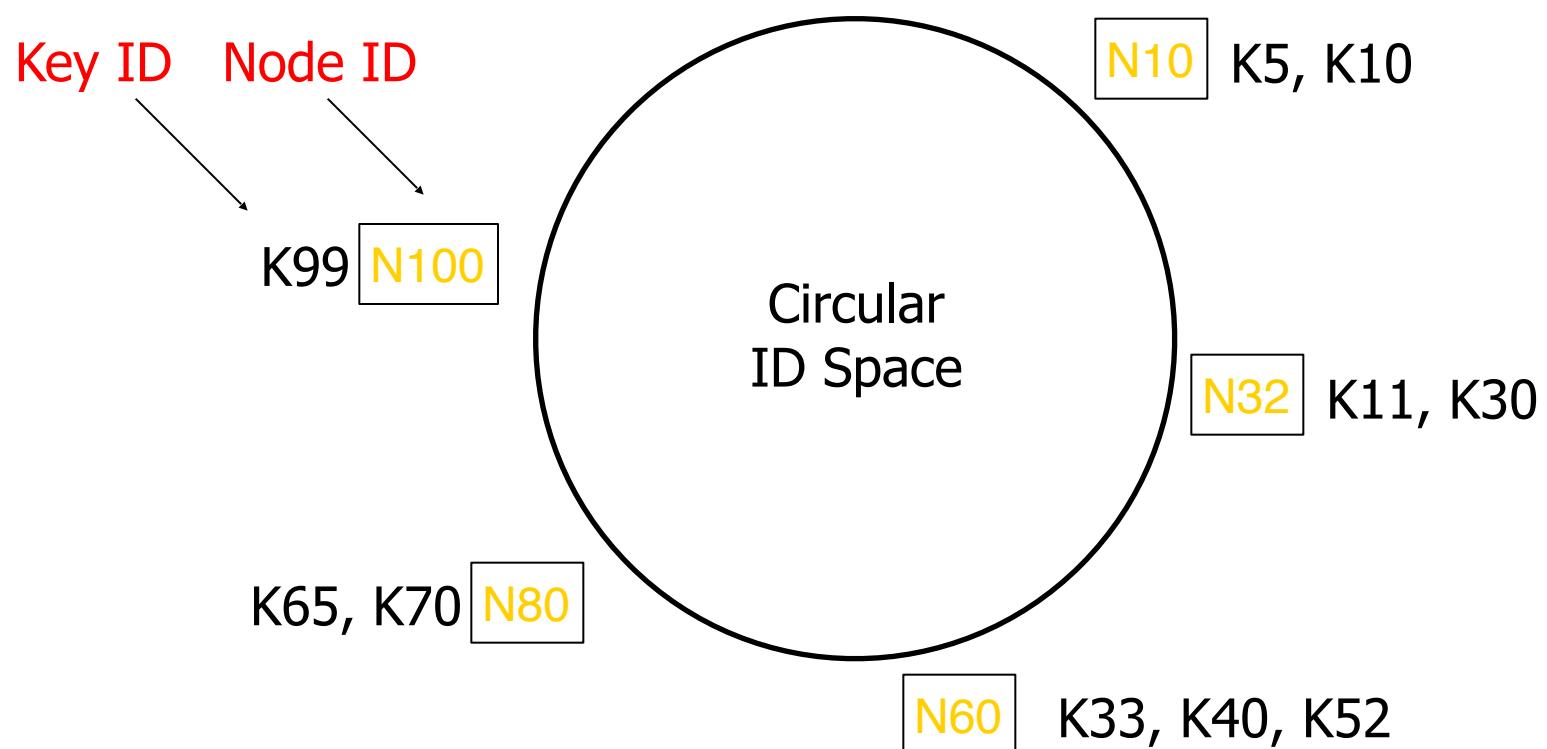
Partition Solution: Consistent hashing & DHT

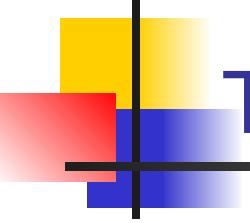
- The output range of a hash function is treated as a fixed circular space or “ring”.



Partition Solution

- Mapping keys to nodes
- Advantages: **incremental scalability, load balancing**





Theoretical foundation

Notation:

- **N** number of nodes,
- **k** number of keys in the system

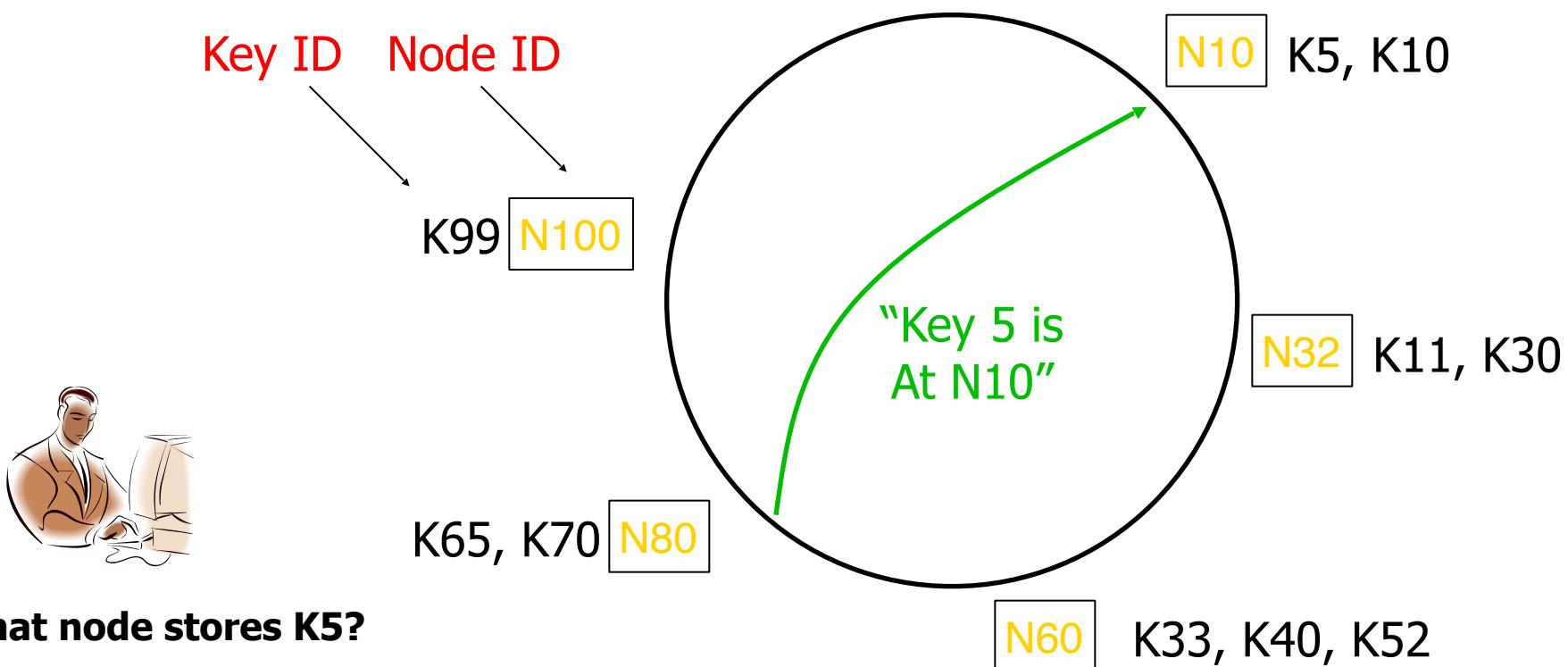
Theorem

- [With high probability] Each node is responsible for at most $(1+\varepsilon)K/N$ keys
 - → Resulting property: Load balancing
 - (if similar nodes)

- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node join failures?
- Issue 4: How to deal with node heterogeneity
- Issue 5: ...

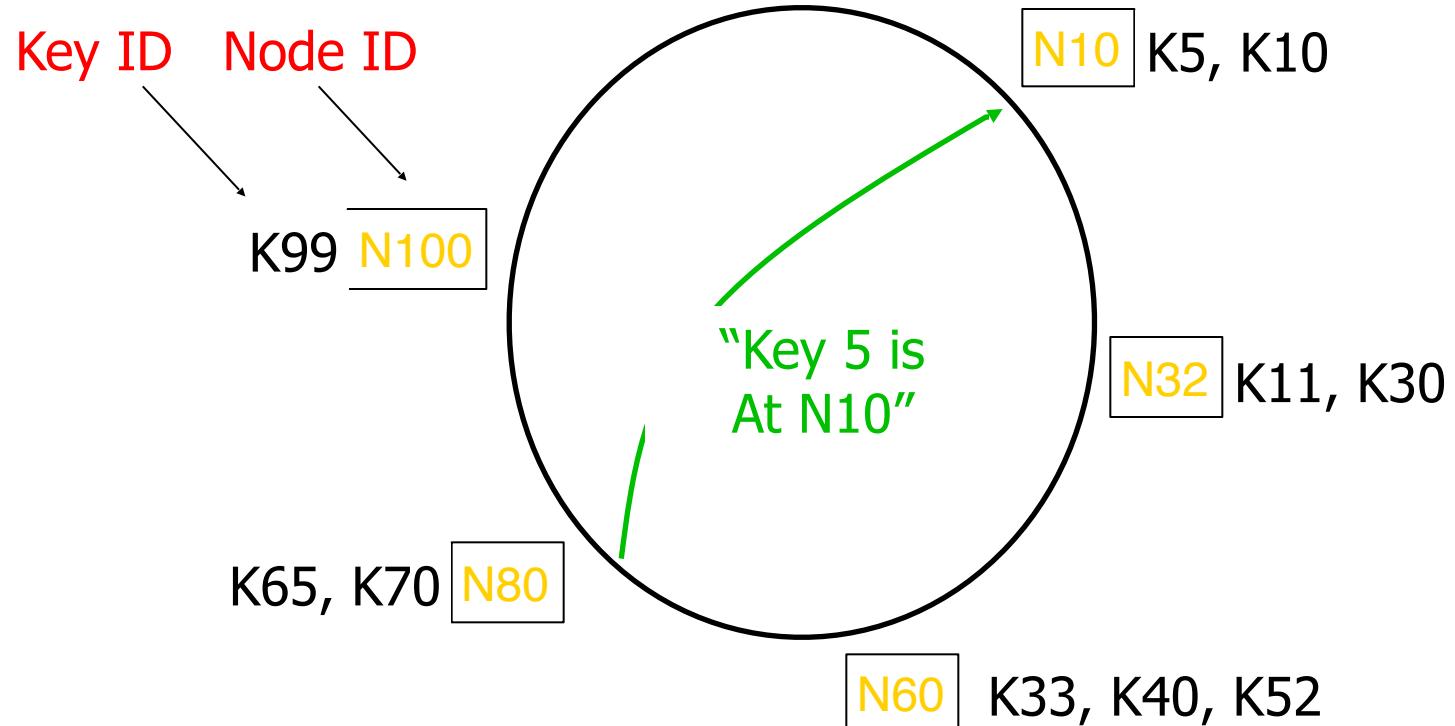
Consistent hashing

- Direct routing.
- Lookup: $O(1)$
- State at each node: $O(N)$

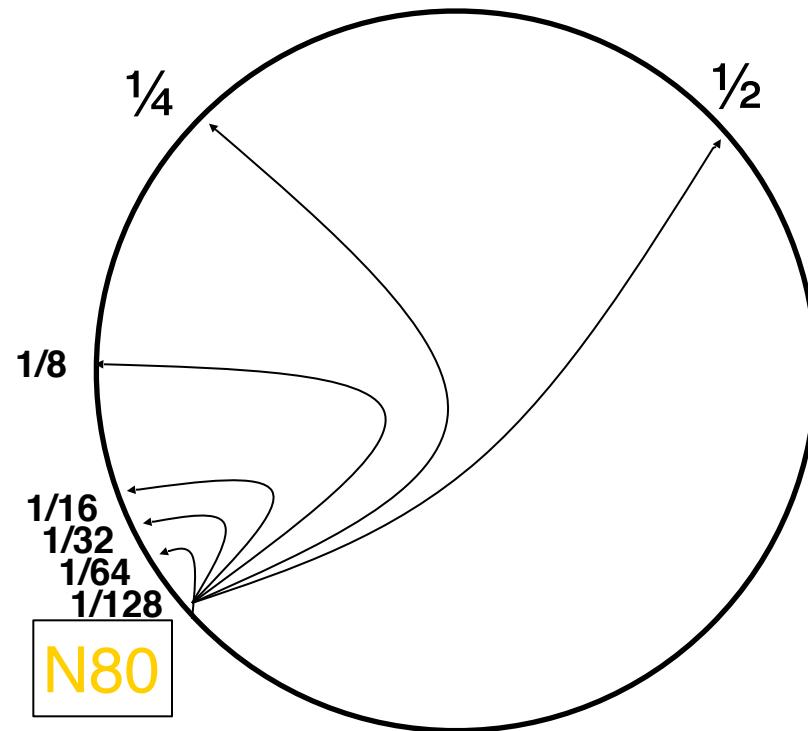


BUT ... Large is the state at each node
O(N); N number of nodes.

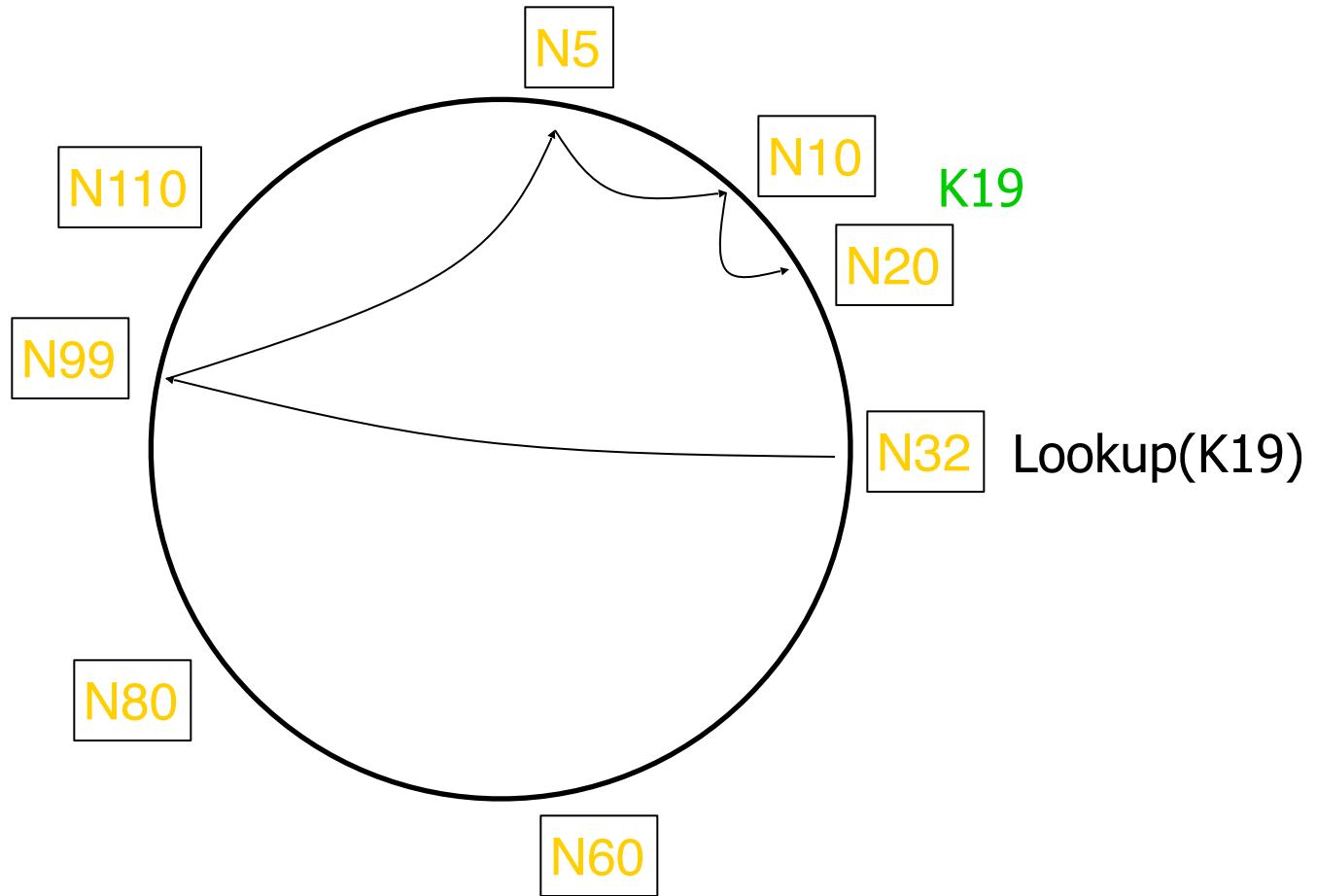
Can we do better?

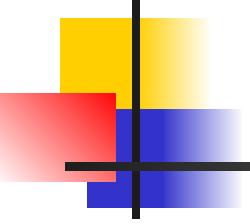


“Finger Table” Accelerates Lookups



- Lookup: $O(\log N)$
- State at each node: $O(\log N)$



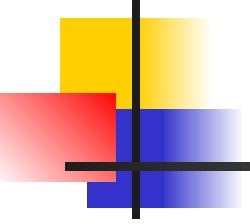


Consistent Hashing

- Lookup: $O(1)$
- Routing state per node: $O(N)$

Distributed Hash Table

- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
 - [maintain routing structure, data placement]
- Issue 4: How to deal with node failures?
- Issue 5: How to deal with node heterogeneity
- Issue 6: ...



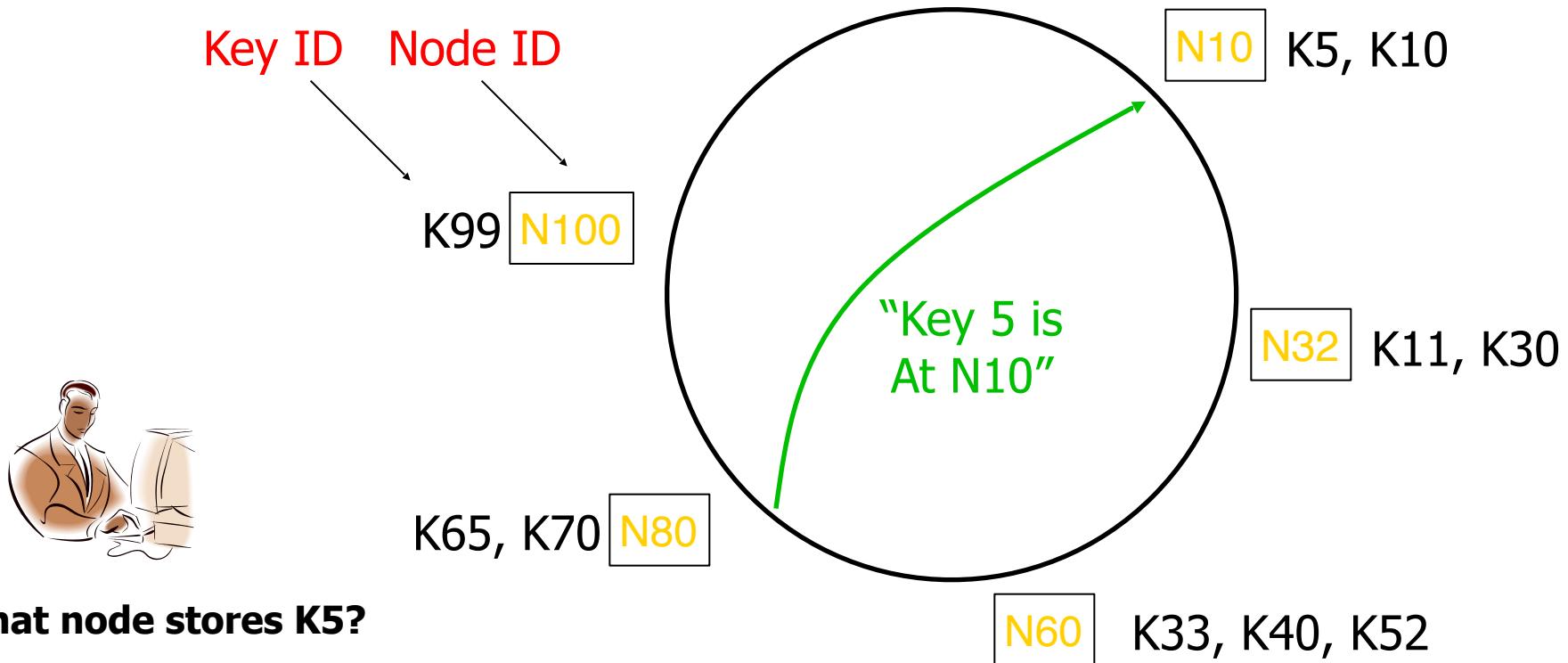
Consistent Hashing

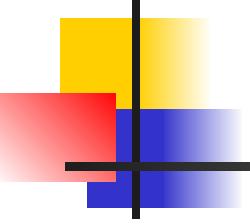
Distributed Hash Table

- | | | |
|---------------------------|--------|-------------|
| ▪ Lookup: | $O(1)$ | $O(\log N)$ |
| ▪ Routing state per node: | $O(N)$ | $O(\log N)$ |
| ▪ Node joins | ?? | |

Consistent hashing

What info to maintain to route correctly?





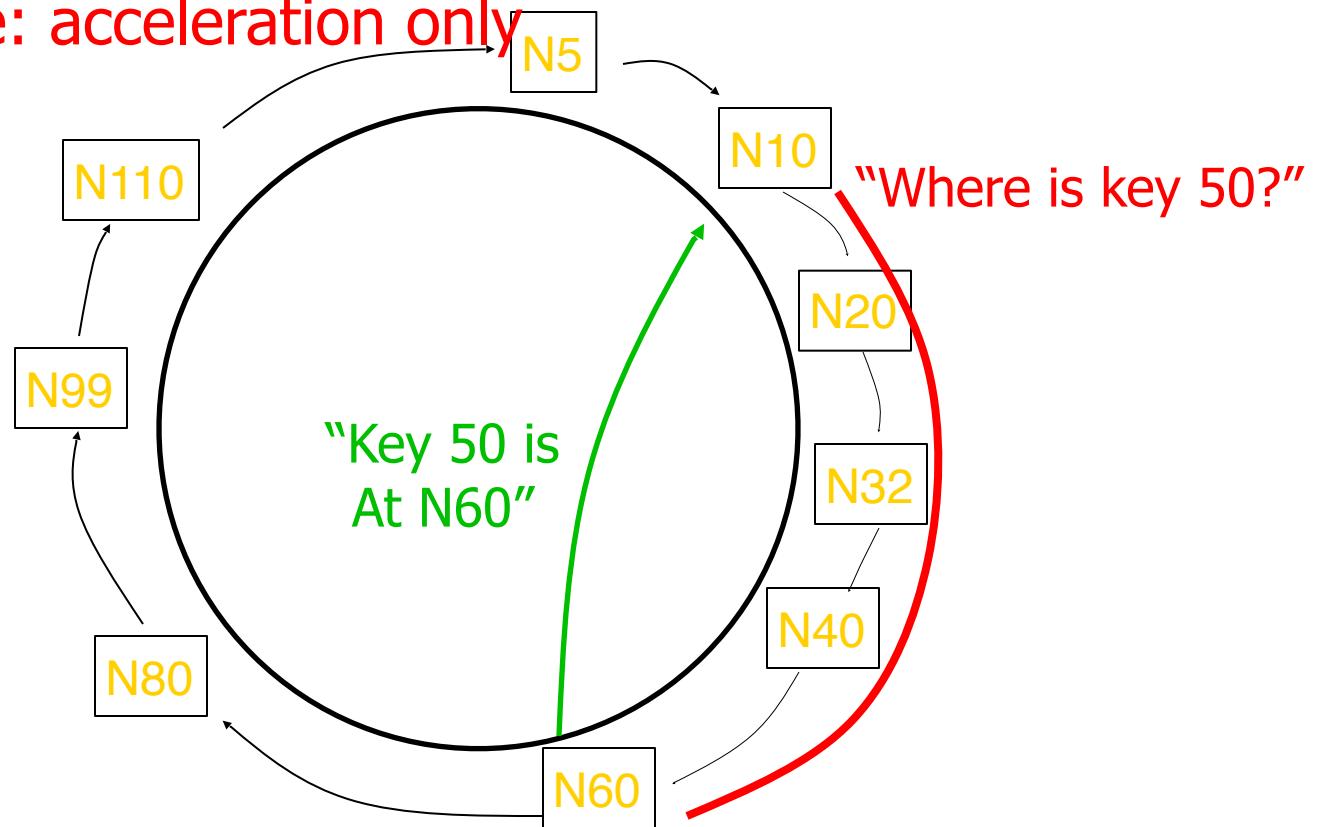
Consistent Hashing

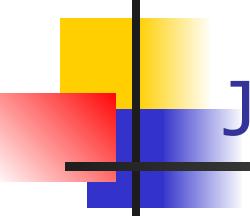
Distributed Hash Table

- | | | |
|---------------------------|--------|-------------|
| ▪ Lookup: | $O(1)$ | $O(\log N)$ |
| ▪ Routing state per node: | $O(N)$ | $O(\log N)$ |
| ▪ Node joins | | ??? |

DHT: What info to maintain to route correctly?

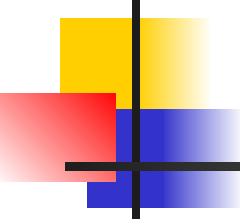
- (at a minimum) Lookups correct if each node can maintain its successor.
- Finger table: acceleration only



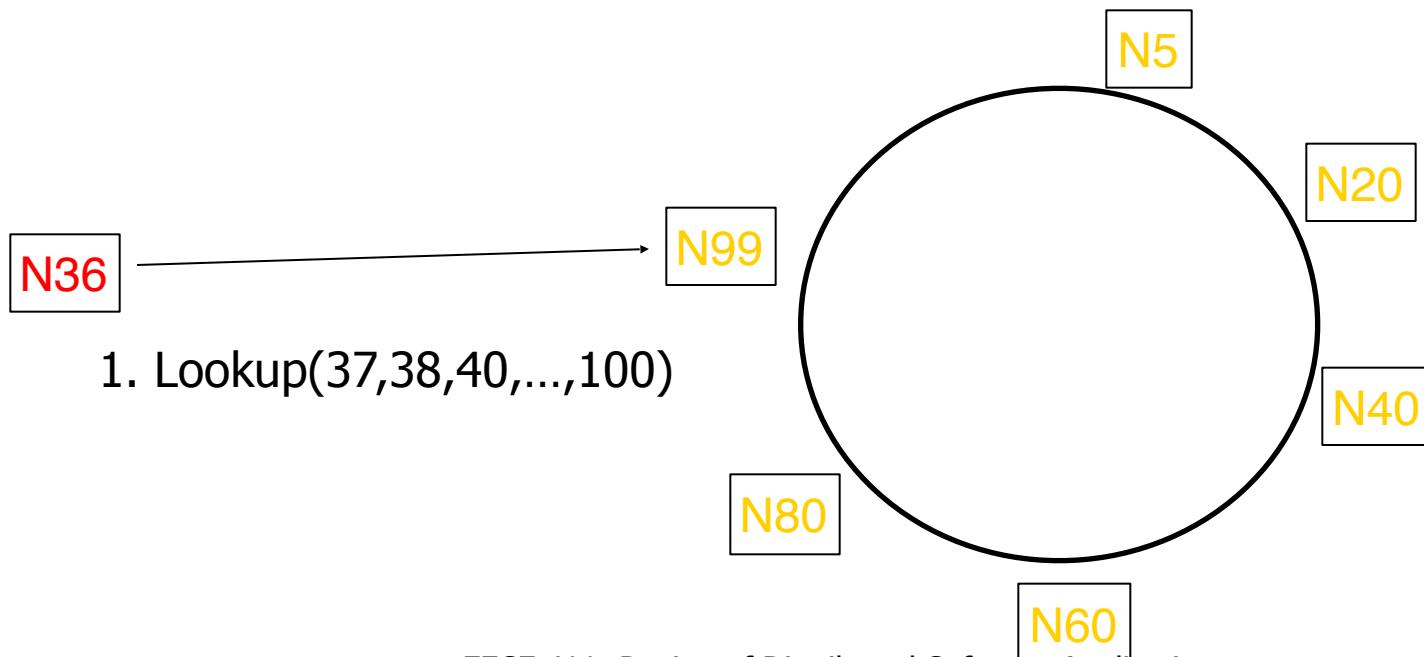


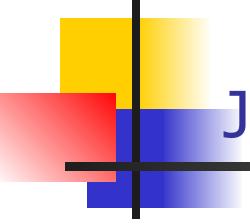
Joining the Ring: DHT

- Three step process
 - Initialize all 'fingers'/shortcuts of new node
 - Update fingers of existing nodes
 - (Transfer keys from successor to new node)
- Two invariants to maintain to insure correctness
 - Each node's successor list is maintained
 - successor(k) is responsible for monitoring k



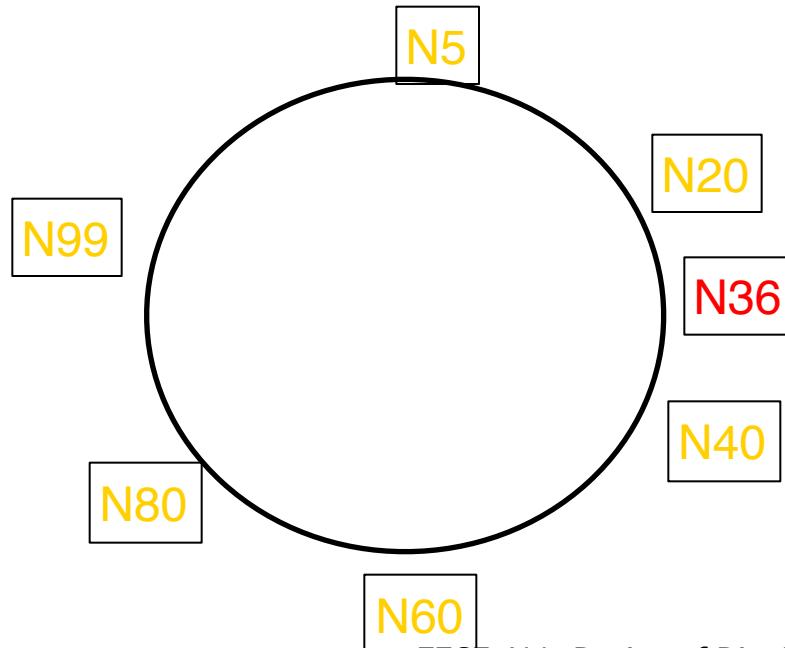
Join: Initialize New Node's Finger Table

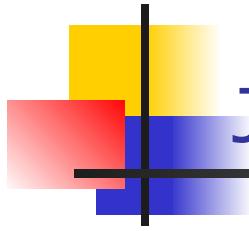




Join: Update Fingers of Existing Nodes

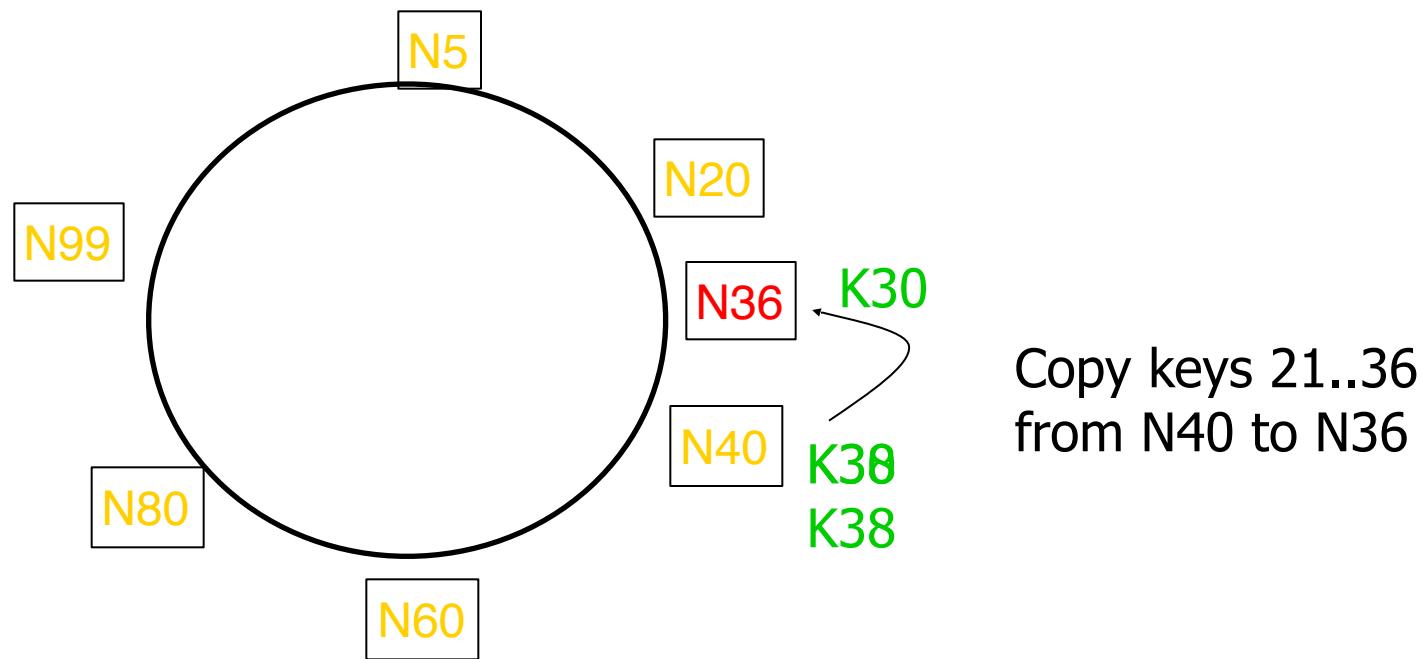
- New node calls update function on existing nodes
- Existing nodes recursively update fingers of other nodes



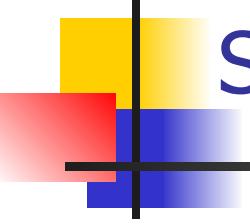


Join: Transfer Keys

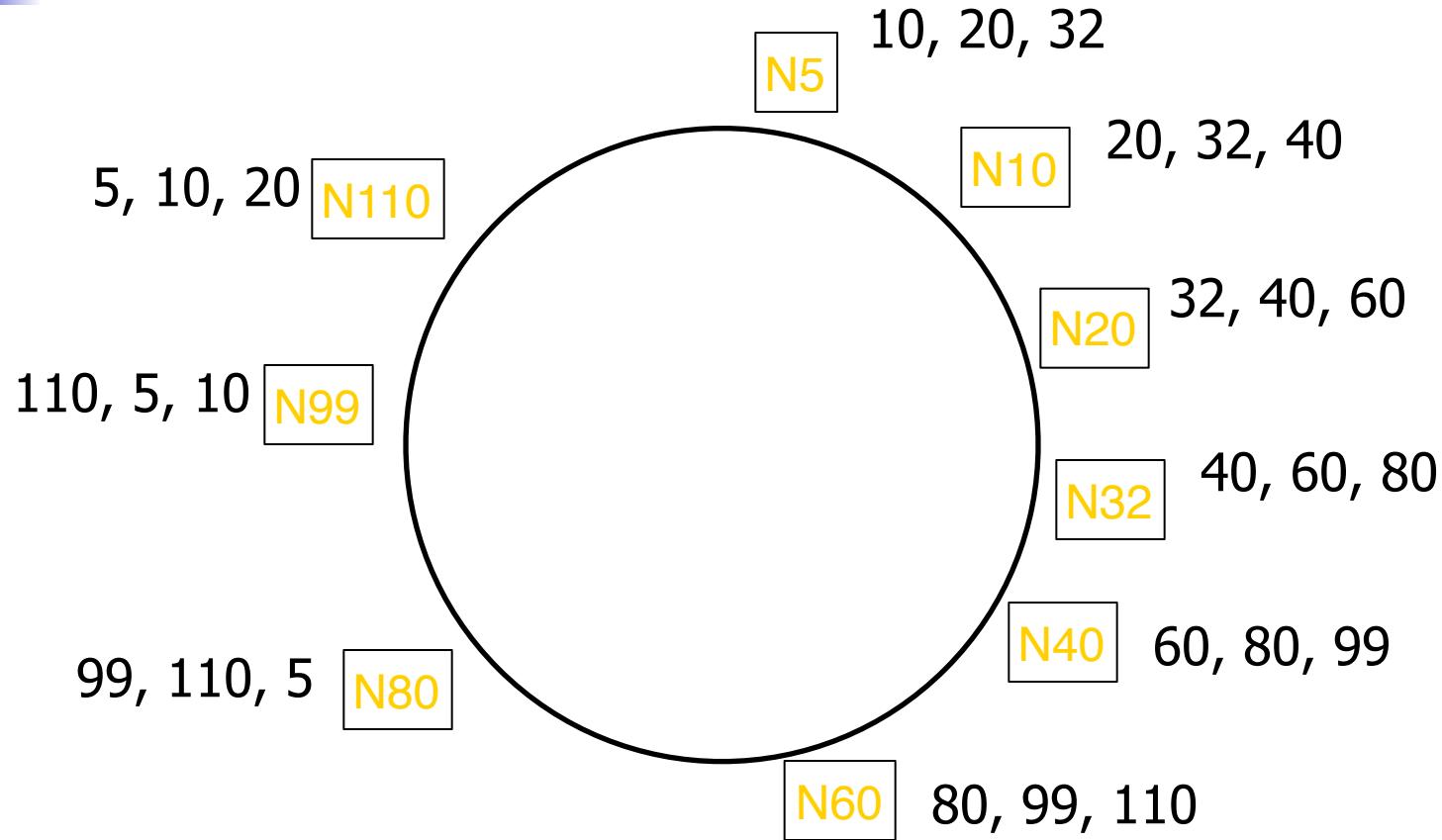
- Only keys in the range are transferred



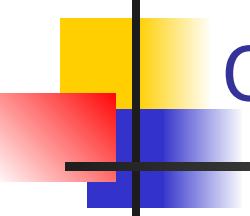
- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
- Issue 4: How to deal with node failures?
 - To maintain routing structure
 - To maintain data
- Issue 5: How to deal with node heterogeneity
- Issue 6: ...



Successor Lists Ensure Robust Lookup



- Each node remembers r successors
- Lookup can skip over dead nodes



Quiz like question

How to dimension the successor list?

(e.g., such that the chance the whole DHT fails if half of the nodes fail is 0.001).

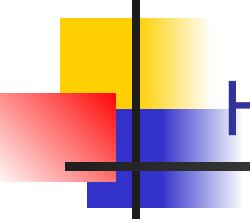
Notation: r - length of successor list

N – nodes in the system

- $P(\text{all successors of a specific node dead}) = (1/2)^r$
 - i.e., $P(\text{this node breaks the ring})$
 - independent failure assumption
- $P(\text{system works}) = P(\text{no broken node in the whole system})$
$$= (1 - (1/2)^r)^N$$

with $r = \log_2(N)$ makes prob. = $1 - 1/N$

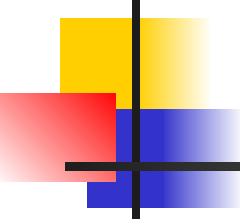
- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- **Issue 3: How to deal with node failures?**
 - To maintain routing structure
 - **To maintain data**
- Issue 4: How to deal with node heterogeneity
- Issue 5: ...



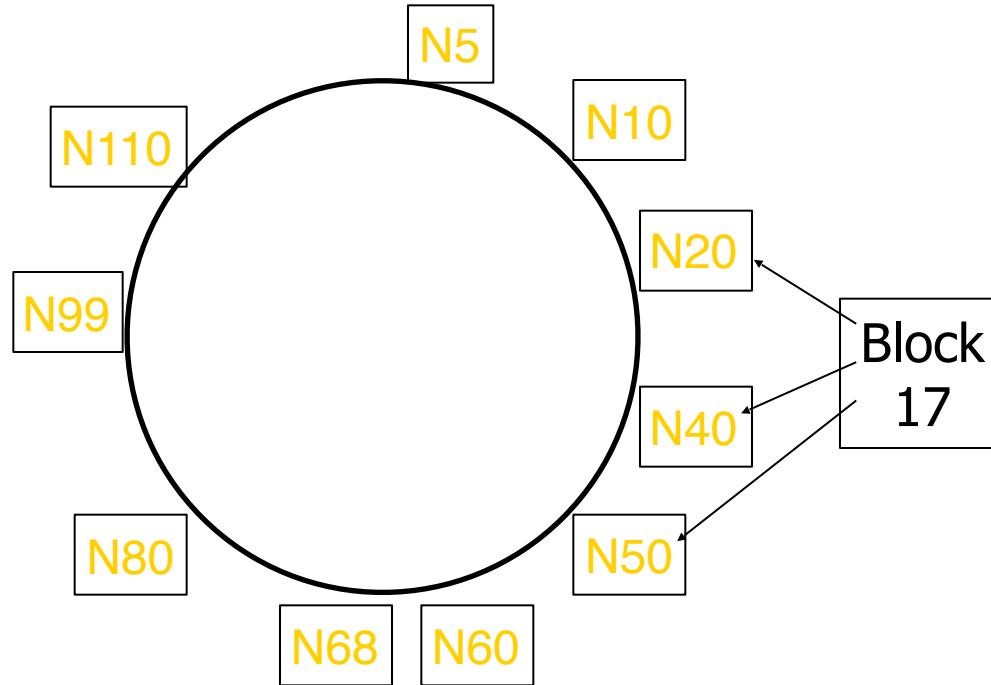
How to maintain data?

- **Nodes failure:** replicate data
 - Pick an uniform choice of nodes to do replication
 - E.g., replicate to R successors
- **Node joins:** migrate data
 - (Lazily) delete unnecessary replicas

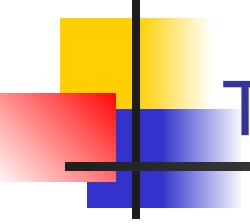
The same solutions work for consistent hashing and DHT



Replicates Blocks at r Successors



- Replicas are easy to find if successor fails
- Hashed node IDs ensure independent failure



Theoretical foundation

Notation:

- N number of nodes,
- k number of keys in the system

Theorems

- [With high probability] Each node is responsible for at most $(1+\varepsilon)K/N$ keys
 - → Load balancing
- [With high probability] A node joining or leaving relocates $O(K/N)$ keys (and only to or from the responsible node)
 - → Local impact of failures

- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node join failures?
 - To maintain routing structure
 - **To maintain data**
- Issue 4: How to deal with node heterogeneity?
- Issue 5: ...

Problem: How to load balance when nodes are **heterogeneous**?

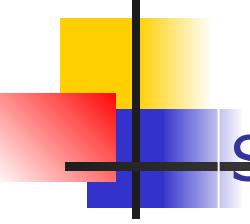
Solution idea: Each node owns an ID space proportional to its 'power'

Virtual Nodes:

- Each physical node is responsible for multiple (similar) virtual nodes.
- Virtual nodes are treated the same

Advantages: load balancing, incremental scalability,

- Dealing with heterogeneity: The number of virtual nodes that a node is responsible for can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.
- When a node joins (if it supports many VN) it accepts a roughly equivalent amount of load from each of the other existing nodes.
- If a node becomes unavailable the load handled by this node is evenly dispersed across the remaining available nodes.

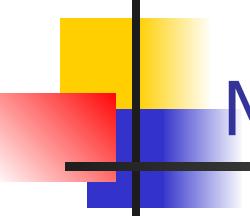


Sample quiz question:

What are the criteria to choose between a system based on a DHT and one based on consistent hashing?

Where are the differences?

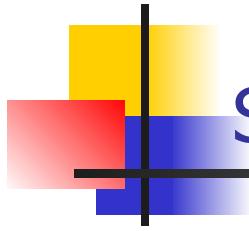
- Key to node assignment?
- Load balancing
- Lookup?
- Information used for lookup?
- Impact of failures?
- Ability to scale?



More issues

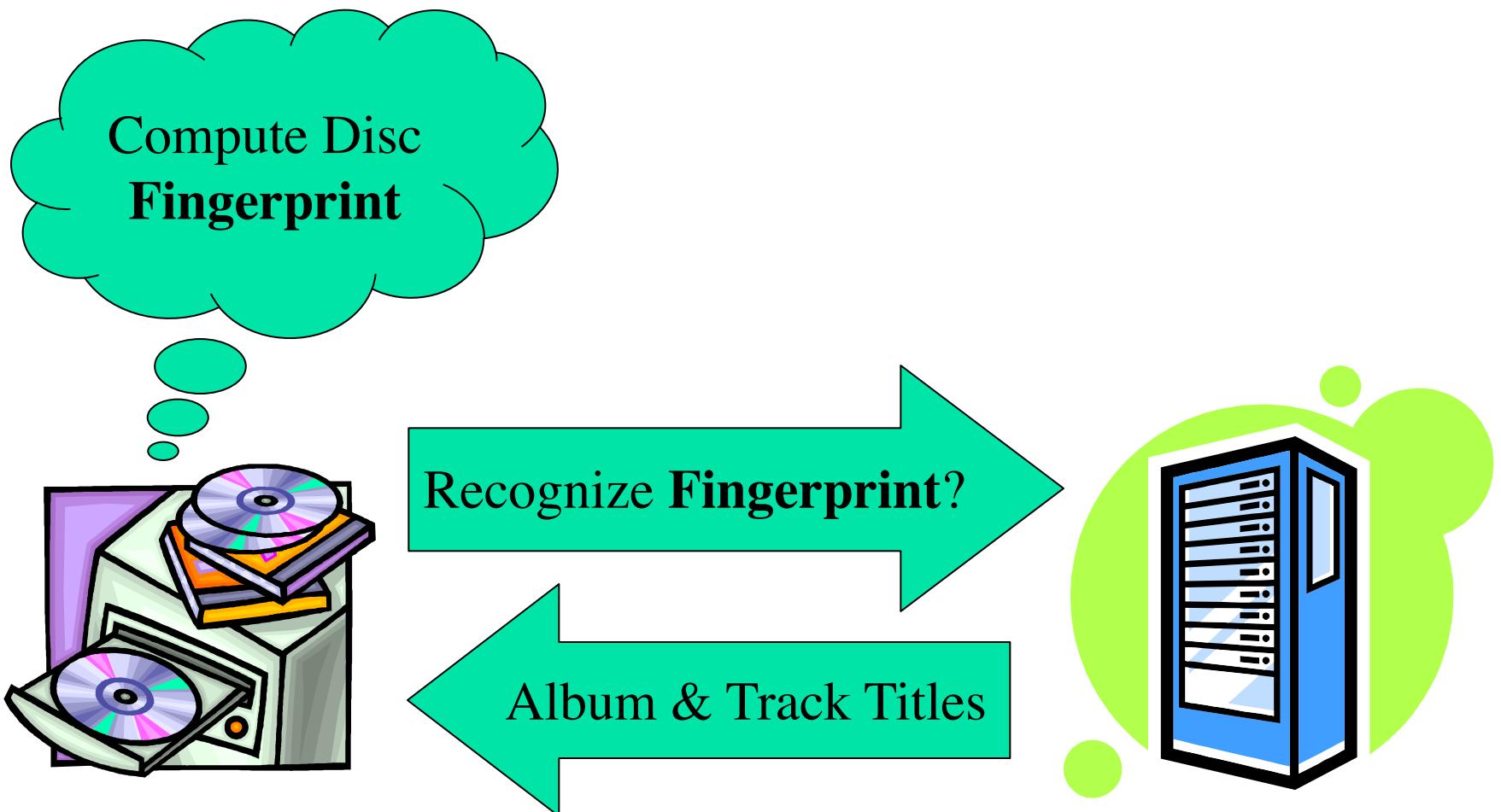
- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node failures/join?
- Issue 4: How to deal with node heterogeneity

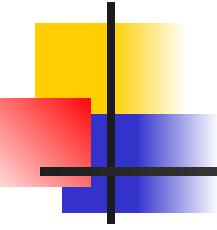
- Issue 6: [DHT] Iterative vs. recursive routing
- Issue 7: Performance optimizations



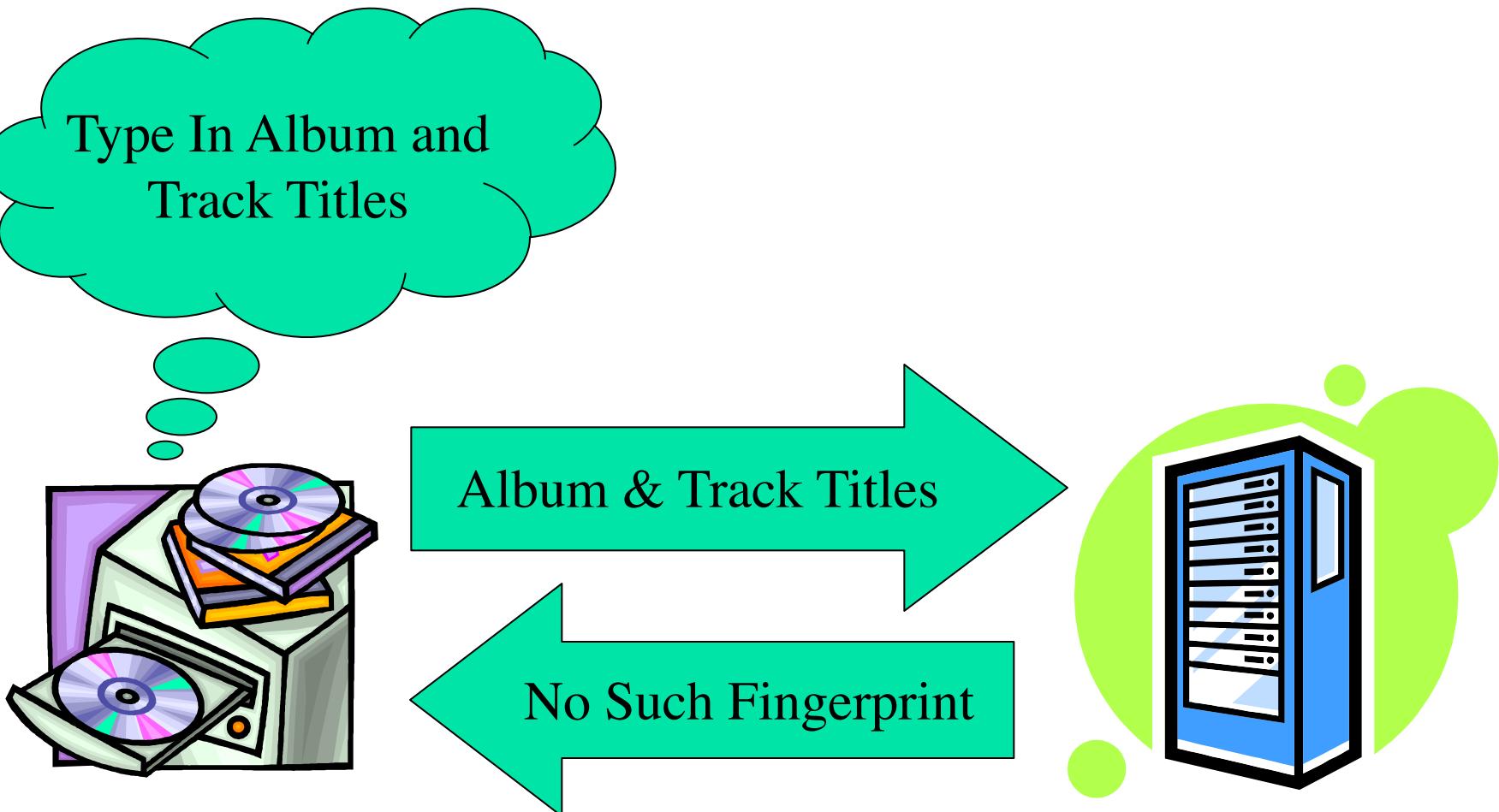
Some applications

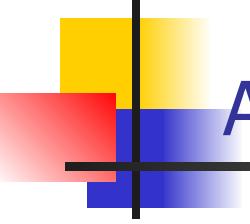
An Example Application: The CD Database





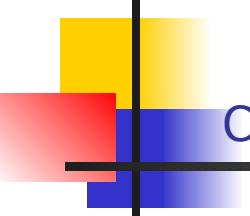
An Example Application: The CD Database



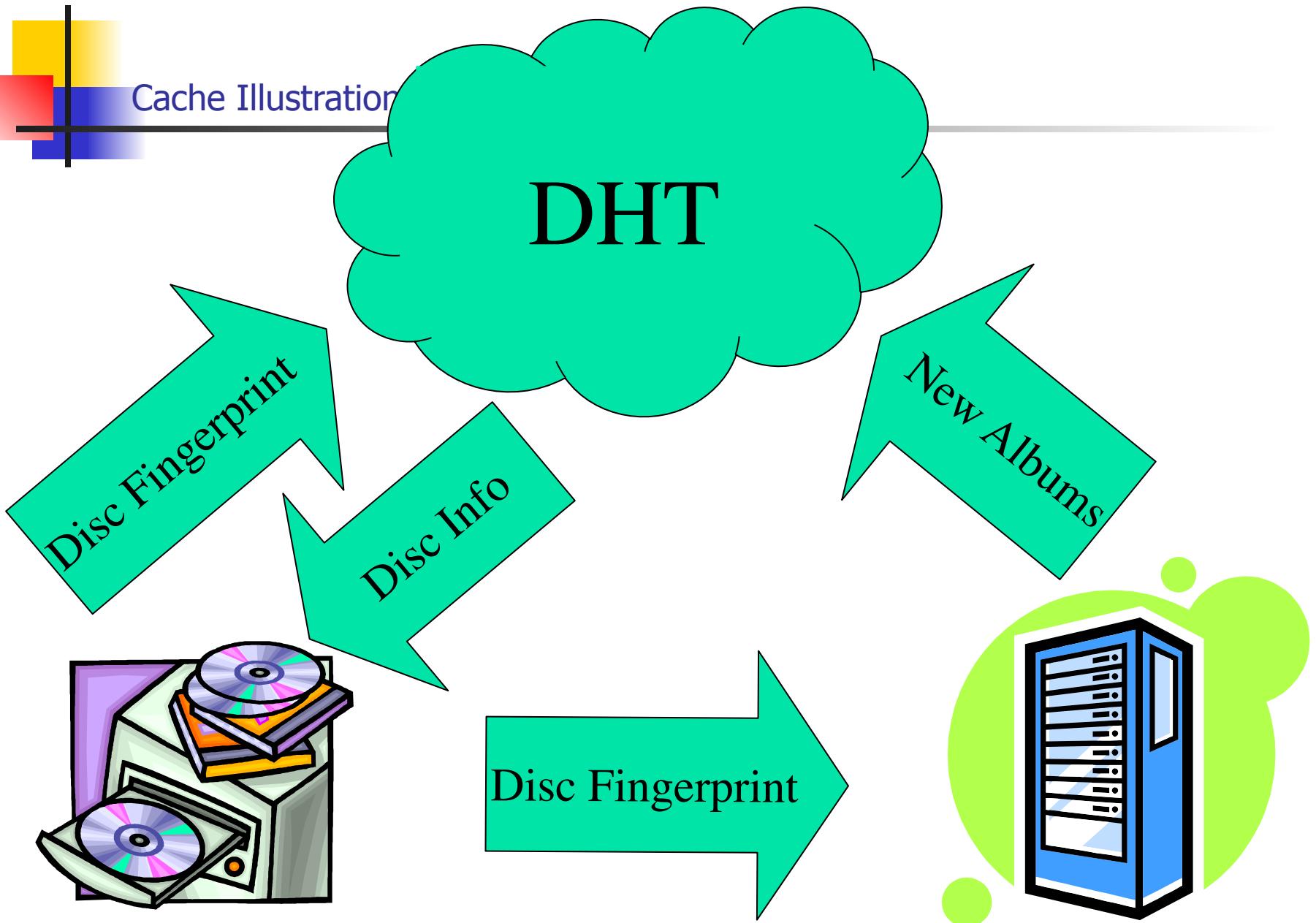


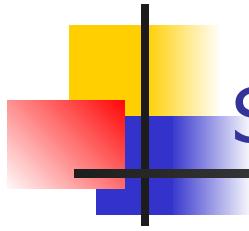
A DHT-Based FreeDB Cache

- FreeDB is a volunteer service
 - Has suffered outages as long as 48 hours
 - Service costs born largely by volunteer mirrors
- Idea: Build a cache of FreeDB with a DHT
 - Add to availability of main service
 - Goal: explore how easy this is to do



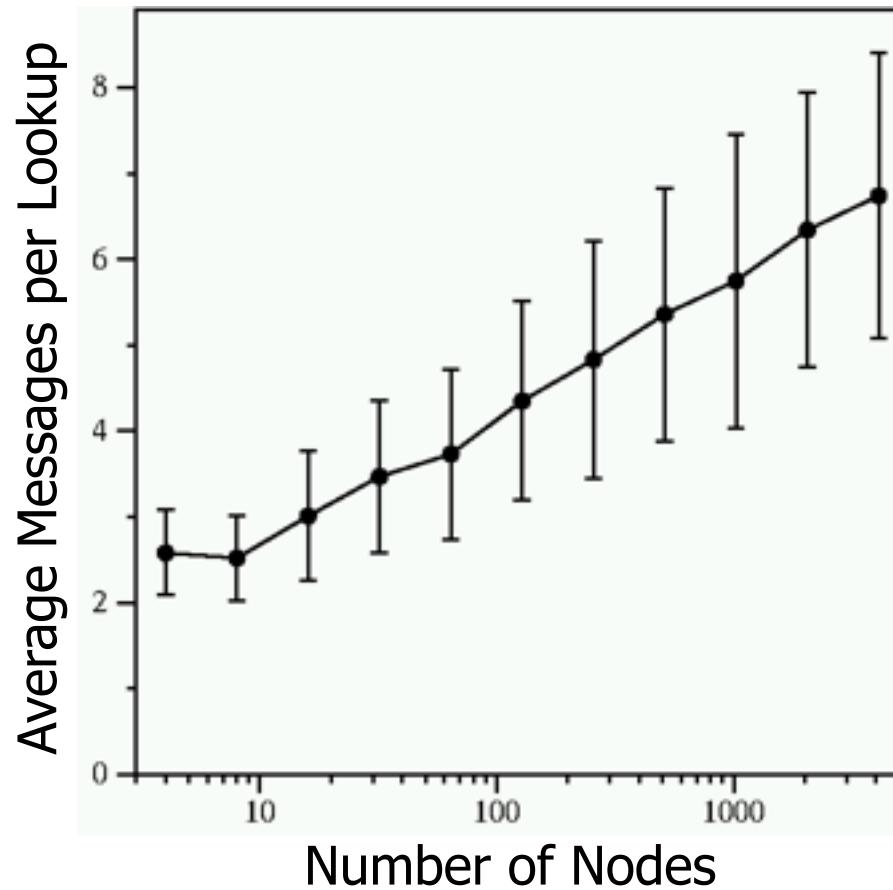
Cache Illustration

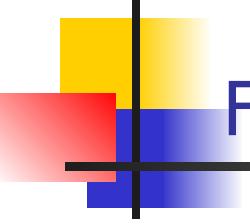




Some experimental results

Chord Lookup Cost Is $O(\log N)$

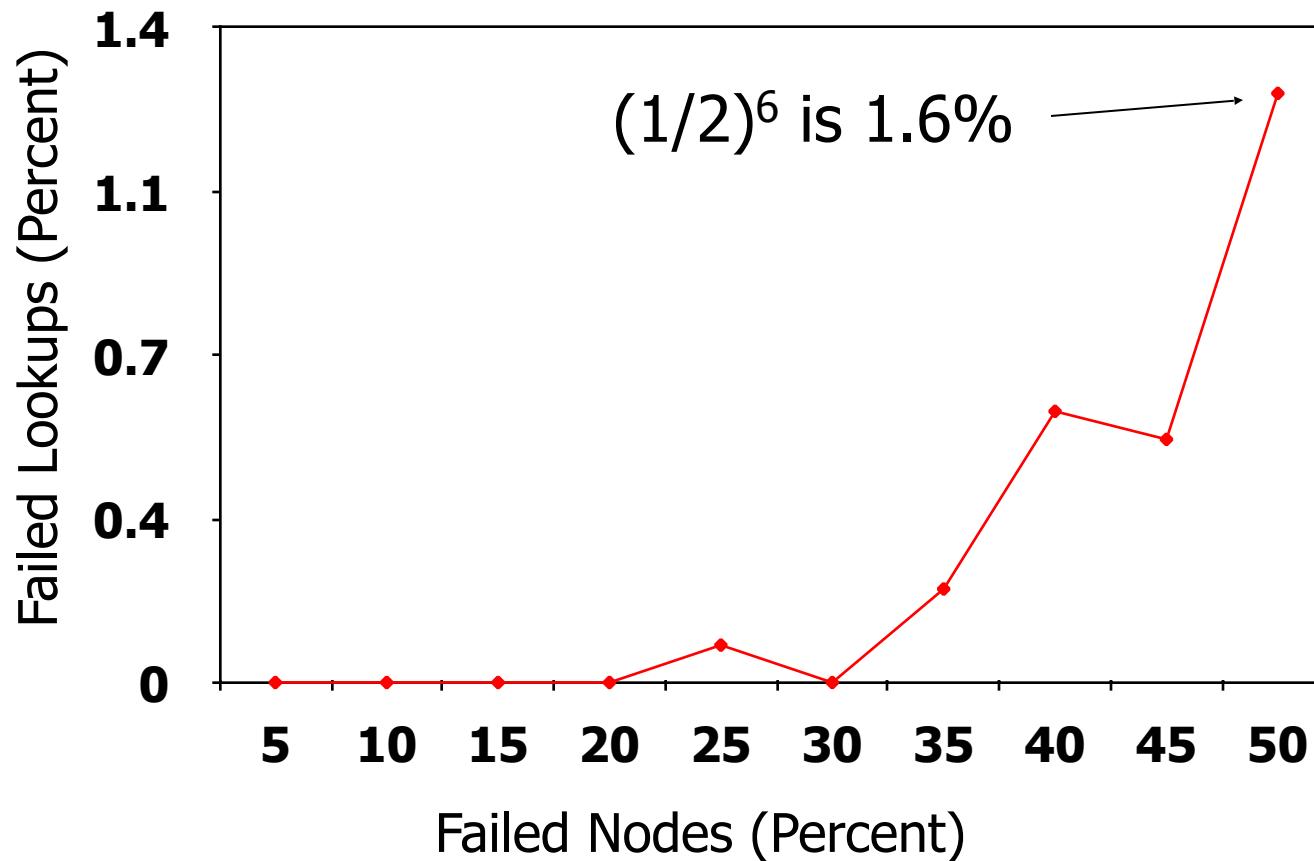




Failure Experimental Setup

- Start 1,000 CFS/Chord servers
 - Successor list has 20 entries
- Wait until they stabilize
- Insert 1,000 key/value pairs
 - Five replicas of each
- Stop X% of the servers
- Immediately perform 1,000 lookups

Massive Failures Have Little Impact



A distributed system is:

- a collection of **independent computers** that appears to its users as a **single coherent system**

Components need to:

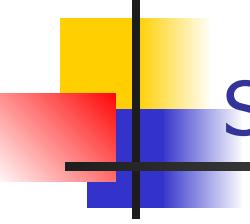
- Communicate
- Cooperate => support needed
 - Naming – enables some resource sharing
 - **Synchronization**

Consider a circular Distributed Hash Table with identifiers in the range [0; 127]. Suppose there are eight participating nodes with identifiers 1, 13, 43, 51, 70, 83, 100 and 115. The DHT is configured so that the successor list has length 2. Also, the DHT is configured so that the finger table has size one: i.e., each peer maintains only one 'shortcut' (or 'finger') – this aims to reduce the search space in half.

- a). Suppose that the following (key,value) pairs should be stored in the DHT: (0,'mama'), (3,'tata'), (7,'zaza'), (15,'bibi'), and (125, 'cici'). Which peers will store each (key,value) pair?
- b.) Assume a search launched at node 13 for key 0. Describe the search process.
- c.) Suppose that peer 13 learns that peer 43 has left the DHT. How does peer 13 update its successor state information? Which peer is now its first successor? Its second successor? Is there any change in the set of keys each peer is responsible for?
- c.) Suppose that a new peer with the identifier 5 wants to join the DHT and it initially only knows the IP address of the peer 53. What steps are taken for peer 6 to join the system? How does the system look like after peer 6 joins?



Synchronization
Physical clocks
Logical clocks



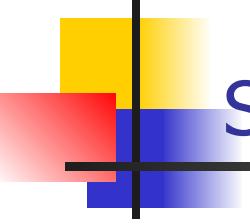
Summary so far ...

A distributed system is:

- a collection of **independent computers** that appears to its users as a **single coherent system**

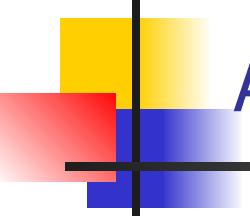
Components need to:

- Communicate
 - Point to point: sockets, RPC/RMI
 - Point to multipoint: multicast, epidemic
- Cooperate
 - Naming to enable some resource sharing
 - Naming systems for flat (unstructured) namespaces: consistent hashing, DHTs
 - Naming systems for structured namespaces
 - **Synchronization**



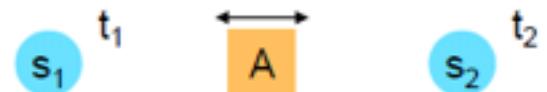
Synchronization to support coordination

- Examples
 - Distributed make
 - Printer sharing
 - Monitoring of a real world system
 - Agreement on message ordering
- Why is synchronization more complex than in a single-box system
 - No global views, multiple clocks, failures



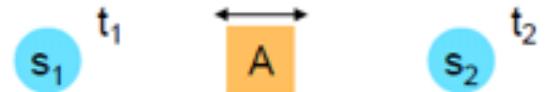
An example ...

- Two shooters in a multiplayer online game shoot (with good accuracy) the same target.
 - Need to decide who gets the point?
- Actual implementations: Generally use replicated state
- Object A (the target) is observed from two vantage points S_1 and S_2 at local times t_1 and t_2 .
 - Need to aggregate everything into one consistent view.

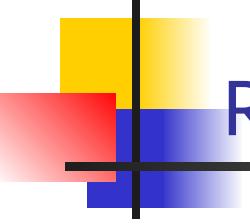


An example ...

- Object A (the target) is observed from two vantage points S_1 and S_2 at local times t_1 and t_2 .
 - Need to aggregate everything into one consistent view.

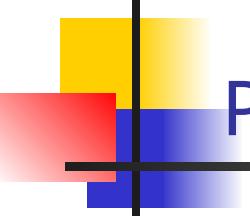


- Issues: Correctness. Fairness. Overheads.



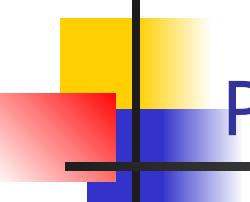
Roadmap

- **Physical clocks**
 - Provide actual / real time
- **'Logical clocks'**
 - Where only ordering of events matters
- **Leader election**
 - How do I choose a coordinator?
- **Mutual exclusion**
 - How does one implement critical regions



Physical clocks (I)

- **Problem:** How to achieve **agreement on time** in a distributed system?
- A possible solution: use Universal Coordinated Time (UTC):
 - Atomic clocks: Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
 - At present, the real time is taken as the average of some 50 cesium-clocks around the world.
 - Introduces a leap second from time to time to compensate for days getting longer.
- UTC is **broadcast** through short wave radio and satellite.
 - Accuracy $\pm 1\text{ms}$ (but if weather conditions considered $\pm 10\text{ms}$)



Physical clocks - underlying model

Suppose we have a distributed system with a UTC-receiver somewhere in it.

Problem: we still have to distribute time to each machine.

Internal mechanism at each node

- Each machine has a **timer**
- Timer causes an **interrupt H times a second**
- Interrupt handler adds 1 to a software clock
- Software clock keeps track of the number of ticks since agreed-upon time in the past.

Physical clocks – main problem: clock drift

Notation: Value of clock on machine p at real time t is $C_p(t)$

Ideally: $C_p(t) == t$ and $dC_p(t) = dt$

Real world: **clock drift**, i.e., $|C_p(t) - t| > 0$

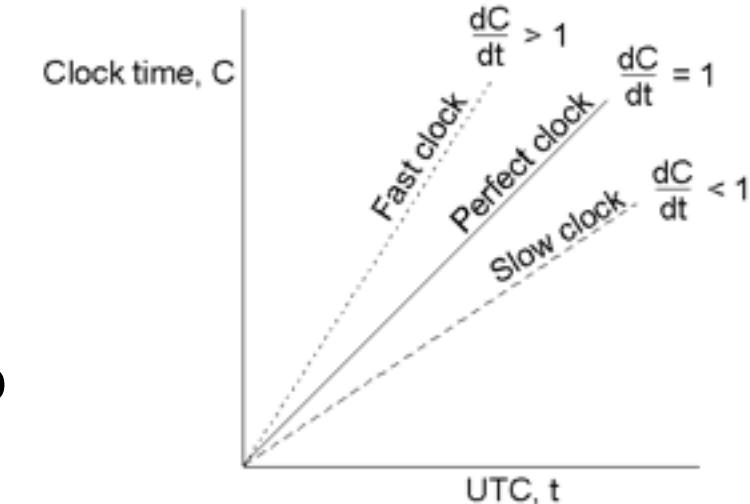
Clock value (C_p) guaranteed to progress:

$$1 - \rho \leq (dC/dt) \leq 1 + \rho$$

ρ -- maximum **drift rate**

Goal: Bound drift, i.e., never let clocks two nodes differ by more than x time units

Solution: synchronize at least every $x/(2\rho)$ seconds.



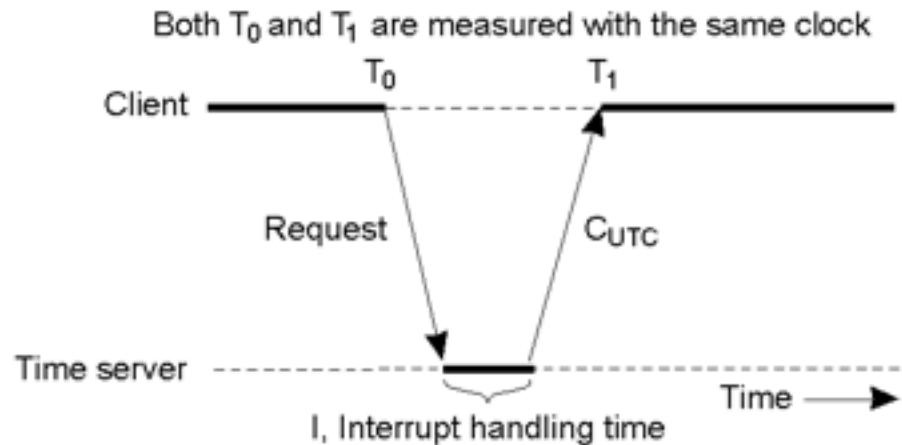
Building a complete system ...

Option I: Every machine asks a **time server** for the accurate time at least once every $x/(2p)$ seconds (Network Time Protocol).

- Need to account for network delays, including interrupt handling and processing of messages.

- **Client updates time to?**

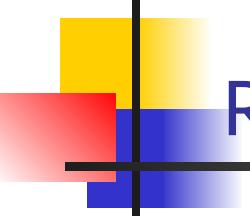
$$T_{\text{new}} = C_{\text{UTC}} + (T_1 - T_0)$$



▪ **Fundamental:** You'll have to take into account that setting the time back is **never** allowed → smooth adjustments.

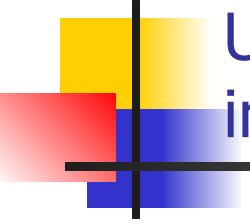
Option II: Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

- **Note:** you don't even need to propagate UTC time.



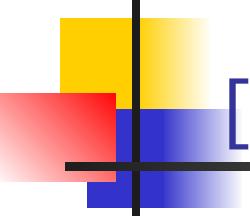
Real world: Network Time Protocol (NTP)

- Stratum 0 NTP servers – receive time from external sources (cesium clocks, GPS, radio broadcasts)
- Stratum N+1 servers synchronize with stratum N servers and between themselves
 - Self-configuring network
- **Survey (N. Minar'99)**
 - > 175K NTP servers
 - 90% of the NTP servers have <100ms offset fro synchronization peer
 - 99% are synchronized within 1s



Uses of (synchronized) physical clocks in the real world

- NTP
- Global Positioning Systems
- Using physical clocks to support at-most-once semantics



[Summary] Physical clocks

- Clock **drift**: time difference between two clocks
- Sources of errors (drift)
 - Variability in time to propagate radio signals. Variability. ($\pm 10\text{ms}$)
 - Clocks are not perfect: **Drift rates**
 - Network latencies are not symmetric
 - Differences in speed to process messages
- System design to limit drift
 - One node holds the 'true' time
 - Other nodes contact this node periodically and adjust their clocks
 - How often?
 - How exactly the adjustment is done?

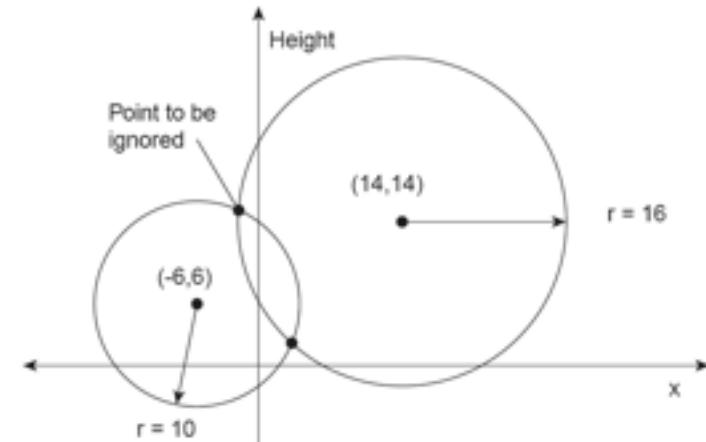
- We've established that clocks can not be perfectly synchronized (and atomic clocks are costly).
- What can I do in these conditions?

- Estimate out how large the drift is
 - Example: GPS systems
- Design the system to take drift into account
 - Example: Server design to provide at-most-once semantics
- Give up physical clocks!
 - Consider only event order - Logical clocks

GPS – Global Positioning Systems (1)

Basic idea: Estimate signal propagation time between satellite and receiver to estimate distance to satellite

- **Strawman:** Assume that the clocks of the satellites and receiver are accurate and synchronized:



- **Real world:**
 - 3D not 2D
 - The receiver's clock is definitely out of synch with the satellite

- Unknowns: X_r , Y_r , Z_r coordinates of the receiver.
- Known:
 - X_i , Y_i , Z_i coordinates of satellite i
 - T_i is the send timestamp on a message from satellite i
- $\Delta T_i = (T_{\text{now}} - T_i)$ is the measured delay of the message sent by satellite i.
- **Distance** to satellite i can be estimated in two ways
 - Propagation time: $d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$
 - Real distance:
- 3 satellites → 3 equations in 3 unknowns. I'M DONE!
- ... BUT: I assumed receiver clock is synchronized!

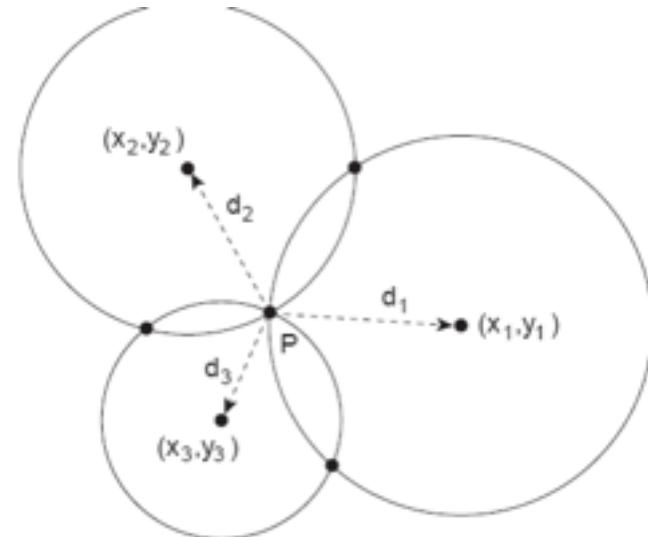
- Unknowns: X_r , Y_r , Z_r coordinates of the receiver.
- Known: X_i , Y_i , Z_i coordinates of satellite i
 - T_i is the send timestamp on a message from satellite i
- $\Delta I_i = (T_{\text{now}} - T_i)$ is the measured delay of the message sent by satellite i.
- **Distance** to satellite i can be estimated in two ways
 - Propagation time: c
 - Real distance: $d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$
- So far I assumed receiver clock is synchronized!
 - What if it needs to be adjusted? $T_{\text{real}} = T_{\text{now}} + \Delta r$
 - $\Delta I = (T_{\text{now}} + \Delta r) - T_i$
 - Collect one more measurement from one more satellite!

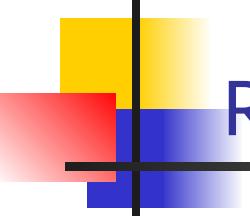
Similar technique other situations:
Computing position in wired networks

Observation: a node P needs at least $k + 1$ landmarks to compute its own position in a k -dimensional space.

Consider two-dimensional case:

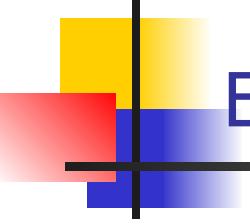
Solution: P needs to solve three equations in two unknowns (x_P, y_P)





Roadmap

- Clocks can not be perfectly synchronized.
- What can I do in these conditions?
 - Figure out how large exactly the drift is
 - Example: GPS systems
 - **Design the system to take clock drift into account**
 - **Example: Efficient server design for at-most-once RPC semantics**
 - Give up physical clocks!
 - Consider only event order - Logical clocks



Efficient at-most-once message delivery

Design point: Server needs to maintain request identifiers to avoid replayed requests

Assumptions:

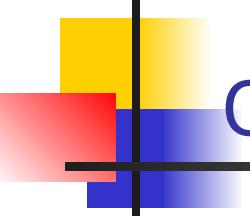
- No bound on message propagation time
- There is a bound on clock drift
 - any two clocks in the system differ by at most MaxClockDrift

Issues

- 1: How long to maintain transaction data?
- 2: How to deal with server failures?
 - (minimize the state that is persistently stored)

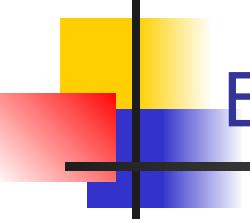
Issue1: How long to maintain transaction data at server?

- Client: may resend messages for up to MaxLifeTime
 - (after that reports error to application)
- Server's goals:
 - 1. Identify message duplicates – **at-most-once delivery**
 - 2. Avoid storing too much state
- Context:
 - Cannot assume bounded message propagation time.
 - Any two clocks in the system differ by at most MaxClockDrift
- Mechanism idea: Server discards messages that have been generated too far in the past
 - Client protocol: client sends transaction id and physical timestamp
 - Server maintains: $G = T_{current} - \text{MaxLifeTime} - \text{MaxClockDrift}$
 - Maintains transaction data only for the interval $[G..T_{now}]$
 - Discards messages with timestamps older than G
 - Ignores (or delays) message that arrive in the future



Quiz-like question

- Previous setup
 - no assumption on maximum message propagation time
 - uses physical clocks (with a bound on max clock drift)
- You can now assume an upper bound B on message propagation time
 - Can you design a solution that does not use physical clocks?



Efficient at-most-once message delivery

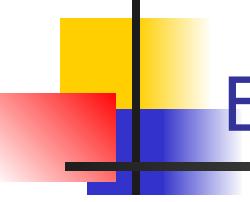
Design point: Server needs to maintain request identifiers to avoid replayed requests

Assumptions:

- No bound on message propagation time
- There is a bound on clock drift
 - any two clocks in the system differ by at most MaxClockDrift

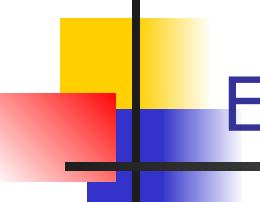
Issues

- 1: How long to maintain transaction data?
- 2: **How to deal with server failures?**
 - (minimize the state that is persistently stored)



Efficient at-most-once message delivery (II)

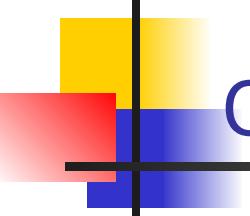
- Issue 2: What to persistently store across server failures?
 - Strawman #1: Persistently store ALL transactions.
 - Strawman #2: Store nothing persistently.



Efficient at-most-once message delivery (II)

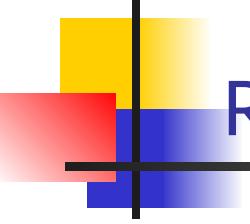
- Issue 2: What to persistently store across server failures?
- Towards a solution: need to have some info to approximate failure time
 - Write current time (CT) to disk every ΔT
 - At recovery read it
 - Failure time is approximated as $G_{failure}$ from last saved CT
 - After recovery – when a new message arrives
 - (let's ignore clock skew for now and assume perfect clocks)
 - Discard messages with timestamp older than $G_{failure} + \Delta T$
 - Process messages with timestamp newer than $G_{failure} + \Delta T$

[Quiz-like question: the two bullets above ignore clock drift.
Should we? If so, how? If not, why not?]



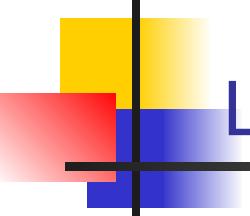
Quiz-like question

- Previous solution
 - no assumption on maximum message propagation time
 - uses physical clocks (with a bound on max clock drift)
- You can now assume an upper bound B on message propagation time
 - Can you design a solution that does not use physical clocks?



Roadmap

- Clocks can not be perfectly synchronized.
- What can I do in these conditions?
 - Figure out how large exactly the drift is
 - Example: GPS systems
 - Design the system to take drift into account
 - Example: Server design to provide at-most-once semantics
 - **Next: Do not use physical clocks!**
 - **Consider only event order - Logical clocks**

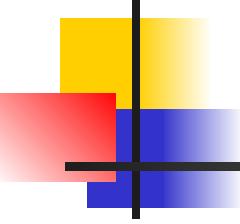


Logical clocks -- Time Revisited

- What's important?
 - The precise time an event occurred?
 - The order in which events occur?

Our example:

- Shooters in a multiplayer online game shoot (and kill) the same target.
 - Need to decide who gets the point



Logical clocks: ROADMAP

Define partial order for events

“happens before” relationship
→

What are the constraints?
What does ‘ordering’ mean?



Logical clocks

Assign timestamp to events such that if $a \rightarrow b$ then $ts(a) < t(b)$

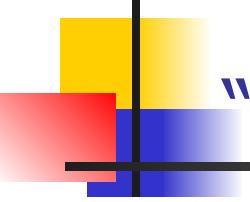
Come up with a system to ‘label’ events that respects these constraints



Build systems

E.g., totally ordered group communication

How is this used?



“Happens-before” relation

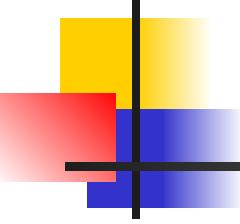
Problem: Need to first **introduce a notion** of ordering before we can order anything.

The **happened-before** relation (notation: “ \rightarrow ”) on the set of events in a distributed system:

- if a, b are events in the same process, and a occurs before b, (in physical time) then $a \rightarrow b$
- if a is the event of sending a message by a process, and b receiving same message by another process then $a \rightarrow b$

Property: transitive

Two events are concurrent if nothing can be said about the order in which they happened (partial order)



Logical clocks: ROADMAP

Define partial order for events

“happens before” relationship
→

What are the constraints?
What does ‘ordering’ mean?



Logical clocks

Assign timestamp to events such that if $a \rightarrow b$ then $ts(a) < t(b)$

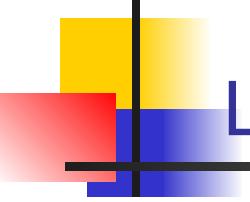
Come up with a system to ‘label’ events that respects these constraints



Build systems

E.g., totally ordered group communication

How is this used?



Logical clocks

- **Goal:** Maintain a **global view** on the system's behavior that is consistent with the 'happened-before' relation?
- **Towards a solution:** attach a timestamp $C(e)$ to each event e , such that:
 - **P1:** If a and b are two events in the same process, and a happened before in physical time b , then we demand that $C(a) < C(b)$.
 - **P2:** If a corresponds to sending a message m , and b to the receipt of that message, then also $C(a) < C(b)$.
- Note: C must only increase
- **Problem:** Need to attach timestamps to all events in the system (consistent with the rules above) when there's no global clock
 - maintain a **consistent** set of logical clocks, one per process.

Problem: Need to attach timestamps to all events in the system

- maintain a **consistent** set of logical clocks, one per process.
- there's no global clock

Solution (Lamport): Each process P_i maintains a **local** counter C_i and adjusts this counter as follows:

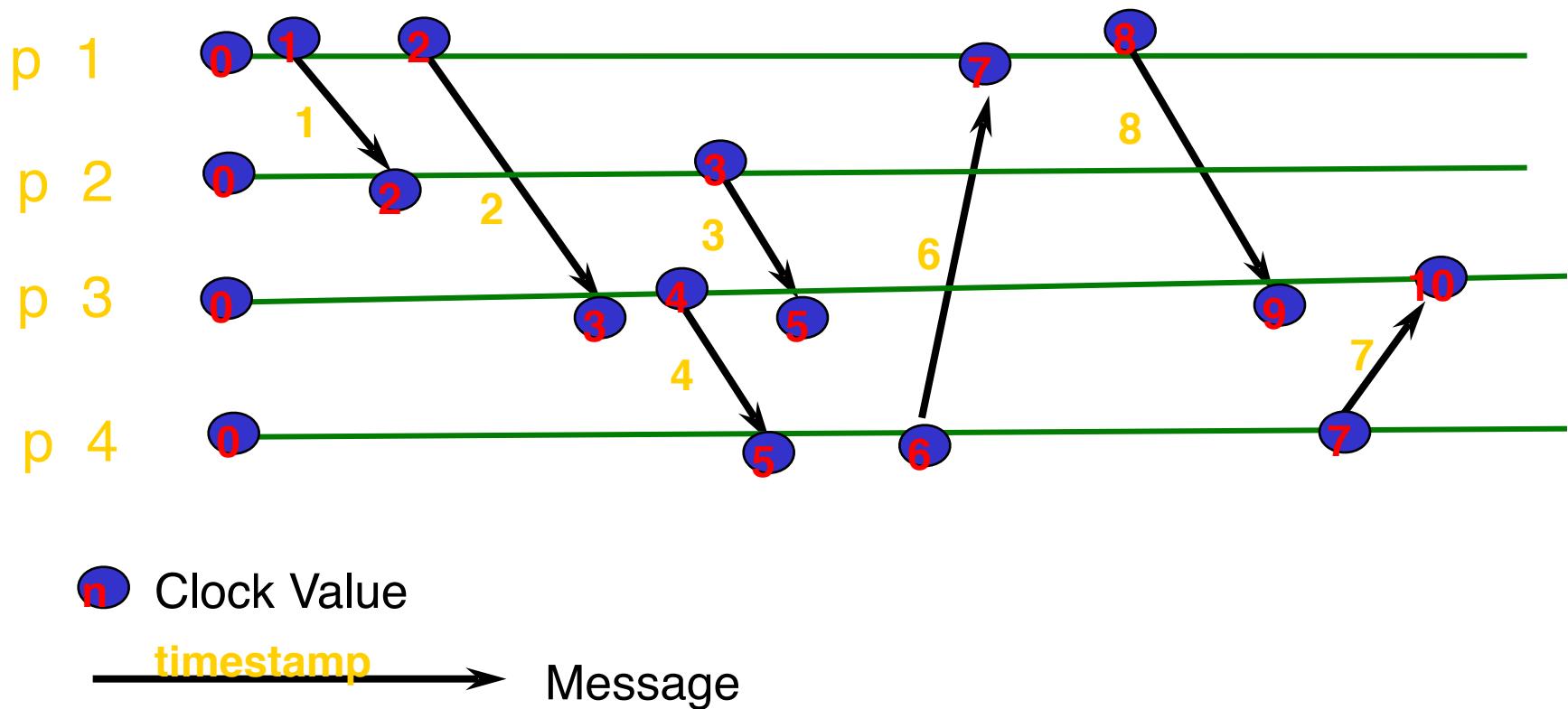
- 1) For any two successive events that take place within P_i , the counter C_i is incremented by 1.
- 2) Each time a message m is sent by process P_i , the message is timestamped $ts(m) = C_i$
- 3) Whenever a message m is received by a process P_j , P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

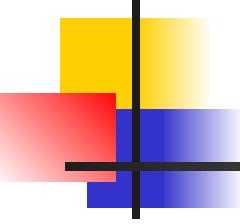
▪ Property **P1** is satisfied by (1); Property **P2** by (2) and (3).

▪ **Note:** it can still happen that two events happen at the same 'time' (have the same logical clock). Avoid this by breaking ties through process IDs.



Physical Time





Quiz-like question

Notation: **timestamp(a)** is the Lamport logical clock associated with event a

By construction if $a \rightarrow b \Rightarrow \text{timestamp}(a) < \text{timestamp}(b)$

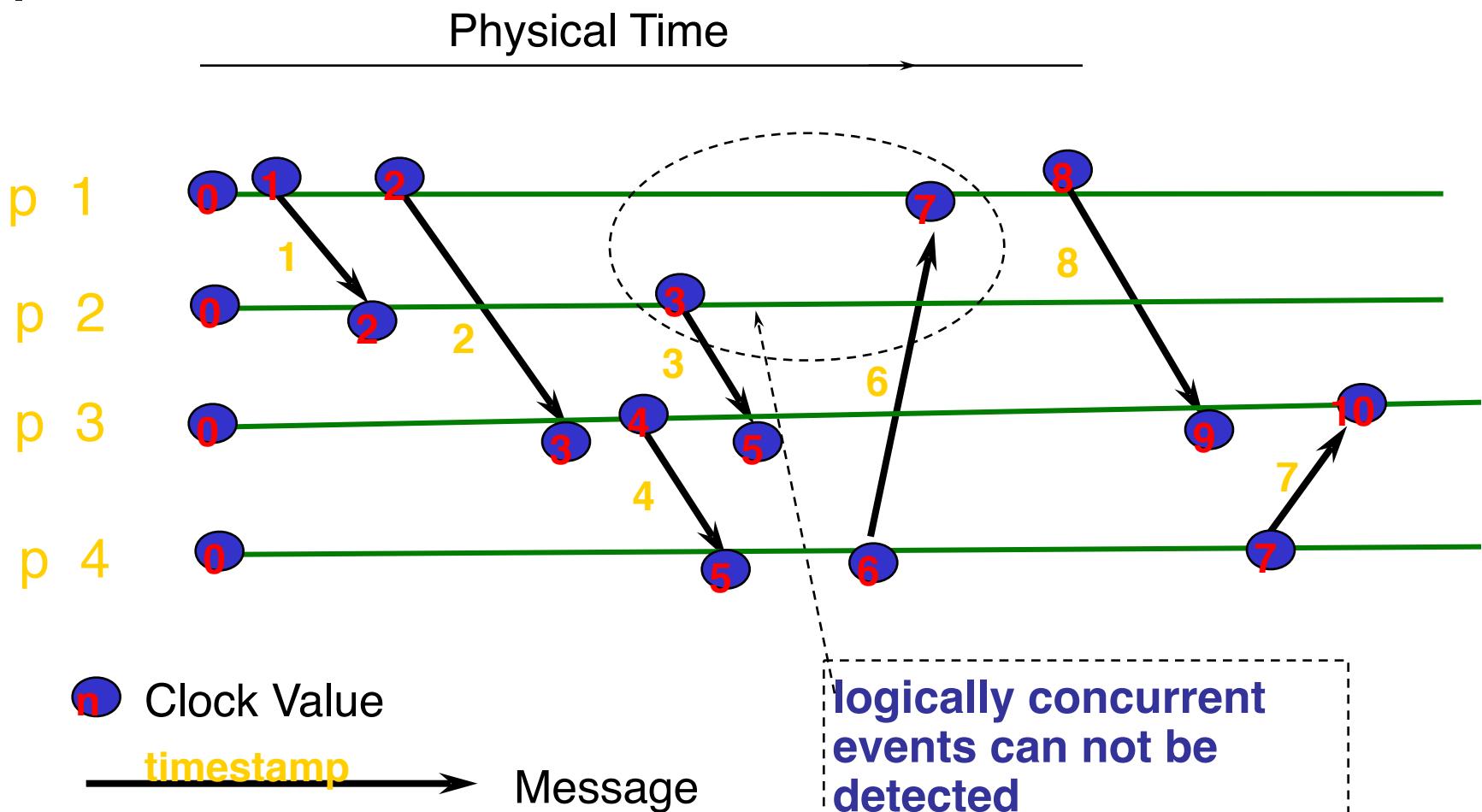
(if **a happens before b**, then **timestamp(a) < timestamp(b)**)

Q: is the converse true?

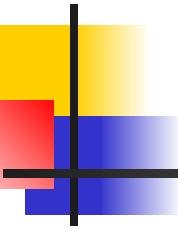
That is: if $\text{timestamp}(a) < \text{timestamp}(b)$  $a \rightarrow b$

If **timestamp(a) < timestamp(b)**, it does NOT imply that **a happens before b**

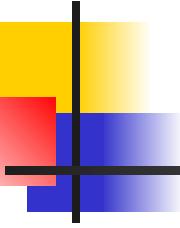
Example



Note: Lamport Timestamps: $3 < 7$, but event with timestamp 3 is concurrent to event with timestamp 7, i.e., events are not in 'happen-before' relation.



MIDTERM MATERIAL UP TO HERE



Define partial order for events

“happens before” relationship



Logical clocks

Assign timestamp to events such that if $a \rightarrow b$ then $ts(a) < t(b)$



Build systems

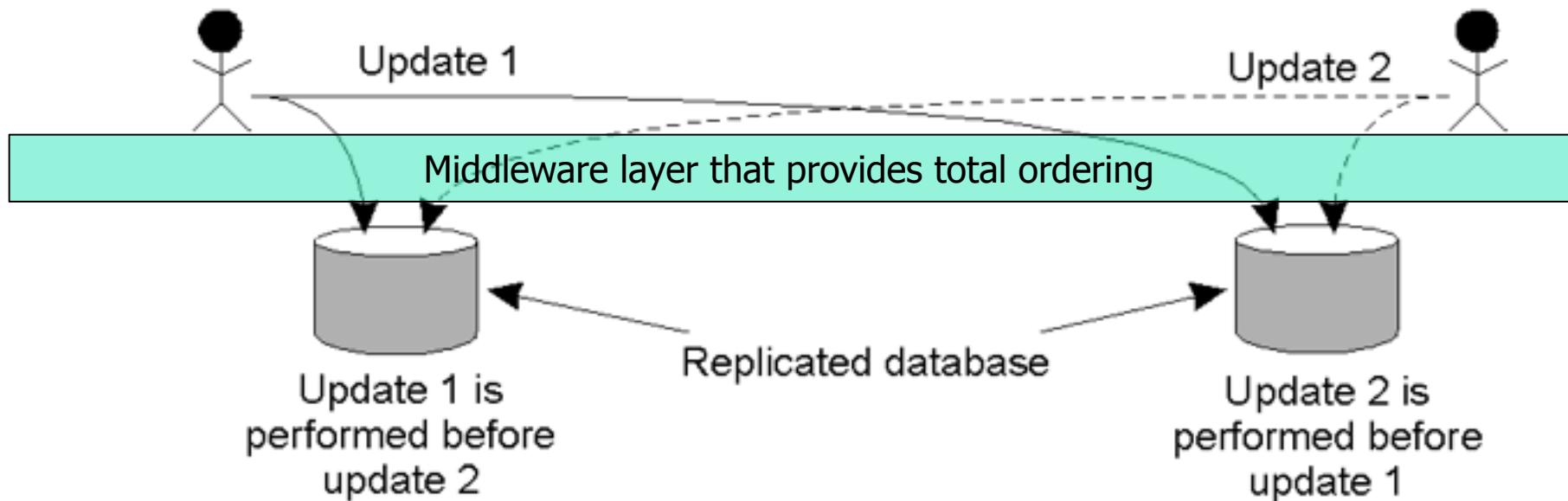
E.g., totally ordered group communication

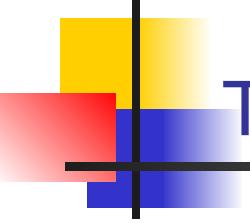
Example use – replicated state machine

Two accounts:

- Initial state: \$100 account balance
- Update 1: add \$100
- Update 2: add 1% monthly interest

Updates need to be performed in the same order at the two replicas!



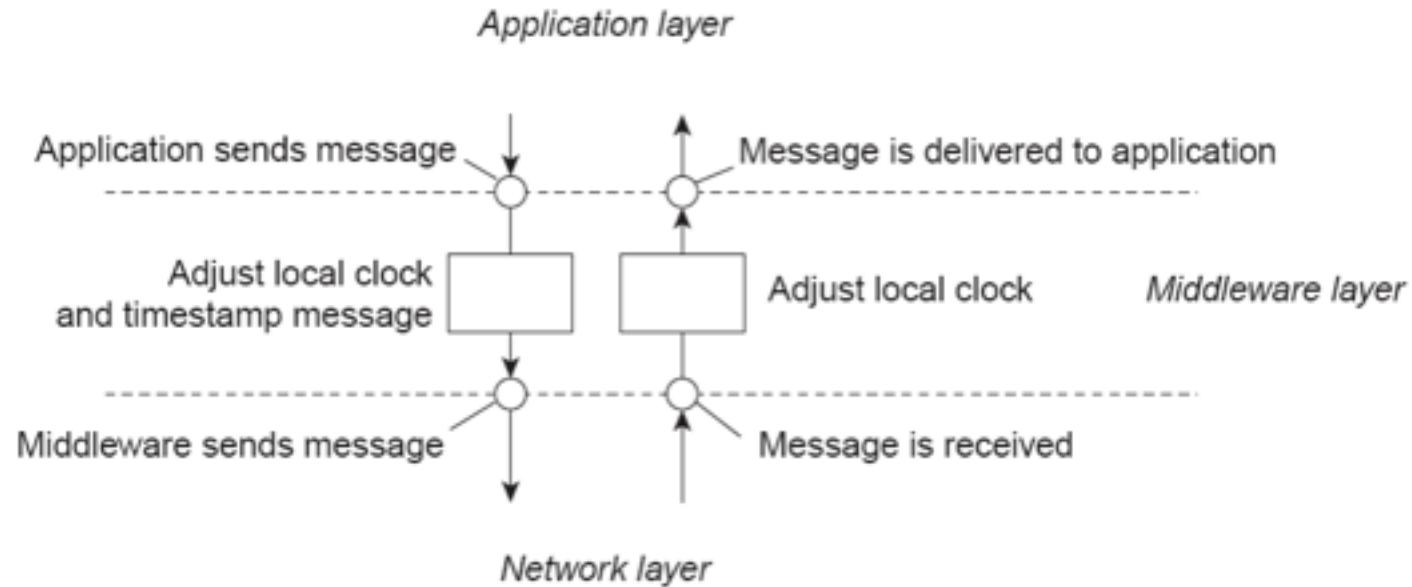


Totally ordered group communication (cont)

Setup (for simplicity):

- Symmetry: all members of the group are both senders and receivers
- Communication substrate: one-to many

Architectural view

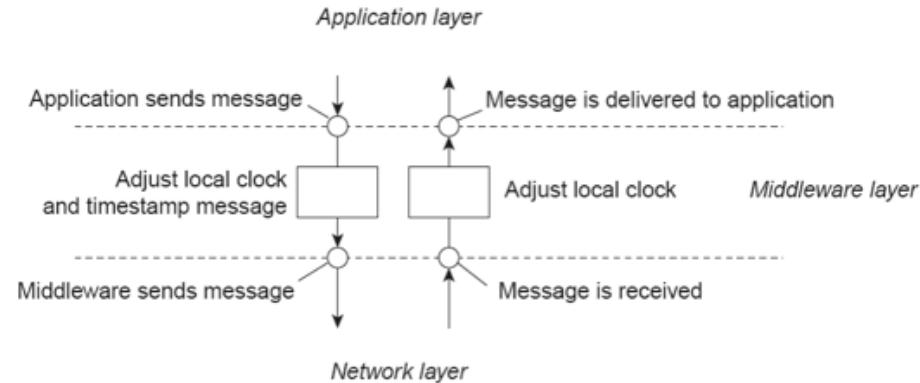


Middleware layer in charge of:

- Local management of logical clocks
 - Stamping messages with (logical) clock times, updating timestamps at message receipt,
 - Message ordering – e.g. by delaying delivery (if necessary)

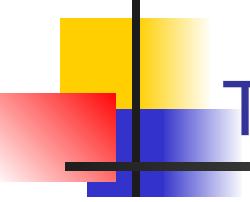
Totally ordered group communication (cont)

- All members are both senders and receivers
- A one-to-many communication substrate



Sketch of a solution:

- Middleware at each member maintains an ordered queue of received messages that have not yet been delivered to the application.
- **Main issue:** when to deliver to application such that, at all endpoints, messages are delivered in the same order.
- Each message is timestamped with local logical time then multicasted
 - When multicasted, also message logically sent to the sender itself
- When receiving a message, the middleware layer
 - Adds message to local queue (ordered by sending timestamp)
 - Acknowledges (using multicast) the message
 - Delivers from top of queue to application only when all acks for message on top have been received (or optimization: see next slide)



Totally Ordered Multicast – Algorithm

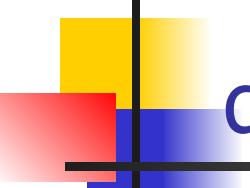
- Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message msg_i received at P_k is queued in $queue_k$, according to its sent timestamp, **and** acknowledged to every other process.
- P_k delivers a message msg_i to its application if:
 - msg_i is at the head of $queue_k$
 - for each process P_x , there is a message or ack in $queue_k$ with a larger sending timestamp.

Guarantee: all messages are delivered in the same order at all destinations

- Nothing is guaranteed about the actual order!

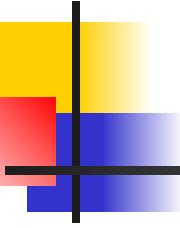
Note: We assume that communication is **reliable** and **FIFO ordered**.

Deliver to application
Sending / Receiving



quiz-like questions

- What's the complexity of the protocol in terms of number of messages
- What happens if we drop channel reliability assumption?
 - Does the protocol still work? If it fails, explain how.
- What happens if we drop channel FIFO assumption?
 - Does the protocol still work?
 - How would you change the previous protocol to still work correctly without this assumption?
- Assume you have a bound on message propagation time in the network. Design a protocol that provides total ordering (and generates less traffic)



Define partial order for events

“happens before” relationship



Logical clocks

Assign timestamp to events such that if $a \rightarrow b$ then $ts(a) < t(b)$



ISSUE

By design

$a \rightarrow b \Rightarrow \text{timestamp}(a) < \text{timestamp}(b)$

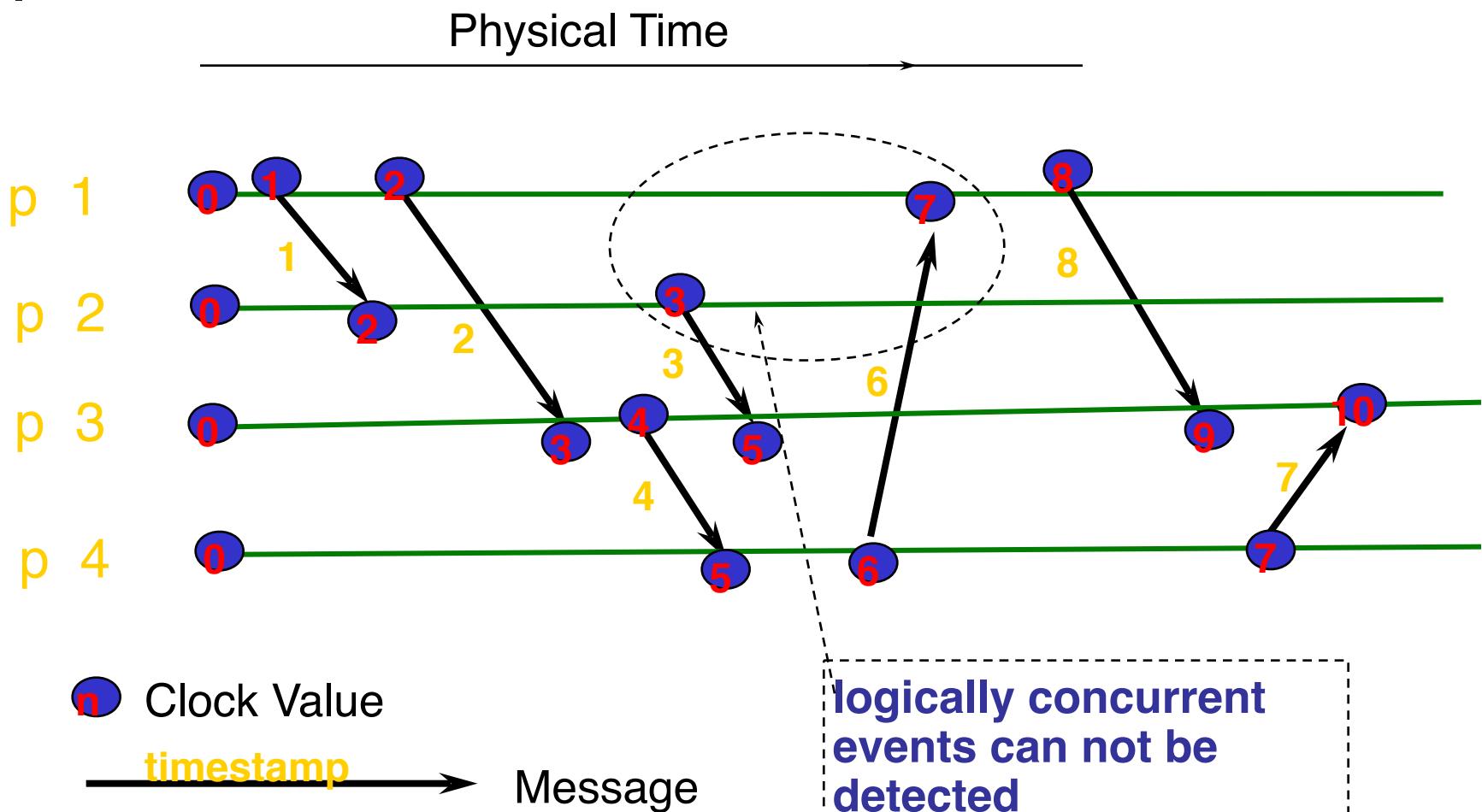
But

If $\text{timestamp}(a) < \text{timestamp}(b)$ one can not reason about relative ordering of a and b

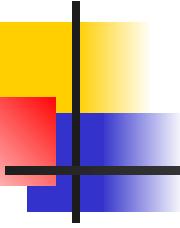
Build systems

E.g., totally ordered group communication

Example



Note: Lamport Timestamps: $3 < 7$, but event with timestamp 3 is concurrent to event with timestamp 7, i.e., events are not in 'happen-before' relation.



Define partial order for events

“happens before” relationship



Vector clocks

Assign timestamps to keep track of event causality

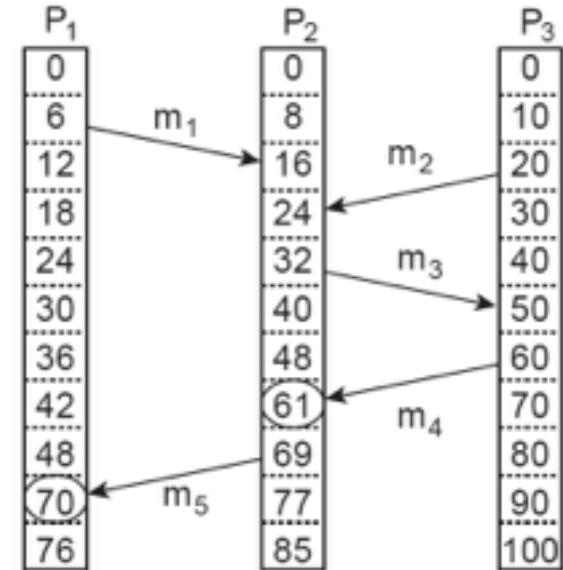


Build systems

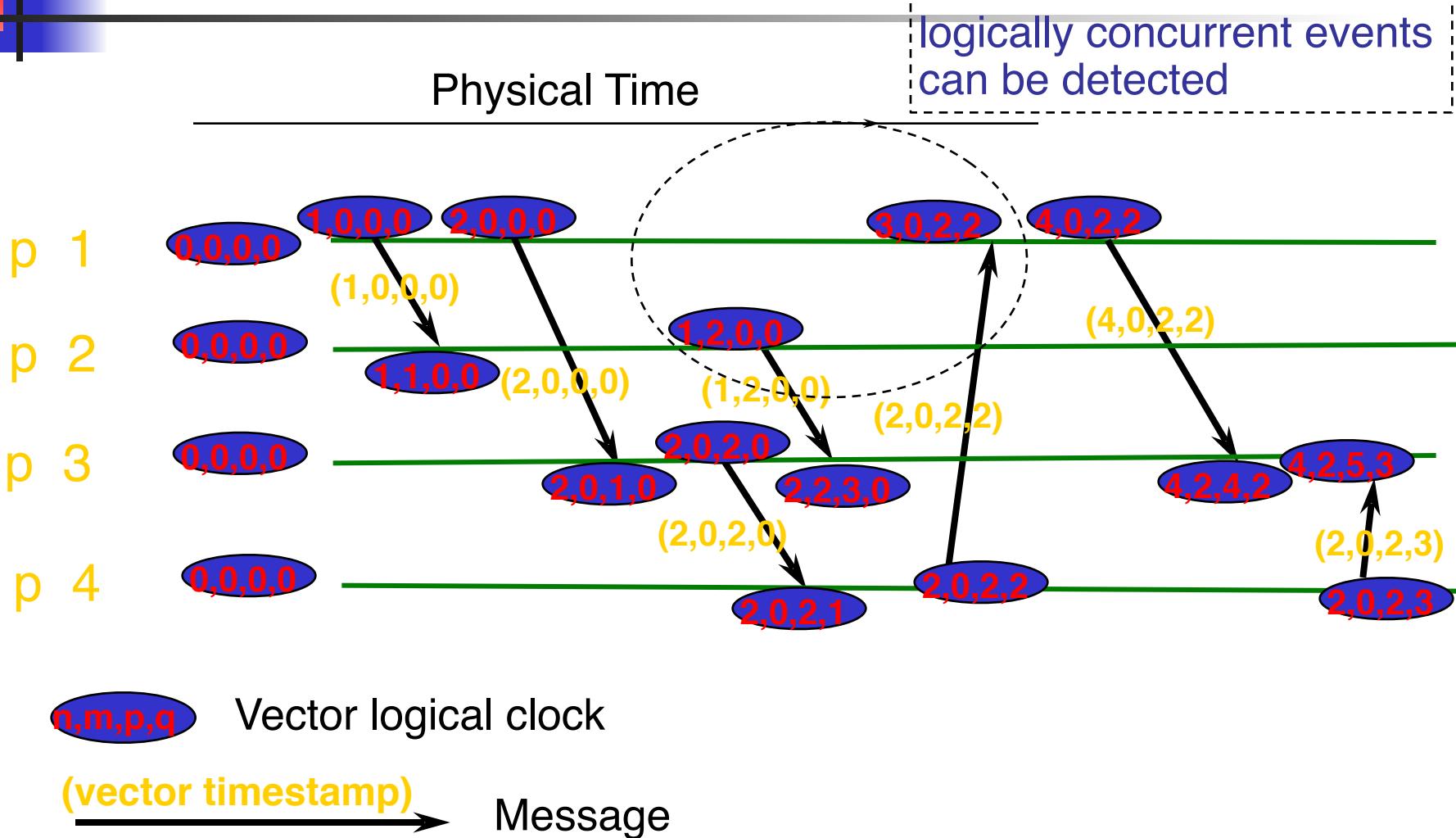
E.g., **causally** ordered group communication

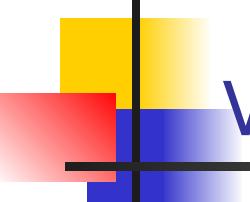
Causality

- Lamport timestamps don't capture causality
 - Introduce more ordering than necessary
 - Example: news postings have multiple independent threads of messages
- To model causality – **vector** timestamps
 - **Intuition**: each item in vector logical clock for one causality thread.



Example: Vector Timestamps

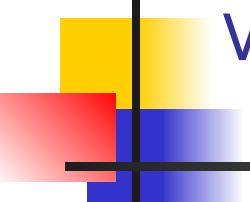




Vector clocks

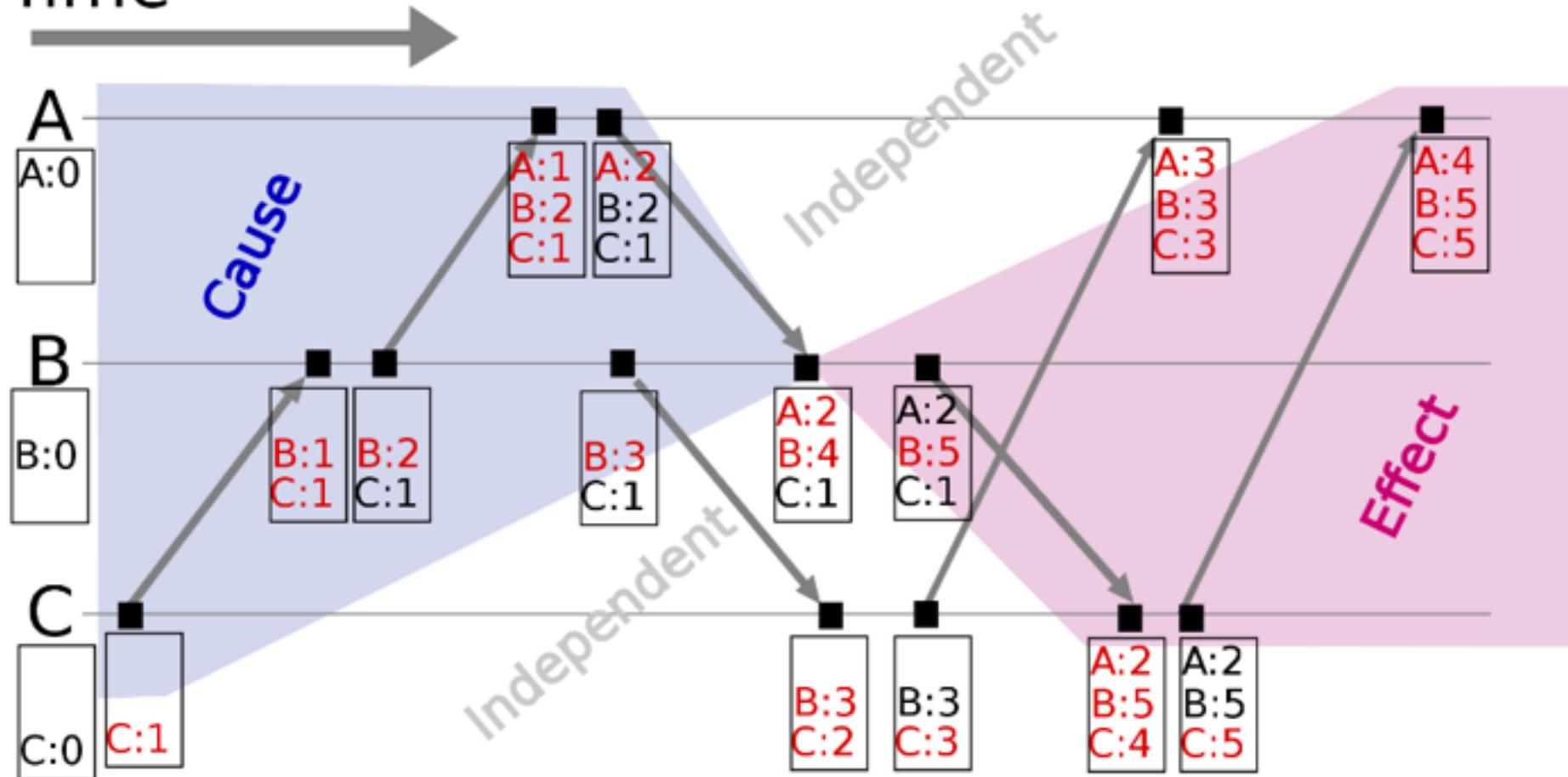
- Each process P_i has an array $VC_i [1..n]$ of clocks (all initially at 0)
 - $VC_i [j]$ denotes the number of events that process P_i knows have taken place at process P_j .
- P_i increments $VC_i [i]$: when an event occurs
 - local event, message sending, message receiving
 - timestamp of the event is vector value
- When **sending**
 - Messages sent by VC_i include a **vector timestamp** $vt(m)$.
 - Result: upon arrival, recipient knows P_i 's timestamp.
- When P_j **receives** a msg from P_i with vector timestamp $ts(m)$:
 - for $k \neq j$: update each $VC_j [k]$ to $\max\{VC_j [k], ts(m)[k]\}$

Note: vector timestamps require a static notion of system membership

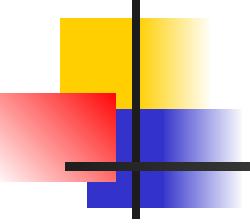


Vector clocks/timestamps enable reasoning about causality

Time



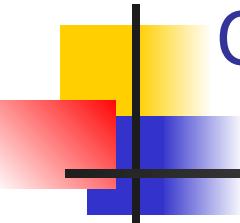
Events in the blue region are the causes leading to event B4, whereas those in the red region are the effects of event B4



Comparing vector timestamps

Notation

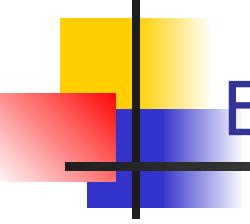
- ❖ $VT_1 < VT_2$,
 - iff forall j ($1 \leq j \leq n$) such that $VT_1[j] \leq VT_2[j]$
 - and
 - exists j ($1 \leq j \leq n$) such that $VT_1[j] < VT_2[j]$
- ❖ VT_1 is **concurrent** with VT_2
 - iff (not $VT_1 < VT_2$ AND not $VT_2 < VT_1$)



Quiz like problem

Show that:

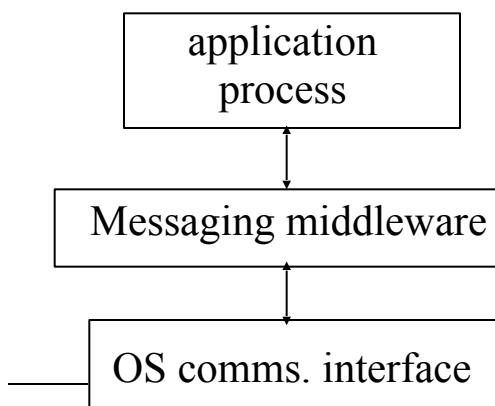
$$\mathbf{a} \rightarrow \mathbf{b} \quad \text{if and only if} \quad \mathbf{vectorTS(a)} < \mathbf{vectorTS(b)}$$



Extending group communication

ASSUMPTIONS

- messages are multicast to (named) process groups
- reliable and fifo channels
- processes don't crash (failure and restart not considered)
- processes behave as specified e.g., send the same values to all processes (i.e., we are not considering Byzantine behaviour)



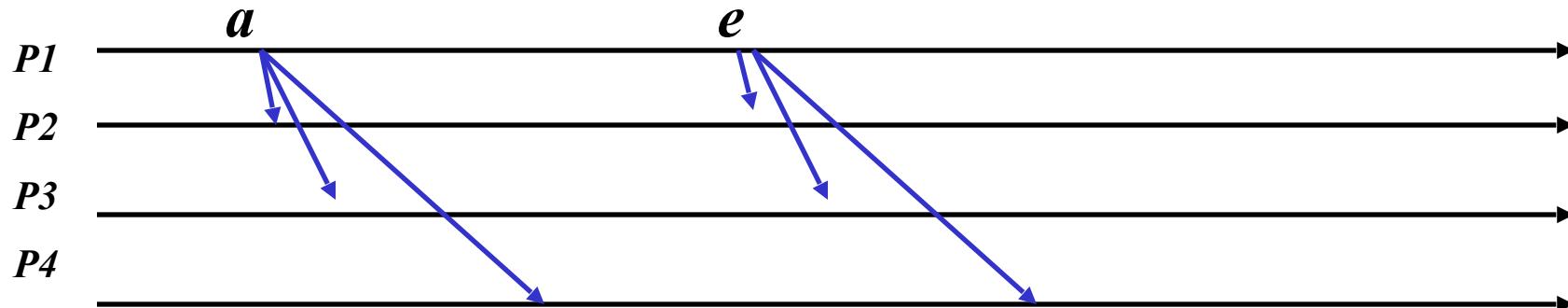
*specifies delivery order of message service
e.g. **total order, FIFO order, causal order**
(last time we looked at 'total order')*

may reorder or delay delivery to application by buffering messages

assume FIFO from each source (done at lower levels)

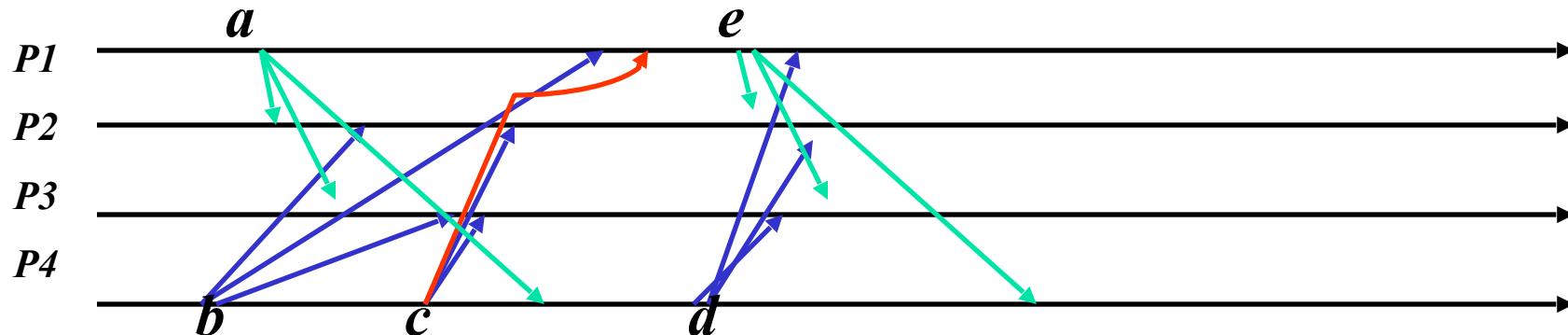
FIFO multicast

- FIFO (or sender ordered) multicast: Messages are delivered in the order they were sent by any single sender
 - Nothing guaranteed for ordering of messages sent by two different senders

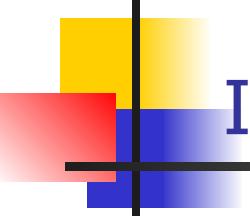


FIFO multicast

- FIFO (or sender ordered) multicast: Messages are delivered in the order they were sent by any single sender
 - Nothing guaranteed for ordering of messages sent by two different senders



- *delivery of c to P1 is delayed until after b is delivered*
- *d and e are received in different order at P2 and P3*

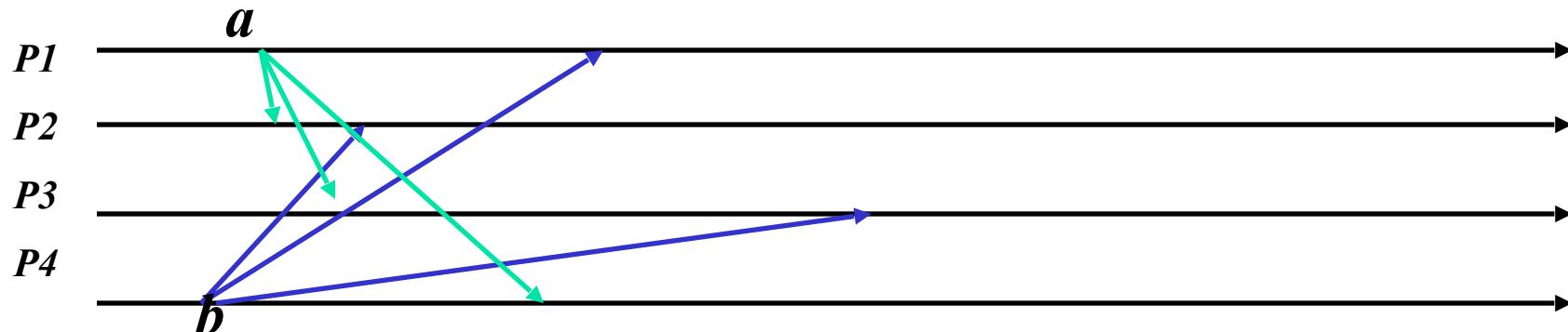


Implementing FIFO multicast

- Basic reliable (i.e., no message loss) multicast has this property
 - Without failures **all we need is to run it on FIFO channels** (like TCP)
 - [Later: dealing with node failures]

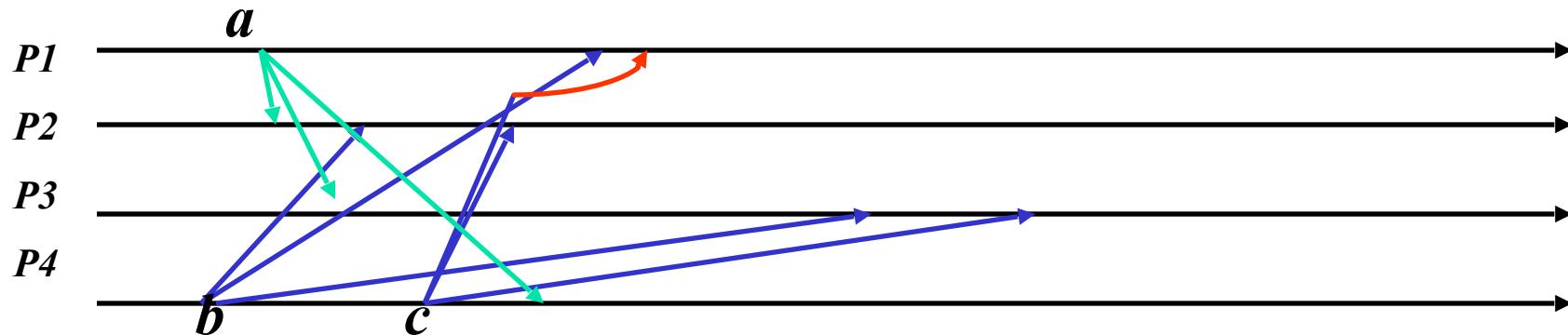
Causal multicast

- Causal (or happens-before ordering)
- If $\text{send}(a) \rightarrow \text{send}(b)$ (i.e., there was some causal relationship)
 - then $\text{deliver}(a)$ occurs before $\text{deliver}(b)$ at common destinations



Ordering properties: Causal

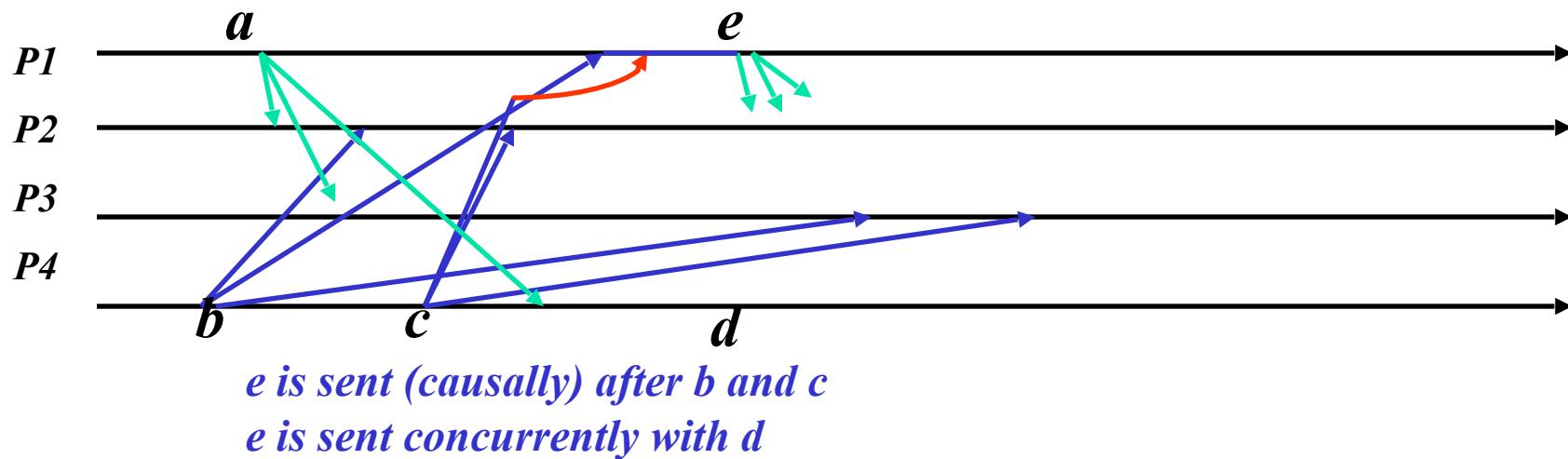
- Causal (or happens-before ordering)
- If $\text{send}(a) \rightarrow \text{send}(b)$ (i.e., there was some causal relationship)
 - then $\text{deliver}(a)$ occurs before $\text{deliver}(b)$ at common destinations



delivery of c to $P1$ is delayed until after b is delivered

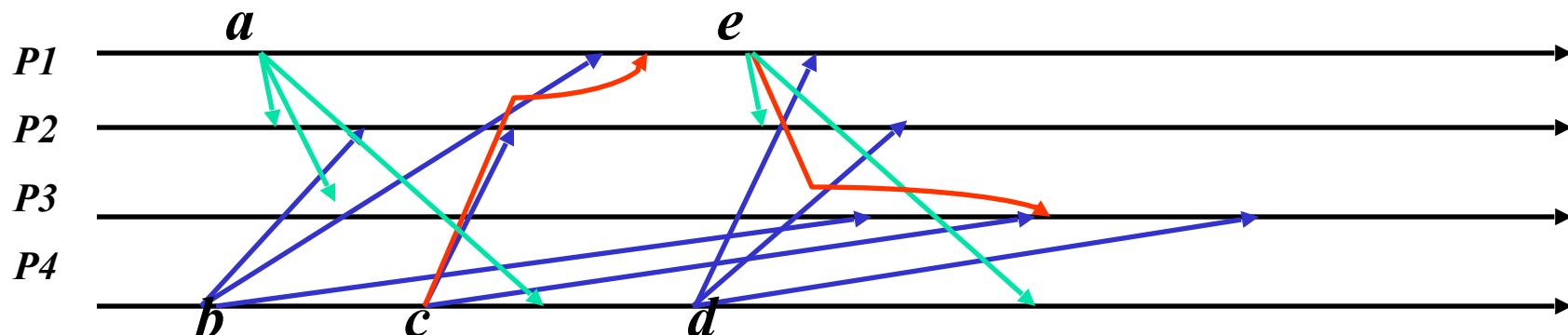
Ordering properties: Causal

- Causal (or happens-before ordering)
- If $\text{send}(a) \rightarrow \text{send}(b)$ (i.e., there was some causal relationship)
 - then $\text{deliver}(a)$ occurs before $\text{deliver}(b)$ at common destinations



Ordering properties: Causal

- Causal (or happens-before ordering)
- If $\text{send}(a) \rightarrow \text{send}(b)$ (i.e., there was some causal relationship)
 - then $\text{deliver}(a)$ occurs before $\text{deliver}(b)$ at common destinations

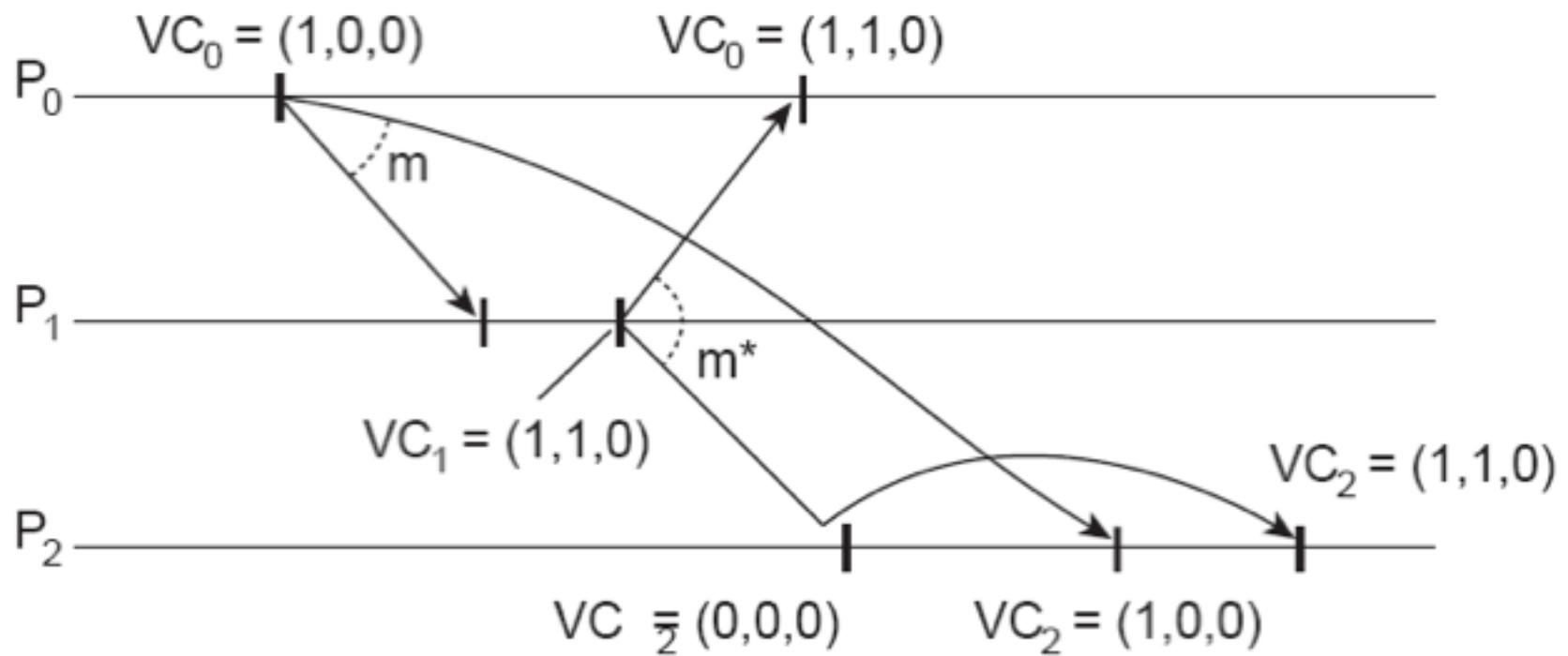


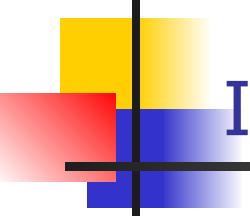
delivery of c to $P1$ is delayed until after b is delivered

delivery of e to $P3$ is delayed until after $b\&c$ are delivered

delivery of e and d to $P2$ and $P3$ in any relative order (concurrent)

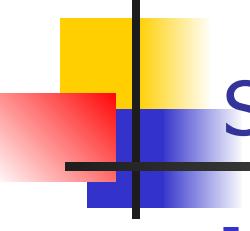
Causally ordered multicast





Implementing causal order multicast

- Start with FIFO multicast
- Strengthen this into a causal multicast by adding vector time
- **Advantage I:** No additional messages needed!
 - Lower overhead
- **Advantage II:** FIFO and causal multicast are asynchronous:
 - Sender doesn't get blocked and can deliver a copy to itself without "stopping" to learn a safe delivery order

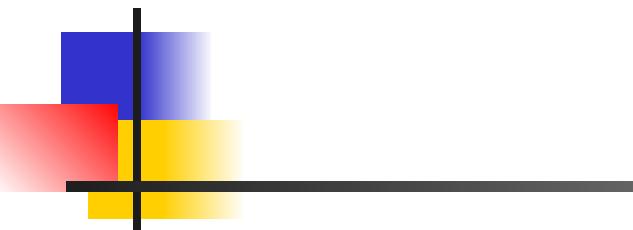


So far ...

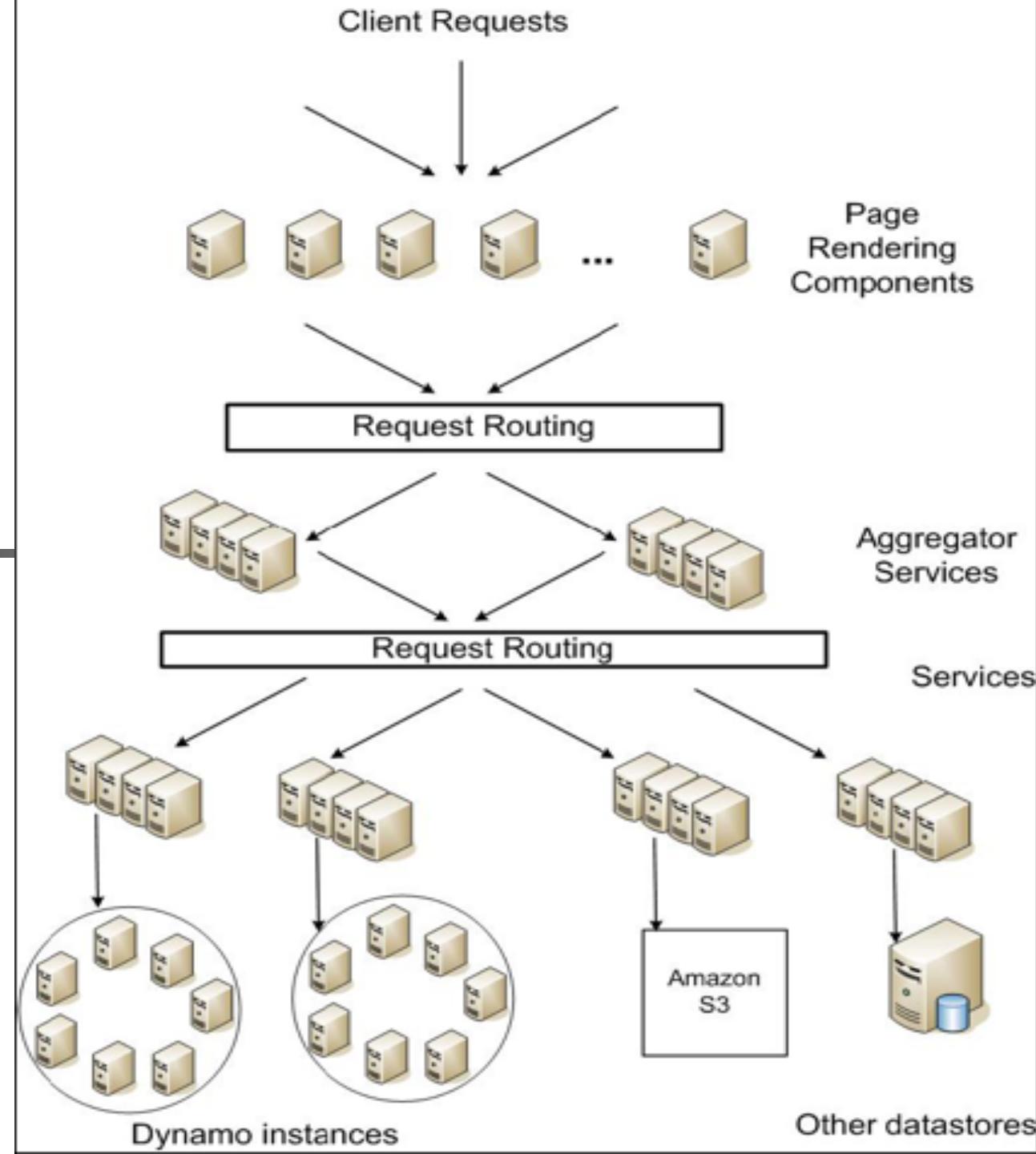
- Physical clocks
 - Two applications
 - Provide at-most-once semantics
 - Global Positioning Systems
 - 'Logical clocks'
 - Where only ordering of events matters
- Next:
 - One application that puts everything together
 - Other coordination primitives
 - Mutual exclusion
 - Leader election

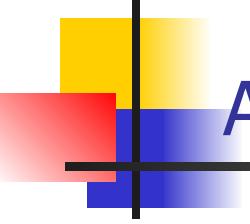
Dynamo: Amazon's Highly Available Key- value Store

(SOSP'07)



Giuseppe DeCandia,
Deniz Hastorun,
Madan Jampani,
Gunavardhan Kakulapati,
Avinash Lakshman, Alex
Pilchin, Swaminathan
Sivasubramanian, Peter
Vosshall
and Werner Vogels





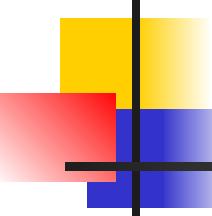
Amazon eCommerce platform

Scale (in 2006)

- > 10M customers at peak times
- > 10K servers (distributed around the world)
- 3M checkout operations per day (peak season)

Problem: Reliability/availability at massive scale:

- Slightest outage has significant financial consequences and impacts customer trust.



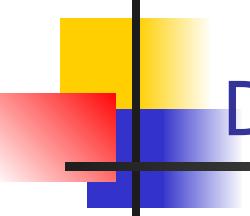
Amazon eCommerce platform - Requirements

Key application requirements:

- Data/service availability the key issue.
 - “Always writeable” data-store
- Low latency – delivered to (almost all) clients/users
 - Example SLA: provide a 300ms response time, for 99.9% of requests, for a peak load of 500requests/sec.
 - Why not average/median?

Architectural requirements

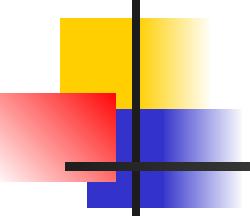
- Incremental scalability
- Symmetry
- Ability to run on a heterogeneous platform



Data Access Model

- Data stored as (key, object) pairs:
 - Interface `put(key, object)`, `get(key)`
 - 'identifier' (key) generated as a hash for object
 - Objects: Opaque

Application examples: shopping carts, customer preferences, session management, sales rank, product catalog, S3



Further assumptions:

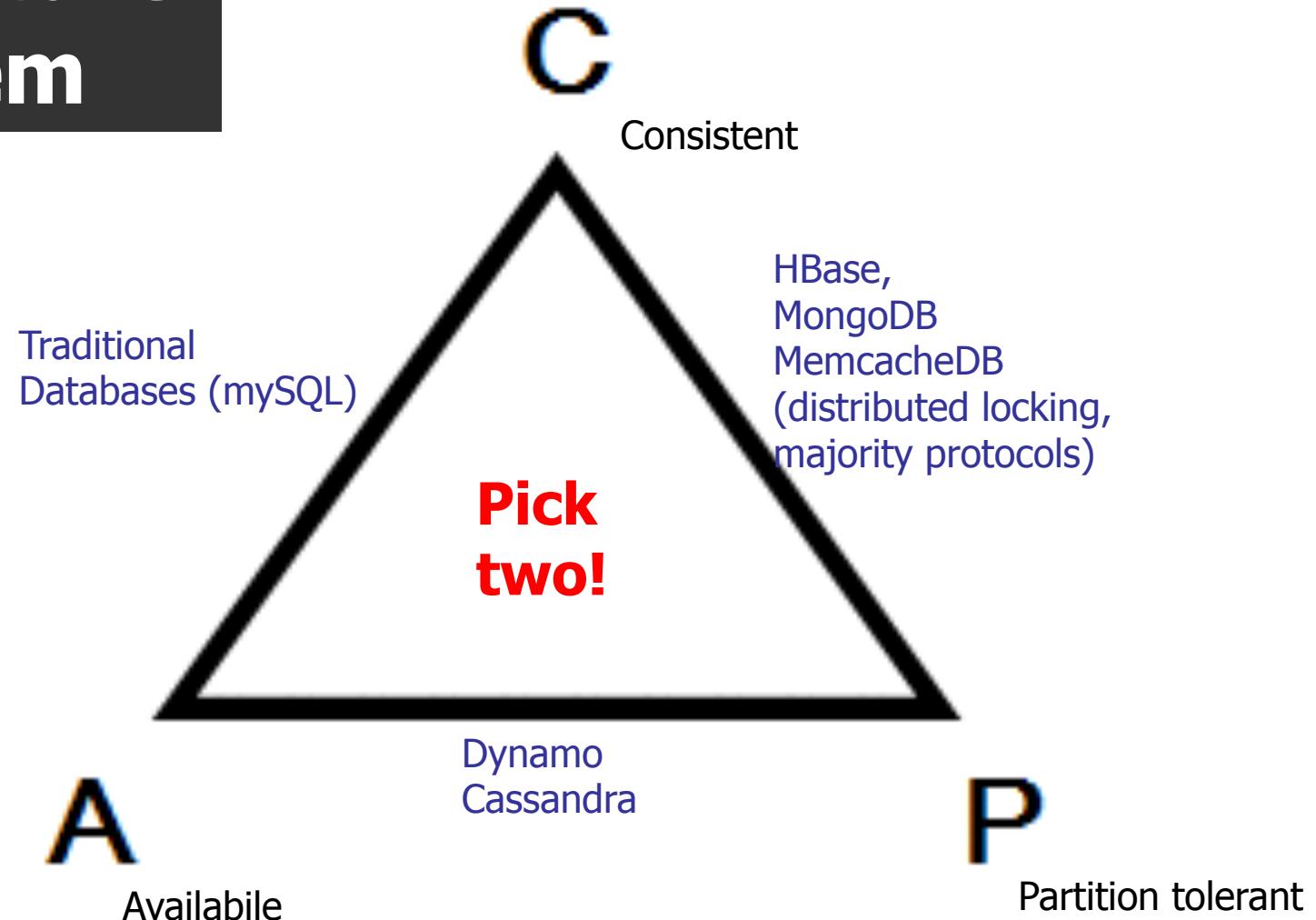
- Relatively small objects (<1MB)
- Query by objectID
- Operations do not span multiple objects
- Friendly (cooperative) environment
- One Dynamo instance per service → 100s hosts/service

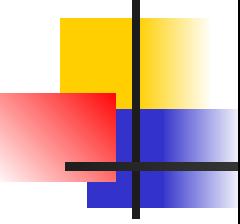
A database?

Concerns: consistent + available + partition tolerant

- Can one get all three?

CAP ~~Conjecture~~ Theorem





THE DATING CONJECTURE



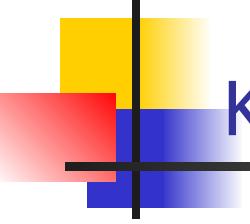
DATING

Requirements:

- High data availability; always writeable data store

Key ideas

- Multiple replicas ...
- ... but avoid synchronous replica coordination ...
 - (used by solutions that provide strong consistency).
 - Tradeoff: Consistency \leftrightarrow Availability
- ... and use 'weak consistency' models to improve availability
 - Then decide: **when** to resolve possible conflicts, and **who** should solve them
 - **When**: at read time (allows providing an "always writeable" data store)
 - **Who**: the application or the data store



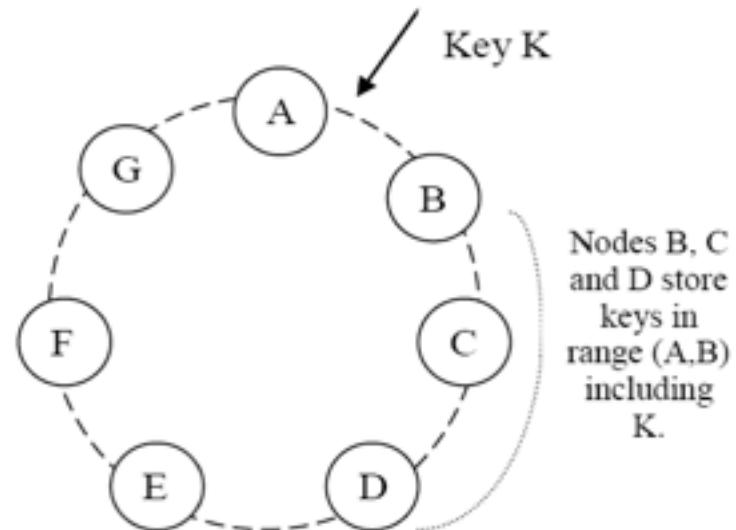
Key technical issues

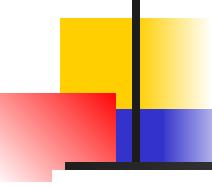
- Partitioning the key/data space
- High availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection

Problem	Technique	Advantage
Partitioning	<ul style="list-style-type: none"> Consistent hashing 	Incremental scalability, load balancing, etc.
High availability for writes	<ul style="list-style-type: none"> Eventual consistency Vector clocks with reconciliation during reads Quorum protocol 	Availability
Handling temporary failures	<ul style="list-style-type: none"> 'Sloppy' quorum protocol and hinted handoff 	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	<ul style="list-style-type: none"> Anti-entropy using Merkle trees 	Synchronizes divergent replicas in the background.
Membership and failure detection	<ul style="list-style-type: none"> Gossip-based membership protocol 	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Partition Algorithm: Consistent hashing

- Each data item is replicated at N hosts (successors)





Quorum systems

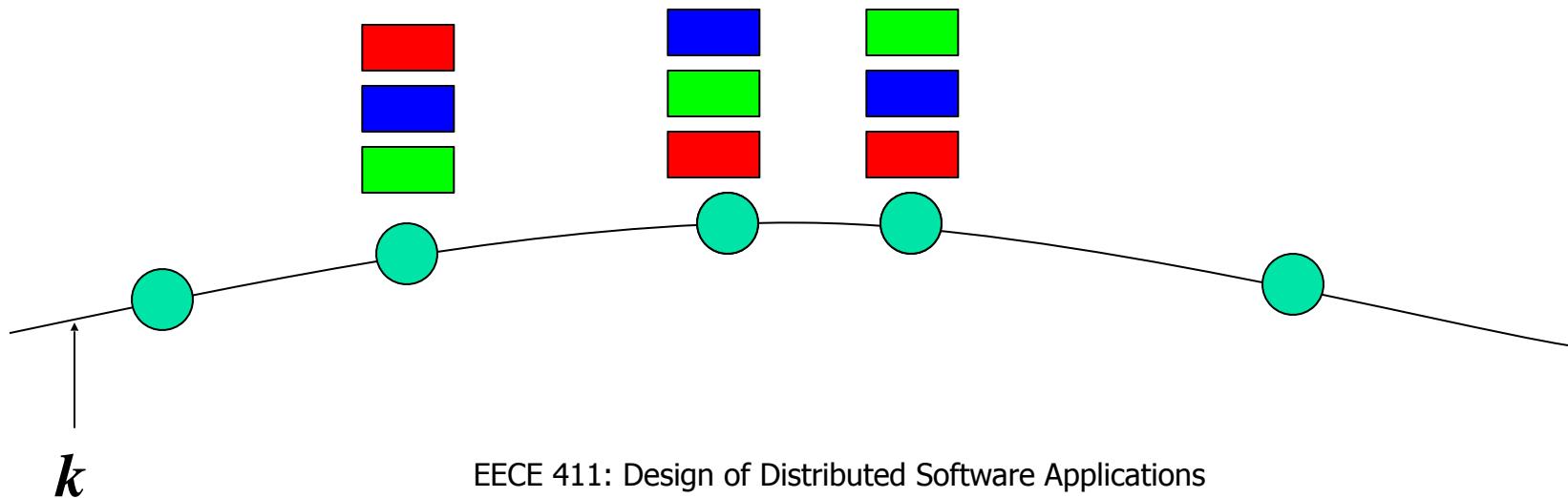
- Multiple replicas to provide durability (and data read availability) ...
 - ... but avoid synchronous replica coordination ...

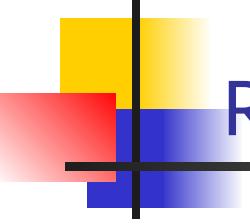
Traditional quorum system:

- R/W: the minimum number of nodes that must participate in a successful read/write operation.
- Problem: the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas.
 - To provide better latency R and W are usually configured to be less than N.
- $R + W > N$ yields a quorum-like system.
 - 'Sloppy quorum' in Dynamo

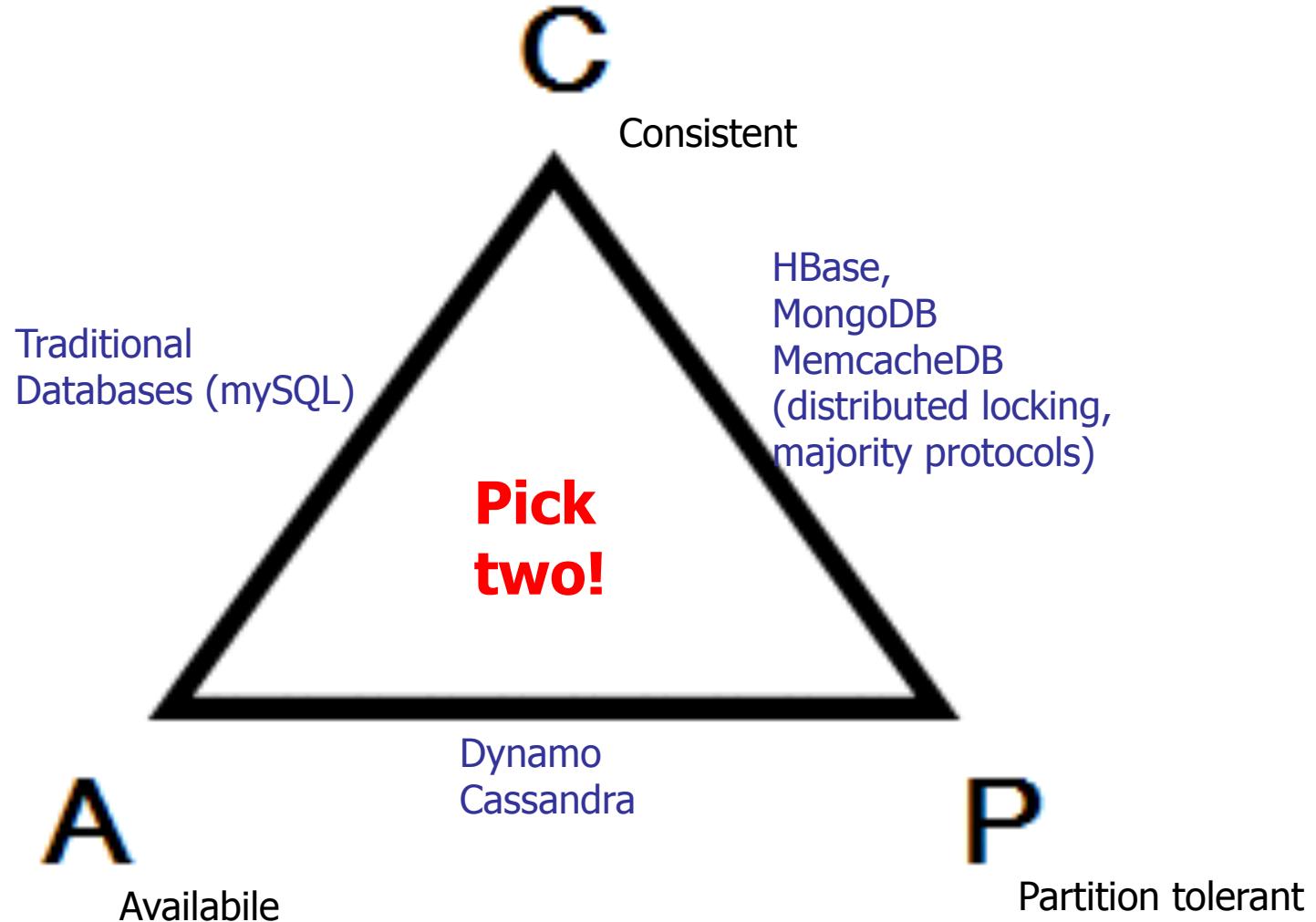
Where replication gets tricky

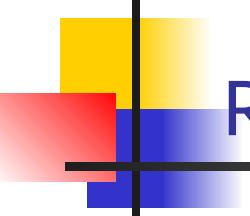
- The claim: Dynamo will replicate each data item on N successors
 - A pair (k, v) is stored by the node closest to k and replicated on N successors of that node
- Why is this hard?
 - Nodes may be slow, fail temporarily ...
 - Solution: sloppy quorum, hinted handoff





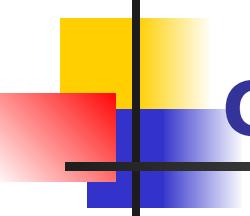
Recap from last time: CAP Theorem





Recap from last time: CAP Theorem

- Conjecture stated by Eric Brewer at the PODC 2000 keynote
- Formally proved by Gilbert and Lynch, 2002
 - “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”. SIGACT News 33(2): 51-59 (2002)
- NB: As with all impossibility results mind the assumptions
 - May do nice stuff with different assumptions



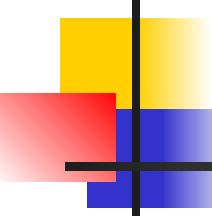
Gilbert/Lynch theorems

Theorem 1: It is impossible in the **asynchronous** network model to implement a read/write data object that guarantees

- Availability
- Atomic consistency

in all executions (including those in which messages are lost)

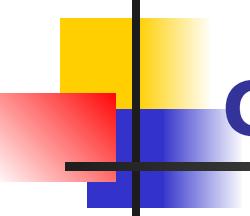
asynchronous networks: no clocks, message delays unbounded



Why Amazon favors availability over consistency?

“even the slightest outage has significant financial consequences and impacts customer trust”

- ... consistency violations may as well have financial consequences and impact customer trust
 - But not in (a majority of) Amazon’s services
 - NB: Billing is a separate story



Gilbert/Lynch theorems

Theorem 2: It is impossible in the **partially synchronous** network model to implement a read/write object store that guarantees

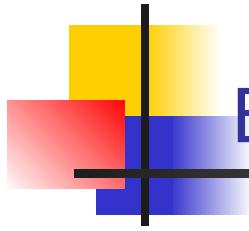
- Availability
- Atomic consistency

in all executions (including those in which messages are lost)

partially synchronous network. Bounds on:

- a) time it takes to deliver messages that are not lost, and
- b) message processing time

exist and are known, but process clocks are not synchronized



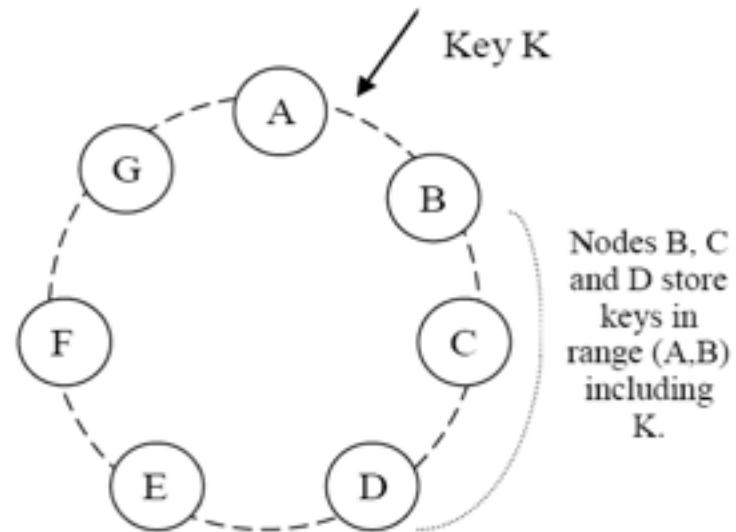
Back to Dynamo design

- Consistent hashing ...

Back to Dynamo....

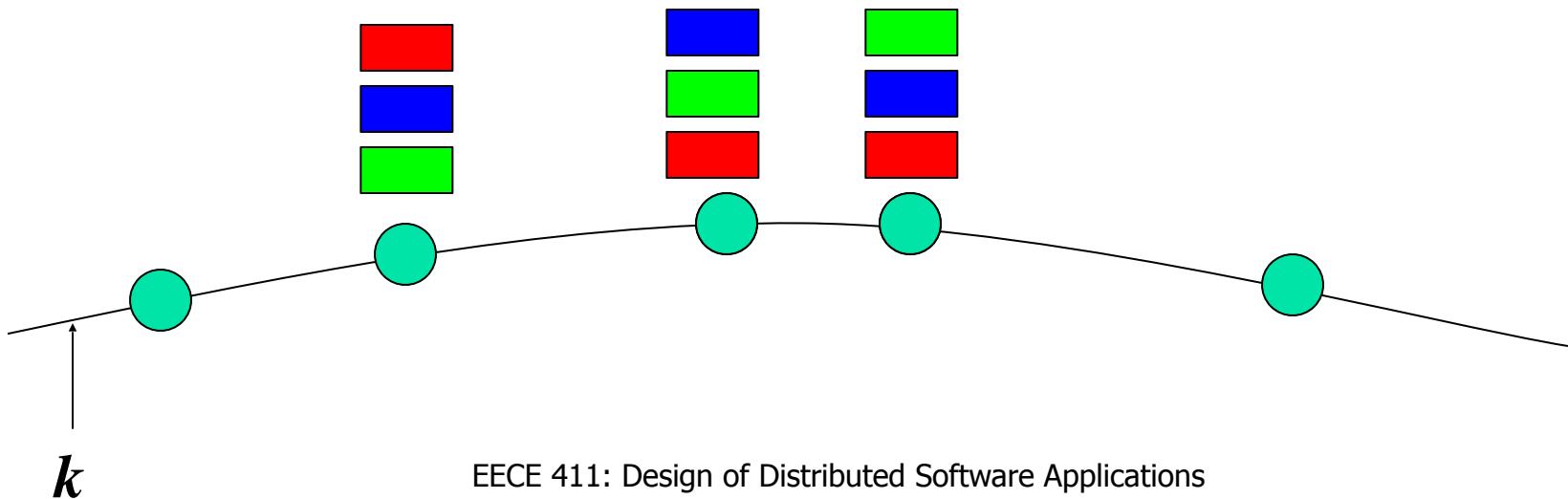
Partition Algorithm: Consistent hashing

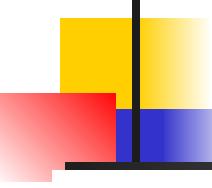
- Each data item is replicated at N hosts (successors)



Where replication gets tricky

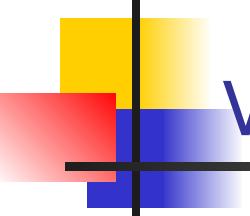
- Dynamo will replicate each data item on N successors
 - A pair (k, v) is stored by the successor closest to k ...
 - ... then node replicates on $N-1$ successors of that node
- Why is this hard?
 - Nodes may be slow, fail temporarily ...
 - Solution: sloppy quorum, hinted handoff





Data versioning

- Multiple replicas ...
 - ... but avoid synchronous replica coordination ...
- The problems this introduces:
 - **when** to resolve possible conflicts?
 - Dynamo's solution: at read time (allows providing an "always writeable" data store)
 - A put() may return before the update has been applied at all the replicas
 - A get() call may return different versions of the same object.
 - **who** should solve them
 - the application → use vector clocks to capture causality ordering between different versions of the same object.
 - the data store → use logical clocks or physical time

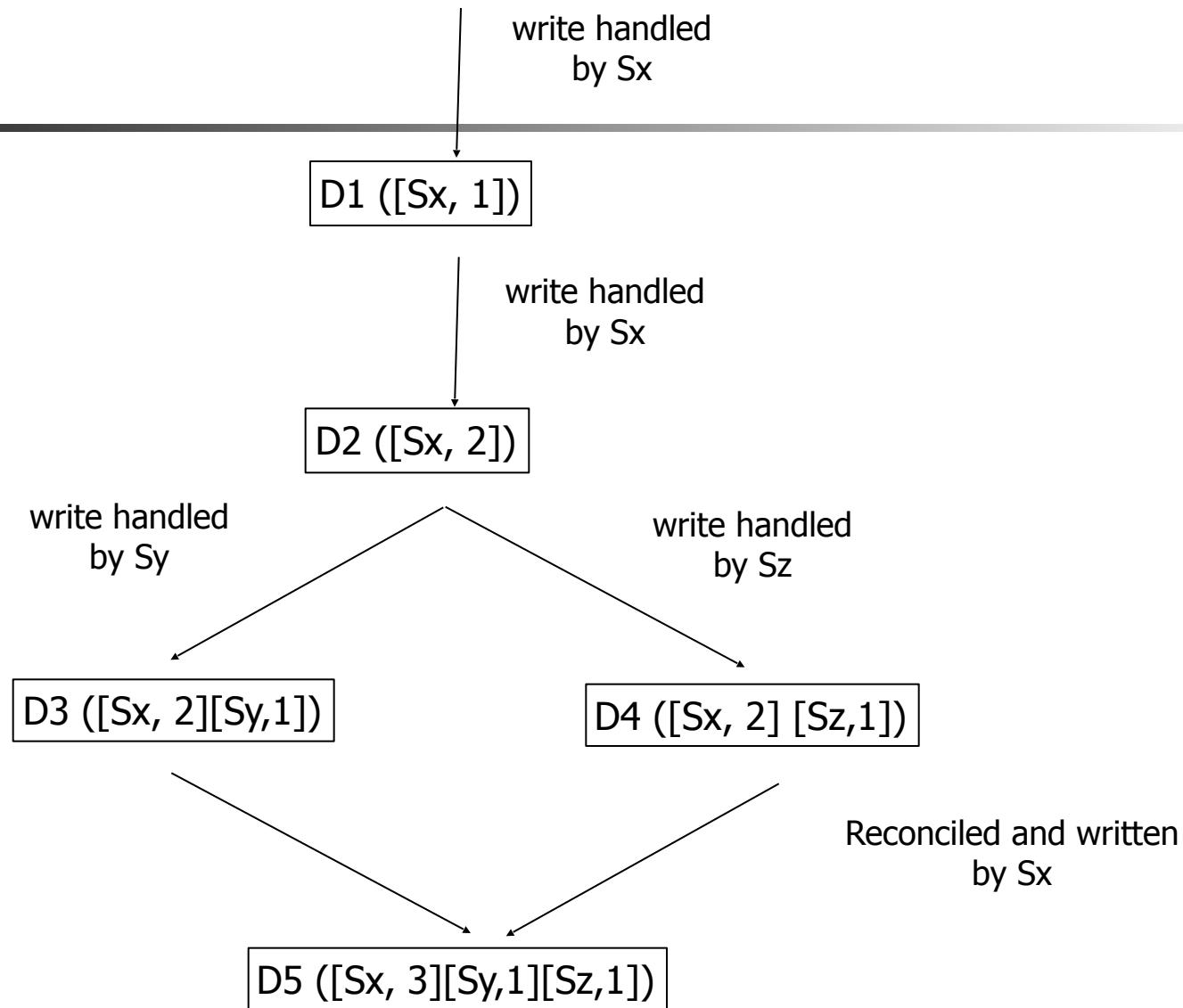


Vector Clocks

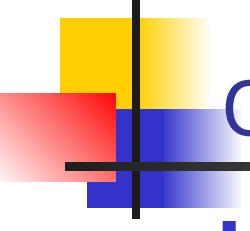
- Each version of each object has one associated vector clock.
 - list of (node, counter) pairs.

Reconciliation:

- If the counters on the first object's clock are less-than-or-equal than all of the nodes in the second clock, then the first is a direct ancestor of the second (and can be ignored).
- Otherwise: application-level reconciliation

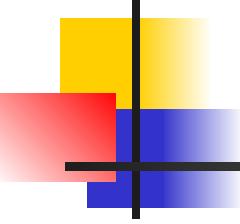


Problem	Technique	Advantage
Partitioning	<ul style="list-style-type: none"> ▪ Consistent hashing 	Incremental scalability, load balancing, etc.
High availability for writes	<ul style="list-style-type: none"> ▪ Eventual consistency ▪ Vector clocks with reconciliation during reads ▪ Quorum protocol 	Availability
Handling temporary failures	<ul style="list-style-type: none"> ▪ 'Sloppy' quorum protocol and hinted handoff 	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	<ul style="list-style-type: none"> ▪ Anti-entropy using Merkle trees 	Synchronizes divergent replicas in the background.
Membership and failure detection	<ul style="list-style-type: none"> ▪ Gossip-based membership protocol 	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.



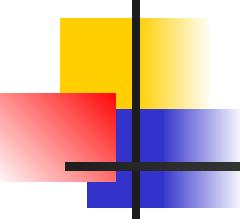
Other techniques

- Node synchronization:
 - Merkle hash tree.



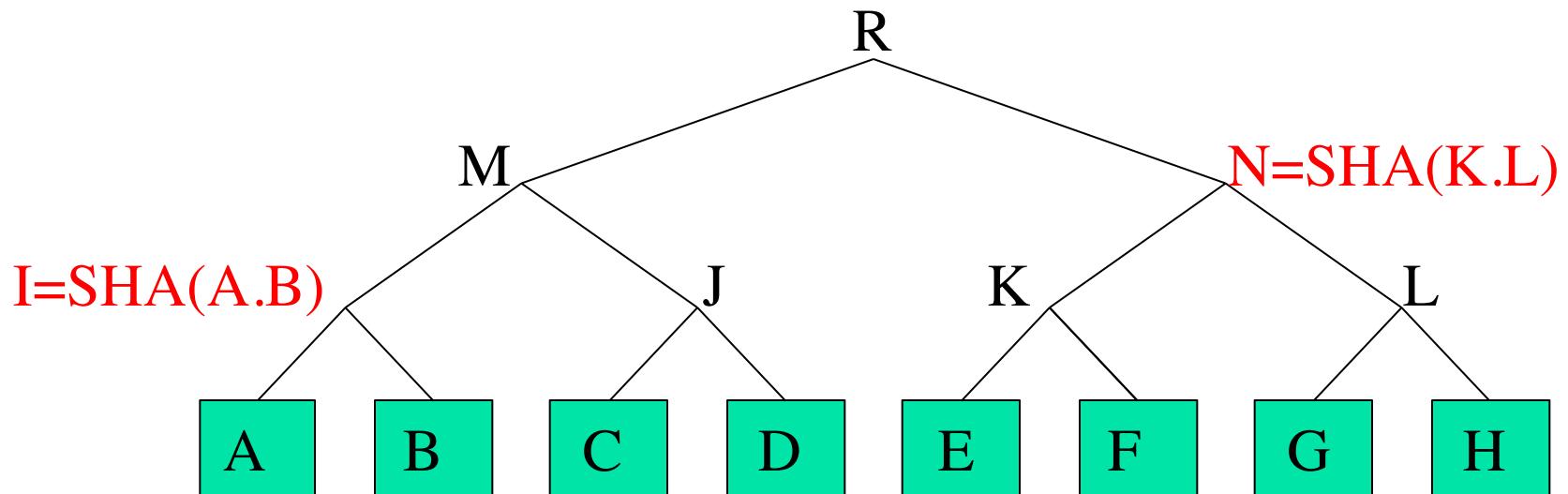
Replication Meets Epidemics

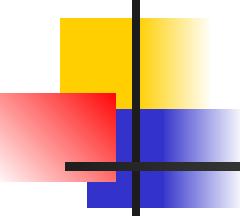
- Candidate Algorithm
 - For each (k,v) stored locally, compute $\text{SHA}(k.v)$
 - Every period, pick a random leaf set neighbor
 - Ask neighbor for all its hashes
 - For each unrecognized hash, ask for key and value
- This is an epidemic algorithm
 - All N members will have all (k,v) in $\log(N)$ periods
 - But (above) the cost is $O(C)$, where C is the size of the set of items stored at the original node



Merkle Trees

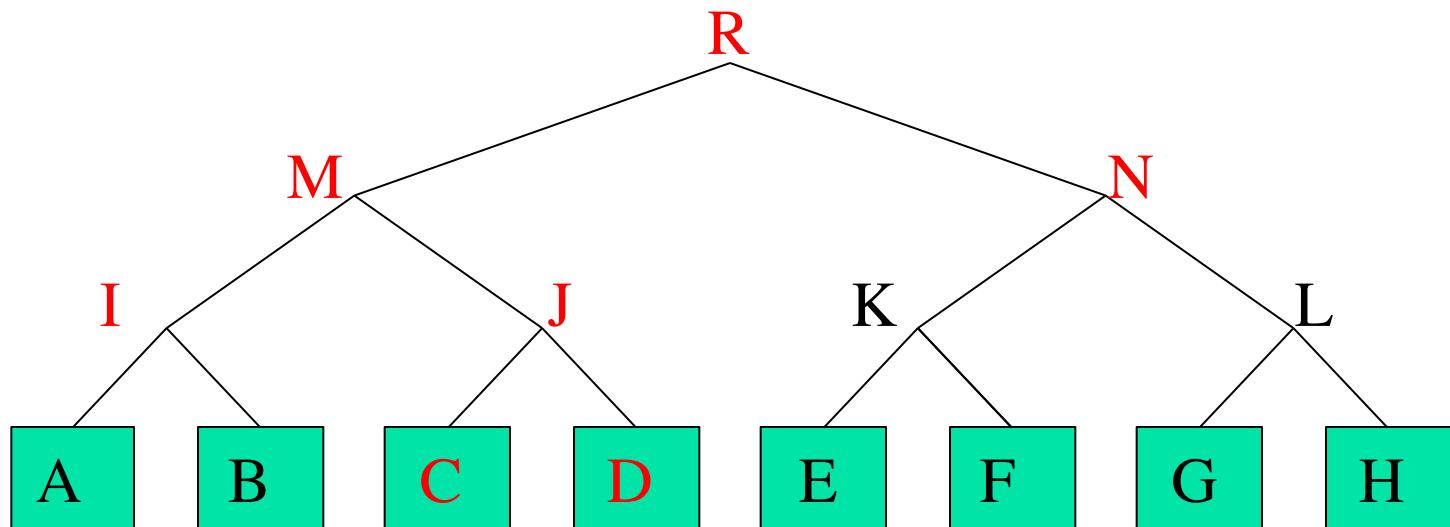
- An efficient summarization technique
 - Interior nodes are the secure hashes of their children
 - E.g., $I = \text{SHA}(A.B)$, $N = \text{SHA}(K.L)$, etc.





Merkle Trees

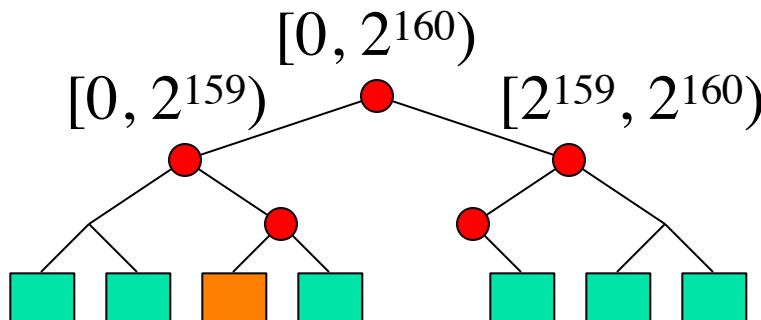
- Merkle trees are an efficient summary technique
 - If the top node is signed and distributed, this signature can later be used to verify any individual block, using only $O(\log n)$ nodes, where $n = \#$ of leaves
 - E.g., to verify block C, need only R, N, I, C, & D



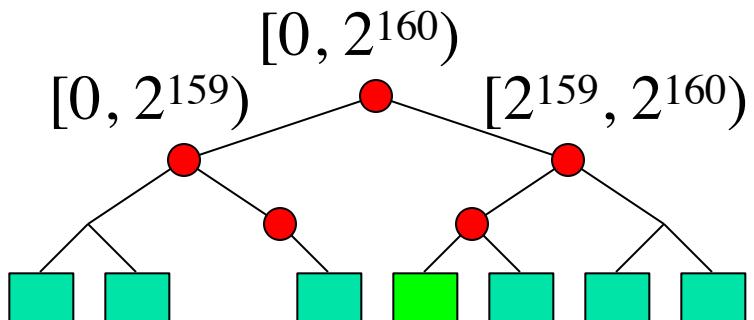
Using Merkle Trees as Summaries

- Improvement: use Merkle tree to summarize keys
 - B gets tree root from A, if same as local root, done
 - Otherwise, recurse down tree to find difference

A's values:



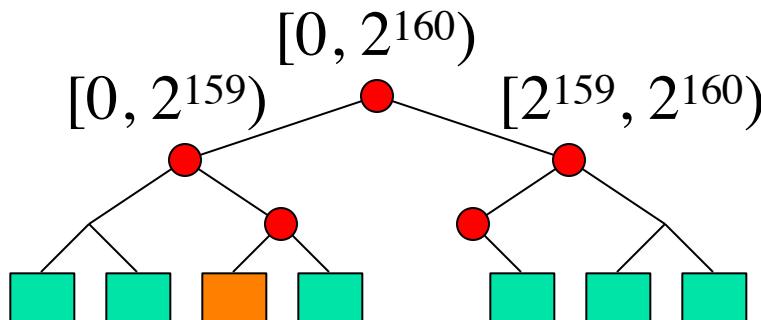
B's values:



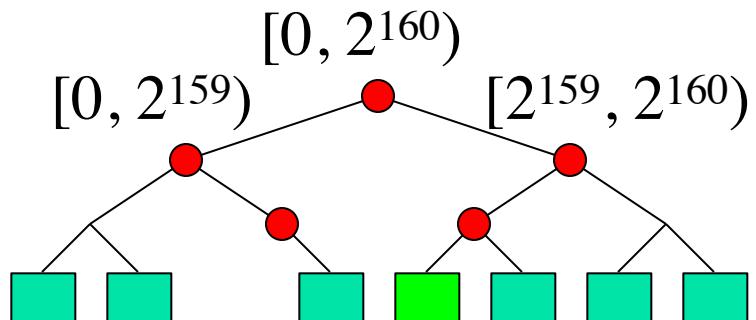
Using Merkle Trees as Summaries

- Improvement: use Merkle tree to summarize keys
 - B gets tree root from A, if same as local root, done
 - Otherwise, recurse down tree to find difference
- New cost is $O(d \log C)$
 - d = number of differences, C = size of disk

A's values:



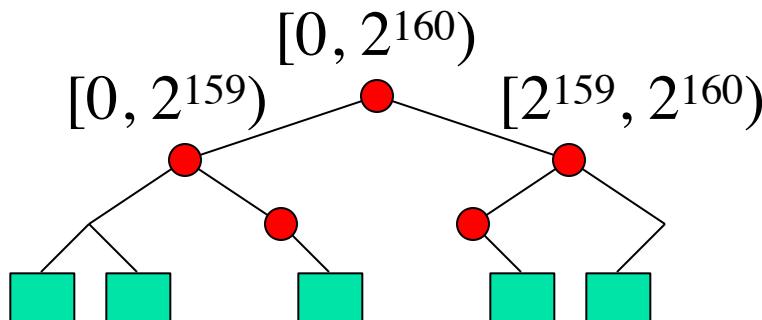
B's values:



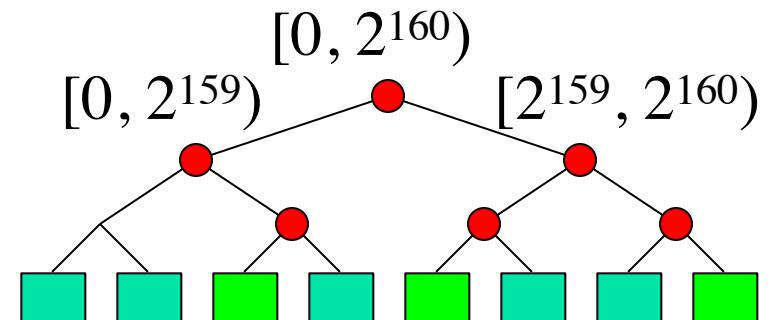
Using Merkle Trees as Summaries

- Still too costly:
 - If A is down for an hour, then comes back, changes will be randomly scattered throughout tree

A's values:



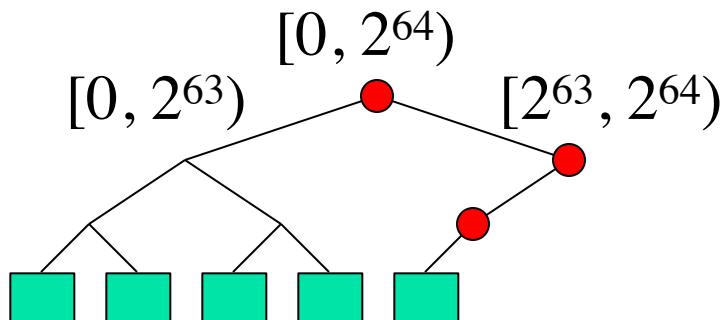
B's values:



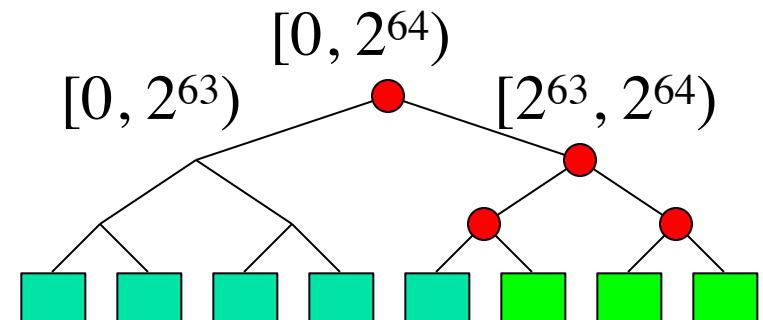
Using Merkle Trees as Summaries

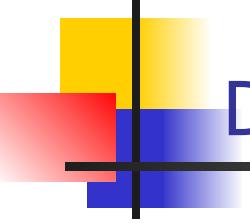
- Still too costly:
 - If A is down for an hour, then comes back, changes will be randomly scattered throughout tree
- Solution: order values by time instead of hash
 - Localizes values to one side of tree

A's values:



B's values:

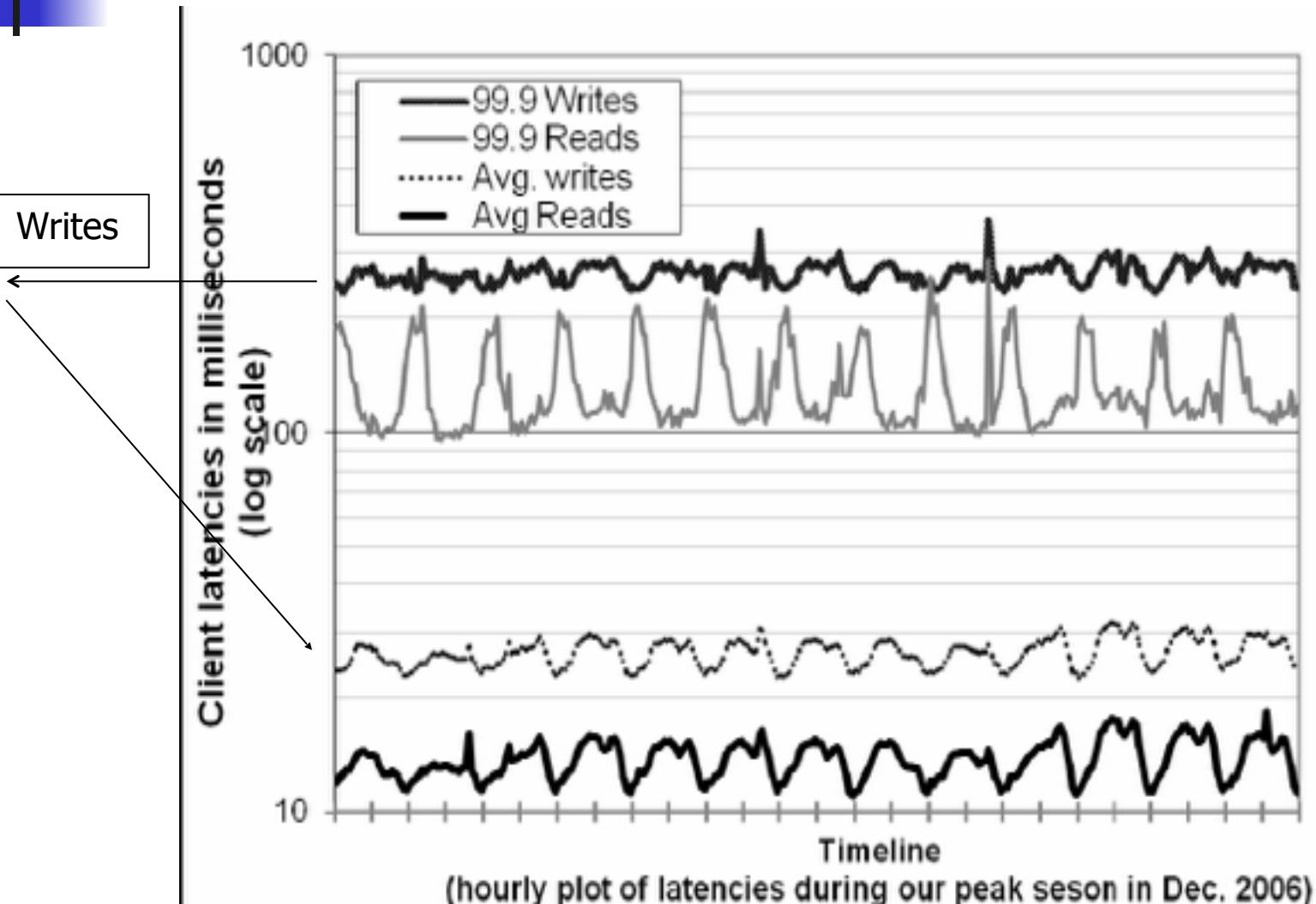




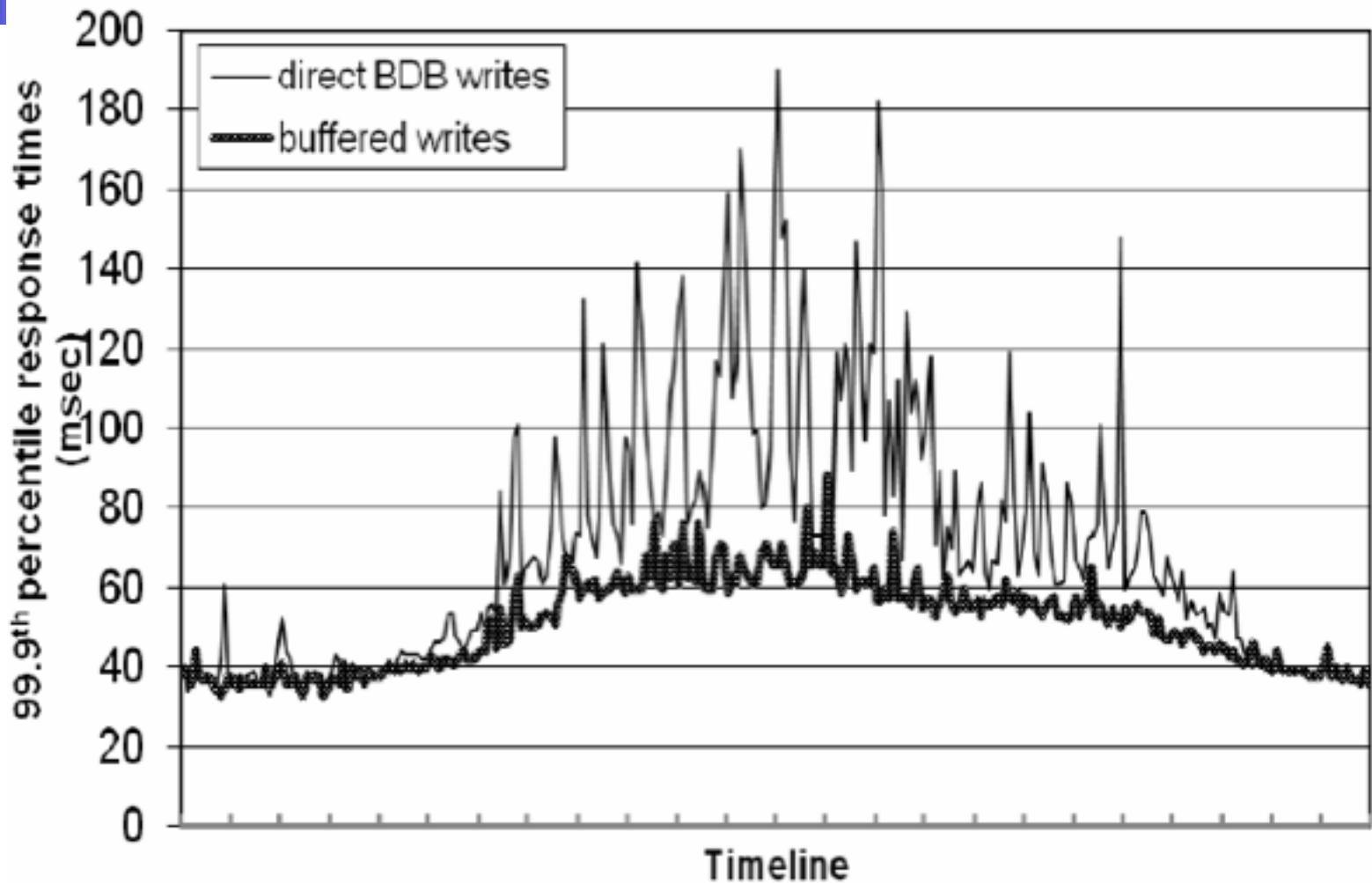
Dynamo Implementation

- Java
 - non-blocking IO
- Local persistence component allows for different storage engines to be plugged in:
 - Berkeley Database (BDB) Transactional Data Store:
object of tens of kilobytes
 - MySQL: larger objects
- Quorum choices (N,W,R) → influence object availability, durability, consistency

Performance evaluation



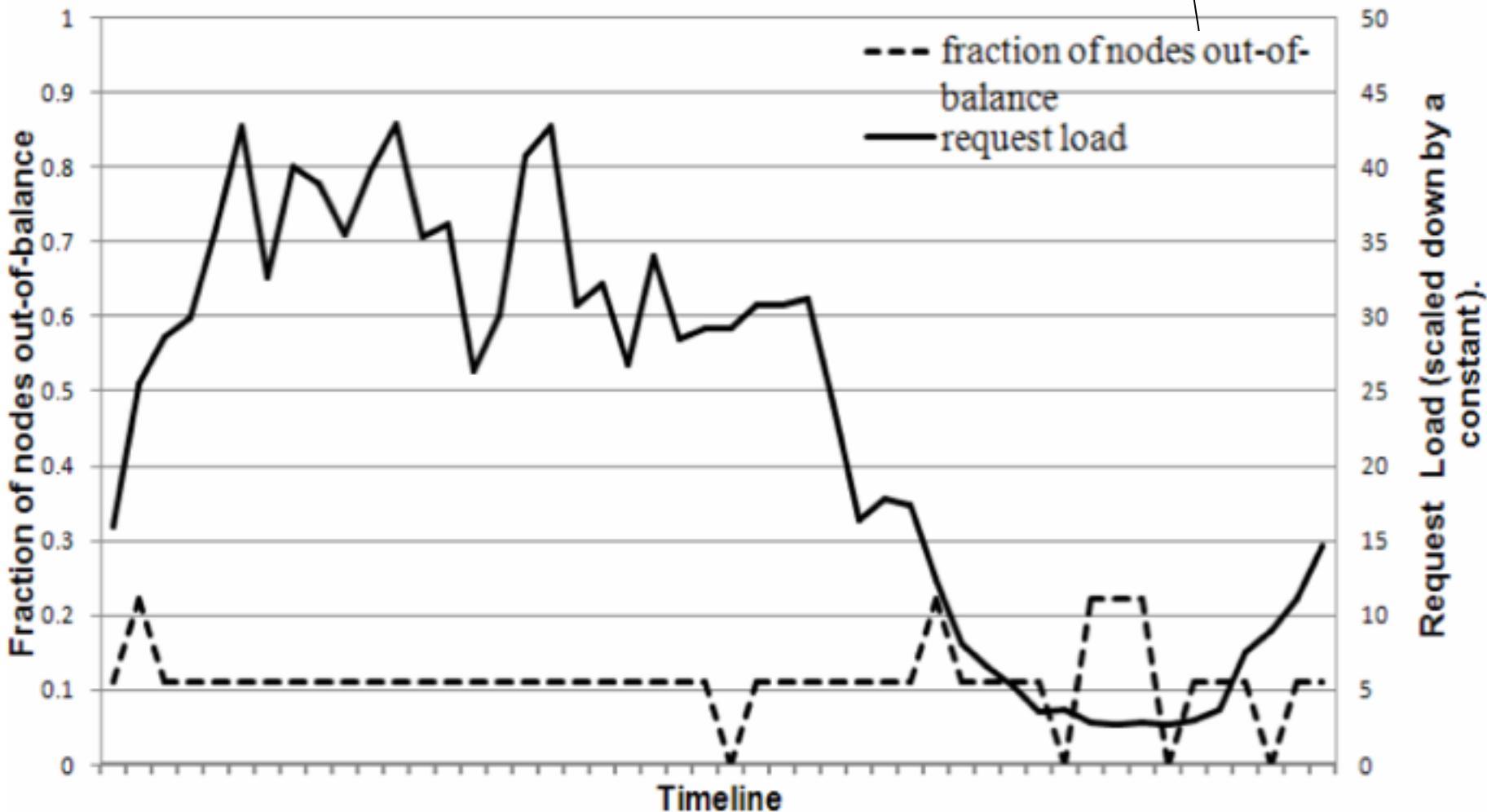
Trading between latency & durability

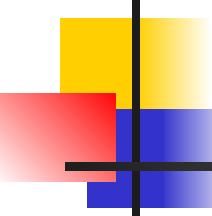


Comparison of performance of 99.9th %-tile latencies for buffered vs. non-buffered writes over 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

Load balance

out-of-balance: nodes with request load above 15% from the average system load





Divergent versions rarely created in practice

1 version → 99.94%

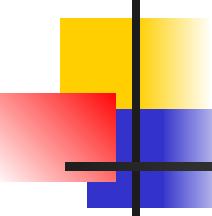
2 versions → 0.0057%

3 versions → 0.00047%

4 versions → 0.00007

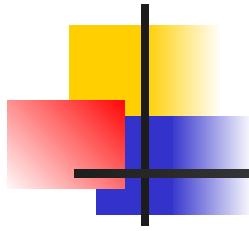
Source: High volume of concurrent writes ... robots?

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	<ul style="list-style-type: none"> ▪ Eventual consistency ▪ Vector clocks with reconciliation during reads 	Version size is decoupled from update rates.
Handling temporary failures	<ul style="list-style-type: none"> ▪ 'Sloppy' quorum and hinted handoff 	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	<ul style="list-style-type: none"> ▪ Anti-entropy using Merkle trees 	Synchronizes divergent replicas in the background.
Membership and failure detection	<ul style="list-style-type: none"> ▪ Gossip-based membership protocol and failure detection. 	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.



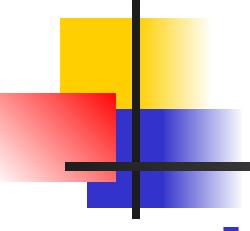
Roadmap

- Clocks can not be perfectly synchronized.
- What can I do in these conditions?
 - Figure out how large is the drift
 - Example: GPS systems
 - Design the system to take drift into account
 - Example: server design to provide at-most-once semantics
 - Do not use physical clocks!
 - Consider only event order
 - (1) Logical clocks (Lamport)
 - But this does not account for causality!
 - (2) Vector clocks!
- Other coordination primitives:
 - Mutual exclusion; leader election



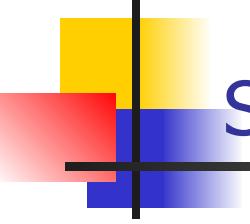
To remember from last time:

Vector clocks



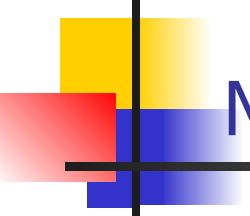
▪ Causality

- So far two usage examples
 - [Causally ordered] Group communication
 - Replica management – determine replica divergence and the causally related stream of updates



So far ...

- Physical clocks
 - Two applications
 - Provide at-most-once semantics
 - Global Positioning Systems
- 'Logical clocks'
 - Where only ordering of events matters
- Other coordination primitives
 - Mutual exclusion
 - Leader election



Mutual exclusion algorithms

Problem: A number of processes in a distributed system want exclusive access to some resource.

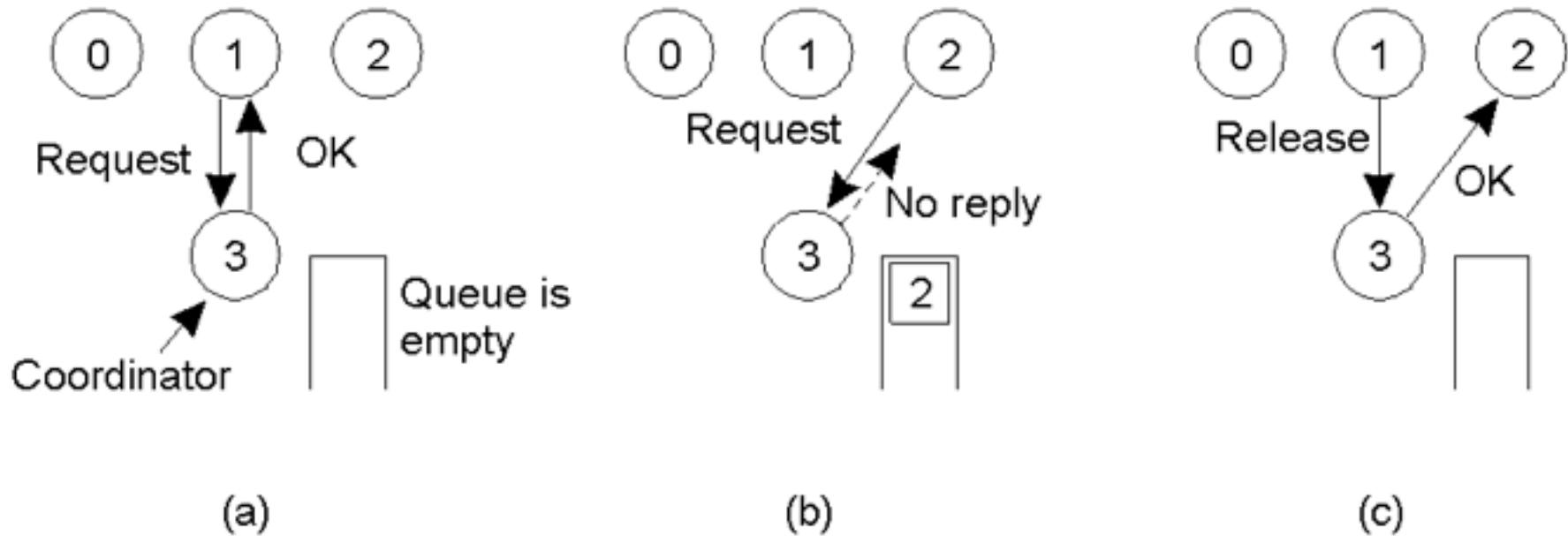
Basic solutions:

- Via a **centralized server**.
- **Multiple/replicated servers:** a voting based protocol
- **Serverless**
 - **Completely distributed**, with no structure imposed.
 - Completely distributed along a **(logical) ring**.

Additional desirable properties:

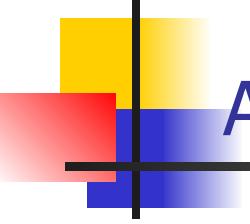
fairness, avoid starvation, ability to deal with failures, low overhead, timeliness

Mutual Exclusion: A Centralized Algorithm



- a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
- b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
- c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

Success measures: fairness, avoid starvation, ability to deal with failures, low overhead, timeliness?



A voting-based protocol

Goal: Deal with server failures

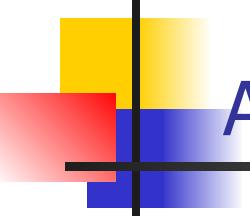
Principle: Voting. Assume the server protecting the resource is replicated n times, we'll call each server replica a **coordinator**

- Client access requires a majority vote from $m > n/2$ coordinators.
- A coordinator always responds immediately to a request.

Failure model: When a coordinator crashes, it will (eventually) recover, but will have forgotten about permissions it had granted.

Correctness: **probabilistic!**

Issue: How robust is this system? What is the probability to make an incorrect 'grant access' decision?



A voting-based protocol (cont)

Principle: Assume n coordinators

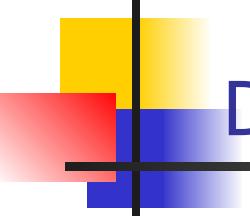
- Access requires a majority vote from $m > n/2$ coordinators.
- A coordinator always responds immediately to a request.

Issue: How robust is this system?

p the probability that a coordinator resets in the next Δt (crashes and recovers immediately)

- $p = \Delta t / T$, where T is the average coordinator lifetime

Quiz—like question: what's the probability of malfunctioning (i.e., to violate mutual exclusion requirement and allow two clients in the critical region)?



Decentralized Mutual Exclusion (cont)

Principle: Assume n coordinator

- Access requires a majority vote from $m > n/2$ coordinators.
- A coordinator always responds immediately to a request.

Issue: How robust is this system?

- p the probability that a coordinator resets in the next Δt (crashes and recovers immediately)
 - $p = \Delta t / T$, where T is the average peer lifetime
- The probability that k out m coordinators reset during Δt
 $P[k] = C(k, m)p^k(1-p)^{m-k}$:
- Violation when at least $2m-n$ coordinators reset

$$P[\text{violation}] = p_v = \sum_{k=2m-n}^n \binom{m}{k} p^k (1-p)^{m-k}$$

EECI With $p = 0.001, n = 32, m = 0.75n, p_v < 10^{-40}$

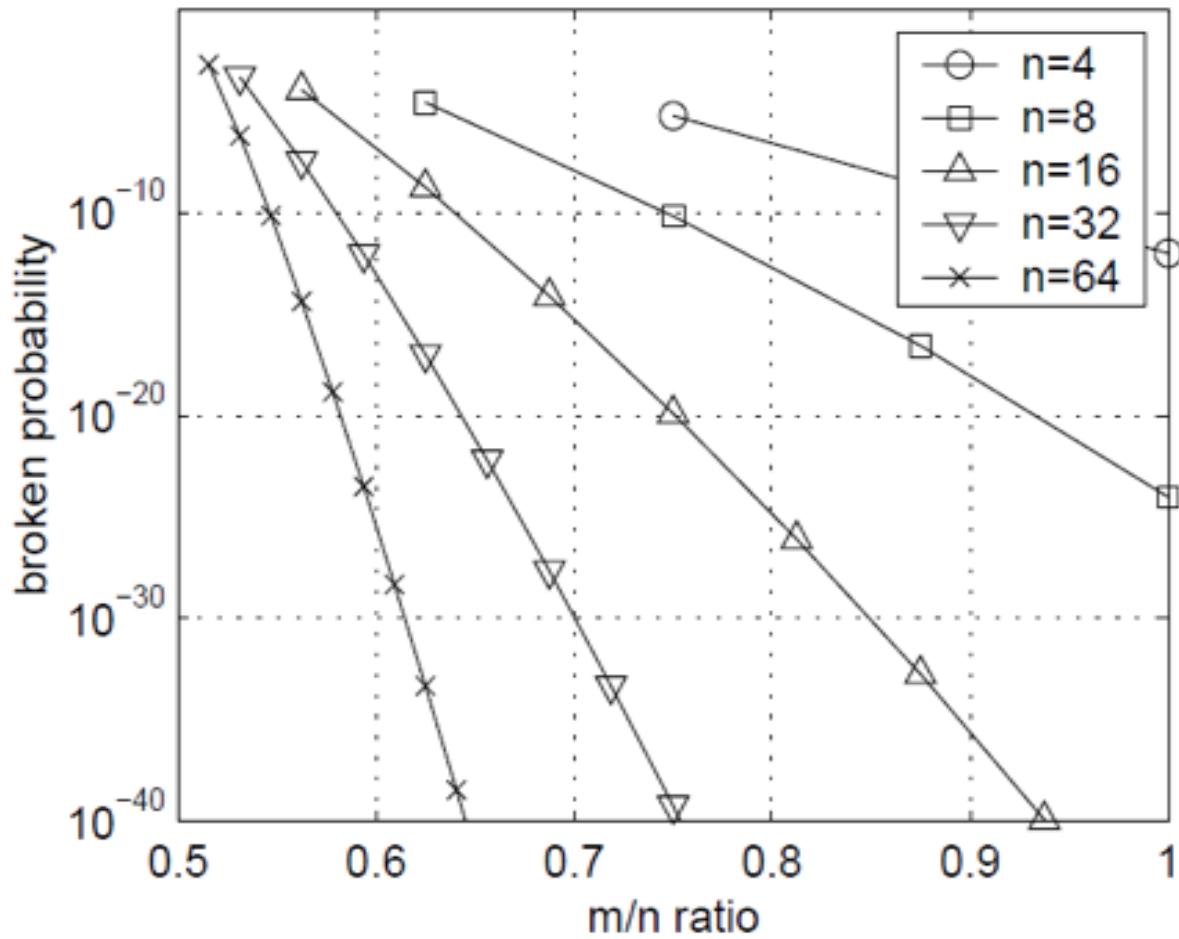


Figure 2. Probability to break exclusivity.

Issues

- Any possible issues with the design?
 - Deadlock?
 - Low efficiency?
 - Fairness?

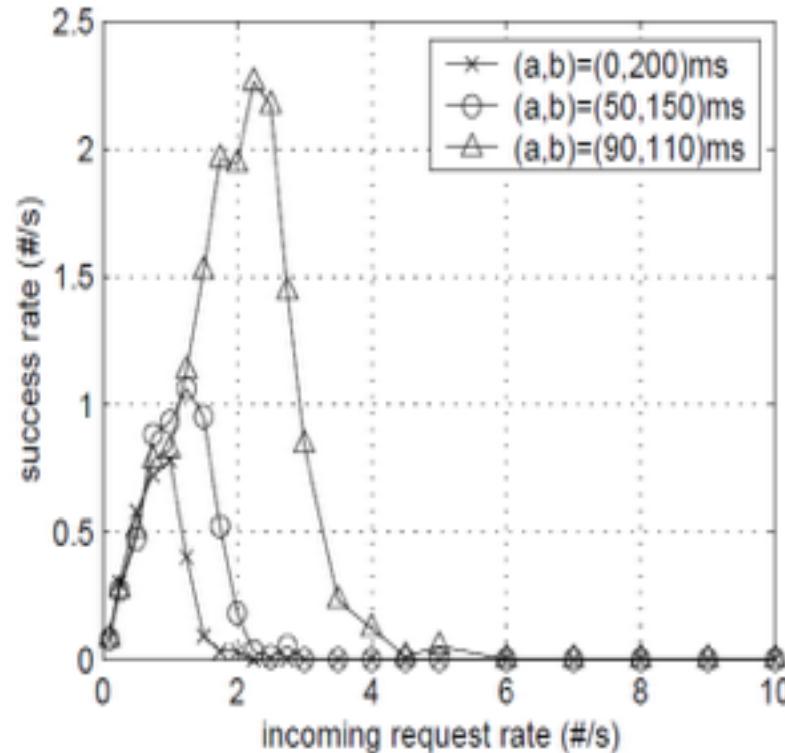
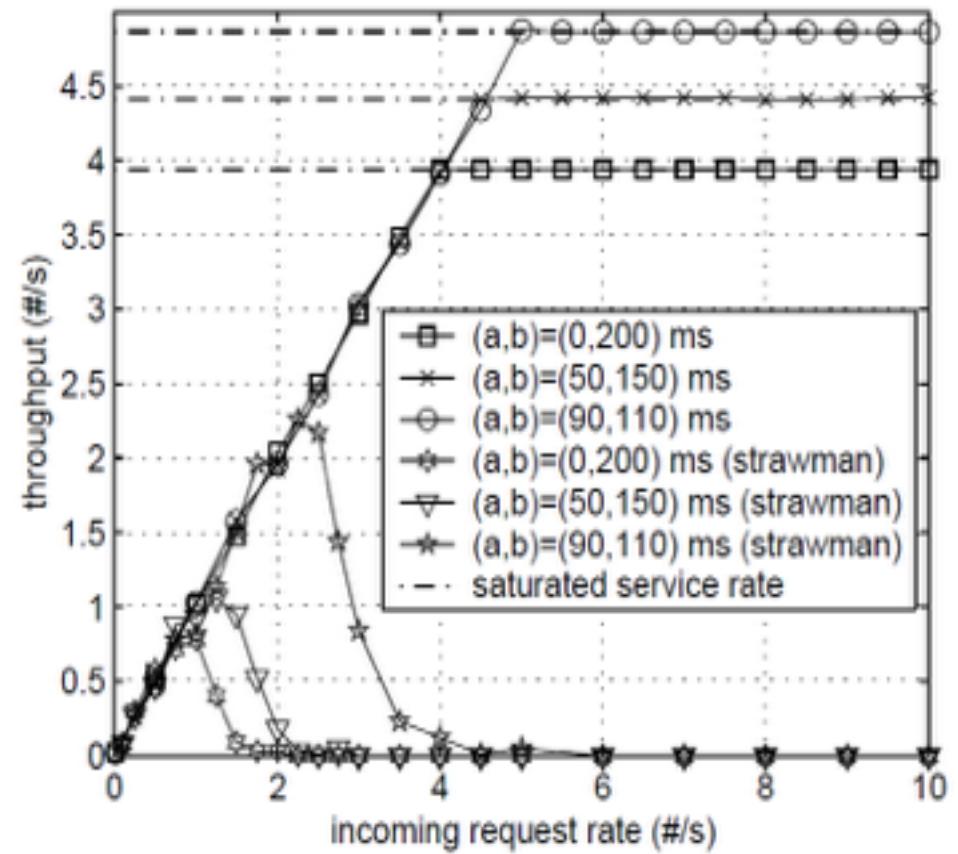
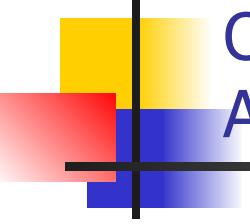


Figure 3. Performance of strawman protocol, with latency uniformly distributed in (a, b) .

Performance issue – starvation





Other solutions for mutual exclusion: A Distributed Algorithm (Ricart & Agrawala)

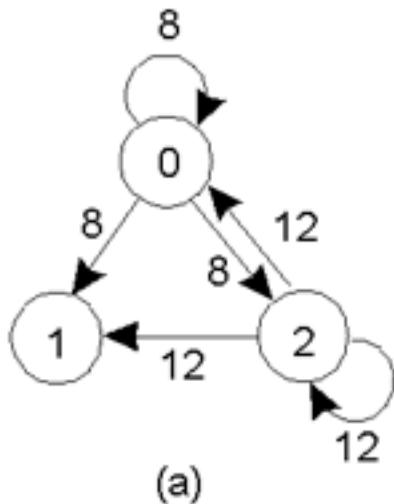
Setup: serverless – clients collaborate to decide who should enter critical region.

Idea: Protocol similar ordered group communication except that acknowledgments aren't sent.

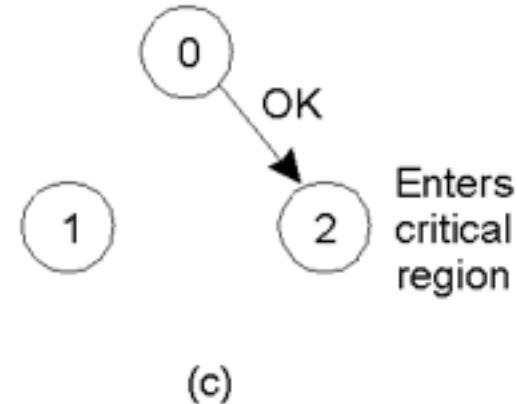
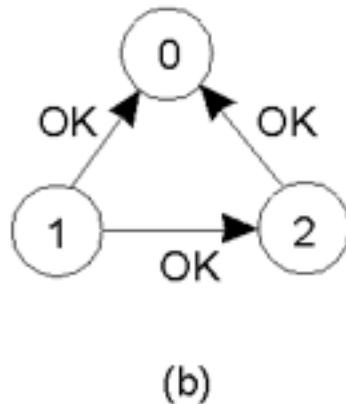
Instead, replies (i.e. grants to entry critical region) are sent only when:

- the receiving process has no interest in the shared resource; or
 - the receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).
-
- In all other cases, reply is **deferred**
 - (results in some more local administration)

Mutual Exclusion: A Distributed Algorithm (II)

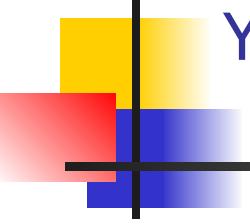


Enters
critical
region



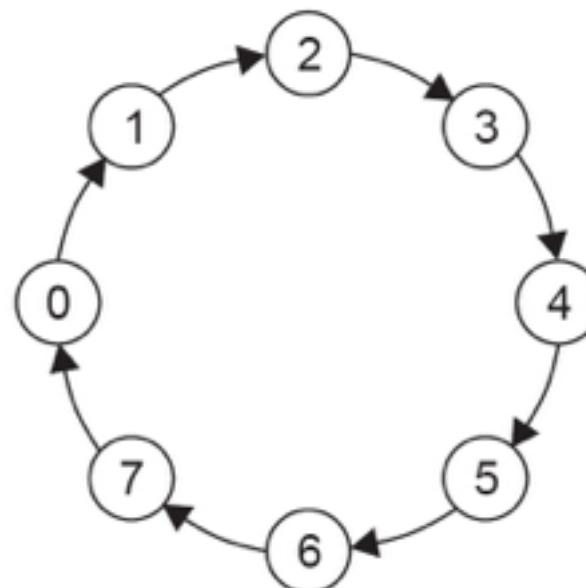
Enters
critical
region

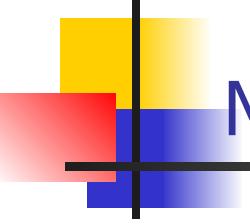
- a) Two processes (0 and 2) want to enter the same critical region at the same moment.
- b) Process 0 has the lowest timestamp, so it wins.
- c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.



Yet one more solution: A Token Ring Algorithm

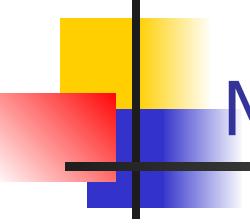
Principle: Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to)





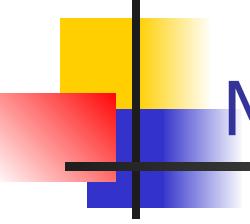
Mutual Exclusion: Algorithm Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized			Coordinator crash
Voting			Starvation, low efficiency
Totally Distributed			Crash of any process
Token ring			Lost token, process crash



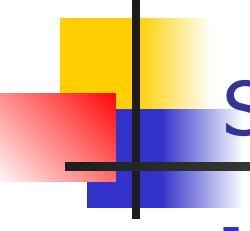
Mutual Exclusion: Algorithm Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3		Coordinator crash
Voting	$3mk$ (k=number of attempts)		Starvation, low efficiency
Totally Distributed	$2(n-1)$		Crash of any process
Token ring	$1..\infty$		Lost token, process crash



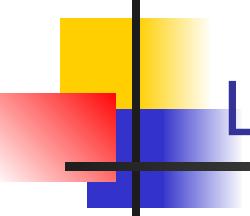
Mutual Exclusion: Algorithm Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Voting	$3mk$ (k=number of attempts)	2m	Starvation, low efficiency
Totally Distributed	$2(n-1)$	$2(n-1)$	Crash of any process
Token ring	$1..\infty$	0 to n-1	Lost token, process crash



So far ...

- **Physical clocks**
 - Two applications
 - Provide at-most-once semantics
 - Global Positioning Systems
- **'Logical clocks'**
 - Where only ordering of events matters
- **Other coordination primitives**
 - Mutual exclusion
 - Leader election: How do I choose a coordinator?

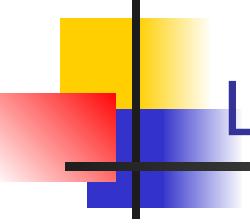


Leader election algorithms

Context: An algorithm requires that some process acts as a coordinator.

Question: how to select this special process **dynamically**.

Note: In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions single point of failure.



Leader election algorithms

Context: Each process has an associated priority (weight).
The process with the highest priority needs to be elected as the coordinator.

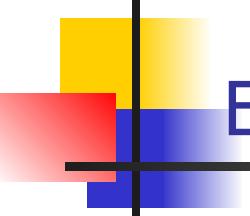
Issue: How do we find the 'heaviest' process?

Two important **assumptions**:

- Processes are uniquely identifiable
- All processes know the identity of all participating processes

Traditional algorithm examples

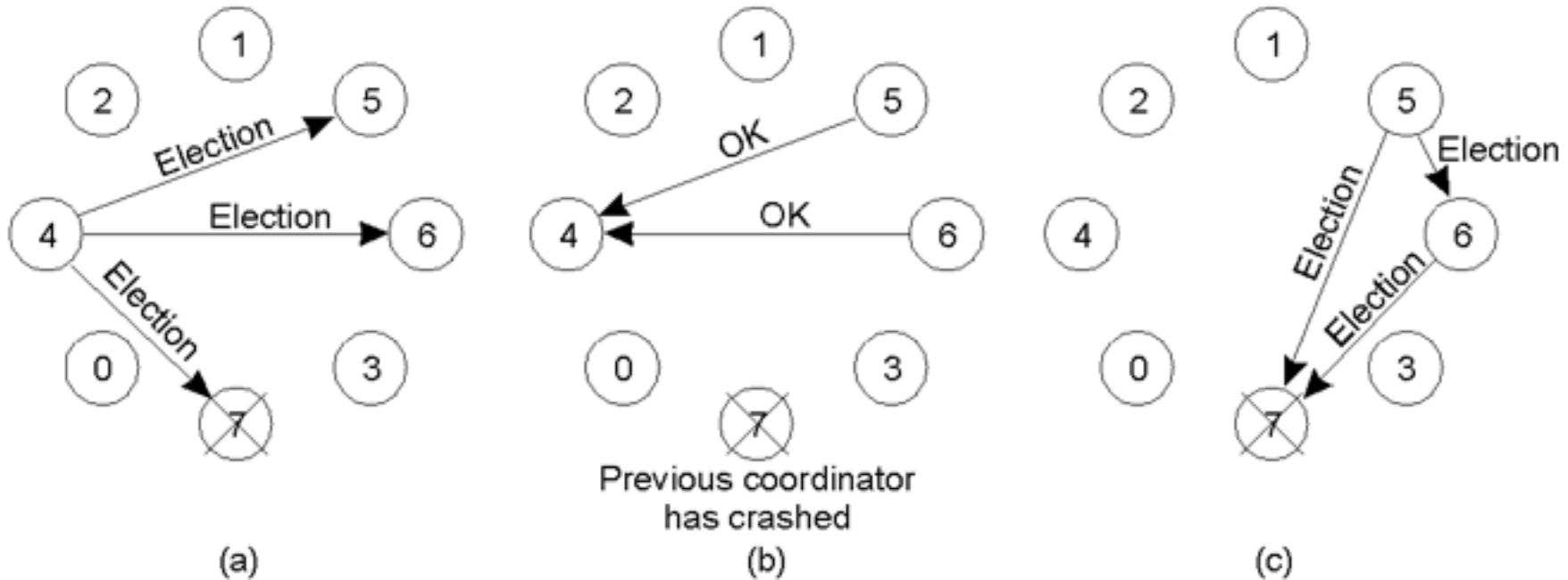
- The bully algorithm
- Ring based algorithm



Election by Bullying

- Any process can just start an election by sending an election message to all other (heavier) processes
- If a process P^{heavy} receives an election message from a lighter process P^{light} , it sends a take-over message to P^{light} . P^{light} is out of the race.
- If a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

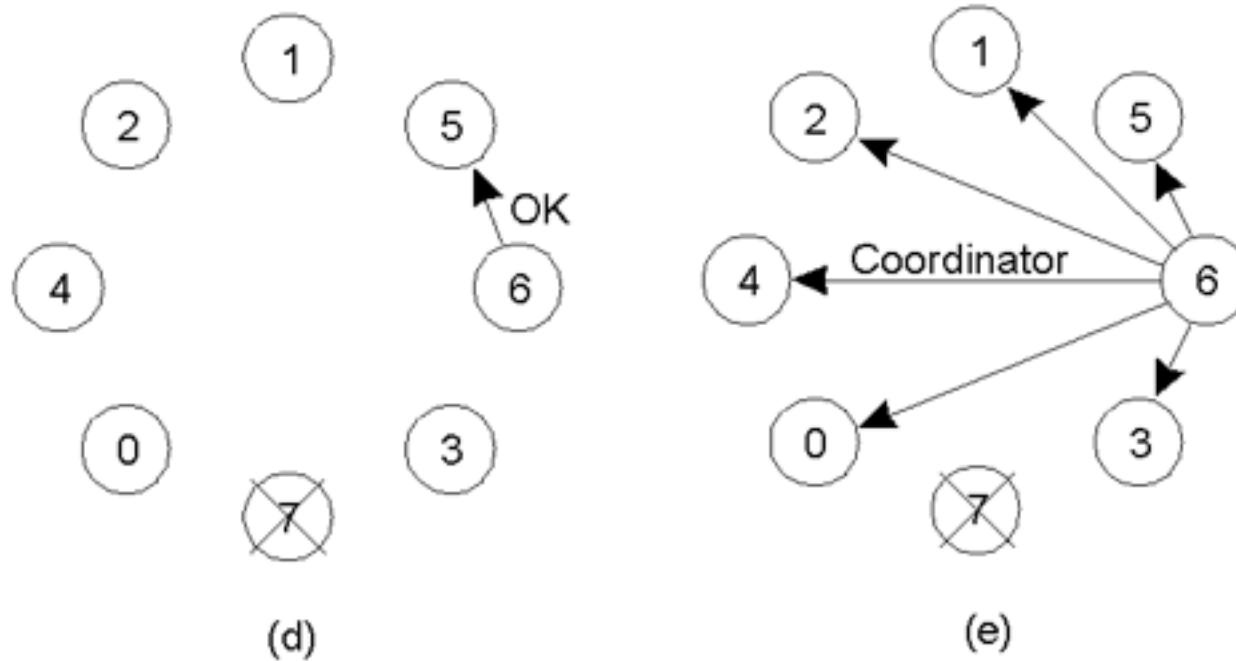
The Bully Algorithm

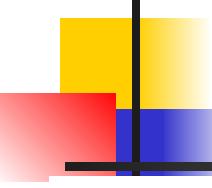


- Process 4 detects 7 has failed and holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election (also send message to 7 as they have not detected 7 failure)

The Bully Algorithm (2)

- d) Process 6 tells 5 to stop
- e) Process 6 wins and announces itself everyone



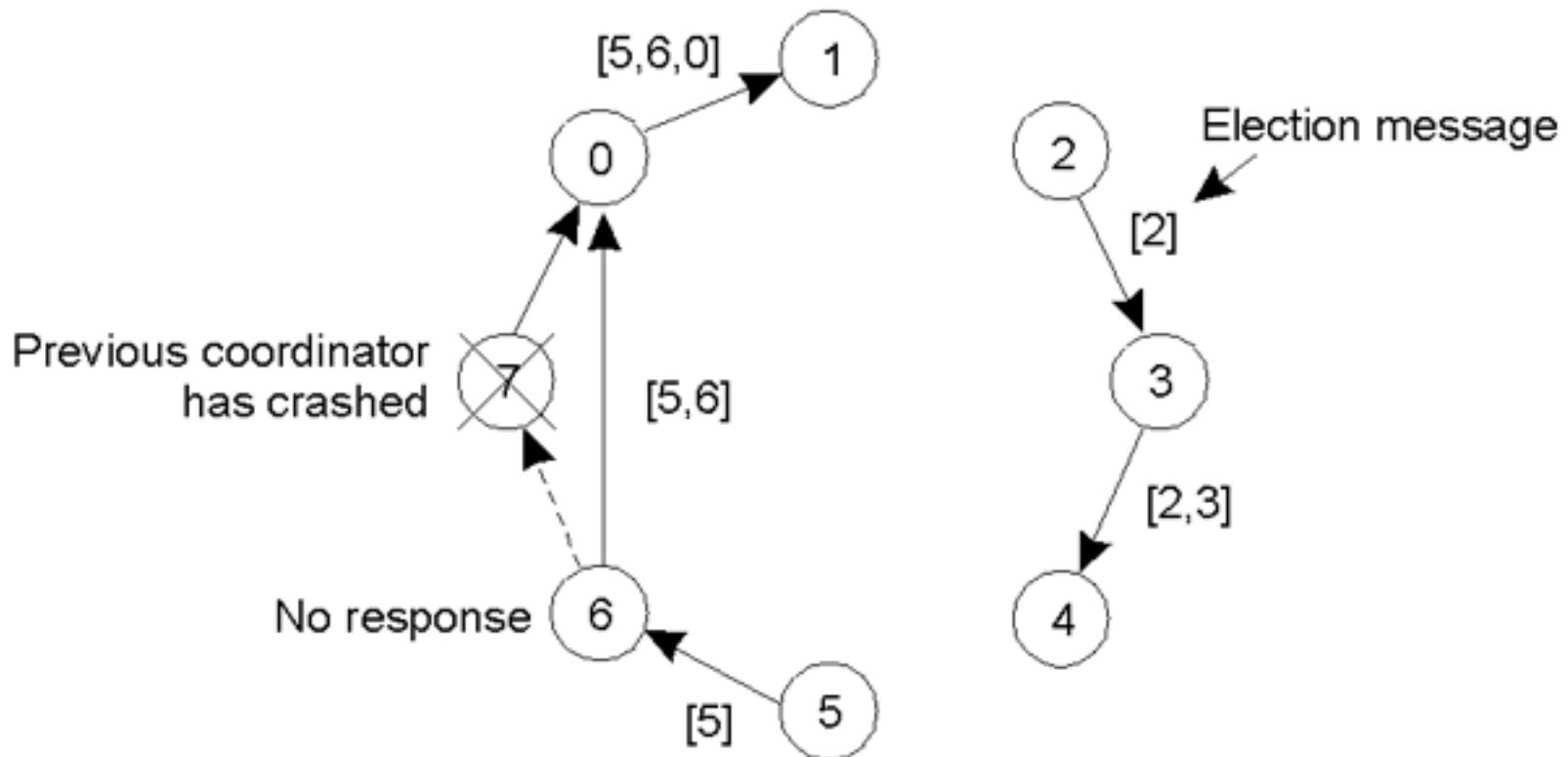


Election in a Ring

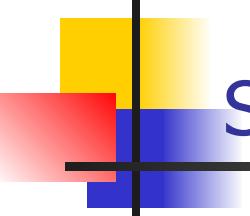
Principle: Organize processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

The Ring Algorithm



- **Question:** What happens if two processes initiate an election at the same time? Does it matter?
- **Question:** What happens if a process crashes during the election?



Summary so far ...

A distributed system is:

- a collection of **independent computers** that appears to its users as a **single coherent system**

Components need to:

- Communicate
 - Point to point: sockets, RPC/RMI
 - Point to multipoint: multicast, epidemic
- Cooperate
 - Naming to enable some resource sharing
 - Naming systems for flat (unstructured) namespaces: consistent hashing, DHTs
 - Naming systems for structured namespaces: EECE456 for DNS
 - **Synchronization: physical clocks, logical clocks, mutual exclusion, leader election**

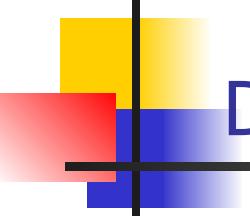


Replication & Consistency

Replication: Creating and using multiple copies of data (or services)

Why replicate?

- Improve system reliability
 - Prevent data loss
 - i.e. increase data durability
 - Increase data availability
 - Note: availability \neq durability
 - Increase confidence: e.g. deal with byzantine failures
- Improve performance
 - Scaling throughput
 - Reduce access times



Data vs. control replication

Data replication

- Web site mirrors, browser caches, DNS

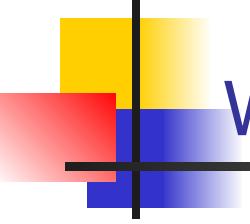
Control replication

- stateless services, e.g., Web server dealing with web-page layout, services in application tier in a 3-tier architecture

Data and control replication

- Stateful services, e.g., critical infrastructure service

We'll look mostly at
this one for the



What are the issues?

Issue 1. Dealing with data changes

- **Consistency models**
 - **What** is the semantic the system implements
 - (luckily) applications do not always require strict consistency
- **Consistency protocols**
 - **How** to implement the semantic agreed upon?

Issue 2. Replica management

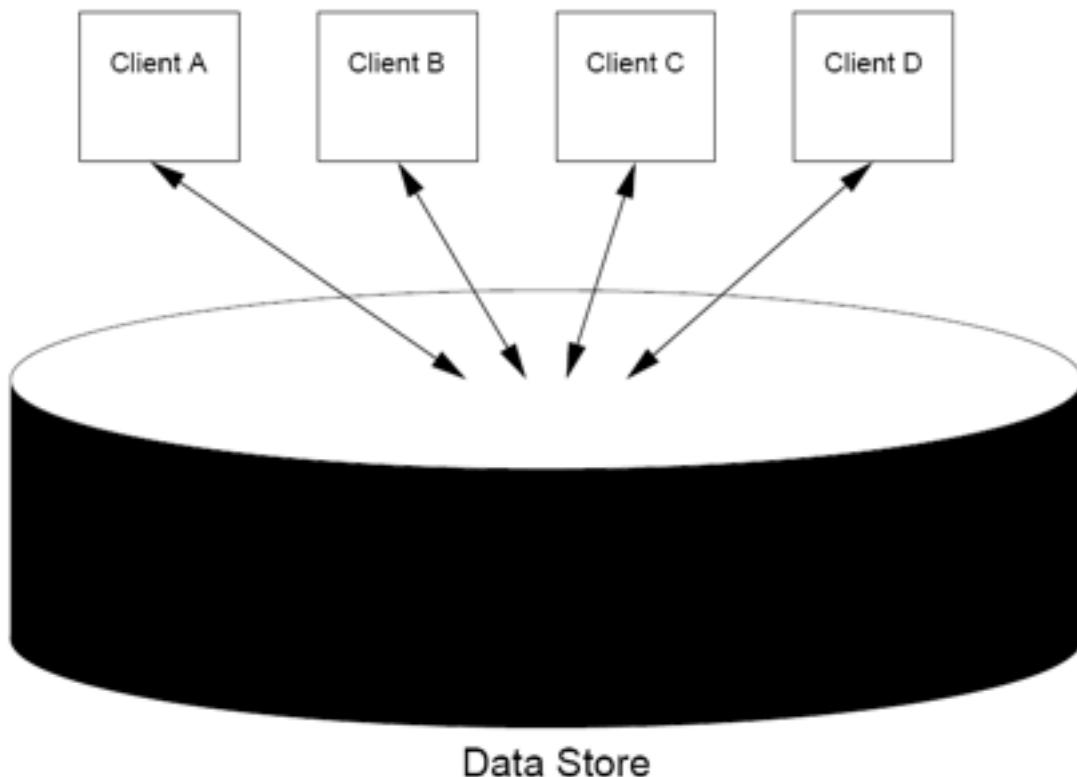
- How many replicas?
- Where to place them?
- When to get rid of them?

Issue 3. Redirection/Routing

- Which replica should clients use?

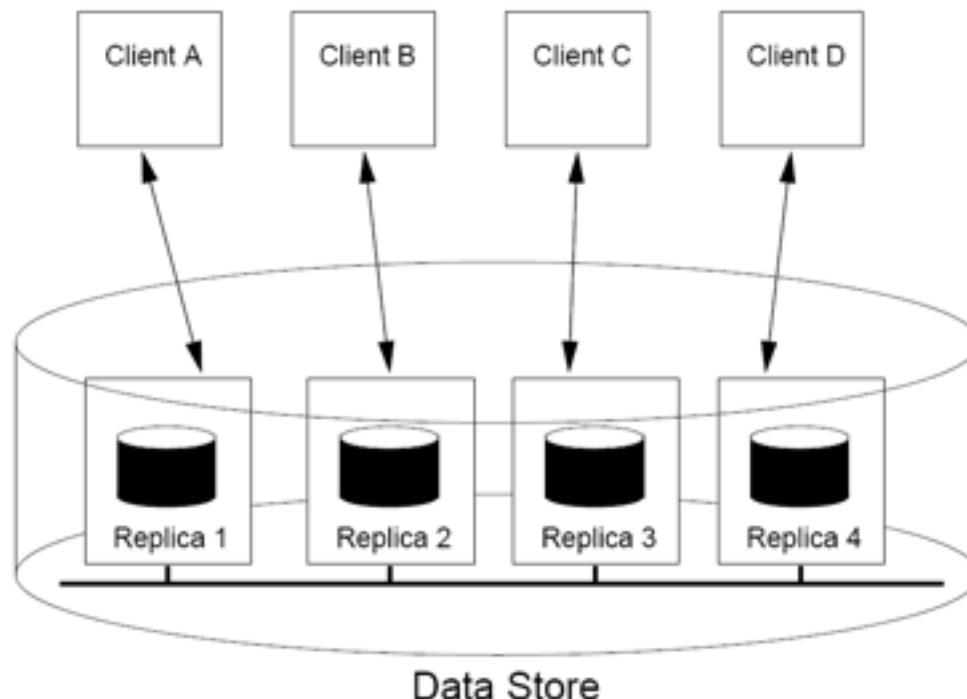
Client's view of the data-store

Ideally: black box -- complete 'transparency' over how data is stored and managed



Management' system view on data store

Management system: Controls the allocated resources and aims to provide transparency



Scalability ← TENSION → Management overheads

To keep replicas **consistent**, we generally need to ensure that all **conflicting** operations are done in the same order everywhere

Conflicting operations: From the world of transactions:

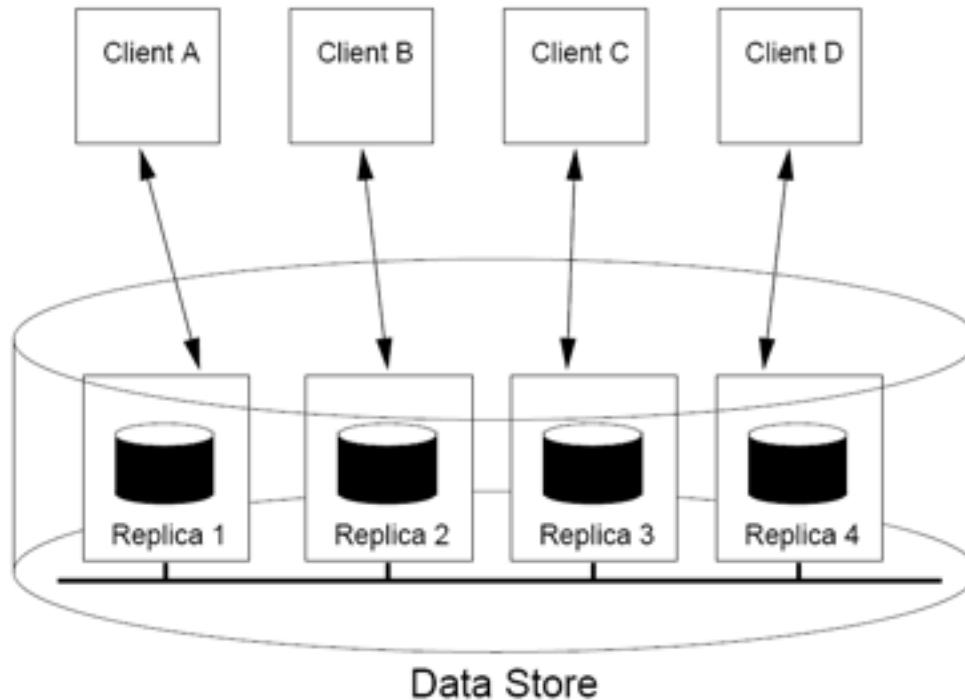
- **Read–write conflict:** a read operation and a write operation act concurrently
- **Write–write conflict:** two concurrent write operations

Problem: Guaranteeing global ordering on conflicting operations may be costly, reducing scalability

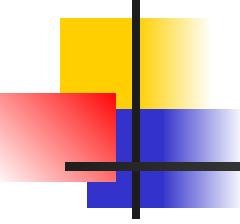
Solution: Weaken consistency requirements so that hopefully global synchronization can be avoided

Consistency model

Management system: Controls the allocated resources and aims to **provide transparency**



Consistency model: Contract between the data store and the clients: The data store specifies the results of read and write operations in the presence of concurrent operations.

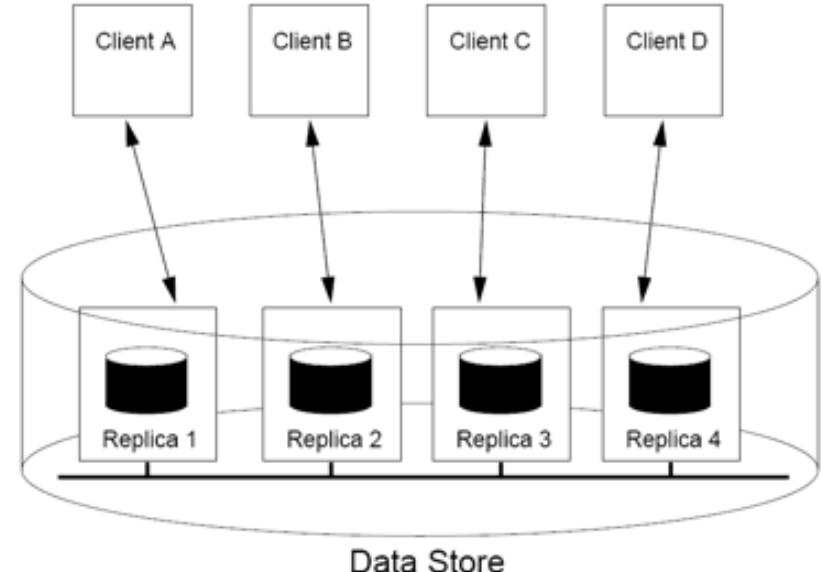


Roadmap for next few classes

- **Consistency models:**
 - contracts between the data store and the clients that specify the results of read and write operations are in the presence of concurrency.
- Protocols
 - **To manage update propagation**
 - To manage replicas:
 - creation, placement, deletion
 - To assign client requests to replicas

Consistency models

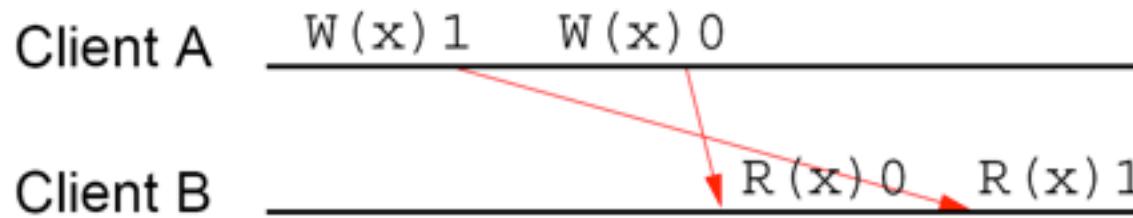
- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Continuous consistency
 - Limit the deviation between replicas
 - Eventual consistency
- **Client centric**
 - Assume client-independent views of the datastore
 - Constraints on operation ordering for each client independently



Notations

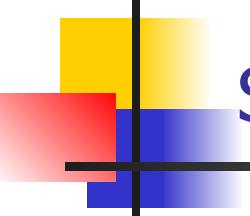
For operations on a Data Store:

- Read: $R_i(x) a$ -- client **i** reads **a** from location **x**
- Write: $W_i(x) b$ -- client **i** writes **b** at location **x**



P1: $W(x)1$

P2: $R(x)NIL$ $R(x)1$



Sequential Consistency

The result of **any** execution is the same as if :

- operations by **all processes** on the **entire 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.

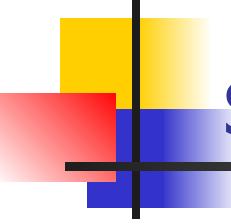
Note: we talk about interleaved execution – **there is some total ordering for all operations** taken together

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

(a)

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

(b)



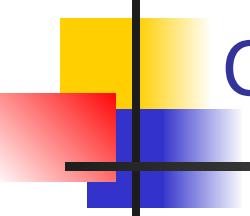
Sequential Consistency (example)

Process1:	Process 2	Process 3
$X \leftarrow 1$	$Y \leftarrow 1$	$Z \leftarrow 1$
Read (Y, Z)	Read (X, Z)	Read (X, Y)
print (Y, Z)	print (X, Z)	print (X, Y)

Is output: 00 00 00 permitted?

What about: 11 11 11?

What about: 00 10 10?



Causal Consistency (1)

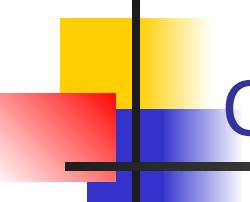
Writes that are causally related must be seen by all processes in the same order.

Concurrent writes may be seen in a different order by different processes

“Causally related” relationship (notation ‘ \rightarrow ’):

- A read is causally related to the write that provided the data the read got.
- A write is causally related to a read that happened before this write in the same process.
- If $\text{write1} \rightarrow \text{read}$, and $\text{read} \rightarrow \text{write2}$, then $\text{write1} \rightarrow \text{write2}$.

Concurrent \Leftrightarrow not causally related



Causal Consistency (Example)

P1: $W(x)a$

$W(x)c$

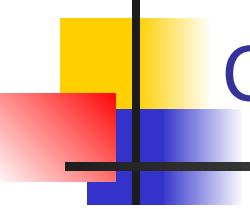
P2: $R(x)a$ $W(x)b$

P3: $R(x)a$ $R(x)c$ $R(x)b$

P4: $R(x)a$ $R(x)b$ $R(x)c$

Note: $W_1(x)a \rightarrow W_2(x)b$, but not $W_2(x)b \rightarrow W_1(x)c$

- Is this sequence allowed with a causally-consistent store?
- Is this sequence allowed with sequentially consistent store?



Causal Consistency: (More Examples)

P1: $W(x)a$

P2: $R(x)a$ $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

SecC: no

CauC: no

P1: $W(x)a$

P2: $W(x)b$

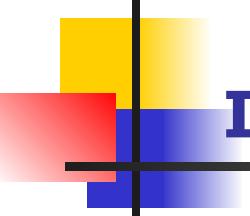
P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

SecC: no

CauC: yes

- Which sequence is allowed with a causally-consistent store?
- Which sequence allowed with sequentially consistent store?



Increasing granularity: Grouping Operations

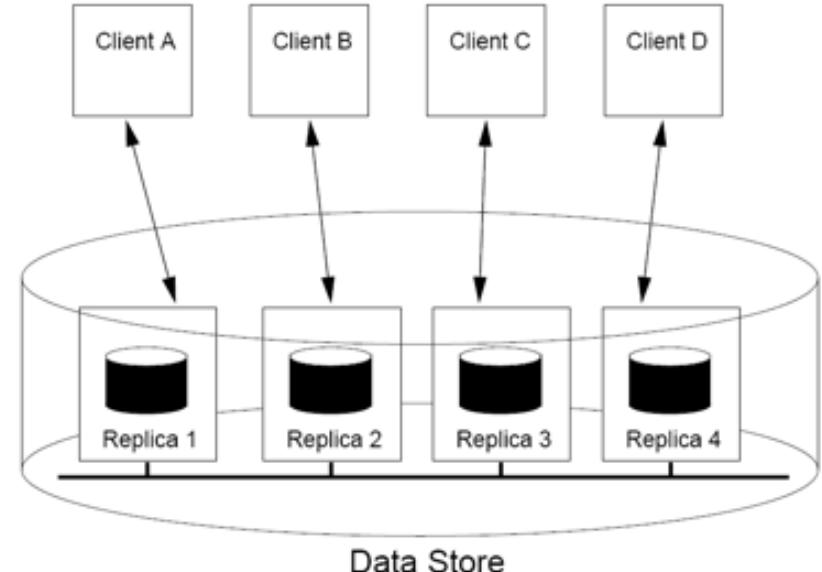
Basic idea: In reality applications do not care that ALL reads and writes are immediately known to other processes. They just care about the **effect of the series**.

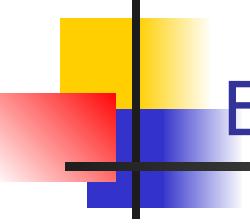
One Solution:

- Introduce '**synchronization variables**'. Read/write access
- Accesses to **synchronization variables** are sequentially consistent.
- No access to a synchronization variable is allowed until all previous writes have completed everywhere.
- No read data access is allowed until all previous accesses to synchronization variables have been performed.

Consistency models

- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Eventual consistency (best effort)
 - Continuous consistency
 - Limit the deviation between replicas
- **Client centric**
 - Assume client-independent views of the datastore
 - Constraints on operation ordering for each client independently





Eventual Consistency

Idea: If no updates take place for a long enough period time, all replicas will gradually (i.e., eventually) become consistent.

No constraint on operation ordering

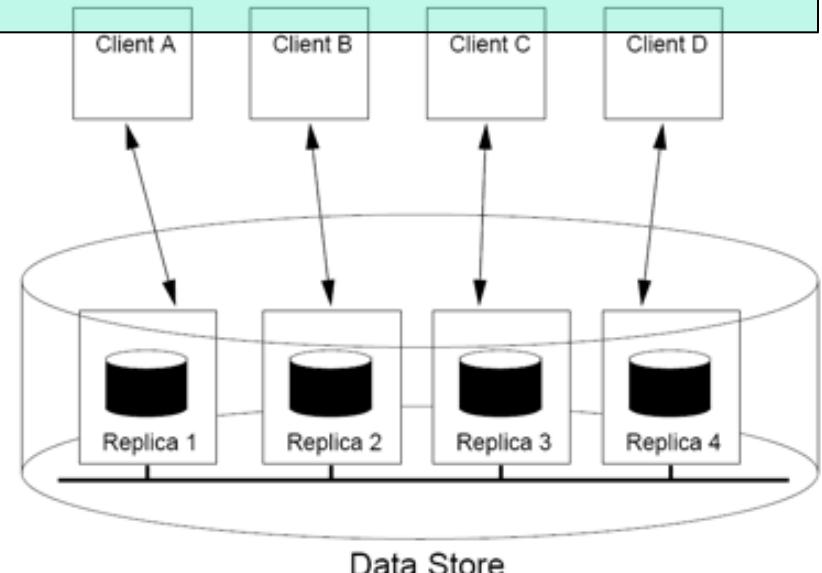
When does this work well?

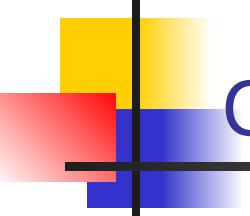
- Mostly read-only workloads, and
- No concurrent updates (e.g., all updates performed through a master replica)

Advantages/Drawbacks?

Consistency models

- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Eventual consistency
 - Continuous consistency
 - Limit the deviation between replicas
- Client centric
 - Assume client-independent views of the datastore
 - Constraints on operation ordering for each client independently





Continuous Consistency

Obs1: We can talk about a degree of consistency

- Goal: Limit the **deviation** between replicas

Obs2: Multiple metrics to measure deviation are possible

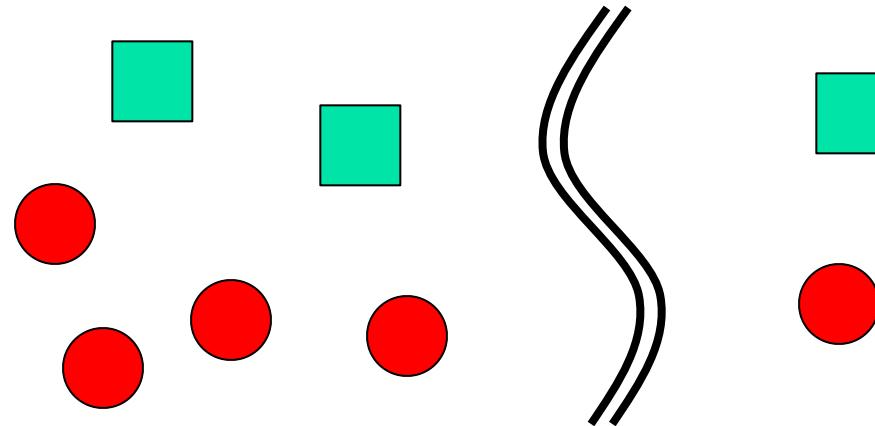
- replicas may differ in their **numerical value**
- replicas may differ in their relative **staleness**
- replicas may differ with respect to (number and order) of **performed update operations**

conit: consistency unit → specifies the data unit over which consistency is to be enforced.

Limitations of Consistency Mechanisms Based on Request Ordering

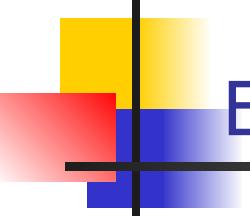
: Replicas

: Clients



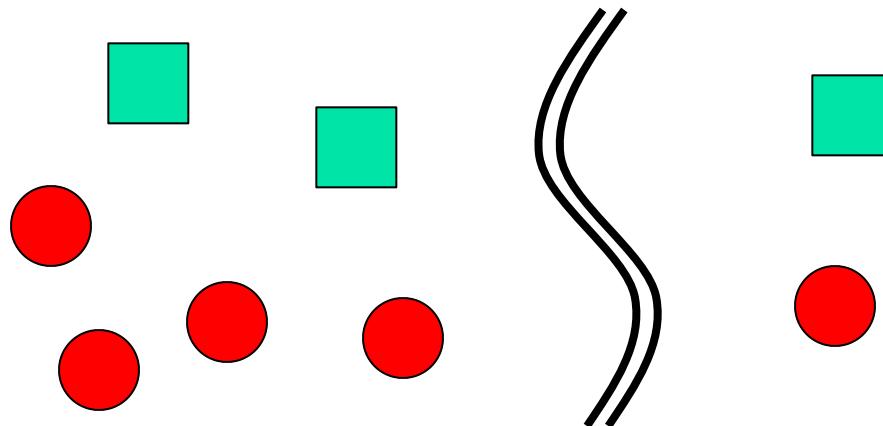
Option 1: **accept reads**
 reject writes **accept reads**
 reject writes

Option 2: **accept reads**
 accept writes **reject reads**
 reject writes



Effects of Continuous Consistency

Policy: each replica can buffer up to 5 writes

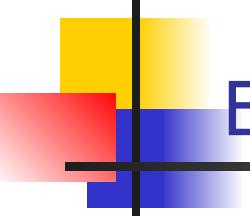


Option 1: **accept reads**
reject writes

accept reads
reject writes

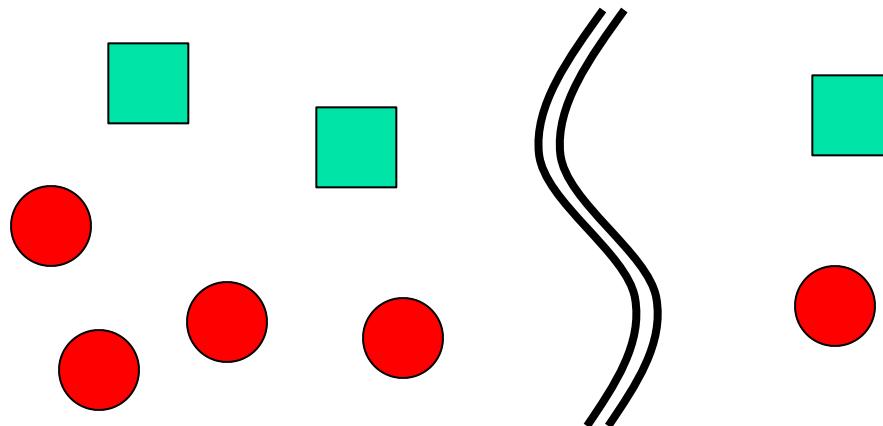
New Option 1: **accept reads**
accept first 10 writes

accept reads
accept first 5 writes



Effects of Continuous Consistency

Policy: each replica can buffer up to 5 writes



Option 2: accept reads
accept writes

reject reads
reject writes

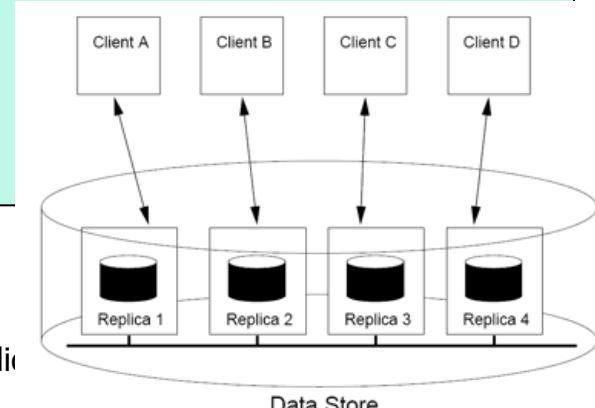
New Option 2: accept reads
accept writes

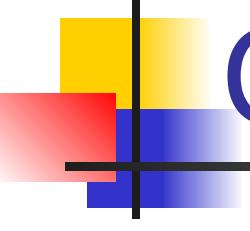
accept first few reads
accept first 5 writes

Consistency models: contracts between the data store and the clients

- Data centric: solutions at the data store level
 - Continuous consistency
 - limit the deviation between replicas, or
 - Eventual consistency
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level

- Client centric
 - Assume client-independent views of the datastore
 - Constraints on operation ordering
 - for each client independently

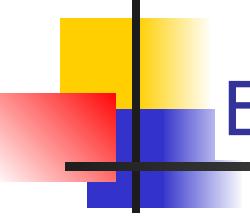




Client-centric Consistency Models

Goal: Avoid system-wide consistency,
by concentrating on what each client
independently wants

(instead of maintaining a global view)



Example: Consistency for Mobile Users

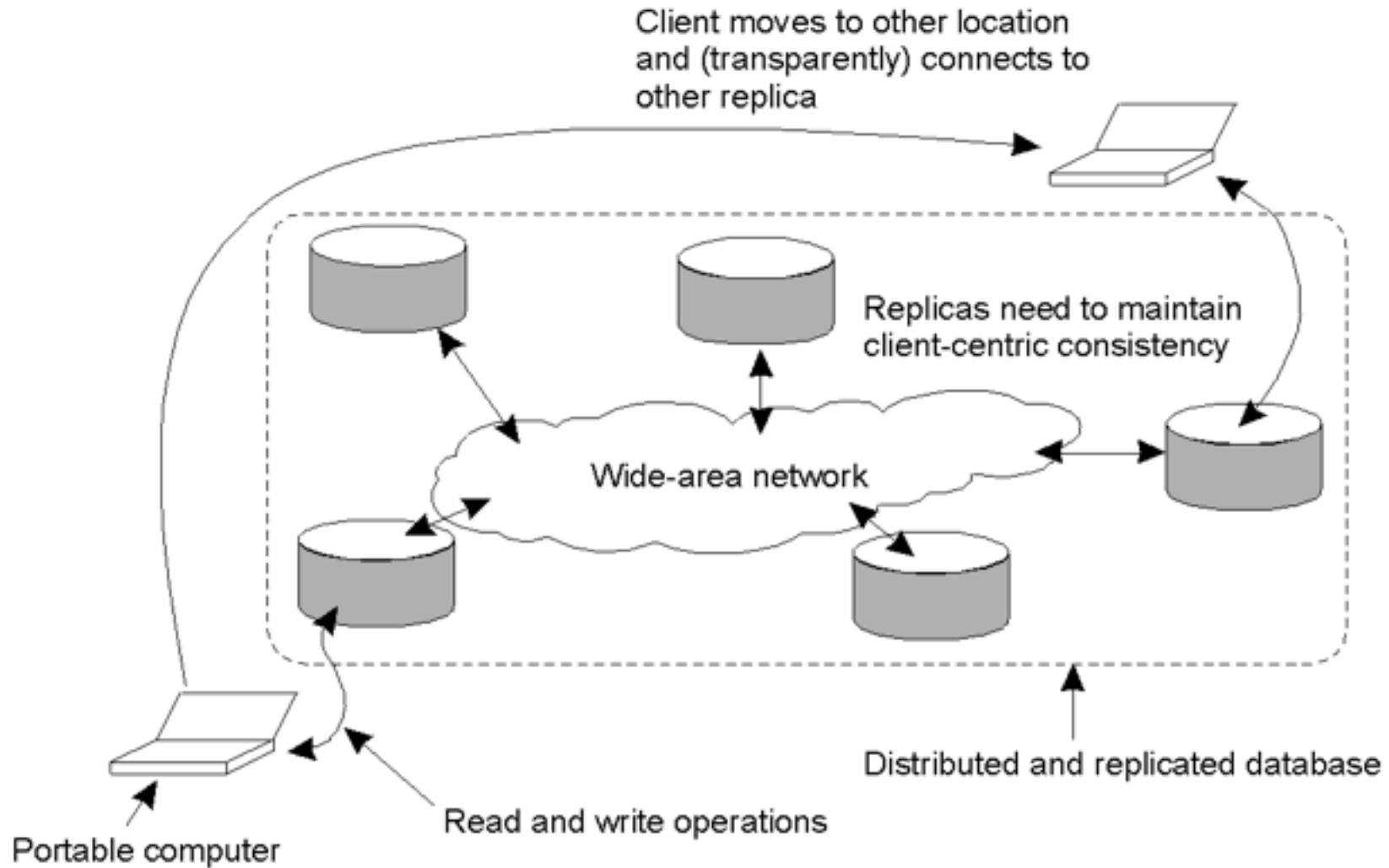
Example: Distributed database to which a user has access through her notebook.

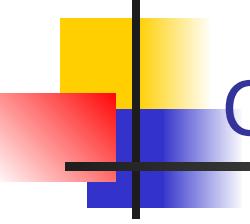
- Notebook acts as a front end to the database.
- At location A user accesses the database with reads/updates
- At location B user continues work, but unless it accesses the same server as the one at location A, she may detect inconsistencies:
 - updates at A may not have yet been propagated to B
 - user may be reading newer entries than the ones available at A:
 - user updates at B may eventually conflict with those at A

Note: The only thing the user really needs is that the entries updated and/or read at A, are available at B the way she left them in A.

- **Idea:** the database will appear to be consistent **to the user**

Example - Consistency for Mobile Users





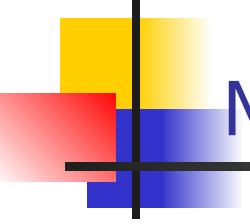
Client-centric Consistency

Idea: Guarantee a degree of data access consistency for a single client/process point of view.

Notations:

- $x_i[t] \rightarrow$ Value of data item x at time t at local replica L_i
- $WS(x_i [t]) \rightarrow$ working set (all write operations) at L_i up to time t on data item x
- $WS(x_i [t]; x_j [t]) \rightarrow$ indicates that it is known that $WS(x_i [t])$ is included in $WS(x_j [t])$

Shorthand: we do not use $[t]$ (it's implied)



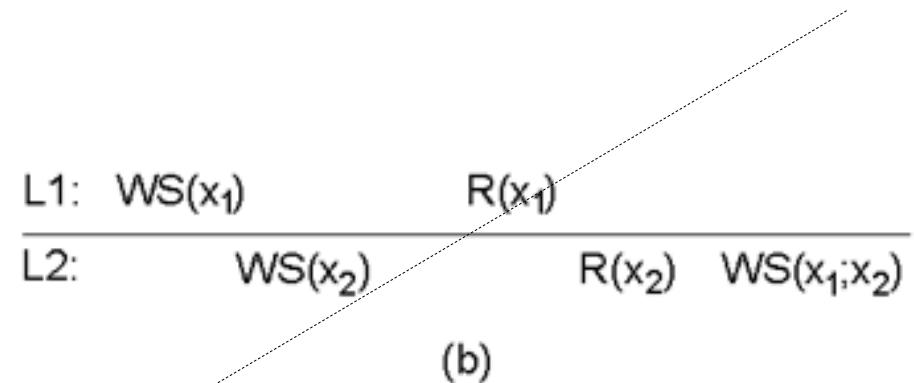
Monotonic-Read Consistency

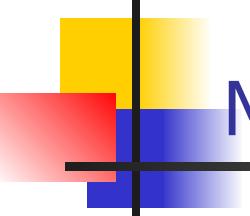
Intuition: Client “sees” the same or newer version of data.

Definition: If a process reads the value of a data item x , any successive read operation on x by that process will always return that same or a more recent value.

L1: WS(x_1)	R(x_1)
L2: WS($x_1;x_2$)	R(x_2)

(a)





Monotonic reads – Examples

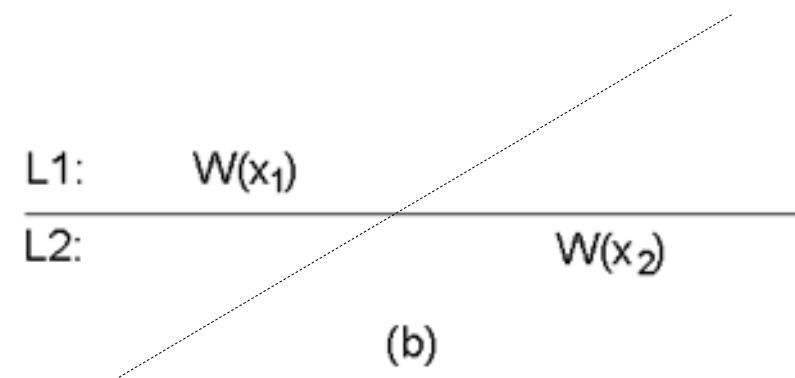
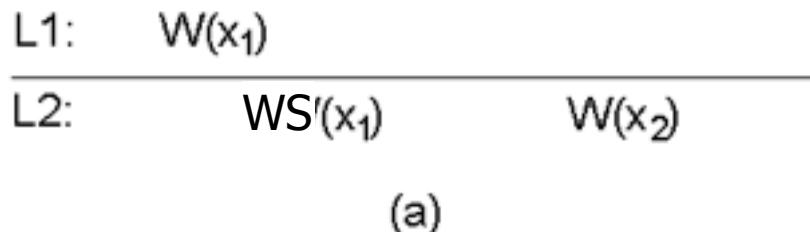
- A personalized news webpage
- Reading (not modifying) incoming e-mail while you are on the move.
 - Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.
- Reading personal calendar updates from different servers.
 - Monotonic Reads guarantees that the user sees always more recent updates, no matter from which server the reading takes place.

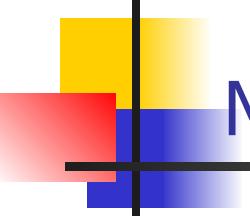
(All the above assume read-only)

Monotonic-Write Consistency

Intuition: A write happens on a replica only if it's brought up to date with preceding write operations on same data (but possibly at different replicas)

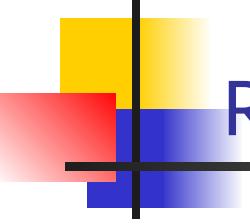
Definition: A write operation by a process on a data item x is completed (on all replicas) before any successive write operation on x by the same process.





Monotonic writes – Examples

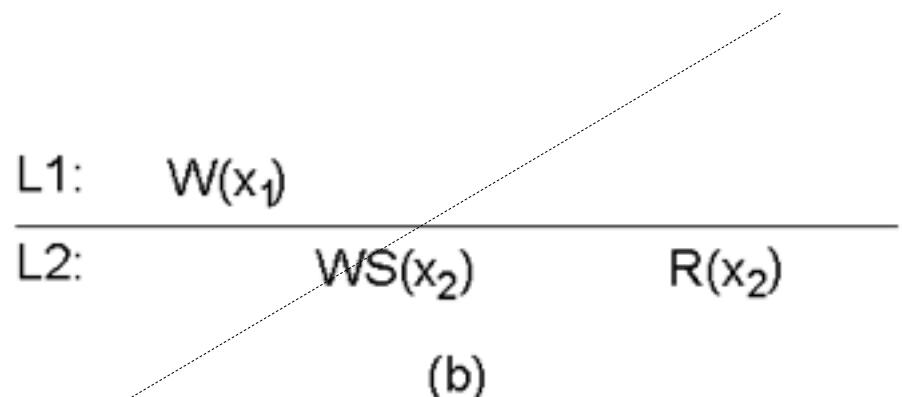
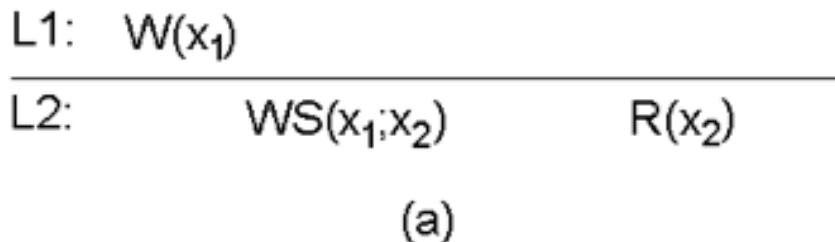
- Updating a program at server S2, and ensuring that all components on which compilation and linking depend, are also placed at S2.
- Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

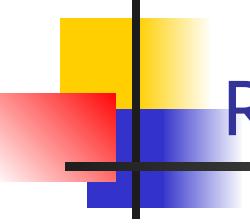


Read-Your-Writes Consistency

Intuition: All previous writes are always completed before any successive read

Definition: The effect of a write operation by a process on data item x , will always be seen by a successive read operation on x by the same process.





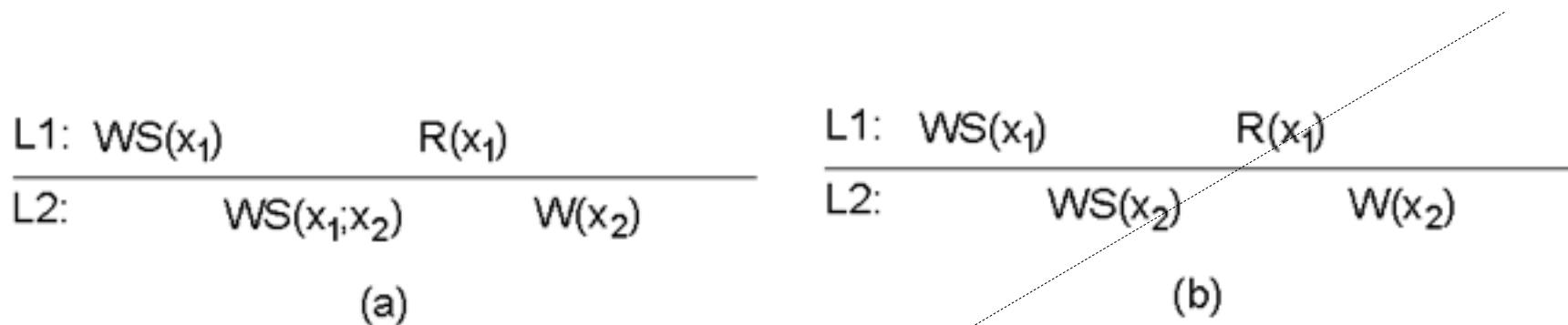
Read-Your-Writes - Examples

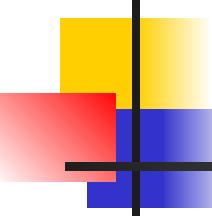
- Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.
- Password database

Writes-Follow-Reads Consistency

Intuition: Any successive write operation on x will be performed on a copy of x that is same or more recent than the last read.

Definition: A write operation by a process on a data item x following a previous read operation on x by the same process, is guaranteed to take place on the same or a more recent value of x that was read.



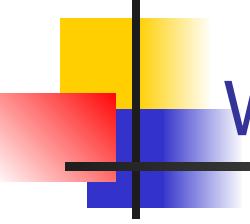


Quizz like questions

What is the crucial difference between data-centric and client-centric consistency models?

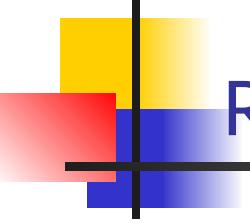
Where does the 'eventual consistency' model fit in this taxonomy: is it data-centric or client-centric?

Consider a system that combines read-your-writes consistency with writes-follow-reads consistency (that is, it provides both). Is this system also sequentially consistent?



Writes-Follow-Reads - Examples

- See reactions to posted articles only if you have the original posting
 - a read “pulls in” the corresponding write operation.



Roadmap

Updating replicas

- **Consistency models** (how to deal with updated data)
 - (luckily) applications do not always require strict consistency
- **Consistency protocols: Implementation issues**
 - How is a consistency model implemented

Replica and content management

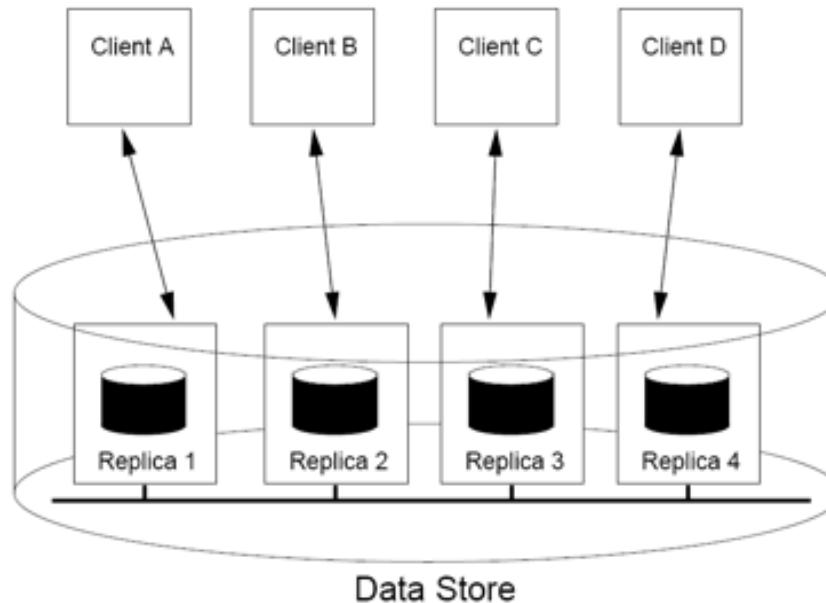
- How many replicas?
- Where to place them?
- When to get rid of them?

Redirection/Routing

- Which replica should clients use?

Reminder: System view

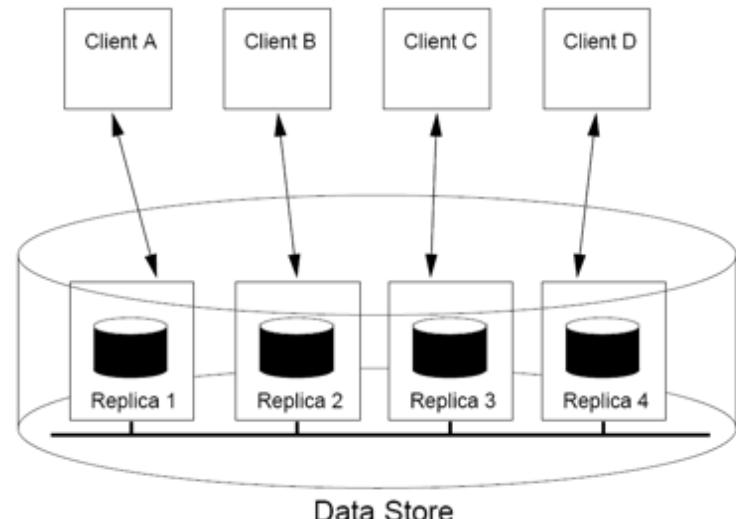
Management system: Controls the allocated resources and aims to provide replication transparency



Consistency model: **contract** between the data store and the clients

Reminder: Types of consistency models

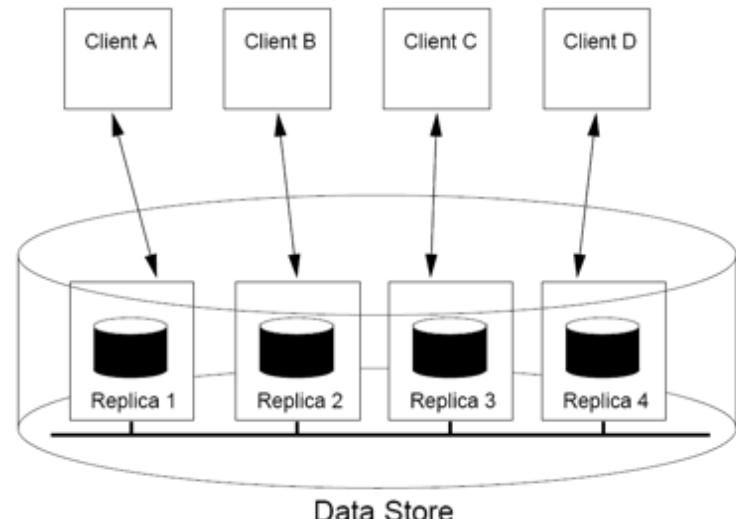
- Data centric: solutions at the data store level
 - Continuous consistency: limit the deviation between replicas, or
 - Constraints on operation ordering at the data-store level
- Client centric
 - Constraints on operation ordering for each client independently

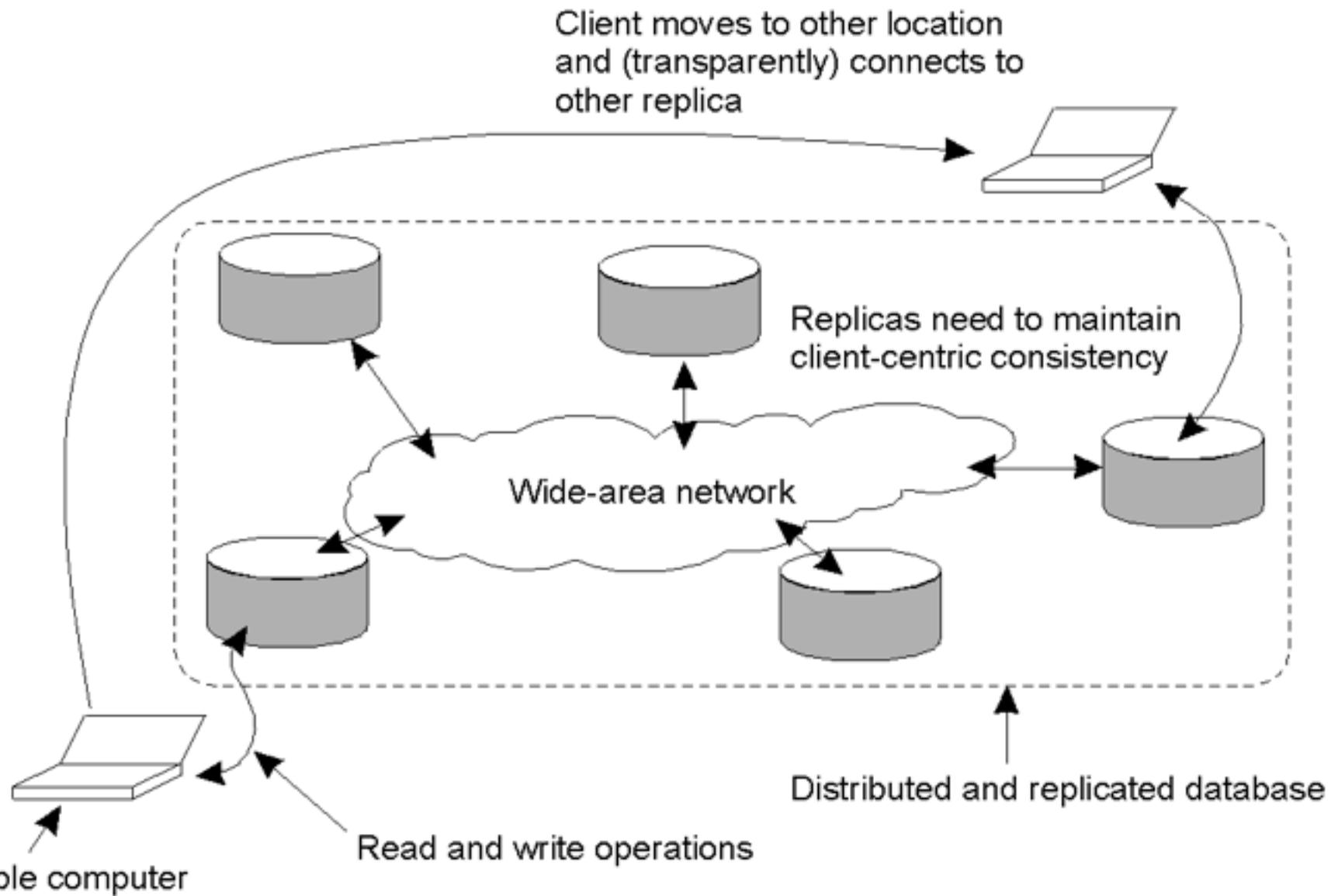


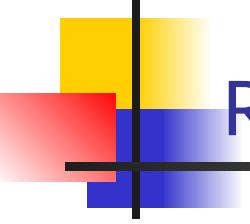
Today: Consistency protocols

Question: How does one design the protocols to implement the desired consistency model?

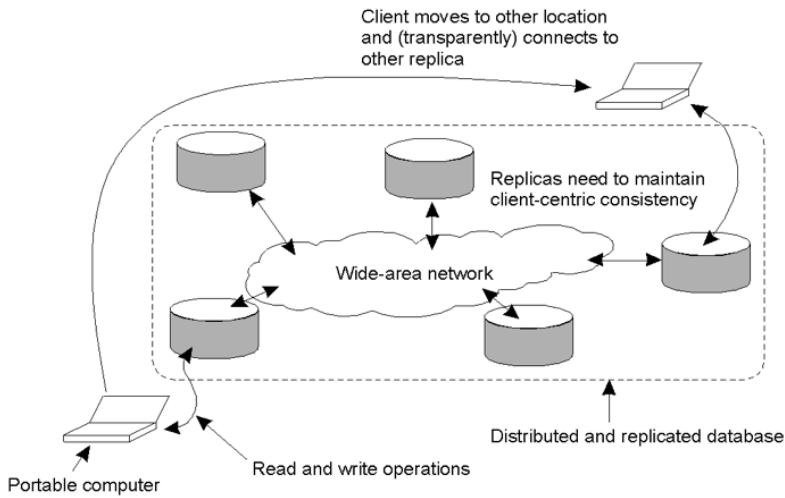
- Data centric
 - Constraints on operation ordering at the data-store level
 - Sequential consistency.
 - Continuous consistency: limit the deviation between replicas
- Client centric
 - Constraints on operation ordering for each client independently







Reminder: Monotonic-Read



Definition: If a process reads the value of a data item x , any successive read operation on x by that process will always return that same or a more recent value.

- Intuition: Client “sees” the same or a newer version of the data.

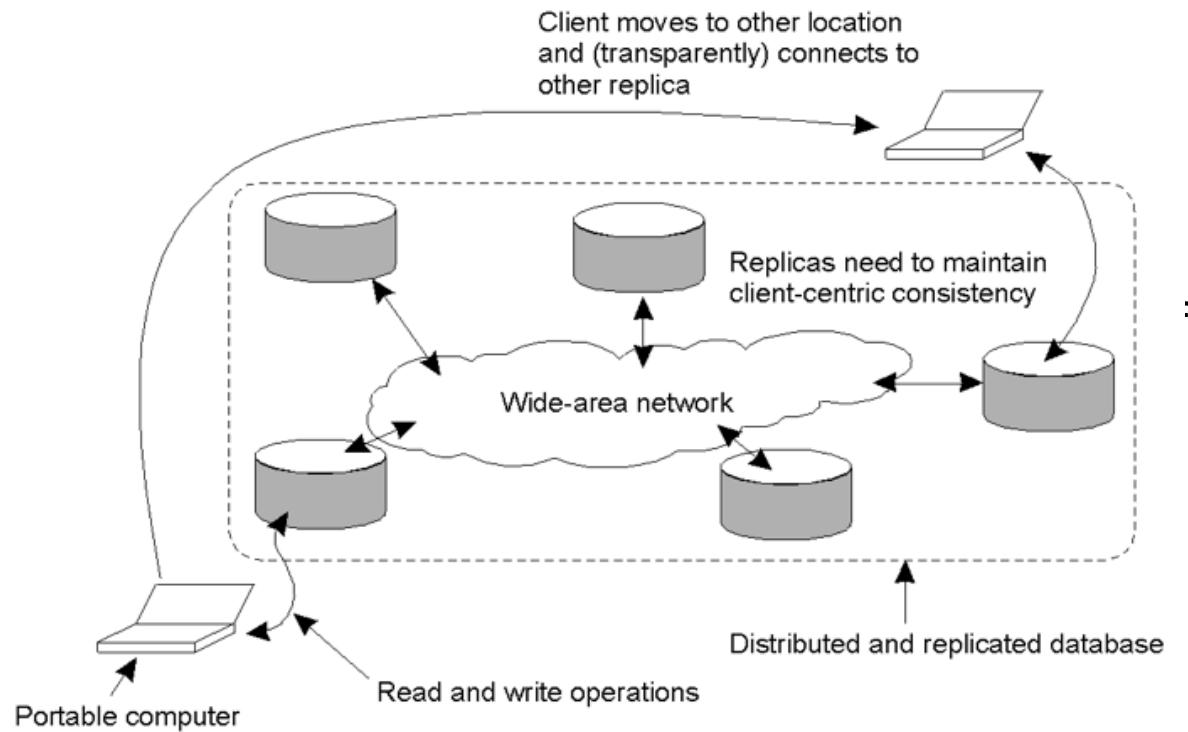
Quizz-like question: how would you implement this?

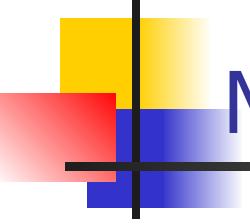
Implementation of monotonic-read consistency

Key intuition: client carries state to identify the last operation(s)

Protocol sketch:

- Globally unique ID
- The client keeps track of:
 - **ReadSet**: the writes performed on various objects
- When a client launches a read operation:
 - Client sends the ReadSet to the server
 - The server checks if the reads have been performed.
 - [If necessary] Fetches unperformed updates
 - Executes the read operation.





More quiz-like questions

Sketch a protocol design for the following consistency model

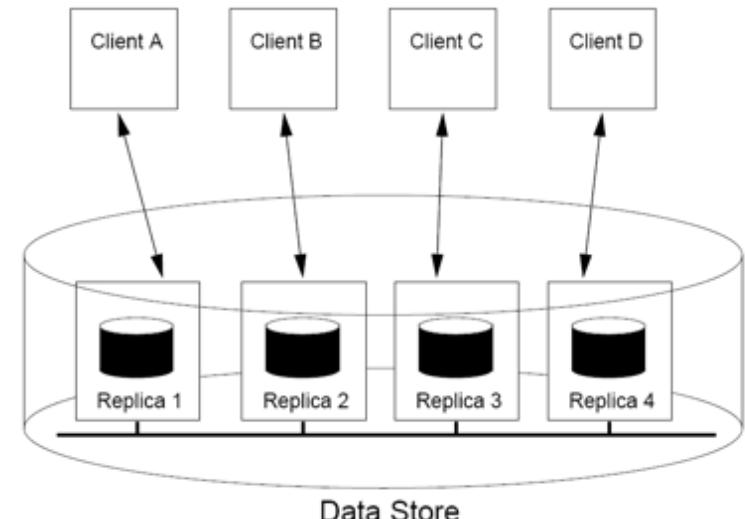
- Monotonic-writes
- Read-your-writes
- Writes-follow-reads

Write pseudo-code

Consistency protocols

Question: How does one design the protocols to implement the desired consistency model?

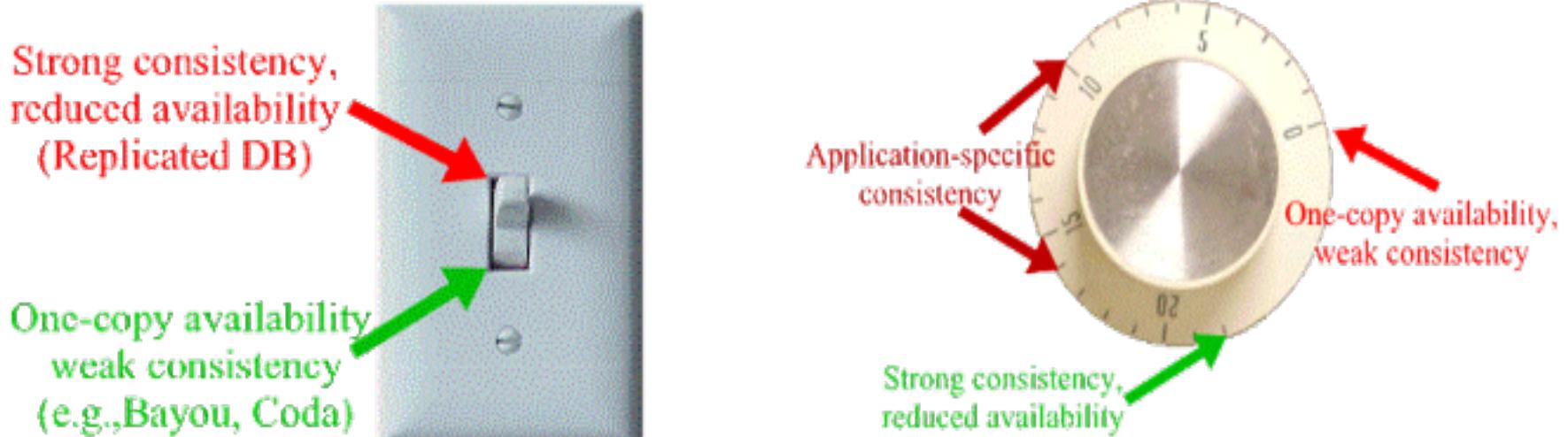
- Data centric
 - Constraints on operation ordering at the data-store level
 - Sequential consistency.
 - **Continuous consistency: limit the deviation between replicas**

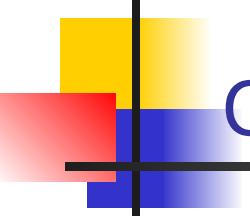


Continuous Consistency

- Obs1: We can talk about a **degree** of consistency
- Goal: Limit the **deviation** between replicas

Metaphor: Knob vs. switch





Continuous Consistency

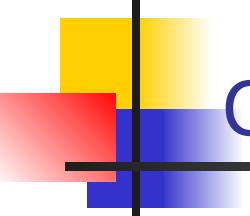
Obs1: We can talk about a degree of consistency

- Goal: Limit the **deviation** between replicas

Obs2: Multiple metrics to measure deviation are possible

- replicas may differ in their numerical value
- replicas may differ in their relative staleness
- replicas may differ with respect to (number and order) of performed update operations

conit: consistency unit → specifies the data unit over which consistency is to be enforced. (e.g., file, variable, folder)



Continuous Consistency

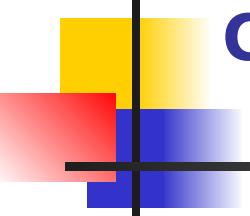
Obs1: We can talk about a degree of consistency

- Goal: Limit the **deviation** between replicas

Obs2: Multiple metrics to measure deviation are possible

- replicas may differ in their numerical value
- replicas may differ in their relative **staleness**
- replicas may differ with respect to (number and order) of performed update operations

conit: consistency unit → specifies the data unit over which consistency is to be enforced. (e.g., file, variable, folder)



Continuous Consistency: Bounding Numerical Errors (I)

Setup: consider a conit (data item) x and let $\text{weight}(W(x))$ denote the numerical change in its value after a write W .

- [for simplicity] Assume that for all $W(x)$, $\text{weight}(W) > 0$
- W is initially forwarded to one of the N replicas: the $\text{origin}(W)$.
- $TW[i, j]$ are the writes executed by server S_i that originated from S_j
$$TW[i, j] = \sum \{\text{weight}(W) \mid \text{origin}(W) = S_j \text{ & } W \text{ in } \log(S_i)\}$$

Note: Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k,k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i,k]$$

Goal: need to ensure that $v(t) - v_i < d$ for every S_i .

Continuous Consistency: Bounding Numerical Errors (II)

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k,k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i,k]$$

Goal: need to ensure that $v(t) - v_i < d$ for every S_i .

Approach: Let every server S_k maintain a **view** $TW_k[i, j]$ for what S_k believes is the value of $TW[i, j]$.

- This information can be piggybacked when an update is propagated.

Note: $0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$

Continuous Consistency: Bounding Numerical Errors (II)

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k,k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i,k]$$

Goal: need to ensure that $|v(t) - v_i| < d$ for every S_i

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$

Goal: need to ensure that $\sum_{\substack{k=1 \\ k \neq i}}^N (TW[k, k] - TW[i, k]) < d$

then, as an approximate bound: $TW[k, k] - TW[i, k] < \frac{d}{N-1}$
for all k

Approach: Let every server S_k maintain a **view** $TW_k[i, j]$ for what S_k believes is the value of $TW[i, j]$.

- This information can be piggybacked when an update is propagated.

Note1: $0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$

Note2: $TW_k[i, k]$: view at node K about updates inserted at k and propagated to I

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$

Goal: need to ensure that $v(t) - v_i < d$ for every S_i .

Approach: Let every server S_k maintain a **view** $TW_k[i, j]$ for what S_k believes is the value of $TW[i, j]$.

- This information can be piggybacked when an update is propagated.

Solution: S_k sends operations from its log to S_i when it sees that $TW_k[i, k]$ is getting too far from $TW[k, k]$,

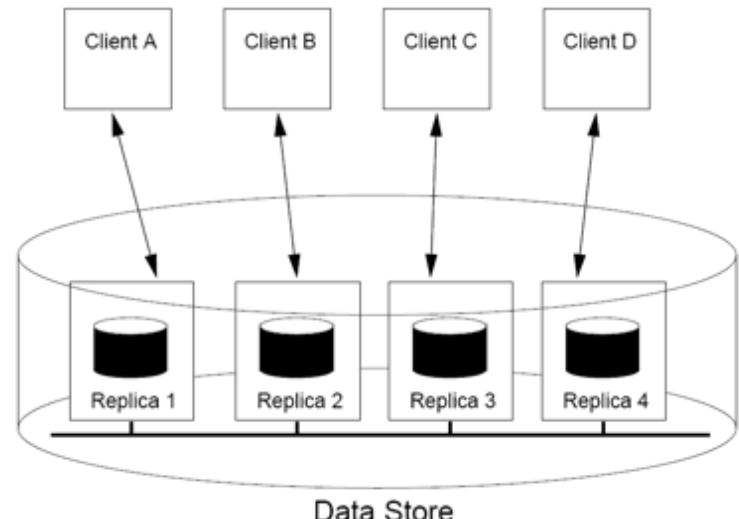
- in particular, when $TW[k, k] - TW_k[i, k] > d_i/(N - 1)$.

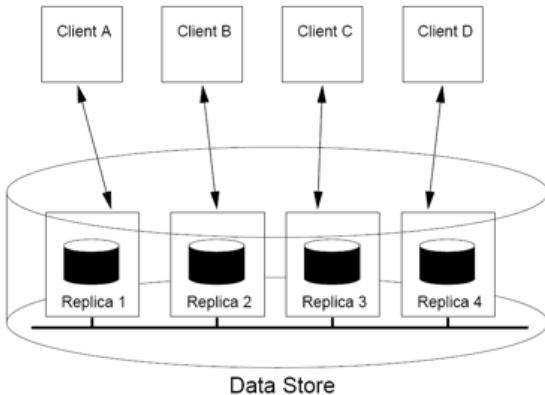
Note: Staleness can be done analogously, by essentially keeping track of what has been seen last from S_k

Consistency protocols

Question: How does one design the protocols to implement the desired consistency model?

- Data centric
 - Constraints on operation ordering at the data-store level
 - Sequential consistency, causal consistency, etc
 - Continuous consistency: limit the deviation between replicas
- Client centric
 - Constraints on operation ordering for each client independently





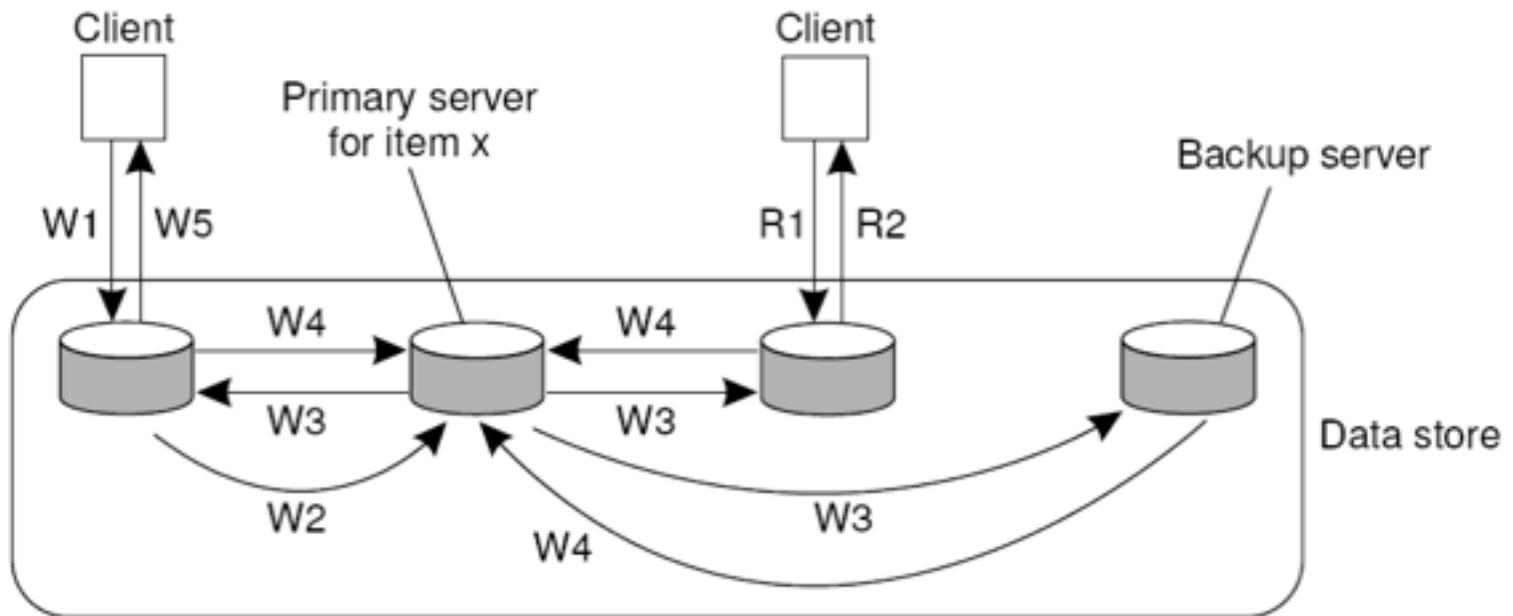
Overview: Data-centric Consistency Protocols

Primary-based

Replicated-write



Primary-Based Protocols: Primary backup



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

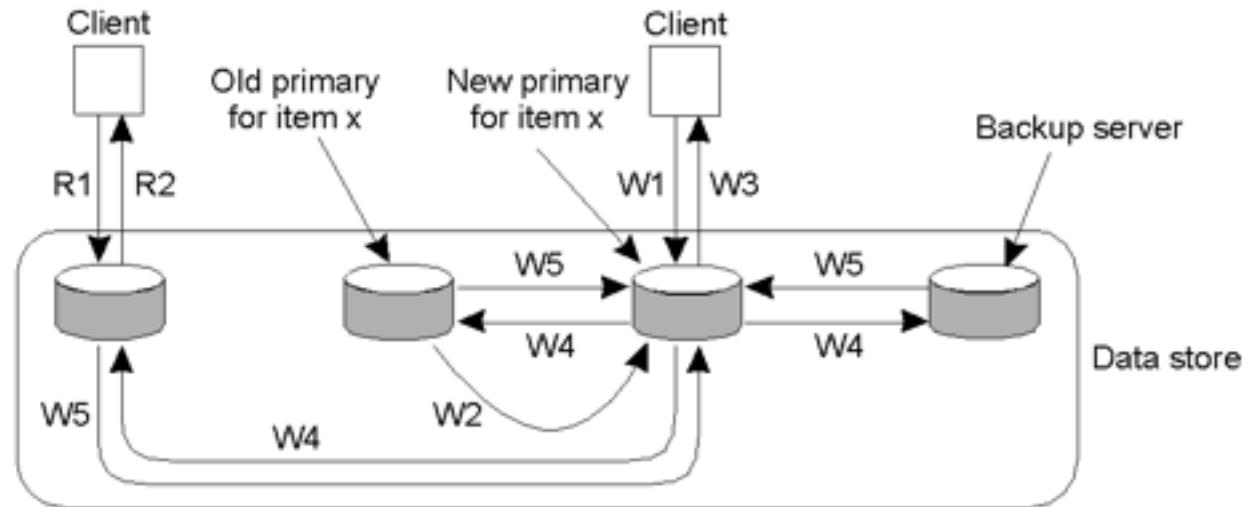
R1. Read request
R2. Response to read

Usage: distributed databases and file systems that require a high degree of fault tolerance. Replicas often placed on same LAN.

Issues: blocking vs. non-blocking

Primary-backup protocol with local writes

Idea: The primary migrates to the node that executes the write

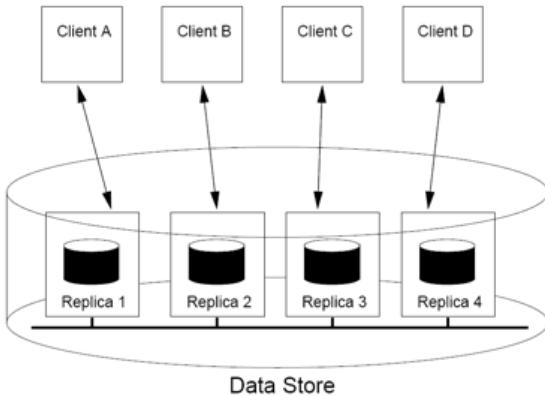


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

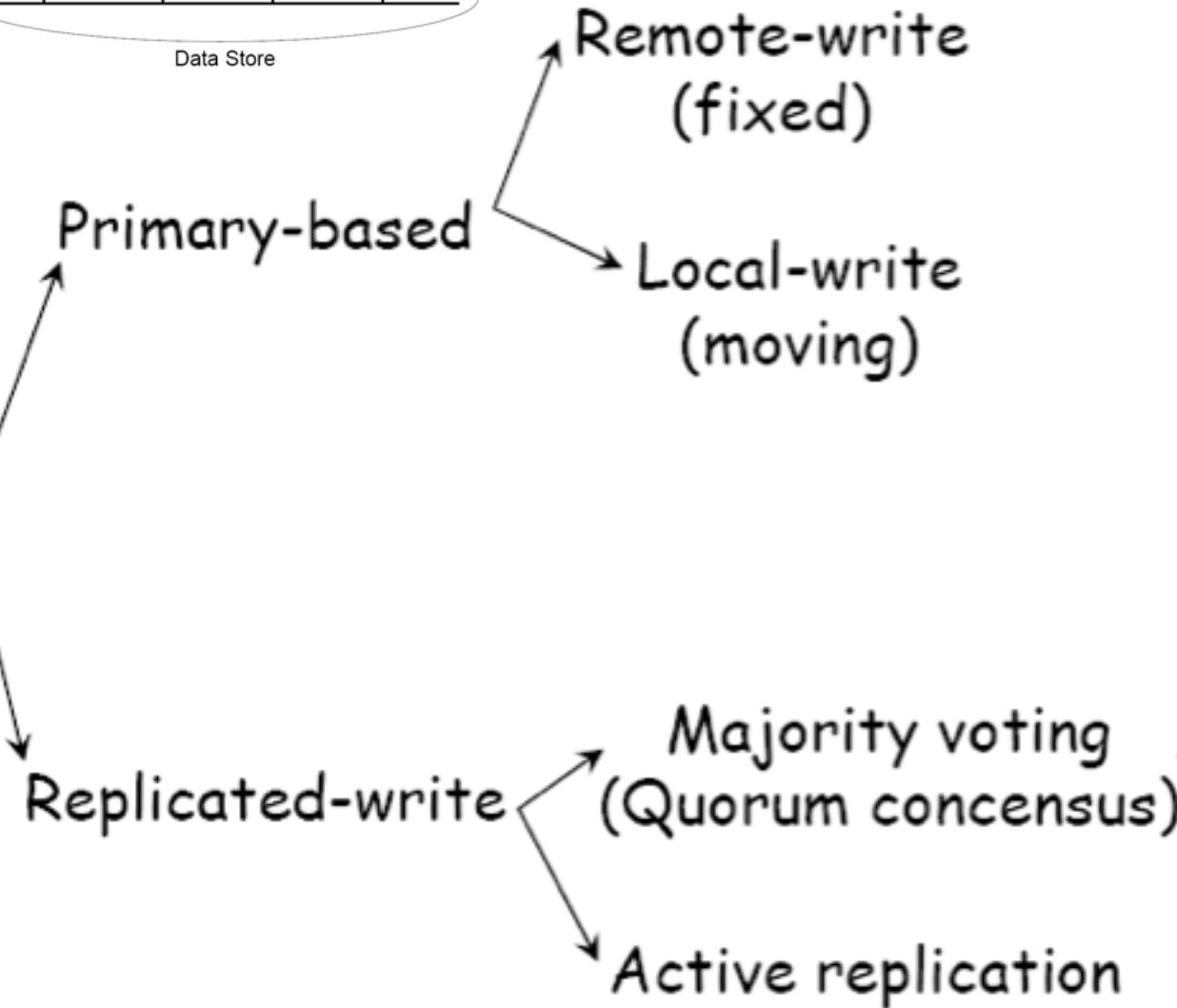
R1. Read request
R2. Response to read

- Usage:** Mobile computing in disconnected mode

- Idea: ship all relevant files to user before disconnecting, update later.



Overview: Data-centric Consistency Protocols



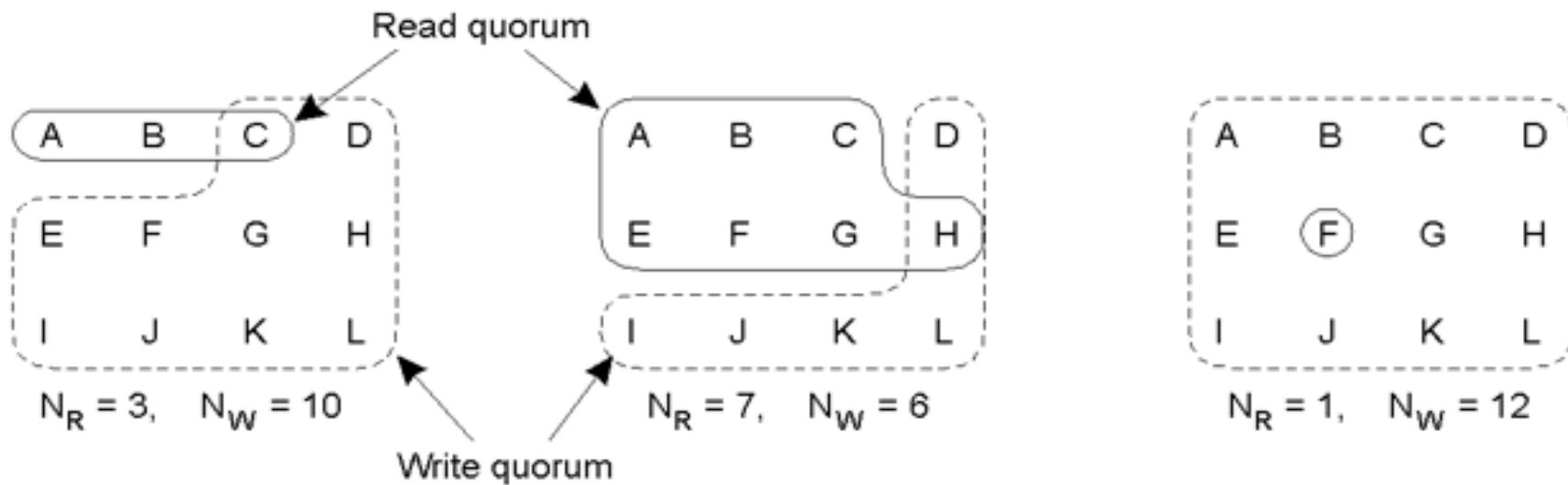
Replicated writes: Quorums

Problem: some replicas may not be available all the time

- leads to low availability (for writes)

Solution: Quorum-based protocols: Ensure that each operation is carried on enough replicas (not all)

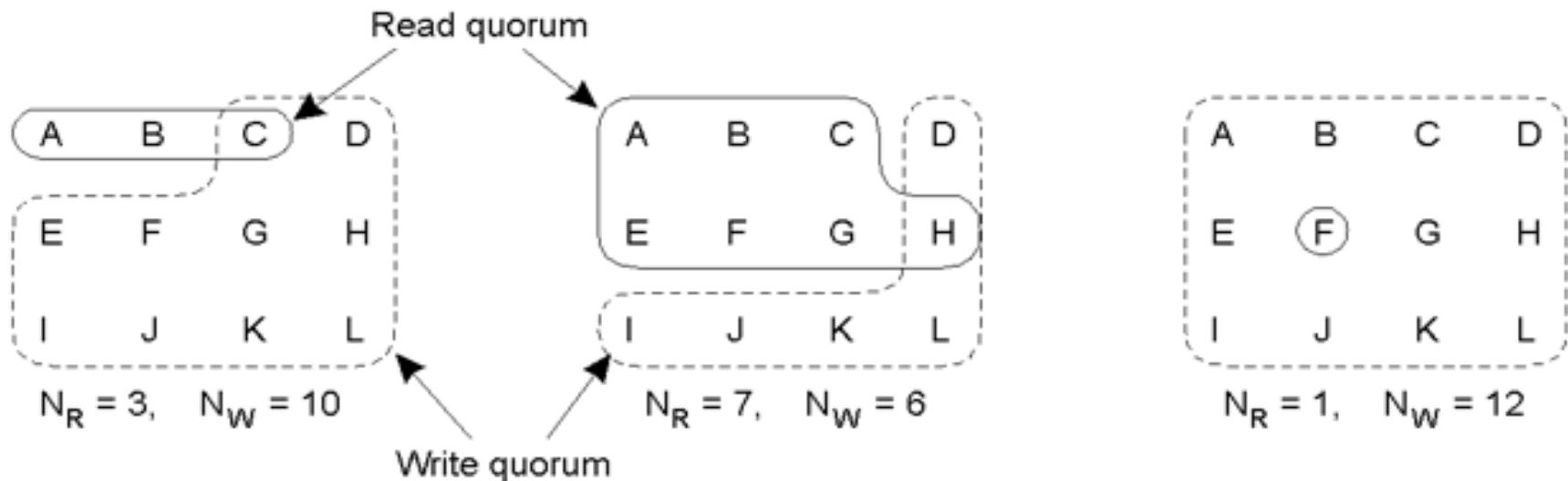
- a majority 'vote' is established before each operation
- distinguish a **read auorum** and a **write auorum**:



What's a valid quorum configuration?

Must satisfy two basic rules

- A **read quorum** should “intersect” any prior write quorum at ≥ 1 processes
 - $Q_r + Q_w > N$
- A **write quorum** should intersect any other write quorum
 - $Q_w + Q_w > N$ results in $Q_w > N/2$



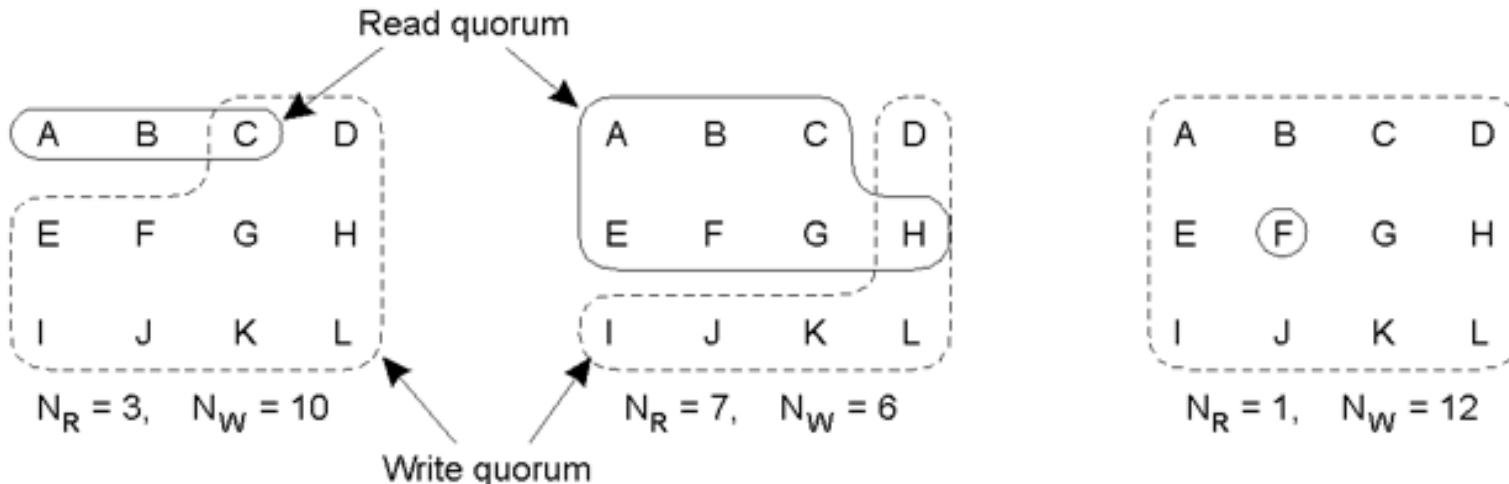
Quorums: Mechanics for read and write operations

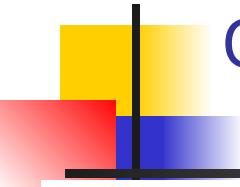
Setup: Replicas have “versions”,

- Versions are numbered (timestamped)
- Timestamps must increase monotonically
 - include a process id to break ties

Read operations:

- Client send RPCs until Q_r processes reply
- Then use the replica with the largest timestamp (the most recently updated replica)





Quorums:

Mechanics for read and write operations

Setup: Replicas have “versions”,

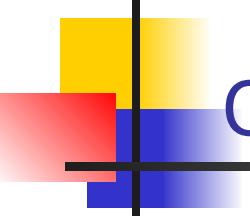
- Versions are numbered (timestamped)
- Timestamps must increase monotonically
 - include a process id to break ties

Reads

- Client send RPCs until Q_r processes reply
- Then use the replica with the largest timestamp (the most recently updated replica)

Writes are trickier:

- Need to determine next version number.
- Need a protocol to protect from concurrent writes
 - Need to make sure all replicas are updated atomically ('commit' like protocol).



Quorum based protocols: Write algorithm

- Client: Contacts all replica hosts and propose the write.
 - "I would like to execute write W_j on data-item I"
- Replica:
 - Locks the replica against other writes
 - Puts the request in a queue of pending writes
 - Sends back: ACK, proposed version-number (e.g., logical clock) , pID
- Client:
 - If $< Q_w$ replies: send ABORT to all participants
 - If $\geq Q_w$ replies:
 - $\text{new_replica_number} = \max [\text{version}, \text{pID}]$
 - Send (COMMIT, new replica number)

Issue 1. Dealing with data changes

- **Consistency models**
 - What is the semantic the system implements
 - (luckily) applications do not always require strict consistency
- **Consistency protocols**
 - How to implement the semantic agreed upon?

Issue 2. Replica management

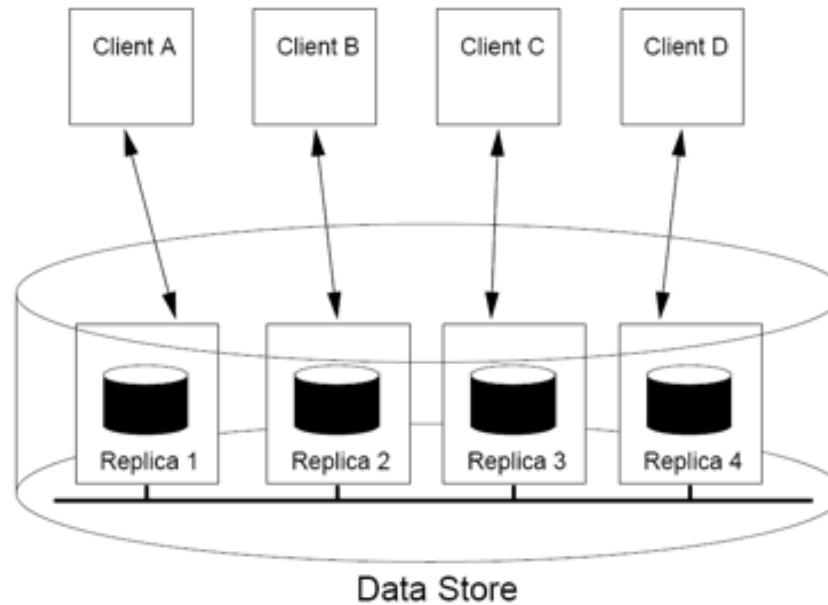
- How many replicas?
- Where to place them?
- When to get rid of them?

Issue 3. Redirection/Routing

- Which replica should clients use?

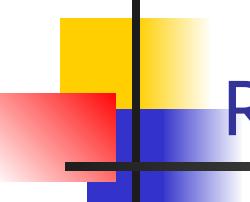
Reminder: System view

Management system: Controls the allocated resources and aims to provide replication transparency



Consistency model: contract between the data store and the clients

- Consistency model to have in mind for next slides: **eventual consistency**
- Example applications: **web content distribution**
 - HTML, videos, etc



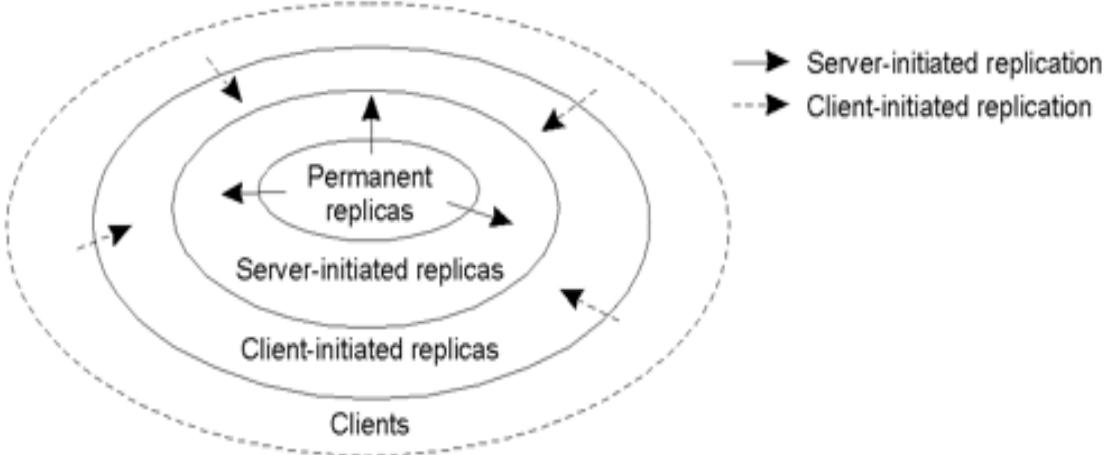
Replica server placement

Problem: Figure out what the best K places are out of N possible locations.

- Select best location out of $N - j$, $j:0..K-1$ for which the **average distance to clients is minimal**. Then choose the next best server.
 - (Note: The first chosen location minimizes the average distance to all clients.)
 - Computationally expensive.
- Select the k largest **autonomous system** and place a server at the best-connected host.
 - Computationally expensive.
- Position nodes in a d-dimensional geometric space, where distance reflects latency. Identify the K regions with highest density and place a server in every one.
 - Computationally cheap.

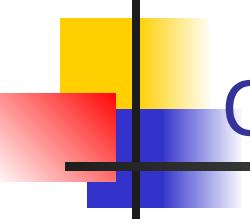
Content replication

Model: We consider
(don't worry whether t



Distinguish different processes: A process is capable of hosting a replica of an object :

- **Permanent replicas:** Process/machine always having a replica (most of the discussion so far!)
- **Server-initiated replica:** Process that can dynamically host a replica on request of another server in the data store
- **Client-initiated replica:** Process that can dynamically host a replica on request of a client (**client caches**)



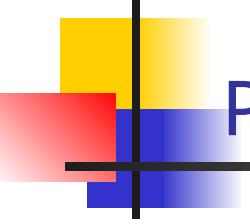
Client initiated replicas (caches)

Issue: What do I propagate when content is dynamic?

State vs. operations

- Propagate only **notification/invalidation** of update (often used for web caches)
- **Transfer data** from one copy to another (often for caching in distributed databases)
- **Propagate the update** operation to other copies (similar to active replication)

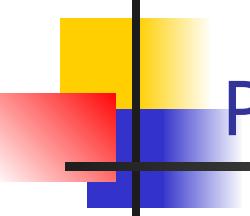
Note: No single approach is the best, but depends highly on (1) available bandwidth and read-to-write ratio at replicas; (2) consistency model



Propagating updates (1/2)

- **Pushing updates**: server-initiated approach, in which the update is propagated regardless of whether the target asked for it.
- **Pulling updates**: client-initiated approach, in which client requests to be updated.

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update message (and possibly fetch update later)	<u>Poll</u> <u>and</u> update
Response time at client	Immediate (or fetch-update time)	Fetch-update time



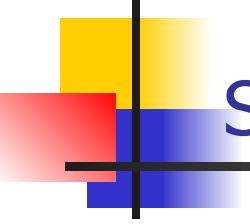
Propagating updates: Leases (2/2)

Observation: We can dynamically switch between pull and push using **leases**:

- Lease: A contract in which the server promises to push updates to the client until the lease expires.

Additional Technique: Make lease expiration time dependent on system's behavior (adaptive leases):

- **Age-based leases:** An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases:** The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases:** The more loaded a server is, the shorter the expiration times become



Summary

Issue 1. Dealing with data changes

- **Consistency models**
 - What is the semantic the system implements
 - (luckily) applications do not always require strict consistency
- **Consistency protocols**
 - How to implement the semantic agreed upon?

Issue 2. Replica management

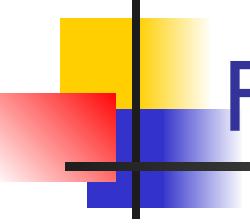
- How many replicas?
- Where to place them?
- When to get rid of them?

Issue 3. Client Side Caching

- How to deal with client initiated replication



"Failure is not an option. It comes bundled with your system." (--unknown)



Fault-tolerant software is inevitable

- Things will crash. Deal with it!
 - Assume you could start with super reliable servers (MTBF of 30 years) –
 - Build computing system with 10 thousand of those
 - Watch one fail per day
- Typical yearly flakiness metrics
 - 1-5% of your disk drives will die
 - Servers will crash at least twice (2-4% failure rate)

([Source](#): Jeff Dean, Google, '12)

Typical first year for a new cluster (Source: Jeff Dean, Google, '12)

- **~0.5 overheating** (power down most machines in <5 mins, ~1-2 days to recover)
- **~1 PDU failure** (~500-1000 machines suddenly disappear, ~6 hours to come back)
- **~1 rack-move** (plenty of warning, ~1000 mach. powered down, ~6h)
- **~1 network rewiring** (rolling ~5% of machines down, 2-day span)
- **~20 rack failures** (40-80 machines disappear, 1-6 hours to get back)
- **~5 racks go wonky** (40-80 machines see 50% packet loss)
- **~8 network maintenances** (4 might cause ~30-minute random connectivity losses)
- **~12 router reloads** (takes out DNS and external vIPs for 100 sec)
- **~3 router failures** (have to immediately pull traffic for an hour)
- **~dozens of minor 30-second blips for DNS**
- **~1000 individual machine failures, hard drive failures**, slow disks, bad memory, misconfigured or flaky machines, etc

More real-world datapoints

Data set	Type of cluster	Duration	#Disk events	# Servers	Disk Count	Disk Parameters	MTTF (Mhours)	Date of first Deploym.	ARR (%)
HPC1	HPC	08/01 - 05/06	474	765	2	1000	1000	2006	100
			124	64	1				
HPC2	HPC	01/04 - 07/06	14	256					
HPC3	HPC	12/05 - 11/06	103	1,532	3				
	HPC	12/05 - 11/06	4	N/A	..				
	HPC	12/05 - 11/06	103	1,532	3				

HPC1	
Component	%
CPU	44
Memory	29
Hard drive	16
PCI motherboard	9
Power supply	2

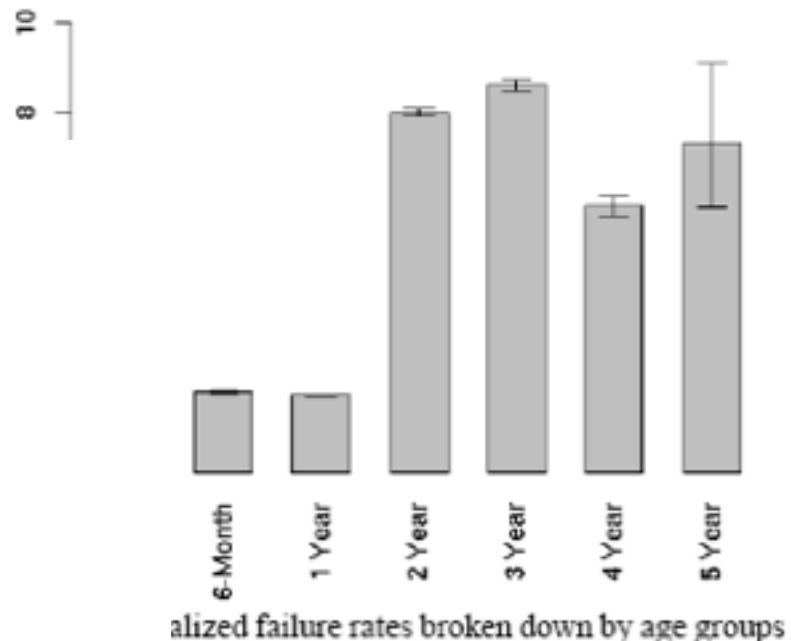
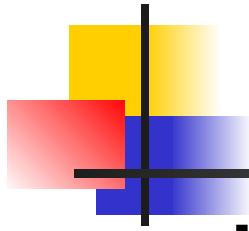


Table 2: Node outages that were attributed to hardware problems broken down by the responsible hardware component. This includes all outages, not only those that required replacement of a hardware component.

s Mean to You?, Bianca Schroeder and

Dietrich Weber, and Luiz André

Applications



Key message:

Reliability/availability must come from software!

Terminology

Failure

The system (or component) is not living up to its specs

Error

Exposed part of the system state that might lead to failure

Fault

The cause of an error

Terminology

Example

Failure

The system (or component) is not living up to its specs

Error

Exposed part of the system state that might lead to failure

Fault

The cause of an error

Cosmic ray hits, causes a bit-flip in a latch.

Terminology

Example

Failure

The system (or component) is not living up to its specs

Error

Exposed part of the system state that might lead to failure

Arithmetic operation gives incorrect result

Fault

The cause of an error

Cosmic ray hits, causes a bit-flip in a latch.

Terminology

Example

Failure

The system (or component) is not living up to its specs

Application crash
(segmentation fault as a result of using the value computed before for addressing)

Error

Exposed part of the system state that might lead to failure

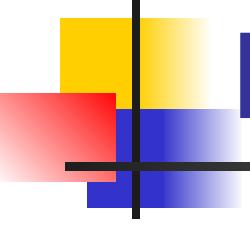
Arithmetic operation gives incorrect result

Fault

The cause of an error

Cosmic ray hits, causes a bit-flip in a latch.

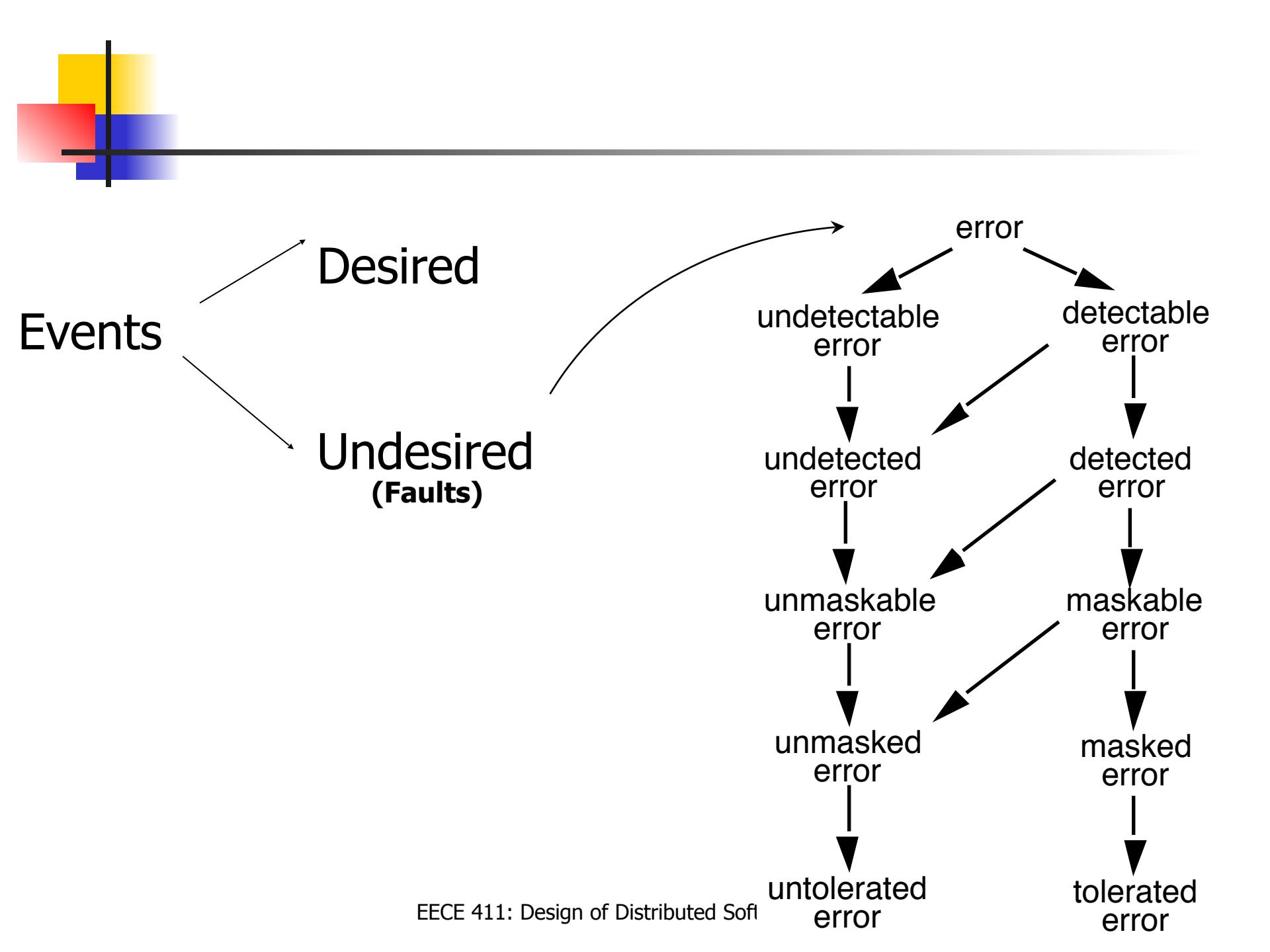
Perspective matters: A component failure, may be the error from a system perspective

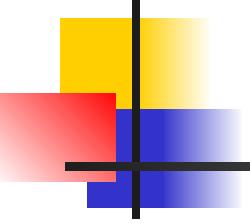


Design Goal

(with regard to fault-tolerance):

Design a distributed system that can recover from partial failures without affecting correctness or significantly impacting overall performance

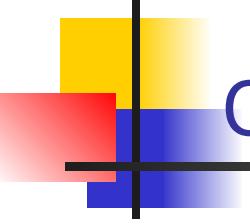




In practice ... cannot protect against everything

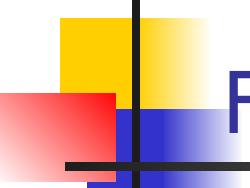
- Unlikely faults (e.g., flooding in the Sahara)
- Expensive to protect faults
(e.g., BIG earthquake)

There are faults we can protect against (e.g., some earthquake, some flooding)



Outline

- Fault models
- Process resilience
- Reliable group communication
- Distributed commit



Fault models (In increasing order of severity)

Crash: A component simply halts, but behaves correctly before halting

Model refinement:

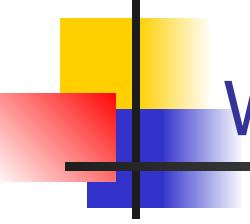
- **Fail-stop:** The component exhibits a crash, but its failure can be detected (either through announcement or timeouts)
- **Fail-silent:** The component exhibits omission or crash failures; clients cannot tell what went wrong

Omission fault: A component fails to respond (send a message) or to receive a message

Timing faults: The output of a component is correct, but response time is outside the specified real-time interval

- E.g., performance failures: too slow

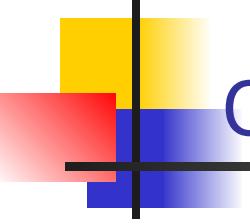
Arbitrary (byzantine) faults: A component may produce arbitrary output and be subject to arbitrary timing failures



What to do to obtain fault tolerance?

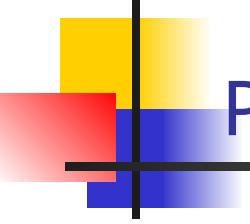
Main approach: **mask errors using redundancy**

- **Information:** add extra information
 - e.g., add error-correction coding, full data replication
- **Time:** re-do computation
- **Physical:** add additional components (SW or HW)
 - Electronic circuits
 - Process replications (replicate servers / components)



Outline

- Fault models
- Process resilience
- Reliable group communication
- Distributed commit



Process Resilience

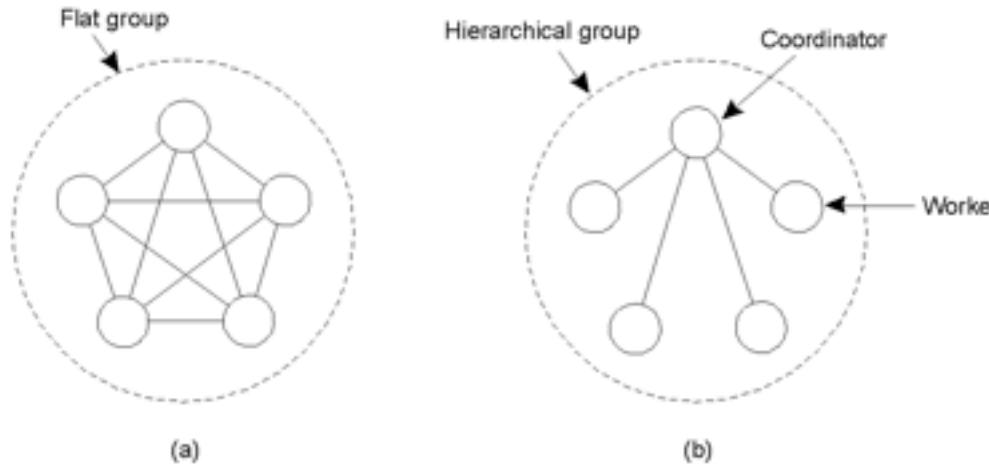
Basic goal: Mask process failures.

Solution: Organize processes as groups

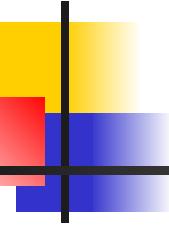
- abstract a collection of processes as a single reliable process
- use “identical” processes
- all members receive and process each messages sent to the group
- Think of it as a replicated state machine

Process Groups: Design choice

- **Flat groups:** Symmetry → good for fault tolerance
 - however, may impose more overhead as control is completely distributed (hard to implement).
- **Hierarchical groups:** communication through a single coordinator
 - not as fault tolerant and scalable, but relatively easy to implement.



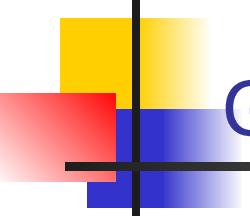
- Issue: group membership



Terminology: a **k-fault tolerant** group can mask any k concurrent failures of group members

- k is the degree of fault tolerance.

Problem: how large a k -fault tolerant group needs to be?

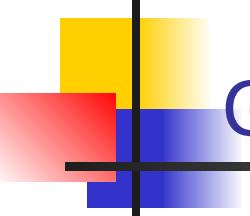


Groups and Failure Masking

Problem: how large a k-fault tolerant group should be?

From a **client perspective** (or hierarchical group):

- If **crash failure semantics** are assumed
 - a total of $k + 1$ members are needed to survive k member failures.
- If **arbitrary (byzantine) failure semantics** are assumed
 - **group output defined by voting collected by the client**
 - a total of $2k+1$ members are needed to survive k member failures



Groups and Failure Masking (2)

Problem: how large a k-fault tolerant group should be?

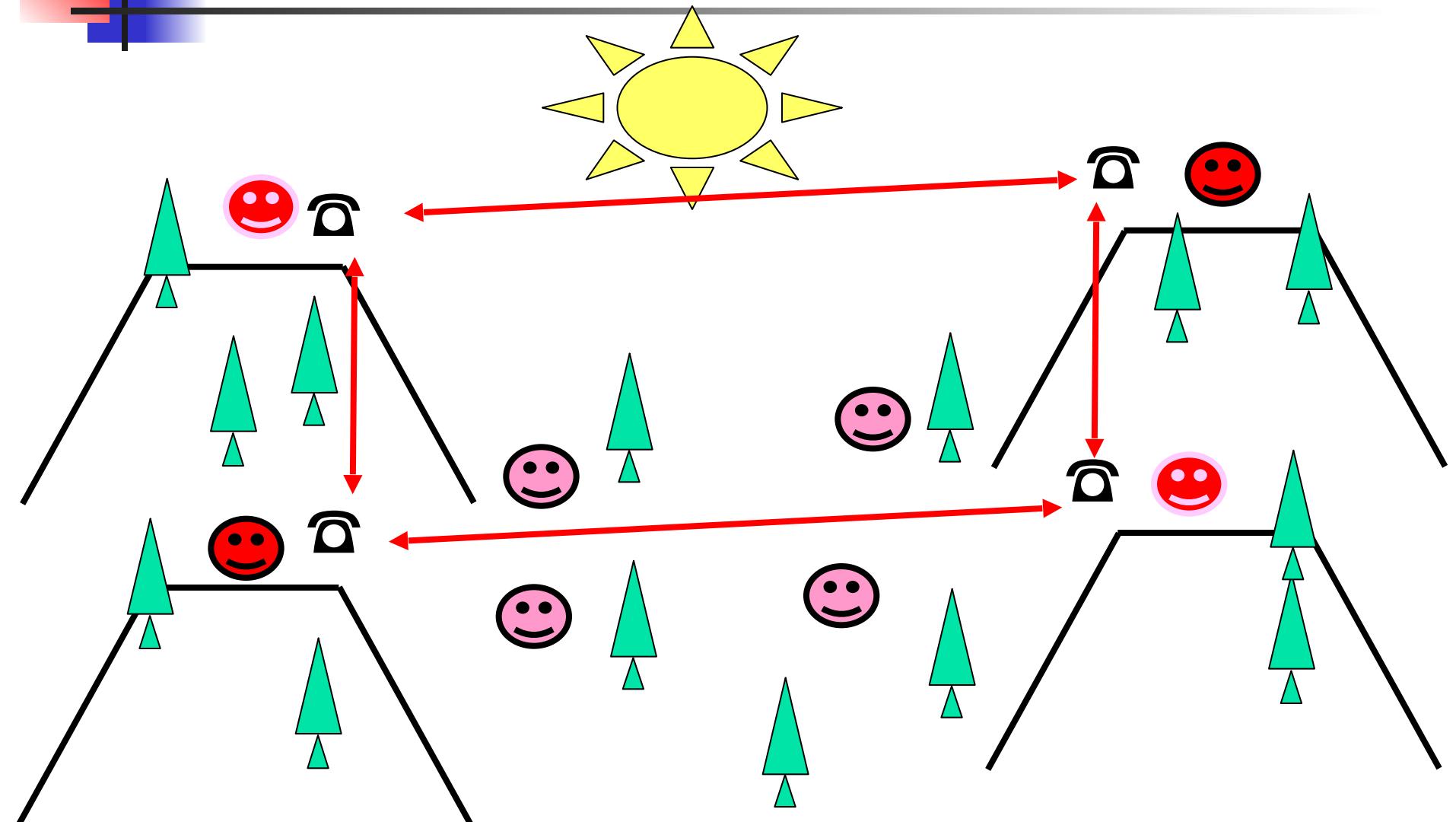
From a **process group** perspective

- Requirement: All correctly functioning group members make the same (correct) decision

Application example: decision making with N controllers

- A plane with N controllers. Each of them:
 - Measures pressure (normally through its own instrument)
 - Gathers pressure estimates from all other controllers
 - Makes a decision based on overall environment conditions
- **Requirement:** Up to K malfunctioning controllers can not influence the decision of the correct controllers

Byzantine Generals Problem



The Problem: Non-faulty group members should reach agreement based on values produced by all other participants

Byzantine generals problem:

N generals aim to attack a castle. Each general observes the situation in the field (eg., troop strength) After sending each other messengers with their observations:

- All loyal generals must reach the same decision (attack/wait)
- The decision should be the one favoured by the loyal generals (based on data gathered from everyone)

Byzantine generals problem:

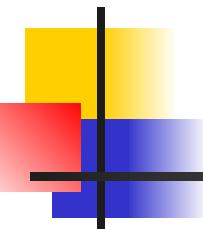
(an equivalent formulation)

One commanding general and $N-1$ lieutenants aim to attack a castle. K traitors in the group (lieutenants / or the commanding general).

- The commanding general gives an order (attack/wait).
- All loyal generals must reach the same decision
 - The decision should be the one issued by the commander if this is loyal.

Setup:

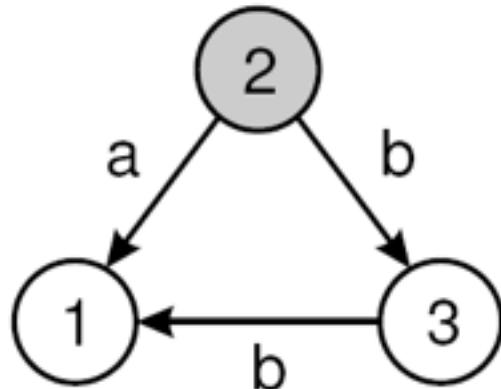
- Reliable, ordered point-to-point communication, bounded delivery time
- Message are not signed.



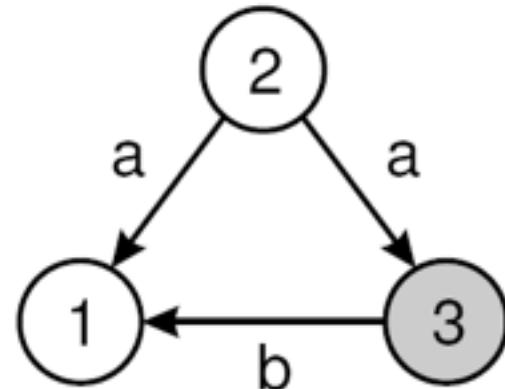
Problem: reaching agreement in a group with
(possibly) faulty processes; faulty communication
channels; **byzantine processes**

What can go wrong?

Process 2 tells
different things



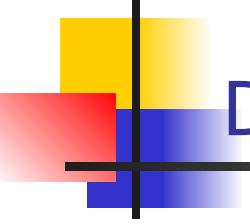
Process 3 passes
a different value



Byzantine Agreement (I)



- 2 loyal generals and one traitor.



Distributed agreement – the general case

Issue: What are the **necessary conditions** for reaching agreement?

Problem space:

- **Process:** Synchronous (operate in lockstep) vs. asynchronous communication
- **Delays:** Are delays on communication bounded?
- **Message ordering:** Are messages delivered in the order they were sent?
- **Channels:** Are channels point-to-point, or multicast?

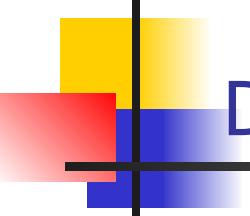
Distributed agreement - the general case

Issue: What are the **necessary conditions** for reaching agreement?

(Note: the textbook has typos on this table)

		Message ordering				Communication delay	
		Unordered		Ordered			
Process behavior		Unicast		Multicast			
		X	X		X	Bounded	
Synchronous				X	X	Unbounded	
Asynchronous					X	Bounded	
					X	Unbounded	

The table illustrates the necessary conditions for distributed agreement based on process behavior and communication characteristics. The columns represent message transmission types (Unicast/Multicast) and message ordering (Unordered/Ordered). The rows represent process behaviors (Synchronous/Asynchronous) and communication delays (Bounded/Unbounded). The presence of an 'X' indicates a necessary condition, while a highlighted cell (Unicast Ordered Multicast) indicates a specific combination that facilitates agreement.

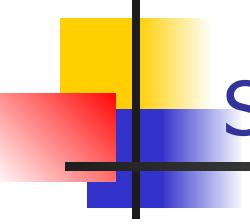


Distributed (Byzantine) agreement (cont)

Main result: Assuming arbitrary failure semantics, we need **$3k + 1$** group members to survive the attack of k faulty members

Intuition: try to reach a majority vote among the group of loyalists, in the presence of k traitors → need $2k + 1$ loyalists.

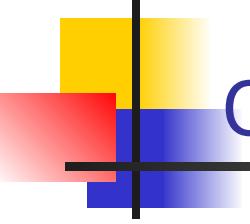
Assumption: unicast communication, synchronous & ordered



Summary so far

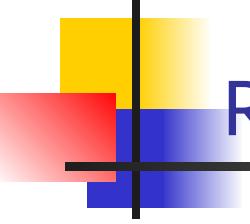
- Fault-tolerance is a must!
- Use redundancy to provide fault tolerance
 - Multiple redundancy techniques: data, time, physical
- Size of a k -fault tolerant process group

		Failure model	
		Fail-stop	Byzantine
Client coordinator			
Group perspective	K+1	2K+1	3K+1



Outline

- Process resilience
- [Already talked about] Reliable client-server communication
- **Reliable (group) communication**
- Distributed commit
- Recovery



Reliable Communication

So far: Concentrated on process resilience (by means of process groups).

Q: What about reliable communication channels?

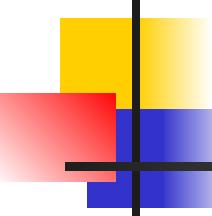
Error detection:

- Framing of packets to enable bit error detection
- Use of frame numbering to detect packet loss

Error correction:

- Add so much redundancy that corrupted packets can be automatically corrected
- Request retransmission of lost, or last N packets

Observation: Most of this work assumes point-to-point communication



[Reminder] Reliable RPC (1/3)

What can go wrong?:

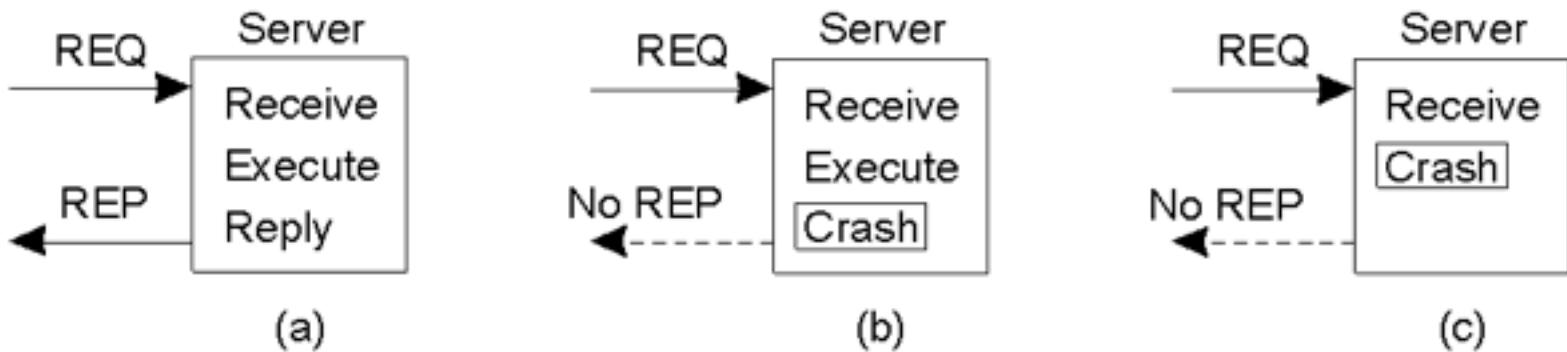
1. Client cannot locate server
2. Client request is lost
3. Server crashes
4. Server response is lost
5. Client crashes

[1:] Client cannot locate server. Relatively simple → just report back to client application

[2:] Client request lost. Just resend message (and use messageID to uniquely identify messages)

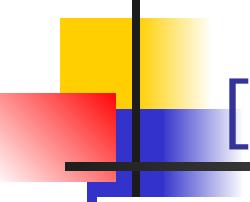
[Reminder] Reliable RPC (2/3)

[3] Server crashes → harder as you don't know what the server has already done:



Problem: decide on what to expect from the server

- **At-least-once-semantics:** The server guarantees it will carry out an operation at least once, no matter what.
- **At-most-once-semantics:** The server guarantees it will carry out an operation at most once.



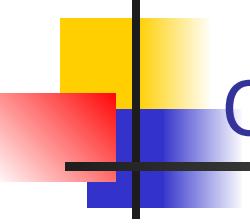
[Reminder] Reliable RPC (3/3)

[4:] **Lost replies** → Detection hard: because it can also be that the server had crashed. You don't know whether the server has carried out the operation

- **Solution:** None,
 - (works sometimes) make your operations **idempotent**: repeatable without any harm done if it happened to be carried out before.

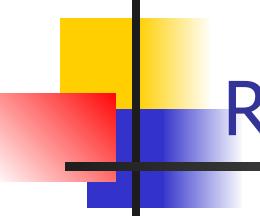
[5:] **Client crashes** → The server is doing work and holding resources for nothing (called doing an **orphan** computation).

- Orphan is killed by client when it reboots
- Broadcast new epoch number when recovering → servers kill orphans
- Require computations to complete in a T time units. Old ones are simply removed.



Outline

- Process resilience
- [Already talked about] Reliable client-server communication
- **Reliable group communication**
- Distributed commit
- Recovery



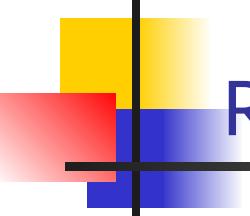
Reliable Multicasting (roadmap)

Model: a multicast channel c with two (possibly overlapping) groups:

- The sender group $\text{SND}(c)$ of processes that submit messages to channel c
- The receiver group $\text{RCV}(c)$ of processes that receive messages from channel c

Possible reliability requirements:

- **Simple reliability:** No messages lost
 - If process $P \in \text{RCV}(c)$ at the time message m was submitted to c , and P does not leave $\text{RCV}(c)$, m should be delivered to P
- **Virtually synchronous multicast:** All active processes receive 'the same thing'
 - Ensure that a message m submitted to channel c is delivered to process $P \in \text{RCV}(c)$ only if m is delivered to all members of $\text{RCV}(c)$



Reliable Multicasting (2/2)

Observation: If we can stick to a local-area network, reliable multicasting is 'easy'

Principle: Let the sender log messages submitted to channel

C:

- If P sends message m, m is stored in a **history buffer**
 - Each receiver acknowledges the receipt of m, and requests retransmission at P when noticing message loss
 - Sender P removes m from history buffer when everyone has acknowledged receipt
-
- **Question:** Why doesn't this scale?

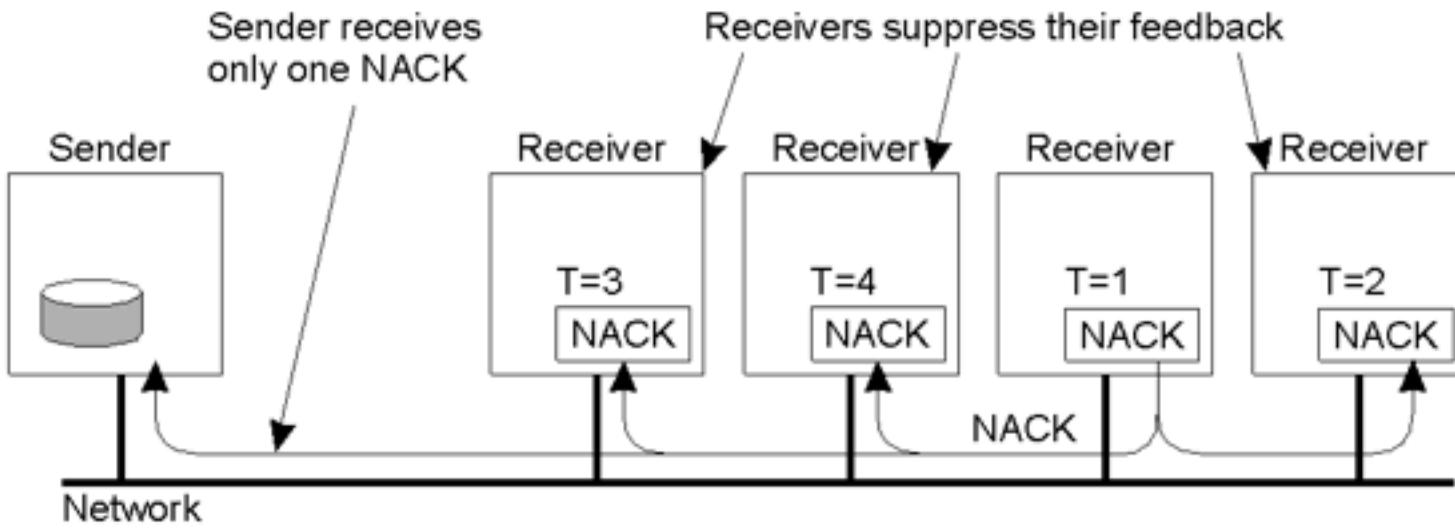
Scalable Reliable Multicasting: Feedback Suppression

Basic idea: A process P suppresses its own feedback when it notices another process Q is already asking for a retransmission

Assumptions:

- All receivers listen to a common feedback channel to which feedback messages are submitted
- Process P schedules its own feedback message randomly, and suppresses it when observing another feedback message

Question: Why is the random schedule so important?

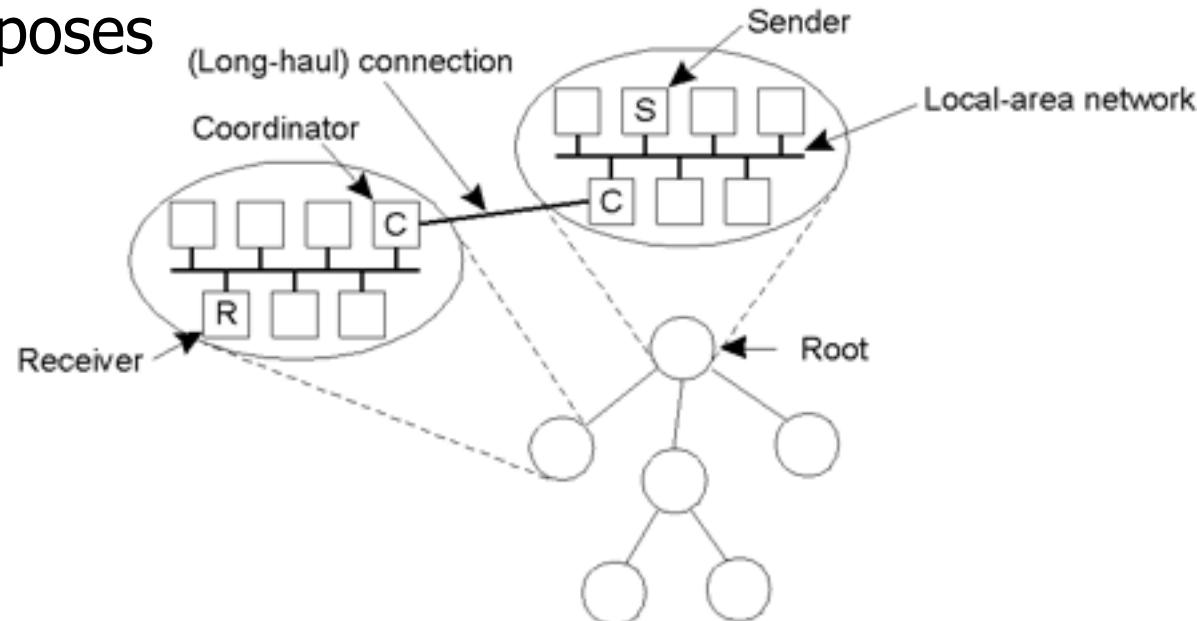


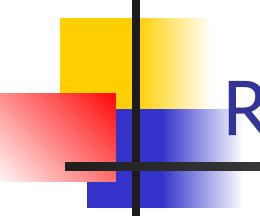
Scalable Reliable Multicasting: Hierarchical Solutions

Basic solution: Construct a hierarchical feedback channel in which all submitted messages are sent only to the root. Intermediate nodes aggregate feedback messages before passing them on.

Question: What's the main problem with this solution?

Observation: Intermediate nodes can easily be used for retransmission purposes





Reliable Multicasting (reoadmap)

Model: a multicast channel c with two (possibly overlapping) groups:

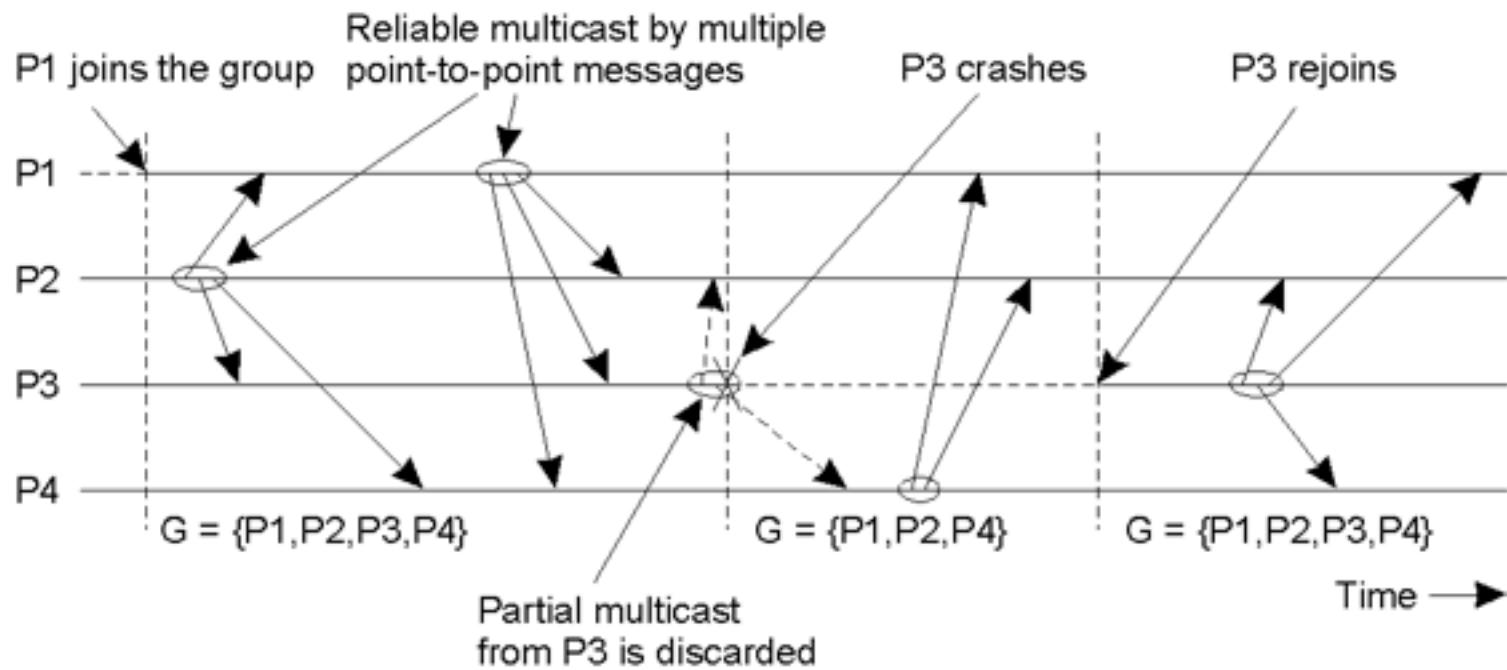
- **The sender group** $\text{SND}(c)$ of processes that submit messages to channel c
- **The receiver group** $\text{RCV}(c)$ of processes that receive messages from channel c

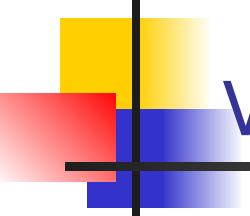
Possible reliability requirements:

- **Simple reliability:** No messages lost
 - If process $P \in \text{RCV}(c)$ at the time message m was submitted to c , and P does not leave $\text{RCV}(c)$, m should be delivered to P
- **Virtually synchronous multicast:** All active processes receive 'the same thing'
 - Ensure that a message m submitted to channel c is delivered to process $P \in \text{RCV}(c)$ only if m is delivered to all members of $\text{RCV}(c)$

Virtual Synchronous Multicast

- **Goal:** Formulate reliable multicasting in the presence of process failures in terms of process groups and changes to group membership
- **Guarantee:** A message is delivered only to the non-faulty members of the current group. All members should agree on the current group membership.



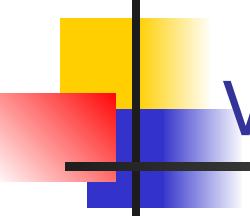


Virtual Synchrony

Notation: Consider views $V = RCV(c) \cup SND(c)$

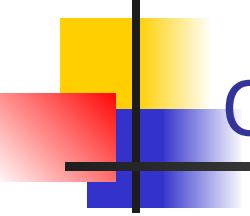
Properties of virtually synchronous multicast:

- For each consistent state, there is a unique view on which all its members agree.
 - Note: implies that **all non-faulty processes see all view changes in the same order**
- If message m is sent to V before a view change then either all $P \in V$ that execute vc receive m , or no processes $P \in V$ that execute vc receive m .
 - Note: **all non-faulty members in the same view get to see the same set of multicast messages.**
- A message sent to view V can be delivered only to processes in V
 - and is discarded by the following views



Virtual Synchrony – Notes

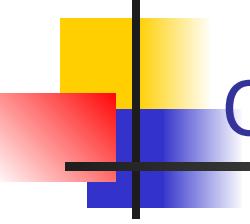
- If a sender $S \in V$ crashes, its multicast message m is flushed before S is removed from V : m will never be delivered after the point that V changes
 - Note: Messages from S may still be delivered to all, or none (nonfaulty) processes in V before they all agree on a new view to which S does not belong
- If a receiver P fails, a message m may be lost but can be recovered as we know exactly what has been received in V .
 - Alternatively, we may decide to change the view and deliver m to members in $V - \{P\}$



Orthogonal concern: Message Ordering (1/2)

- Observation: Virtually synchronous behavior is independent from the ordering of message delivery.
 - The only issue is that messages are delivered to an agreed upon group of receivers.
- Example: Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

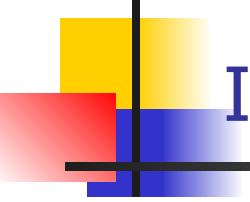
Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	



Orthogonal concern: Message ordering (2)

Six different possibilities of virtually synchronous reliable multicasting.

Multicast	Basic Message Ordering	Total-ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes



Implementing Virtual Synchrony (1/4)

Assumptions:

- Point-to-point communication in the underlying network:
 - Reliable, in-order delivery (TCP-like semantics)
- Multicast implemented as a sequence of point-to-point transmissions
 - But sender can fail before sending to all receivers

Requirements

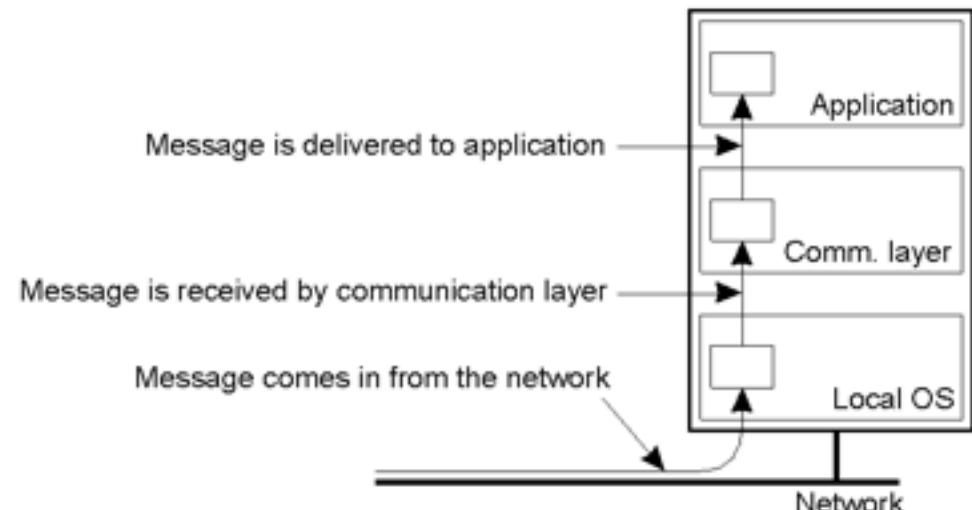
- All messages send while in view G_i are delivered to all non-faulty processes in G_i before the next group membership change
 - G_{i+1} might be installed before m is delivered

Issues

- How to detect a process is missing a message

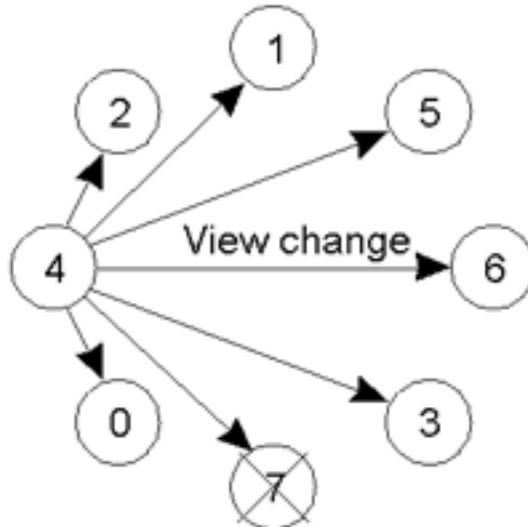
Implementing Virtual Synchrony (2/3)

- Setup
- Solution clue:
 - Every process in view G_i keeps m until it knows for sure that all other members in G_i have received m
 - Terminology: m is **stable** if received by all processes in G_i
 - Only stable messages are delivered

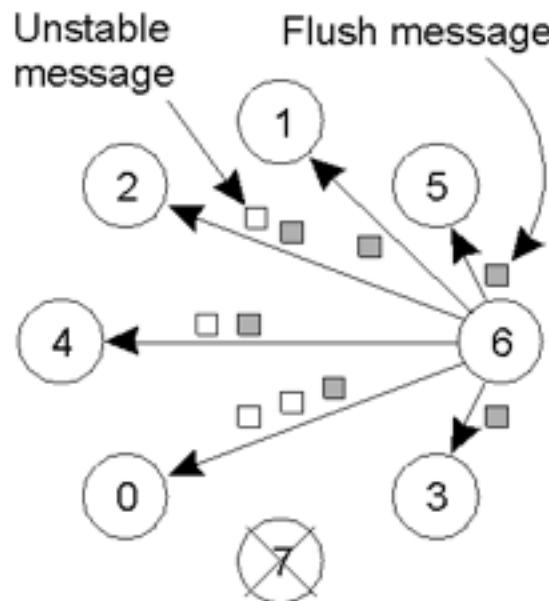


Implementing Virtual Synchrony (2/3)

- Solution clue: Only stable messages are delivered

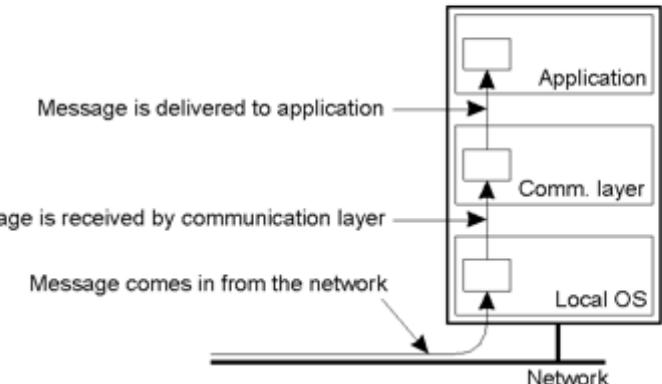
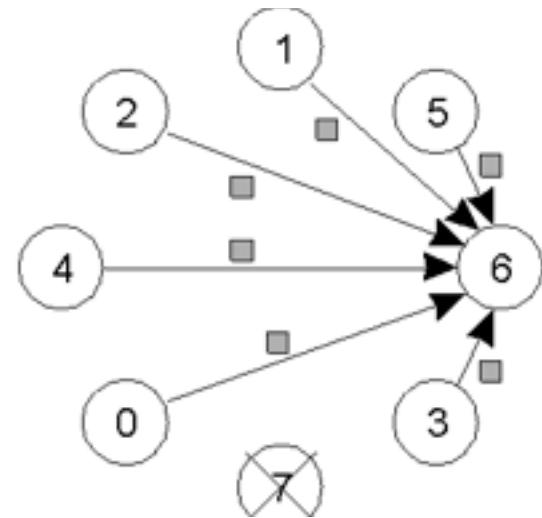


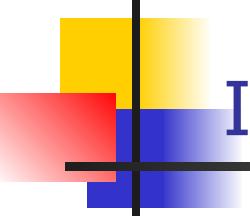
4 notices that 7 has crashed and generates a 'view change' message



6 sends all its unstable messages followed by a flush message

6 installs a new view once it has received a flush message from everyone.

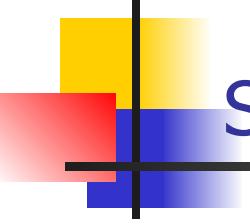




Implementing Virtual Synchrony (3/3)

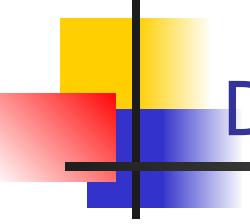
Algorithm sketch

- P detects a view change (node join / leave)
 - Forwards any unstable message in G_i to all processes in G_i
 - P sends a 'flush' message
 - P collects a 'flush response' from everyone
 - P installs new view
- Q (another process)
 - When receiving m in the view G_i (the current one) it delivers it
 - (after ordering)
 - Logs the message (it is not yet stable)
 - When receiving a flush message
 - Multicasts all its unstable messages
 - Sends the 'flush response'
 - Installs new view
- Control messages so that each process knows what are the messages received by everyone else



Summary so far

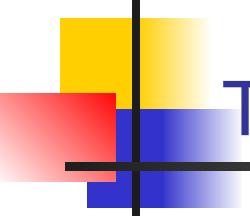
- Terminology
 - Fault types
 - Failure models
- Process resilience
 - Process groups, K-fault tolerance
 - Agreement in distributed systems
- Reliable group communication
- Distributed commit



Distributed commit

Essential issue: Given a computation distributed across a process group, how can we ensure that either all processes commit to the final result, or none of them do (**atomicity**)?

- Two-phase commit algorithms
- Three-phase commit algorithms



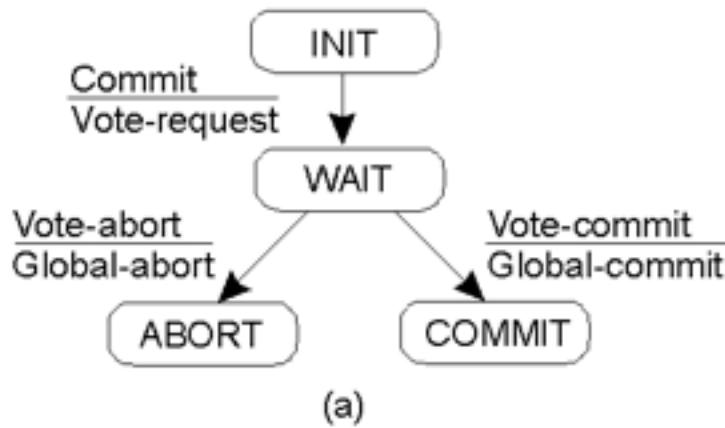
Two-Phase Commit (1/2)

Model: The client who initiated the computation acts as coordinator; processes required to commit are the participants

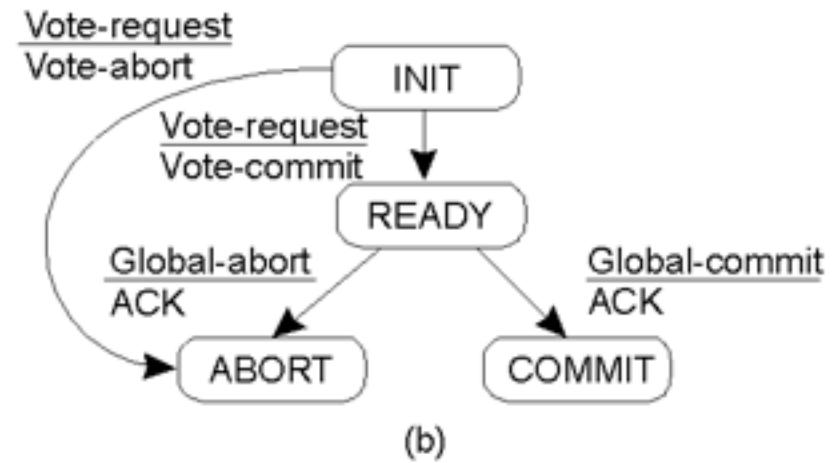
- **Phase 1a:** Coordinator sends `vote_REQUEST` to participants (also called a **pre-write**)
- **Phase 1b:** When participant receives `vote_REQUEST` it returns either YES or NO to coordinator. If it sends NO, it aborts its local computation
- **Phase 2a:** Coordinator collects all votes; if all are YES, it sends `COMMIT` to all participants, otherwise it sends `ABORT`
- **Phase 2b:** Each participant waits for `COMMIT` or `ABORT` and handles accordingly.

Two-Phase Commit – Finite State Machine

Coordinator



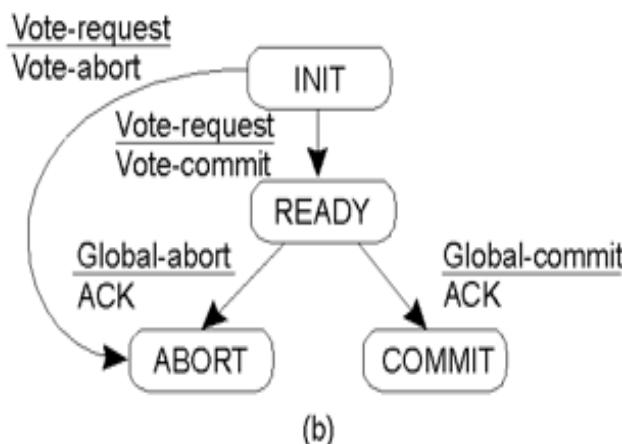
Participant



2PC → Failing Participant (1/2)

Scenario: Consider participant crash in one of its states, and the subsequent recovery to that state

- **Initial state:** No problem, as participant was unaware of the protocol
- **Ready state:** Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make → log the coordinator's decision
- **Abort state:** Merely make entry into abort state idempotent, e.g., removing the workspace of results
- **Commit state:** Also make entry into commit state idempotent, e.g., saving workspace to storage.



Intuition: When distributed commit is required, having participants use temporary workspaces to keep their results facilitates simple recovery in the presence of failures.

2PC → Failing Participant (2/2)

Alternative: When a recovery is needed to the Ready state, check what the other participants are doing.

- This approach avoids having to log the coordinator's decision.

Assume recovering participant P contacts another participant Q:

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Result: If all participants are in the ready state, the protocol blocks. Apparently, the coordinator is failing.

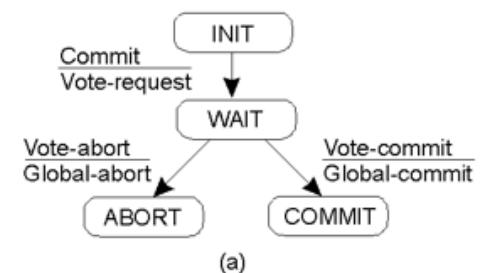
2PC → Failing Coordinator

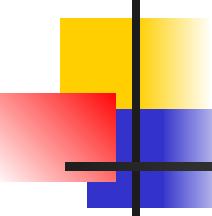
Observation: The real problem lies in the fact that the coordinator's final decision may not be available for some time (or actually lost)

Alternative: Let a participant P in the ready state timeout when it hasn't received the coordinator's decision; P tries to find out what other participants know.

Question: Can P not succeed in getting the required information?

Observation: Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes





Three-Phase Commit (1/2)

- **Phase 1a:** Coordinator sends `vote_REQUEST` to participants
- **Phase 1b:** When participant receives `vote_REQUEST` it returns either YES or NO to coordinator. If it sends NO, it aborts its local computation
- **Phase 2a:** Coordinator collects all votes; if all are YES, it sends `PREPARE` to all participants, otherwise it sends `ABORT`, and halts
- **Phase 2b:** Each participant waits for `PREPARE`, or waits for `ABORT` after which it halts
- **Phase 3a:** (Prepare to commit) Coordinator waits until all participants have ACKed receipt of `PREPARE` message, and then sends `COMMIT` to all
- **Phase 3b:** (Prepare to commit) Participant waits for `COMMIT`

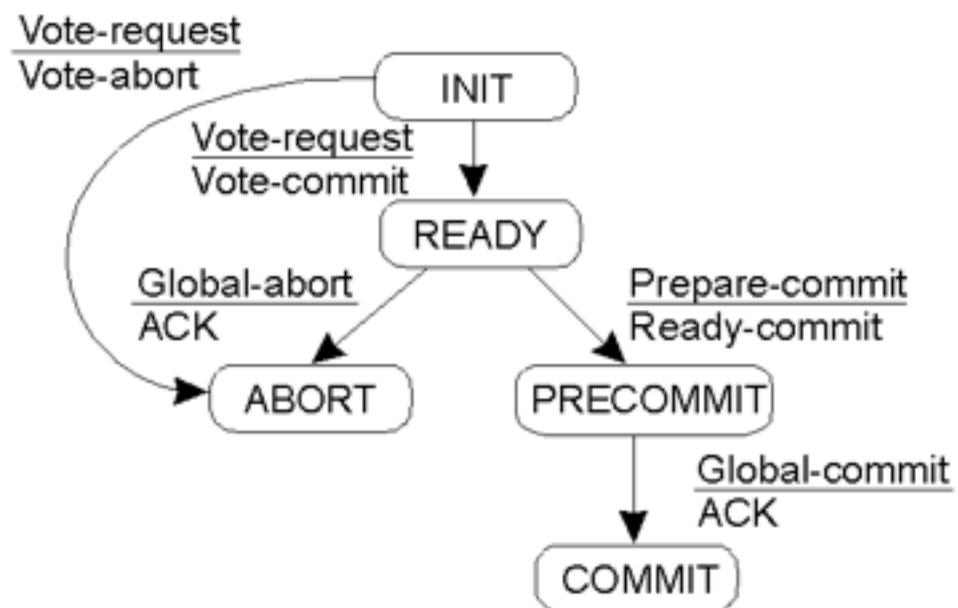
3PC → Finite state machines

Coordinator



(a)

Participant



(b)

3PC → Failing Participant

Problem: Can P find out what it should do after a crash in the ready or pre-commit state, even if other participants or the coordinator failed?

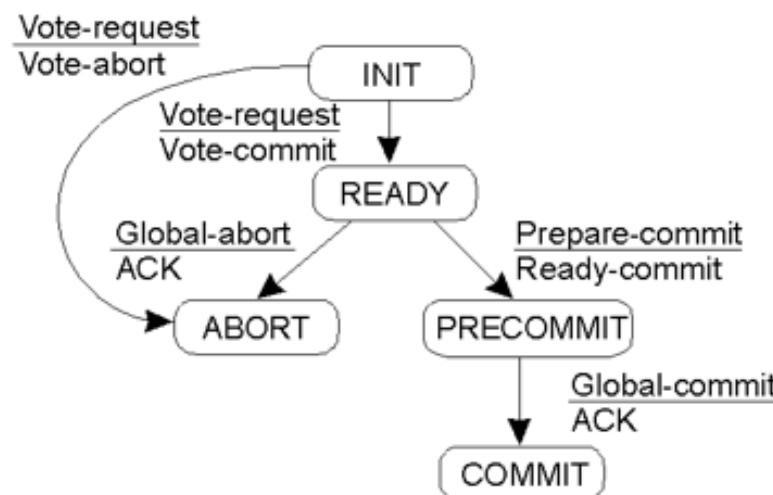
Solution idea: Coordinator and participants on their way to commit, never differ by more than one state transition

Consequences

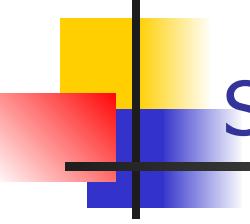
- If a participant times out in ready state, it can find out at the coordinator or other participants whether it should abort, or enter pre-commit state.
- If a participant already made it to the pre-commit state, it can always safely commit (but is not allowed to do so for the sake of failing other processes)



(a)



(b)



Summary

- Terminology
 - Fault types
 - Failure models
- Process resilience
 - Process groups, K-fault tolerance
 - Agreement in distributed systems
- Reliable group communication
- Distributed commit