CC5212-1
PROCESAMIENTO MASIVO DE DATOS
OTOÑO 2015

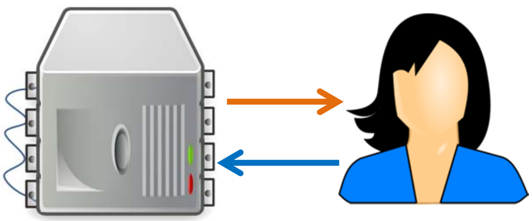
Lecture 3: Distributed Systems II

Aidan Hogan
aidhog@gmail.com

TYPES OF DISTRIBUTED SYSTEMS …

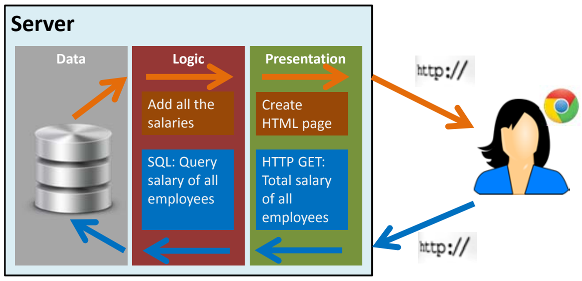
Client–Server Model

- Client makes request to server
- Server acts and responds

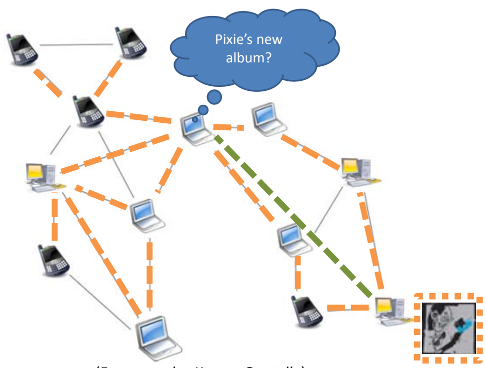


(For example: *Email, WWW, Printing, etc.*)

Client–Server: *Three-Tier Server*



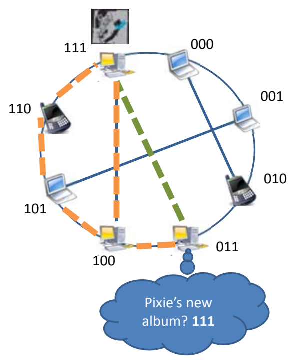
Peer-to-Peer: *Unstructured*



(For example: *Kazaa, Gnutella*)

Peer-to-Peer: *Structured (DHT)*

- Circular DHT:
  - Only aware of neighbours
  - O(*n*) *lookups*
- Implement shortcuts
  - Skips ahead
  - Enables binary-search-like behaviour
  - O(log(*n*)) *lookups*



Desirable Criteria for Distributed Systems

- Transparency:
  - Appears as one machine
- Flexibility:
  - Supports more machines, more applications
- Reliability:
  - System doesn't fail when a machine does
- Performance:
  - Quick runtimes, quick processing
- Scalability:
  - Handles more machines/data efficiently

Eight Fallacies (to avoid)

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Severity of fallacies vary in different scenarios!
Which fallacies apply/do not apply for:

- Gigabit ethernet LAN?
- BitTorrent
- The Web

LIMITATIONS OF DISTRIBUTED COMPUTING: CAP THEOREM

But first … ACID

Have you heard of ACID guarantees in a database class?

For traditional (non-distributed) databases …

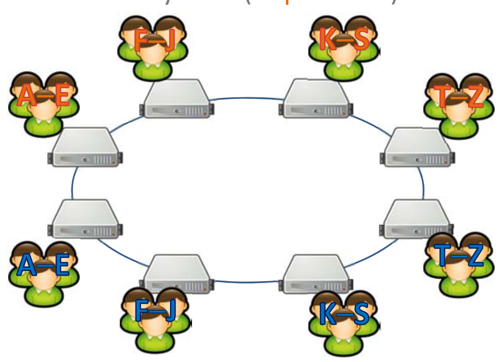
1. **A**tomicity:
   - Transactions all or nothing: fail cleanly
2. **C**onsistency:
   - Doesn't break constraints/rules
3. **I**solation:
   - Parallel transactions act as if sequential
4. **D**urability
   - System remembers changes

What is CAP?

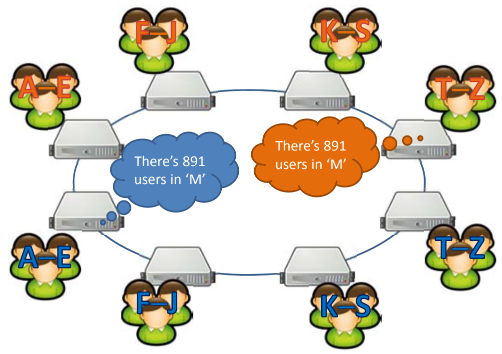
Three *guarantees* a distributed sys. could make

1. **C**onsistency:
   - All nodes have a consistent view of the system
2. **A**vailability:
   - Every read/write is acted upon
3. **P**artition-tolerance:
   - The system works even if messages are lost

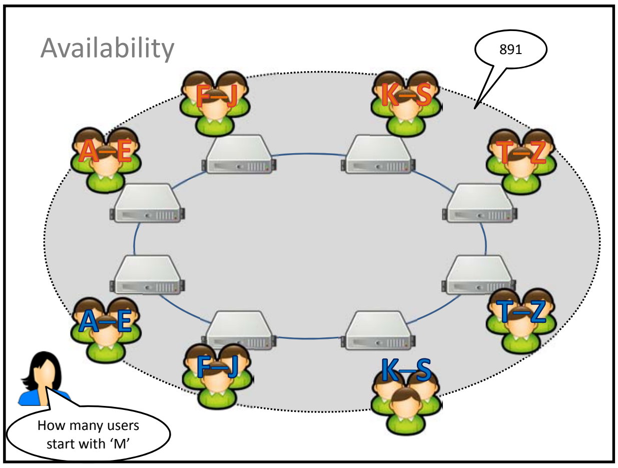
A Distributed System (Replication)



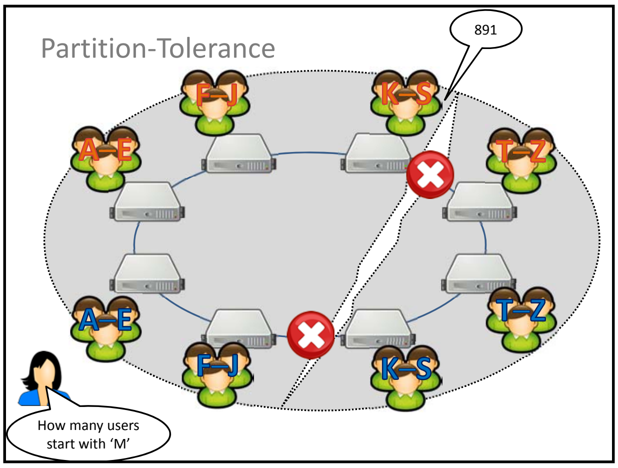
Consistency



Availability



Partition-Tolerance



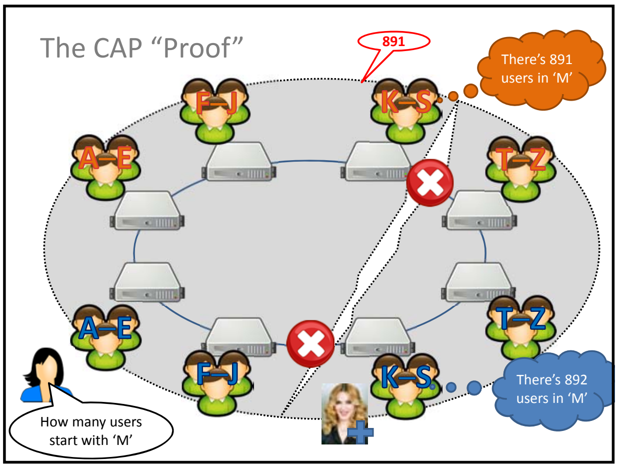
The CAP Question

Can a distributed system guarantee consistency (all nodes have the same up-to-date view), availability (every read/write is acted upon) and partition-tolerance (the system works even if messages are lost) at the same time?

What do you think?

The CAP Answer

The CAP "Proof"



The CAP "Proof" (in boring words)

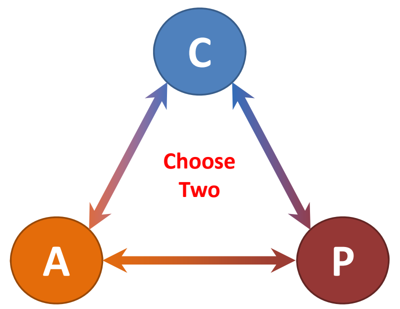
- Consider machines m_1 and m_2 on either side of a partition:
  - If an update is allowed on m_2 (Availability), then m_1 cannot see the change: (loses Consistency)
  - To make sure that m_1 and m_2 have the same, up-to-date view (Consistency), neither m_1 nor m_2 can accept any requests/updates (lose Availability)
  - Thus, only when m_1 and m_2 can communicate (lose Partition tolerance) can Availability and Consistency be guaranteed

The CAP Theorem

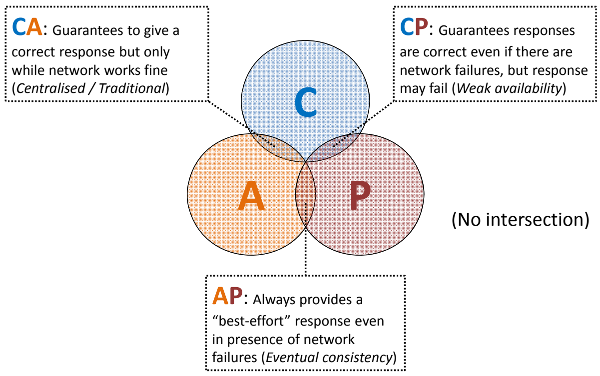
A distributed system cannot guarantee consistency (all nodes have the same up-to-date view), availability (every read/write is acted upon) and partition-tolerance (the system works even if messages are lost) at the same time.

("Proof" as shown on previous slide ☺)

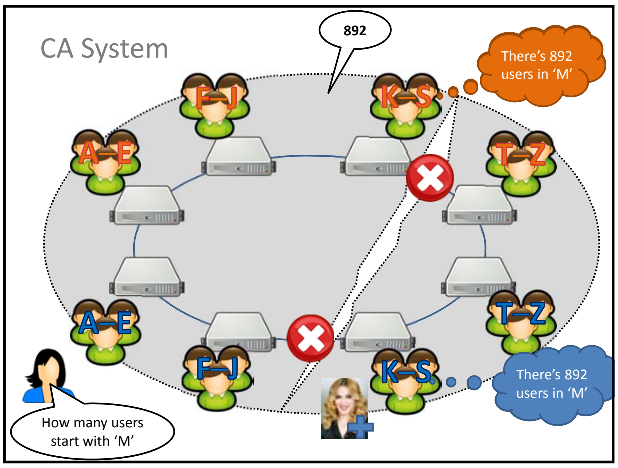
The CAP Triangle



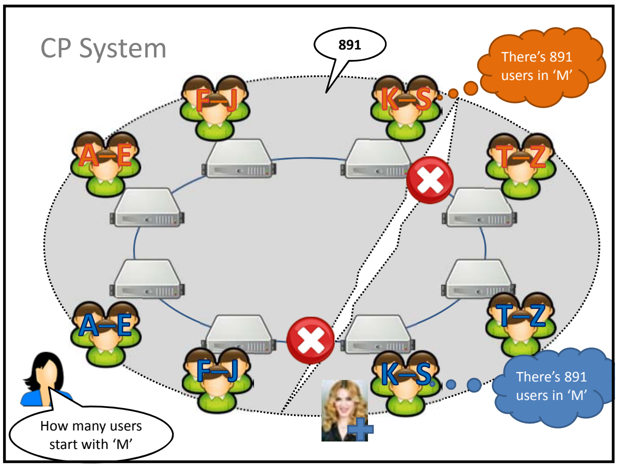
CAP Systems



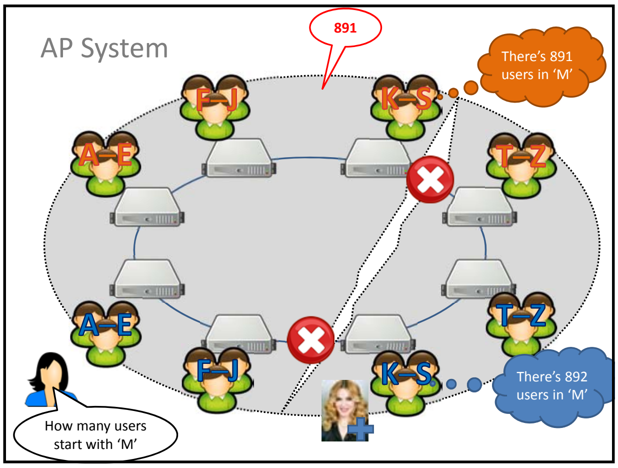
CA System



CP System



AP System



BASE (AP)

In what way was Twitter operating under BASE-like conditions?

- **B**asically **A**vailable
  - Pretty much always "up"
- **S**oft State
  - Replicated, cached data
- **E**ventual Consistency
  - Stale data tolerated, for a while

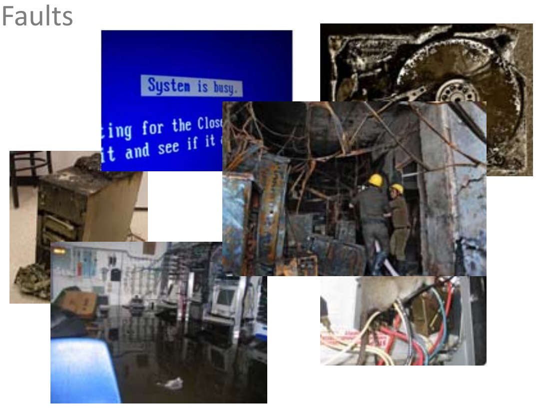
The CAP Theorem

⚠

- C,A in CAP ≠ C,A in ACID
- Simplified model
  - Partitions are rare
  - Systems may be a mix of CA/CP/AP
  - C/A/P often continuous in reality!
- But concept useful/frequently discussed:
  - How to handle Partitions?
    - Availability? or
    - Consistency?

FAULT TOLERANCE / CONSENSUS SYNCHRONOUS VS. ASYNCHRONOUS

Faults



Synchronous vs. Asynchronous

- Synchronous distributed system:
  - Messages expected by a given time
    - E.g., a clock tick
  - Missing message has meaning
- Asynchronous distributed system:
  - Messages can arrive at any time
    - Delay is finite but not known
  - Missing message could arrive any time!

Lunch Problem

Bob

10:30AM. Alice, Bob and Chris work in the same city. All three have agreed to go downtown for lunch today but have yet to decide on a place and a time.

Alice

Chris

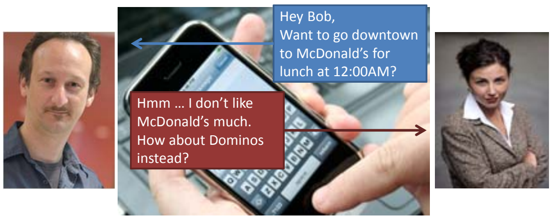
Asynchronous Consensus: Texting

10:45 AM. *Alice tries to invite Bob for lunch …*

11:35 AM. *No response. Should Alice head downtown?*

Asynchronous Consensus: Texting

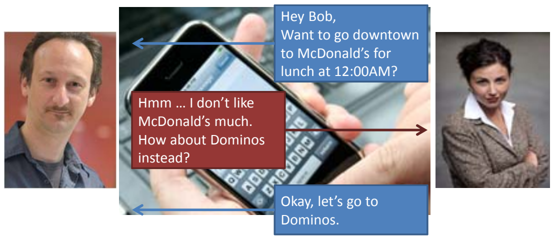
10:45 AM. *Alice tries to invite Bob for lunch …*



11:42 AM. *No response. Where should Bob go?*

Asynchronous Consensus: Texting

10:45 AM. *Alice tries to invite Bob for lunch …*



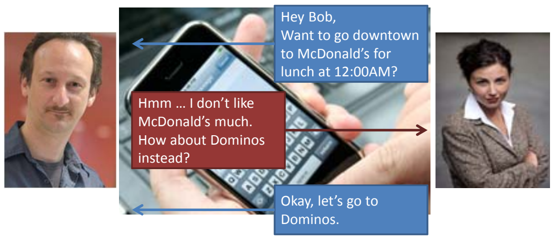
11:38 AM. *No response. Did Bob see the acknowledgement?*

Asynchronous Consensus

- Impossible to guarantee!
  - A message delay can happen at any time and a node can wake up at the wrong time!
  - Fischer-Lynch-Patterson (1985): No consensus can be guaranteed if there is even a single failure
- But asynchronous consensus can happen
  - As you should realise if you've ever successfully organised a meeting by email or text ;)

Asynchronous Consensus: Texting

10:45 AM. *Alice tries to invite Bob for lunch …*



11:38 AM. *No response. Bob's battery died. Alice misses the train downtown waiting for message, heads to the cafeteria at work instead. Bob charges his phone …*

Heading to Dominos now. See you there!

Asynchronous Consensus: Texting

How could Alice and Bob find consensus on a time and place to meet for lunch?

Synchronous Consensus: Telephone

10:45 AM. *Alice tries to invite Bob for lunch …*

10:46 AM. **Clear consensus!**

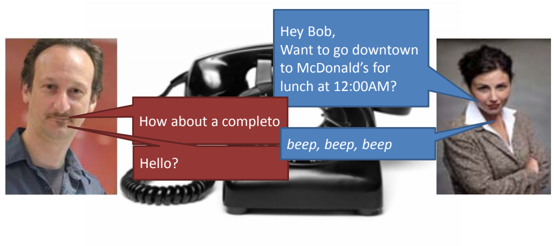
Synchronous Consensus

- Can be guaranteed!
  - But only under certain conditions …

What is the core difference between reaching consensus in synchronous (texting or email) vs. asynchronous (phone call) scenarios?

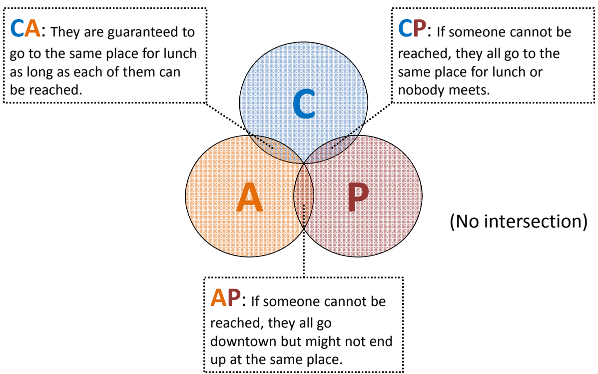
Synchronous Consensus: Telephone

10:45 AM. *Alice tries to invite Bob for lunch …*



10:46 AM. **What's the protocol?**

CAP Systems (for example …)



A Consensus Protocol

- Agreement/Consistency [Safety]: All working nodes agree on the same value. Anything agreed is final!
- Validity/Integrity [Safety]: Every working node decides at most one value. That value has been proposed by a working node.
- Termination [Liveness]: All working nodes eventually decide (after finite steps).

- Safety: Nothing bad ever happens
- Liveness: Something good eventually happens

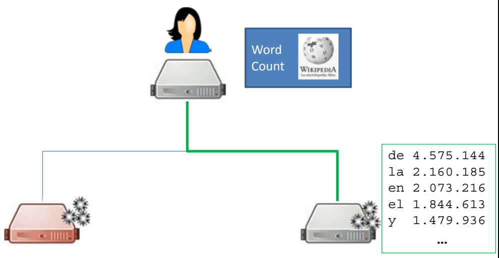
A Consensus Protocol for Lunch

- Agreement/Consistency [Safety]: Everyone agrees on the same place downtown for lunch, or agrees not to go downtown.
- Validity/Integrity [Safety]: Agreement involves a place someone actually wants to go.
- Termination [Liveness]: A decision will eventually be reached (hopefully before lunch).

FAULT TOLERANCE: FAIL–STOP VS. BYZANTINE

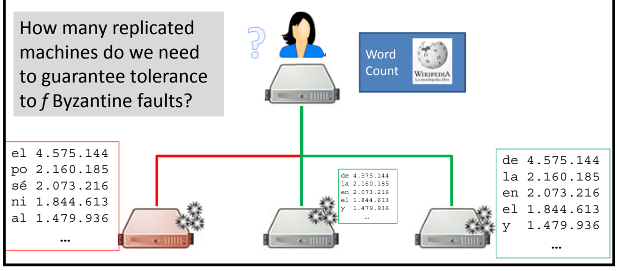
Fail–Stop Fault

- A machine fails to respond or times-out (often hardware or load)
- Need *at least* f+1 replicated machines? (beware asynch.!)
  - f = number of clean failures



Byzantine Fault

- A machine responds incorrectly/maliciously (often software)
- Need *at least* **2**f+1 replicated machines?
  - f = number of (possibly Byzantine) failures



Fail–Stop/Byzantine

- Naively:
  - Need f+1 replicated machines for fail–stop
  - Need 2f+1 replicated machines for Byzantine
- Not *so* simple if nodes must agree beforehand!
- **Replicas must have consensus to be useful!**

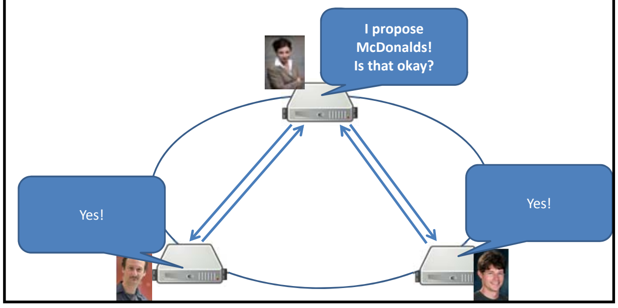
CONSENSUS PROTOCOL: TWO-PHASE COMMIT

Two-Phase Commit (2PC)

- Coordinator & cohort members
- **Goal:** Either all cohorts commit to the same value or no cohort commits to anything
- Assumes synchronous, fail-stop behaviour
  - Crashes are known!

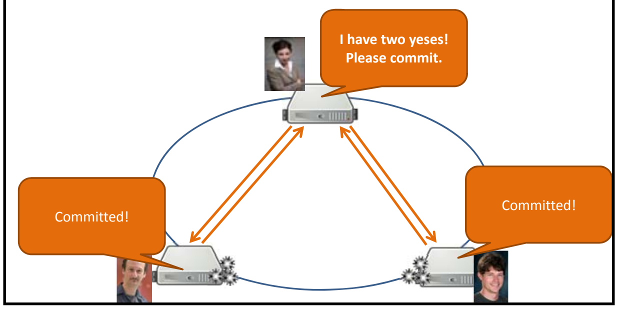
Two-Phase Commit (2PC)

1. Voting:



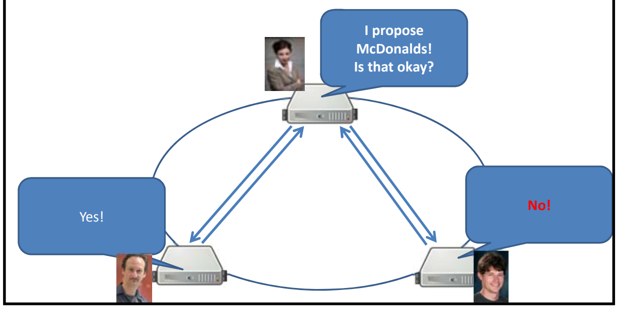
Two-Phase Commit (2PC)

2. Commit:



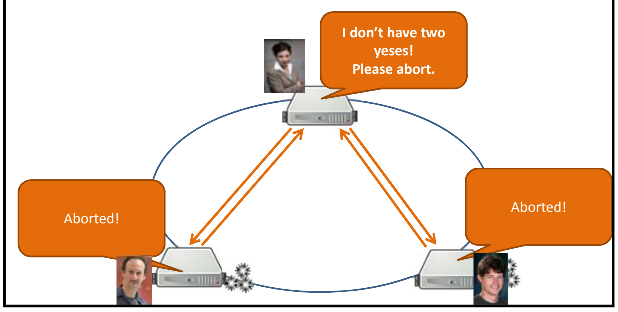
Two-Phase Commit (2PC) [Abort]

1. Voting:



Two-Phase Commit (2PC) [Abort]

2. Commit:



Two-Phase Commit (2PC)

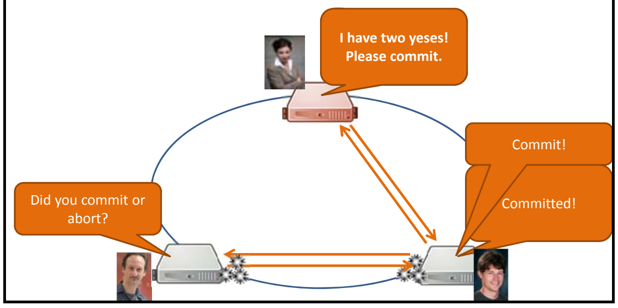
1. Voting: A coordinator proposes a commit value. The other nodes vote "yes" or "no" (they cannot propose a new value!).
2. Commit: The coordinator counts the votes. If all are "yes", the coordinator tells the nodes to accept (commit) the answer. If one is "no", the coordinator aborts the commit.

- For *n* nodes, in the order of 4*n* messages.
  - 2*n* messages to propose value and receive votes
  - 2*n* messages to request commit and receive acks

Two-Phase Commit (2PC)

What happens if the coordinator fails?

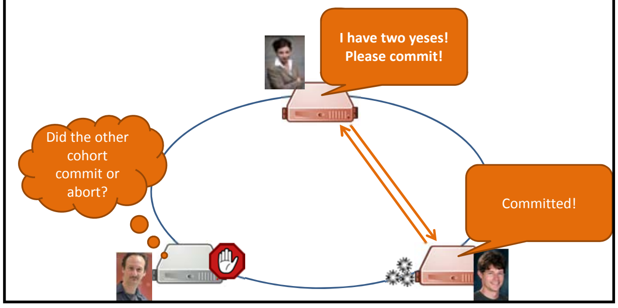
- Cohort members know coordinator has failed!



Two-Phase Commit (2PC)

What happens if a coordinator and a cohort fail?

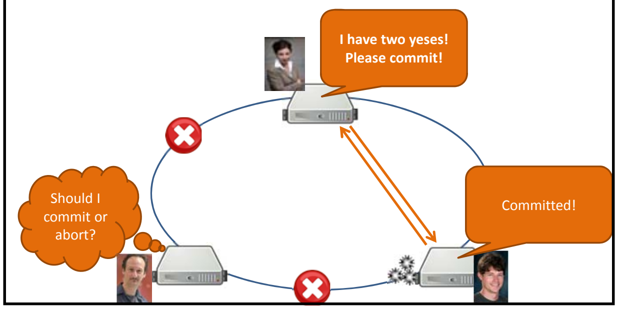
Not fault-tolerant!



Two-Phase Commit (2PC)

What happens if there's a partition?

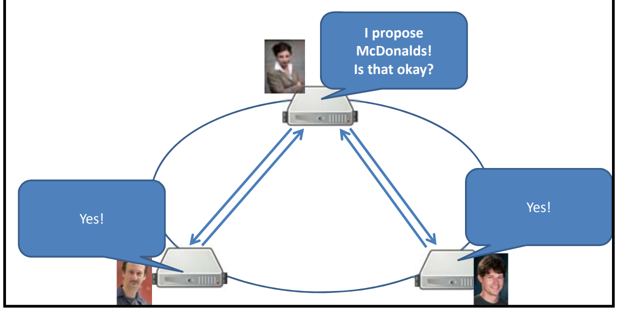
Not fault-tolerant!



CONSENSUS PROTOCOL: THREE-PHASE COMMIT

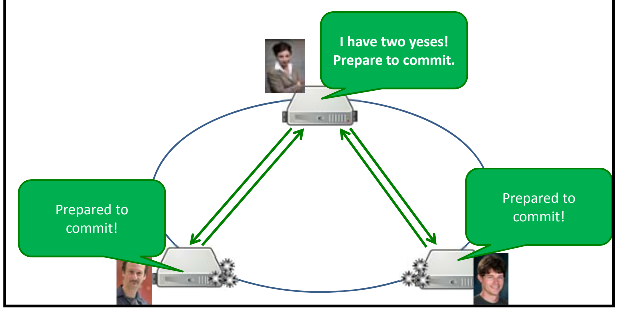
Three-Phase Commit (3PC)

1. Voting:



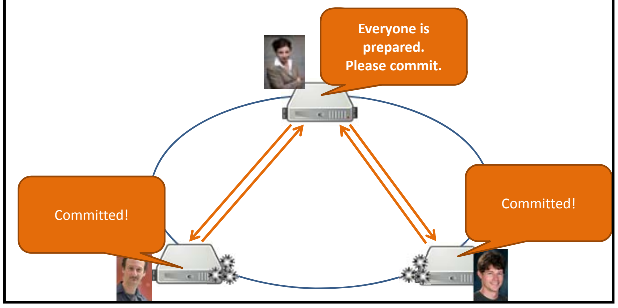
Three-Phase Commit (3PC)

2. Prepare:



Three-Phase Commit (3PC)

3. Commit:



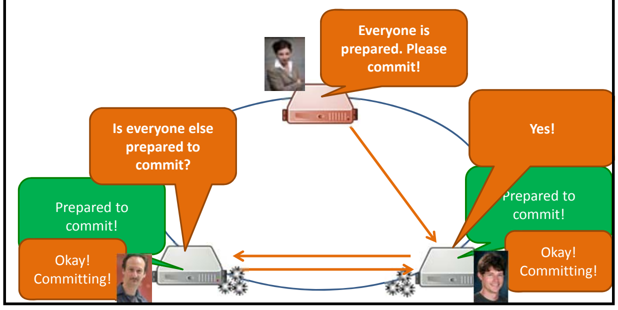
Three-Phase Commit (3PC)

1. Voting: (As before for 2PC)
2. Prepare: If all votes agree, coordinator sends and receives acknowledgements for a "prepare to commit" message
3. Commit: If all acknowledgements are received, coordinator sends "commit" message

- For *n* nodes, in the order of 6*n* messages.
  - 4*n* messages as for 2PC
  - +2*n* messages for "prepare to commit"+ "ack."

Three-Phase Commit (3PC)

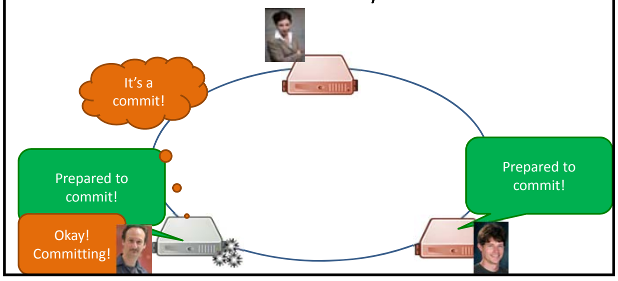
What happens if the coordinator fails?



Three-Phase Commit (3PC)

What happens if coordinator and a cohort member fail?

- Rest of cohort know if abort/commit!



Two-Phase vs. Three Phase

Did you spot the difference?

- In 2PC, in case of failure, one cohort may already have committed/aborted while another cohort doesn't even know if the decision is commit or abort!
- In 3PC, this is not the case!

Two/Three Phase Commits

- Assumes synchronous(-like) behaviour!
- Assumes knowledge of failures!
  - Cannot be guaranteed if there's a network partition!
- Assumes fail–stop errors

How to decide the leader?

We need a leader for consensus … so what if we need consensus for a leader?

CONSENSUS PROTOCOL: PAXOS

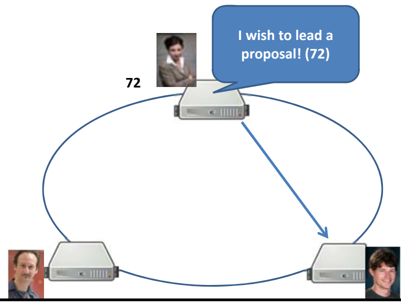
Turing Award: Leslie Lamport

- One of his contributions: PAXOS

LESLIE LAMPORT
United States – 2013
CITATION
For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

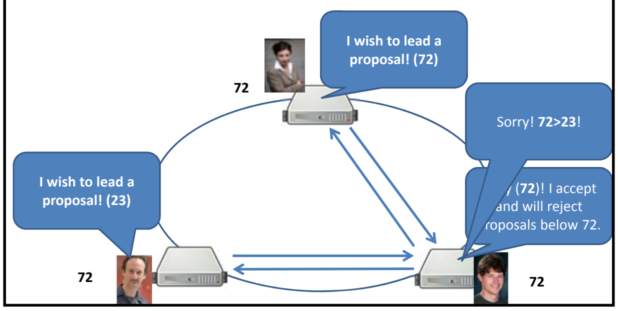
PAXOS Phase 1a: Prepare

- A coordinator proposes with a number *n*



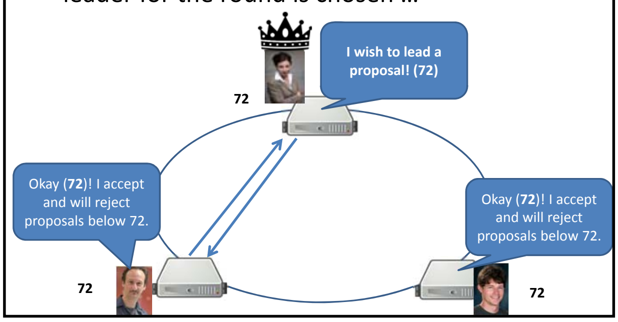
PAXOS Phase 1b: Promise

- By saying "okay", a cohort agrees to reject lower numbers



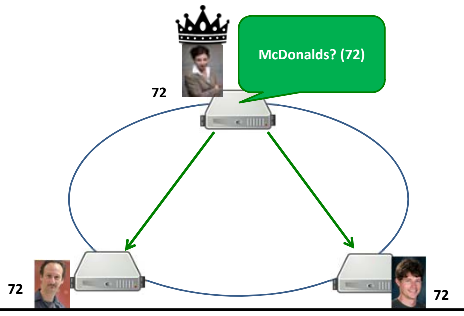
PAXOS Phase 1a/b: Prepare/Promise

- This continues until a majority agree and a leader for the round is chosen …



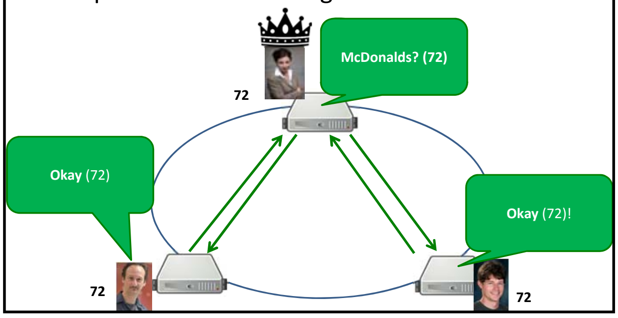
PAXOS Phase 2a: Accept Request

- The leader must now propose the value to be voted on this round …



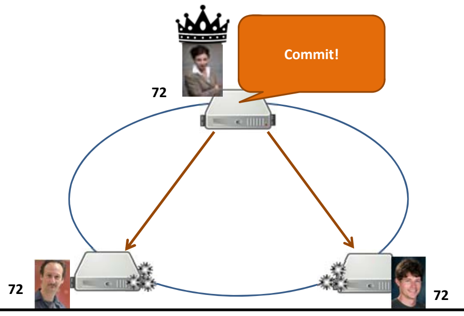
PAXOS Phase 2b: Accepted

- Nodes will accept if they haven't seen a higher request and acknowledge …

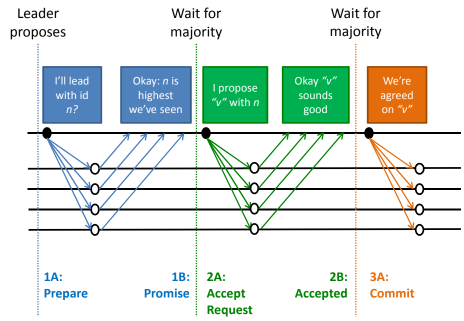


PAXOS Phase 3: Commit

- If a majority pass the proposal, the leader tells the cohort members to commit …



PAXOS Summary



PAXOS: No Agreement?

- If a majority cannot be reached, a new proposal is made with a higher number (by another member)

PAXOS: Failure Handling

- Leader is fluid: based on highest ID the members have stored
  – If Leader were fixed, PAXOS would be like 2PC
- Leader fails?
  – Another leader proposes with higher ID
- **Leader fails and recovers** (asynchronous)?
  – Old leader superseded by new higher ID
- Partition?
  – Requires majority / when partition is lifted, members must agree on higher ID

PAXOS: Guarantees

- Validity/Integrity:
  - Value proposed by a leader
- Agreement/Consistency:
  - A value needs a majority to pass
  - Each member can only choose one value
  - Other proposals would have to try convince a majority node!
  - Therefore only one agreed value can be chosen!

PAXOS In-Use

Google

Chubby: "Paxos Made Simple"

LAB II REVIEW: EXTERNAL SORTING

Lab II Review

What are (1) the strengths and (2) weaknesses of doing the word count (or other large-scale processing tasks) using external sorts compared with using main memory?

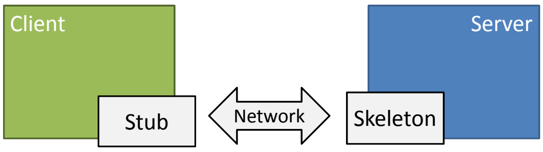
LAB III PREVIEW: JAVA RMI OVERVIEW

Why is Java RMI Important?

We can use it to quickly build distributed systems using some standard Java skills.

What is Java RMI?

- RMI = Remote Method Invocation
- Remote Procedure Call (RPC) for Java
- Ancestor of CORBA (in Java)
- Stub / Skeleton model (TCP/IP)



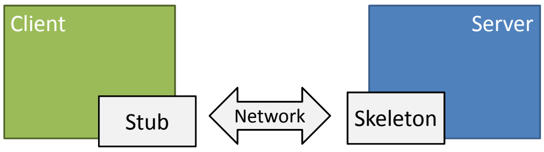
What is Java RMI?

Stub (Client):

- Sends request to skeleton: marshalls/serialises and transfers arguments
- Demarshalls/deserialises response and ends call

Skeleton (Server):

- Passes call from stub onto the server implementation
- Passes the response back to the stub



Stub/Skeleton Same Interface!

```java
package org.mdp.dir;

import java.io.Serializable;

/**
 * This is the interface that will be registered in the server.
 * In RMI, a remote interface is called a stub (on the client-side)
 * or a skeleton (on the server-side).
 *
 * An implementation is created and registered on the server.
 *
 * Remote machines can then call the methods of the interface.
 *
 * Note: every method *must* throw RemoteException!
 *
 * Note: every object passed or returned *must* be Serializable!
 *
 * @author Aidan
 *
 */
public interface UserDirectoryStub extends Remote, Serializable{
    public boolean createUser(User u) throws RemoteException;

    public Map<String,User> getDirectory() throws RemoteException;

    public User removeUserWithName(String un) throws RemoteException;
}
```

Client

Server

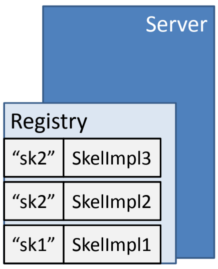
Server Implements Skeleton

```java
package org.mdp.dir;

import java.util.HashMap;

 * This is the implementation of UserDirectoryStub.
public class UserDirectoryServer implements UserDirectoryStub {

    private static final long serialVersionUID = -6025896167995177840L;
    private Map<String,User> directory;

    public UserDirectoryServer(){
        directory = new HashMap<String,User>();
    }

     * Return true if successful, false otherwise.
    public boolean createUser(User u) {
        if(u.getUsername()==null)
            return false;

        directory.put(u.getUsername(), u);

        System.out.println("New user registered! Bienvendio a ...\n\t"+u);
        return true;
    }

     * Returns the current directory of users.
    public Map<String, User> getDirectory() {
        return directory;
    }

     * Just an option to clean up if necessary!
    public User removeUserWithName(String un) {
        System.out.println("Removing username '"+un+"'. Chao!");
        return directory.remove(un);
    }
}
```

Server

Server Registry

- Server (typically) has a Registry: a Map
- Adds skeleton *implementations* with key (a string)



Server Creates/Connects to Registry

```java
// create registry
Registry registry = LocateRegistry.createRegistry(port);
```

OR

```java
// connect to registry
Registry registry = LocateRegistry.getRegistry(hostname, port);
```

Server

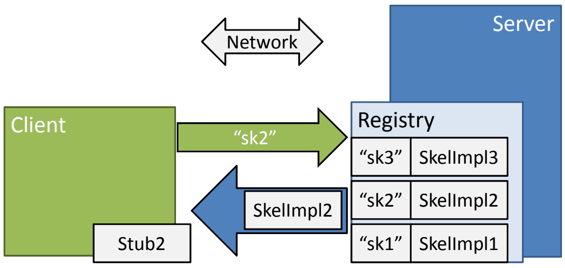
Server Registers Skeleton Implementation As a Stub

```
// create a remote stub to make it
// ready for incoming calls
Remote stub = UnicastRemoteObject.exportObject(new UserDirectoryServer(),0);

// register stub in registry under a key stub-name
String stubname = "mensaje";
registry.bind(stubname, stub);
```

Server

Client Connecting to Registry

- Client connects to registry (port, hostname/IP)!
- Retrieves skeleton/stub with key



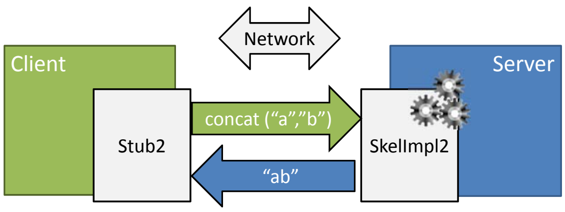
Client Connecting to Registry

```
String hostname = "server.com";
int port = 1985;
String stubname = "mensaje";

// first need to connect to the remote registry on the given
// IP and port
Registry registry = LocateRegistry.getRegistry(hostname, port);

// then need to find the interface we're looking for
UserDirectoryStub stub = (UserDirectoryStub) registry.lookup(stubname);
```

Client

Client Calls Remote Methods

- Client has stub, calls method, serialises arguments
- Server does processing
- Server returns answer; client deserialises result



Client Calls Remote Methods

```
// now we can use the stub to call remote methods!!
Map<String,User> users = stub.getDirectory();
System.err.println(users.toString());

User u = new User("aidhog", "Aidan Hogan", "10.0.114.59", 1509);
stub.createUser(u);

users = stub.getDirectory();
System.err.println(users.toString());

stub.removeUserWithName("aidhog");

users = stub.getDirectory();
System.err.println(users.toString());
```

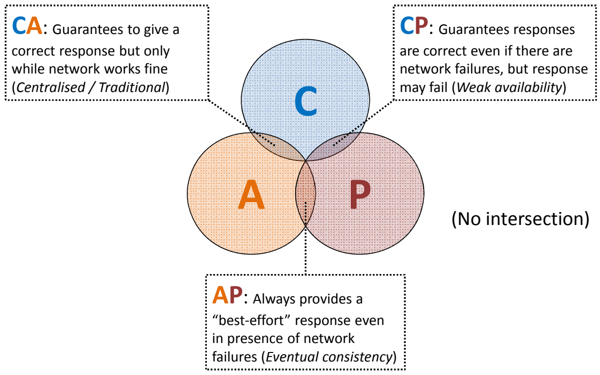
Client

Java RMI: Remember …

1. Remote calls are pass-by-value, not pass-by-reference (objects not modified directly)
2. Everything passed and returned must be Serialisable (`implement Serializable`)
3. Every stub/skel method *must* throw a remote exception (`throws RemoteException`)
4. Server implementation can only throw `RemoteException`

RECAP

CAP Systems



Consensus for CP-systems

- Synchronous vs. Asynchronous
  - Synchronous less difficult than asynchronous
- Fail–stop vs. Byzantine
  - Byzantine typically software (arbitrary response)
  - Fail–stop gives no response

Consensus for CP-systems

- Two-Phase Commit (2PC)
  - Voting
  - Commit
- Three-Phase Commit (3PC)
  - Voting
  - Prepare
  - Commit

Consensus for CP-systems

- PAXOS:
  - 1a. Prepare
  - 1b. Promise
  - 2a. Accept Request
  - 2b. Accepted
  - 3. Commit

Java: Remote Method Invocation

- Java RMI:
  - Remote Method Invocation
  - Stub on Client Side
  - Skeleton on Server Side
  - Registry maps names to skeletons/servers
  - Server registers skeleton with key
  - Client finds skeleton with key, casts to stub
  - Client calls method on stub
  - Server runs method and serialises result to client

Questions?