CC5212-1

Procesamiento Masivo de Datos

Otoño 2023

Lecture 2

Distributed Systems

Aidan Hogan
aidhog@gmail.com

Processing massive data needs Distributed Systems ...

Monolithic vs. Distributed Systems

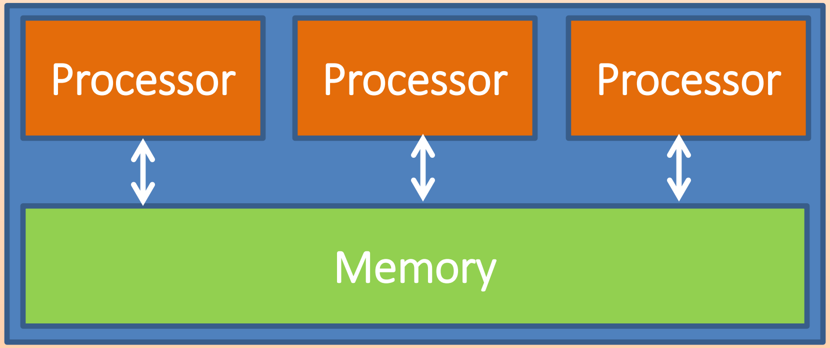
- One machine that's *n* times as powerful?

- *n* machines that are equally as powerful?

Parallel vs. Distributed Systems

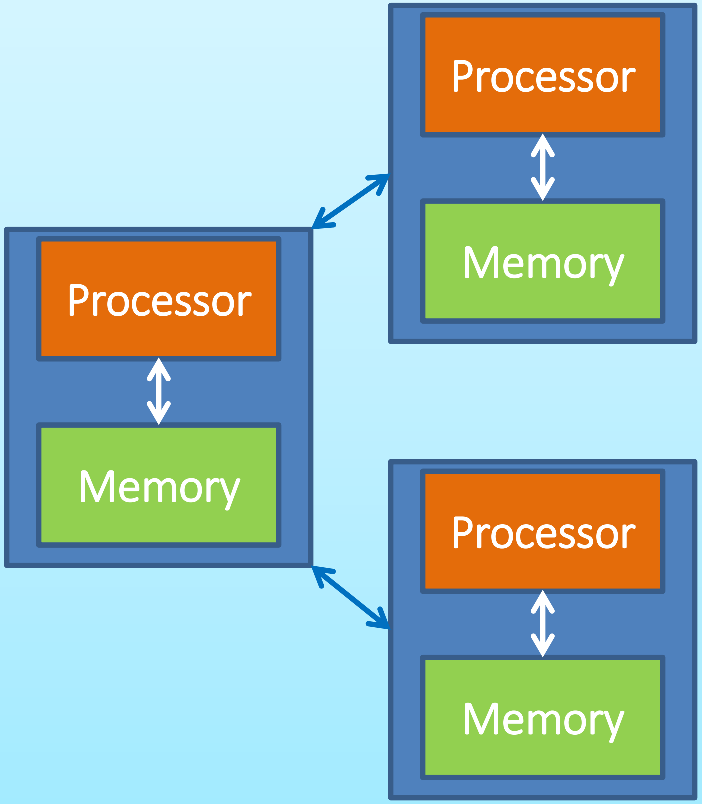
- Parallel System

  often *shared memory*



- Distributed System

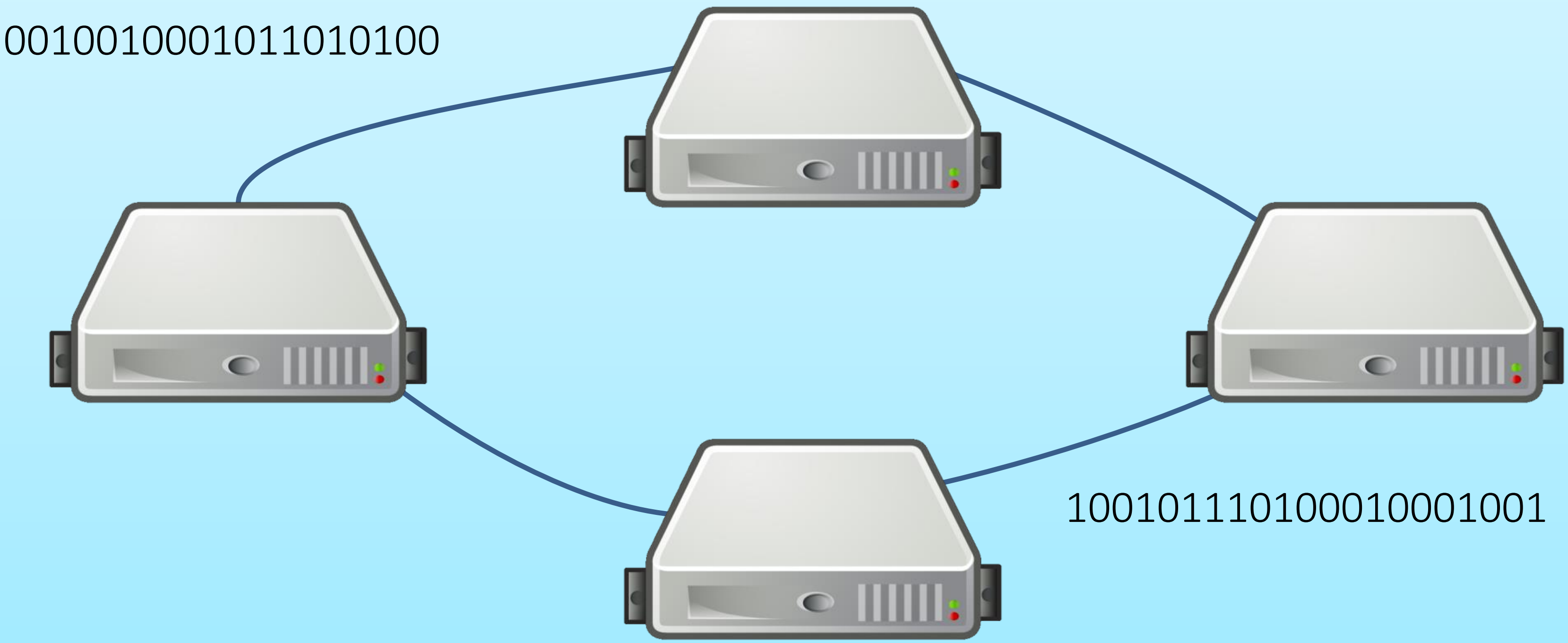
  often *shared nothing*



What is a Distributed System?

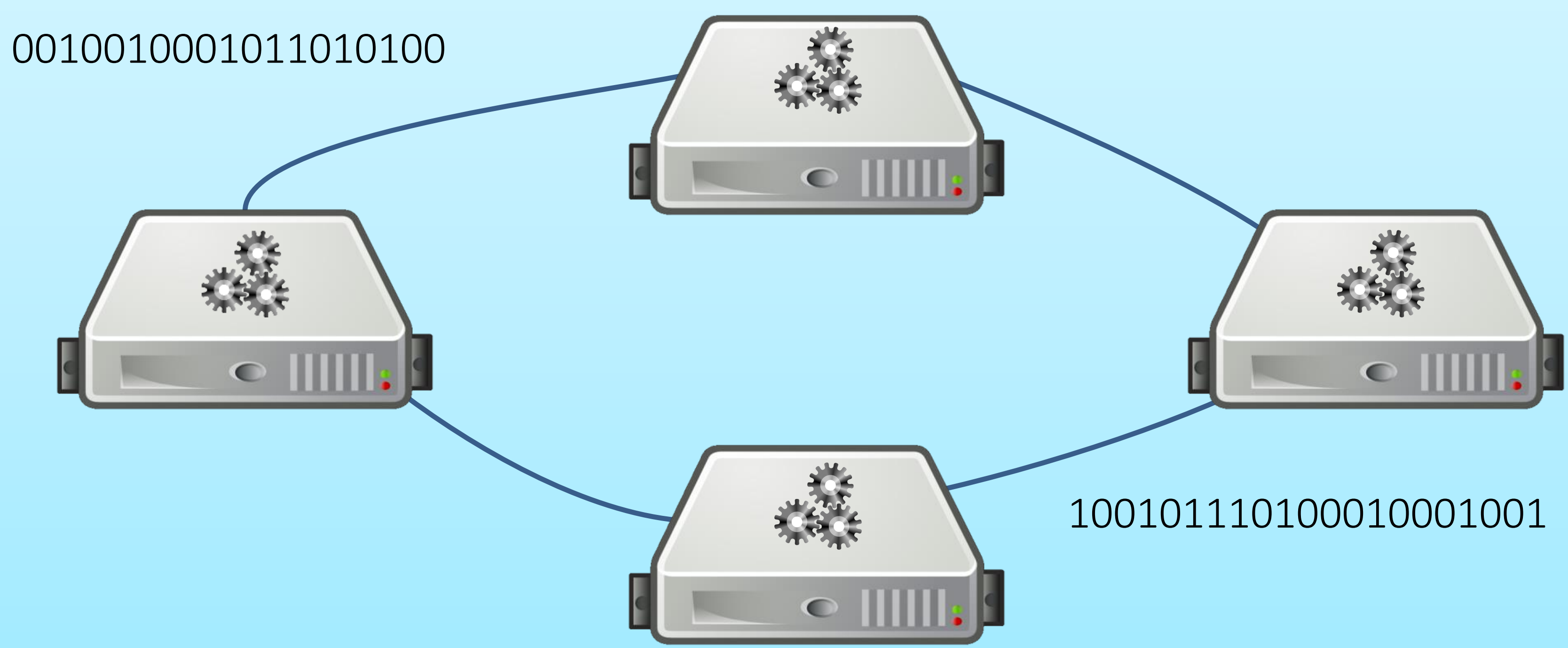
A distributed system is a system that enables a collection of **independent** computers to communicate in order to solve a common goal.

They have three important properties ...



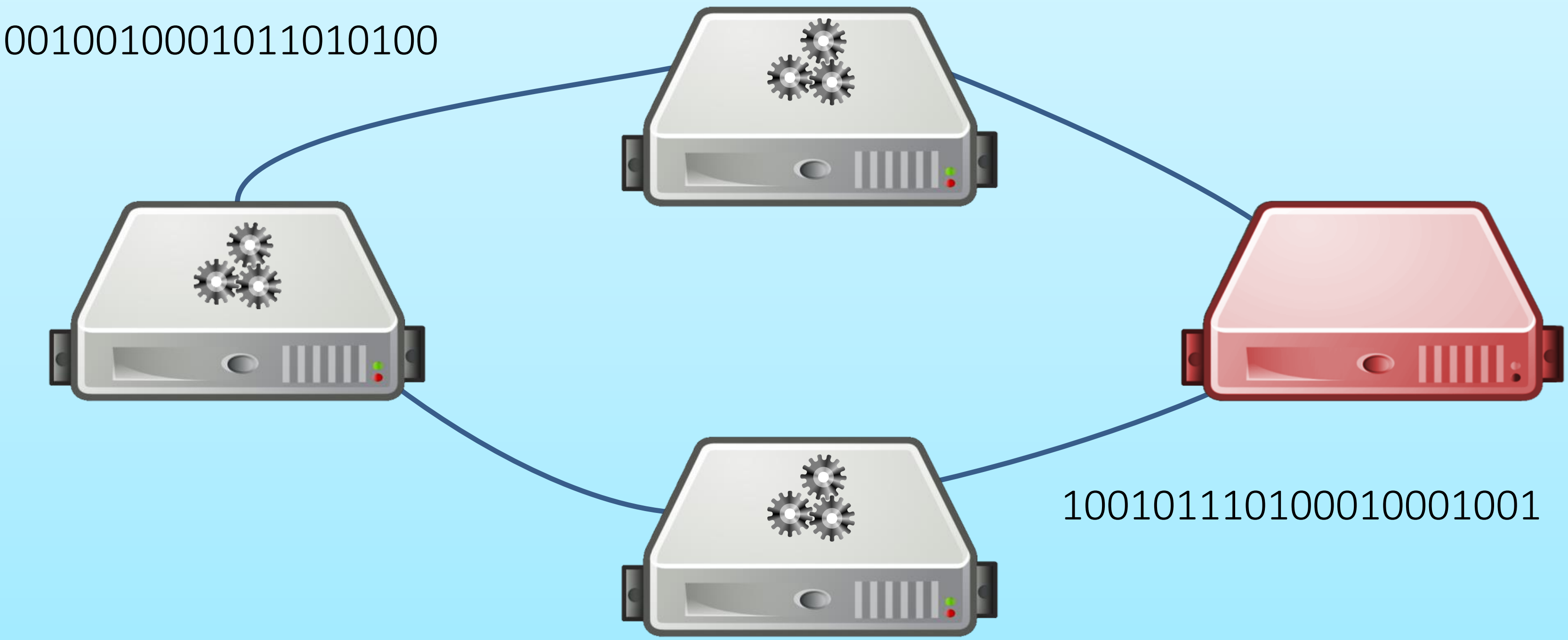
What is a Distributed System?
Three properties ...

1. Concurrency



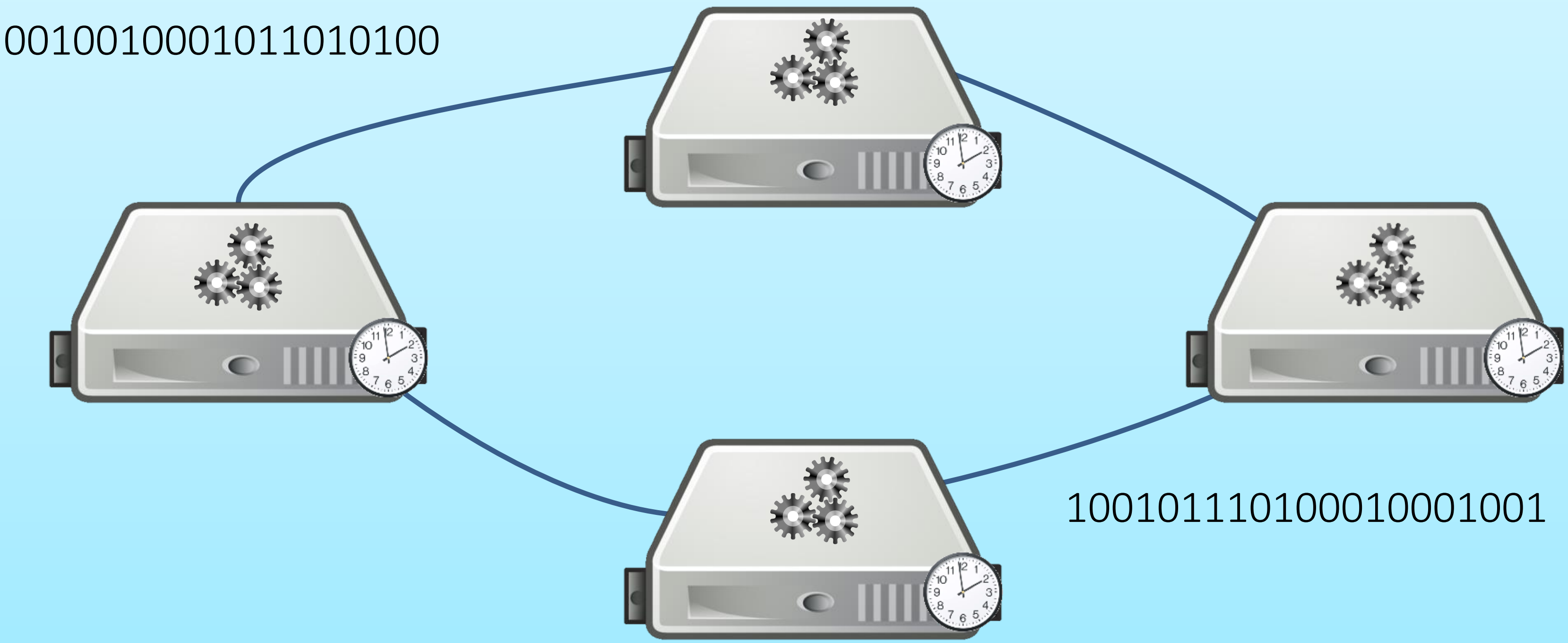
What is a Distributed System?
Three properties ...

1. Concurrency
2. Independent failures



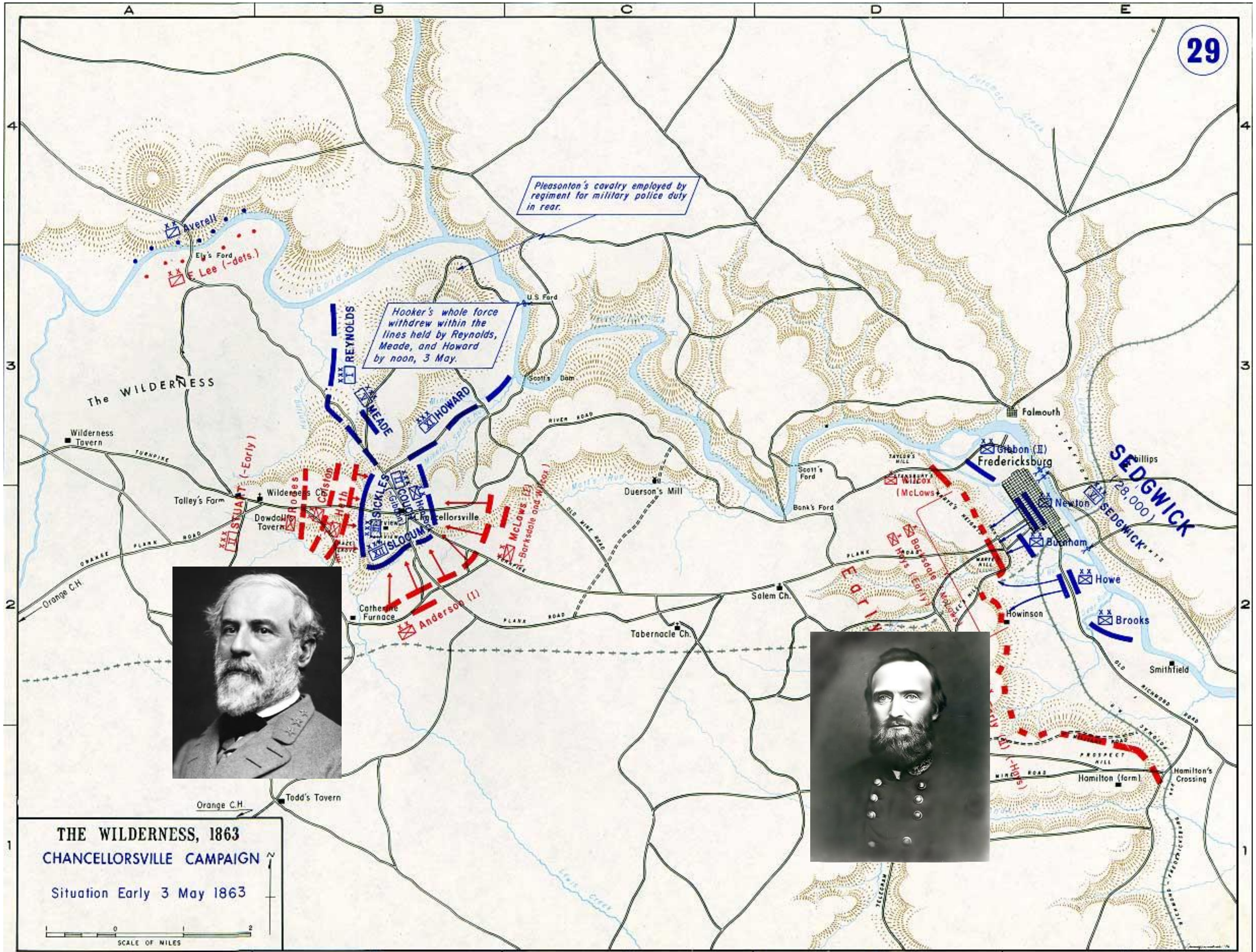
What is a Distributed System?
Three properties ...

1. Concurrency
2. Independent failures
3. No global clock



Challenges of distributed systems

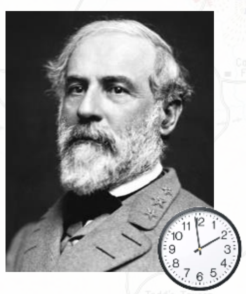
Two General's Problem

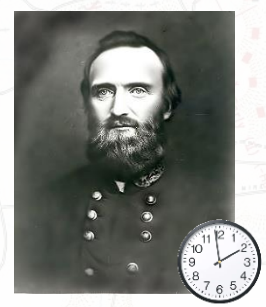


Two General's Problem

- Two generals need to agree a time to attack
  - They can send messengers on horse-back
  - Messengers can be killed en route

How can the generals coordinate a time for attack?

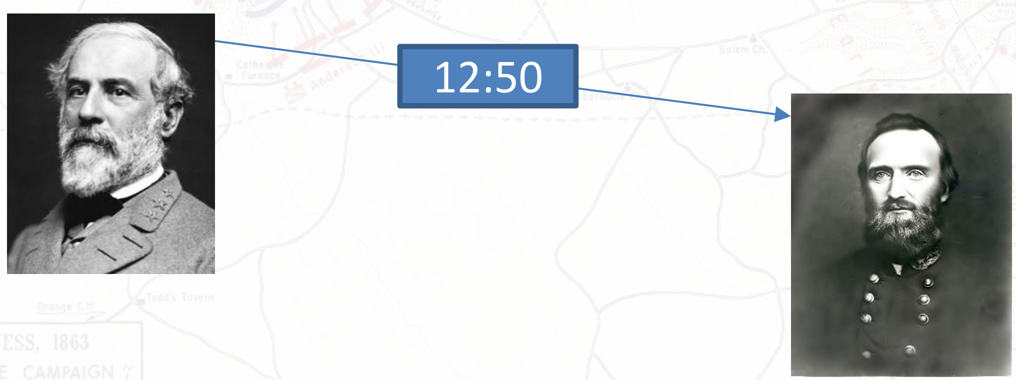




Two General's Problem

- Two generals need to agree a time to attack
  - They can send messengers on horse-back
  - Messengers can be killed en route

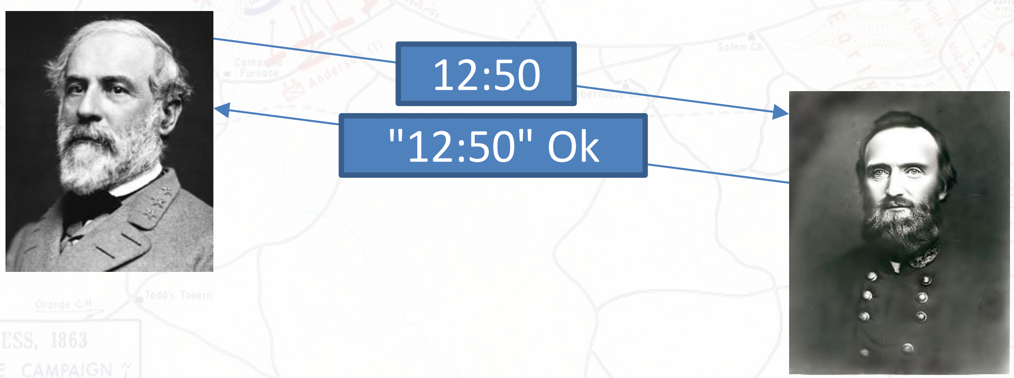
How can the generals coordinate a time for attack?



Two General's Problem

- Two generals need to agree a time to attack
  - They can send messengers on horse-back
  - Messengers can be killed en route

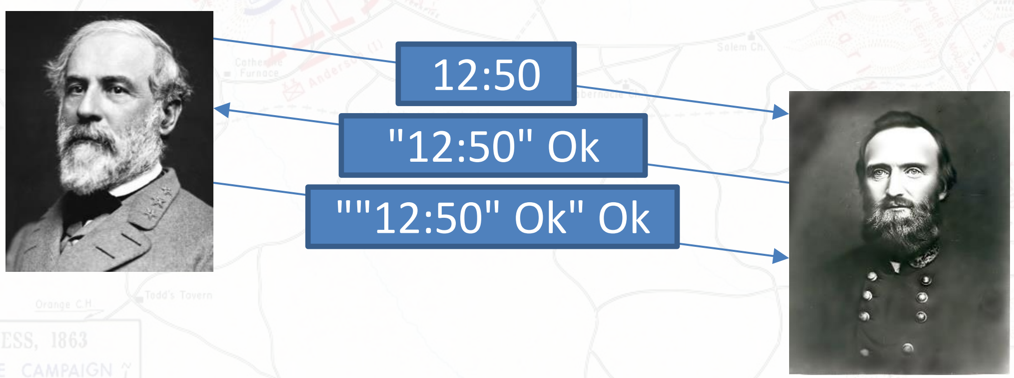
How can the generals coordinate a time for attack?



Two General's Problem

- Two generals need to agree a time to attack
  - They can send messengers on horse-back
  - Messengers can be killed en route

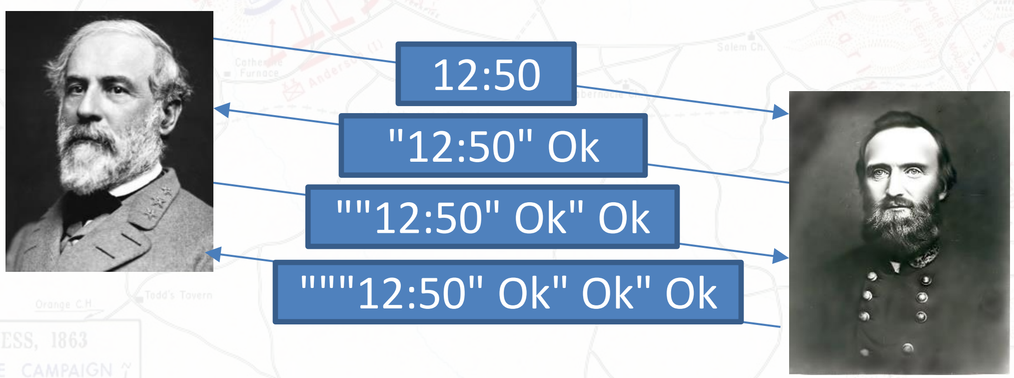
How can the generals coordinate a time for attack?



Two General's Problem

- Two generals need to agree a time to attack
  - They can send messengers on horse-back
  - Messengers can be killed en route

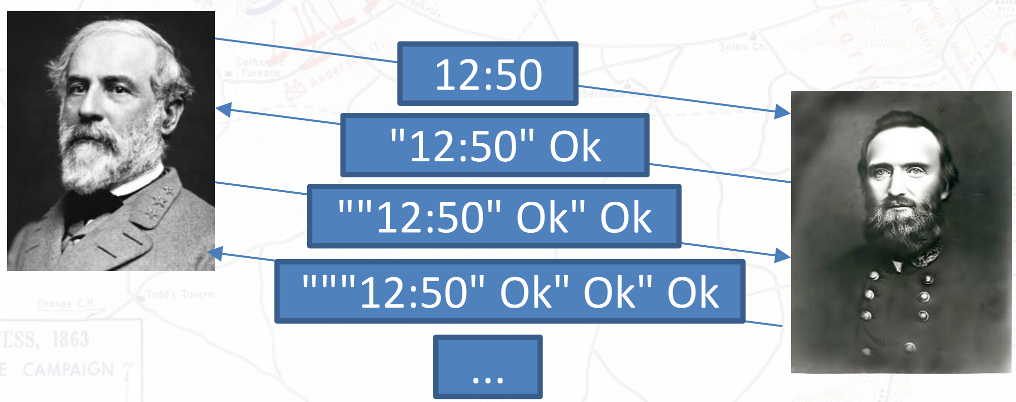
How can the generals coordinate a time for attack?



Two General's Problem

- Two generals need to agree a time to attack
  - They can send messengers on horse-back
  - Messengers can be killed en route

How can the generals coordinate a time for attack?

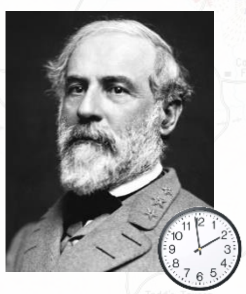


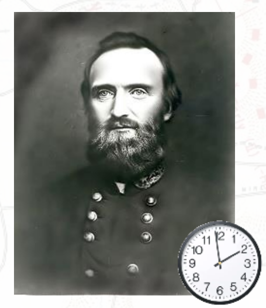
Two General's Problem

- Two generals need to agree a time to attack
  - They can send messengers on horse-back
  - Messengers can be killed en route

So how can we solve this problem?

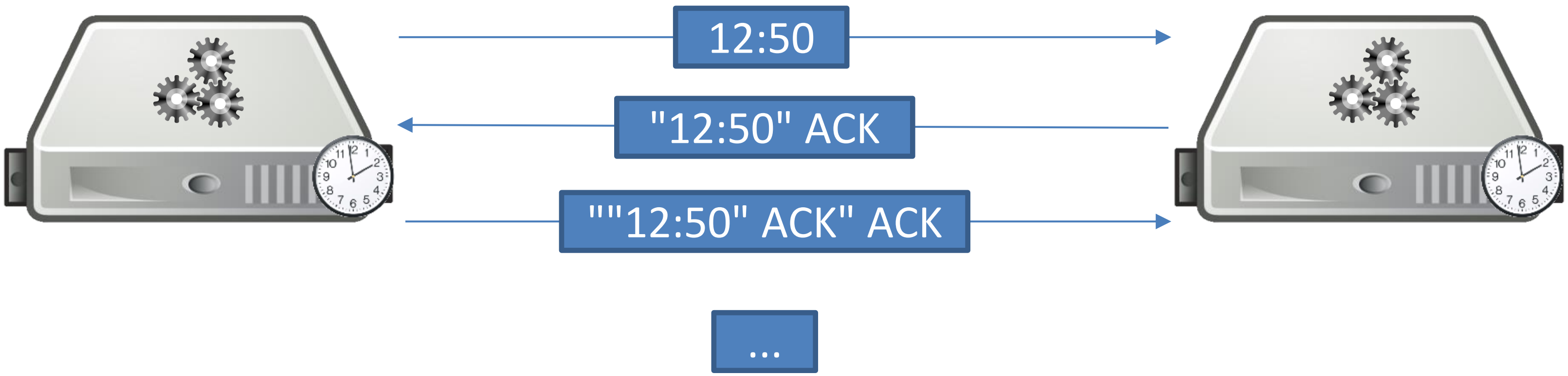
Umm, we cannot.





Two General's Problem

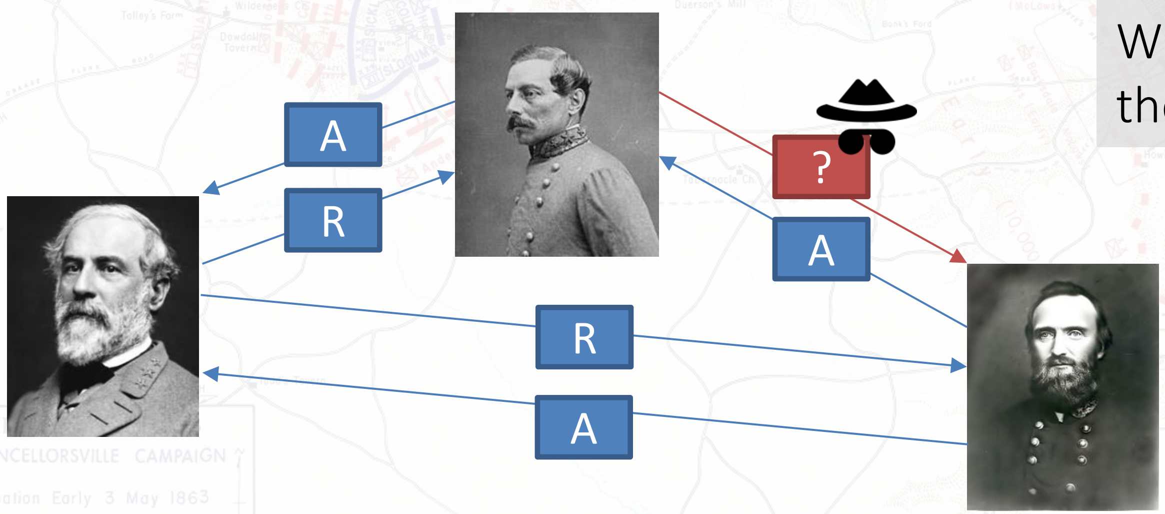
- An unsolvable problem in distributed systems



- Mitigation:
  - Improve communication reliability
  - Understand that communication is unreliable
  - Idempotency tokens (avoid repeated requests)
  - Pre-define a default time/action/value

Byzantine General's Problem

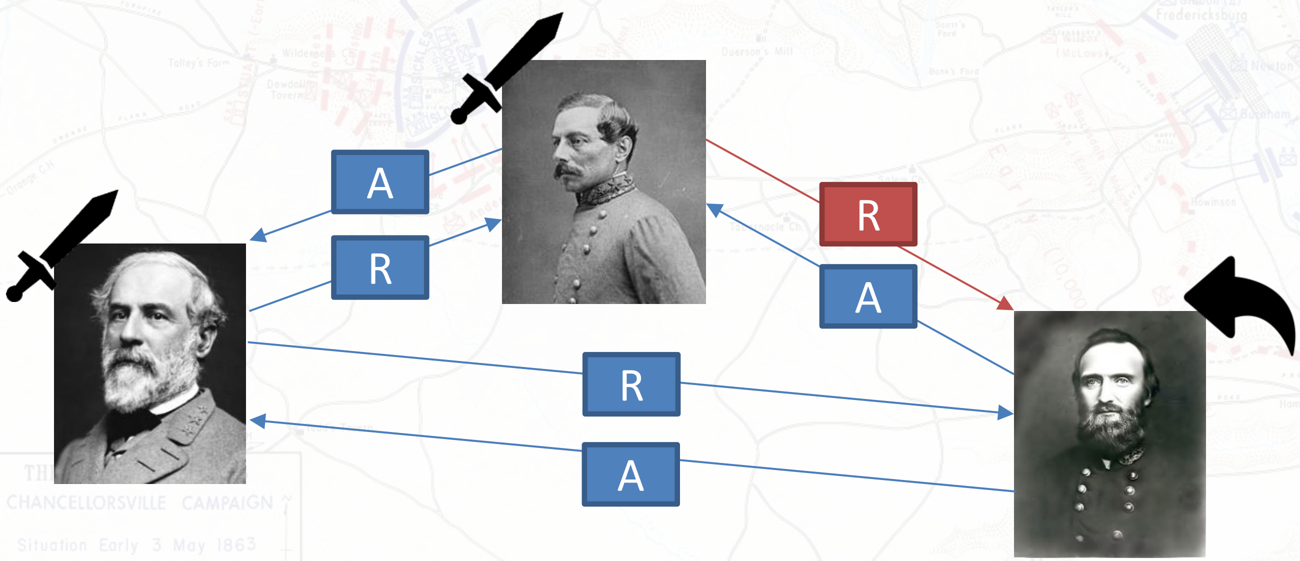
- Many generals need to agree on whether to attack or retreat
  - They must attack or retreat in unison
  - They can send messengers on horse-back
  - Messengers can be replaced with spies



What should the spy do?

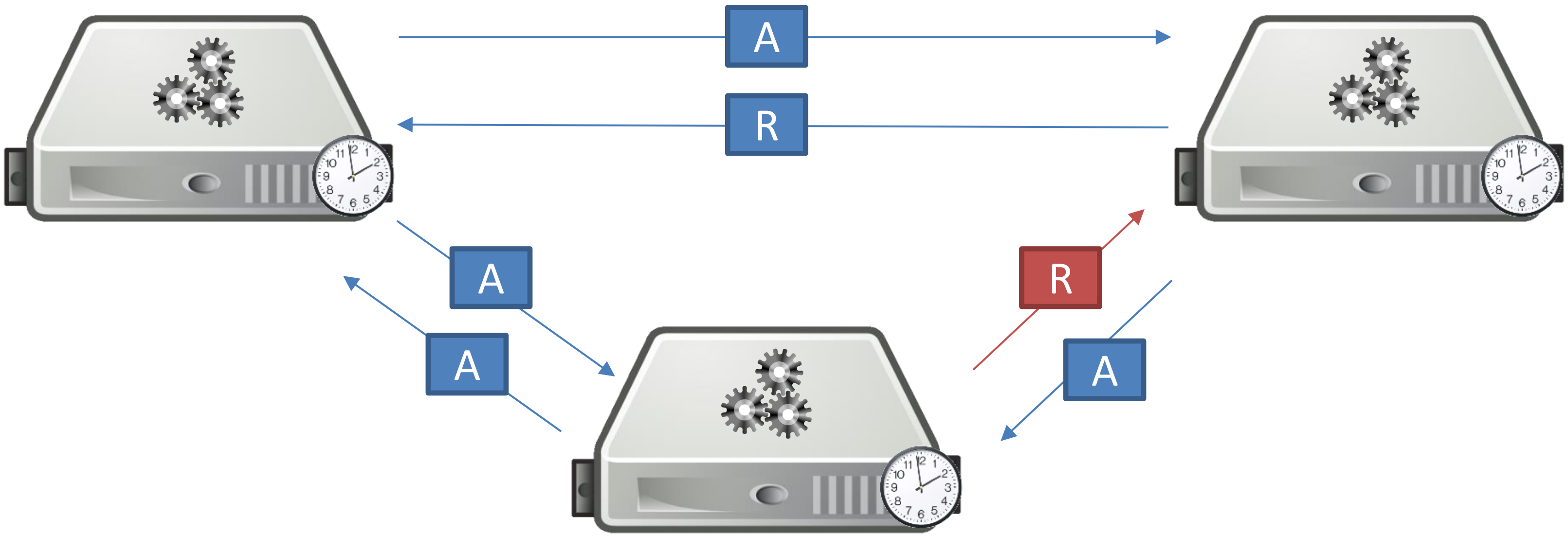
Byzantine General's Problem

- Many generals need to agree on whether to attack or retreat
  - They must attack or retreat in unison
  - They can send messengers on horse-back
  - Messengers can be replaced with spies



Byzantine General's Problem

- Again, a problem for distributed systems



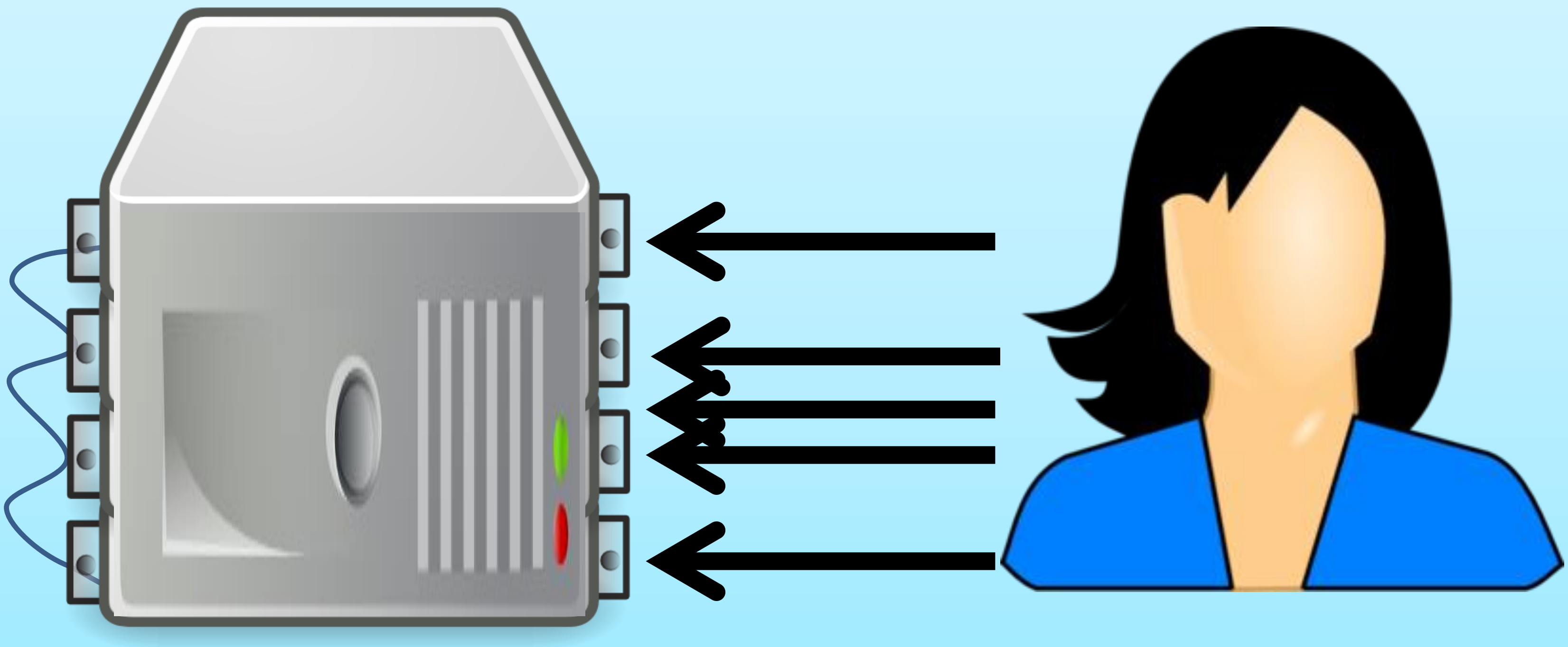
- Mitigation:
  - Consensus protocols (majority rules)
  - Multiple messages / null messages

What makes a good Distributed System?

A Good Distributed System …

Transparency

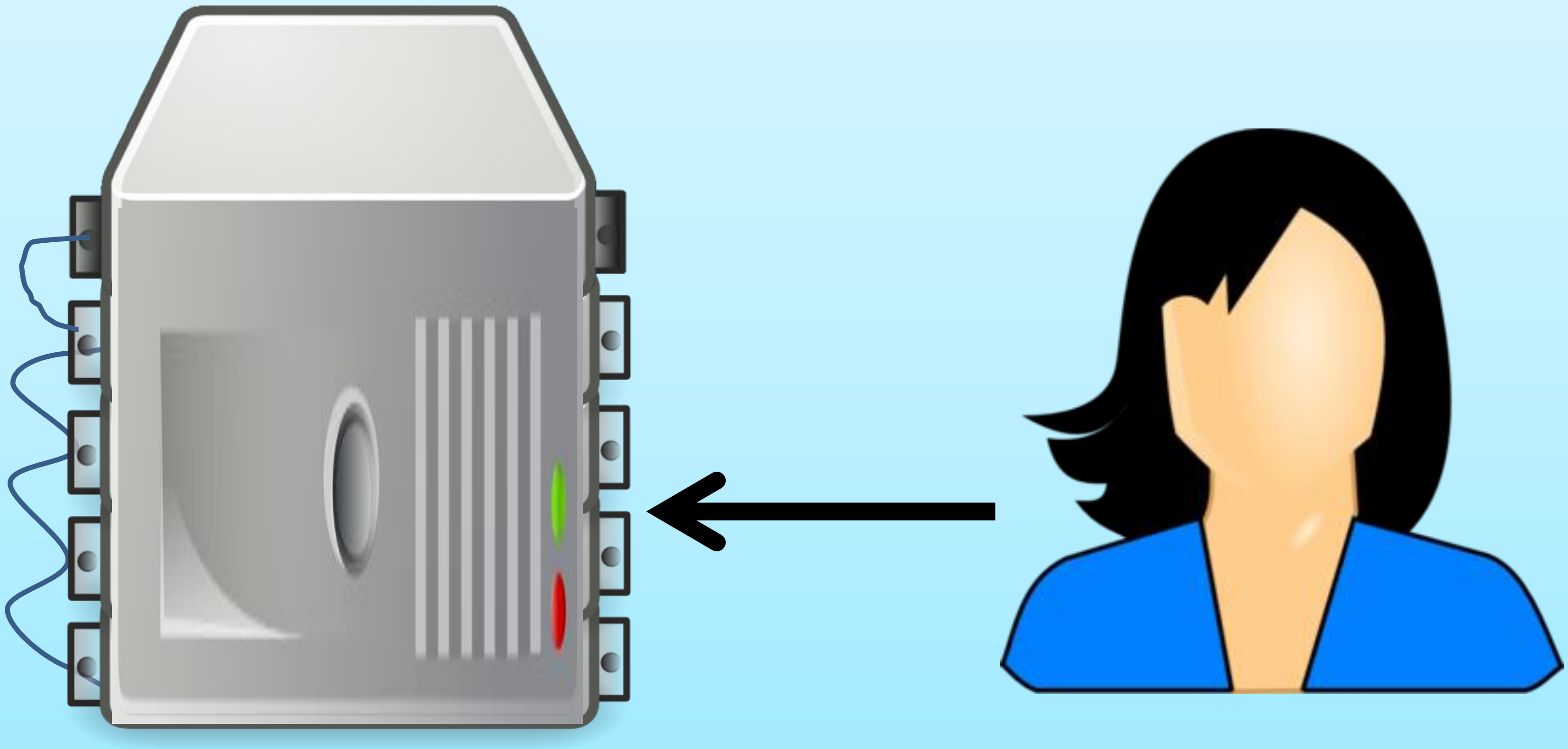
… looks like one system



A Good Distributed System …

Flexibility

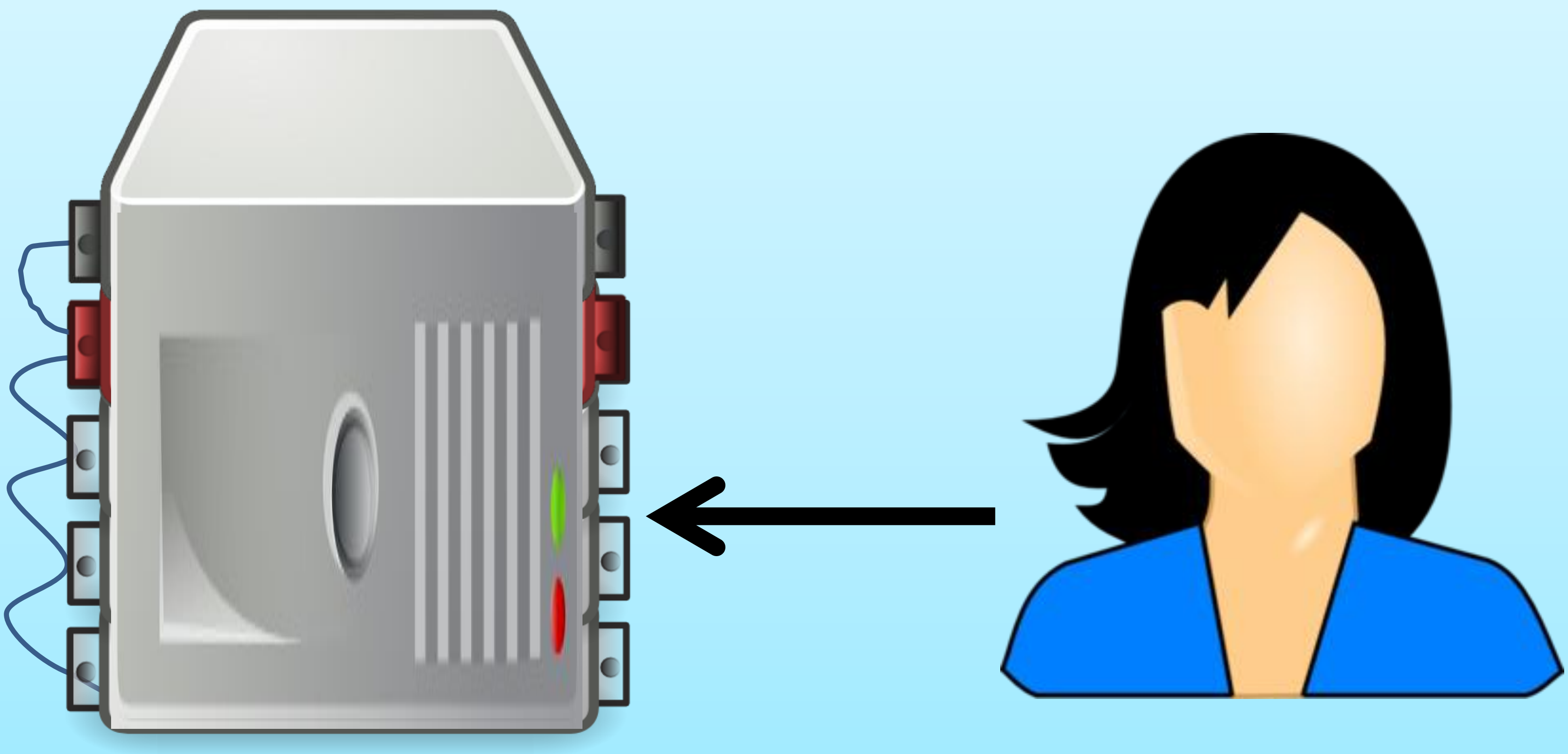
… can add/remove machines quickly and easily



A Good Distributed System …

Reliability

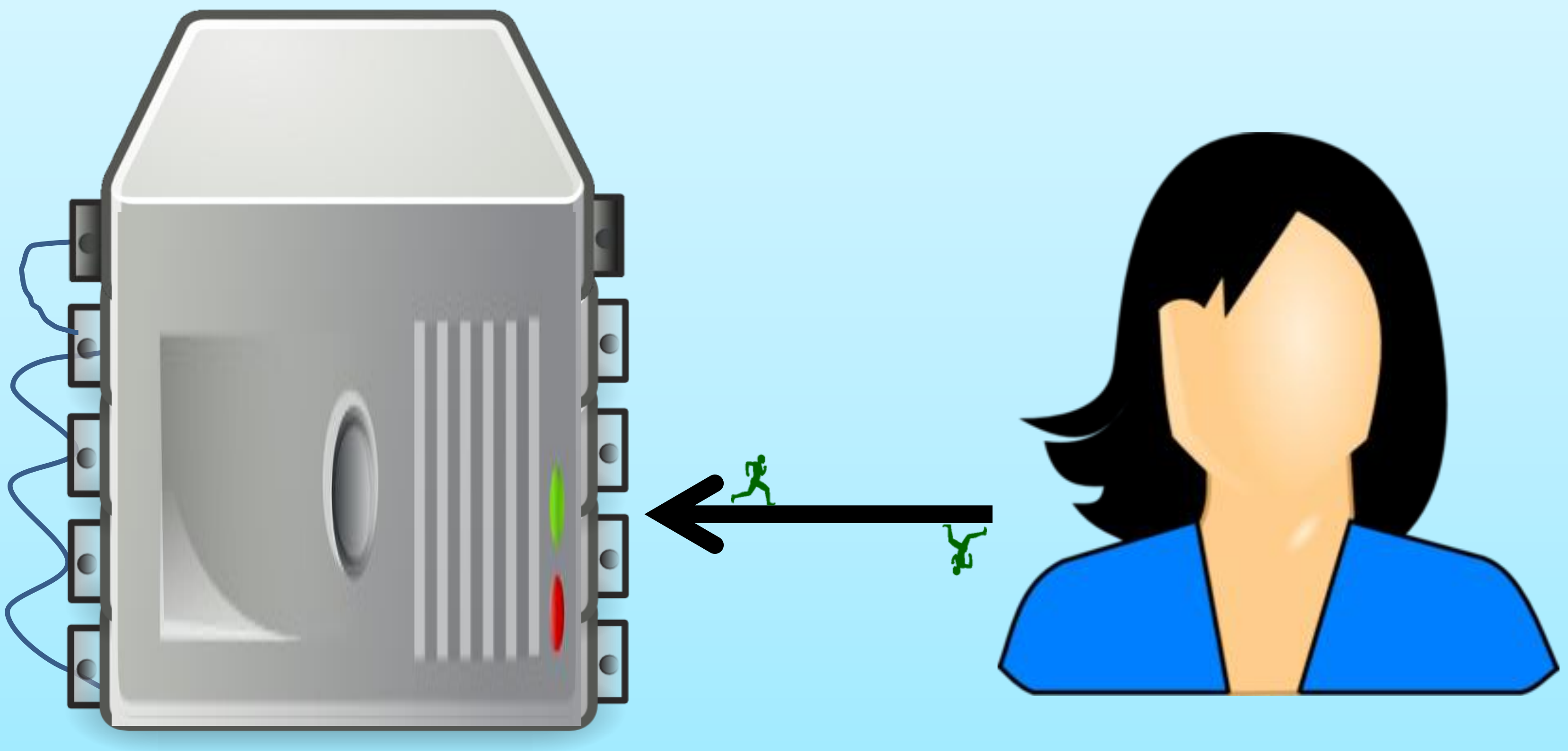
… avoids failure / keeps working in case of failure



A Good Distributed System …

Performance

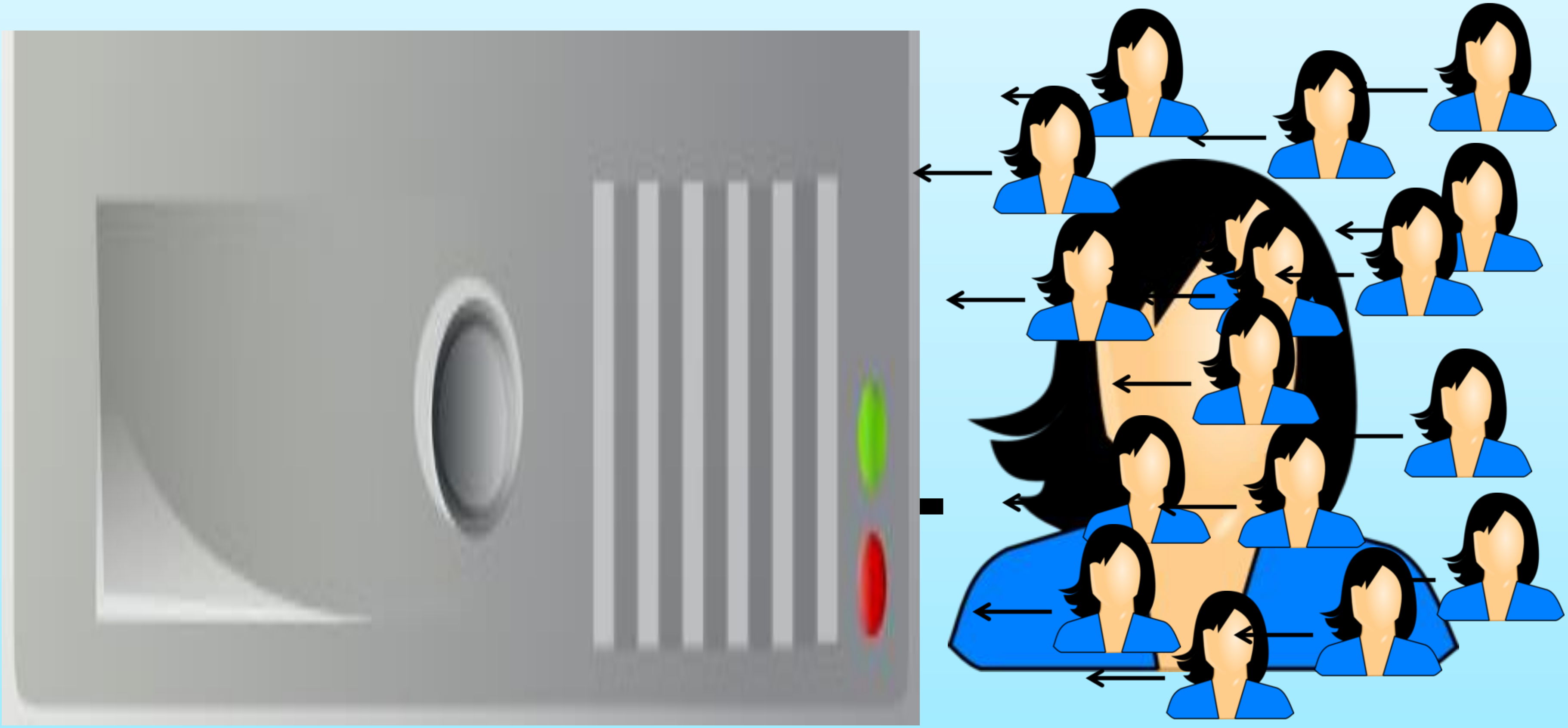
… does stuff quickly



A Good Distributed System …

Scalability

… ensures the infrastructure scales



A Good Distributed System …

Transparency

… looks like one system

Flexibility

… can add/remove machines quickly and easily

Reliability

… avoids failure / keeps working in case of failure

Performance

… does stuff quickly

Scalability

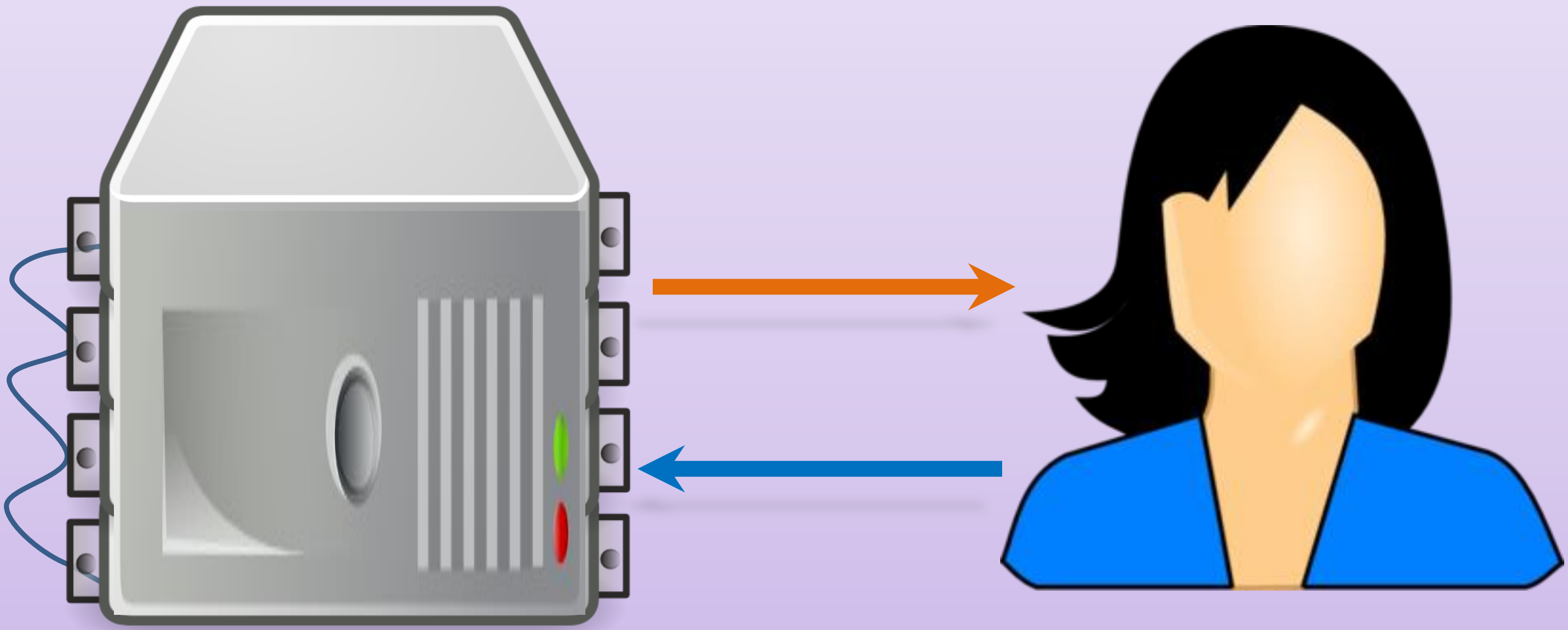
… ensures the infrastructure scales

Distributed Systems:
Client–Server Architecture

Client–Server Model

Client makes request to server

Server acts and responds



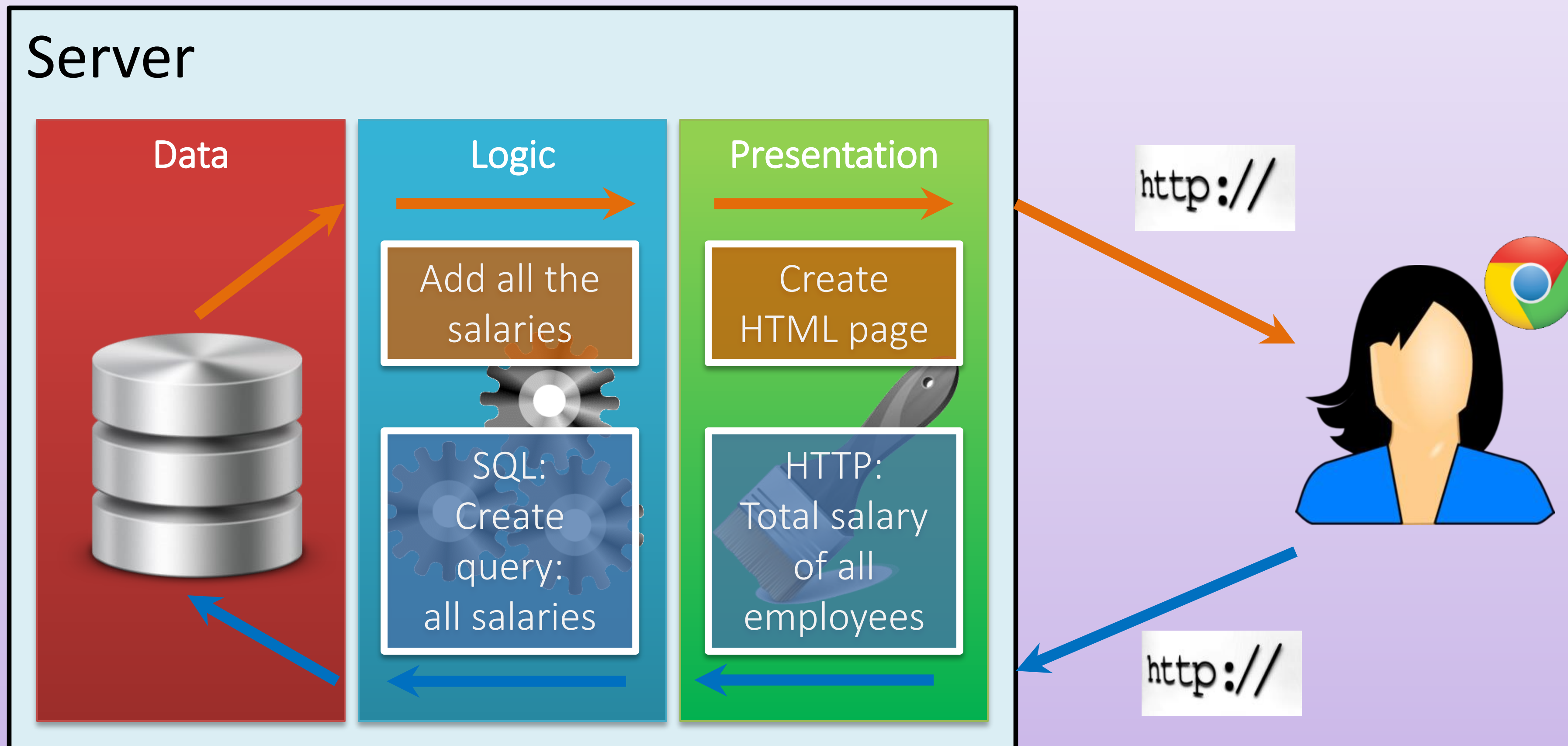
For example?

Web, Email, DropBox, ...

Client–Server: Three-Tier Server

Three Layer Architecture

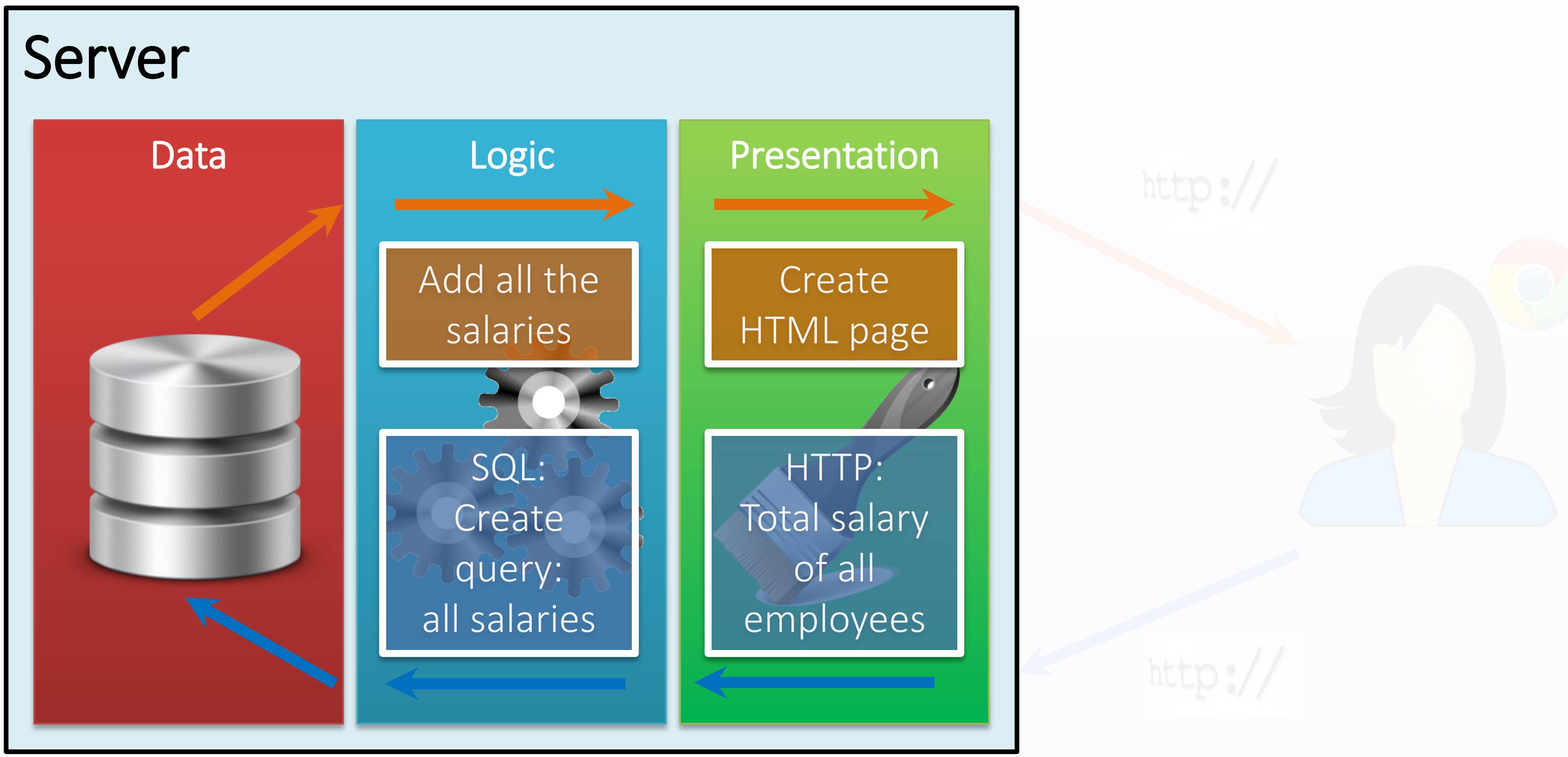
1. Data | 2. Logic | 3. Presentation



Client–Server: Three-Tier Server

<u>Server</u> can be a distributed system! ⚠

Server ≠ Physical Machine

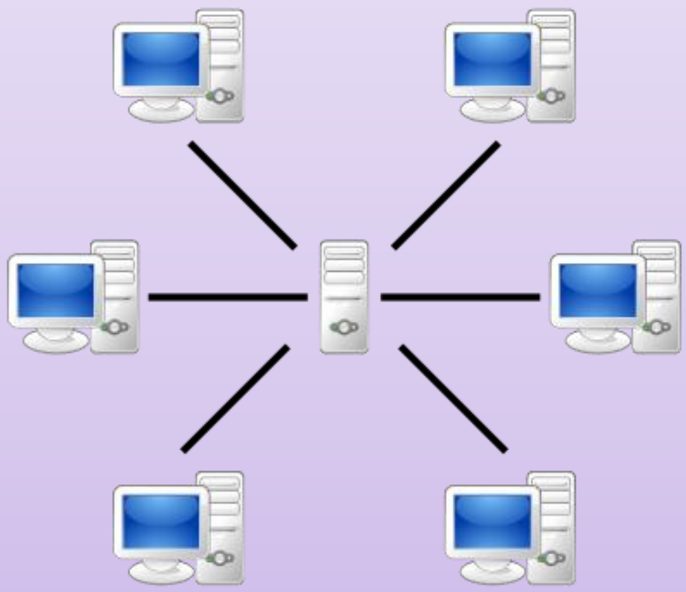


Distributed Systems:
Peer-to-Peer (P2P) Architecture

Peer-to-Peer (P2P)

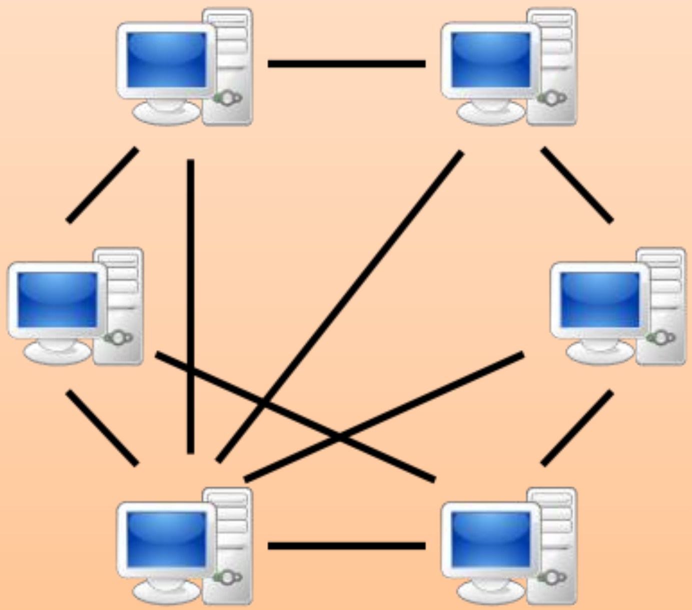
Client–Server

- Client interacts directly with server



Peer-to-Peer (P2P)

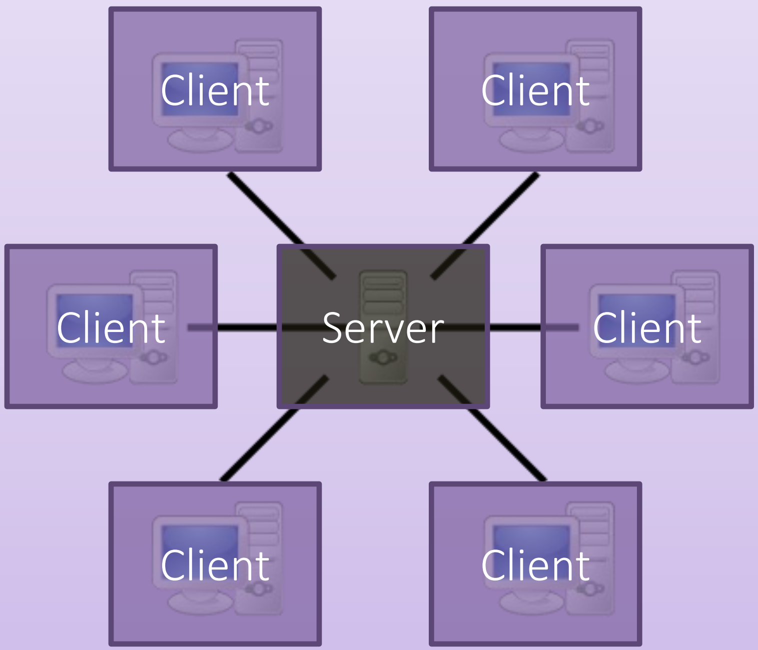
- Peers interact directly with each other



Peer-to-Peer (P2P)

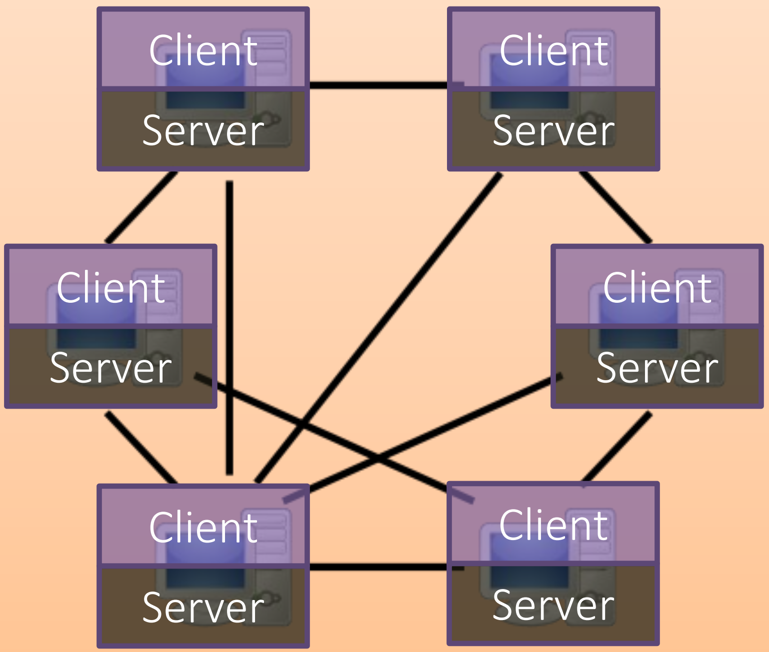
Client–Server

- Client interacts directly with server



Peer-to-Peer (P2P)

- Peers interact directly with each other



Peer-to-Peer (P2P)

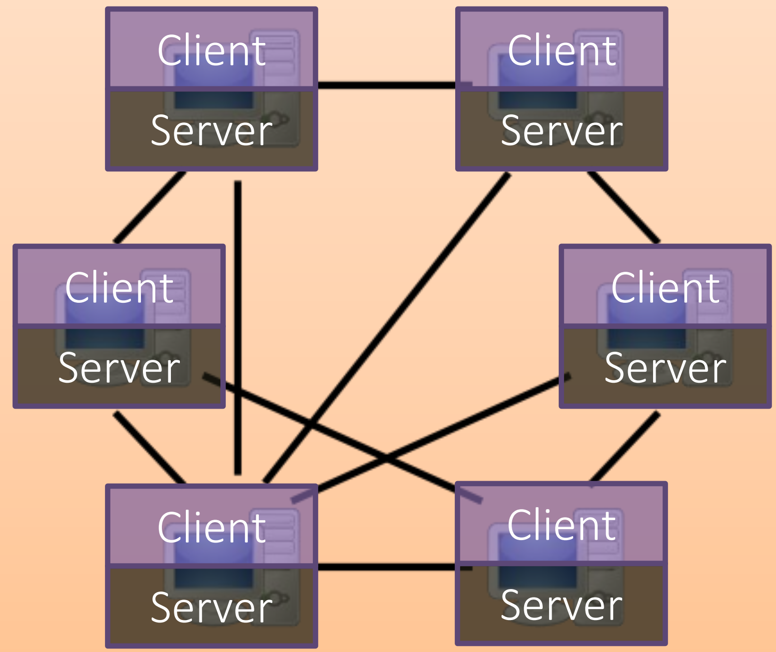
Client–Server

- Client interacts directly with server

Examples of P2P systems?

Peer-to-Peer (P2P)

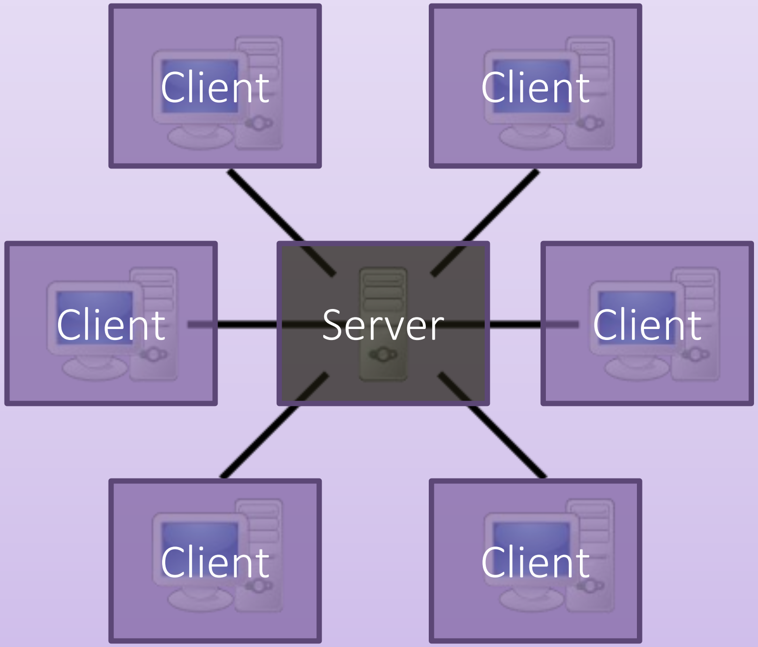
- Peers interact directly with each other



Peer-to-Peer (P2P)

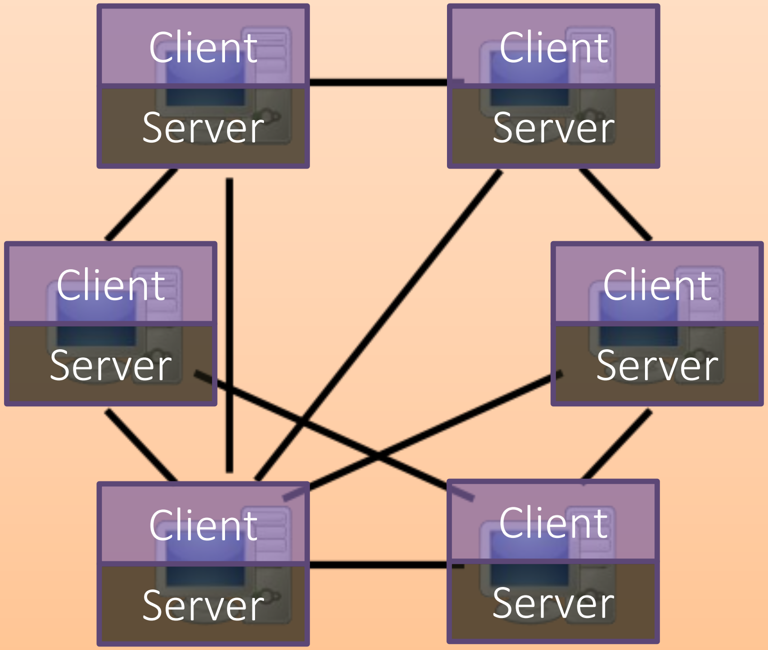
File Servers (DropBox):

- Clients interact with a central file server



P2P File Sharing (e.g., Bittorrent):

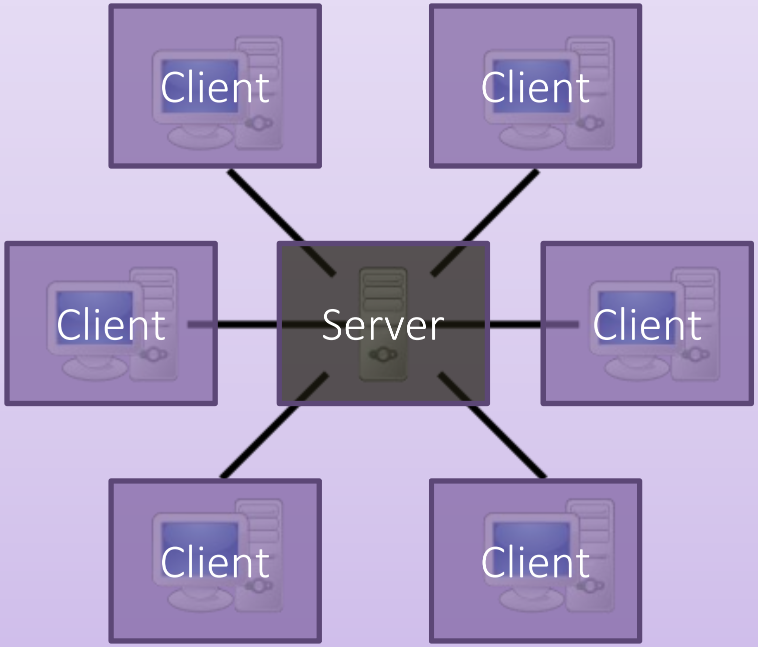
- Peers act both as the file server and the client



Peer-to-Peer (P2P)

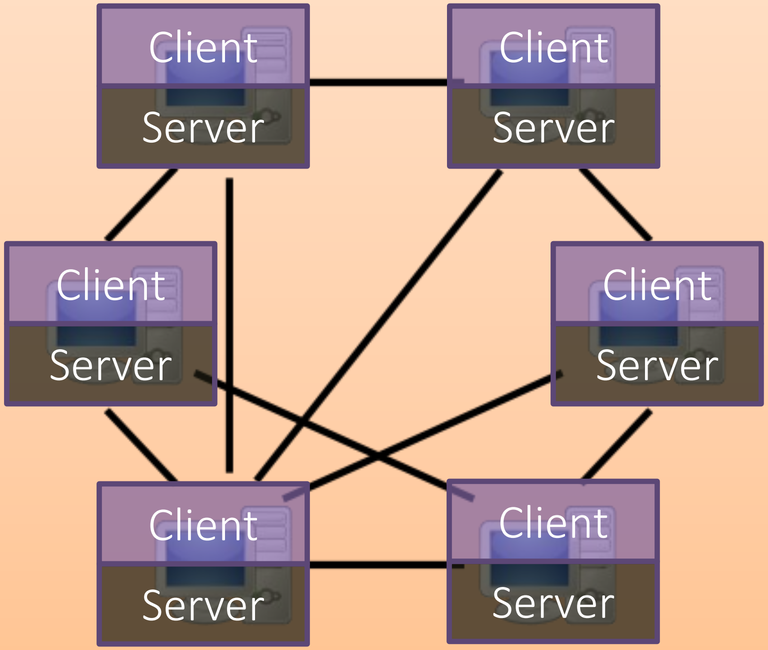
Online Banking:

- Clients interact with a central banking server



Cryptocurrencies (e.g., Bitcoin):

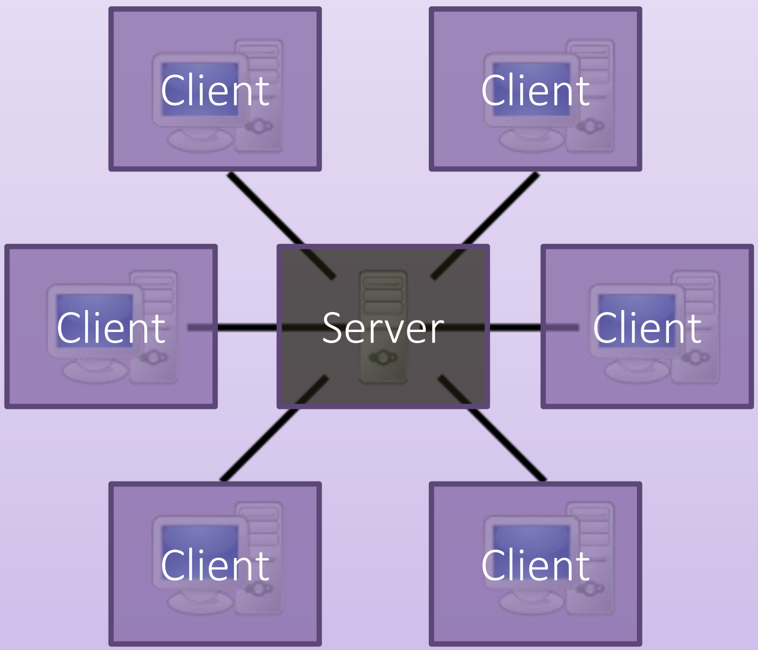
- Peers act both as the bank and the client



Peer-to-Peer (P2P)

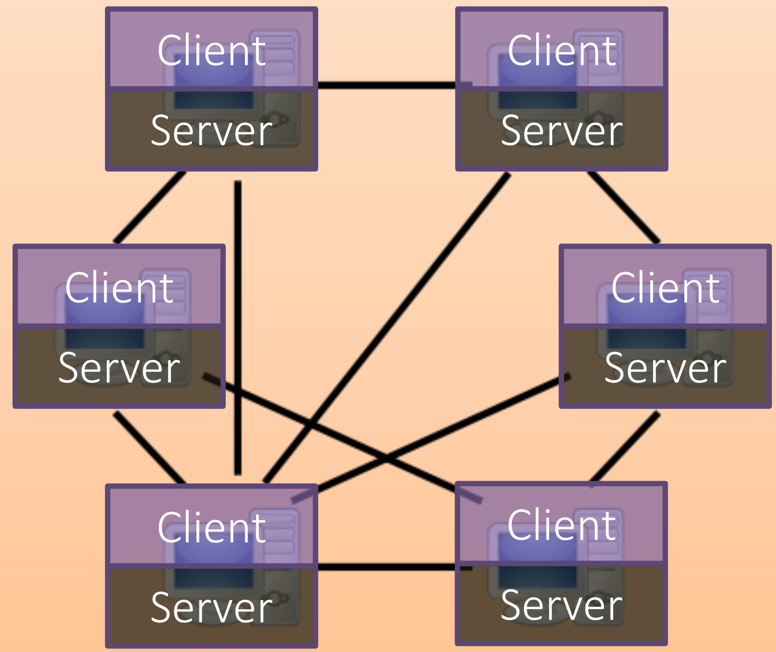
SVN:

- Clients interact with a central versioning repository

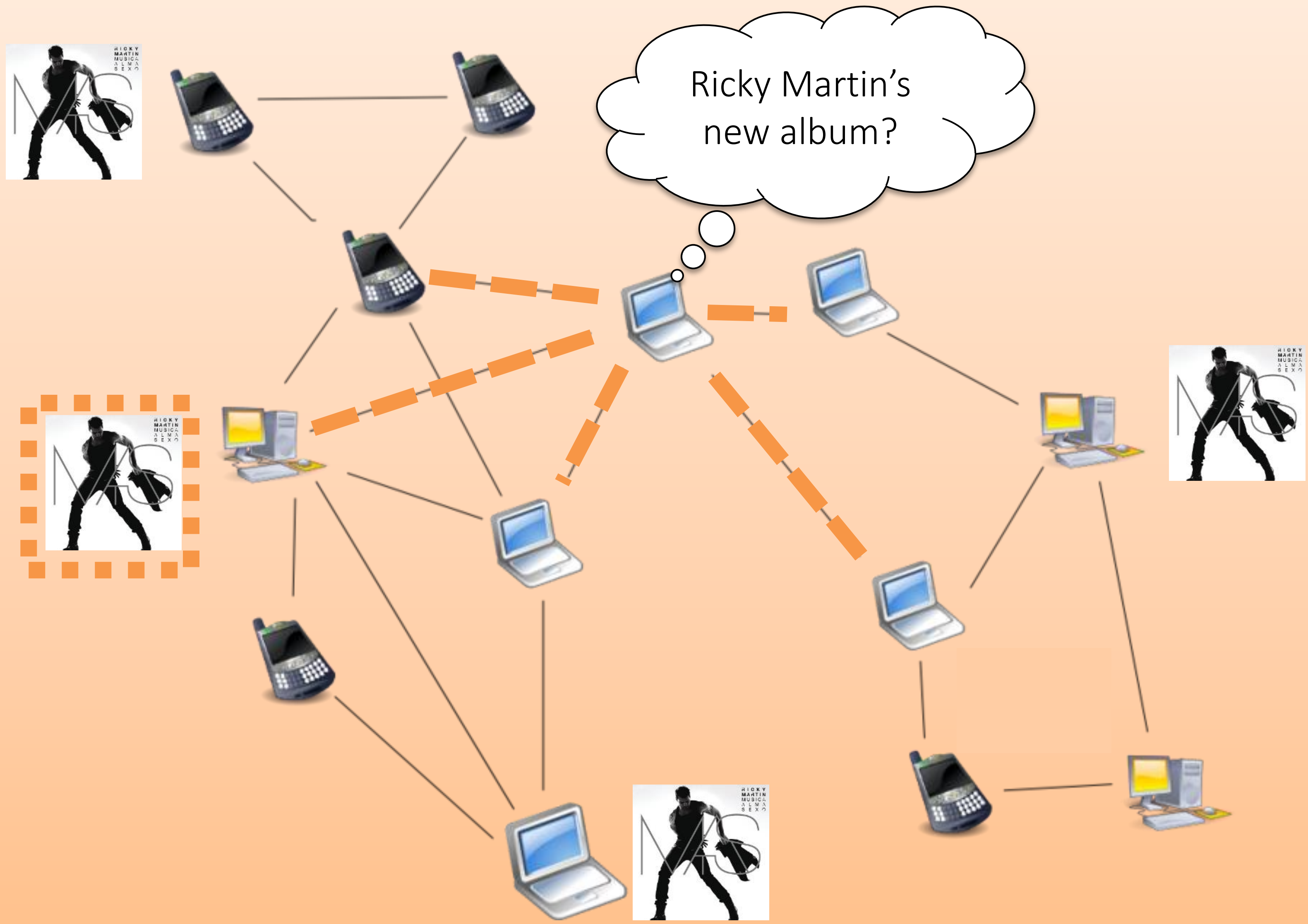


GIT:

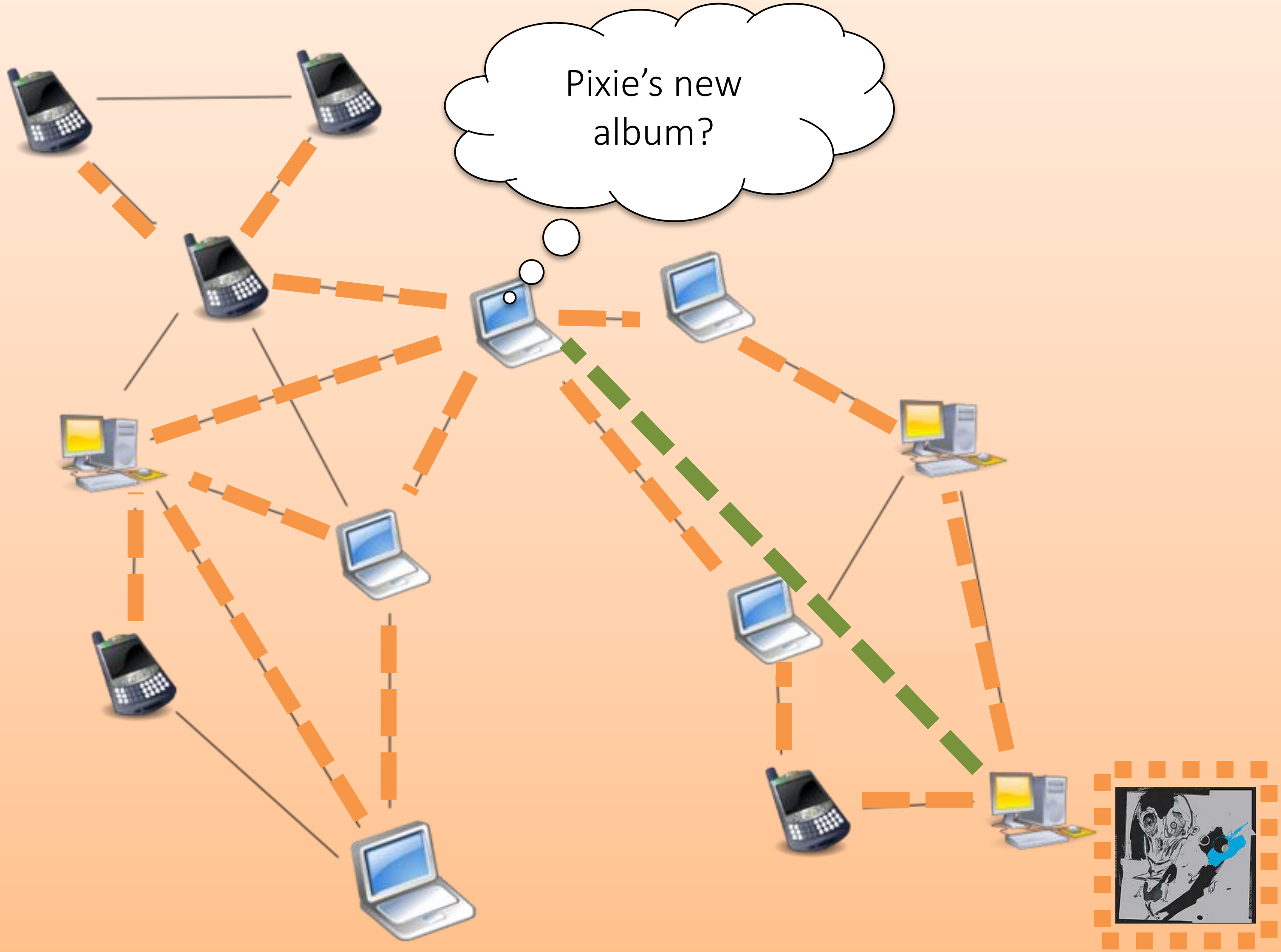
- Peers have their own repositories, which they sync.



Peer-to-Peer: Unstructured (flooding)

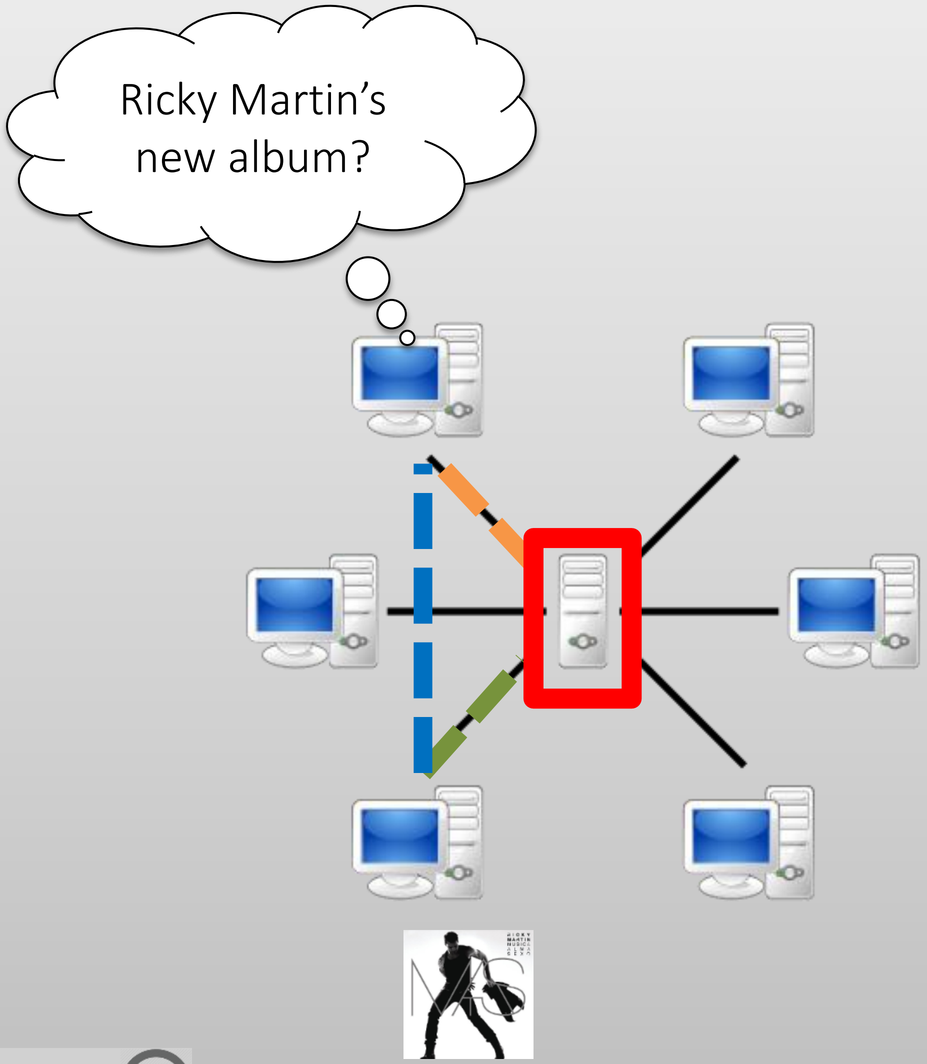


Peer-to-Peer: Unstructured (flooding)



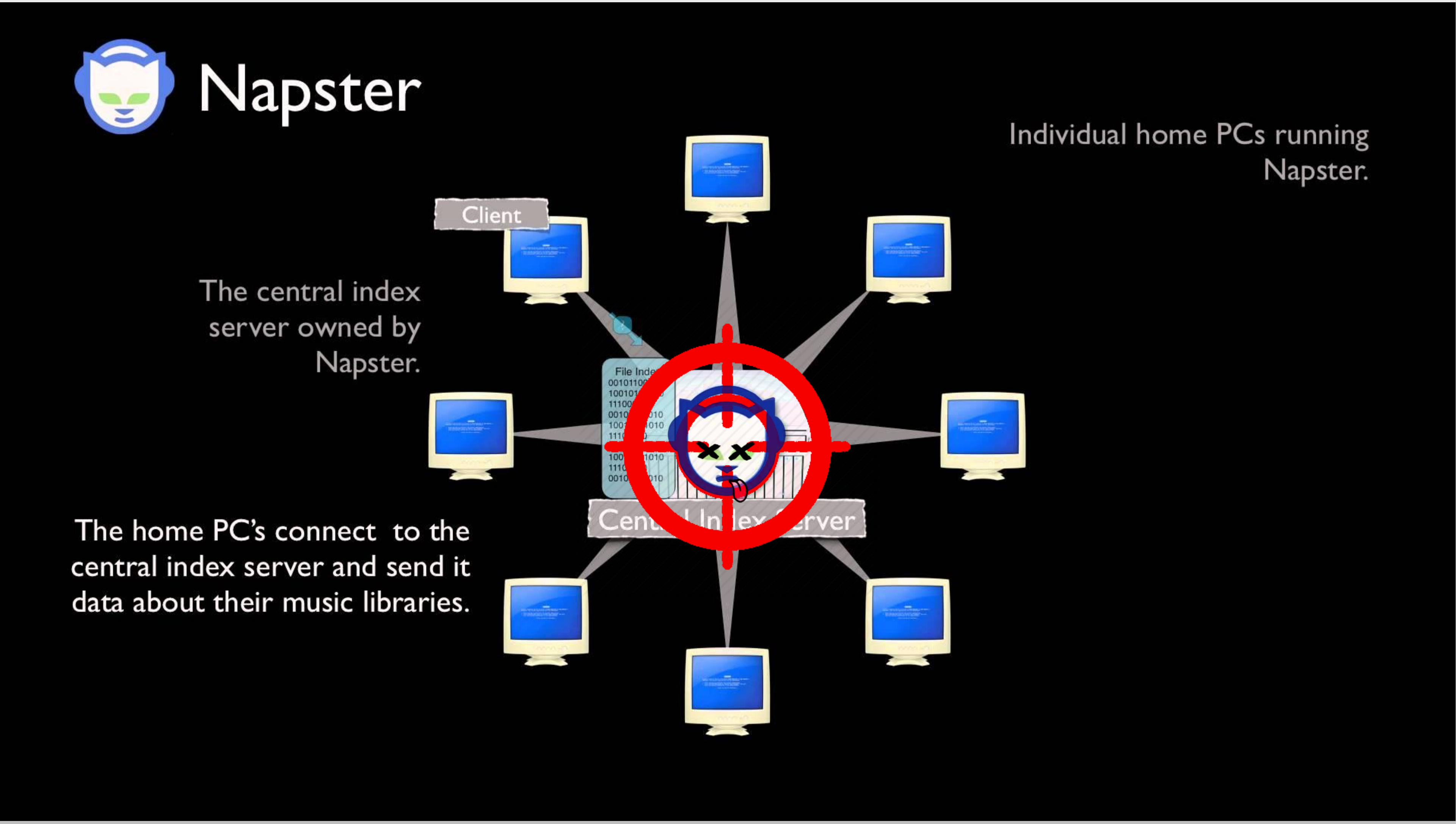
Peer-to-Peer: Structured (Central)

- In central server, each peer registers
  - Content
  - Address
- Peer requests content from server
- Peers connect directly

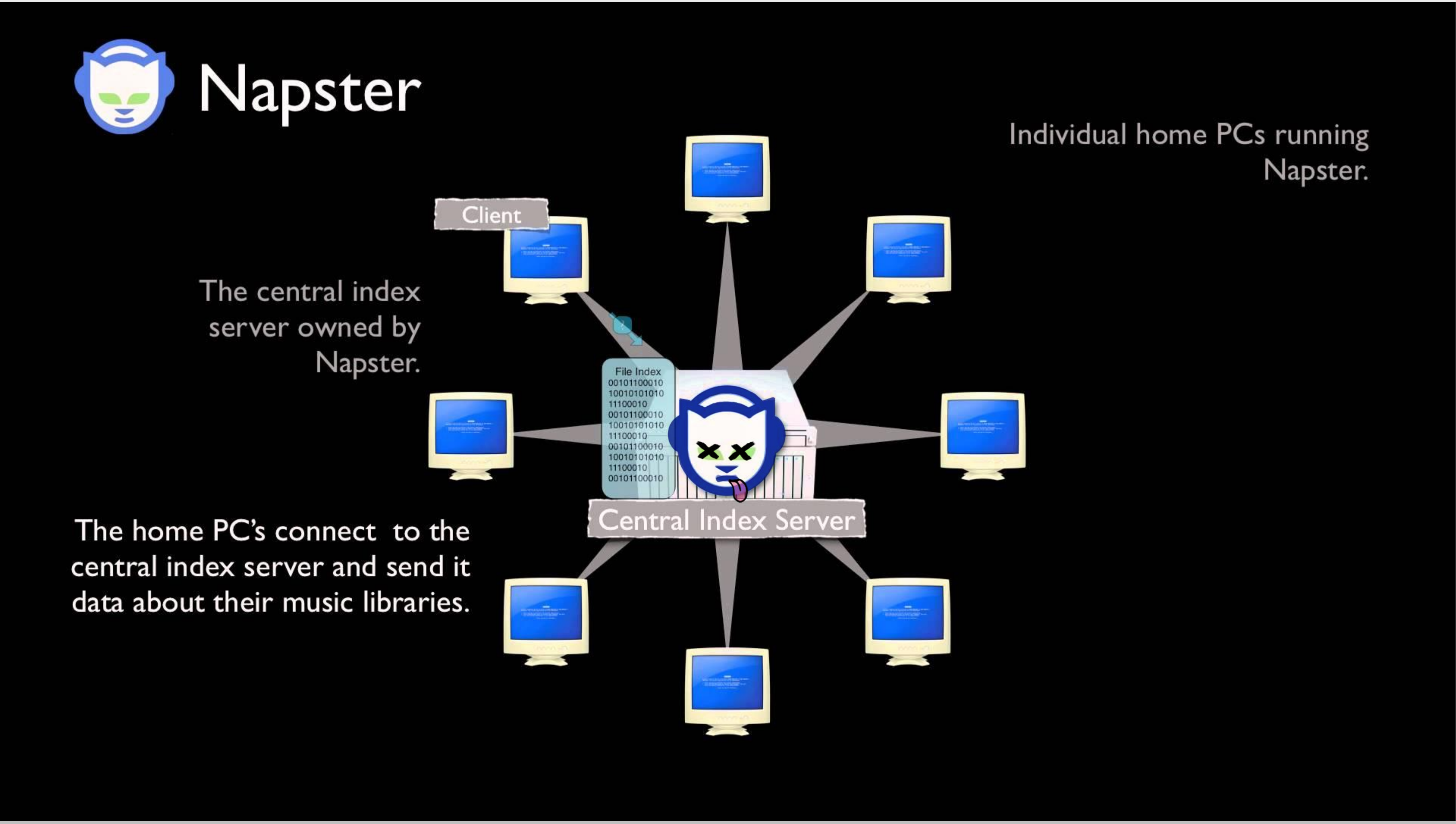


Advantages / Disadvantages?

Dangers of SPoF: not just technical

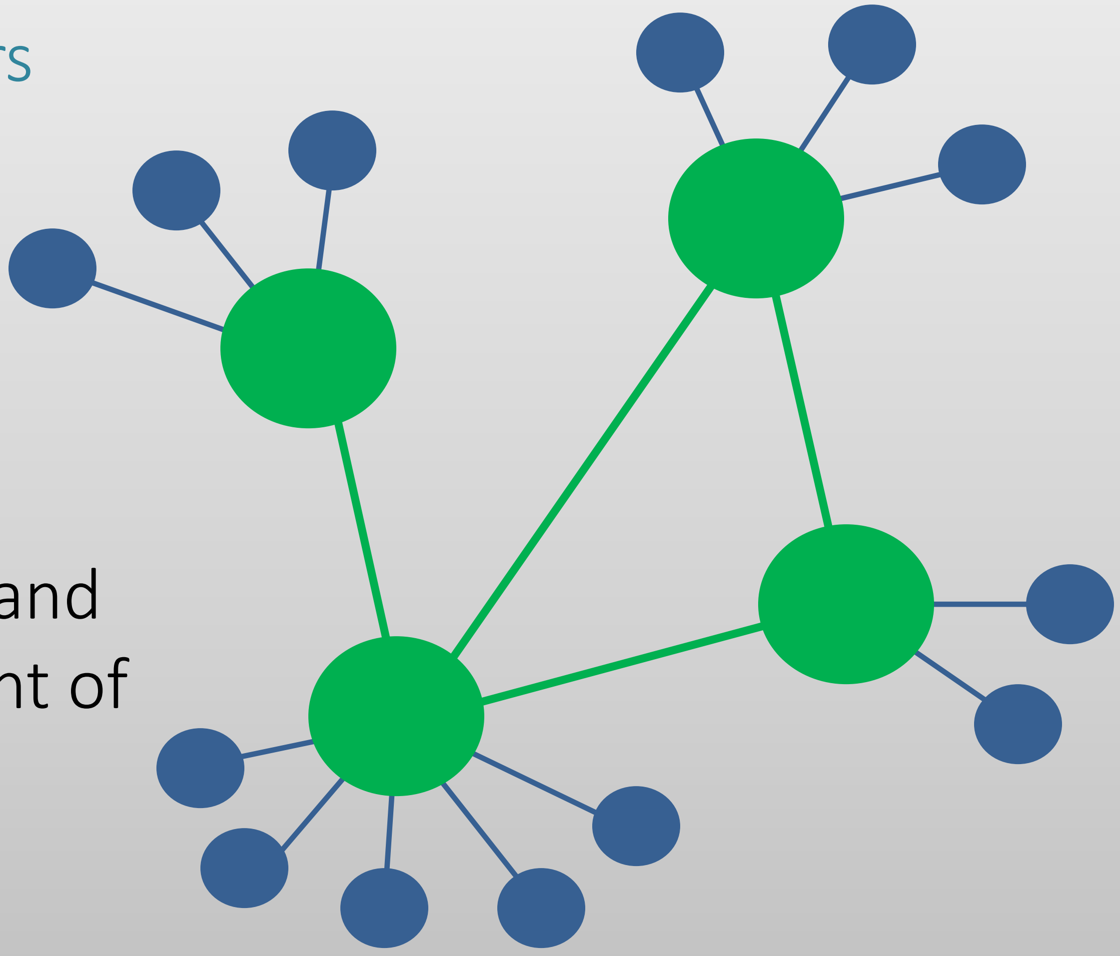


Dangers of SPoF: not just technical



Peer-to-Peer: Structured (Hierarchical)

Super-peers and peers



- Super-peers index and organise the content of local peers

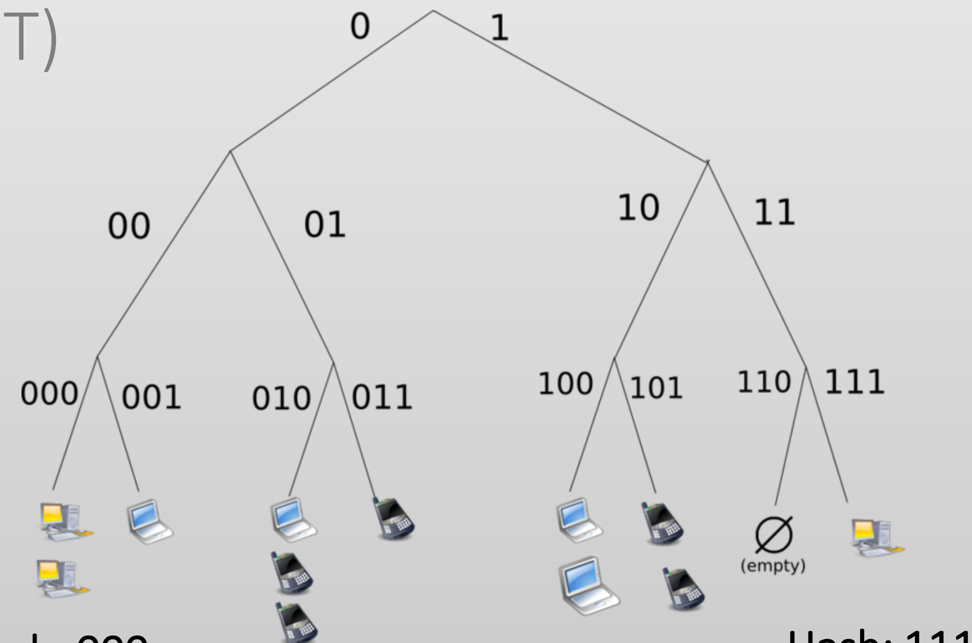
Advantages / Disadvantages?

Peer-to-Peer: Structured (Distributed Index)

Often a:

Distributed Hash Table (DHT)

- `(key,value)` pairs
- Hash on `key`
- Insert with `(key,value)`
- Peer indexes `key` range



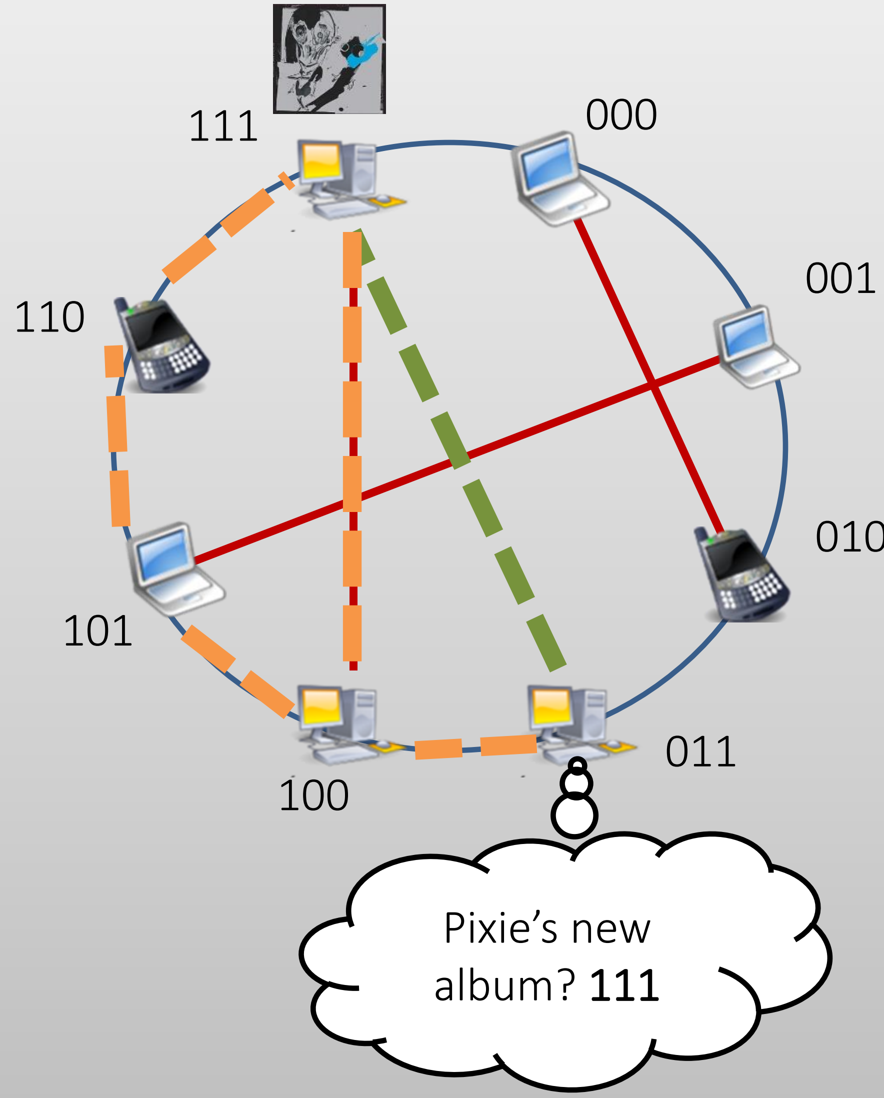
Hash: 000

Hash: 111

Advantages / Disadvantages?

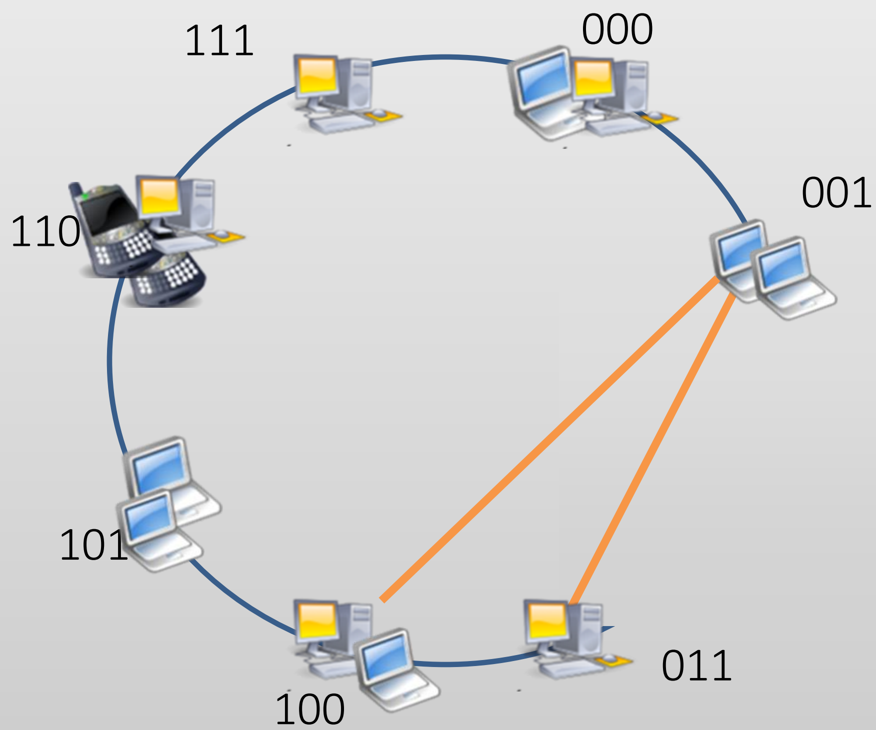
Peer-to-Peer: Structured (DHT)

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



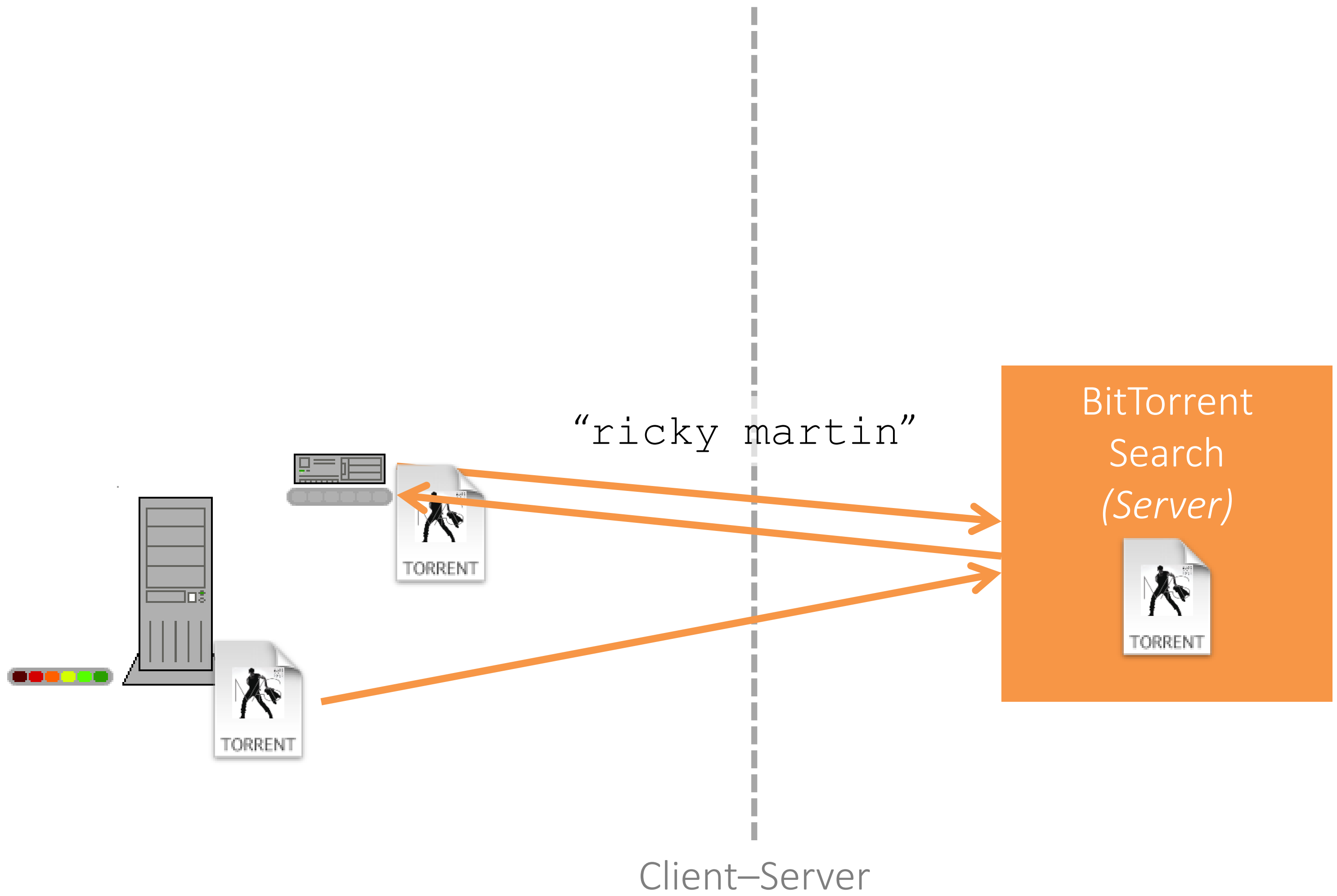
Peer-to-Peer: Structured (DHT)

- Handle peers leaving (churn)
  - Keep *n* successors
- New peers
  - Fill gaps
  - Replicate

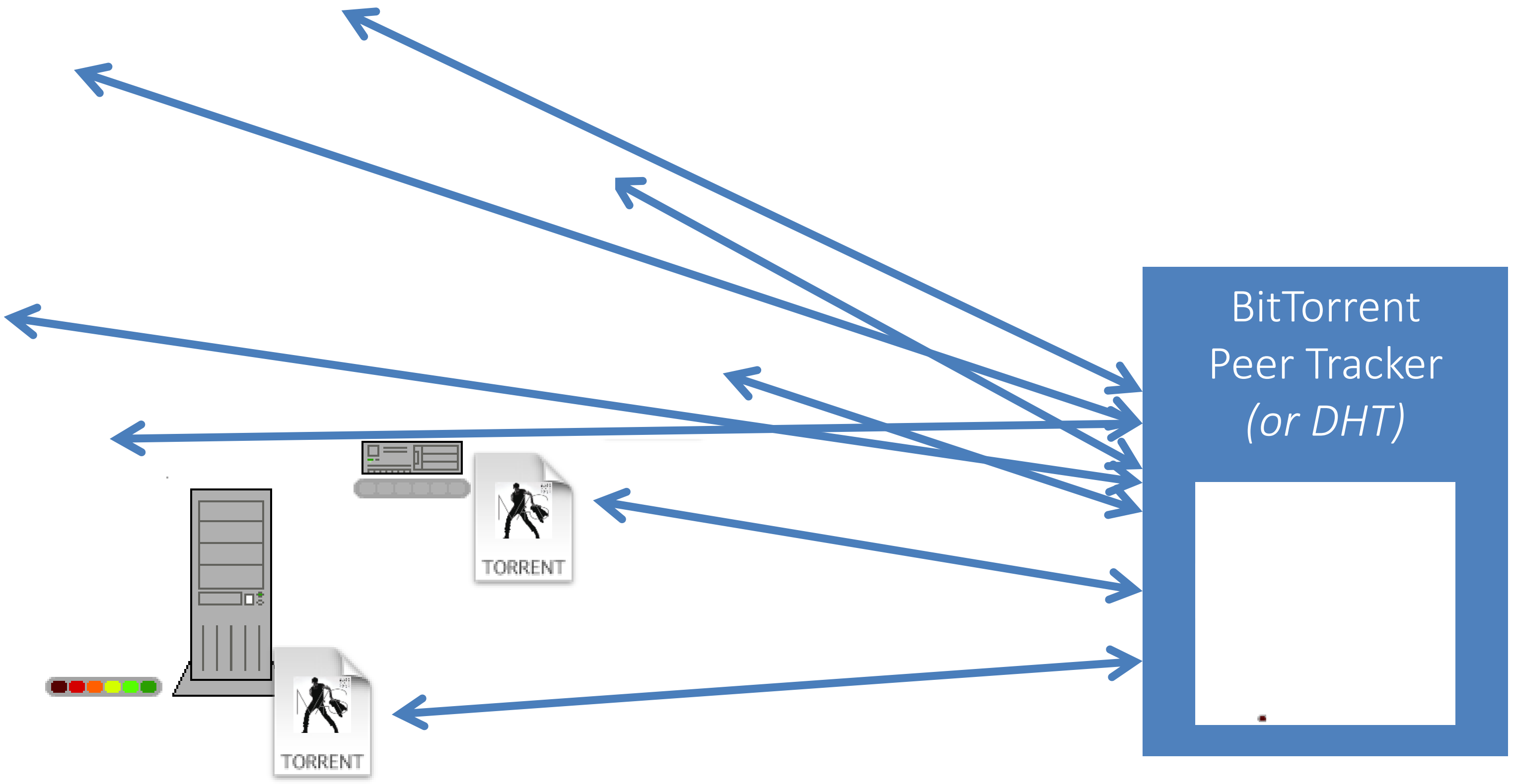


DISTRIBUTED SYSTEMS:
HYBRID EXAMPLE (BITTORRENT)

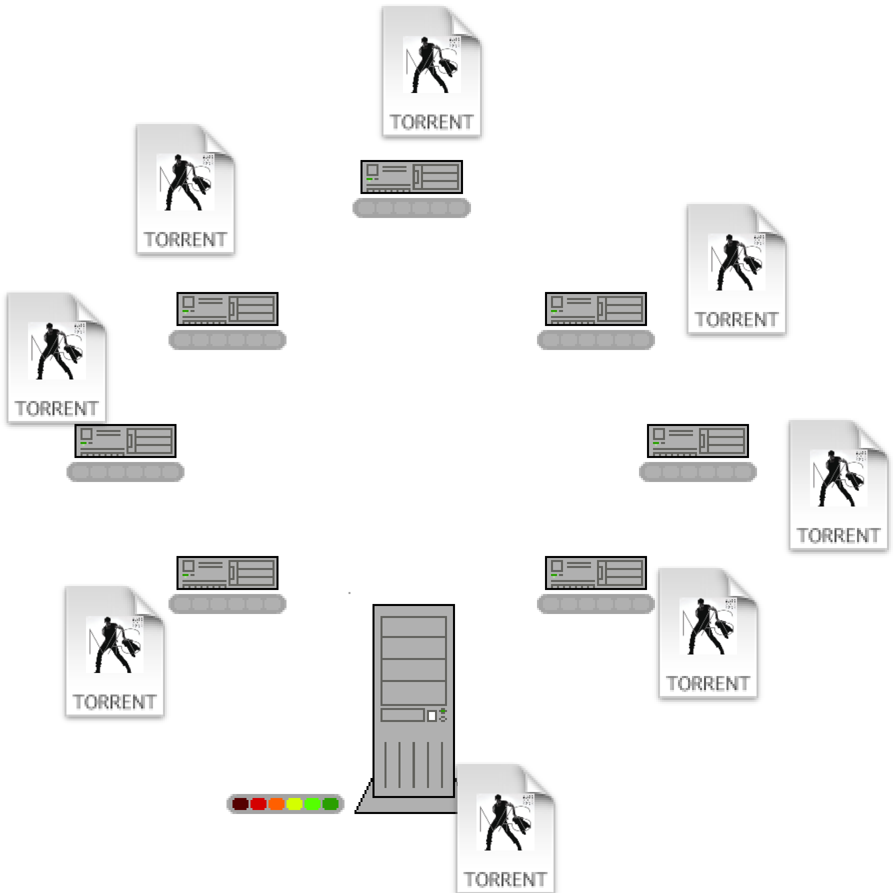
Bittorrent: Search Server



Bittorrent: Tracker



Bittorrent: File-Sharing



Bittorrent: Hybrid

Uploader

1. Creates torrent file
2. Uploads torrent file
3. Announces on tracker
4. Monitors for downloaders
5. Connects to downloaders
6. Sends file parts

Downloader

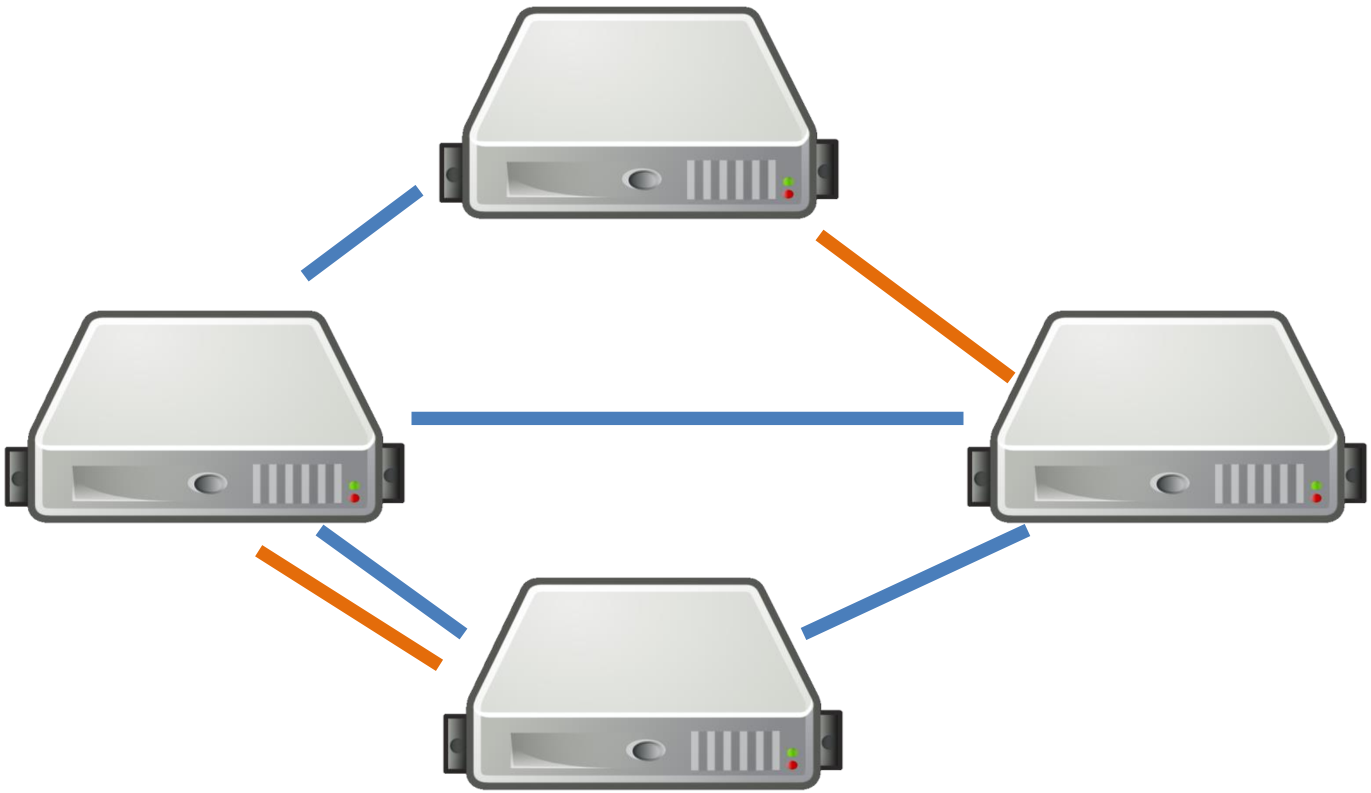
1. Searches torrent file
2. Downloads torrent file
3. Announces to tracker
4. Monitors for peers/seeds
5. Connects to peers/seeds
6. Sends & receives file parts
7. Watches illegal movie

Local / Client–Server / Structured P2P / Direct P2P

Distributed Systems:
In the Real World

Physical Location: Cluster Computing

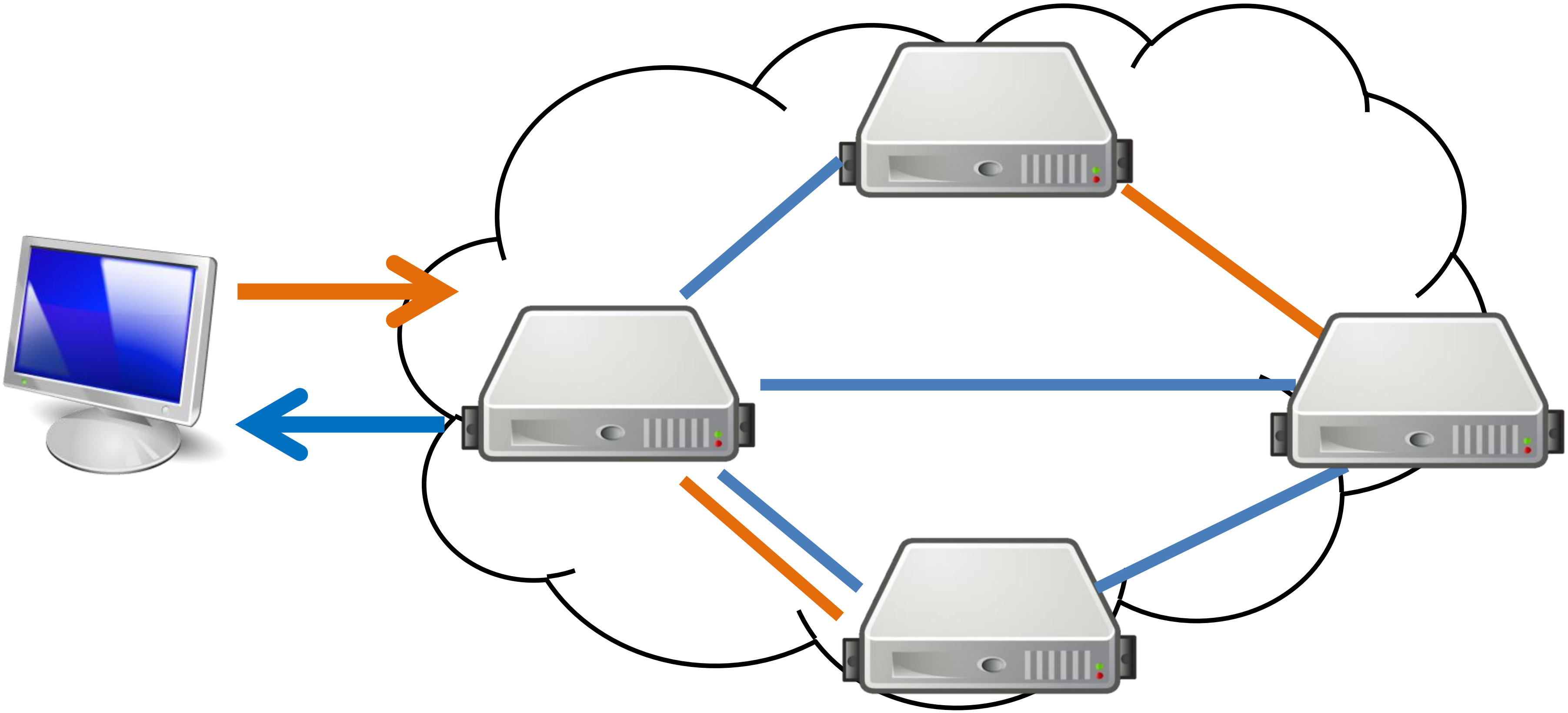
- Machines (typically) in a central, local location; e.g., a local LAN in a server room



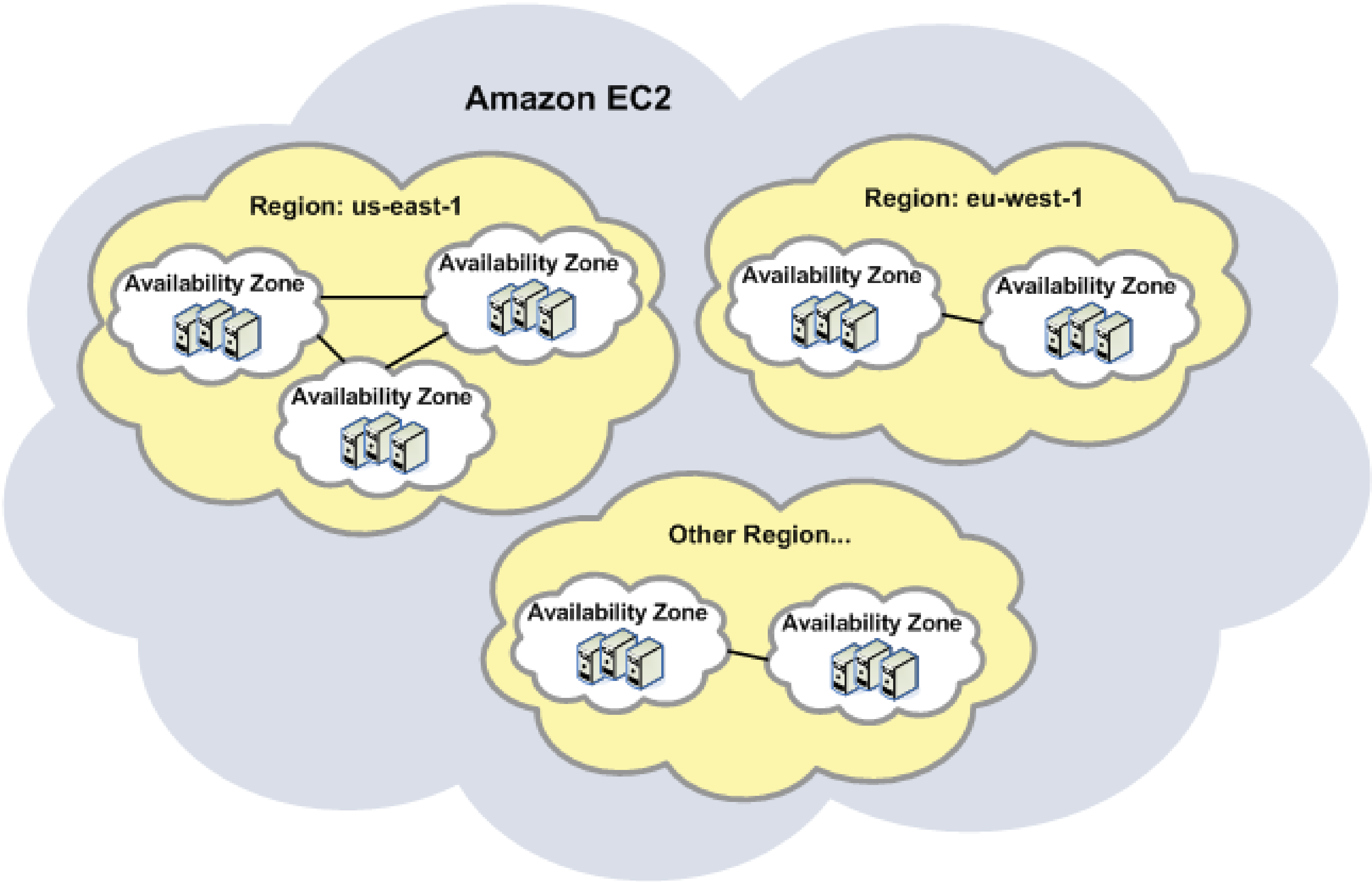
Physical Location: Cluster Computing

Physical Location: Cloud Computing

- Machines (typically) in a central remote location; e.g., Amazon EC2

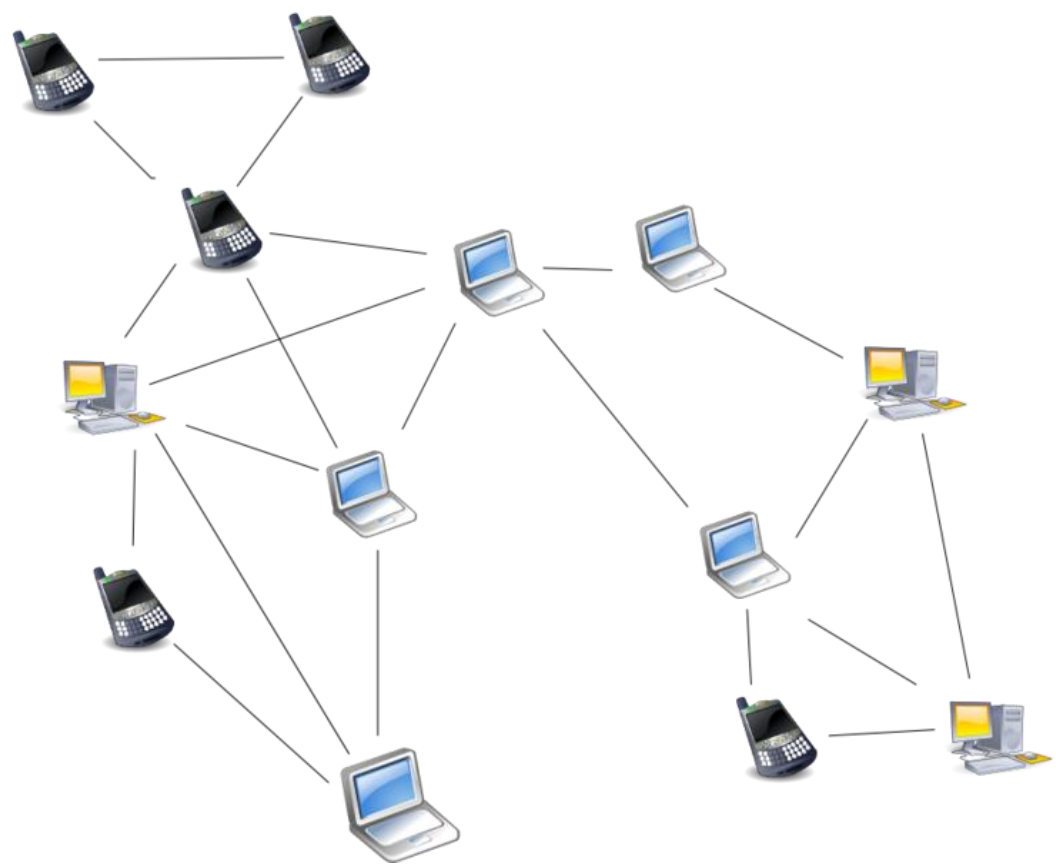


Physical Location: Cloud Computing



Physical Location: Grid Computing

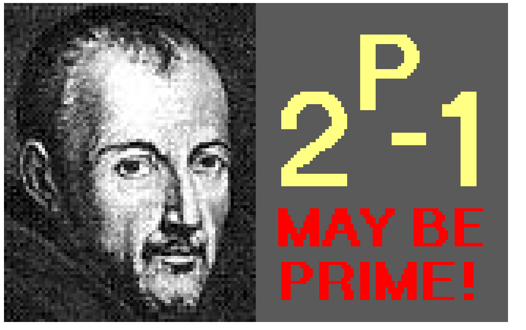
- Machines in diverse locations



Physical Location: Grid Computing

Physical Location: Grid Computing

$$2^{82,589,933} - 1$$



Physical Locations

- Cluster computing:
  - *Typically* centralised, local
- Cloud computing:
  - *Typically* centralised, remote
- Grid computing:
  - *Typically* decentralised, remote

Lab II Preview:
Distributed System

Messaging System

Distributed messaging system

Central server (optional; IP known globally)

Peer machines (IP unknown to other machines initially)

How can we design a system such that:

- Peers find the IPs of other peers
- Peers can send and receive messages to/from other peers

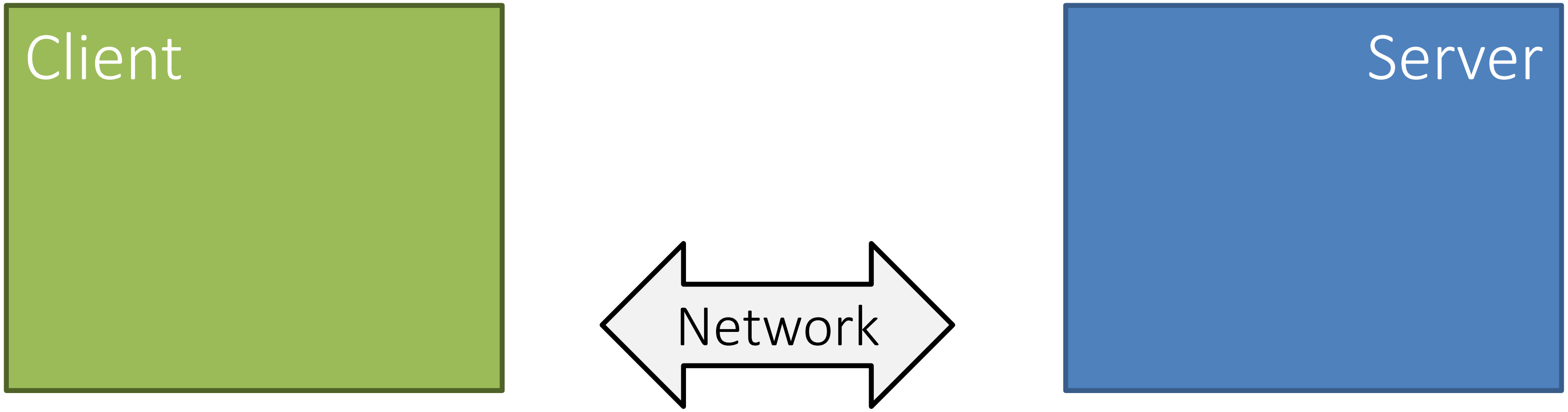
Lab II Preview:
Java RMI Overview

Why is Java RMI Important?

We can use it to quickly build distributed systems using Java and the Internet.

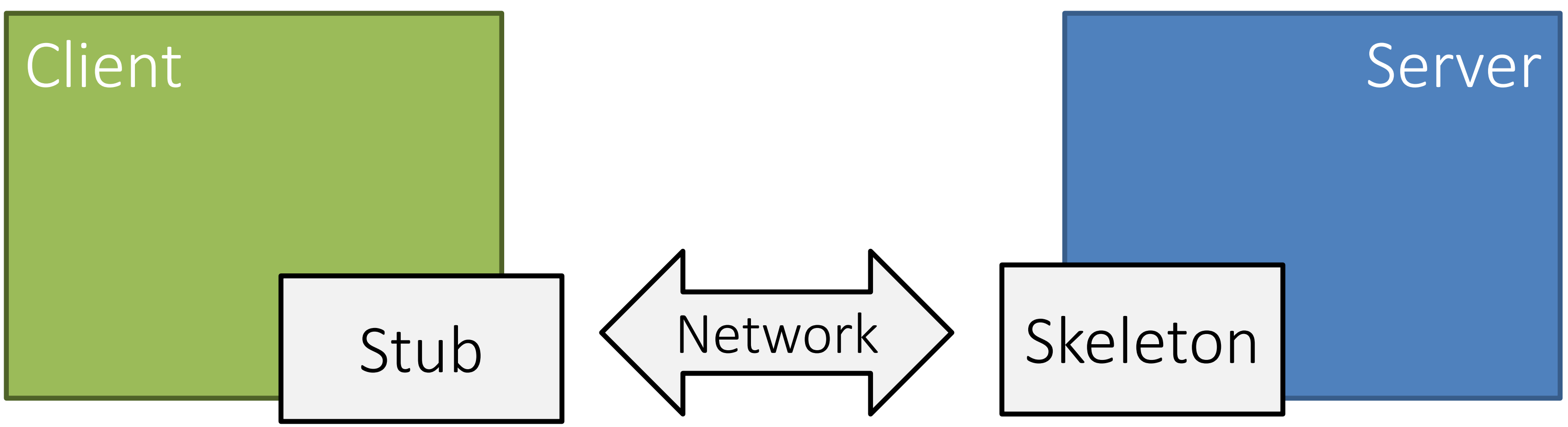
What is Java RMI?

- Server: has Java code implemented
- Client: wants to call Java code on server
  (possibly sending arguments and receiving a return value)



What is Java RMI?

- RMI = Remote Method Invocation
- Stub / Skeleton model (TCP/IP)



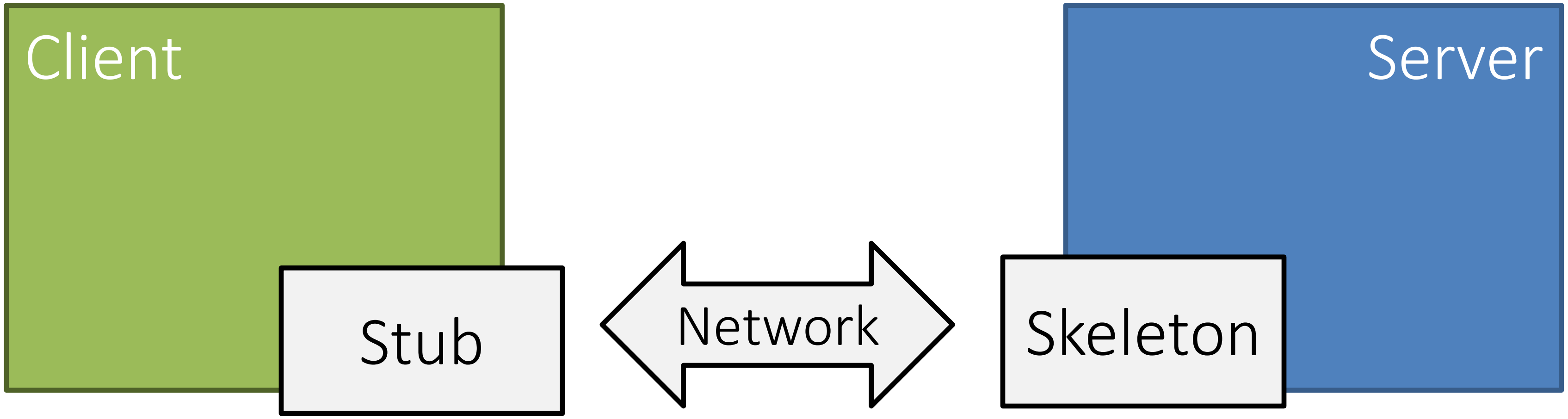
What is Java RMI?

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);
    }
}
```

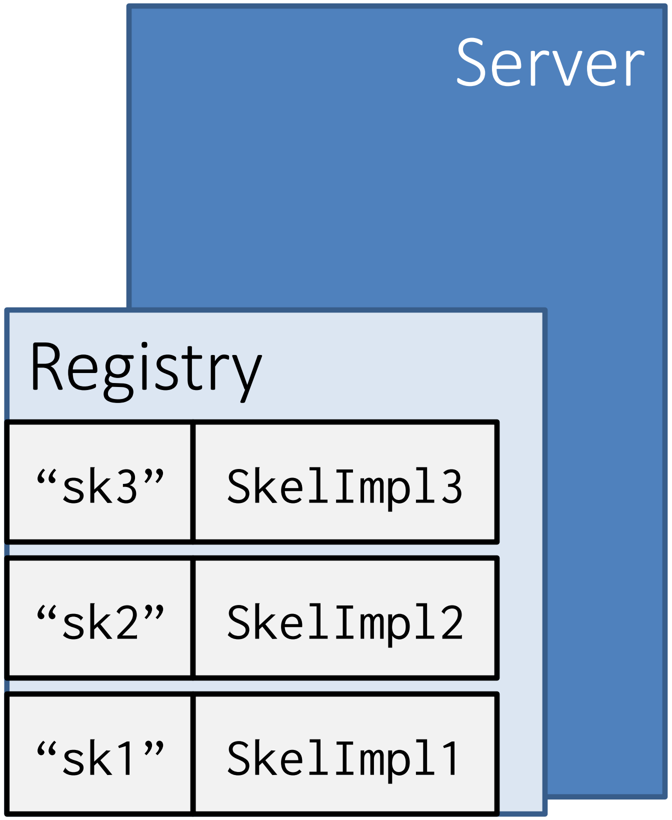
Problem?

Synchronisation:
(e.g., should use ConcurrentHashMap)

Server

Server Registry

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



Server Creates/Connects to Registry

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

OR

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

Server

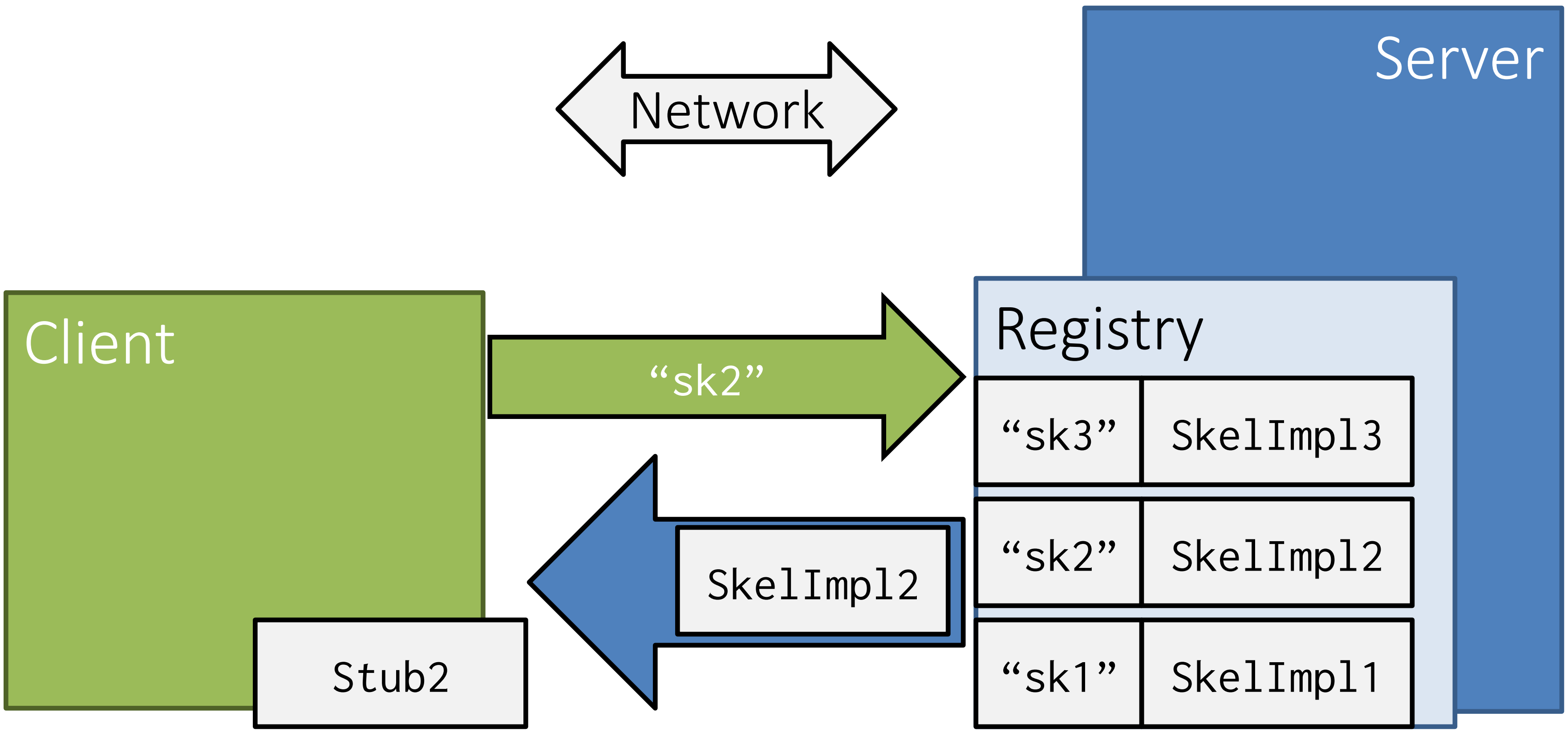
Server Registers Skeleton Implementation

```java
// 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



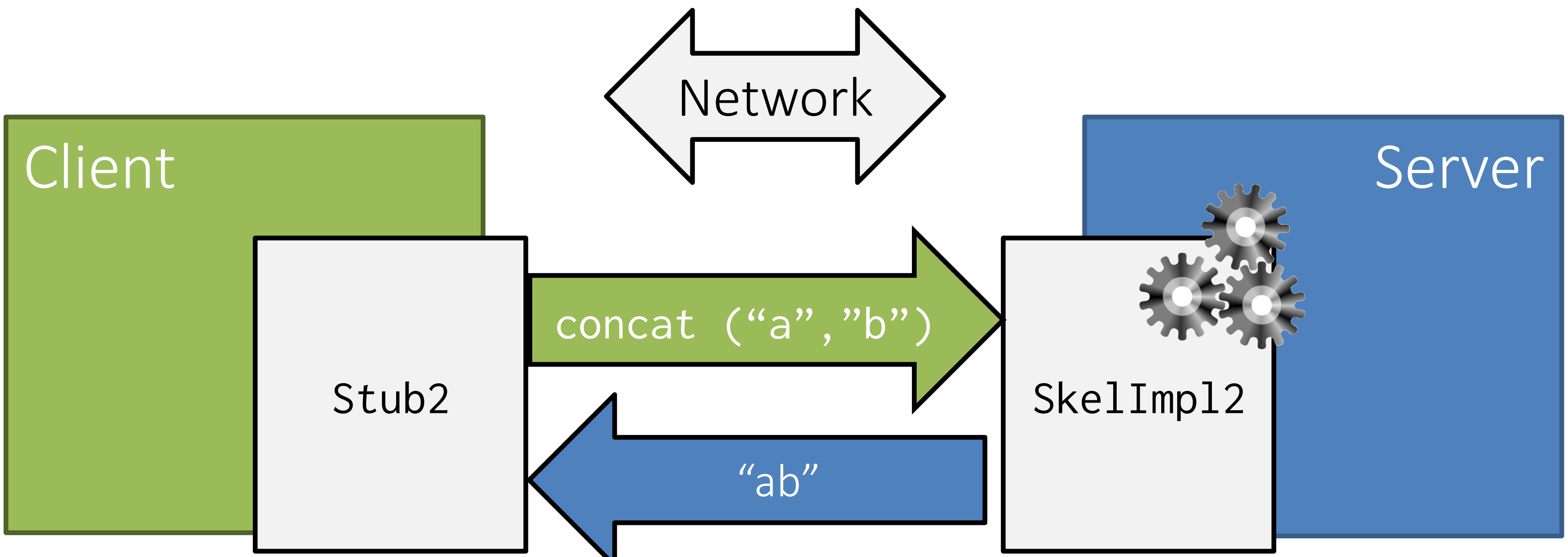
Client Connecting to Registry

```java
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

```java
// 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`

Questions?