CC5212-1
Procesamiento Masivo de Datos
Otoño 2016

Lecture 2: Distributed Systems I

Aidan Hogan
aidhog@gmail.com

MASSIVE DATA NEEDS DISTRIBUTED SYSTEMS …

Monolithic vs. Distributed Systems

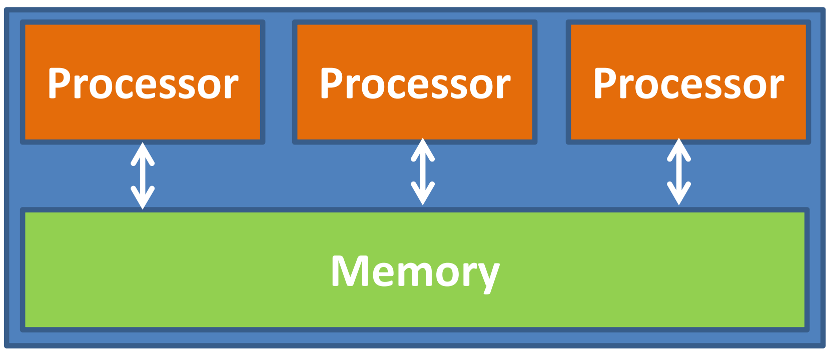
- One machine that's *n* times as powerful?

vs.

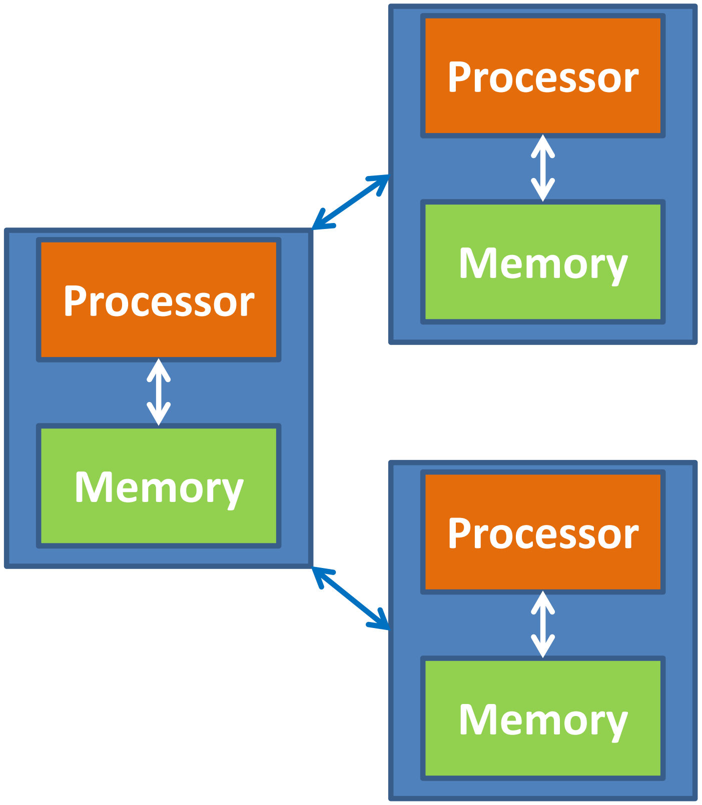
- *n* machines that are equally as powerful?

Parallel vs. Distributed Systems

- Parallel System
  - often = *shared memory*

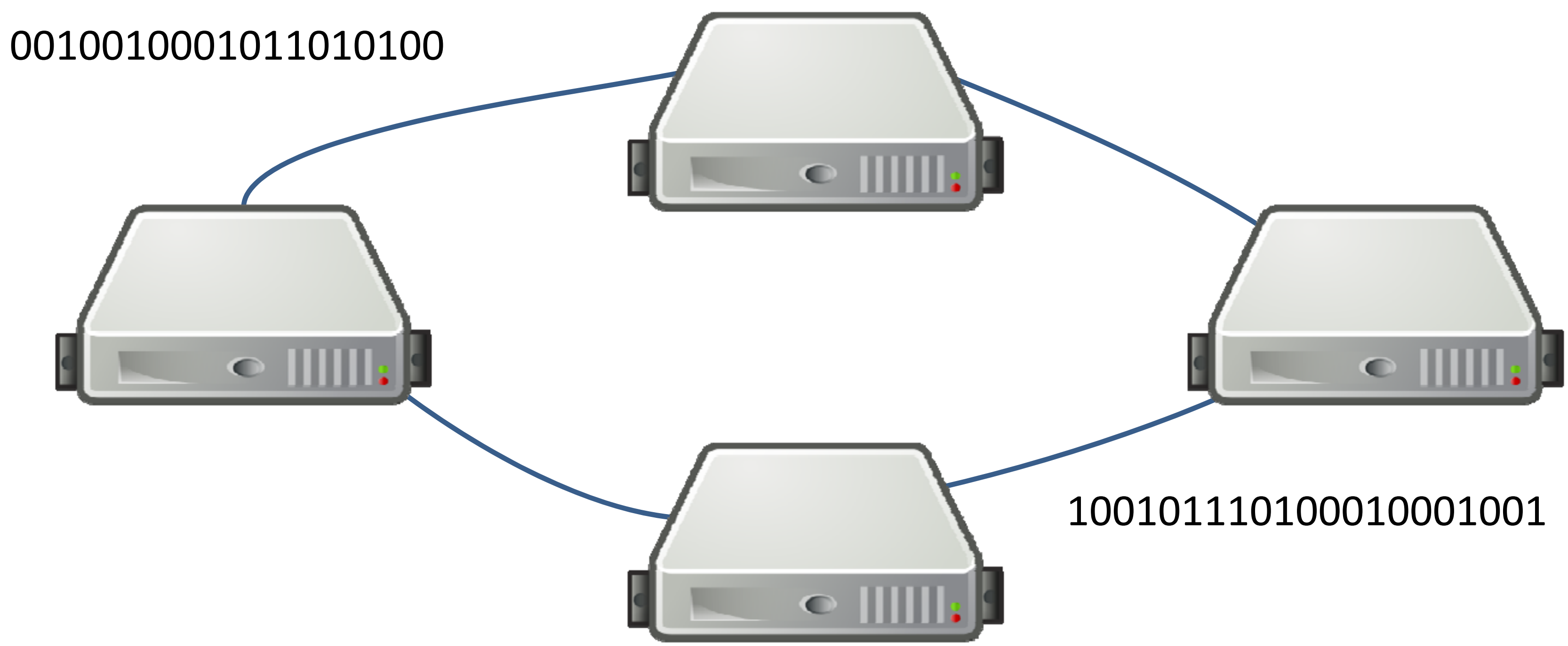


- Distributed System
  - often = *shared nothing*



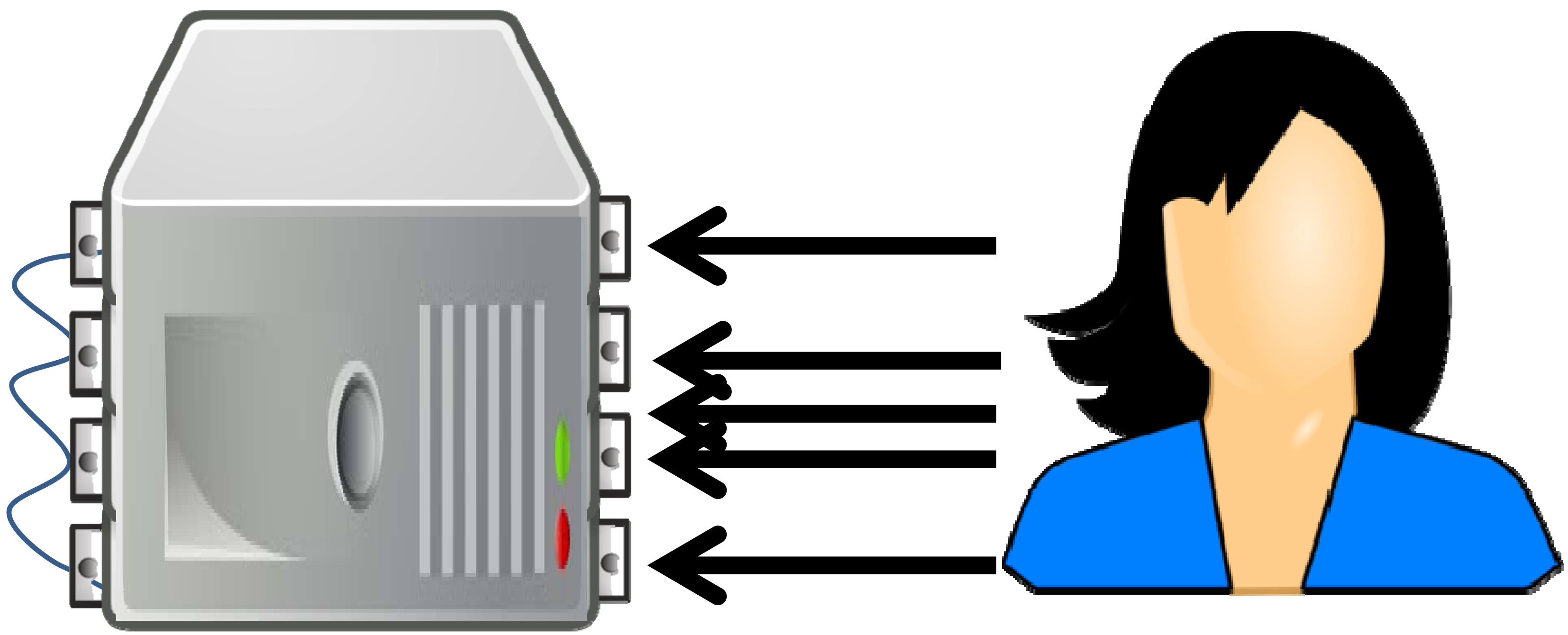
What is a Distributed System?

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



What is a Distributed System?

*"An **ideal** distributed system is a system that makes a collection of independent computers look like one computer (to the user)."*



Disadvantages of Distributed Systems

(Possible) Advantages

- Cost
  - Better performance/price
- Extensibility
  - Add another machine!
- Reliability
  - No central point of failure!
- Workload
  - Balance work automatically
- Sharing
  - Remote access to services

(Possible) Disadvantages

- Software
  - Need specialised programs
- Networking
  - Can be slow
- Maintenance
  - Debugging sw/hw a pain
- Security
  - Multiple users
- Parallelisation
  - Not always applicable

WHAT MAKES A GOOD DISTRIBUTED SYSTEM?

Distributed System Design

*"An **ideal** distributed system is a system that makes a collection of independent computers look like one computer (to the user)."*

- Transparency: Abstract/hide:
  - Access: How different machines are accessed
  - Location: What machines have what/if they move
  - Concurrency: Access by several users
  - Failure: Keep it a secret from the user

Distributed System Design

- Flexibility:
  - Add/remove/move machines
  - Generic interfaces

- Reliability:
  - Fault-tolerant: recover from errors
  - Security: user authentication
  - Availability: uptime/total-time

Distributed System Design

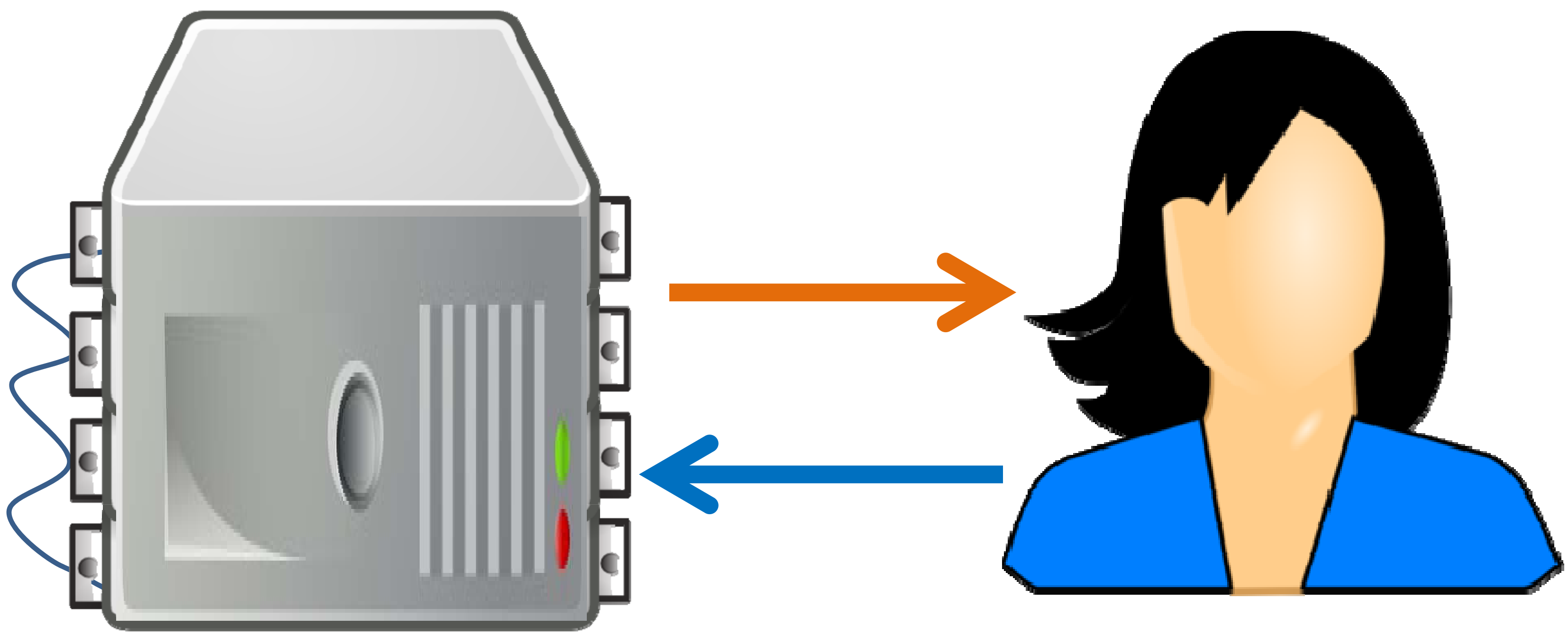
- Performance:
  - Runtimes (processing)
  - Latency, throughput and bandwidth (data)

- Scalability
  - Network and infrastructure scales
  - Applications scale
  - Minimise global knowledge/bottlenecks!

DISTRIBUTED SYSTEMS:
CLIENT–SERVER ARCHITECTURE

Client–Server Model

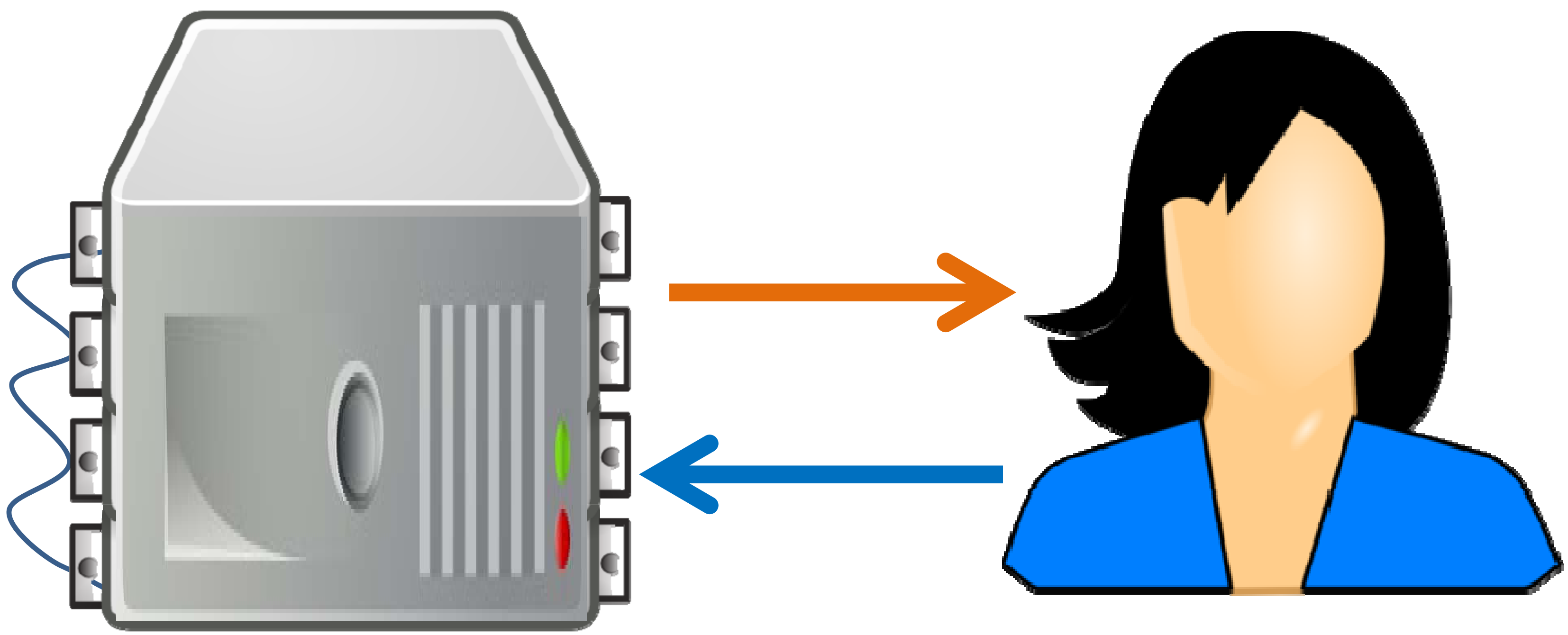
- Client makes request to server
- Server acts and responds



(<u>For example</u>: *Email, WWW, Printing, etc.*)

Client–Server: *Thin Client*

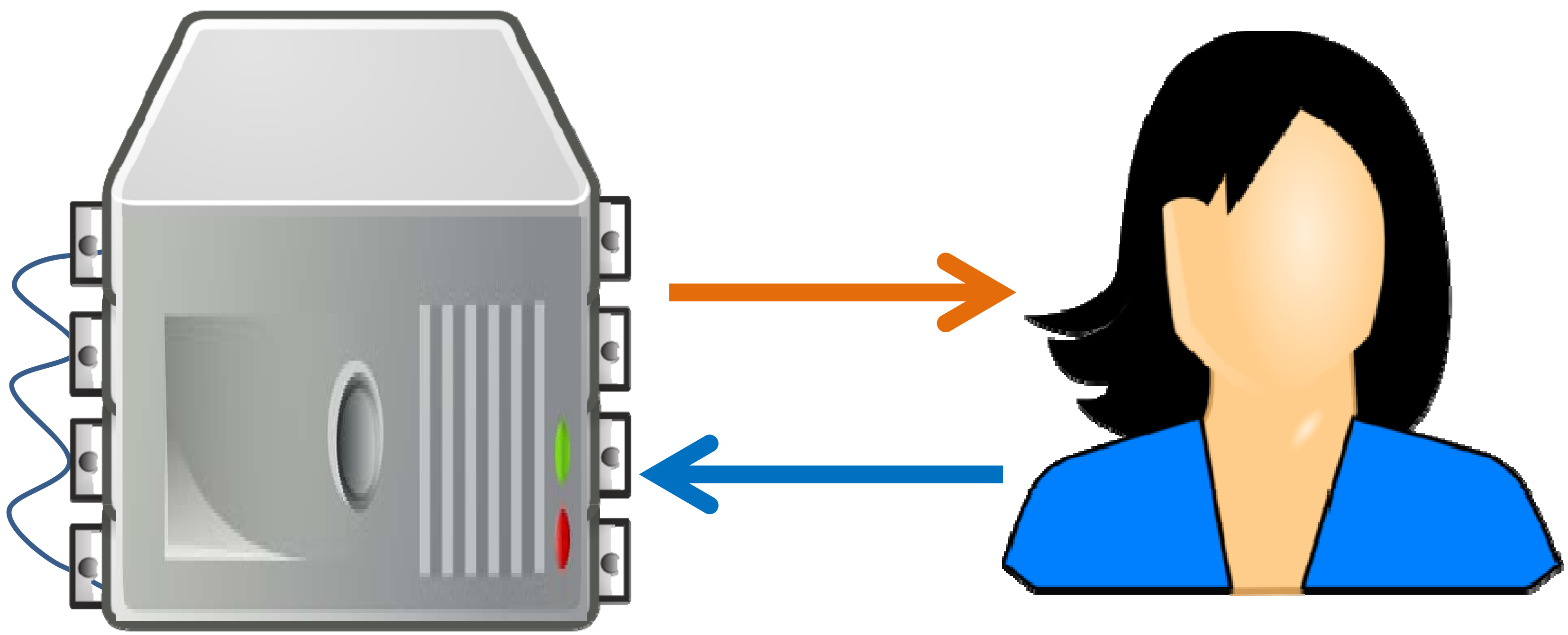
- Few computing resources for client: I/O
  - Server does the hard work



(<u>For example</u>: *PHP-heavy websites, SSH, email, etc.*)

Client–Server: *Fat Client*

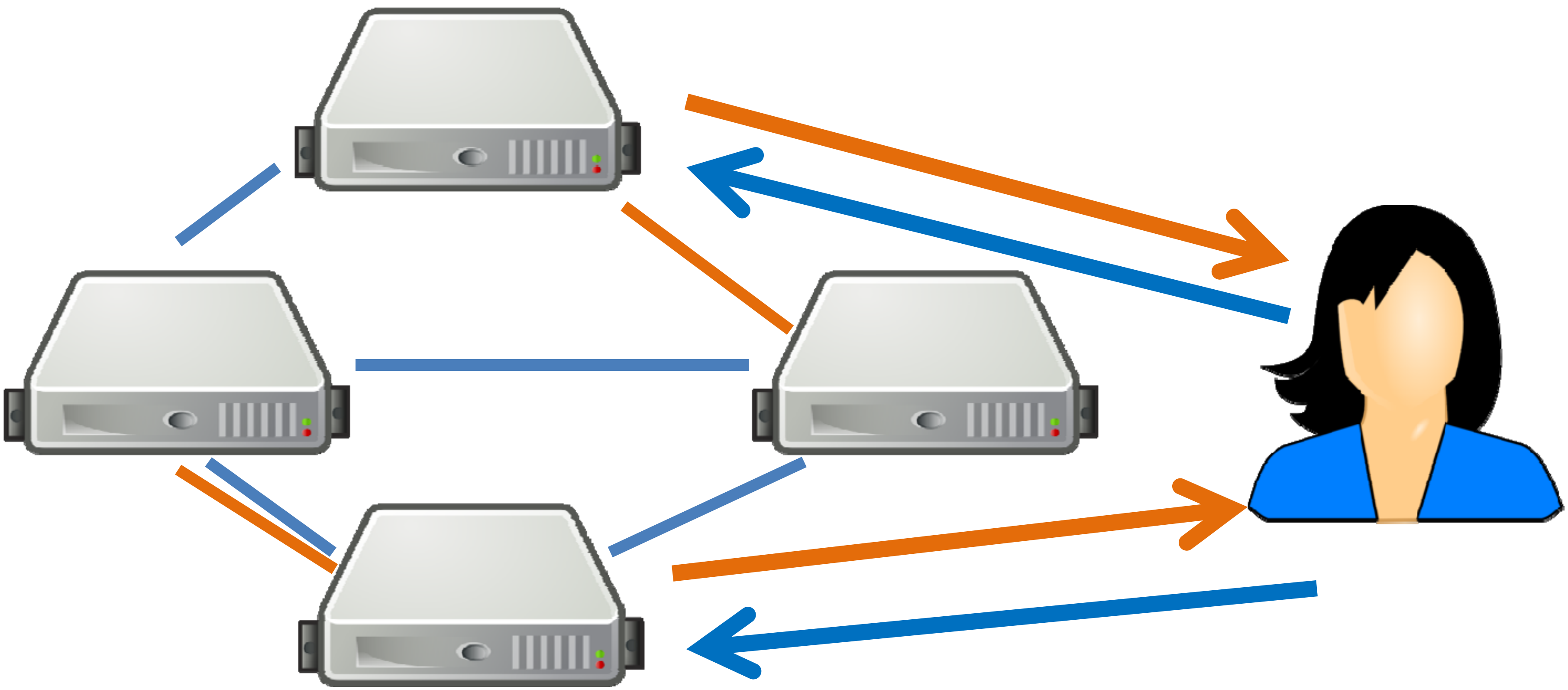
- Fuller computing resources for client: I/O
  - Server sends data: computing done client-side



(For example: *Javascript-heavy websites, multimedia, etc.*)

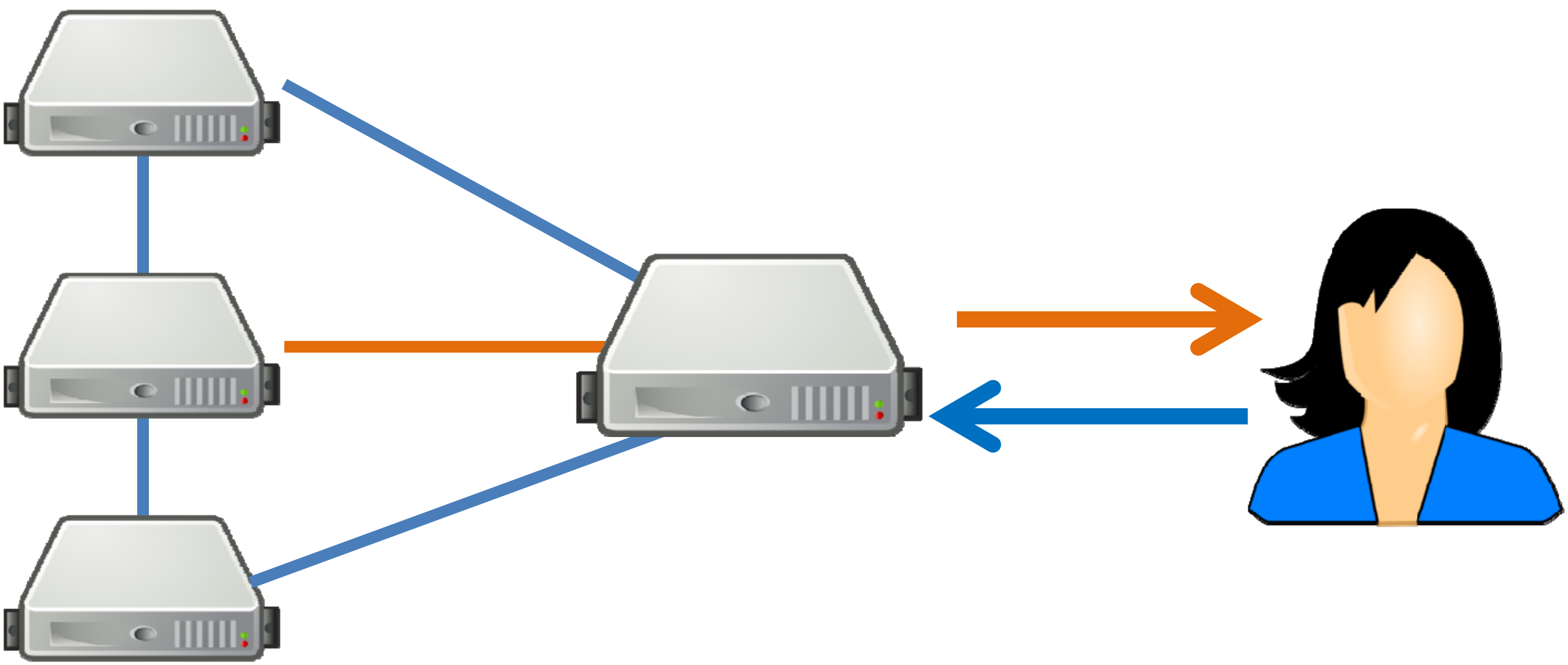
Client–Server: *Mirror Servers*

- User goes to any machine (replicated/mirror)

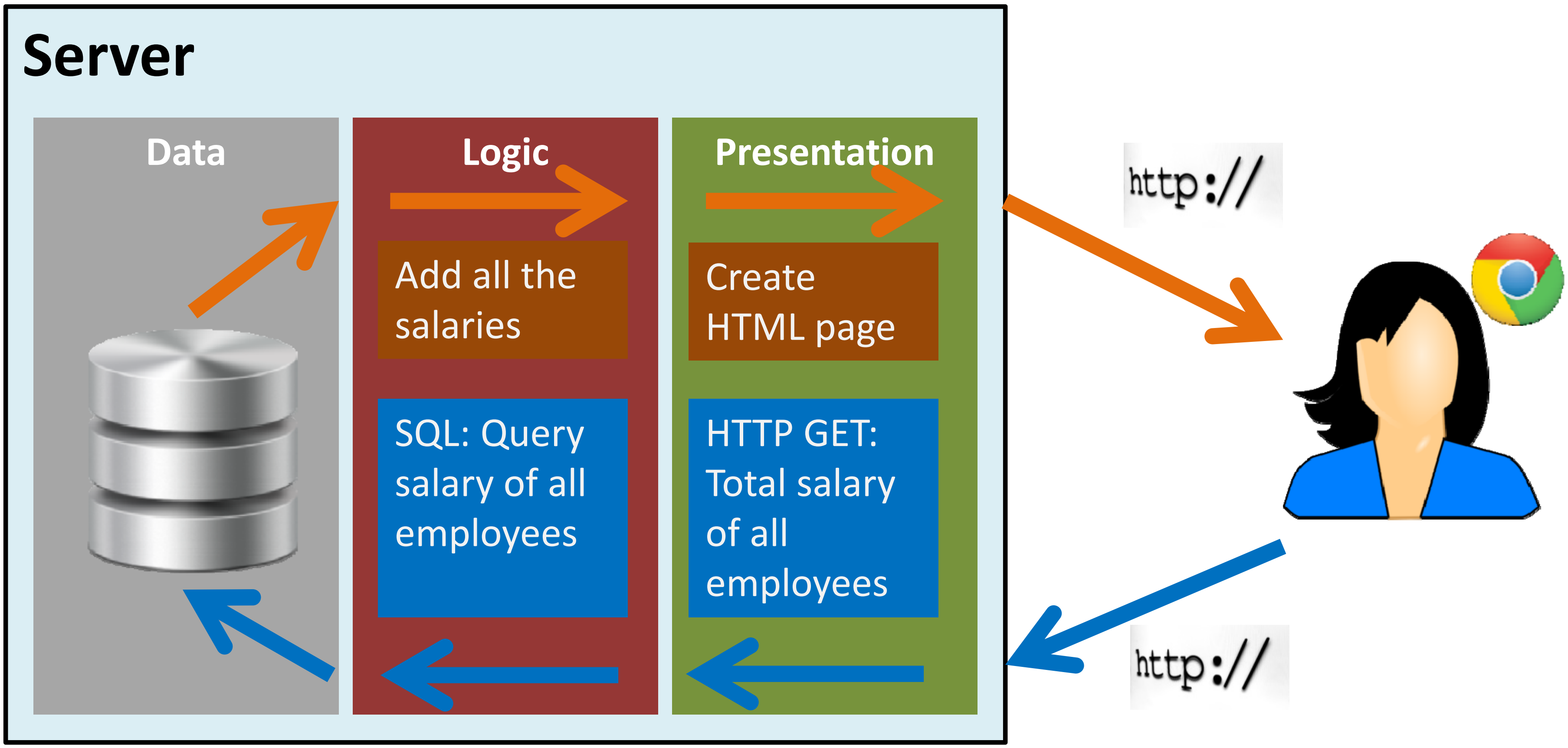


Client–Server: *Proxy Server*

- User goes to “forwarding” machine (proxy)

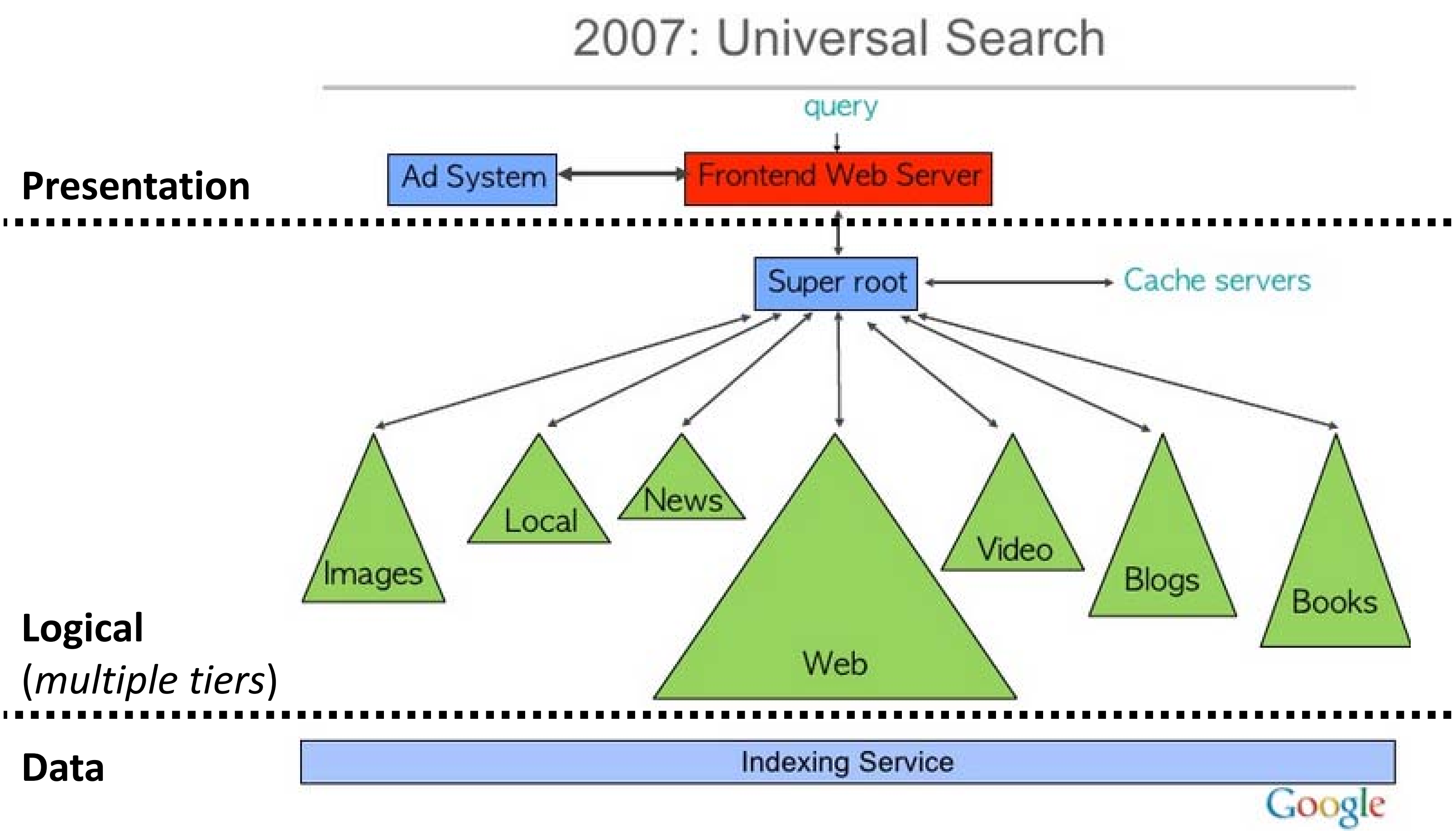


Client–Server: *Three-Tier Server*



Client–Server: *n-Tier Server*

- Slide from Google's Jeff Dean:

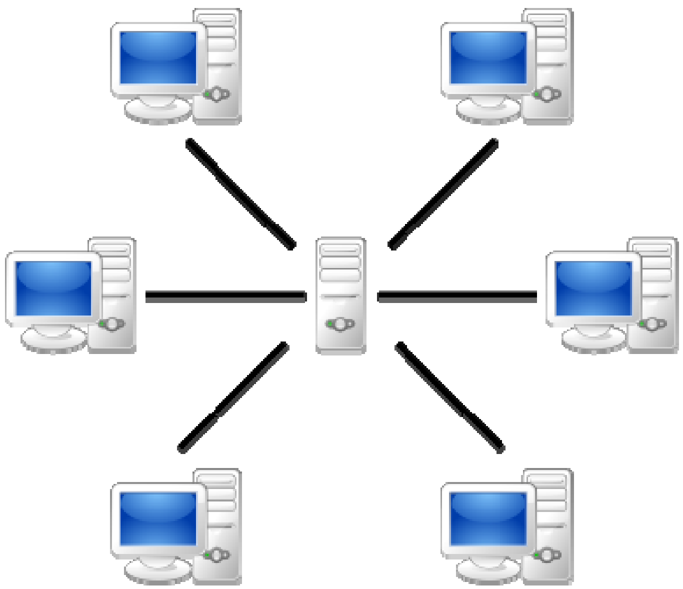


DISTRIBUTED SYSTEMS:
PEER-TO-PEER ARCHITECTURE

Peer-to-Peer (P2P)

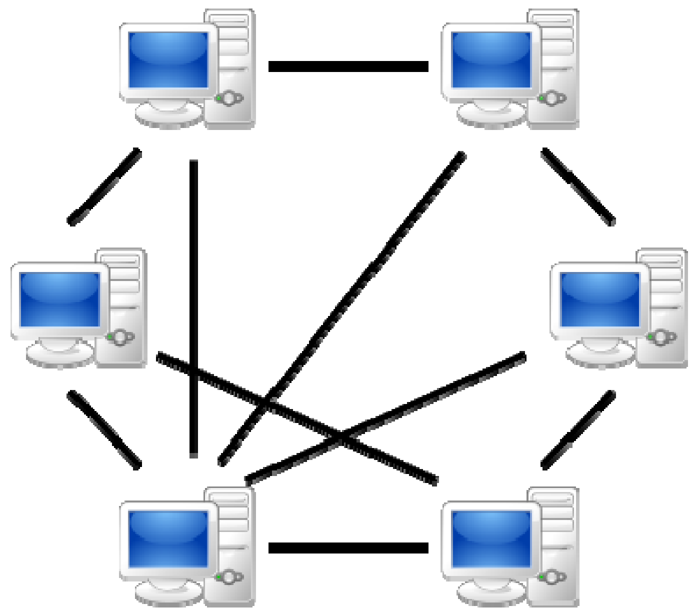
Client–Server

- Clients interact directly with a “central” server

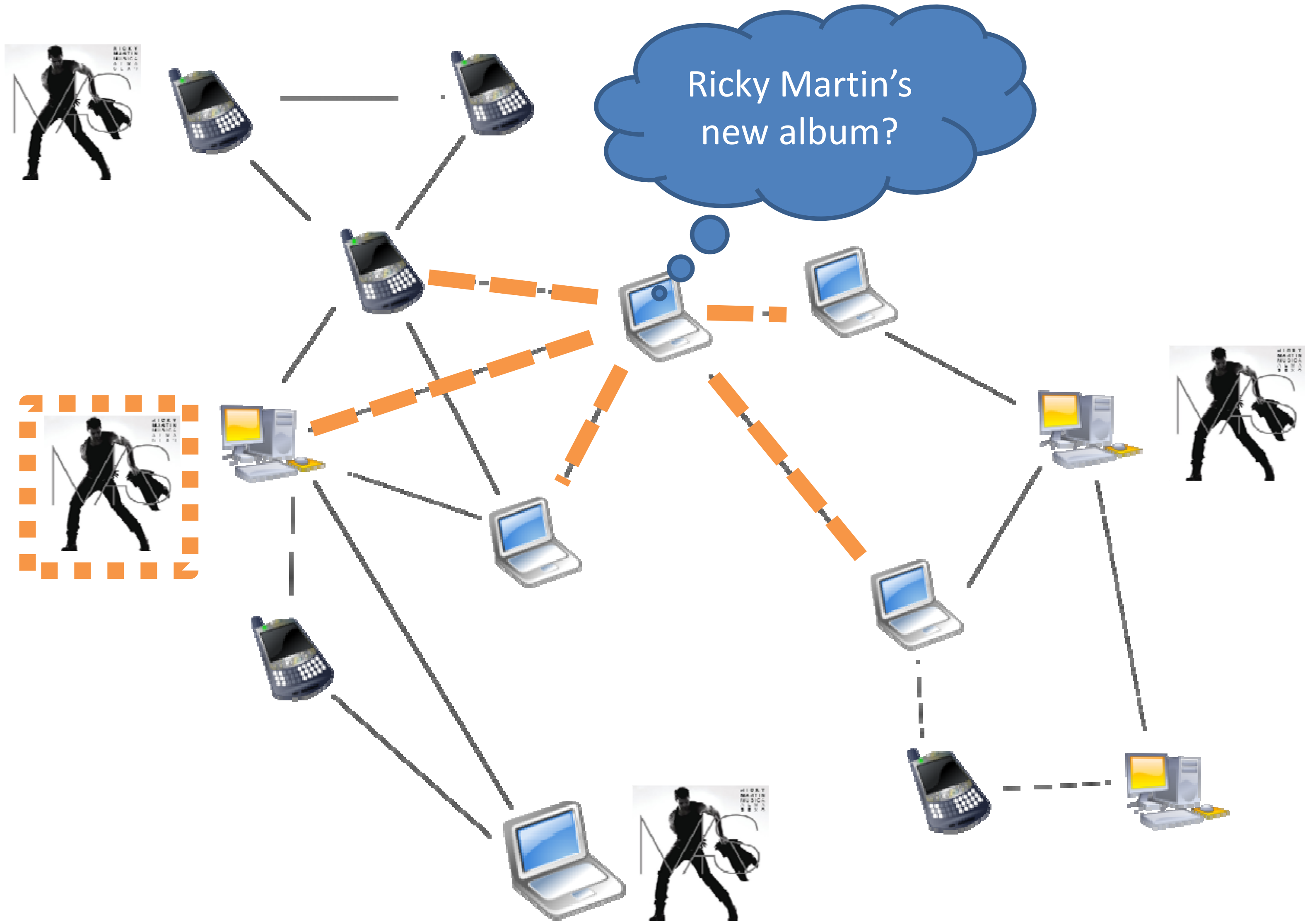


Peer-to-Peer

- Peers interact directly amongst themselves

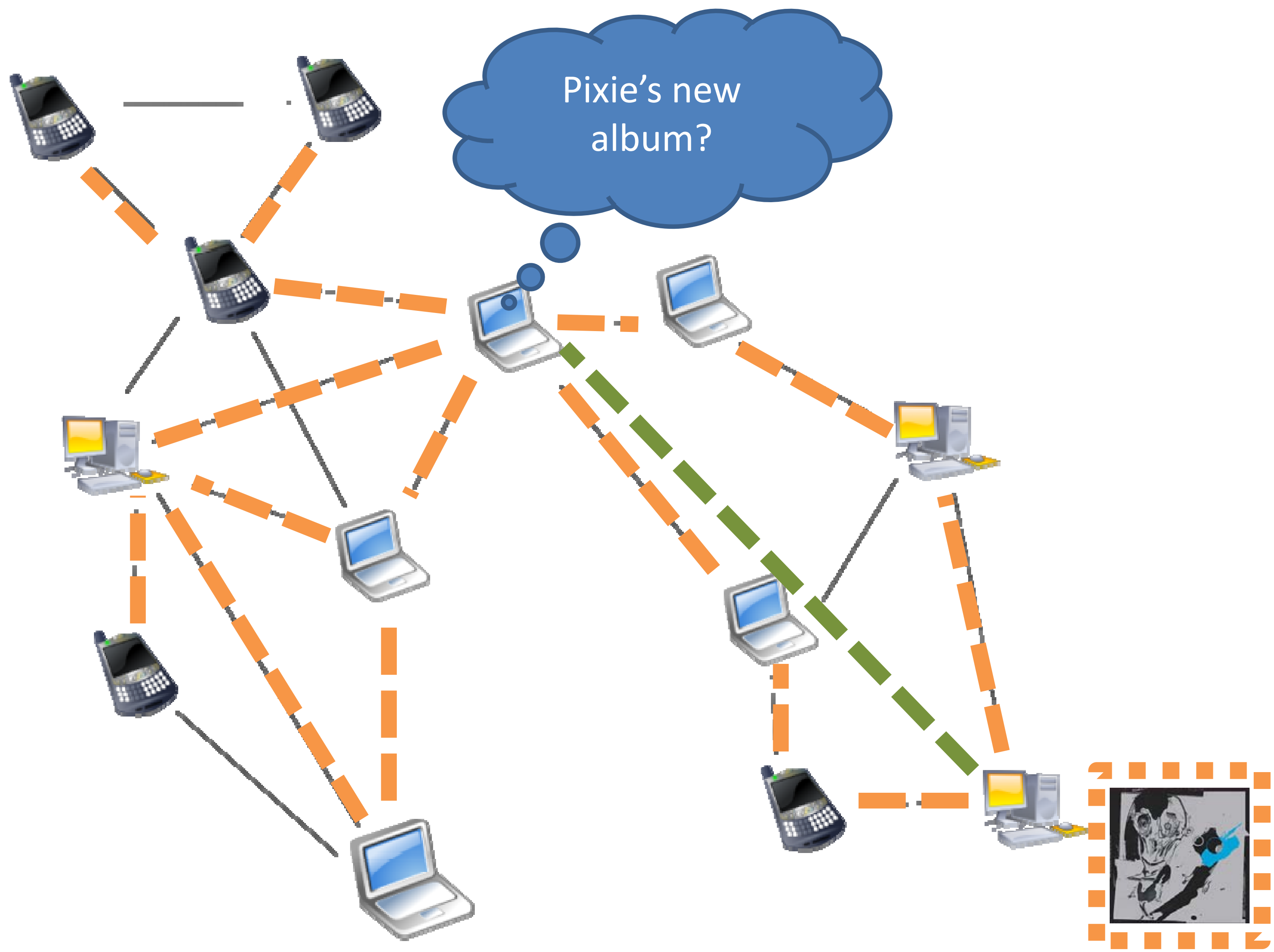


Peer-to-Peer: *Unstructured (flooding)*



(For example: *Kazaa, Gnutella*)

Peer-to-Peer: *Unstructured (flooding)*

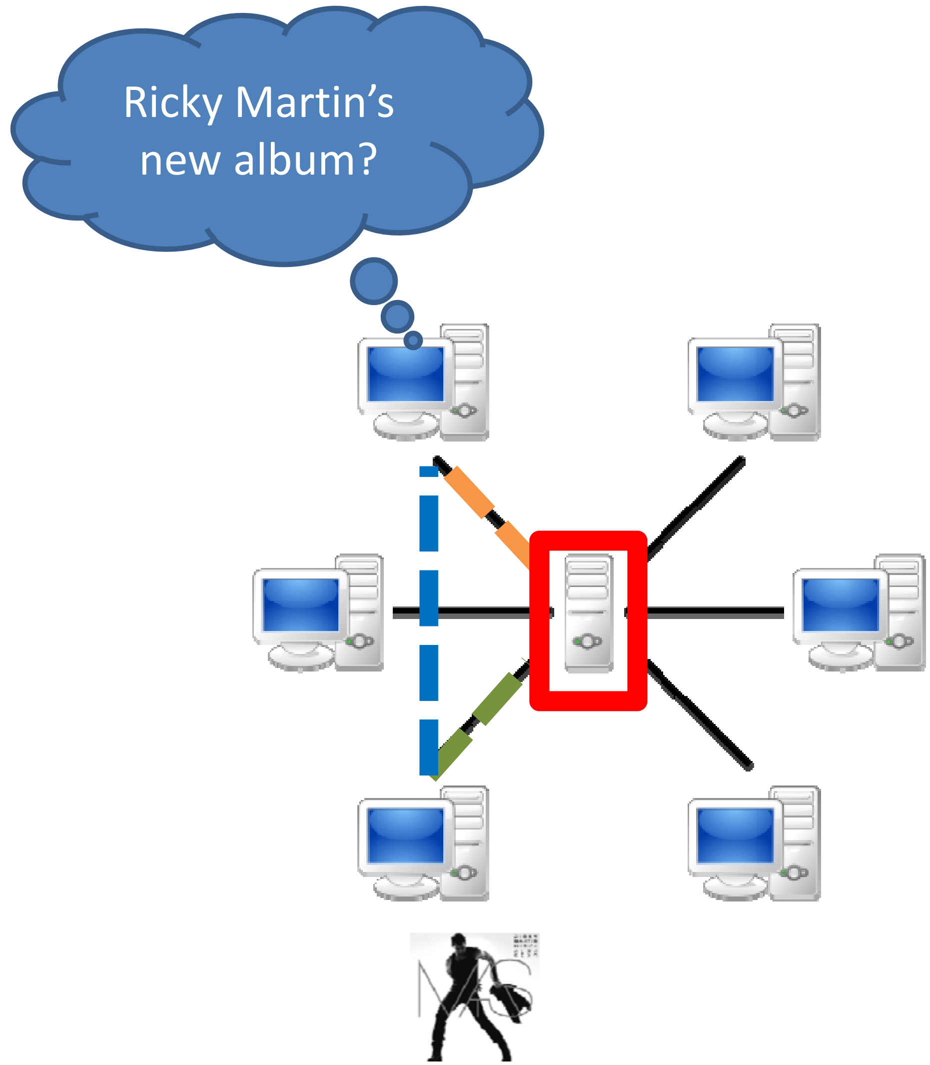


(<u>For example</u>: *Kazaa, Gnutella*)

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

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

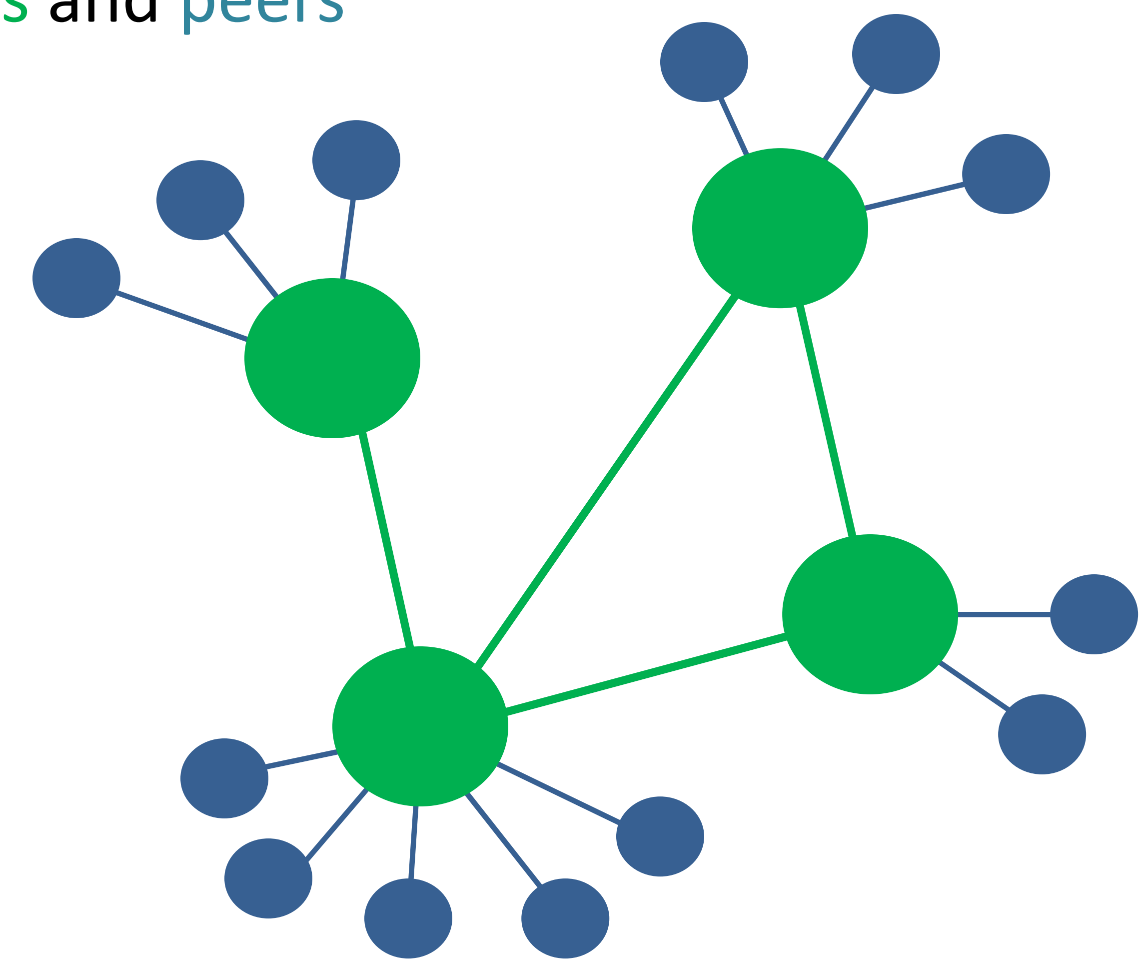
- Central point-of-failure



(For example: *Napster … central directory was shut down*)

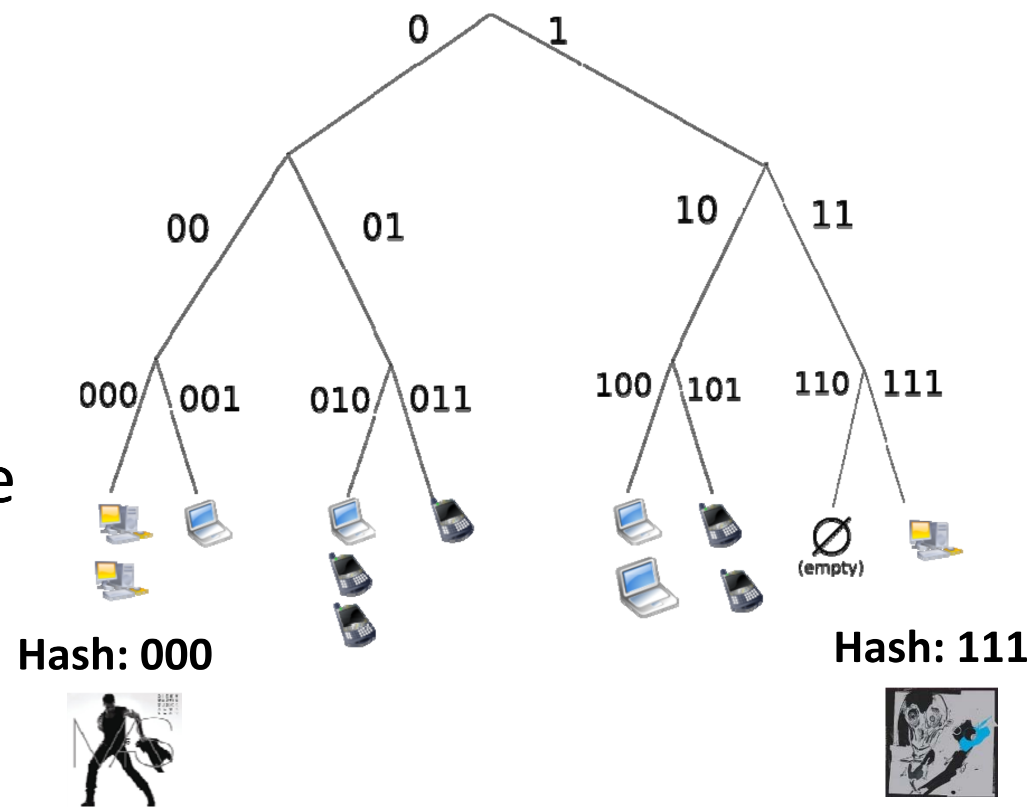
Peer-to-Peer: *Structured* (*Hierarchical*)

- Super-peers and peers



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

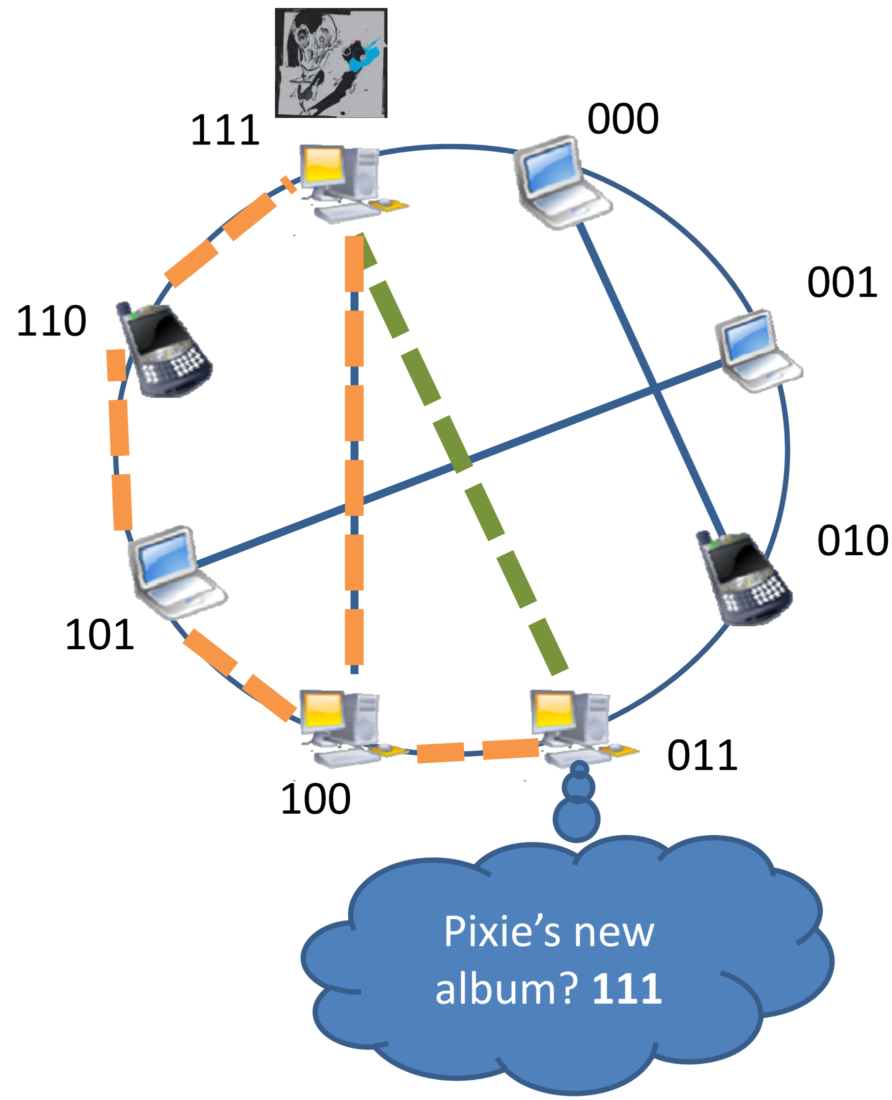
- Distributed Hash Table
- (*key*,*value*) pairs
- *key* based on hash
- Query with *key*
- Insert with (*key*,*value*)
- Peer indexes *key* range



(<u>For example</u>: *Bittorrent's Tracker*)

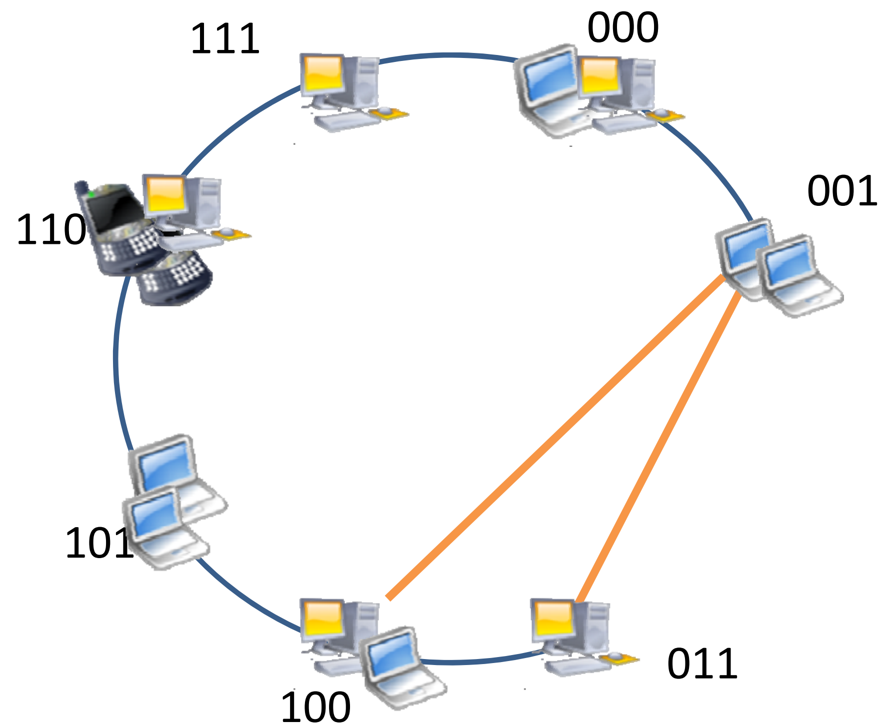
Peer-to-Peer: *Structured (DHT)*

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



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

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



Comparison of P2P Systems

For Peer-to-Peer, what are the benefits of (1) central directory vs. (2) unstructured, vs. (3) structured?

1) Central Directory

- Search follows directory (1 lookup)
- Connections → $O(n)$
- **Central point of failure**
- Peers control their data
- No neighbours

2) Unstructured

- **Search requires flooding (*n* lookups)**
- Connections → $O(n^2)$
- No central point of failure
- Peers control their data
- Peers control neighbours

3) Structured

- Search follows structure (log(n) lookups)
- Connections → $O(n)$
- No central point of failure
- **Peers assigned data**
- Peers assigned neighbours

P2P vs. Client–Server

What are the benefits of Peer-to-Peer vs. Client–Server?

Client–Server

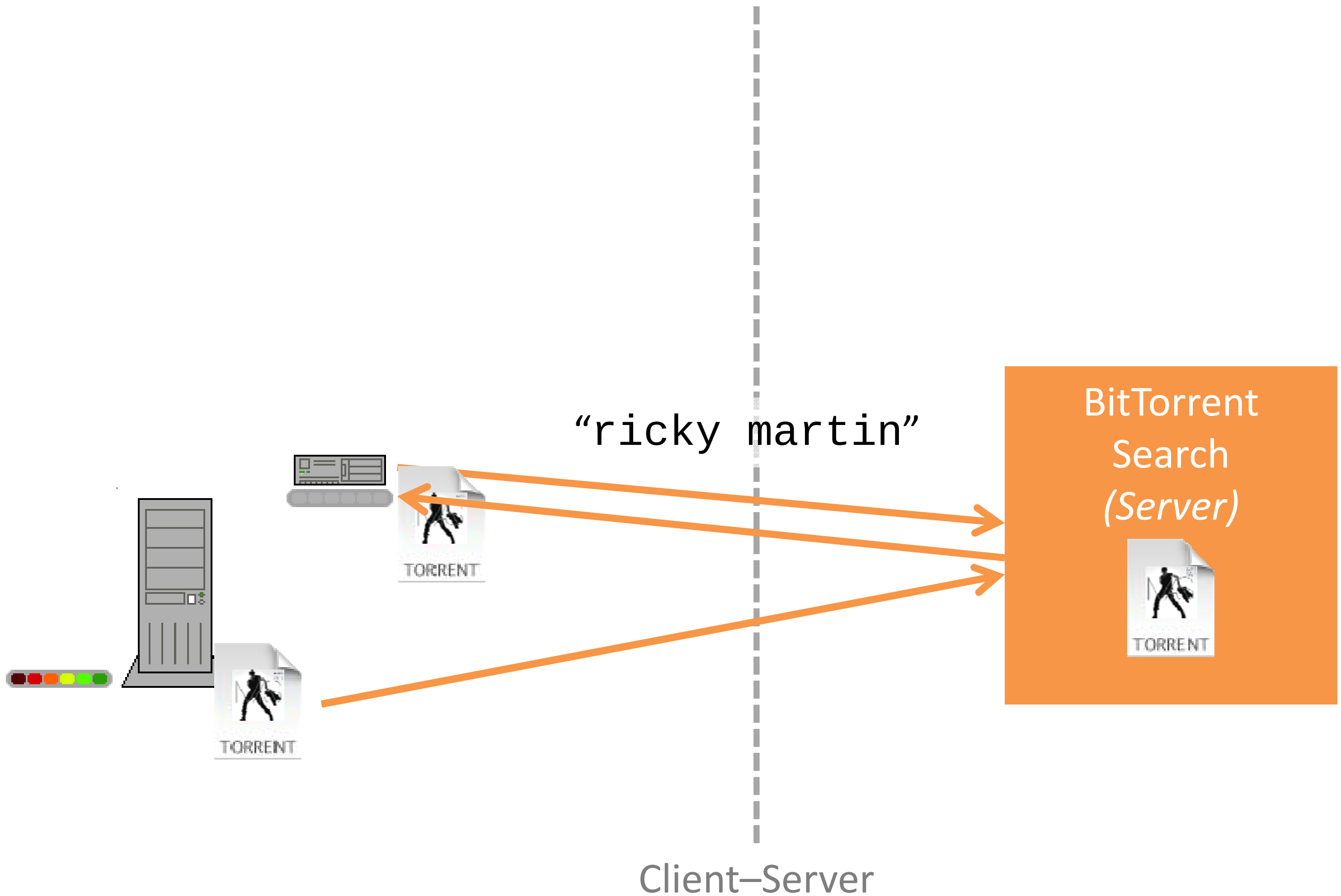
- Data lost in failure/deletes
- Search easier/faster
- Network often faster (to websites on backbones)
- Often central host
  - Data centralised
  - Remote hosts control data
  - Bandwidth centralised
  - Dictatorial
  - Can be taken off-line

Peer-to-Peer

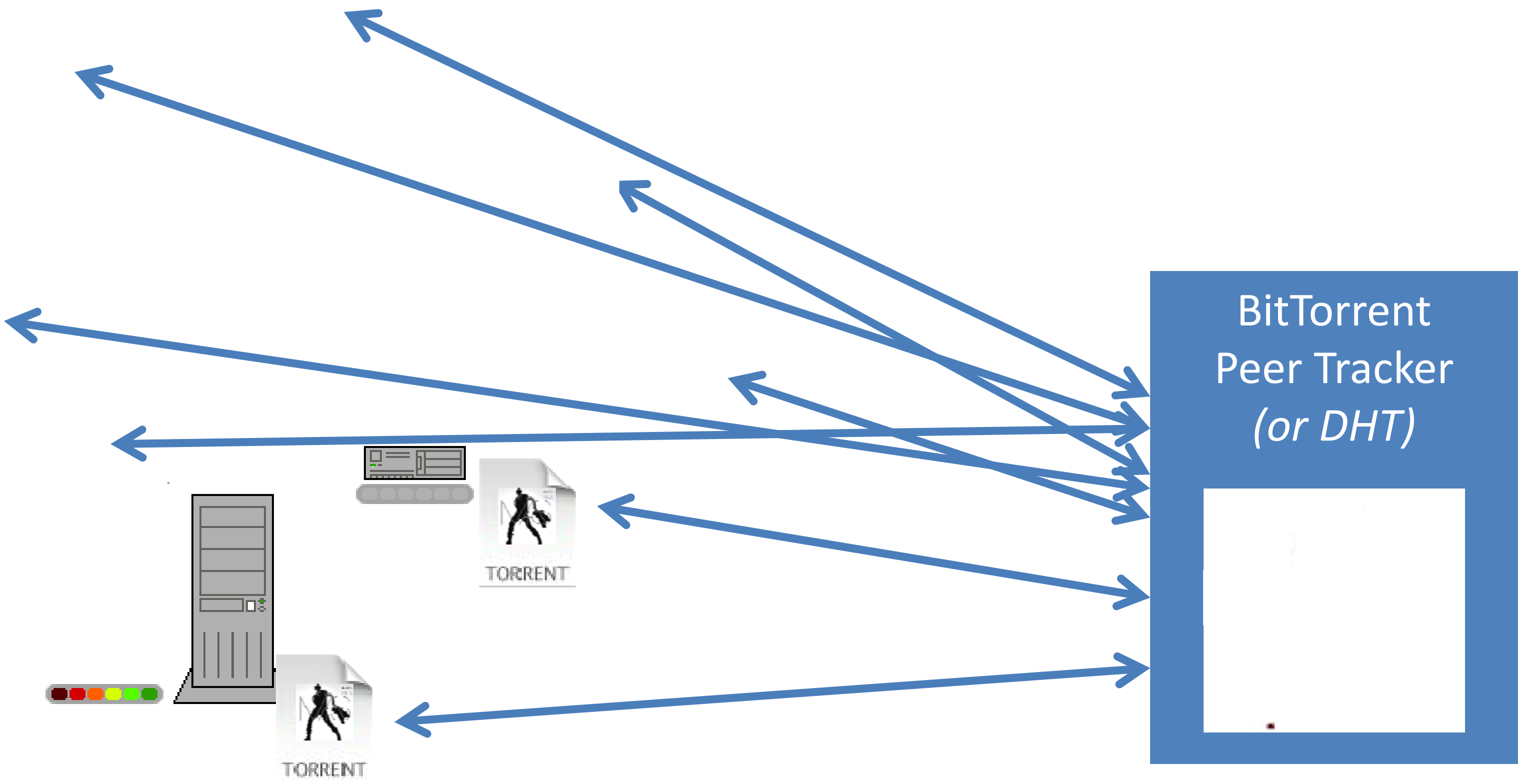
- May lose rare data (churn)
- Search difficult (churn)
- Network often slower (to conventional users)
- Multiple hosts
  - Data decentralised
  - Users (often) control data
  - Bandwidth decentralised
  - Democratic
  - Difficult to take off-line

DISTRIBUTED SYSTEMS: HYBRID EXAMPLE (*BITTORRENT*)

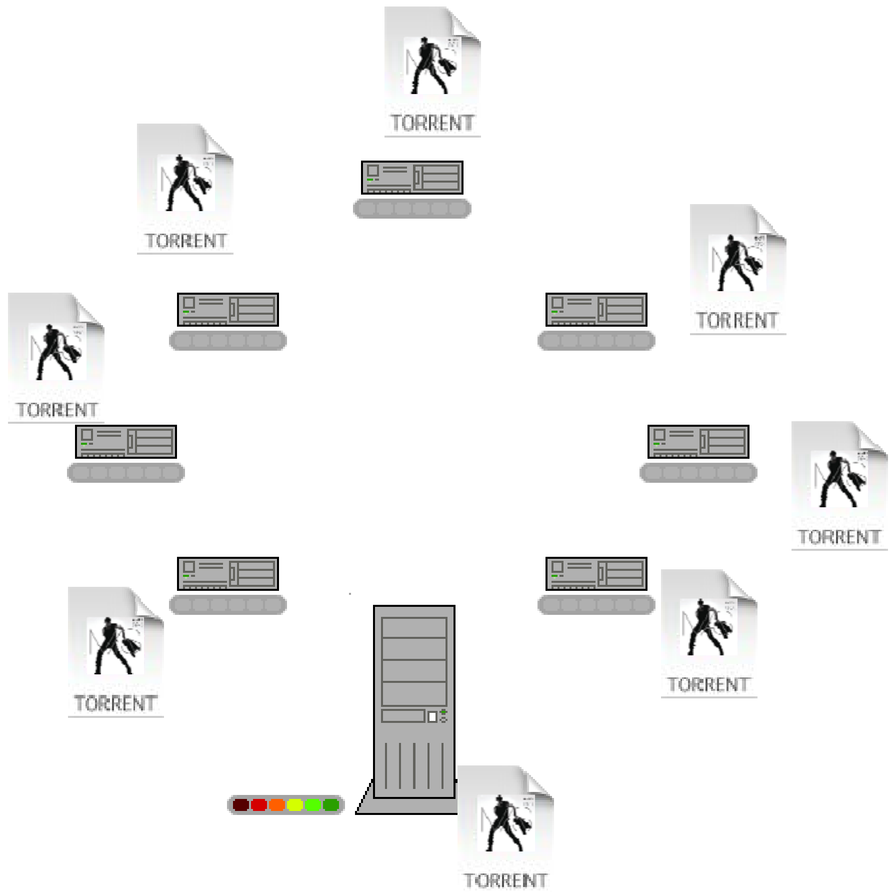
BitTorrent: Search Server



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
(Torrent Search Engines target of law-suits)

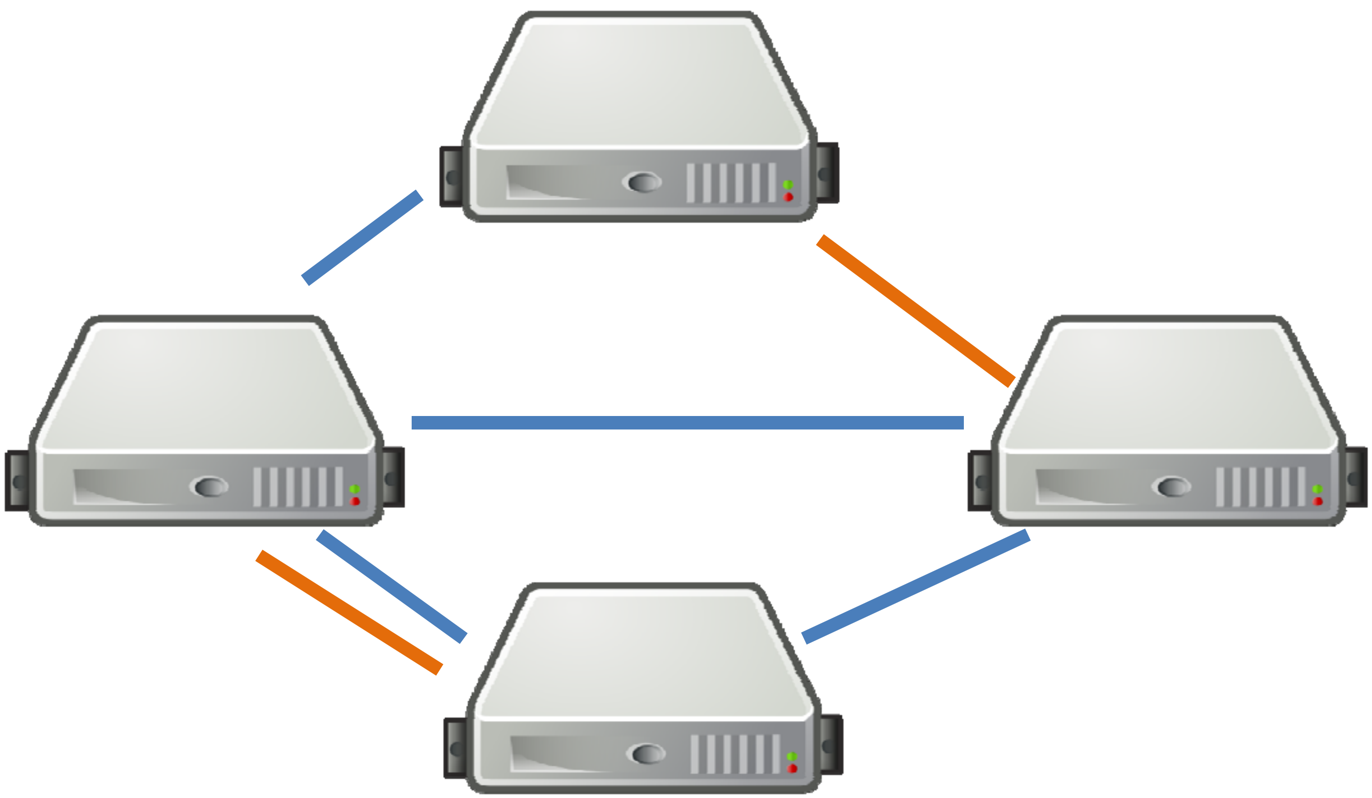
DISTRIBUTED SYSTEMS:
IN THE REAL WORLD

Real-World Architectures: Hybrid

- **Often hybrid!**
  - Architectures herein are simplified/idealised
  - No clear black-and-white (just good software!)
  - For example, *BitTorrent* mixes different paradigms
  - But good to know the paradigms

Physical Location: Cluster Computing

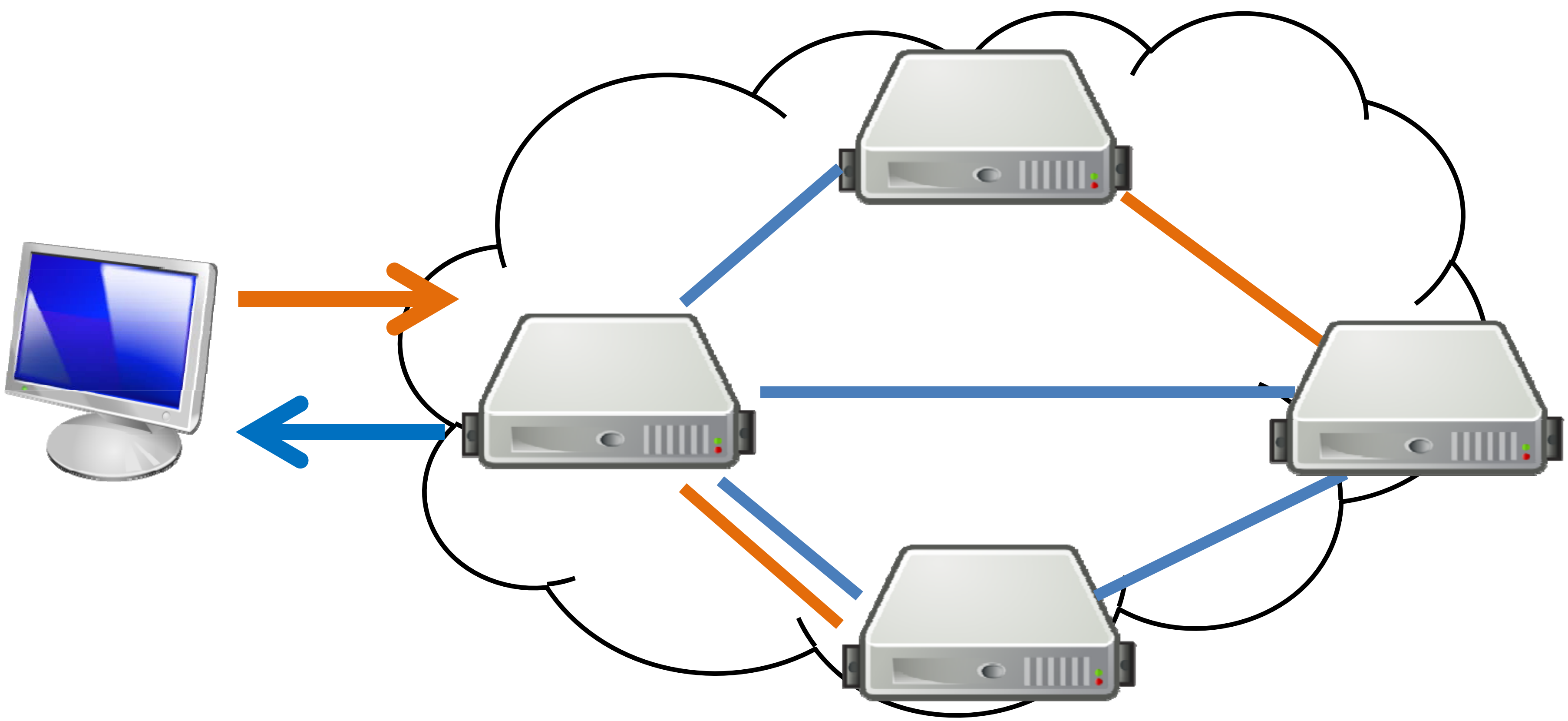
- Machines (typically) in a central, local location; e.g., a local LAN in a server room



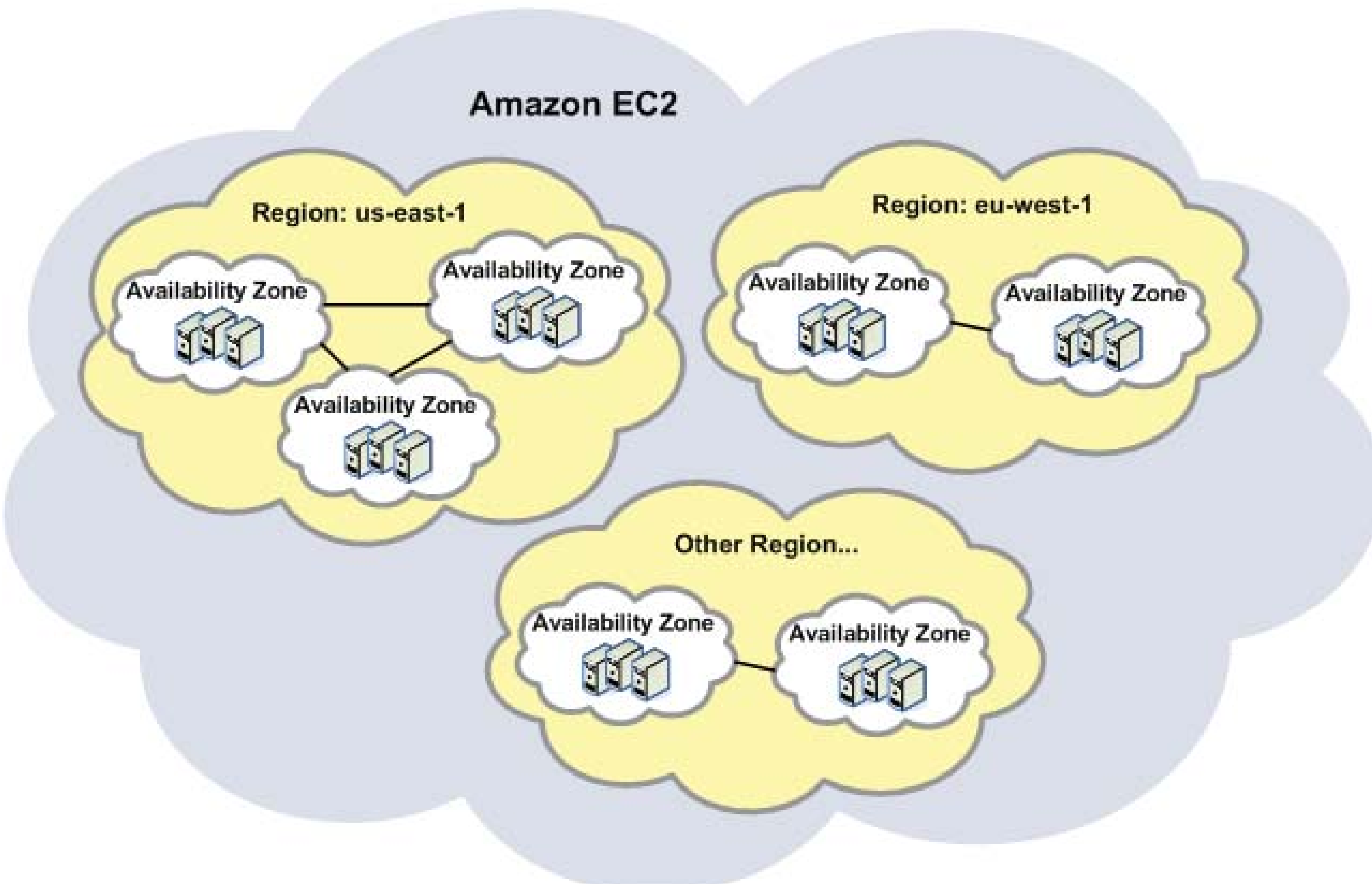
Physical Location: Cluster Computing

Physical Location: Cloud Computing

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

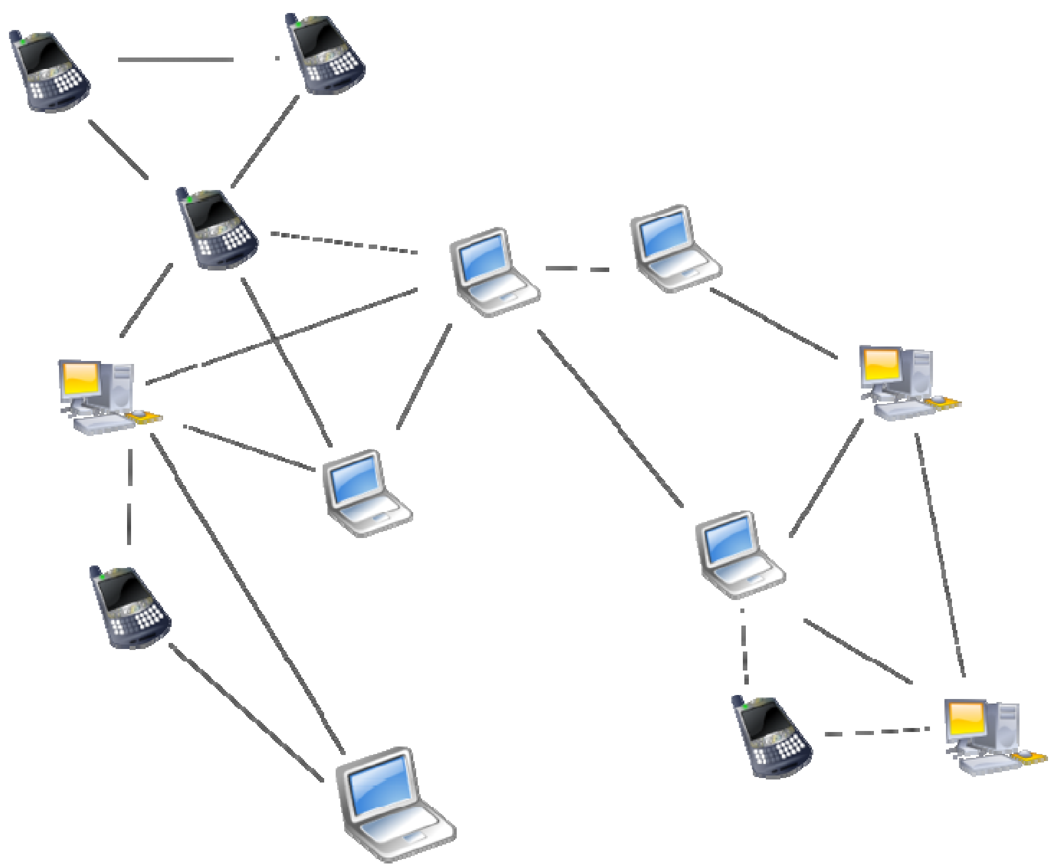


Physical Location: Cloud Computing



Physical Location: Grid Computing

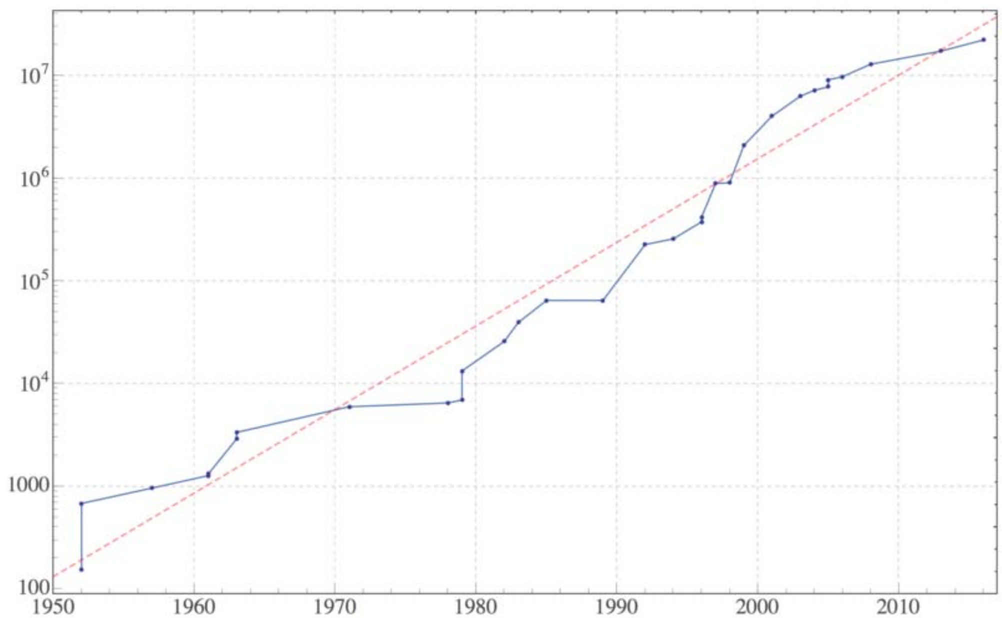
- Machines in diverse locations

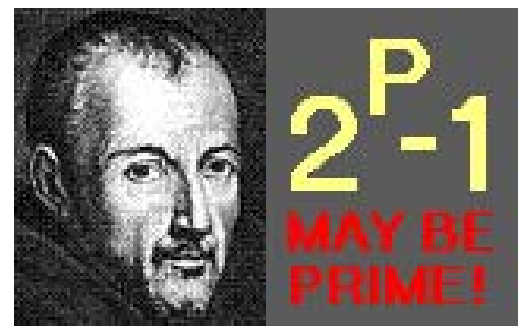


Physical Location: Grid Computing

Physical Location: Grid Computing

$$2^{74,207,281} - 1$$





PrimeGrid

Physical Locations

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

LIMITATIONS OF DISTRIBUTED SYSTEMS: EIGHT FALLACIES

Eight Fallacies

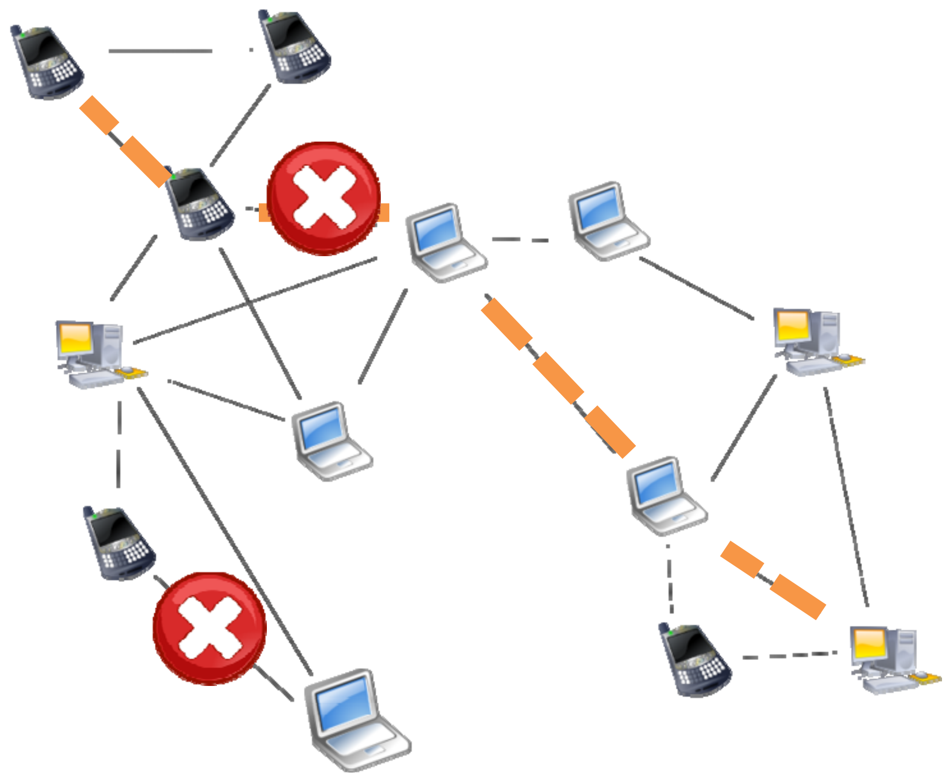
- By L. Peter Deutsch (1994)
  - James Gosling (1997)

"Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences." — L. Peter Deutsch

- Each fallacy is a **false statement**!

1. The network is reliable

Machines fail, connections fail, firewall eats messages

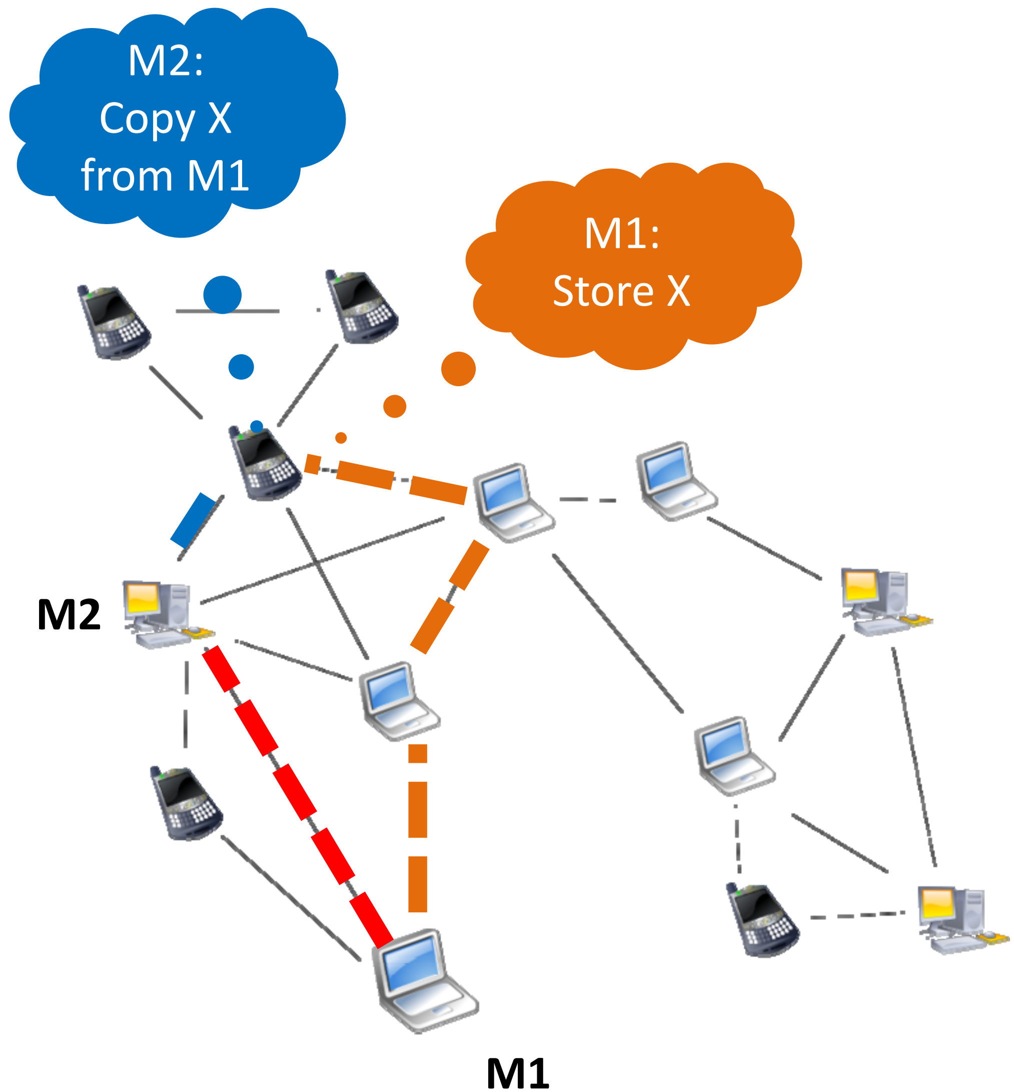


- flexible routing
- retry messages
- acknowledgements!

2. Latency is zero

There are significant communication delays

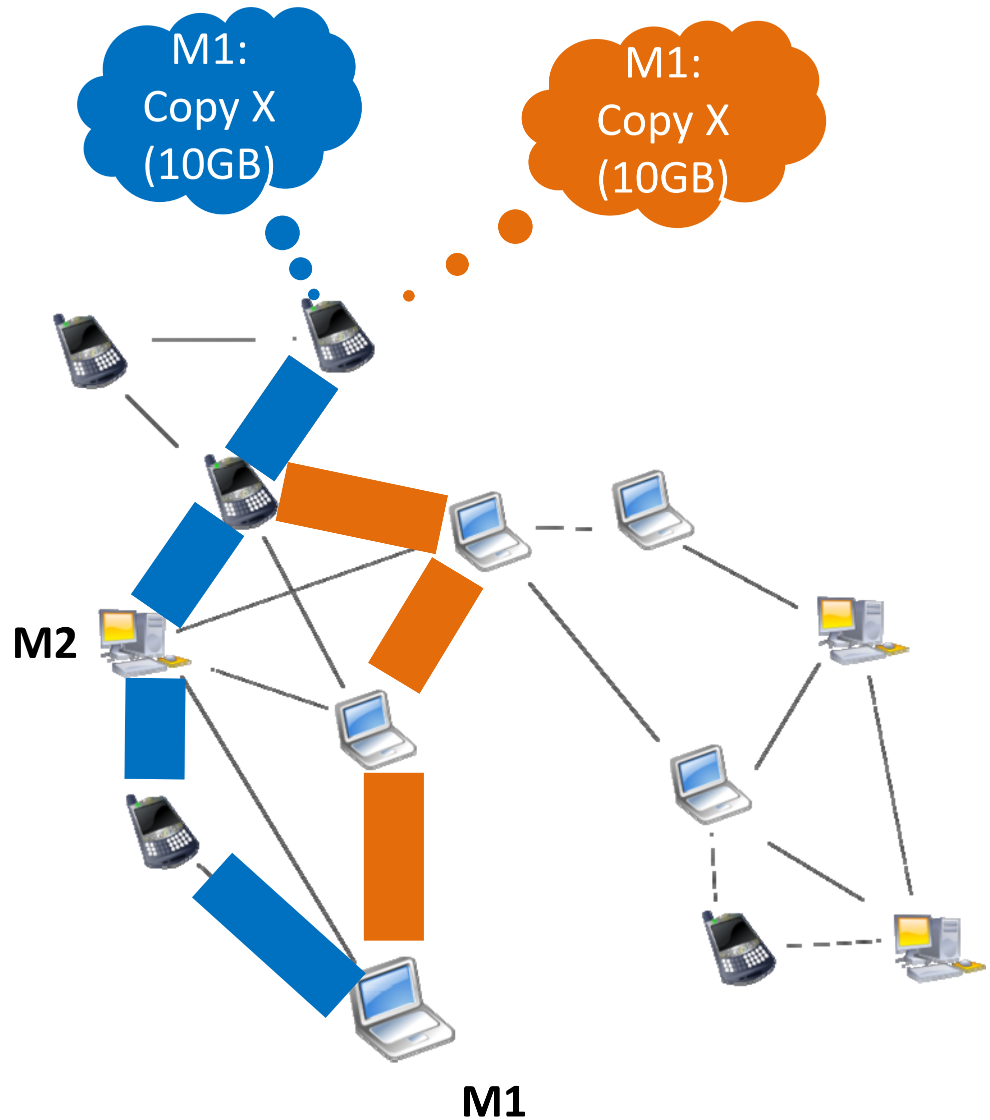
- avoid "races"
- local order ≠ remote order
- acknowledgements
- minimise remote calls
  - batch data!
- avoid waiting
  - multiple-threads



3. Bandwidth is infinite

Limited in amount of data that can be transferred

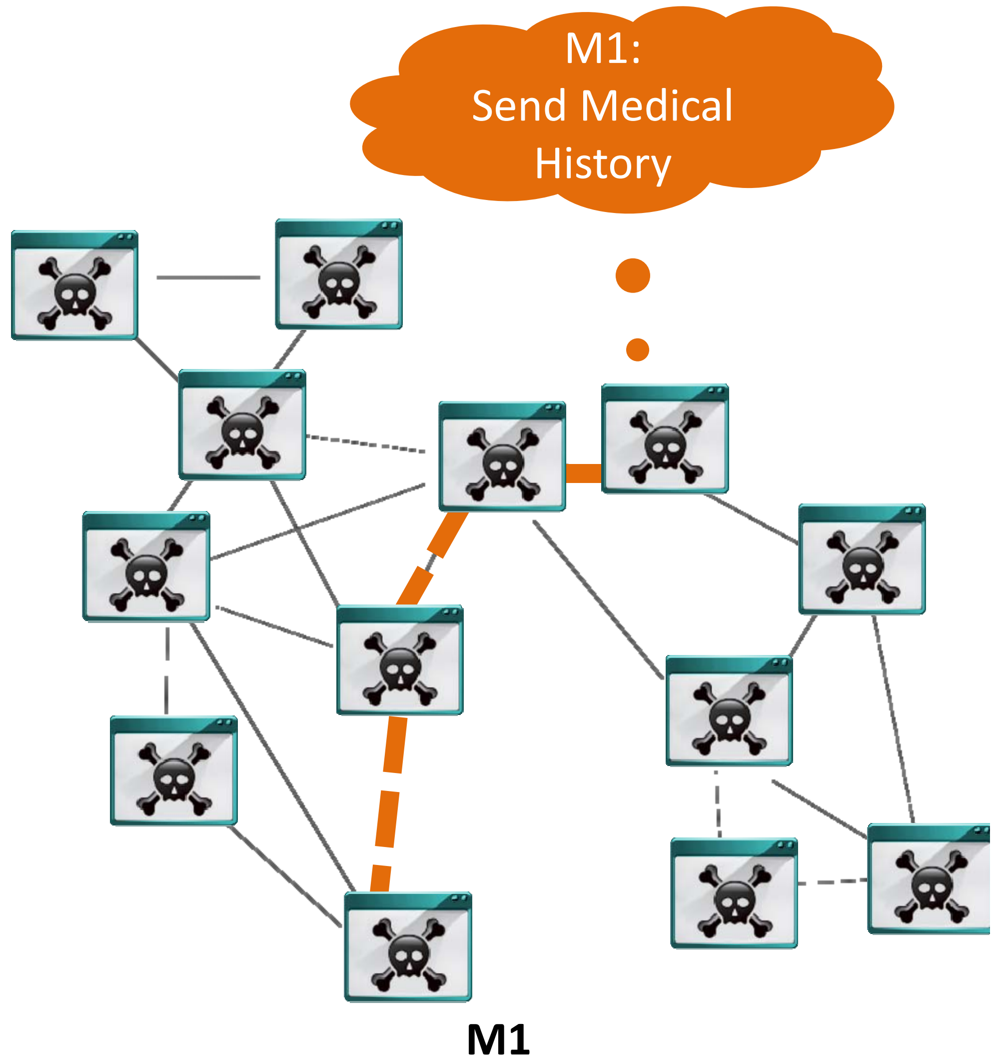
- avoid resending data
- avoid bottlenecks
- direct connections
- caching!!



4. The network is secure

Network is vulnerable to hackers, eavesdropping, viruses, etc.

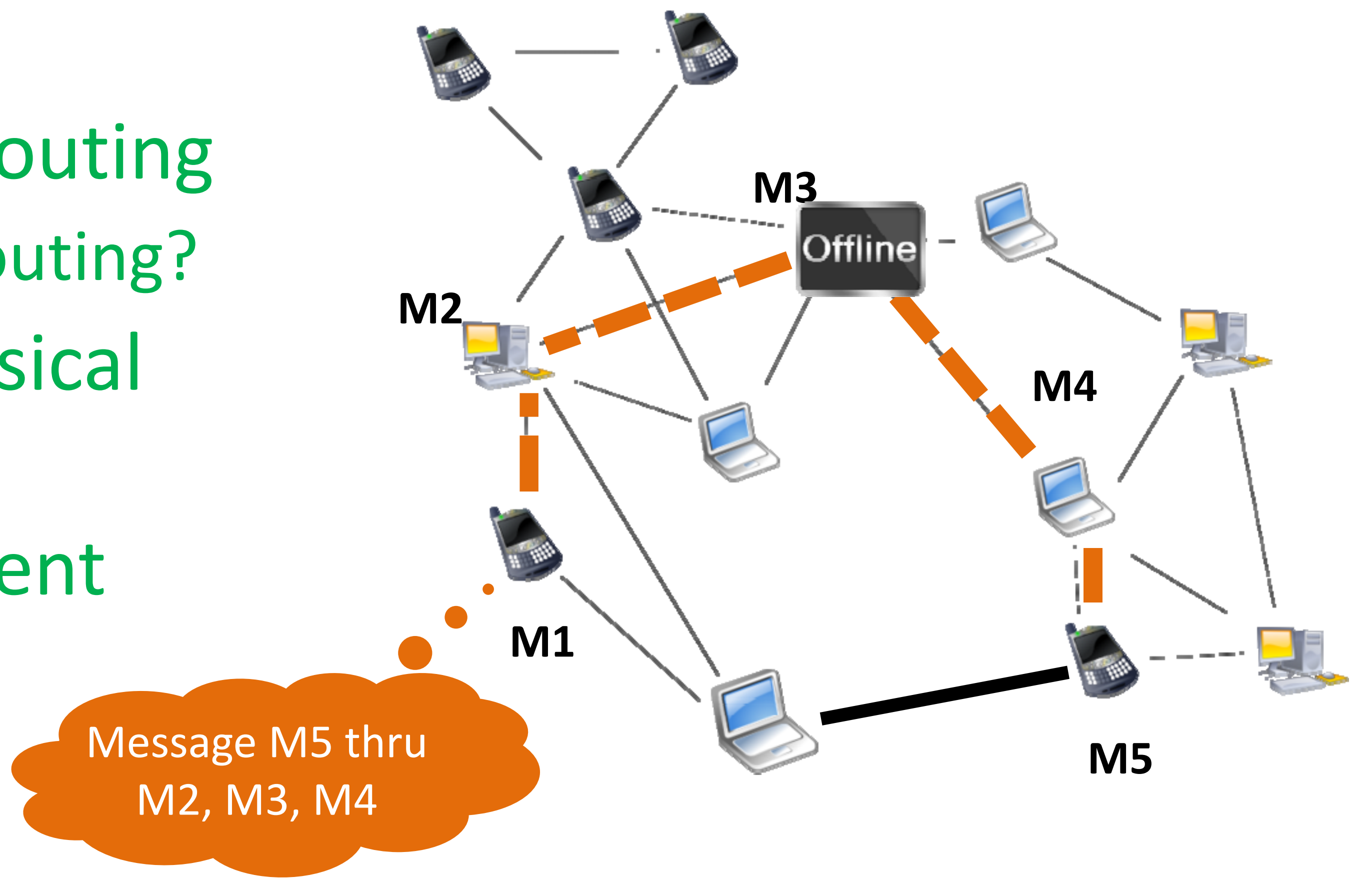
- send sensitive data directly
- isolate hacked nodes
  - hack one node ≠ hack all nodes
- authenticate messages
- secure connections



5. Topology doesn't change

How machines are physically connected may change ("churn")!

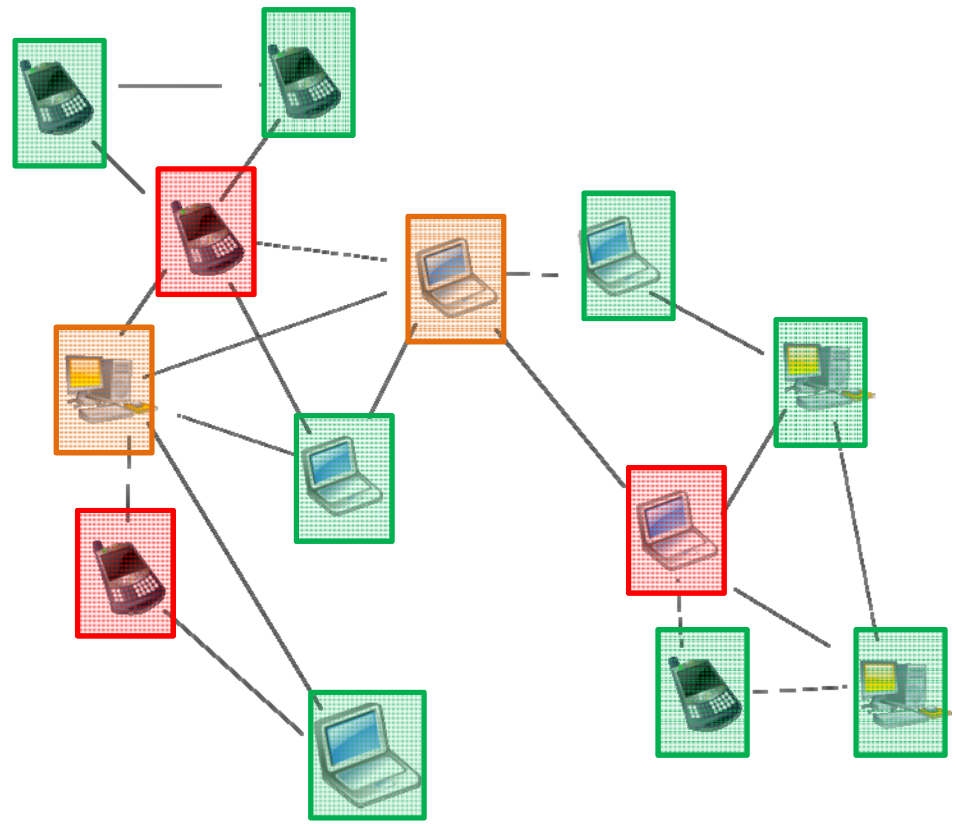
- avoid fixed routing
  - next-hop routing?
- abstract physical addresses
- flexible content structure



6. There is one administrator

Different machines
have different policies!

- Beware of firewalls!
- Don’t assume most recent version
  - Backwards compat.

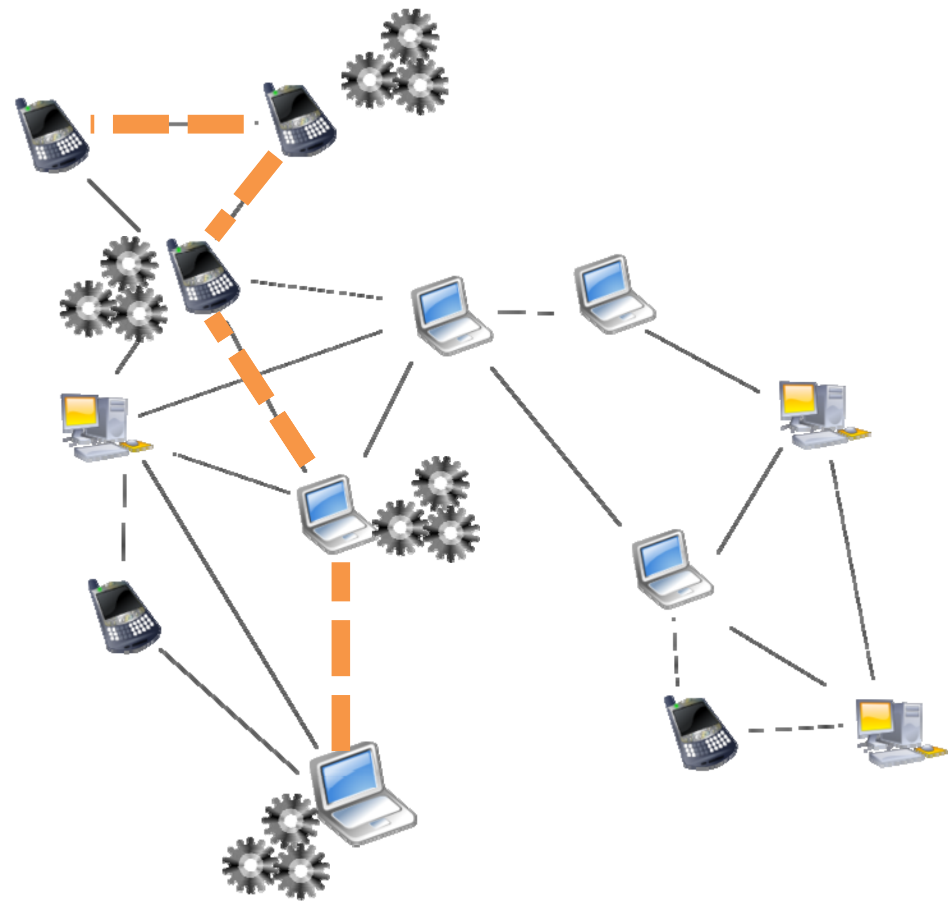


7. Transport cost is zero

It costs time/money to transport data: not just bandwidth

(*Again*)

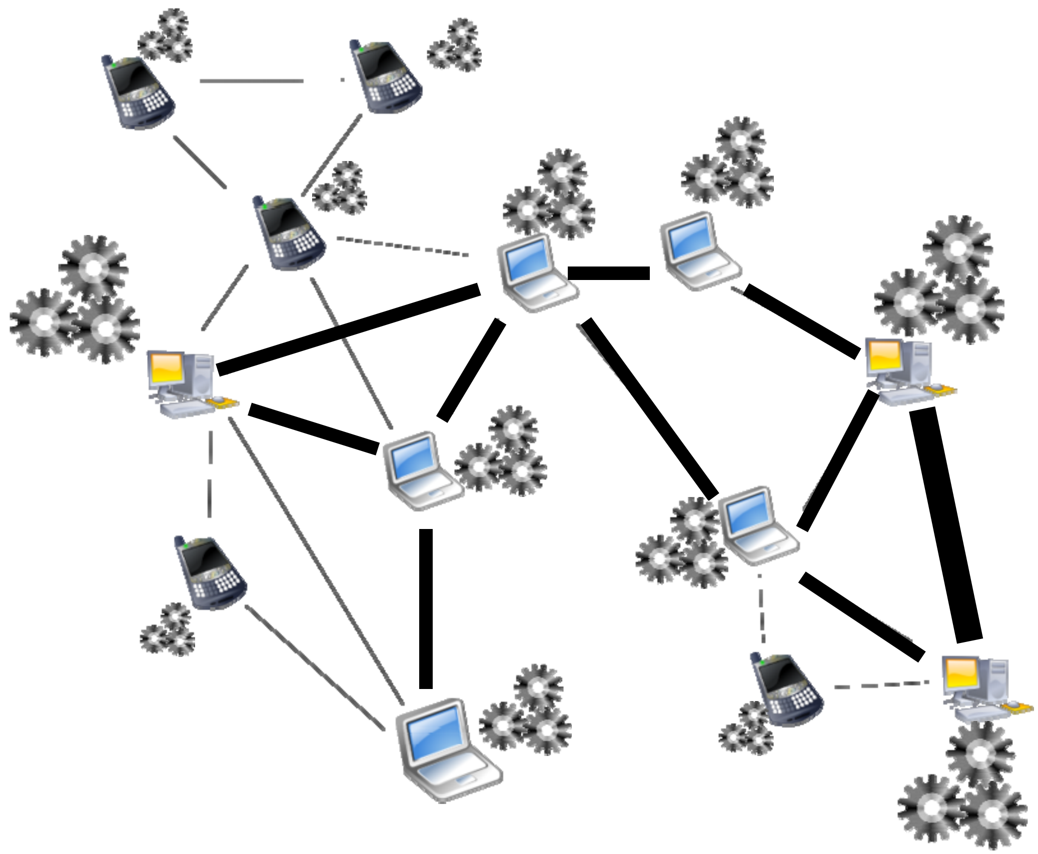
- minimise redundant data transfer
  - avoid shuffling data
  - caching
- direct connection
- compression?



8. The network is homogeneous

Devices and connections are not uniform

- interoperability!
- route for speed
  - not hops
- load-balancing



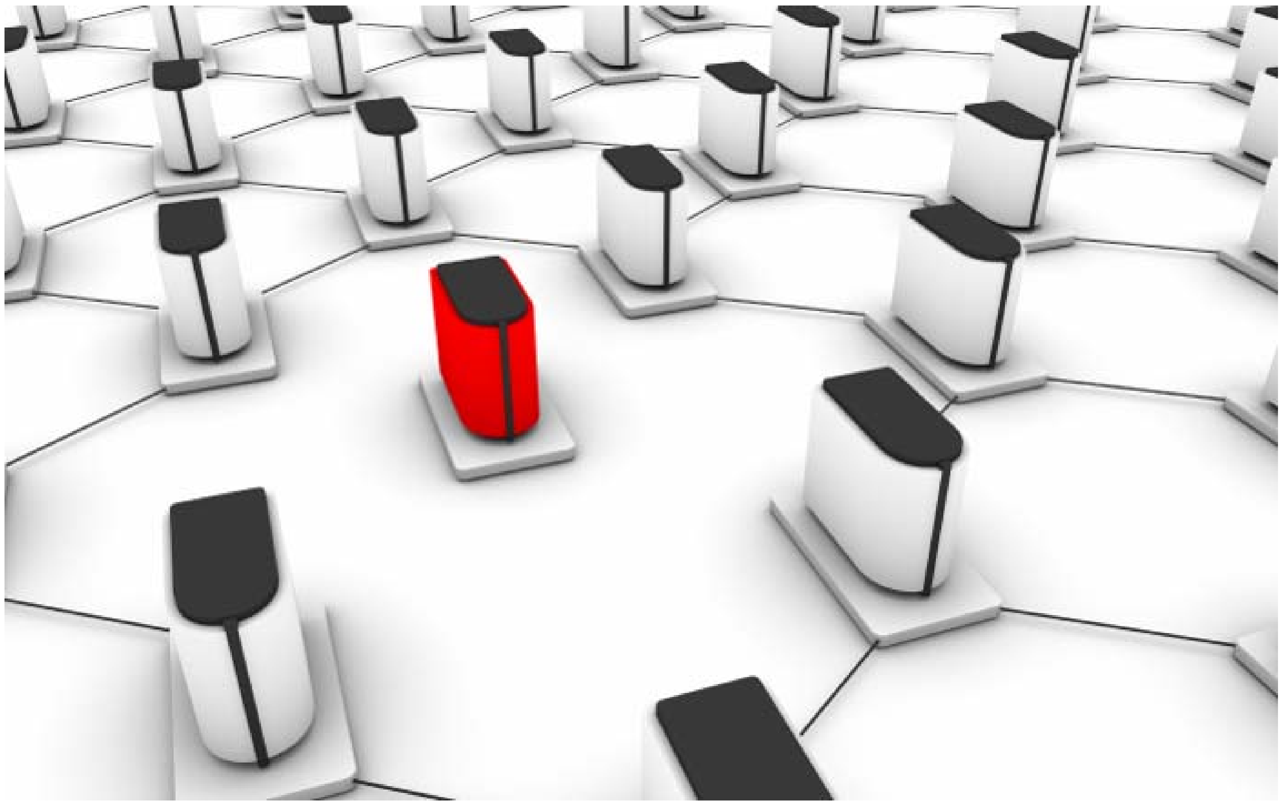
Eight Fallacies (to avoid)

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

Discussed later: Fault Tolerance



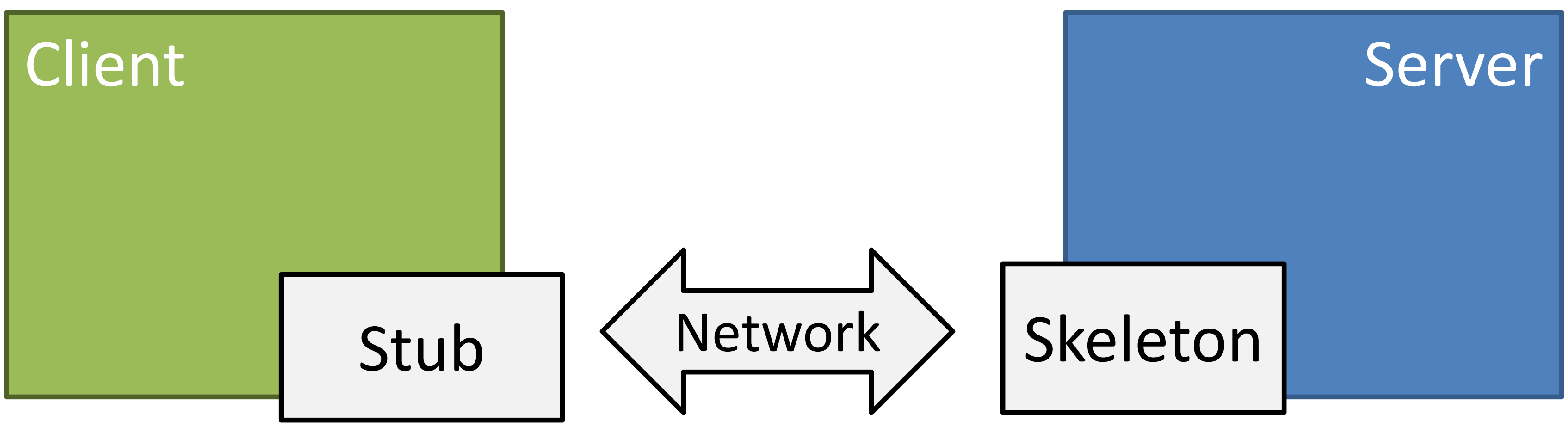
LAB II PREVIEW:
JAVA RMI OVERVIEW

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
- Predecessor of CORBA (in Java)
- Stub / Skeleton model (TCP/IP)



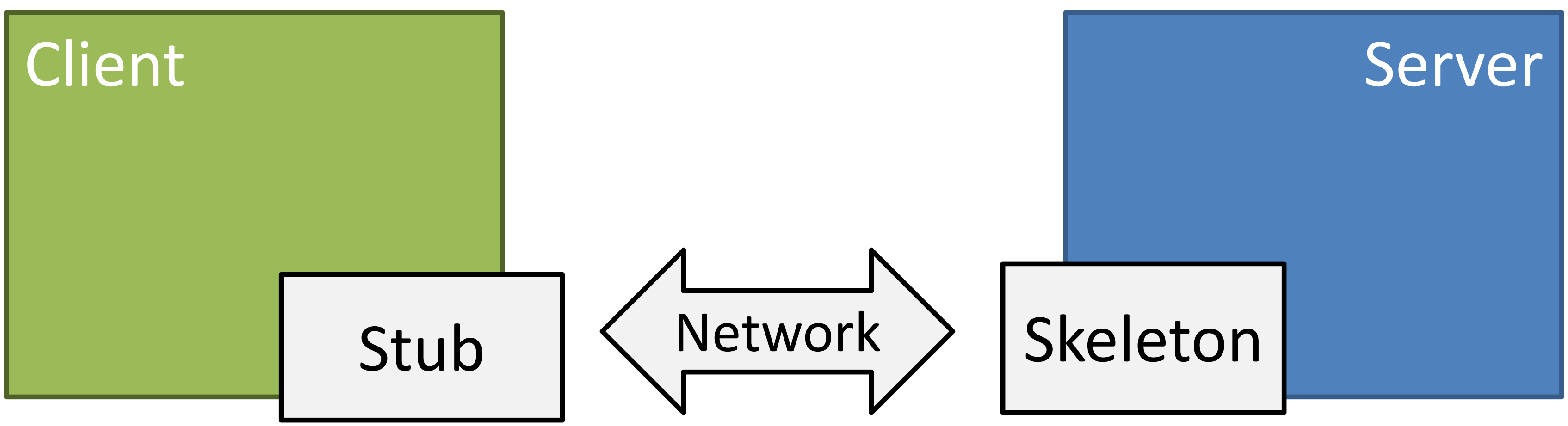
What is Java RMI?

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);
    }
}
```

Problem?

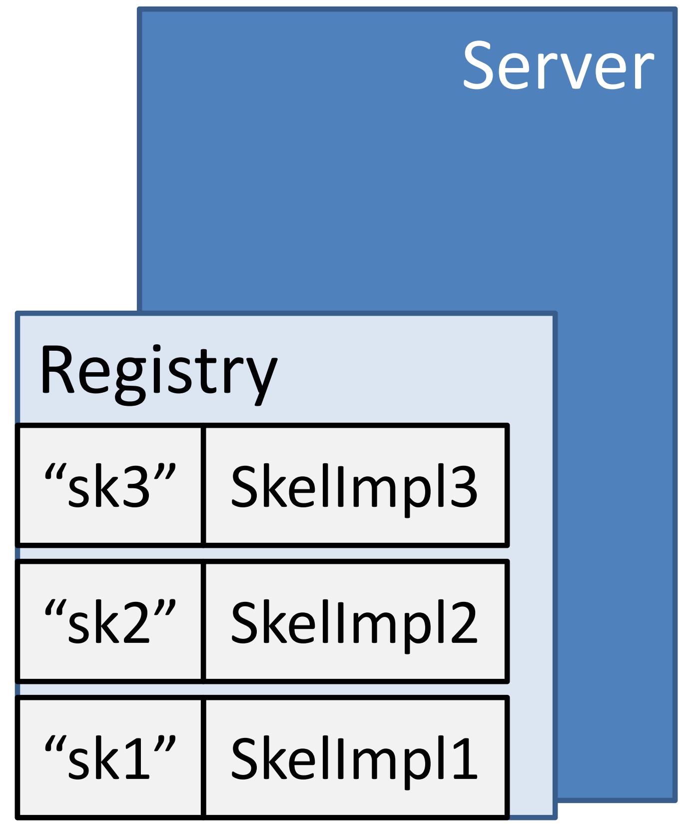
Synchronisation: (e.g., should use ConcurrentHashMap)

[Thanks to Tomas Vera ☺]

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

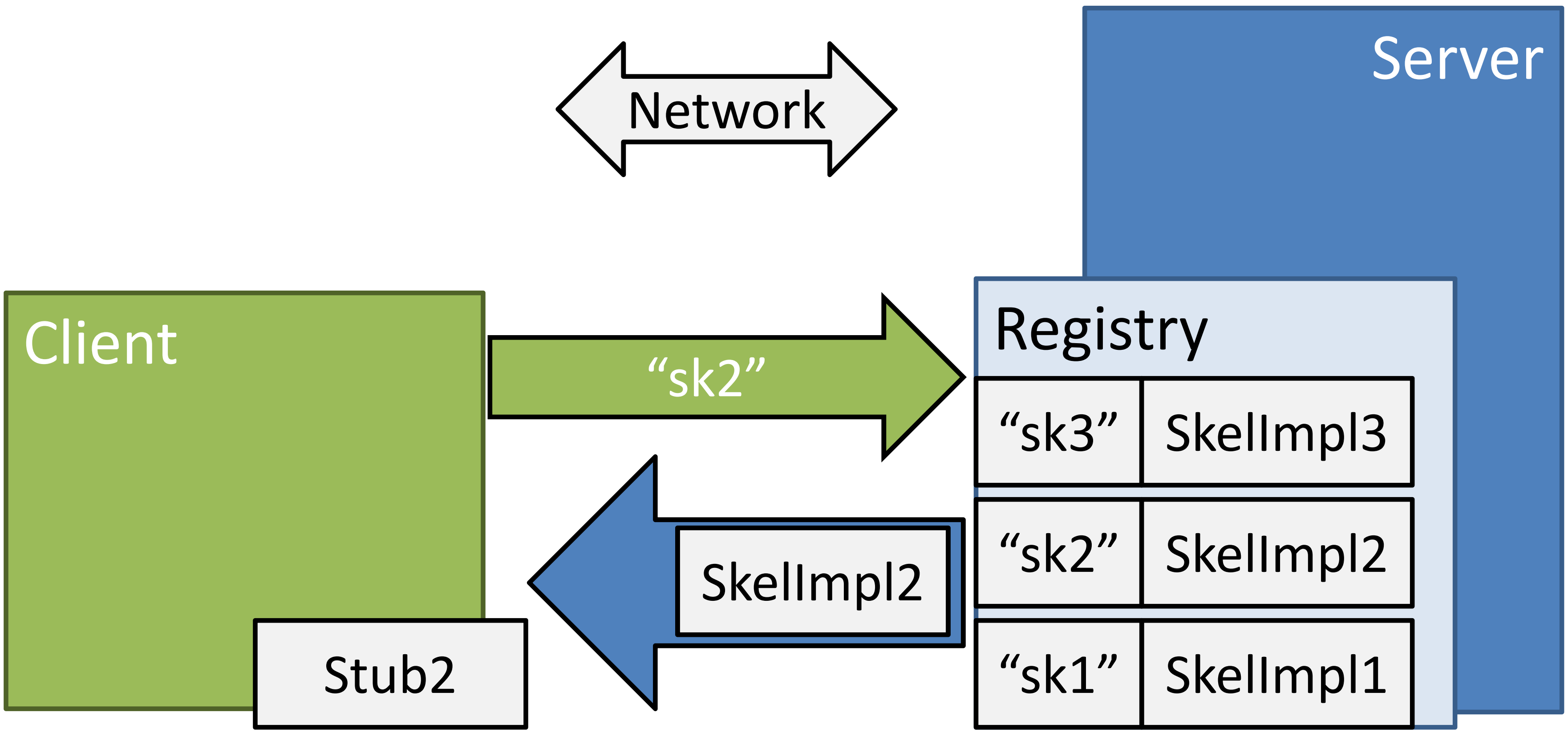
Server Registers Skeleton Implementation As a Stub

```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



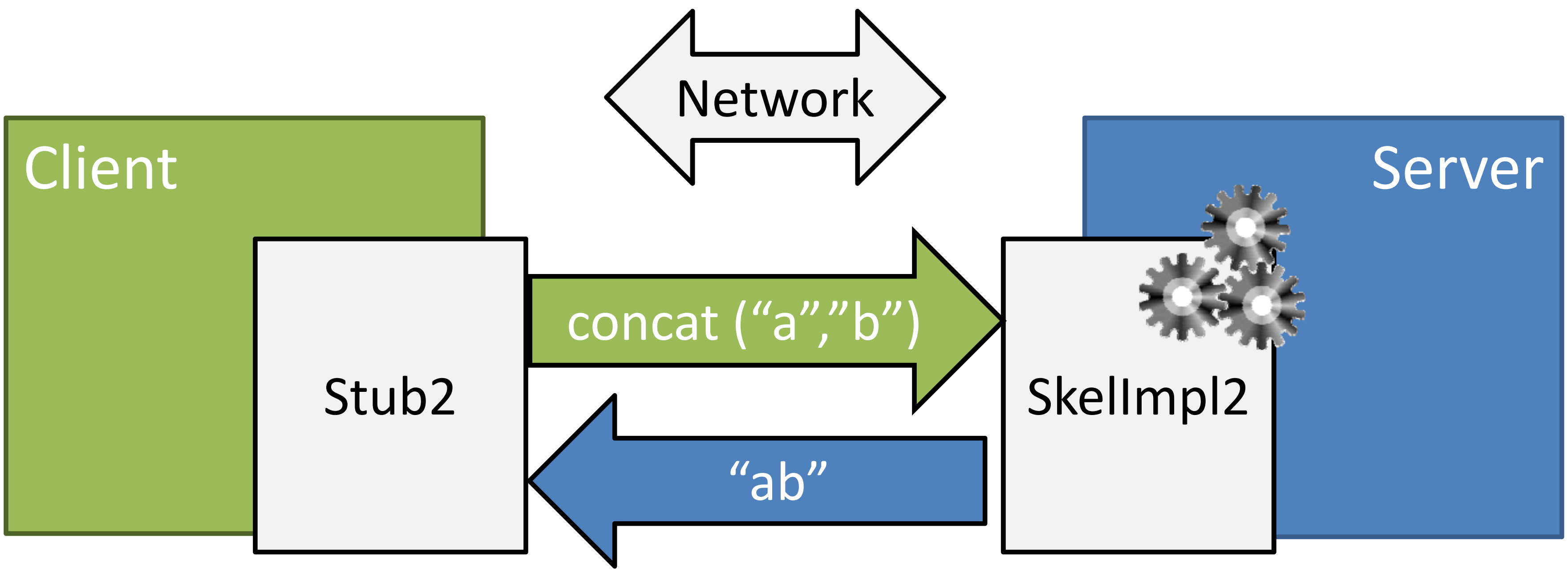
Client Connecting to Registry

```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`

RECAP

Topics Covered (Lab)

- External Merge Sorting
  - When it doesn't fit in memory, use the disk!
  - Split data into batches
  - Sort batches in memory
  - Write batches to disk
  - Merge sorted batches into final output

Topics Covered

- What is a (good) Distributed System?
- Client–Server model
  - Fat/thin client
  - Mirror/proxy servers
  - Three-tier
- Peer-to-Peer (P2P) model
  - Central directory
  - Unstructured
  - Structured (Hierarchical/DHT)
  - BitTorrent

Topics Covered

- Physical locations:
  - Cluster (local, centralised) vs.
  - Cloud (remote, centralised) vs.
  - Grid (remote, decentralised)
- 8 fallacies
  - Network isn't reliable
  - Latency is not zero
  - Bandwidth not infinite,
  - etc.

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?