Objectives

- Resume discussion of models
  - Talk a bit more about architectural models
  - Talk about fundamental models
- Begin our discussion of network technologies
  - Today: a birds-eye view of the issues and general concepts.
  - Next week: sockets and RMI to give you exposure to specific network technologies.
  - After next week, you will be given your first programming assignment for hands on work with RMI.

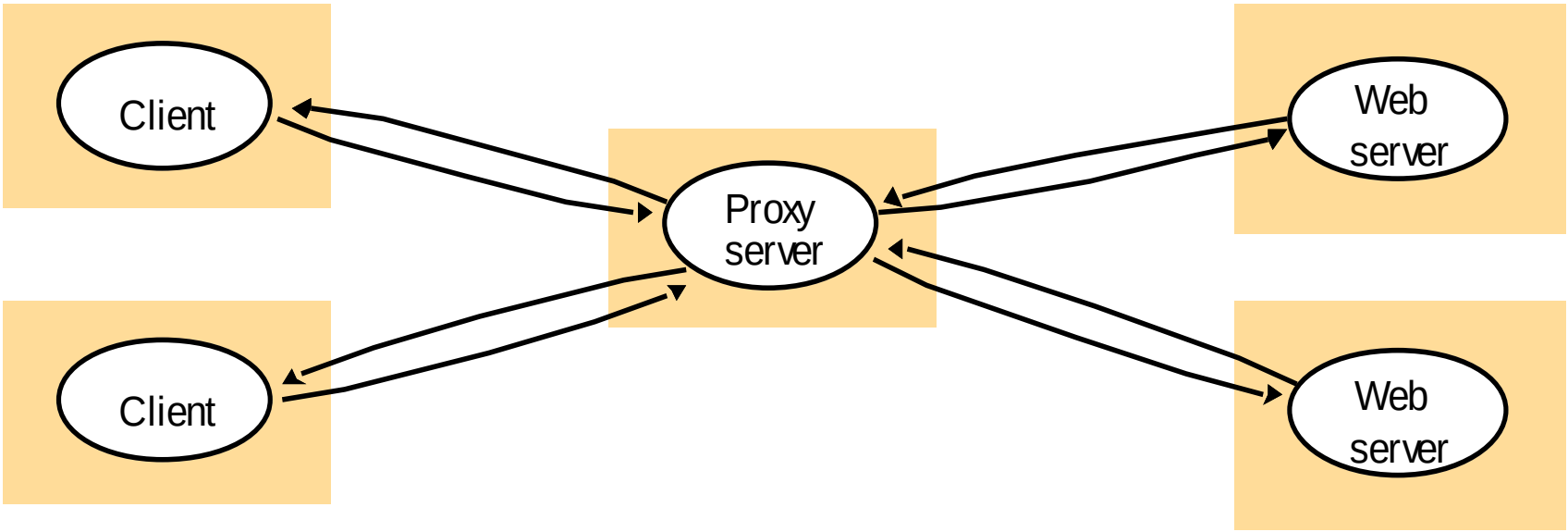
Architectural model

- Recall:
  - The architectural model is concerned with:
    - What are the components of the system?
    - What are their roles?
    - Where are they located?
  - From this, we can break distributed systems up into some coarse and familiar classes:
    - Client/server
    - Peer-to-peer
    - Multiple servers
    - Proxy servers
    - Etc…

Proxy Servers and Caches

- Proxy servers are used to increase availability and performance of services by reducing the load on the network and servers.
  - Separation of service functionality.
  - Sharing of proxy server among clients.
- A cache stores recently used data objects closer to users of the objects than the actual objects themselves.
  - Cache is checked first when data requested.
  - If present in cache, data provided from there.
  - Otherwise, fetched from actual server.
  - Cache must provide facility to check if page is up-to-date, especially if server is unaware of presence of cache.

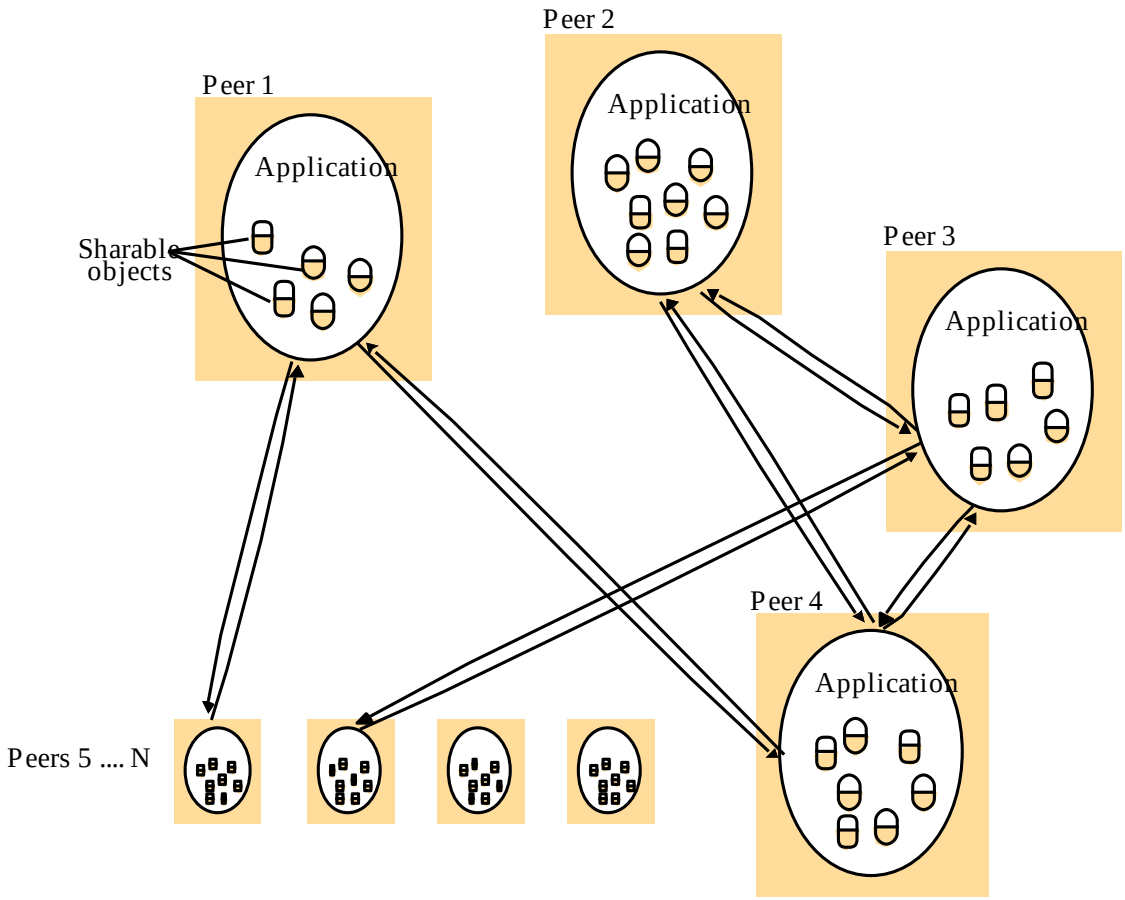
Example: Web Proxy Server



Peer-to-peer

- Unlike client/server, peer-to-peer is based on a set of processes with equal status in the distributed system who cooperatively perform some sequence of operations to achieve a goal.
  - No distinction between clients and servers.
- Code in peer processes maintains consistency of application-level resources and synchronizes application-level actions when necessary.
- No dedicated machines.
  - This may not be strictly true, especially considering popular P2P services like BitTorrent with trackers. These aid in finding peers initially though, and do not necessarily participate directly in the P2P activity.
- *Examples*: BitTorrent, Gnutella, “Bonjour” chat programs, file sharing.

Peer-based distributed application



Process Peer Model

- What about idle or lightly loaded workstations?
- Sharing of computing resources
  - Either let idle machine sit idle
  - Or run useful jobs on unused computing resources
- Alternatively
  - Treat machines as collection of CPUs and memory
  - Assign processes to run on resource on demand
  - Users won't need heavy duty workstations locally
    - GUI locally
    - Remote machine for heavy processing
  - Computational model of Plan9

Thin clients

- Systems that run code remotely but provide user interactivity locally.
- X-terminals were a form of this. The thin client ran an X server, and displayed programs running remotely.
- We see this today in systems like remote desktop services such as VNC.
  - Examples: Apple Remote Desktop, “GotoMyPC.com”.
  - Today we see systems where the web browser is the thin client, with the app running elsewhere but displayed in an applet or service running in the browser.

Grid computing

- Addresses the issue of making costly resources available to a wide user base.
- Provide users seamless access to:
  - Storage capacity
  - Compute capacity
  - Network bandwidth between storage and computing
- Growing in popularity for scientific computing.
- On demand resource allocation
- Adaptive to variations in load and reliability

Cloud computing

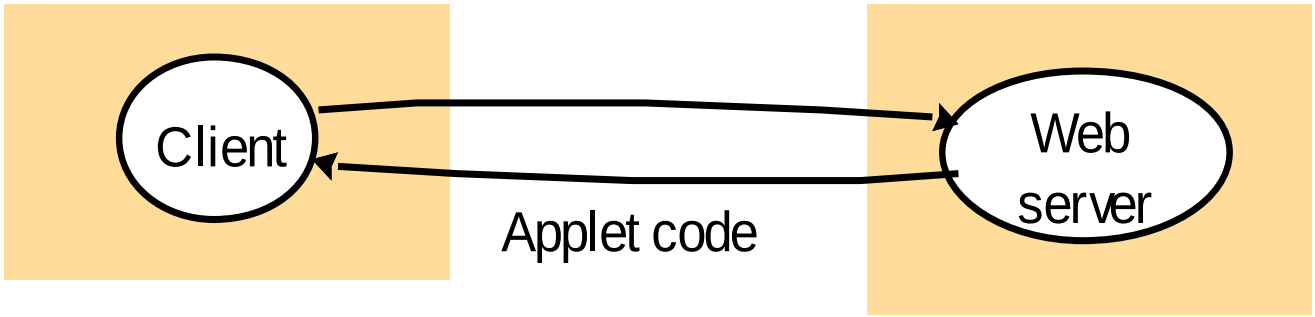
- Transparent access for users to compute and storage resources for a metered cost.
  - Virtualization: Multiplex hardware resources transparently to multiple users.
  - Reliability: Through redundancy on cloud provider end.
    - (Outages may occur though)
  - Performance can be tuned dynamically.
  - Provider can add resources to the cloud to scale it.
  - Transparent access via network technology.
- Primarily driven by economic factors. You buy the cycles/space that you need, and the leftovers are used by others instead of wasted. Your cloud usage can grow as your load changes.
- *Example*: Amazon cloud services.

Variations on Client/Server Model

- Mobile code moves between computer systems.
- Applets are a well-known and widely used example.
  - Example: Java applets, Flash apps.
- Helps to achieve better performance
  - Eliminates some delays and variability due to network communication
- Accessing services means running code that can invoke their operations
  - Pull model: client initiates
  - Push model: server initiates

Web Applets

a) client request results in the downloading of applet code



b) client interacts with the applet

Multi-Tier Client/Server Architectures

- Two-tier architecture
  - Common from 1980s to 1990s
  - UI on user desktop
  - Application services on server
    - Example: old text-based business applications. Does anyone remember those old telnet-based apps to login to a VMS system with text menus and interface?
  - Performance deteriorates with large user communities
    - Server can get overloaded managing connections.
    - Legacy services may end up running in environments poorly adapted for networking.
    - Databases are performance hogs.

Multi-Tier Client/Server Architectures

- Three-tiered architectures address issues with two-tier.
- Client
  - User interface
  - Some data validation and formatting
- Middle tier
  - Queuing and scheduling of user requests
  - Transaction processing
  - Connection management
  - Format conversion
- Backend server
  - Database
  - Legacy application processing/interface

Architectures

- We have briefly seen the major, common system architectures that you will encounter in practice.
- So, when one is designing a distributed system to address some problem or provide a service, how do we determine which architectural model or pattern is best?
- Look at the sorts of requirements that will drive the design.

Design requirements

- Performance
  - Responsiveness (especially for interactive services)
  - Throughput
  - Load balance
- Quality of service
  - Reliability
  - Security
  - Performance
  - Adaptability to changes in resource availability.
- Caching and replication
- Dependability
  - Fault tolerance
  - Security

Design requirements

- Looking at the requirements of a specific problem, you can see how some architectures are more appropriate than others.
- Example case: Quality of service is required for serving up static web pages. They will not change frequently, and users want almost instant responses.
  - Infrequent changes: Replication is easy, as is caching in proxies closer to users.
  - Quality of service: Need fast response, so we want information close to users. Proxies again.
  - So, sounds like a proxy server mirroring one or more servers is the right choice.

Fundamental models

- Make assumptions about the system explicit
- Make generalizations about what is possible given these assumptions.
- These generalizations can be in the form of
  - Algorithms
  - Formal proofs
  - Descriptions in a formal logical framework
- The generalizations are useful because they facilitate formal analysis to draw conclusions about:
  - Correctness
  - Performance
  - Security

Fundamental models

- We wish to capture the following aspects of distributed systems in fundamental models:
  - Interactions in the presence of delays that occur in the real world.
  - Failures
  - Security

Interaction model

- How a set of processes communicate to achieve some task.
- Similar to the model of the sequence of algorithmic steps to accomplish something.
- In distributed systems, we call these distributed algorithms. They are composed of:
  - The steps taken by each participant.
  - The messages transmitted between them to coordinate.
- Processes are assumed to have private state that is revealed through messages only.

Interaction model

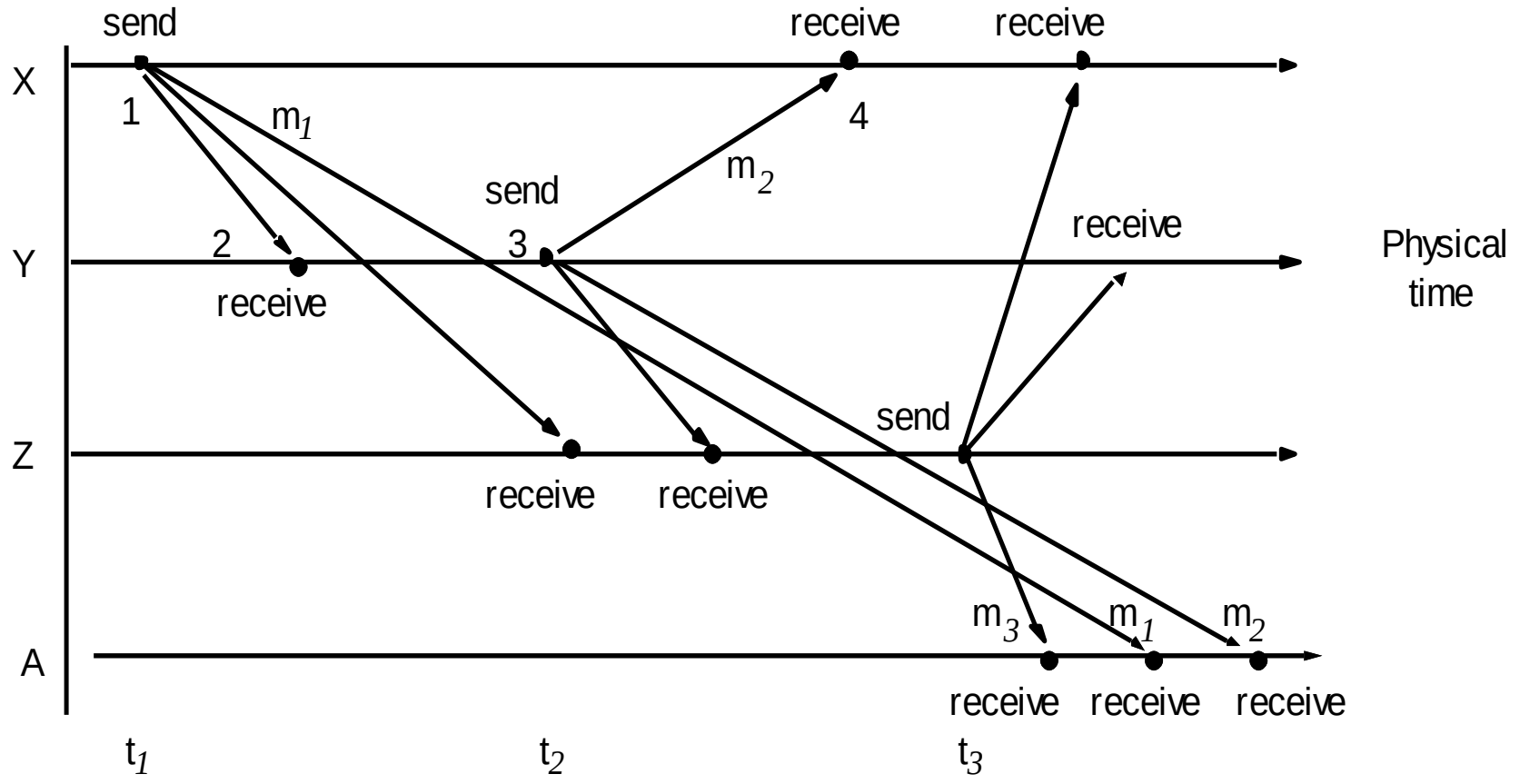
- Two big limiting factors
  - Communication performance
    - Latency: Amount of time to send a minimal message from point A to point B.
    - Bandwidth: Amount of information that can be transmitted per unit time.
    - Jitter: Variability in the time to deliver each of a sequence of messages. This is very critical to consider in multimedia apps.
  - A single global notion of time is impossible to maintain.
    - Each computer has a single internal clock.
    - Two clocks read simultaneously will yield different values. Real clocks drift from “real time” and each other.
    - Synchronization methods required to correct for drift.

Variants in Interaction Model

- Based on these limiting factors, we can coarsely break interaction models into two classes.

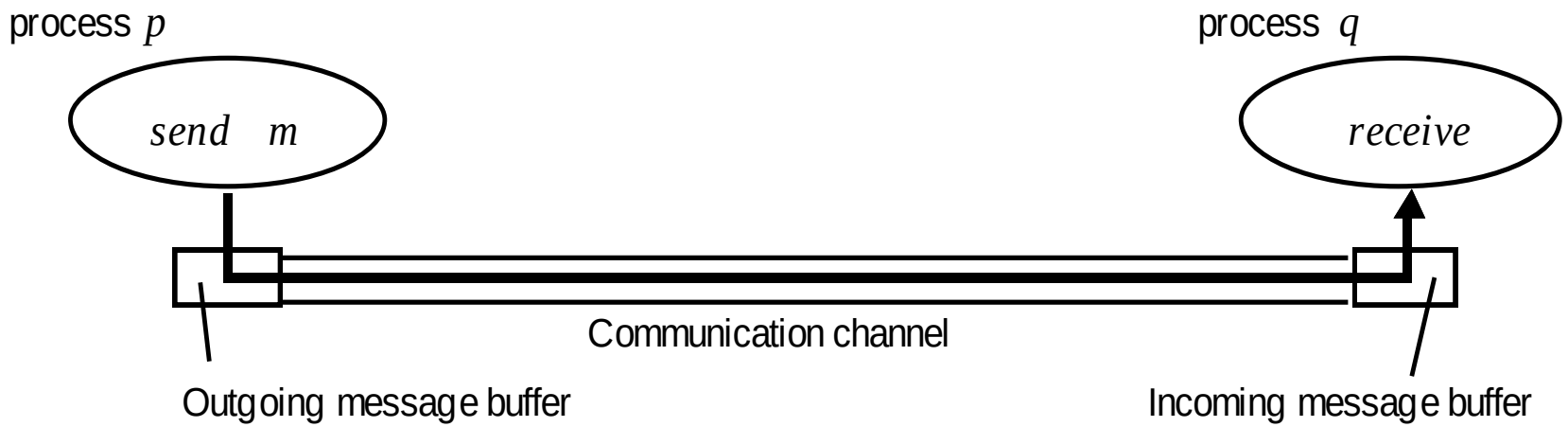
- Synchronous systems
  - Time to execute a step has known lower/upper bounds.
  - Messages received in a known bounded time.
  - Processes with local clocks drift at a known bound for the drift rate from real time.
- Asynchronous systems
  - No assumptions on bounds for three points above.
  - Real systems tend to be asynchronous.

Real-time ordering of events



Processes and Communication Channels

We typically assume a simplified channel when we want to reason about communication between processes for modeling purposes.



Failure model

- Omission failures
  - Communication channel fails to perform some action it was supposed to.
- Arbitrary failure
  - Any type of error can occur. These are pretty bad.
  - Examples: Undetected data corruption, reordering, duplication.
- Timing failure
  - Failure to respond in some time interval. Related to synchronous systems with expected bounds.

Masking failures

- Hiding a failure by converting it to a less-bad type.
  - Checksums to change arbitrary failure into an omission failure by determining that a packet is corrupt and must be rejected.
  - Protocols can make some failures totally vanish to the user or application. E.g.: Retries or resends when omissions occur.
  - One of the points of middleware layers is to provide this masking.
    - Checksums on messages.
    - Logical time to order messages and prevent dupes and out of order messages.

Omission and Arbitrary Failures

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send,* but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Timing Failures

Class of Failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

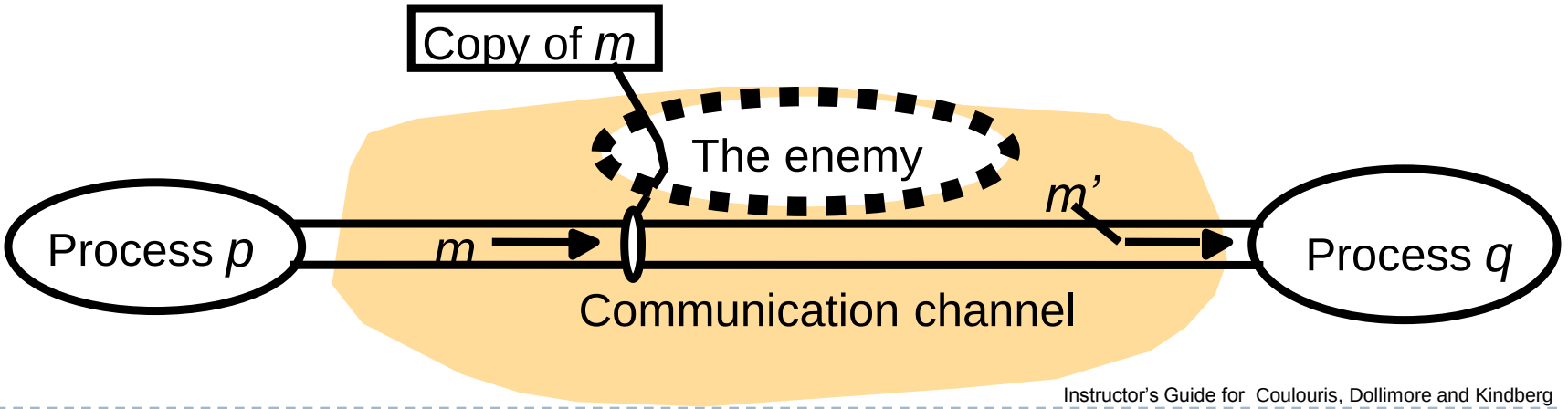
Reliability of one-to-one communications

- Validity and integrity define reliability characteristics, related both to security and failure.

- Validity
  - Any message in an outgoing buffer eventually makes it to a corresponding incoming buffer.
- Integrity
  - Message received is identical to the one that was sent without duplication.
  - Threats to integrity:
    - Protocols that retransmit but don't reject multiply transmitted messages.
    - Malicious users who inject, replay, or tamper with messages.

Security models

- Protection of objects
  - Specification and enforcement of access rights.
- The enemy
  - Threats to processes
    - Spoofing
  - Threats to communication channels
    - Secure channels help address this



Detection and prevention of threats

- Cryptography and shared secrets
  - Public key cryptography
  - Certificates
- Authentication
- Secure channels

- A fundamental model for security may be the handshaking and exchange of messages related to key or certificate authentication.
- Similarly, hashes and checksums can be used to detect threats/attacks similar to their use to detect corruption due to hardware or software failures.

Other threats

- Denial of service (DoS) attacks
  - Overwhelm a service with useless requests to prevent the service from servicing legitimate users.
- Mobile code
  - Viruses
  - Worms
  - Trojan horses
  - Generic “malware”