

# Distributed Software Development Fundamentals

Chris Brooks

Department of Computer Science  
University of San Francisco

# Outline

- Networking overview
  - Seven-layer model
- Intro to Distributed Systems
  - Characteristics
  - Desirable Properties
  - Dealing with Time

# Outline

- Networking overview
  - Seven-layer model
- Intro to Distributed Systems
  - Characteristics
  - Desirable Properties
  - Dealing with Time

# TCP/IP in 30 minutes

- Goal: Understand how a network transmits messages at different layers.
- How is a network composed?
- What really happens when Firefox opens a connection to a web server?
- Note: this will be an overview: for more details, take the networking class.

# Layering

- Modern network design takes advantage of the idea of *layering*
- A particular service or module is constructed as a black box.
- Users of that service do not need to know its internals, just its interface.
- This makes it easy to later build new modules (or layers) that use the lower layers.
- For example, HTTP is built on top of TCP.
  - A web browser does not typically need to worry about the implementation of TCP, just that it works.
- Unlike OO modules, the layers in a networked system comprise protocols that span multiple machines.

# The OSI seven-layer model

- ISO (a standards body) developed a reference model called OSI that defines the different layers needed for communication, and specifies which should do each job.
- The goal is to produce an open protocol that allows for heterogeneous, extensible systems.
- A *protocol* is a specification describing the order and format of messages.
- An open protocol is one in which all of this information is publicly available.

# The OSI seven-layer model

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

# Message transmission across layers

- An application (such as a web browser) wants to send a message to another computer.
- That application constructs a message and passes it to the application layer.
- The application layer attaches a header to the message and passes it to the presentation layer.
- The presentation layer attaches a header and passes it to the session layer, and so on.

# Message transmission across layers

- On the other end, the message is received by the physical layer, who strips off the appropriate header and passes the message up to the data link layer.
- This continues until the message reaches the application layer of the receiving machine.
- High-level layers don't need to worry about lower-level layers.
- Lower-level layers treat everything from higher layers as data to be sent.

# Layers and packets

- Each layer constructs a packet containing a portion of the data to be transmitted.
- This packet has a data section, and a header.
  - The header contains origin and destination information, checksums, sequence numbers, and other identifying information.
- When a message is sent by TCP, a packet is constructed and passed down to the IP layer.
- This entire packet then becomes the data portion of the IP packet, which is passed down to the network layer, and so on.
- On the other end, the lowest layer removes the header and checks the data integrity, then passes the data portion up to the next layer.

# Physical Layer

- This is the lowest-level layer, responsible for transmitting 0s and 1s.
- Governs transmission rates, full or half-duplex, etc.
- A modem works at the physical layer.
- Lots of interesting problems at this level that we won't get into ...

# Data Link Layer

- The data link layer provides error handling for the physical layer.
- Individual bits are grouped together into frames.
- A checksum is then computed to detect transmission errors.
- The data link layer can then request a retransmission of an error is detected.
- Messages are numbered; receiver can request re-transmission of any message in a sequence.
- Each frame is a separate, distinct message.
- The Data link layer provides error-free transmission to upper-level layers.

# Network Layer

- The network layer is responsible for routing and flow control.
- The network layer removes the data link header and examines the resulting packet for a destination, and then forwards it as appropriate.
- the Internet Protocol (IP) is one of the best-known network-layer protocols.
- Primary role: move packets from a sending host to a receiving host. This involves:
  - Routing: determine the path that a packet should take to get to its destination.
  - Forwarding: When an incoming packet is received, place it on the output link that takes it to the next hop in its route.

# Network Layer

- A router contains a *forwarding table* - when an incoming packet is received, the router compares it to this table to determine where to send it next.
  - This is forwarding.
- These forwarding tables are configured by means of a routing algorithm.
- For example, the link-state algorithm is a version of Dijkstra's algorithm: computes a global routing table.
- Internet routing algorithms (such as BGP) are more complex, and use a decentralized routing table.
- In a nutshell, BGP lets subnets figure out how to reach other subnets via a gateway. That gateway is then responsible for routing within the subnet.

# Transport Layer

- The network layer still operates at the level of individual packets, or datagrams.
- Packets may get lost, or arrive out of order.
- TCP is a transport-level protocol that provides *connection-oriented* service.
  - Guaranteed, in-order delivery.
  - State is maintained.
- This layer will also manage quality-of-service and some congestion control.
- UDP is also a transport level protocol, albeit one that does not provide connection-oriented delivery.

# Session Layer

- The session layer was designed to provide support for access rights and synchronization.
- In practice, it is not widely used, and is not present in the TCP/IP suite.

# Presentation Layer

- The presentation layer controls display of packet information.
- This may include encryption/decryption, compression, translation between character formats.

# Application Layer

- This is the layer that most of us are most familiar with.
- It consists of user-level protocols built on top of the existing layers.
  - HTTP
  - FTP
  - SMTP
  - P2P protocols
  - Instant messaging
  - RTSP/streaming video
  - etc.

# An example: HTTP

- HTTP is the protocol that drives the Web.
  - A side note/axe to grind: WWW  $\neq$  Internet!!
- It is a stateless protocol that uses TCP as its underlying protocol.
  - The client sends a request, which is processed by the server.
  - The server sends a reply, and the exchange is ended.

# HTTP requests

- HTTP has a very simple message format.

```
GET /~brooks/index.html HTTP/1.1
```

```
Host: www.cs.usfca.edu
```

```
Connection: close
```

```
User-agent: Mozilla/4.0
```

```
Accept-language: en
```

- You can try this out for yourself with telnet ...

# HTTP

- There are lots of wrinkles and extensions to HTTP
  - Cookies to help save state
  - CGI, SOAP to pass data and execute code as the result of an HTTP request.
  - Web caching to store data closer to clients.
- These are all possible because HTTP is an open protocol.
- This is also what makes it possible for different companies to write web browsers and web servers that seamlessly work together.

# Summary

- The modern networking stack can be conceptually broken into a set of layers.
- Each layer has a specific, well-defined function.
  - Acts as a black box
- Higher-level layers build on the functionality of lower-level layers.
- We'll be primarily concerned with the Transport and Application layers.

# Outline

- Networking overview
  - Seven-layer model
- Intro to Distributed Systems
  - Characteristics
  - Desirable Properties
  - Dealing with Time

# What is a Distributed System?

- What is a distributed system?
  - (Coulouris) “A distributed system is one in which hardware or software components communicate or coordinate their actions only by passing messages.”
  - (Tanenbaum) “A distributed system is a collection of independent computers that appear to the users of the system as a single computer.”
  - (Lamport) “You know you have one when the crash of a computer you’ve never heard of stops you from getting any work done.”
- All of these get at different aspects of the issue ...

# Advantages of a distributed system

- Can share expensive resources or data
- Economics
  - A collection of PCs can provide better price/performance than a single mainframe.
- Speed
  - A distributed system will often have more computing power than a single mainframe.
- Inherent distribution
  - Often, your data/users/resources are geographically distributed

# Advantages of a distributed system

- Reliability
  - If one node fails, the rest of the system can continue
- Incremental growth
  - Components can be added or replaced in small increments.

# Disadvantages of distributed systems

- Software design is much more complicated.
  - Lack of appropriate tools/languages
  - Disagreement on principles: how much should users know about the system? How much the system handle on a user's behalf?
- Potential network saturation
- Privacy and security issues
  - Allowing resources to be shared can lead to data leakage
- Extra sysadmin work

# Design Issues

- Transparency
- Flexibility
- Dependability
- Performance
- Scalability

# Transparency

- The goal of transparency is a *single-system image*
  - From the user's POV, it looks like a single machine.
- Types of transparency:
  - Location transparency - Users cannot tell where their resources are actually located.
  - Migration transparency - Resources can move without changing their names.
  - Replication transparency - the number of copies of a resource is hidden from users.
  - Concurrency transparency - Users can share resources without being aware of other users.
  - Parallelism transparency - A task can be run on multiple machines without the user being aware of it.

# Transparency

- Is transparency always a good thing? What is the downside?

# Flexibility

- Flexibility refers to how easy or difficult it is to change or reconfigure a system.
- The research question is how to best provide flexibility.
- In the OS world, this debate shows up in the comparison of monolithic kernels and microkernels.
  - Monolithic kernel - Provides most services on its own
  - Microkernel - Only handles a simple set of services.  
Most other services are implemented at the user level.
- Microkernel is very flexible and modular; services can be added, deleted, or moved without much reconfiguration.
- Monolithic kernel gives better performance.

# Reliability

- There are several different aspects of reliability:
  - Availability: what fraction of the time is the system usable?
  - Integrity: Data must be kept consistent. (this sometimes clashes with availability)
  - Security: Unauthorized usage must be prevented.
  - Fault tolerance: How unpleasantly does the system fail? Is data lost? Can recovery happen?

# Performance

- Performance is trickier than it appears.
- Lots of possible metrics:
  - Response time
  - Throughput
  - System utilization
  - Network capacity
- Typically, communication costs dominate
- This leads to a *coarser-grained* parallelism than we would see in a parallel computer.

# Types of process failure

- In looking at different protocols and algorithms, we'll want to know what types of failure they are resistant to.
  - Failstop (or crash): Process halts and remains in that state: failure can usually be detected.
  - Send omission: a process fails to send messages, or halts in the middle of sending.
  - Receive omission: a process fails to receive a message properly, or halts while receiving.
  - General omission: combination of send and receive omission
  - Byzantine failure: Process behaves in an arbitrarily incorrect way.
  - Timing failure: clock drift exceeds allowable bounds.

# Types of Communication Failure

- We also must consider failures that happen in the network:
  - Crash: a link stops completely.
  - Omission: A link fails to transmit some of its messages.
  - Byzantine: A link can exhibit any possible behavior, including generating spurious messages.
- Note: A Byzantine failure can be treated the same as an attacker/intruder.

# Communication paradigms

- Reliable communication: messages are guaranteed to eventually arrive.
- In-order: messages are guaranteed to arrive in the order they are sent.

# Communication paradigms

- Asynchronous: there is no bound on message delay
- Synchronous:
  - Known upper bound  $b$  on message delay
  - Every process  $p$  has a local clock  $C_p$  which drifts at a rate of  $r > 0$  and  $\forall p$  and  $\forall t > t'$ :
$$(1 + r)^{-1} \leq \frac{C_p(t) - C_p(t')}{t - t'} \leq (1 + r)$$
  - In English, clock drift has an upper and lower bound.
  - Also, bounds on the amount of time needed for a process to execute a single step.
- Synchronous communication allows you to implement approximately synchronized clocks, even in the presence of failure.

# Dealing with time

- One of the fundamental problems in distributed systems is dealing correctly with time.
- Not only when things happened, but what order things happened in.
- We would like for all processes to see *relevant* changes in the same order.
  - Example: updating a replicated database.
- Depending on the communication model, this may be quite difficult.
- Insight: often, it doesn't matter exactly what time an operation happens, but what order events occur in.
- (exception: hard real-time systems)

# Global time servers

- NTP is an Internet Protocol that allows your machine to synchronize its clock with a remote source, thereby keeping it accurate.
- Is that all we need to do?

# Global time servers

- NTP is an Internet Protocol that allows your machine to synchronize its clock with a remote source, thereby keeping it accurate.
- Is that all we need to do?
- Maybe. Maybe not.
  - What if we don't have an Internet connection, or NTP is blocked by our firewall?
  - Can we guarantee that all users use the same remote time server?
  - How often should they update?
  - What if users don't do this?

# Logical time

- The algorithms we'll look at in this class will not need to depend on the *absolute* time that something happens.
- Instead, we'll be interested in the *logical* time, or *causal order* in which events occur.
- As long as all processes agree on the order in which a set of events that influence each other occurs, we're OK.
- We'll spend time next week looking at this problem.

# Summary

- There are lots of desirable properties and design issues for distributed systems.
  - Performance, scalability, reliability, flexibility, transparency
  - Often, we must sacrifice one for another
  - Some (e.g. Parallel transparency) are not possible with today's technology.
- Communication can be either *synchronous* or *asynchronous*
- Time is a very sticky problem to deal with in distributed systems.
- Characterizing types of failure will help us identify what our algorithms and systems can and cannot stand up to.