Fundamentals of Distributed Systems – CSE 40822 – Cloud Computing
Prof Douglas Thain, University of Notre Dame, January 2016.

Caution: These are high level notes that I use to organize my lectures. You may find them useful for reviewing main points, but they aren't a substitute for participating in class.

My definitions:

- A **distributed system** is a set of processes communicating over a network.

- A **cloud** is a particular kind of distributed system: a virtual resource pool with an interface that (mostly) hides the complexity of using the resource.

To understand clouds, we must first have a handle on distributed systems in general, so this week is a crash course in operating systems, networks, and then distributed systems, which is the combination of the two.

Crash Course in Operating Systems

The earliest machines had no OS, which made sharing and portability hard. A modern OS exists to share resources between competing users, and to allow programs to move portably between different machines.

Layers of a conventional operating system.
    Hardware: IDE, Ethernet, VGA
    Drivers: Disk, Network, Video
    Abstractions: Filesystem, TCP/IP
    System Calls: open/read/write/fork/exec
    Applications: firefox, emacs gcc

A **process** is a running program that has its own private address space, and is protected from interference by other programs. It is both a unit of concurrency and a unit of independent failure. (i.e. A process can be safely killed.)

A **thread** is an additional unit of concurrency that can run inside a process. But it is not a unit of independent failure: threads cannot be killed in any reliable way.

Multiprocess Server Example:
    ssh process on client, sshd on the server.
    sshd forks on connect, then forks user's shell
        What happens on failures?
        What happens for multiple users?

HTTPD Example

Browser on client, httpd on the server.
Single process: httpd serves files directly.
(What happens on large file downloads?)
Multi process: httpd forks on every request.
(What happens when you have too many clients?)

Crash Course in Networking

Architecture of the Internet:

LAN/switch connects machines over short distances.
(Ethernet, Token Ring, Wireless, DSL, etc…)
Machines communicate by sending short packets with a header.

WAN/routers connects LANs over wide areas.
The Internet Protocol provides a common messaging format across network technologies, carried as a payload in LAN packets.

Diagram of OSI Networking Layers:

Physical:	Cat5, Optical, RF
Data Link:	Ethernet, Token Ring, 802.11
Network:	Internet Protocol
Transport:	TCP / UDP
Application:	HTTP, FTP, DNS …

Most Commonly Used Protocols:

UDP – Short messages, unreliable delivery
TCP - Long streams, ordered, reliable delivery.
(But in what sense is TCP "reliable" ?)

Idealized Vision of the Internet

Anyone can send data to anyone else!
Core of the network is dumb and unreliable.
End points have all the reliability and policy.

Reality of the Internet

Manually configured firewalls blocks all sorts of traffic.
(Even the good guys have to ask for permission to communicate.)
Shortage of addresses in IPV4.

Abstract view of the Internet from applications:

Send packets to remote hosts.
They may arrive, or they may not.
It is up to the other side to acknowledge in some way!

Introduction to Distributed Computing

We would like to build distributed systems that work as simply and reliably as non-distributed systems, but it simply isn't possible. Distributed systems are fundamentally different than standalone machines in (at least) four ways outlined by "A Note on Distributed Computing":

Latency
Memory
Partial Failure
Concurrency
and Autonomy (says Prof Thain)

"A Note" discusses this common fallacy:

"Let's take an existing program, break it into pieces (functions, objects, modules, etc) and then connect the pieces over the network. Now we have a usable distributed system that works just like the original system."

(This is the key idea in RPC, CORBA, DCOM, RMI, and many other similar systems.)

It does not work because distributed systems are fundamentally different.

Easy to show with a thought experiment:

Suppose you have a regular program makes use a library that implements a stack data structure with the operations push(x) and x=pop().

We want to share the stack among multiple distributed users, so put the stack in a separate server process, and have it accept and return messages. If the client sends "push(x)", the server responds with "ok". If the clients sends "pop()" the server responds with "x", which is the value at the top of the stack.

Messages can be lost, so if the client doesn't get a response in a reasonable amount of time, it simply sends the request message again.

Finally, add a little bit of code ("stubs") to each side, so that the program (and the stack library) still function as normal.

Questions to consider:

What happens if the client's messages are lost?
What happens if the server's responses are lost?
What happens if multiple clients do this simultaneously?

Small group discussions:

Design a solution to this problem. Change the messages exchanged so that no data is lost, and the stack still works as desired.

Moral of the story:

Interfaces to distributed systems must be designed from scratch to accommodate failure and concurrency!

Example: Two phase commit in credit card processing.