What is a “distributed system”?

- Nearly all systems today are distributed in some way
  - they use email
  - they access files over a network
  - they access printers over a network
  - they’re backed up over a network
  - they share other physical or logical resources
  - they cooperate with other people on other machines
  - they access the web
  - they receive video, audio, etc.

Distributed systems are now a requirement

- Economics dictate that we buy small computers
- Everyone needs to communicate
- We need to share physical devices (printers) as well as information (files, etc.)
- Many applications are by their nature distributed (bank teller machines, airline reservations, ticket purchasing)
- To solve large problems, we need to get large collections of small machines to cooperate together (e.g., Google’s search infrastructure, BOINC (SETI@home))

Loosely-coupled systems

- Earliest systems used simple explicit network programs
  - FTP (rcp): file transfer program
  - telnet (rlogin/rsh): remote login program
  - mail (SMTP)
- Each system was a completely autonomous independent system, connected to others on the network

Closely-coupled systems

- A distributed system becomes more “closely-coupled” as it
  - appears more uniform in nature
  - runs a “single” operating system
  - has a single security domain
  - shares all logical resources (e.g., files)
  - shares all physical resources (CPUs, memory, disks, printers, etc.)
- In the limit, a distributed system looks to the user as if it were a centralized timesharing system, except that it’s constructed out of a distributed collection of hardware and software components
Tightly-coupled systems

- A “tightly-coupled” system usually refers to a multiprocessor
  - runs a single copy of the OS with a single job queue
  - has a single address space
  - usually has a single bus or backplane to which all processors and memories are connected
  - has very low communication latency
  - processors communicate through shared memory

Some issues in distributed systems

- Transparency (how visible is the distribution)
- Security
- Reliability
- Performance
- Scalability
- Programming models
- Communication models

Example: Grapevine distributed mail service

- Xerox PARC, 1980
  - cf. Microsoft Outlook/Exchange today!!!!!
- Goals
  - cannot rely on integrity of client
  - once the system accepts mail, it will be delivered
  - no single Grapevine computer failure will make the system unavailable to any client either for sending or for receiving mail
- Components
  - GrapevineUser package on each client workstation
  - Registration Servers
  - Message Servers
- Implementation: Remote Procedure Call

Grapevine: Functional diagram

Grapevine: Sending a message

- User prepares message using mail client
- Mail client contacts GrapevineUser package on same workstation to actually send message
- GrapevineUser package
  - contacts any Registration Server to get a list of Message Servers
  - contacts any Message Server to transmit message
    - presents source and destination userids, and source password, for authentication
    - Message Server uses any Registration Server to authenticate
    - sends message body to Message Server
      - Message Server places it in stable storage and acknowledges receipt

Grapevine: Transport and buffering

- For each recipient of the message, Message Server contacts any Registration Server to obtain list of Message Servers holding mail for that recipient
- Sends a copy of the message to one of those Message Servers for that recipient
Grapevine: Retrieving mail

- User uses mail client to contact GrapevineUser package on same workstation to retrieve mail
- GrapevineUser package
  - contacts any Registration Server to get a list of each Message Server holding mail for the user ("inbox site")
  - contacts each of these Message Servers to retrieve mail
    - presents user credentials
    - Message Server uses any Registration Server to authenticate
    - acknowledges receipt of messages so that the server can delete them from its storage

Grapevine: Scalability

- Can add more Registration Servers
- Can add more Message Servers
- Only thing that didn’t scale was handling of distribution lists
  - the accepting Message Server was responsible for expanding the list (recursively if necessary) and delivering to an appropriate Message Server for each recipient
  - some distribution lists contained essentially the entire user community
- Jeff Dean (Google) told us they don’t even think about more than two decimal orders of magnitude
  - fundamental design decisions will need to change
  - advances in technology will make it possible

Example: Google search infrastructure

- It’s likely that Google has several million machines
  - But let’s be conservative – 1,000,000 machines
  - A rack holds 176 CPUs (88 1U dual-processor boards), so that’s about 6,000 racks
  - A rack requires about 50 square feet (given datacenter cooling capabilities), so that’s about 300,000 square feet of machine room space
    - more than 6 football fields of real estate – although of course Google divides its machines among dozens of datacenters all over the world
  - A rack requires about 10kw to power, and about the same to cool, so that’s about 120,000 kw of power, or nearly 100,000,000 kwh per month ($10 million at $0.10/kwh)
    - Equivalent to about 20% of Seattle City Light’s generating capacity

• There are multiple clusters (of thousands of computers each) all over the world

  - Many hundreds of machines are involved in a single Google search request (remember, the web is 400+TB)
  1. DNS routes your search request to a nearby cluster

A cluster consists of Google Web Servers, Index Servers, Doc Servers, and various other servers (ads, spell checking, etc.)

- These are cheap standalone computers, rack-mounted, connected by commodity networking gear
  
  2. Within the cluster, load-balancing routes your search to a lightly-loaded Google Web Server (GWS), which will coordinate the search and response
• The index is partitioned into “shards.” Each shard indexes a subset of the docs (web pages). Each shard is replicated, and can be searched by multiple computers – “index servers”

3. The GWS routes your search to one index server associated with each shard, through another load-balancer

4. When the dust has settled, the result is an ID for every doc satisfying your search, rank-ordered by relevance

• The docs, too, are partitioned into “shards” – the partitioning is a hash on the doc ID. Each shard contains the full text of a subset of the docs. Each shard can be searched by multiple computers – “doc servers”

5. The GWS sends appropriate doc IDs to one doc server associated with each relevant shard

6. When the dust has settled, the result is a URL, a title, and a summary for every relevant doc

– Enormous volumes of data
– Extreme parallelism
– The cheapest imaginable components
  – Failures occur all the time
  – You couldn’t afford to prevent this in hardware
– Software makes it
  – Fault-Tolerant
  – Highly Available
  – Recoverable
  – Consistent
  – Scalable
  – Predictable
  – Secure

How on earth would you enable mere mortals write hairy applications such as this?

• Recognize that many Google applications have the same structure
  – Apply a “map” operation to each logical record in order to compute a set of intermediate key/value pairs
  – Apply a “reduce” operation to all the values that share the same key in order to combine the derived data appropriately

• Build a runtime library that handles all the details, accepting a couple of customization functions from the user – a Map function and a Reduce function

• That’s what MapReduce is
  – Supported by the Google File System and the Chubby lock manager
  – Augmented by the BigTable not-quite-a-database system