# Distributed Systems
# 8L for Part IB

Lecture 8

Dr Robert N. M. Watson

1

## Last Time

- Looked at replication in distributed systems
- **Strong consistency**:
  - Approximately as if only one copy of object
  - Requires considerable coordination on updates
  - Transactional consistency & quorum systems
- **Weak consistency**:
  - Allow clients to potentially read stale values
  - Some guarantees can be provided (FIFO, eventual, session), but at additional cost to availability
- **Service replication**:
  - **Stateless** (easy!) or **Passive** (primary/backup) common, **Active** (state-machine replication) less so

2

## Access Control

- Distributed systems may want to allow access to resources based on a security policy
- As with local systems, three key concepts:
  - **Identification**: who you are (e.g. user name)
  - **Authentication**: proving who you are (e.g. password)
  - **Authorization**: determining what you can do
- Can consider authority to cover actions an authenticated subject may perform on objects
  - **Access Matrix** = set of rows, one per subject, where each column holds allowed operations on some object

3

## ACLs and Capabilities

- Access matrix is typically large & sparse:
  - Just keep non-NULL entries by column or by row
- **Access Control Lists**:
  - Keep columns, i.e. for each object O, keep list of subjects and their allowable access
  - ACLs stored with objects (e.g. local filesystems)
  - Bit like a guest list on the door of a night club
- **Capabilities**:
  - Keep rows, i.e. for each subject S, keep list of objects and the allowable access to them
  - Capabilities stored with subjects (e.g. processes)
  - Bit like a key or access card that you carry around

4

## Access Control in Distributed Systems

- In single systems usually have small number of users (subjects) and large number of objects:
  - e.g. a few hundred users in a Unix system
  - Easy to track subjects (e.g. effective user id of current process), and to keep ACL with objects (e.g. with files)
- Distributed systems are large & dynamic:
  - Can have huge (and unknown?) number of users
  - Interactions over the network – may not have explicit 'log in' and associated process per user
- Capability model is a more natural fit:
  - Client presents capability with request for operation
  - System only performs operation if capability checks out

5

## Cryptographic Capabilities

- Privileged server can issue capabilities
  - e.g. has secret key **k** and a one-way function **f**()
  - Issues a capability <*oid*, *access*, **f**(**k**, *oid*, *access*) >
  - Simple example is **f**(k,o,a) = **sha1**(k|o|a)
- Client transmits capability with request
  - If server knows **k**, can check if operation allowed
  - (otherwise can ask privileged server to validate)
- Can use same capability to access many servers
  - And one server can use it on your behalf
  - e.g. allow web tier to access objects on storage tier

6

## Capabilities: Pros and Cons

- Relatively simple and pretty scalable
- Allow anonymous access (i.e. server does not need to know identity of client)
  - And hence easily **allows delegation**
- However this also means:
  - Capabilities can be stolen (unauthorized users)…
  - … and are **difficult to revoke** (like someone cutting a copy of your house key)
- Can address these problems by:
  - Having time-limited validity (e.g. 30 seconds)
  - Incorporating version into capability, and storing version with the object: increasing version => revoke all access

7

## Combining ACLs and Capabilities

- Recall one problem with ACLs was inability to scale to large number of users (subjects)
- However in practice we may have a small-ish number of authority levels
  - e.g. moderator versus contributor on chat site
- Can use to build **role-based access control**:
  - Have (small-ish) well-defined number of roles
  - Store ACLs at objects based on roles
  - Allow subjects to **enter** roles according to some rules
  - Issue capabilities which attest to current role

8

## Role-Based Access Control

- General idea is very powerful
  - Separates { principal → role }, { role → privilege }
  - Developers of individual services only need to focus on the rights associated with a role
  - Easily handles evolution (e.g. an individual moves from being an undergraduate to an alumnus)
- Possible to have sophisticated rules for role entry:
  - e.g. enter different role according to time of day
  - or entire role hierarchy (1B student <= CST student)
  - or parametric/complex roles ("the doctor who is currently treating you")

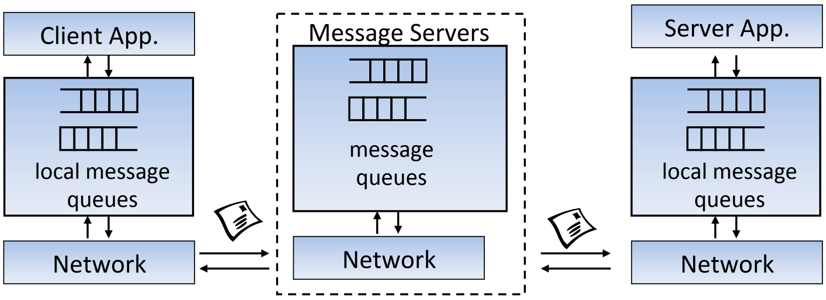
9

## Single-System Sign On

- Distributed systems inherently involve a number of different machines
  - Frustrating to have to authenticate to each one!
- Single-system sign on aims to ease user burden while maintaining good security
  - e.g. Kerberos, Microsoft Active Directory let you authenticate to a single **domain controller**
  - Get a session key and a ticket (~= a capability)
  - Ticket is for access to the **ticket-granting server** (TGS)
  - When wish to e.g. log on to another machine, or access a remote volume, s/w asks TGS for a ticket for that resource
- Some wide-area schemes too (OpenID, Shibboleth)

10

# Coordination Services




- Earlier looked at middleware support for RPC/RMI
  - Imperative and (typically) synchronous interaction
- An alternative is **message-oriented middleware**
  - Communication via asynchronous messages
  - Messages stored in **message queues**

11

# MOM: Pros and Cons

- **Asynchronous interaction**
  - Client and server are only loosely coupled
  - Messages are queued
  - Good for application integration
- Support for **reliable delivery service**
  - Keep queues in persistent storage
- Processing of messages by message server(s)
  - May do filtering, transforming, logging, …
  - Networks of message servers
- But pretty low-level ('packet level') interactions, and still just point-to-point messages with no typing...
- Examples: IBM MQSeries, Java Message Service (JMS)

12

# Publish-Subscribe

- Get more flexibility with publish-subscribe:
  - **Publishers** advertise and publish **events**
  - **Subscribers** register interest in **topics** (i.e. a set of properties of events)
  - **Event-service** notifies interested subscribers of published events
- Keeps asynchronous (decoupled) nature of message-oriented middleware but:
  - Allows 1-to-many communication
  - Dynamic membership (publishers and subscribers can join or leave at any time)

13

# Publish-Subscribe: Pros and Cons

- Pub/sub useful for 'ad hoc' systems such as embedded systems or sensor networks:
  - Client(s) can 'listen' for occasional events
  - Don't need to define semantics of entire system in advance (e.g. what to do if get event <X>)
- Leads to natural "reactive" programming:
  - when <X>, <Y> occur then do <Z>
  - event-driven systems like Apama can help understand business processes in real-time
- But:
  - Can be awkward to use if application doesn't fit
  - And difficult to make perform well…

14

## Simplifying Distributed Systems

- Traditional middleware systems provide a number of 'medium-level' abstractions
  - Naming and directory services
  - Synchronous RPC and asynchronous events
  - Group communication and ordered multicast
  - Failure detectors and membership protocols
  - Consensus schemes (2PC, 3PC, Paxos, …)
  - Capabilities and access control
- However still rather tricky to actually build a distributed system in the real world
- Recent advances in full (?!) distribution transparency

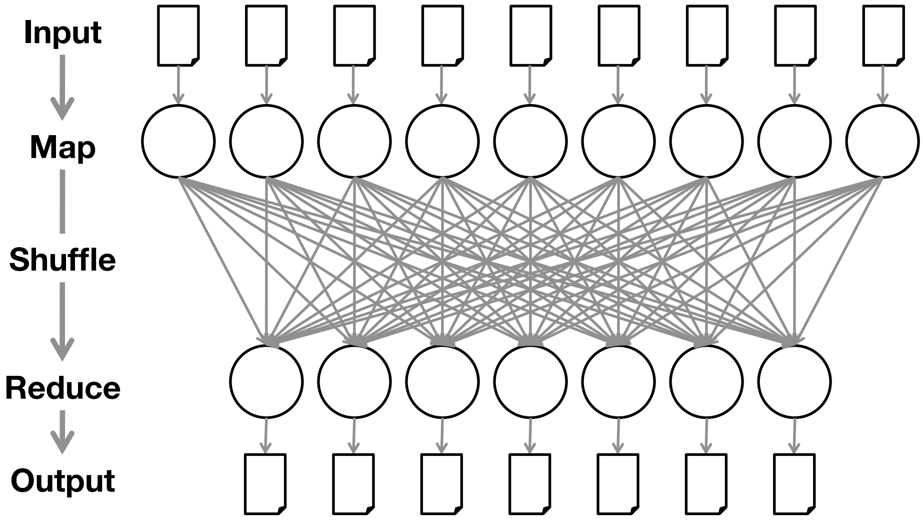
15

## Google's MapReduce

- Programming framework for datacenter scale
  - Run a program across 100's or 10,000's machines
- Framework takes care of:
  - Parallelization, distribution, load-balancing, scaling up (or down) & fault-tolerance
- Programmer provides two methods ;-)
  - map(key, value) -> list of (key', value') pairs
  - reduce(key', value') -> result
  - Inspired by functional programming

16

# MapReduce: The Big Picture




17

# Example Programs

- **Sorting** data is trivial (map, reduce both identity function)
  - Works since the shuffle step essentially sorts data
- **Distributed grep** (search for words)
  - map: emit a line if it matches a given pattern
  - reduce: just copy the intermediate data to the output
- **Count URL access frequency**
  - map: process logs of web page access; output <URL, 1>
  - reduce: add all values for the same URL
- **Reverse web-link graph**
  - map: output <target, source> for each link to *target in a page*
  - reduce: concatenate the list of all source URLs associated with a target. Output <target, list(source)>

18

## MapReduce: Pros and Cons

- **Extremely simple**, and:
  - Can auto-parallelize (since operations on every element in input are independent)
  - Can auto-distribute (since rely on underlying GFS distributed file system)
  - Gets fault-tolerance (since tasks are idempotent, i.e. can just re-execute if a machine crashes)
- Doesn't really use ***any*** of the sophisticated algorithms we've seen (though does use storage replication)
- However not a panacea:
  - Limited to batch jobs, and computations which are expressible as a map() followed by a reduce()

19

## Other Frameworks

- MapReduce stems from 2004, and Google (and others) have done a lot since then
- If interested check out Apache Hadoop
  - http://hadoop.apache.org/
- Includes HDFS and Hadoop (clones of GFS and MapReduce respectively), as well as:
  - Cassandra (scalable multi-master database), and
  - Zookeeper (coordination/consensus service)
- Lots of ongoing research in this space
  - Current hot topics involve dealing with iterative and/or real-time computations

20

## Summary (1)

- Distributed systems are everywhere
- Core problems include:
  - Inherently concurrent systems
  - Any machine can fail…
  - … as can the network (or parts of it)
  - And we have no notion of global time
- Despite this, we can build systems that work
  - Basic interactions are request-response
  - Can build synchronous RPC/RMI on top of this …
  - Or asynchronous message queues or pub/sub



## Summary (2)

- Coordinating actions of larger sets of computers requires higher-level abstractions
  - Process groups and ordered multicast
  - Consensus protocols, and
  - Replication and Consistency
- Various middleware packages (e.g. CORBA, EJB) provide implementations of many of these:
  - But worth knowing what's going on "under the hood"
- Recent trends towards even higher-level:
  - MapReduce and friends