Distributed Systems
8L for Part IB

Handout 4

Dr. Steven Hand

Weak Consistency

- Maintaining strong consistency has costs:
  - Need to coordinate updates to all (or Q_w) replicas
  - Slow… and will block other accesses for the duration
- **Weak consistency** provides fewer guarantees:
  - e.g. C1 updates (replica of) object x at S3
  - S3 lazily propagates changes to other replicas
  - Other clients can potentially read old ("stale") value
- Considerably **more efficient**:
  - Write is simpler, and doesn't need to wait for communication with lots of other replicas…
  - … hence is also **more available** (i.e. fault tolerant)

FIFO Consistency

- As with group communication primitives, various ordering guarantees possible
- **FIFO consistency**: all updates at S_i occur in the same order at all other replicas
  - As with FIFO multicast, can buffer for as long as we like!
  - But says nothing about how S_i's updates are interleaved with S_j's at another replica (may put S_j first, or S_i, or mix)
- Still useful in some circumstances
  - e.g. single user accessing different replicas at disjoint times
  - Essentially primary replication with primary=last accessed

Eventual Consistency

- FIFO consistency doesn't provide very nice semantics:
  - e.g. we write first version of file f to S_1
  - later we read f from S_2, and write version 2
  - later again we read f from S_3 – changes lost!
- What happened?
  - Update from S_1 arrived to S_3 after those from S_2, who thus overwrote them (stoooopid S_3)
- A desirable property in weakly consistent systems is that they converge to a more correct state
  - i.e. in the absence of further updates, every replica will eventually end up with the same latest version
- This is called **eventual consistency**

Implementing Eventual Consistency

- Servers S_i keep a **version vector** $V_i(O)$ for each object
  - For each update of O on S_i, increment $V_i(O)[i]$
  - (essentially a vector clock reused as a version number)
- Servers synchronize pair-wise from time to time
  - For each object O, compare $V_i(O)$ to $V_j(O)$
  - If $V_i(O) < V_j(O)$, S_i gets an up-to-date copy from S_j;
    if $V_j(O) < V_i(O)$, S_j gets an up-to-date copy from S_i.
- If Vi(O) ~ Vj(O) we have a **write-conflict**:
  - Concurrent updates have occurred at 2 or more servers
  - Must apply some kind of reconciliation method
  - (similar to revision control systems, and equally painful)

Example: Amazon's Dynamo

- Storage service used within Amazon's WS
  - By Amazon itself, and by 3rd party service providers
- Designed to emphasize availability above consistency:
  - SLA to ensure bounded response time 99.99% of the time
  - if customer wants to add something to shopping basket and there's a failure… still want addition to 'work'
  - Even if get (temporarily) inconsistent view… fix later!
- Built around notion of a so-called **sloppy quorum**:
  - Have N, Q_w, Q_r as before … but don't actually require that $Q_w > N/2$, or that $(Q_w + Q_r) > N$
  - Instead make tunable: **lower Q values = higher availability**
  - Also let system continue during failure; add a new replica

Session Guarantees

- Eventual consistency seems great, but how can you program to it?
  - Need to know something about what guarantees are provided to the client
- These are called **session guarantees**:
  - Not system wide, just for one (identified) client
  - Client must be a more active participant, e.g. client maintains version vectors of objects it has read & written
- Example: **Read Your Writes (RYW)**:
  - if C_i writes a new value to x, a subsequent read of x should see this update … even if C_i is now reading from a different replica
  - Need C_i to remember highest id of any update it made
  - Only read from a server if it has seen that update

Session Guarantees & Availability

- There are a variety of session guarantees
  - All deal with allowable state on replica given history of accesses by a specific client
  - (further examples included in additional, non-examinable material downloadable from course web page)
- Session guarantees are weaker than strong consistency, but stronger than ‘pure’ weak consistency:
  - But this means that they **sacrifice availability**
  - i.e. choosing not to allow a read or write if it would break a session guarantee means not allowing that operation!
  - ‘pure’ weak consistency would allow the operation
- Can we get the best of both worlds?

Consistency, Availability & Partitions

- Short answer: No ;-)
- The CAP Theorem (Brewer 2000, Gilbert & Lynch 2002) says you can only guarantee two of:
  - **Consistent data, Availability, Partition-tolerance**
- … in a single system.
- In local-area systems, can sometimes drop partition-tolerance by using redundant networks
- In the wide-area, this is not an option:
  - **Must choose between consistency & availability**
  - Most Internet-scale systems ditch consistency
- **NB**: this doesn't mean that things are always inconsistent, just that they're not always guaranteed to be consistent

Replication and Fault-Tolerance

- Can also use replication for a **service**:
- Easiest is for **stateless services**:
  - Simply duplicate functionality in K machines
  - Clients use any (e.g. closest), fail over to another
- Very few totally stateless services, but e.g. much of the web only has per-session soft-state:
  - State generated per-client, lost when client leaves
- Commonly used to scale multi-tier web farms:
  - First and second tiers (web servers and app servers) only have per-session soft-state => trivial to replicate
  - (clients are independent, so no coordination needed)
  - Third tier (storage/db tier) either partitioned (disjoint clients on different servers), or implements consistent replication

Primary/Backup (Passive) Replication

- A solution for stateful services is **primary/backup**:
  - Backup server takes over in case of failure
- Based around persistent logs and system checkpoints:
  - Periodically (or continuously) checkpoint primary
  - If detect failure, start backup from checkpoint
- A few variants trade-off fail-over time:
  - **Cold-standby**: backup server must start service (software), load checkpoint & parse logs
  - **Warm-standby**: backup server has software running in anticipation – just needs to load primary state
  - **Hot-standby**: backup server mirrors primary work, but output is discarded; on failure, enable output

Active Replication

- Have K replicas running at all times
- Front-end server acts as an **ordering node**:
  - Receives requests from client and forwards them to all replicas using totally ordered multicast
  - Replicas each perform operation and respond to front-end
  - Front-end gathers responses, and replies to client
- Typically require replicas to be "**state machines**":
  - i.e. act deterministically based on input
  - Idea is that all replicas operate 'in lock step'
- Active replication is expensive (in terms of resources)…
  - … and not really worth it in the common case.
  - However valuable if consider **Byzantine failures**

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

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

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

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

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

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

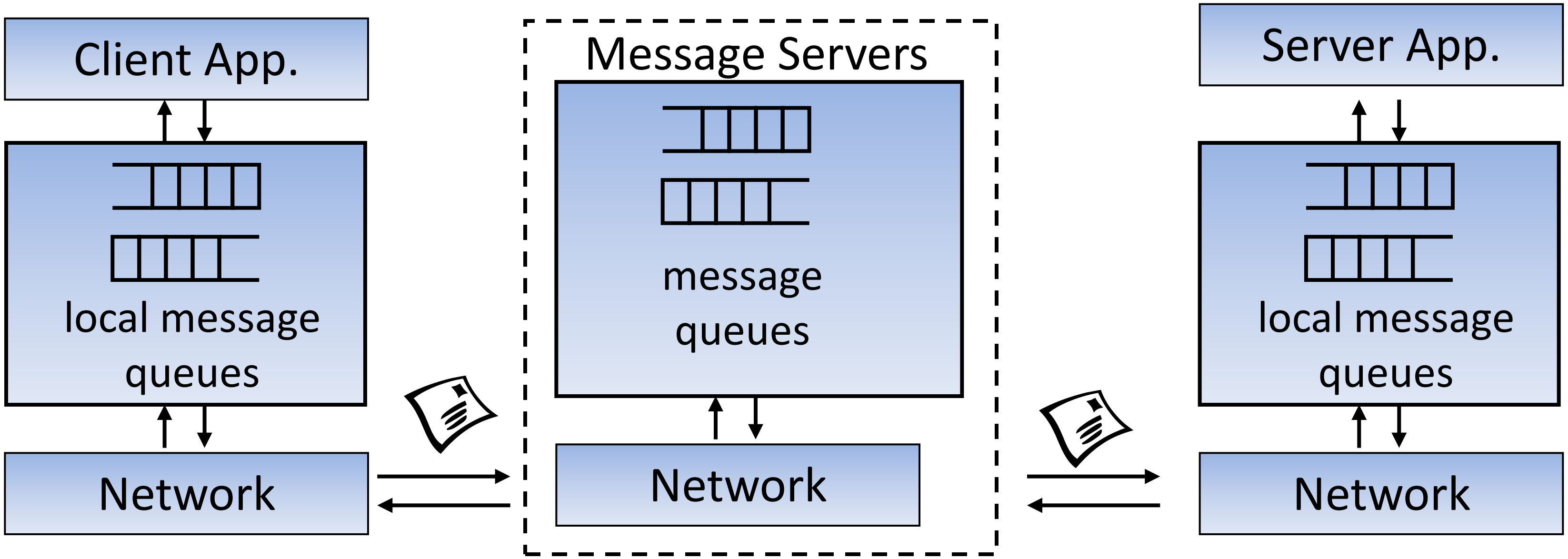
Role-Based Access Control

- 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")

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)

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**

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)

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)

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…

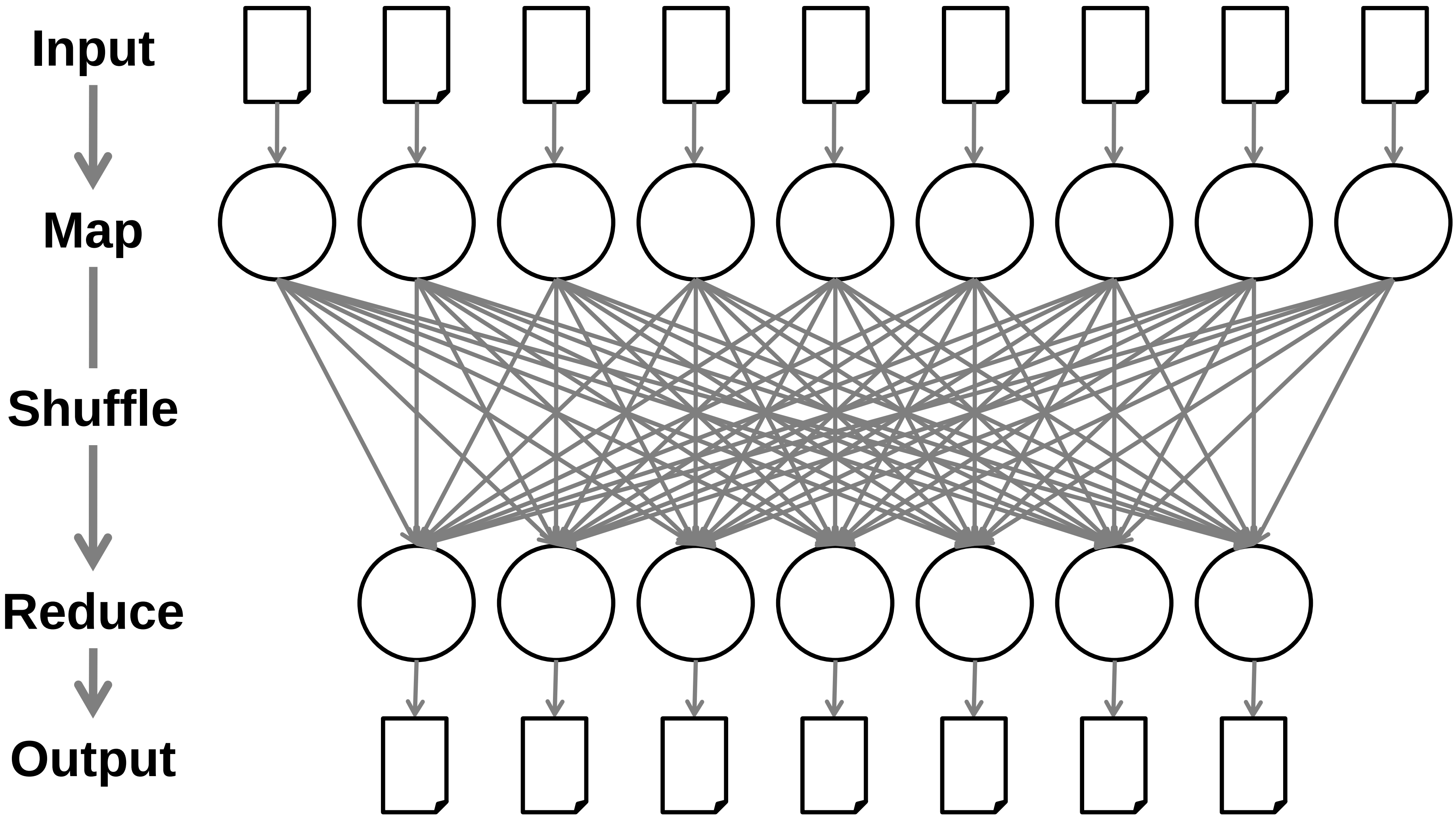
Simplifying Distributed Systems

- 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

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’ list) -> result
  - Inspired by functional programming

MapReduce: The Big Picture



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)>

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()

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

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