Quick course recap:

Remote procedure call: application code can be ignorant of where the implementation runs, except (!) for potential for failures and performance

Ordering of events in a distributed system: what does concurrency mean?

Memory consistency models: linearizability, serializability, eventual consistency, causal consistency, processor ordering – all of these and more you find in real systems

Cache coherence: how we can implement memory consistency with state as to who has what cached; as we saw in the first problem set, without this its pretty hard to achieve any reasonable consistency model

With disconnected operation, we need a different model: e.g., source code control where clients get a copy of the data, and then check in their changes, resolving any problems as they occur

Can we be correct and make progress when there can be failures? Not both! But with two phase commit, we can be able to coordinate actions across multiple nodes despite failures, and with Paxos we can coordinate actions such that we are correct in all cases, and

We then looked at a set of case studies:

Facebook illustrated typical sharded architecture: front-ends assemble the web page, memcache provides good performance in the common case, and the storage server is the backstop. Even so, multiple data centers makes for a challenging design problem.

We spent a while diving into Google's architecture:
Chubby – a lock server and cache manager based on Paxos
MapReduce and Spark – a simple programming model for data-oriented apps
GFS – large scale block storage with app-specific recovery
BigTable – a key-value store, where related data is stored together, using log-deltas for efficient storage and lookup
Spanner – extends BigTable to multiple data centers, using real-time for better performance

And we added a few other systems:
Dynamo – Amazon uses a key value store with eventual consistency and consistent hashing for load balancing

BitTorrent – mesh structure prevents any single path from being a single point of failure or bottleneck; because its an open p2p system, need to worry about incentives for users to contribute resources

And we very briefly discussed some current vulnerabilities with web security. (By the way, I asked the security group what you should do given the current state of the world: they said they keep their passwords on a piece of paper.)

Hints: This paper tries to pull everything together.

Design patterns for successful systems – what can we learn from the various systems we've studied?

3 main areas: simplicity, performance and reliability. Often have tradeoffs: e.g., performance often makes a system more complex and less reliable. Of the three, you might think reliability was the most important! But simplicity means you can ship faster – draw curve of value vs. time to ship. Facebook clearly optimizes for shipping new features faster.

So huge value to getting better at managing your own work – what gets in the way?

Rate vs. level: which would you rather be? Knowledgeable, or able to learn quickly?

Optimizations are for a time and place: Moore's Law means that there are often new tradeoffs to tackle, and old tradeoffs to ignore. E.g., in past, no way that Facebook's architecture would have worked, but now it does. Christensen graphic.

Onto hints paper. Suggests several guidelines:

1. keep interfaces simple and stable, even at the cost of performance: RPC, mapreduce, centralized server design where you can get away with it, as in GFS Amazon would claim that eventual consistency is as good as you can get in practice (if you are latency-sensitive), so don't try to promise programmers more than you can deliver

reuse design pattern, not code: many of the systems do logging, consistent hashing, Paxos and/or 2 phase commit, state machine replication, reconciliation after disconnection. very few reuse code!

leave it to the client: memcache – client implements cache consistency; GFS – application error recovery; Dynamo's application-specific reconciliation

build one to throw one away? mapreduce arose out of seeing the same design pattern re-used many times. Spanner replaced Megastore replaced BigTable.

speed:

caches: not just cache data, but also cache results. most of the systems we've discussed cache lookup data, or fetch ahead blocks during a scan

hints: cached results that might be invalid, but that can be detected as invalid on use.
Chubby server connection IDs are cached, if wrong, server will let us know
GFS -> chunkserver locations
BigTable -> tablet server IDs

Hints are especially important in systems with failures, as any node might have failed between time the data was cached and the time it was needed

shed load off the fast path:
Facebook memcache cache hit
eventual consistency in Dynamo (don't make every lookup pay Paxos penalty)
Spanner updates can be very quick in the common case
Overprovision:
use enough memory so that index lookup can be simple and fast, as in GFS in knowing the chunkservers, Dynamo in knowing its member IDs
Also in Chubby: make sure clients don't overwhelm service

reliability: end to end, keep reliability simple – GFS
log updates: transactions, Bayou, BigTable, …
atomic operations: paxos, transactions and two phase commit, Bayou
restartable/idempotent: paxos, transaction log