Eventual consistency

[I find reading old papers to be kind of cool – you see exactly how far technology has changed. The Bayou folks had built what was essentially a pda or a smartphone, in the early 90's. But you can see from the paper that they needed to make a bunch of assumptions that turned out to be non-issues: e.g., that sending data packets over cell will be way too expensive.]

Finishing up cache coherence

At a high level, an astounding achievement. Semantics of one copy – all reads and writes are done in a total order, but in fact there's lots and lots of concurrency. Potentially very many copies of data, that are made into a single copy only as needed.

Anecdote: in our description, 3 states in write back cache coherence, but in AMD implementation in hardware, reported that the "simple" description has more than 180 state. Why? Lots of potential corner cases we only touched on – what happens if you get a cache lookup while the data is being fetched? Using a write buffer also makes things more complex.

***Scalability of directory information: to be serializable, need to have a bit per processor, or a list of processors caching each block. (why there isn't cache coherence of web pages.) We could try to scale by distributing the callback state as a tree of callbacks.

***We mentioned that NFS allows for transparent recovery when server or clients fail, using idempotent operations and TTL based coherence. Can we get transparent recovery with callbacks? (Recover by asking others what server state was.)

*** With callback state, a read or write must stall until the owner/read-copy replies. Is there a way to do non-blocking cache coherence? See multiversion concurrency control.

Distributed shared memory. I mentioned at the very end that we can implement cache coherence, provided we can catch reads/writes to potentially shared data – we can allow them to go forward in the common case that they aren't shared or that they are read-shared, but we need to be more careful with data that's written. And so we don't only need to do cache coherence in hardware, we can do it in software or even in the OS – emulating shared memory across a set of machines.

False sharing: what happens if two pieces of data are stored together? Means they will be fetched together – great if they are used at the same time. But what if one is modified, while the other is read-only! potential for poor performance: e.g., if one item is being written by one CPU, and the other being written by a different CPU.

One reason you need to do automatic cache coherence at a fairly fine-grained level – you have the potential for a lot of interference.

Started out by saying that RPC's and cache coherence are duals of each other – you can implement one in terms of the other.

Example:

Gradient computation – one cell has its edge condition set by its neighbors. So if you have 100 processors, split your computation up into squares that compute locally and only communicate with its neighbors. You can do this with RPC – compute your local state, and send it to your neighbors. Or you can do this with cache coherence. Which is more efficient?

They are going to be about the same, as long as we can ignore false sharing.
Each iteration, each CPU writes its boundary, and reads its neighbor's boundary.
The write invalidates the remote cached copy; the read fetches the new value.
With an RPC, you'd just fetch the new value. Matter of taste whether its easier to program if you can just treat your program as having shared memory.

Examples of systems that provide weaker consistency guarantees.
DNS: multiple replicas, needed for throughput – a huge number of clients reading data from the DNS servers at the same time. E.g., for the root DNS, several hundred replicas spread around the globe. Caching to reduce the load on the server.

How does update work? Update one replica, that replica replies, and in the background it propagates the change to the other replicas.

Implication: may not read your own writes! Update DNS, read the value, and if you happen to connect to a different replica, you'll see the old value!

Makes programming difficult: update, spin, check, spin, check, spin…

Might at least want local operations to appear in processor order – the order that the processor made them. Can you version #'s to fix this: when write, get back a token that you can use in future, please put the next read/write after this one.
Can you generalize to sets of related updates?

What about disconnected operation? How should google docs work?

Suppose one central server.

a) prefetch things you will need in future
b) allow others to change files while you are disconnected (no exclusive access)

c) log all changes to local disk – replay log preserves order of writes
d) apply log at server when connected
e) conflict resolution

Conflict resolution policy: changes appear in processor order, applied at time when notebook reconnects.

Is that sequentially consistent? Linearizable? Does it matter?

How can we resolve conflicts?

Ex: What happens on directory updates to different files by different users? Or two deletes followed by a create? What about different updates to different parts of the same file by different users?

This is roughly how CVS/SVN works – fetch a copy, put updates into central repository. Resolve conflicts by hand when merge is done.

Git/Bayou allow merges without a central master. In Bayou, idea was to be able to merge peers that are connected with each other, but not with a central master. In git, merge code between developers on a sub-branch, or across different sub-branches.

Another goal: support weakly connected operation – minimize amount of communication to do merge. For git, needed to support 10K developers simultaneously working on Linux – many are well connected, but many aren't.

You can think of git as a simplification of Bayou – it came 15 years later, so they could benefit from skipping the parts they didn't need.

Git: can split off a branch, or pull updates from any branch. Merge creates a new version, that integrates changes on both branches.

Git: Every clone keeps entire sequence of updates from start of repository, so can always know which updates have been applied into any specific branch.

Bayou: Goal is automatic resolution, so that everyone must use the same order of operations. Between concurrent operations, in git, assumes you resolve any conflicts, and so order of updates doesn't matter when there's no conflict.

Bayou: Conflict resolution is programmed and deterministic – each write is provided some code to run when a conflict occurs, e.g., for a calendar program, what to do if someone else booked the room while you were disconnected.

In Bayou, need to pick some order of a merge (should A or B get the room reservation). Want this order to be stable! Might be that some other node, C, has an update that gets put earlier, but that we haven't heard about yet.

OK to exchange updates with neighbor – this defines a specific order for A, B. However, neither is committed/stable until primary says that it is committed, since (primary gets to resolve concurrent updates, if they haven't been by an earlier merge).

Ultimately, the primary decides which order to commit, but it has to decide consistently with every other prior merge.

Example: A, B, C, D – A updates, merges with C. B updates merges with A. So now we have B' < A' on A, B. on C we have A'. D modifies, merges with C. D' < A'.
Merge D, A -> D' < B' < A'

In Bayou, this means we need to keep log of changes until write is committed, but after that we can discard the log.