Problem Set Solutions

CPS 212, 12/13/98

Here are the solutions to the problem set that was due on Friday 12/11. These questions are fair game for the final exam.

1. The TSAE protocol is identical to the Bayou protocol, except that it uses two-way exchanges (irrelevant) and it commits using the ack stamp approximation to acknowledgment vectors, rather than a primary commit protocol (relevant). There is one other important distinction, which I had blurred in class, and which the question was intended to bring out. In class, I had said that a timestamp vector element \( T_{SA}[i] \) is the accept stamp of the latest update originating from site \( i \) that \( A \) has received. This is OK for Bayou, but not for TSAE. If site \( i \) stops generating updates, then the corresponding element of the TS vectors on other nodes will never advance, and updates originating from other sites will never commit. The best solution is to follow traditional vector clock dogma and set \( T_{SA}[i] \) to be the value of \( i \)'s logical clock at the time of \( A \)'s last (transitive) contact from \( i \). In this way, the clocks and acknowledgement stamps will continue to advance as long as \( i \) continues to participate in anti-entropy exchanges (which it will as long as it is up), even if \( i \) is not generating any updates.

Most of you had apparently assumed (reasonably) that timestamp vectors were being managed exactly as vector clocks all along, and said so, presumably after examining the slides to reconstruct just exactly what were the distinctions between the Bayou and TSAE protocols. I gave full credit for these answers.

2. Consider first the tentative updates. Every node retains tentative updates in its update log in causal accept stamp order. In an anti-entropy exchange, the sender transmits updates in this order, with the oldest update first. The receiver retains the existing causal accept stamp order both for the new updates and for the updates it already has: it never reorders either stream of updates, but only merges the two streams. This means that the receiver need only roll back to the insertion point in its log (easily determined from the accept stamp of the first update transmitted in the exchange), and then roll forward to merge, selecting the next update to apply by choosing the earlier of the next update in each stream. This follows directly from the causal order of tentative updates: once a causal order has been assigned, it never changes.

Now consider the committed updates. The sender may notify the receiver of new committed updates, or it may inform the receiver that tentative updates it already holds have committed. In either case, the committed updates are ordered before any tentative updates, and the sender transmits the committed updates or notifications in their final CSN order before sending any tentative updates. Thus the rollback-once-and-merge property is preserved on the receiver.
3. The key point here is that Bayou requires a complete replica at each site in order to execute its dependency checks and merge procedures, which have access to the entire database state. In contrast, Coda determines currency of each file independently. Thus a Coda site can act as a file-grained cache, while a Bayou site cannot in general function as a cache. The tradeoff is that a Bayou application can define an appropriate notion of just what is an “update conflict”, whereas Coda cannot distinguish non-conflicting updates to the same file, and it does not recognize conflicting updates to multiple files, i.e., it assumes that files are completely independent.

4. A node receiving a write notice for a dirty page must avoid confusing its own (new) updates with the (old) updates covered by the write notice. The node does not need to generate the deltas (diff) immediately, but the diff must be completed before the remote deltas are applied to the page, so there is little to be gained by delaying it.

The old updates must be separated from the new so that other nodes can determine how to order the updates later. To see why, consider the following scenario involving variables $x$ and $y$ on the same page $p$. Node A sets $x = 1$ under the scope of lock X, while node B modifies $y$ under the scope of lock Y. Page $p$ is now dirty on both A and B. Suppose now that B receives the write notice for $p$ from A, e.g., perhaps because B acquires a third lock Z (transitively) from A. B must generate its deltas for $p$ on receiving the write notice. To see why, consider what could happen if B fetched and applied the new value of $x = 1$ from A before generating its deltas. Suppose now that A, still holding lock X, does the following: set $x = 2$, acquire lock Y from B, and read the value of $x$. What value will be returned for $x$? The correct value is 2, but the system will return the old value $x = 1$.

Here is why the system returns the old value for $x$. B will send a write notice for $p$ to A with lock Y, and A will request the corresponding deltas before reading $x$. If B has not already generated its deltas, it must generate them when the request arrives. When it diffs the current state of $p$ against its old twin, which was taken before the delta for $x = 1$ arrived from A, the resulting deltas will include $x = 1$. A fetches these deltas from B and applies them to $p$, overwriting its more recent write of $x = 2$. This causes A to return the old value for $x$ on the subsequent read.

Here’s how the problem is fixed by generating deltas for $p$ when receiving a write notice. If B diffs before applying $x = 1$, its diffs will include only the updates it actually made (the update to $y$). This will yield the correct behavior. It works even if B subsequently modifies $y$ again or perhaps some other variable on $p$ before passing lock Z to A. In this case, B will take a new twin and later generate a second set of diffs containing the new updates, but not including the $x = 1$ value it received from A.

5. The answer here is that a pessimistic concurrency control scheme will block the second transaction “forever”, whereas the Thor-like optimistic concurrency control scheme will repeatedly abort (restart) it “forever”. Neither will make forward progress until the two-phase commit unsticks, but the optimistic scheme will do a lot more work to stand still.