A Technical Perspective of CausalMesh

Philip A. Bernstein
Microsoft Research
philbe@microsoft.com

Replication is an important abstraction in many layers of computer systems: processor caches, main memory, file systems, database systems, and key-value stores. In all these cases, copies of the same data item must exhibit some form of consistency. It could be a weak one, such as *value convergence*: "if all write operations to a data item x stopped, then all copies of x would converge to the same value." This says nothing about the sequence of writes—only that the last write that modifies each data item is the same everywhere.

A stronger form of consistency is *consistent write ordering*: each application behaves as if writes execute in the same order on all copies. Thus, if an application issues a sequence of reads of data item x on different copies, each read returns a value that is no earlier than the previous value it read (i.e., it never sees time moving backward).

Consistency also applies to values of different data items. Consider a database that is replicated at servers S and S', and is accessed by applications A, B, and C. Suppose S executes the sequence of operations $w_A[x]$, $r_B[x]$, $w_B[y]$, $r_C[y]$, where r is a read, w is a write, x and y are data items, and the subscript denotes who issued the operation.

After $r_C[y]$ executes, suppose C executes $r_C[x]$ at server S'. Note that the value that $w_B[y]$ wrote could depend on the value that $r_B[x]$ read from $w_A[x]$. Therefore, even though $r_C[x]$ executes at a different server, to avoid seeing time move backward we expect $r_C[x]$ either reads the value written by $w_A[x]$ or a later value of x, which could happen if x was written again at S' after $w_A[x]$ and before $r_C[x]$. This is an example of causal consistency.

More generally, a replicated database is *causally consistent* if it behaves as if *dependent operations* execute in the same order on all copies. *Dependent operation* is defined as follows: An operation is dependent on previous operations of the same application. A read is dependent on the write from which it read. And dependencies are transitive: if q depends on p and p depends on o then q depends on o.

CasualMesh ensures causal consistency and value convergence in distributed, replicated servers. It targets key-value cache servers, but the algorithm is more general than that. It applies to any replicated database where the application can change the server it is querying.

Its challenging goal is to allow an application's successive read and write operations to execute at different servers, without ever rejecting an operation or waiting for the server to be synchronized with others. This is hard, because the delayed processing of a write at some servers might lead to a violation of causal consistency. In our running example, if it takes too long for $w_A[x]$ to propagate to S', then $r_C[x]$ might execute before $w_A[x]$, thereby violating causal consistency.

CausalMesh accomplishes its goal by using a new mechanism called a *dual cache*. Its consistent part, called *C-cache*, contains values that have been written to all servers. Its inconsistent part, called *I-cache*, contains values that may not have been written to all servers.

Servers are totally ordered in a ring. When a server executes a write operation, it stores the new value in its I-cache and then forwards the write to the next server in the ring. The last server to receive the write knows the write is consistent, so it stores the write in its C-cache and can optionally report that fact to other servers.

An application A's read operation always executes on a server's C-cache. If A's later operations execute at a different server, say S', A knows that S' executed all the writes that A read earlier. But some of those writes might be in S''s I-cache, because S' does not yet know those writes have propagated everywhere.

To solve this problem, A keeps a list of its dependencies, called *dep*, and a list of its writes, called *local*. When it executes a read at a different server, say S', it *integrates* its dependencies by telling S' that the writes A depends on are consistent. (They must be consistent, because an application in A's dependency chain previously read them.) If any of those writes are in S'''s I-cache, then S' moves them to its C-cache. This ensures that A reads from a causally consistent server. When A executes a write at a different server S', it integrates its dependencies and adds writes in *local* that S' has not seen to S'''s I-cache. Integration is why CausalMesh can process reads without ever waiting for a server to be synchronized with others.

A remaining problem is conflicting writes. In our running example, when C moves from S to S', C tells S' that $w_A[x]$ is consistent. We do not want S' to overwrite another consistent write that executed later than $w_A[x]$. To determine this, CausalMesh uses a standard mechanism, vector clocks, to resolve conflicting updates by determining which executed later. This ensures value convergence.

In addition to atomic reads and writes, CausalCache also supports atomic read-write transactions that satisfy a weak form of snapshot isolation. To learn about this and other interesting details of *CausalMesh*, read on.