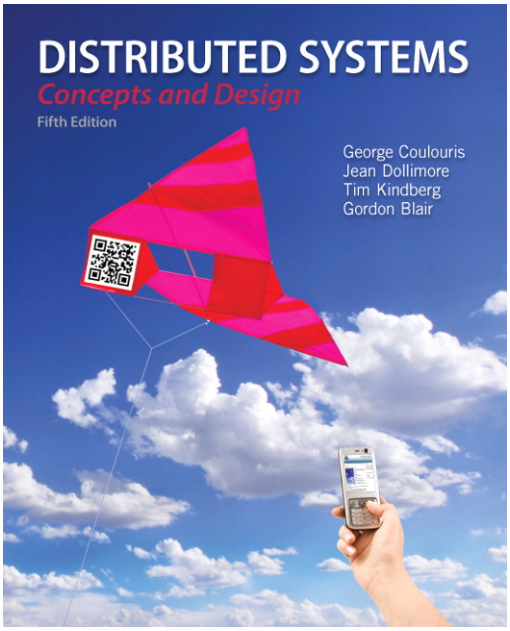
Credit: Distributed Systems: Concepts and Design, 5th Edition

George Coulouris, Cambridge University
Jean Dollimore, Formerly of Queen Mary, University of London
Tim Kindberg, matter 2 media
Gordon Blair, Lancaster University

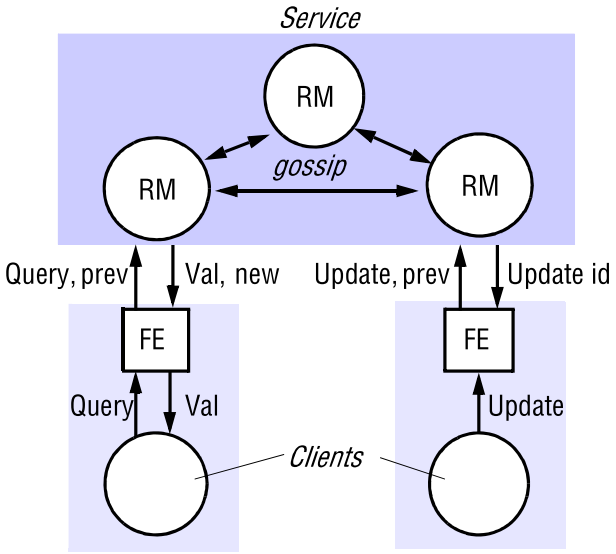


18.4 Case studies of highly available services: The gossip architecture, Bayou and Coda

In this section, we consider how to apply replication techniques to make services highly available. Our emphasis now is on giving clients access to the service – with reasonable response times – for as much of the time as possible, even if some results do not conform to sequential consistency. For example, the user on the train described at the beginning of this chapter may be willing to cope with temporary inconsistencies between copies of data such as diaries if they can continue to work while disconnected and fix any problems later.

In Section 18.3, we saw that fault-tolerant systems transmit updates to the replica managers in an 'eager' fashion: all correct replica managers receive the updates as soon

Figure 18.5 Query and update operations in a gossip service



as possible and they reach collective agreement before passing control back to the client. This behaviour is undesirable for highly available operation. Instead, the system should provide an acceptable level of service using a minimal set of replica managers connected to the client. And it should minimize how long the client is tied up while replica managers coordinate their activities. Weaker degrees of consistency generally require less agreement and so allow shared data to be more available.

We now examine the design of three systems that provide highly available services: the gossip architecture, Bayou and Coda.

18.4.1 The gossip architecture

Ladin *et al.* [1992] developed what we call the *gossip architecture* as a framework for implementing highly available services by replicating data close to the points where groups of clients need it. The name reflects the fact that the replica managers exchange 'gossip' messages periodically in order to convey the updates they have each received from clients (see Figure 18.5). The architecture is based upon earlier work on databases by Fischer and Michael [1982] and Wuu and Bernstein [1984]. It may be used, for example, to create a highly available electronic bulletin board or diary service.

A gossip service provides two basic types of operation: *queries* are read-only operations and *updates* modify but do not read the state (the latter is a more restricted definition than the one we have been using). A key feature is that front ends send queries and updates to any replica manager they choose, provided it is available and can provide reasonable response times. The system makes two guarantees, even though replica managers may be temporarily unable to communicate with one another:

Each client obtains a consistent service over time: In answer to a query, replica managers only ever provide a client with data that reflects at least the updates that the

client has observed so far. This is even though clients may communicate with different replica managers at different times, and therefore could in principle communicate with a replica manager that is 'less advanced' than one they used before.

Relaxed consistency between replicas: All replica managers eventually receive all updates and they apply updates with ordering guarantees that make the replicas sufficiently similar to suit the needs of the application. It is important to realize that while the gossip architecture can be used to achieve sequential consistency, it is primarily intended to deliver weaker consistency guarantees. Two clients may observe different replicas even though the replicas include the same set of updates, and a client may observe stale data.

To support relaxed consistency, the gossip architecture supports *causal* update ordering, as we defined it in Section 15.2.1. It also supports stronger ordering guarantees in the form of *forced* (total and causal) and *immediate* ordering. Immediate-ordered updates are applied in a consistent order relative to *any* other update at all replica managers, whether the other update ordering is specified as causal, forced or immediate. Immediate ordering is provided in addition to forced ordering, because a forced-order update and a causal-order update that are not related by the *happened-before* relation may be applied in different orders at different replica managers.

The choice of which ordering to use is left to the application designer and reflects a trade-off between consistency and operation costs. Causal updates are considerably less costly than the others and are expected to be used whenever possible. Note that queries, which can be satisfied by any single replica manager, are always executed in causal order with respect to other operations.

Consider an electronic bulletin board application, in which a client program (which incorporates the front end) executes on the user's computer and communicates with a local replica manager. The client sends the user's postings to the local replica manager and the replica manager sends new postings in gossip messages to other replica managers. Readers of bulletin boards experience slightly out-of-date lists of posted items, but this does not usually matter if the delay is on the order of minutes or hours rather than days. Causal ordering could be used for posting items. This would mean that in general postings could appear in different orders at different replica managers but that, for example, a posting whose subject is 'Re: oranges' will always be posted after the message about 'oranges' to which it refers. Forced ordering could be used for adding a new subscriber to a bulletin board, so that there is an unambiguous record of the order in which users joined. Immediate ordering could be used for removing a user from a bulletin board's subscription list, so that messages could not be retrieved by that user via some tardy replica manager once the deletion operation had returned.

The front end for a gossip service handles operations that the client makes using an application-specific API and turns them into gossip operations. In general, client operations can either read the replicated state, modify it or both. Since in gossip updates purely modify the state, the front end converts an operation that both reads and modifies the state into a separate query and update.

In terms of our basic replication model, an outline of how a gossip service processes queries and update operations is as follows:

1. *Request*: The front end normally sends requests to only a single replica manager at a time. However, a front end will communicate with a different replica manager when the one it normally uses fails or becomes unreachable, and it may try one or more others if the normal manager is heavily loaded. Front ends, and thus clients, may be blocked on query operations. The default arrangement for update operations, on the other hand, is to return to the client as soon as the operation has been passed to the front end; the front end then propagates the operation in the background. Alternatively, for increased reliability, clients may be prevented from continuing until the update has been delivered to $f + 1$ replica managers, ensuring that it will be delivered everywhere despite up to f failures.

2. *Update response*: If the request is an update, then the replica manager replies as soon as it has received the update.

3. *Coordination*: The replica manager that receives a request does not process it until it can apply the request according to the required ordering constraints. This may involve receiving updates from other replica managers, in gossip messages. No other coordination between replica managers is involved.

4. *Execution*: The replica manager executes the request.

5. *Query response*: If the request is a query, then the replica manager replies at this point.

6. *Agreement*: The replica managers update one another by exchanging *gossip messages*, which contain the most recent updates they have received. They are said to update one another in a *lazy* fashion, in that gossip messages may be exchanged only occasionally, after several updates have been collected, or when a replica manager finds out that it is missing an update sent to one of its peers that it needs to process a request.

We now describe the gossip system in more detail. We begin by considering the timestamps and data structures that front ends and replica managers maintain in order to maintain update ordering guarantees. Then, in terms of these, we explain how replica managers process queries and updates. Much of the processing of vector timestamps needed to maintain causal updates is similar to the causal multicast algorithm of Section 15.4.3.