Replication and consistency

Table of contents

Replication

In distributed systems, **replication** normally refers to creating multiple copies of data.

There are two key reasons to replicate data:

1. To enhance reliability.
2. To improve performance.

[1, P. 356]

Replication can improve reliability by introducing redundancy. If one data store fails, replicated data stores can continue to serve requests.

Replication can also improve performance. Replicating data across different geographical locations can improve performance by ensuring data is closer to users. Replicating data can also improve performance by increasing the number of data stores available to handle requests, thereby increasing the overall capacity of the system [1, P. 356].

A major downside to replicating data is that it introduces the problem of keeping replicas up-to-date with each other (keeping the data consistent).

State vs operations

There are two main approaches to replication: state transfer and state machine replication.

State transfer refers to sending the replicated state from one node to the other when required.

State machine replication treats each replica as a state machine that will maintain consistency as long as it receives the same inputs and applies them in the same order as the primary. Operations are sent from the primary to the replicas and then applied by each replica.

CAP theorem

CAP stands for consistency, availability, and partition tolerance. The **CAP theorem** asserts that any networked shared-data system can have only two of the three desirable CAP properties at one time [2, P. 23].

In CAP theorem, **consistency** is defined to mean that all nodes see the same data at the same time [2, P. 23].

Note: The definition of consistency used in CAP theorem is different to the definition of consistency used in ACID.

Availability is defined to mean that each request to the system receives a response [2, P. 23].

Partition tolerance is defined as the ability to tolerate network partitions [2, P. 23].

Since partition tolerance is required when communicating over a network (partitions are always possible in a network), designers must choose between consistency (C) or availability (A). Often, databases are described as AP or CP depending on which property they forfeit in the event of a network partition.

In reality, partitions are rare. This has led Google to offer an "effectively" highly available *and* strongly consistent database—Google Spanner. In the case of a partition, Spanner will choose consistency over availability, but network partitions on Google's private network are so rare that users can assume that the system is CA (missing reference)[2, P. 23].

Consistency models

A **consistency model** is the contract between a process and a data store that guarantees certain properties [1, P. 359].

Strong consistency models

Strong consistency is where each read of data item x is guaranteed to see the latest write to x (across all data stores).

Strong consistency allows you to program as if the underlying data wasn't replicated.

The tradeoff with strong consistency is that the the system will not be available in the case of a network partition.

Strong consistency also results in higher latency than alternative models due to the cost of keeping data consistent (using e.g., transactions or synchronization variables) [1, P. 375].

There are various forms of strong consistency:

- **Linearizable consistency** is where all operations appear to have executed atomically and in an order that is consistent with the real-time ordering of operations.
- **Sequential consistency** is where all processes see the same order of operations as all other processes [1, P. 365].

The difference between linearizable consistency and sequential consistency is that linearizable consistency requires that the order of operations seen by the processes is the same as the real-time order that they were issued in.

Weak consistency models

Weak consistency trades strong consistency guarantees for improved availability and latency.

Eventual consistency is where a level of inconsistency is tolerated. Eventually consistent data stores converge towards identical copies of each other [1, P. 373]. There are many sub-categories of eventual consistency that are better specified, such as last-write-wins, but often they are grouped under the umbrella term of eventual consistency.

Client-centric consistency models

Client-centric consistency models are concerned with the consistency between the client and the data store (rather than consistency between data stores).

Monotonic reads guarantee that once a process reads a value for x, it will never read any older values for x [1, P. 378].

Monotonic writes guarantee that a write by a process on data item x is completed before any future writes on x by the same process [1, P. 379].

Read-your-writes consistency is where a replica is at least current enough to reflect any write operations made by process P when a read operation by P is made [1, P. 381].

Writes-follow-reads consistency is where write operations on a data item x by a process P following a previous read operation on x by P are guaranteed to take place on either the same version of x that was read by P or a more recent version of x [1, P. 382].

Consistency protocols

A **consistency protocol** describes an implementation of a consistency model [1, P. 396].

The most common consistency protocols are primary-based protocols and replicated-write protocols.

Primary-based protocols

In primary-based protocols, each data item has an associated primary that is responsible for coordinating write operations on that data item [1, P. 399].

Remote-write protocols

In **remote-write protocols**, the primary is a fixed remote server [1, P. 399].

Any updates to a data item are forwarded to that data item's primary, which then applies the update to itself and forwards the update to backup servers. The backup servers apply the updates and send an acknowledgement to the primary, which then sends an acknowledgement to the client that initiated the update [1, P. 399].

An alternative approach is to make writes non-blocking, and have a primary respond immediately to clients as soon as the primary has updated its own data store [1, Pp. 399-400].

Primary-backup protocols provide sequential consistency, since all processes will see the same ordering as the primary [1, P. 400].

Active replication

In active replication, operations (or state updates) are sent to each replica which then applies the operations to update its state [1, P. 401].

Active replication requires operations to be executed in the same order on all replicas. One way to achieve this is by using a sequencer server to order and assign a sequence number to each operation [1, P. 401].

Quorum-based protocols

Quorum-based protocols use voting to determine whether read and write operations can be completed. Clients must acquire permission from other nodes to either write or read replicated data items [1, P. 402].

A **read quorum** (R) is the minimum number of replicas required to agree to a read [3, P. 150].

A **write quorum** (W) is the minimum number of replicas required to agree to a write [3, P. 150].

One way to ensure that each read/write set contains at least one node with current data is by setting a read quorum and a write quorum of $(N/2)+1$ servers, where N is the number of replicas. In other words, a majority of the replicas must also agree to perform the read/write.

During each write operation, the updated replicas are given a new version number (which is the same number for each replica). The version number can then be used during future reads and writes to determine if a version is current or not. If all the nodes in a read set have the same version, then this is the most recent value, otherwise the highest version number is the most recent value [1, P. 402].

The values of R and W can be configured to change the reliability and performance of a system [3, P. 152].

There are many different variations of quorum-based protocols.

Chain replication

Chain replication works by replicating data across a chain of servers [4].

The servers are treated as a linked list with each node containing a link to the next node in the chain (its successor). One node is designated as the head and one node is designated as the tail. The head receives writes, which are then propagated through the chain. All reads go to the tail node. [4, P. 3].

Chain replication offers strong consistency since all reads are processed serially by the tail node [4, P. 3].

CRAQ

CRAQ (Chain Replication with Apportioned Queries) is a variation of chain replication where read operations are spread across all nodes in the chain, improving the overall read performance of the system [5, P. 2].

Object versions are identified with a monotonically increasing number. When a node receives a new version for an object, the node appends the object to its object list. If the node is not a tail node, it marks the version as dirty and propagates the new version to its successor node. If the node is the tail, it marks the version as clean and notifies other nodes that it has written the version by sending an ack. When a predecessor node receives an ack, it marks the corresponding version as clean [5, P. 3].

When a node receives a read request for an object it does one of two things:

- If the latest version is clean then the node returns the value.
- If the latest version is dirty then the node forwards the request to the tail [5, P. 3].

CRAQ works best for read-mostly workloads [5].

References

[1] A. Tanenbaum and M. van Steen, *Distributed Systems*, 3.01 ed. Pearson Education, Inc., 2017.

[2] E. Brewer, “CAP Twelve Years Later: How the ’Rules’ Have Changed,” *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[3] D. K. Gifford, “Weighted voting for replicated data,” pp. 150–162, 1979.

[4] R. Van Renesse and F. Schneider, “Chain Replication for Supporting High Throughput and Availability.,” 2004, pp. 91–104.

[5] J. Terrace and M. J. Freedman, “Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads,” in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USA, 2009, p. 11.