CMPSCI 677 Operating Systems **Spring 2013**

Lecture 16: March 26

Lecturer: Prashant Shenoy *Scribe: Nicholas Monath*

16.1 Replication

16.1.1 Types of Replication

Replication can be one of two types:

Data Replication Multiple replicas (copies) of data are stored on multiple machines. All replicas represent the same data. Typically this refers to data stored on disk, but one could also replicate memory state as well.

Computation Replication Same application runs on multiple machines, e.g. code is replicated on multiple machines.

16.1.2 Reasons for Replication

Reliability If a machine holding the data crashes, then the whole system will not have access to the data. Having replicas of data will ensure high availability of data Also protects against data corruption, e.g. if one machine's data becomes corrupted it can retrieve an uncorrupted copy of the data from another machine.

Performance Replication can be used to scale a distributed system, and serve large number of requests. Geo-Replication gives better latencies, by serving clients using closest replica to those clients.

16.1.3 Replication Issues

Key questions in system design regarding replication:

1. When to replicate? *Should it be done statically or should it be done dynamically when the load of the system exceeds a certain threshold*
2. How many replicas to create? *Number of replicas should be sufficient to handle the volume of requests coming into the system*
3. Where should the replicas be located? *Depends on where the users of the system are located geographically*

Consistency – Typically issues concerning replication are application specific. It is up to the application designer to choose what kind of replication is needed. One of the major issues of replication is Consistency. The system should guarantee that all replicas are consistent and represent same data. There are notions in consistency known as Strong Consistency, and Weak Consistency, based on the guarantee of consistency given to the user.

16.2 Replication & Consistency

16.2.1 Design Choices

There are two places where one can enforce consistency:

Approach 1 Application will be responsible for replication, and consistency. This approach offers flexibility for application designers to choose their own mechanisms. In this case, designer will be responsible for consistency, which is an added burden for designer.

Approach 2 System (middleware) takes care or replication, and consistency. This simplifies the application development but makes object-specific solutions harder

16.2.2 Replication and Scaling

Replication and caching are both used for scaling systems; the two are related but have a few distinct differences. Caches are a performance optimization; it does not take place at the application layer. When a request comes in to the system, the cache is searched for the data required to process that request. If the data is present in the cache, the request is serviced with the data from the cache; otherwise, the data is fetched from the system as if there were no cache. Note that the cache is distributed and the consistency of the cache will need to be enforced.

There are several things to keep in mind regarding consistency guarantees. Suppose we have a data object that is replicated N times. These consistency policies depend upon applications and their requirements. The consistency technique you select depends on the read and write frequency of the data. If the read frequency is much less than the write frequency, we pay an unnecessarily high overhead to maintain consistency. Each write requires messages to update the all the copies of the data. However the data item is rarely read and so the changes are rarely 'noticed' in the system. If the write frequency were much less than the read frequency, than the overhead is not unnecessary.

16.3 Consistency Semantics

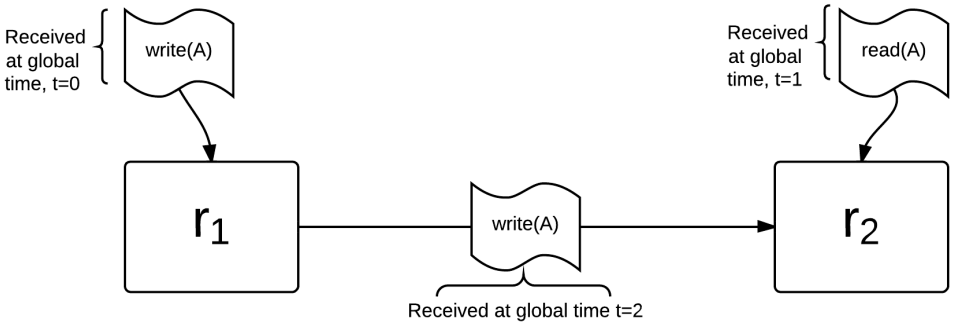
16.3.1 Data-Centric Consistency Models

The consistency models described below are what is known as Data-Centric. They enforce consistency from the point of view of the data store. They are a contract between processes and the data store. If processes obey certain rules, data store will work correctly. In servicing a read operation, the models return the value of the last write the requested data item. One model differs from another in the way the 'last' write is defined.

16.3.2 Strict Consistency

Strict Consistency policy assumes that all machines in the system have global clock, and imposes that every write operation is reflected immediately on every other replica in the system. That means that any read always returns the result of the most recent write. It also assumes that a write is immediately visible to all processes. This is a strong assumption and is difficult to achieve in real systems because not all processes are connected to the same replica and so network delays can be variable.

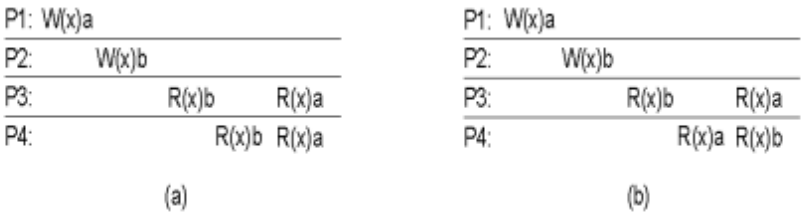
An example of why this is difficult to guarantee is: Suppose we have two replicas r_1 and r_2. A write on data item A comes in to r_1, which we'll call $write(A)$. It is r_1's job to forward $write(A)$ to r_2. However, it could be the case that a read request comes into r_2 before it receives the forwarded $write(A)$ from r_1, this would not satisfy strict consistency. This is shown in the below figure:



16.3.3 Sequential Consistency

This is a weaker form of consistency compared to Strict Consistency. Sequential Consistency assumes that all operations are executed in a sequential order. All processes agree on same interleaving, while preserving their own program order. There is no global clock in this policy.

In the figure below, the ordering (a) is sequentially consistent, but ordering (b) is not because in Process 3 in (b) the read $R(x)b$ happens before $R(x)a$ and in Process 4 in (b) the read $R(x)b$ happens after $R(x)a$.



16.3.4 Linearizability

Linearizability is stronger than sequential consistency and weaker than strict consistency. It assumes sequential consistency and makes the added restriction that for two events x and y, if the timestamp of x, $TS(x)$, is less than $TS(y)$, then x should precede y in the sequence of events. Note that this differs from *serializability*, which is used at a transaction level. Linearizability is at a finer granularity, the read/write level.

For example, consider the following three processes. As we can see that the variables in one process are referenced by the others, we know that the processes are passing messages:

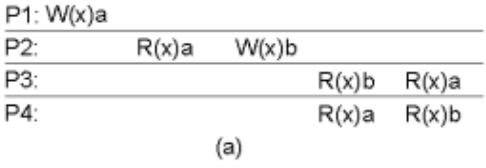
Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

Below are four valid orderings of the operations in the processes:

x = 1;	x = 1;	y = 1;	y = 1;
print ((y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print(y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011 (a)	Signature: 101011 (b)	Signature: 110101 (c)	Signature: 111111 (d)

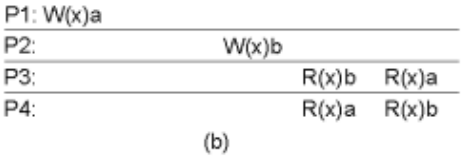
16.3.5 Causal Consistency

Similar to linearizability, this policy says that causally related writes must be seen by all processes in the same order, but concurrent writes may be seen in different orders on different machines. The following is an example of a permitted and not permitted ordering of events in two processes:



(a)

Not permitted



(b)

Permitted

16.3.6 Other Models

FIFO Consistency A weak form of consistency that respects the program order inside of a process. Writes from a process are seen by others in the same order. Writes from different processes may be seen in a different order (even if causally related). This relaxes causal consistencies. FIFO consistency requires that writes from a process should be seen in the same order.

Using Granularity of Critical Sections This policy uses granularity of critical section, instead of individual read/write. Instead of synchronizing on a per write basis, synchronize at a given point in time. Before entering (or exiting) a critical section, pause and make sure that everything is consistent. Referred to as Entry and Release consistency.

16.3.7 Eventual Consistency

Eventual consistency is a weak consistency policy that makes no guarantees about consistency other than that over time all replicas will see all updates and reach a consistent state. This means that there is not guarantee any consistency while the updates are being made. Examples include DNS and NIS.

Eventual consistency is often implemented with pairwise exchange. A replica randomly chooses another replica to give changes to.

A canonical problem with eventual consistency is *write-write conflicts*. The same data item is updated in two different places at more or less the same time and when the changes propagate they conflict. The user must resolve the conflicts.

16.3.8 Client-centric Consistency Models

Rather than ordering reads and writes from the perspective of the data store, order them from the perspective of the client. Below are four different client-centric models:

Monotonic Reads: Once read, subsequent reads on that data items return same or more recent values.

Monotonic Writes: A write must be propagated to all replicas before a successive write by the same process. Like FIFO consistency.

Read your writes: $read(x)$ always returns $write(x)$ by that process. This means the version of a data item written by a process is the version that is read by that process.

Writes follow reads: write(x) following read(x) will take place on same or more recent version of x

16.3.9 Epidemic Protocols

Epidemic protocols is an implementation of eventual consistency. Used in *Bayou* system from Xerox PARC. A system that is collection of weakly connected replicas – e.g. mobile devices. Treat update as an *infectious disease*. Perform pairwise-exchange of updates – try to "infect" other replicas as quickly as possible. An **infective store** is a data store with an update it is willing to spread. A **susceptible store** is a data store that is not yet updated.

16.3.9.1 Spreading of an Epidemic

Anti-Entropy A server picks another server randomly and exchange all the updates with that server. Updates are made either by Push-Only or Pull-Only, or both approaches. In Push-Only, server pushes or sends the data to the receiver server but does not receive any data, whereas in Pull-Only receiver server extracts data from sender server, but does not send any. A pure push-only approach will not spread updates quickly, because it is a randomized protocol, and there can be instances where few machines will never

see the update. This can be solved by used by implementing both Pull-only or initial push with Pull-only approaches.

Rumor Mongering (gossip) It is a randomized protocol. A server, as soon as it sees an update, randomly picks a machine and pushes the update. If that receiving machine has already seen this update, server will stop spreading the updates with a probability of 1/k. Rumor mongering protocol will not guarantee that all replicas will receive updates. Chances of a replica not receiving the update is exponential, which is s= e $^{(k+1)(1-s)}$, where "s" is the probability of not receiving the update.

16.3.9.2 Removing Data

If a copy of data has been deleted at a replica, machines should be able to distinguish between deleted copy and no copy. This can be solved by marking the copy to be deleted as "deleted", and not actually deleting it, which is analogously called as "issuing death certificate". But there should be some kind of clean up that actually deletes the copy.