Consistency and Replication

Reasons for Replication

Two primary reasons for replicating data:

- To increase the Reliability of a system (A server might crash, simply switch to another replica), better protection against data corruption
- Other reason is performance. When a distributed system needs to scale in terms of size or in terms of geographical area it covers. Example: when more processes needs to access data and having replication can divide that workload. In terms of geographical area, we may need to provide a copy of a document by accessing a server in the proximity.

Problems with replication:

- Having multiple copies may lead to consistency problems. Modifying a copy makes it different to the rest of copies. Solution: No local copies, a server can validate cached copies of a data $\rightarrow$ may degrade performance

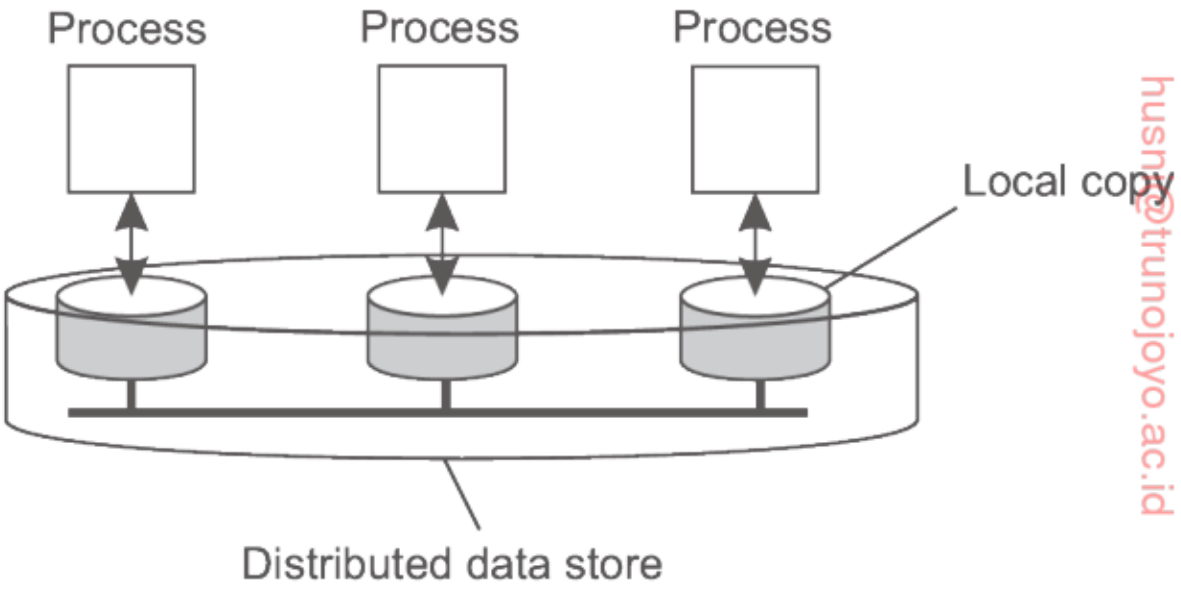
Replication as scaling technique

On one hand, scalability problems can be applying cache and replication, leading to improved performance. On the other hand, to keep all the copies consistent requires global synchronization, which is inherently costly in terms of performance.

In many cases, the only real solution is to relax the consistency constraints. In other words, if we can relax the requirement that updates need to be executed as atomic operations, we may be able to avoid global synchronizations, and thus may gain performance. The price may be that all copies may not always the same everywhere.

Data-centric Consistency Models

Shared memory, shared (distributed) database, or a (distributed) file system $\rightarrow$ **Data Store**.
A Data store may physically distributed across multiple machines. Since it is distributed, each process is assumed to have local copy of the entire data store.



A **consistency model** is essentially a contract between processes and the data store. It says if the process agree to obey certain rules, the store promises to work correctly.

Continuous Consistency

There is no such thing as best solution to replicating data. Replicating data poses consistency problems. So we have to loosen consistency. No general rules for loosening consistency.

3 independent axes for defining inconsistencies:

1. Deviation in numerical values between replicas
2. Deviation in staleness between the replicas (which replica is a fresh copy)
3. Deviation with respect to ordering of update operations.

Deviation in numerical values

Example: Stock market records, deviation between replicas should not be more than $2. This is **absolute numerical deviation**. Alernatively, **relative numerical deviation** can be defined like no more than 5% difference.

Can also be in terms of numerical updates. One replica should not be ahead of the other in terms of commits.

Deviation in Staleness

Staleness refers to the last time a replica was updated. Can provided old data as long as it is not *too old*.

Deviation in ordering of updates

Ordering can be different, as long as the differences remain bounded. Consequence could be rolling back and applying in a different ordering.

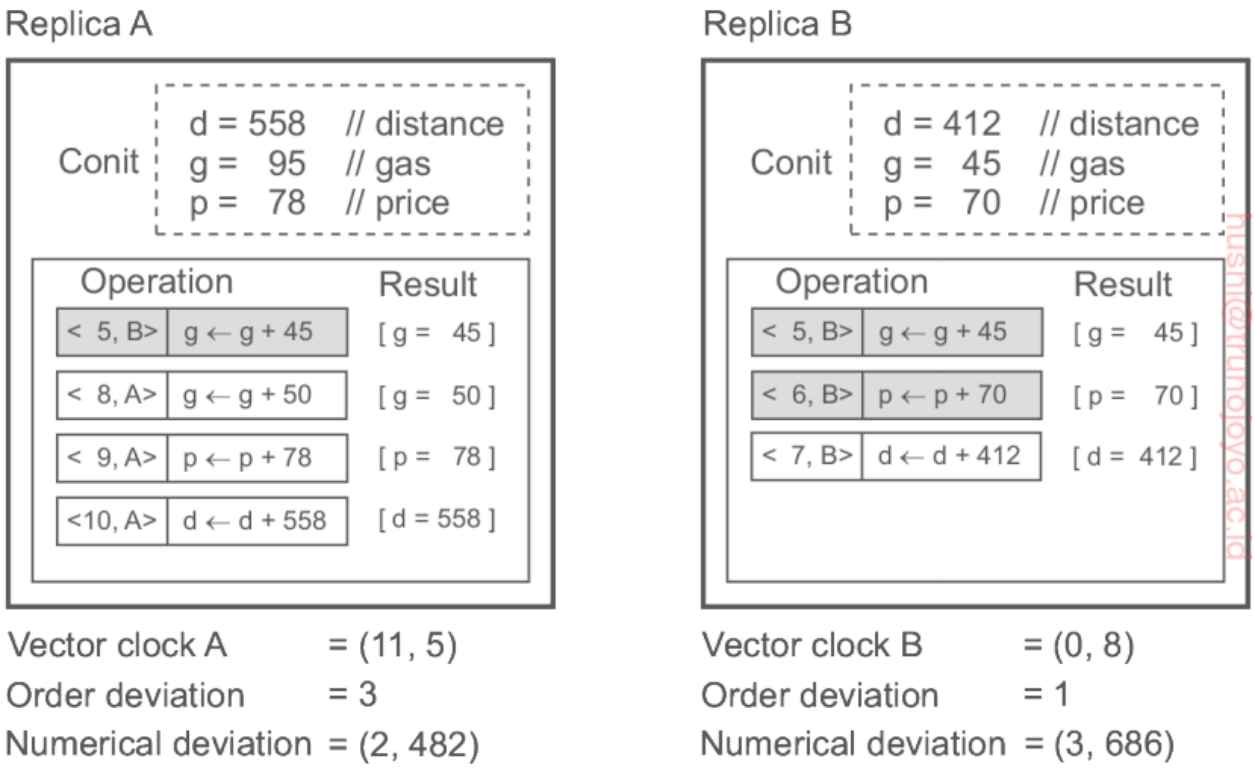
Notion of a Conit

Conit = **Consistency unit**.

Specifies a unit over which consistency is to be measured. Example: in our stock-exchange, conit could be a single stock.

The task of the servers is to keep the conit "consistently" replicated. To this end, each server maintains a **2D vector clock**.

$< T, R >=$ An operation that was carried out by replica R at logical time T.



Items in gray are committed to their respective local stores and cannot be rolled back. Three tentative operations at A is the case of **order deviation** of value 3. B has order deviation of 1.

Clock = (C_A, C_B)
Numerical Deviation = $(operations,\ sum\ of\ the\ values\ deviated)$

Now we can use these notations to restrict specific deviations.

Issues with conits:

- In order to enforce consistency, we need to have protocols.
- Program developers need to specify the consistency requirements for their applications.

Consistent Ordering of Operations

Sequential Consistency

$W_i(x)a \rightarrow$ Process P_i writes value a to data item x.

$R_i(x)b \rightarrow$ Process P_i writes reads x and returns b.

Figure 7.4: Behavior of two processes operating on the same data item. The horizontal axis is time.

$P2$ first read NIL because the change of x to its local data store was not propagated, it took some time but it came later.

A data store is said to be sequentially consistent when it satisfies:

The result of any execution is the same as if the operations by all processes on the data store was executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

What this definition means that when processes run concurrently on different machines, any valid interleaving of read and write operations is acceptable behaviour, but all processes see the same interleaving of operations.

Causal Consistency

The **causal consistency model** represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally (acting as a cause) related and those that are not.

If event b is caused or influenced by an earlier event a, causality requires that everyone else must first see a, then see b.

Example: Process P_1 writes a data item x. Process P_2 reads data item x and writes y. These may be causally related since computation of y may have been depended on x.

Operations that are not causally related are said to be **concurrent**.

Condition:

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in different order on different machines.

Grouping Operations

Mutual exclusion, Critical section, synchronization variables or locks.

The Following criteria should be met:

- Acquiring a lock can only succeed only when all updates to its associated shared data have completed. (Other processes are done with the memory location where it wishes to perform

operations).

- Exclusive access to a lock can only succeed if no other process has exclusive or non-exclusive access to that lock.
- Non exclusive access to lock is only allowed when any previous exclusive access to that lock has been completed.

Consistency vs Coherence

A **consistency model** describes what can be expected with respect to that set when multiple processes concurrently operate on that data. The set is then said to be consistent if it adheres to the rules described by the model.

When data consistency is concerned with a set of data items, **coherence models** describe what can be expected to hold for only a single data item. In this case, we assume that a data item is replicated; it is said to be coherent when the various copies abide to the rules as decribed by its associated consistency model. In effect, it means that in the case of concurrent writes, all processes will eventually see the same order of updates taking place.

Eventual Consistency

If no updates take place in a long time, all replicas will gradually become consistent, that is, having exactly the same data stored. This form of consistency is called **eventual consistency**.

Example: Web pages that do not update for a long time which users might find tolerable or they could be unware that the page has been updated, DNS Servers and CDNs (no write-write conflicts).

In the absence of write-write conflicts, all replicas will converge toward identical copies of each other. Eventual consistency essentially requires only that updates are guaranteed to propogate to all replicas. Often cheap to implement. Write-write conflicts can easily be solved when there are only a handful processes allowed to write, on global write will declared as the *winner*.

Client-centric consistency models

If a user accesses different replicas eventual consistency will fail. This problem can be alleviated with the help of **client-centric consistency**.

In essence, client centric consistency provides guarantees *for a single client* concerning the consistency of accesses to a data store by that client. No guarantees are given for concurrent access by different clients.

Monotonic Reads

A data store is said to provide montonic-read consistency if the following condition holds:

If a process reads the value of a data item x, any successive read operation on x by that process will always return the same value or a more recent value.

Once a process has seen a value of x, it will never see an older version of x.

Example: Consider a mailbox. If a user opens his mailbox in San Fransisco and later flies to New York and opens the mailbox again, monotonic-read guarantees that the messages that were opened in the mailbox in San Fransisco will also be in the mailbox when it is opened in New York.

Monotonic Writes

Many times, it is important that write operations are propogated in the correct order to all copies of the data store. The following condition holds:

A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

Thus, completing a write operation means that the copy in which successive write operation is performed reflects the effect of the previous write operation by the same process, no matter where the operation was initiated.

Resembles data-centric FIFO consistency meaning write operations by the same process are performed in the correct order everywhere.

Read your writes

The following condition holds:

The effect of a write operation by a process on x will be seen by the successive read operation by the same process on x.

In other words, a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

Its absence is sometimes experienced when updating web documents and subsequently viewing the effects, because of caching by the browser or the server. Read your writes guarantees that the cache will be invalidated when the page is updated, so that the updated file is fetched and displayed.

Replica Management

Placement of replicas:

- Placing server replicas
- Placing content replicas

Replica server placement is concerned with finding the best locations to place a server that can host a data store.

Content placement deals with finding the best server for placing content.

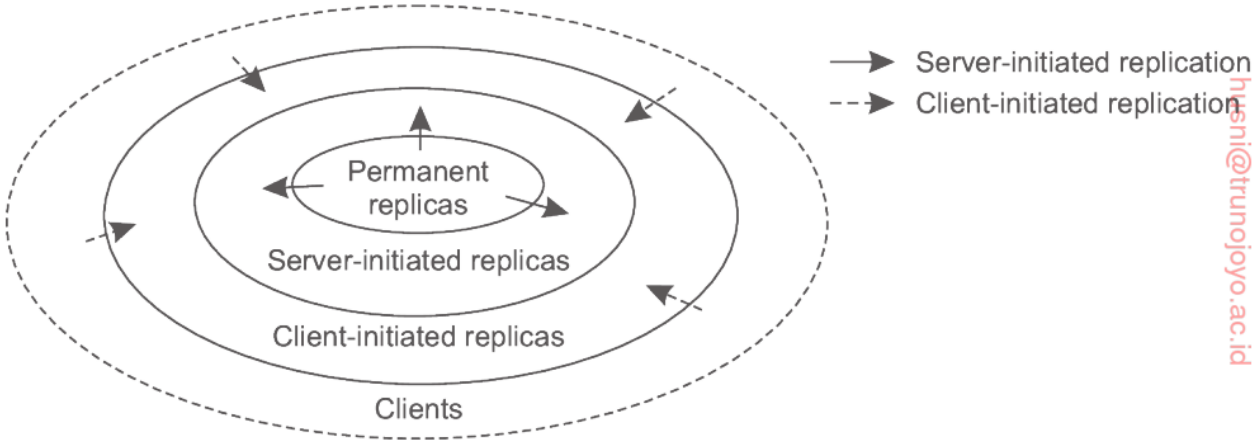
Finding the best server location

(Write it on your own, not gven much in the book, only that it is a commercial problem rather than optimization problem).

Content Replication and Placement

3 different types of replicas distinguished logically:

- Permanent Replicas
- Server-initiated replicas
- Client-initiated replicas



Permanent Replicas

- Can be considered as the initial sets of replicas that constitute a distributed data store.
- Number of permanent replicas are small.
- **Mirroring**: a web site is copied to a limited number of servers, called **mirror sites**, which are geographically spread across the internet. In most cases, the clients simply choses one of the various mirror sites from a list offered to them.
- **Shared-nothing architecture**: database is distributed and replicated across a number of servers, emphasising that neither disks nor main memory are shared by processors.

Server initiated Replicas

- Copies of a data store that exist to enhance performance, and created at the initiative of the owner of the data store.
- Are used for placing read-only copies close to clients.

Client initiated Replicas

- More commonly known as **client caches**. Client cache is a local storage facility used by the client to temporarily store a copy of the data it has just requested.
- Managing the cache is left entirely to the client.
- Are only used to improve access times to data.

- Caches can also be shared by the clients.
- Normally cache is placed on the same machine as client or on the same LAN as client.

Content Distribution

Also deals with the propagation of updated content to the relevant replicated servers.

State vs operations

What should be propagated? Three Possibilities:

- Propagate only a notification of an update
- Transfer data from one copy to another
- Propagate the update operation to the copies

NOTIFICATION OF AN UPDATE

This is what invalidation protocols do. Informs others that an update has taken place, and the data that they contain is no longer valid. Can only notify which part of the data has updated. Works best when there are many update calls compared to read operations

For example: if 2 updates happen with no read operation between them, then the first update will become useless.

TRANSFERRING DATA

Can occur when the read-to-write ratio is relatively high. The Probability that modified data will be read before another write is high.

PROPAGATE OPERATION

In this approach, no data odifications is transferred at all, instead it tells each replica which operation it should perform (and sending only the parameters values that the operation needs). This approach is called **active replication**, assumes that the each replica is presented with a process capable of actively keeping its associated data up to date by performing operations. Uses minimal bandwidth costs, provide the size of parameters is small.

Pull vs push Protocols

In **push based** or **server based protcols**, updates are propagated to replicas without those replicas even asking for the updates. Used between permanent and server initiated repliacs. Generally applied when strong consistency is required. Efficient in the sense that every pushed update can be expected to be of use to atleast one or more readers.

In **pull based protocols** or **client based protocols**, a server or client requests another server to send it any updates it has at that moment. Often used by client caches. Client can first check with the server if it has any updates on which it updates the content in the cache or otherwise it sends the data directly from the cache. Efficient when read-to-update ratio is low. **Important Issue**: Server needs to keep track of all client caches.

HYBRID SYSTEM

A **lease** is a promise by the server that it will push updates to the client for a specified amount of time. When the lease expires, the client needs to poll the server and pull in the modified data if necessary. An alternative is that the client requests for a new lease.

AGE-BASED LEASE

Given out on data items depending on the last time the item was modified. The underlying assumption is that data that has not been modified for a long time can be expected to remain unmodified for some time yet to come.

RENEWAL FREQUENCY BASED LEASE

A server will hand out a long lasting lease to a client whose cache often needs to be refreshed.

STATE BASED LEASE

When the server realises that it is gradually becoming overloaded, it lowers the expiration time of new leases it hands out to the clients. Now it will have to keep track of fewer clients as leases will expire quickly.

Unicasting vs Multicasting

In **unicast** communication, when a server that is part of the data store sends its update to N other servers, it does so by sending N seperate messages, one to each server.

In **multicasting**, the underlying network takes care of sending a message efficiently to multiple receivers. Can be efficiently combined with push based approach. Whereas unicast works well with pull based approach where there is only a single receiver.

Managing Replicated Objects

2 Issues:

- We need a means to prevent concurrent execution of multiple invocations on the same object. Access to internal data within an object should be serialized.
- Also we need to ensure that all changes to replicated state of an object should be same.

Both of the above tasks is handled by middleware.

In many cases, designing replicated objects is done by first designing a single object, locally locking it, and subsequently replicating it. The role of the middleware is that if a client invokes a method on a replicated object, the invocation is passed to the replicas and the object servers in the same order everywhere.

Consistency Protocols

A **consistency protocol** describes an implementation of a specific consistency model.

Continuous Consistency

Bounding numerical deviation

Current Value of x = v_i (Resulted from certain writes at a server S_i)

Actual Value of x = v

Then,

$$v - v_i \leq \delta_i$$

δ_i = upper bound

Writes submitted to a server S_i will need to be propagated to all other servers.

The Whole idea is when a server S_k notices that S_i has not been staying in the right pace with the updates that has been submitted to S_k, it forwards the writes from its log to S_i.

Bounding staleness deviations

$$RVC_k[i] = t_i$$

The above notation means that S_k has seen all the writes that have been submitted to S_i up to time t_i.

Simple Approach:

Whenever server S_k notices that $t_k - RVC_k[i]$ is about to exceed a spciefied limit, it starts pulling in writes that originated from S_i with a timestamp later than $RVC_k[i]$.

Note that in this case, a replica server is responsible keeping its copy if x up to date regarding writes that has been issued elsewhere.

Bounding ordering deviations

It is bounded by specifying the maximal length of the queue of tentative writes.

Approach:
When the length of this local queue exceeds a specified maximal length, at that point, a server no longer accept any newly submitted writes, but will attempt to commit tentative writes by negotiating with other servers in which order the writes should be executed.

Primary based protocols

In these protocols, each data item x has an associated primary, which is responsible for coordinating write operations on x. Now the primary can at a remote server or local.

Remote-write protocols

All write operations need to be forwarded to a fixed single server. Reads can be carried out locally. Such schemes are also known as **primary-backup protocols**.

A process wanting to perform a write operation on data item x, forwards that request to the **primary server** for x. The primary server performs the update on its local copy of x, and subsequently forwards the update to the backup servers. Each backup server performs the update as well, and send an acknowledgement back to the primary server. When all backups have updated their local copy, the primary sends an acknowledgement back to the intial process.

The update operation is implemented as a blocking operation. Alternative: nonblocking operations, as soon as the primary has updated its local copy, it returns an acknowledgement. After that, it tells the backups to perform the update as well.

Local-write protocols

The primary copy migrates between processes that wish to perform a write operation. Whenever a process wants to update data item x, it locates the primary copy of x, and subsequently moves it to its own location. Can only be acheived if a nonblocking protocol is followed by which the updates are propagated to replicas after the primary has finished.

Nonblocking approach: A central server can temporarily allow one of the replicas to perform a series of local updates. When the replica is done, the updates are propagated to the central server, from where they are distributed to other replica servers.

Replicated-write protocols

Write operations can be carried out at multiple replicas instead of one.

Active Replication

Each replica has an associated process that carries out the update operations. The operation is sent to each replica. However, it is possible to send the update.

PROBLEM: Operations need to carried out in the same order everywhere. What is need is a total-ordered multicast mechanism. A practical approach to accomplish this is by a means of a central coordinator, called a **sequencer**, which assigns a request a unique sequence number.

Quorum-based Protocols

Voting is used. Example: Consider a distributed file system and suppose that a file has been replicated on N servers. We could make a rule that stating that to update a file, a client must first contact $N/2+1$ servers and get them to agree to the update. Once they have agreed, the file is changed and a new version number is associated with it. The same happens wih read operation and the most recent file is returned.

When it was first introduced, to read a file of which N replicas exists, a client needs to assemble a **read qourum**, of N_R servers and to modify a file a **write quorum**, of N_W servers. With

$$N_R + N_W > N$$

$$N_W > N/2$$

The first is used to prevent read-write conflicts while the second one is used to prevent write-write conflicts.

Cache-coherence Protocols

Coherence detection: when inconsistencies are actually detected.

Classification of dynamic detection-based-protocols:

- When a during a transaction a cached data item is accessed, the client needs to verify whether that data item is still consistent with version stored at the server.
- Let the client proceed while the verification is taking place.
- Verify whether the cached data are up to date only when the transaction commits.

Coherence enforcement: how caches are kept consistent with copies stored at servers.

2 approaches:

- Server sends an invalidation to all the caches whenever a data is modified.
- Simply propagate the update.

Write-through caches: Allow the clients to modify the cached data, and forward the update to the servers. Clients are granted exclusive write permissions. Performance is improved as all the operations are local.

Write-back caches: Allowing multiple writes to take place before informing the servers.

Implementing client centric consistency

Each write operation W is assigned a globally unique identifier. Such an ID is assigned to the server to which the write has been submitted. We refer to this server as **origin** of W. Then for each client, we keep tracks of two sets of writes. The read set for a client consists of writes relevant for the read operations performed by the client, and the write set for the writes relevant for the write operations performed by the client.

When a client performs a read operation at a server, that server is handed the client's read set to check whether all the identified writes have taken place locally on the server. If not, it is first brought up to date by contacting other servers. Alternatively, the read operation if forwarded to a server where read operations have already taken place. Next the client's read set is updated.