The University *of* New South Wales School *of* Computer Science & Engineering

COMP9243 — Week 3a (13s1)

Ihor Kuz

Replication and Consistency

Replication

Replication involves creating and maintaining copies of services and data provided by a distributed system. Unlike communication, without which it is impossible to build a distributed system, replication is not a fundamental principle. This means that it is possible to build a distributed system that does not make use of replication.

Replication does, however, become important when reliability, performance, and scalability of a distributed system are key concerns. In the case of reliability, creating many redundant copies of a service improves that service's availability. With multiple servers available to clients, it is less likely that a malfunction of one of them will render the whole service unavailable. Likewise, if the data on a server becomes corrupt, data stored at replicas can be used to restore the correct state. With regards to performance, replicating services helps to reduce the load on individual servers. Likewise, by placing replicas close to clients the impact of communication can be greatly reduced. Finally, replication is a key technique for improving a system's scalability. As a service grows, creating more replicas allows the service to scale along with the growth.

When considering the replication of services, there are two types of replication possible: *data replication* and *control replication*. In the first case, only a service's data is replicated. Processing and manipulation of the data is performed either by a non-replicated server, or by clients accessing the data. A typical example of data replication is a replicated (also known as mirrored) FTP site. Web browsers with caches are another example of data replication. In the second case, only the control part of the service is replicated while the data remains at a single centralised server. This form of replication is generally used to to improve or maintain performance by spreading the computational load over multiple servers. It is also possible to combine data and control replication, in which case both the data and control are replicated. They may be replicated together (i.e., both control and data are placed on the same replica servers), or separately (i.e., data is replicated on different servers than control).

During the design and implementation of replication in a distributed system, there are a number of issues that must be addressed. The most important of these is keeping the copies of replicated data consistent. Furthermore, it is important to decide how replicas propagate updates amongst each other, where to place the replicas, how many replicas to create, when to add and remove replicas, etc.

Distributed Data-Store

The following model of a distributed data-store will be used during the further discussion of replication. A data store is a generic term for a service that stores data. Examples of data stores include: shared memory, databases, file systems, objects, web servers, etc. A data store stores *data items* (depending on the data store, a data item could be a page of memory, a record, a file, a variable, a Web page, etc.). Clients wishing to access data from a data store connect to the data store and perform read and write operations on it. The exact nature of a client connection depends on the underlying data store and could be through a network connection or direct access. We abstract from this detail by assuming that, from the clients point of view, the time required to communicate with the data store is insignificant.

From a client's point of view a data store acts like a centralised service hosted on a single server. Internally, however, the data store consists of multiple servers (called *replica servers*) each

containing a copy of all the data items stored in the data store. Each replica server runs a *replica manager* process, which receives operation invocation requests from clients and executes the operations locally on its copy of the data. The replica manager is also responsible for communicating with replica managers running on the other replica servers. We refer to the combination of a replica manager running on a replica server as a *replica*. A client always connects to a single replica. Figure 1 illustrates this model.

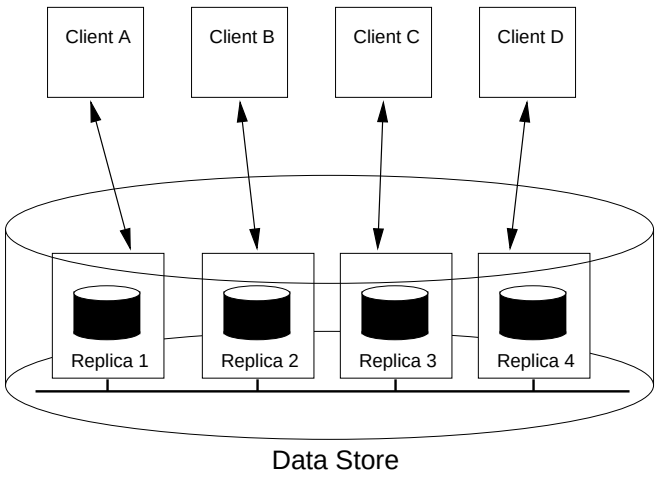

Figure 1: The data store model

The operations performed by clients on a data store will be represented on a time line, an example of which is shown in Figure 2. In this figure, time flows to the right (the figure assumes an absolute global time, and the position of the operations on the timeline reflect their ordering according to this time). For the operations there are three relevant times: the time of issue, the time of execution, and the time of completion. Arrows show the time of execution of operations on remote replicas. Read operations are always performed locally, while writes are assumed to be performed locally first and then propagated to remote replicas.

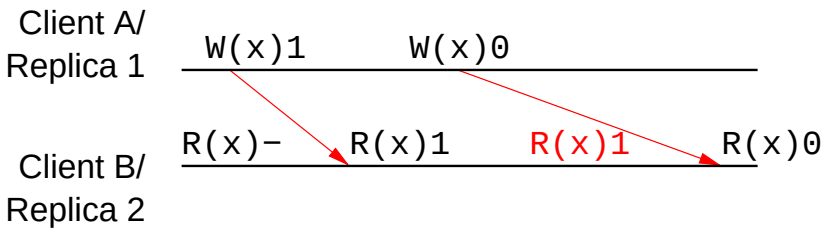

Figure 2: An example timeline of two clients accessing a distributed data-store

Consistency

When we replicate data we must ensure that when one copy of the data is updated all the other copies are updated too. Depending on how and when these updates are executed we can get inconsistencies in the data store (i.e., all the copies of the data are not the same). There are two ways in which the data store can be inconsistent. First the data could be stale, that is, the data at some replicas has not been updated, while others have. Staleness is typically measured using time or versions. As long as updates are reliably propagated to all replicas, given enough time (and a lack of new updates) stale data will eventually become up to date. The other type of inconsistency occurs when operations are performed in different orders at different replicas. This

can cause more problems than staleness because applying operations in different orders can lead to different results that cannot be consolidated by simply ensuring that all replicas have received all updates. In the following we concentrate on consistency with regards to the ordering of operations at different replicas.

Because clients on different machines can access the data store concurrently, it is possible for clients to invoke *conflicting operations*. Two (or more) operations on the same data item are conflicting if they may occur concurrently (that is each is invoked by a different client), and at least one of them is a write. We distinguish between read-write conflicts (where only one operation is a write) and write-write conflicts (where more than one operation is a write). In order for a data store to be consistent all write-write conflicting operations must be seen in an agreed upon order by the clients.

All operations executed at a single replica occur in a particular order. This is called the replica's *partial ordering*. Central to reasoning about consistency is the notion of an interleaving of all the partial orderings into a single timeline (as though the operations were all performed on a single non-replicated data store). This is called the *total ordering*.

Consistency Models

https://www.cs.colostate.edu/~cs551/CourseNotes/Consistency/TypesConsistency.html#:~:text=FIFO%20Consistency%20Model%3A%20For%20FIFO,different%20order%20by%20different%20processes.

In a nondistributed data-store the program order of operations is always maintained (i.e., the order of writes as performed by a single client must be maintained). Likewise, data coherence is always respected. This means that if a value is written to a particular data item, subsequent reads will return that value until the data item is modified again. Ideally, a distributed data-store would also exhibit these properties in its total ordering. However, implementing such a distributed data store is expensive, and so weaker models of consistency (that are less expensive to implement) have been developed.

A consistency model defines which interleavings of operations (i.e., total orderings) are acceptable (*admissible*). A data store that implements a particular consistency model must provide a total ordering of operations that is admissible.

Data-Centric Consistency Models

The first, and most widely used, class of consistency models is the class of data-centric consistency models. These are consistency models that apply to the whole data store. This means that any client accessing the data store will see operations ordered according to the model. This is in contrast to client-centric consistency models (discussed later) in which clients request a particular consistency model and different clients may see operations ordered in different ways.

Strict Consistency The strict consistency model requires that any read on a data item returns a value corresponding to the most recent write on that data item. This is what one would expect from a single program running on a uniprocessor. A problem with strict consistency is that the interpretation of 'most recent' is not clear in a distributed data store. A strict interpretation requires that all clients have a notion of an absolute global time. It also requires instant propagation of writes to all replicas. Due to the fact that it is not possible to achieve absolute global time in a distributed system and the fact that communicating between replicas can never be instantaneous, strict consistency is impossible to implement in a distributed data store.

A model that is close to strict consistency, but that is possible to implement in a distributed data store, is the linearisable consistency model. In this model the requirement of absolute global time is dropped. Instead all operations are ordered according to a timestamp taken from the invoking client's loosely synchronized local clock. Linearisable consistency requires that all operations be ordered according to their timestamp. This means that all operations are executed in the same order at all replicas. Note that although it is possible for a distributed data store to implement this model, it is still very expensive to do so. For this reason linearisable consistency is rarely implemented.

Sequential Consistency Linear consistency is expenisve to implement because of the time ordering requirement. The sequential consistency model drops this requirement. In a data store that provides sequential consistency, all clients see all (write) operations performed in the same order. However, unlike in the linearisable consistency model where there is exactly one valid total ordering, in sequential consistency there are many valid total orderings. The only requirement is that all clients see the same total ordering.

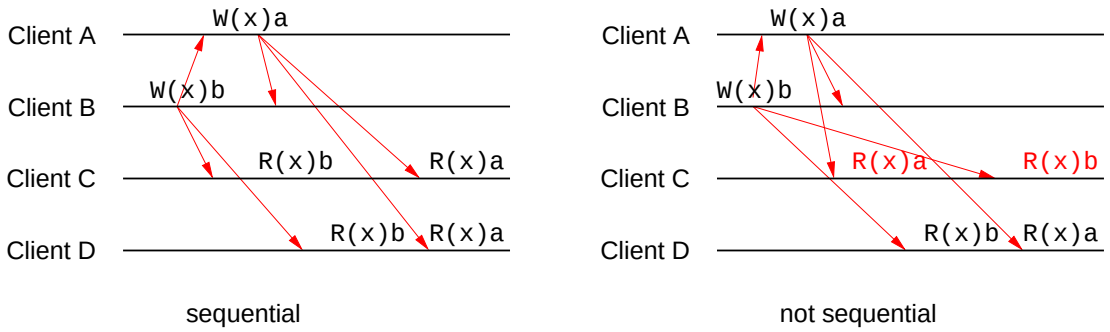

Figure 3: An example of a valid and an invalid ordering of operations for the sequential consistency model

Figure 3 shows an example of a valid and an invalid ordering of operations for the sequential consistency model. In the example of invalid ordering the two write operations are executed in a different order on the replicas associated with client C and client D. This is not admissible with the sequential consistency model.

It has been shown that there is fixed minimum cost for implementations of sequential consistency. It is possible to provide an implementation where reads are instantaneous but writes have a significant overhead, or an implementation where writes are instantaneous but reads have a significant overhead. In other words, changing the implementation to improve read performance makes write performance worse and vice versa.

Causal Consistency Often the requirement that all operations are seen in the same order is not important. The causal consistency model weakens sequential consistency by requiring that only causally related write operations are executed in the same order on all replicas. Two writes are causally related if the execution of one write possibly influences the value written by the second write. Specifically two operations are causally related if:

- A read is followed by a write in the same client
- A write of a particular data item is followed by a read of that data item in any client.

If operations are not causally related they are said to be concurrent. Concurrent writes can be executed in any order, as long as program order is respected.

Figure 4 shows an example of a valid and an invalid ordering of operations for the causal consistency model. In the example of an invalid ordering we see that the write performed by client B (`W(x)b`) is causally related to the previous write performed by client A (`W(x)a`). As such, these writes must appear in the same (causal) order at all replicas. This is not the case for client D where we see that `W(x)a` is executed after `W(x)b`.

FIFO Consistency The FIFO (or Pipelined RAM) consistency model, weakens causal consistency in that it removes limitations about the order of any concurrent operations. FIFO consistency requires only that any total ordering respect the partial orderings of operations (i.e., program order).

Figure 5 provides an example of a valid and invalid ordering for FIFO consistency. In the invalid ordering example client D does not observe the writes coming from client A in the correct order (i.e., `W(x)c` is executed before `W(x)a`).

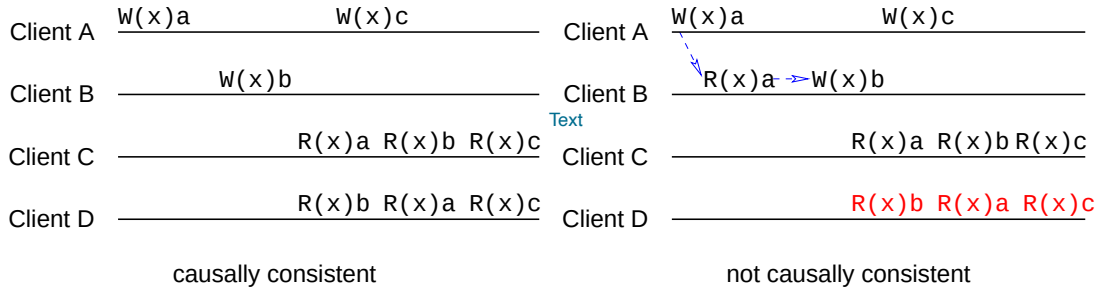

Figure 4: An example of a valid and an invalid ordering of operations for the causal consistency model

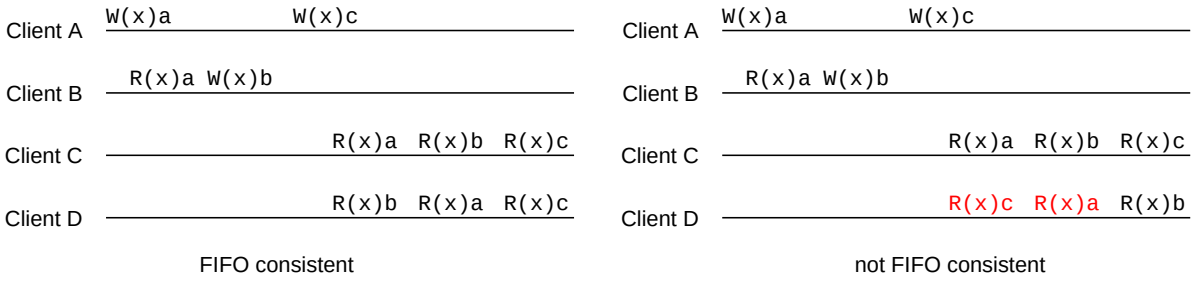

Figure 5: An example of a valid and an invalid ordering of operations for the FIFO consistency model

Weak Consistency Whereas the previous consistency models specified the ordering requirements for all operations on the data store, the following data-centric models drop this requirement dealing instead with the ordering of groups of instructions. They do this by defining groups of operations (comparable to *critical sections*) and only dictating requirements for the ordering of these groups, rather than individual operations.

The first of these models is the weak consistency model. in this model critical sections are delimited using operations on *synchronisation variables* (e.g., locks). In this model performing a synchronise operation on a synchronisation variable causes the following to happen. First, all local writes are completed and the updated data items are propagated to all other replicas. Second, all updates from other clients are executed locally (that is, the replica makes sure that its copy of the data is up-to-date). Essentially, weak consistency imposes a sequentially consistent ordering of synchronise operations.

Release Consistency The functionality of the synchronise operation as defined in the weak consistency model can be split into two separate functions: bringing local state up-to-date and propagating local updates to all other replicas. Because there is only one operation weak consistency requires both to occur whenever entering and leaving a critical section. This is generally not necessary. The release consistency model makes the distinction between the two functions explicit by associating the first with an `acquire()` operation and the second with a `release()` operation. The model requires a client to call `acquire()` when entering a critical section and `release()` when leaving it. This ensures that when entering a critical section all data is up-to-date, while when leaving the section all updates are made available to other replicas.

A slight modification of the release consistency model, called lazy release consistency, requires that updates are only propagated and executed when an `acquire()` operation is performed. This saves much communication when a critical section is performed repeatedly by a single client.

Entry Consistency The entry consistency model is similar to lazy release consistency, except that the synchronisation variables are explicitly associated with specific data items. These data

items are called *guarded* data-items. Use of synchronisation variables in the entry consistency model is as follows. In order to write to a guarded data-item a client must acquire that item's synchronisation variable in exclusive mode. This means that no other clients can hold that variable. When the client is done updating the data item it releases the associated synchronisation variable. When a client wishes to read a particular data item it must acquire the associated synchronisation variable in nonexclusive mode. Multiple clients may hold a synchronisation variable in nonexclusive mode.

When performing an acquire, the client fetches the most recent version of the data item from the synchronisation variable's owner (A synchronisation variable's owner is the last client that performed an exclusive acquire on it).

Although this, and the previous weak consistency models, result in extra complexity for the programmer, it is possible to hide the use of synchronisation variables by associating guarded data-items with objects. Invoking a method on an object then automatically invokes the associated acquire and release operations.

0.0.1 CAP Theory and Eventual Consistency

In 2000 Eric Brewer claimed (and in 2002 it was proven [GL02]) that in a replicated data store, of the three desired properties, consistency, availability, and partition tolerance, only two can ever be guaranteed at once. This is called the *CAP theorem*. What it means is that in a system that can survive a network partition (that is, a system that continues to function, and does not outright fail, when it is split into two or more parts that are (temporarily) unable to communicate with each other) it is only possible to provide consistency (specifically the property that a read provides the results of the latest write) or availability (that a write is always accepted and processed in a timely fashion), but not both.

Since network partitions are statistically very likely in large distributed systems, the CAP theorem presents a real limitation for modern, large-scale distributed systems. This has led to the increasing popularity of *eventual consistency* [Vog08].

The eventual consistency model weakens the temporal aspect of consistency, guaranteeing only that, if no updates take place for a while, eventually all the replicas will contain the same data. This model generally applies when there are few conflicting operations and means that only the ordering of writes to the same data item is respected.

The eventual consistency model requires that the data store experience few read-write conflicts (e.g., because there are many more reads than writes), and that there are few if any write-write conflicts (e.g., because all writes are performed by the same client). Also, it is imperative that clients accept temporary inconsistencies (i.e., staleness). Typical examples of systems that allow eventual consistency are DNS and the Web. In both systems it takes new data a while to propagate and replace old data stored in caches.

Client-Centric Consistency Models

The data-centric consistency models had an underlying assumption that the number of reads was approximately equal to the number of writes, and that concurrent writes occur often. Client-centric consistency models, on the other hand, assume that clients perform more reads than writes and that there are few concurrent writes. They also assume that clients can be mobile, that is, they will connect to different replicas during the course of their execution.

Client-centric consistency models are based on the eventual consistency model but offer per-client models that hide some of the inconsistencies of eventual consistency. Client-centric consistency models are useful because they are relatively cheap to implement.

For the discussion of client-centric consistency models we extend the data store model and notation somewhat. The change to the data store model is that the client can change which replica it communicates with (i.e., the client is mobile). We also introduce the concept of a *write set* (`WS`). A write set contains the history of writes that led to a particular value of a particular

data item at a particular replica. When showing timelines for client-centric consistency models we are now concerned with only one client performing operations while connected to different replicas

Monotonic Reads The monotonic-reads model ensures that a client will always see progressively newer data and never see data older than what it has seen before. This means that when a client performs a read on one replica and then a subsequent read on a different replica, the second replica will have at least the same write set as the first replica. This is shown in Figure 6. The figure also shows an invalid ordering for monotonic reads. This ordering is invalid because the write set at the second replica does not yet contain that from the first.

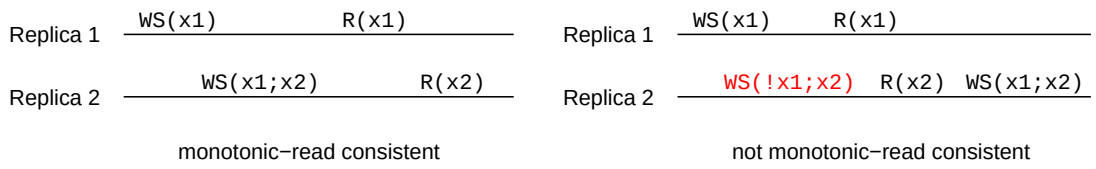

Figure 6: An example of a valid and invalid ordering for the monotonic reads consistency model

Monotonic Writes The monotonic-writes model ensures that a write operation on a particular data item will be completed before any successive write on that data item by the same client. In other words, all writes that a client performs on a particular data item will be sequentially ordered. This is essentially a client-centric version of FIFO consistency (the difference being that it only applies to a single client). Figure 7 shows an example of a valid and invalid ordering for monotonic writes consistency. The example of the invalid ordering shows that the write performed at replica 1 has not yet been executed at replica 2 when the second write is performed at that replica.

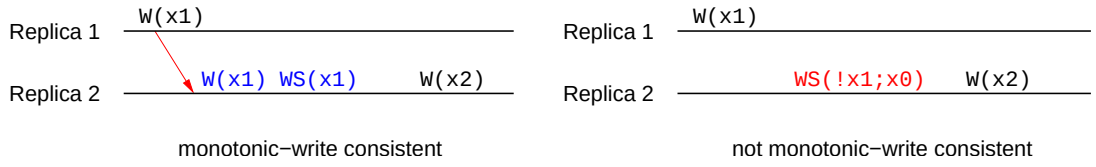

Figure 7: An example of a valid and invalid ordering for the monotonic writes consistency model

Read Your Writes In the read you writes consistency model, a client is guaranteed to always see its most recent writes. Figure 8 shows an example of read your writes ordering. The figure also shows an example where the client does not see its most recent write at another replica. In this case, the write set at replica 2 does not contain the most recent write operation performed on replica 1.

Ensures that a client always sees its own previous writes.

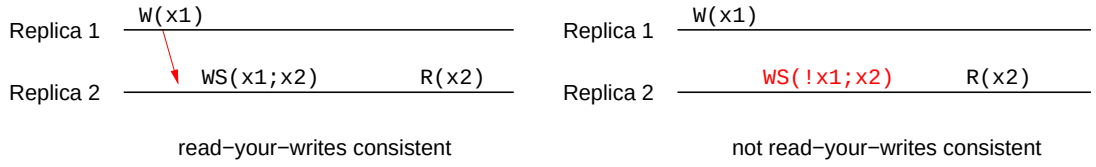

Figure 8: An example of a valid and invalid ordering for the read your writes consistency model

Write Follows Reads This model states the opposite of read your writes, and guarantees that a client will always perform writes on a version of the data that is at least as new the last version

Ensures that writes are based on the latest version of the data seen by the client.

it saw. Figure 9 shows an example of write follows reads ordering. In the example of the non write follows reads ordering, the two replicas do not have the same write set (and the one on replica 2 is also not newer than the one on replica 1). This means that the read and the write operations are not performed on the same state.

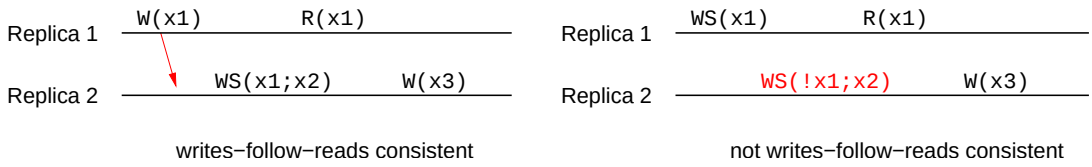

Figure 9: An example of a valid and invalid ordering for the write follows reads consistency model

Consistency Protocols

Having discussed various consistency models, it is now time to focus on the implementation of these models. A consistency protocol provides an implementation of a consistency model in that it manages the ordering of operations according to its particular consistency model. In this section we focus on the various ways of implementing data-centric consistency models, with an emphasis on sequential consistency (which includes the weak consistency models as well).

There are two main classes of data-centric consistency protocols: primary-based protocols and replicated-write based protocols. Primary-based protocols require that each data item have a primary copy (or home) on which all writes are performed. In contrast, the replicated-write protocols require that writes are performed on multiple replicas simultaneously.

The primary-based approach to consistency protocols can further be split into two classes: remote-write and local-write. In remote-write protocols writes are possibly executed on a remote replica. In local-write writes are always executed on the local replica.

Single Server The first of the remote-write protocols is the single server protocol. This protocol implements sequential consistency by effectively centralising all data and foregoing data replication altogether (it does, however, allow data distribution). All write operations on a data item are forwarded to the server holding that item's primary copy. Reads are also forwarded to this server. Although this protocol is easy to implement, it does not scale well and has a negative impact on performance. Note that due to the lack of replication, this protocol does not provide a distributed system with reliability.

Primary-Backup The primary-backup protocol allows reads to be executed at any replica, however, writes can still only be performed at a data item's primary copy. The replicas (called backups) all hold copies of the data item, and a write operation blocks until the write has been propagated to all of these replicas. Because of the blocking write, this protocol can easily be used to implement sequential consistency. However, this has a negative impact on performance and scalability. It does, however, improve a system's reliability. Furthermore, while it is possible to make the write nonblocking, greatly improving performance, such a system would no longer guarantee sequential consistency.

Migration The migration protocol is the first of the local-write protocols. This protocol is similar to single server in that the data is not replicated. However, when a data item is accessed it is moved from its original location to the replica of the client accessing it. The benefit of this approach is that data is always consistent and repeated reads and writes occur quickly, with no delay. The drawback is that concurrent reads and writes can lead to thrashing behaviour where the data item is constantly being copied back and forth. Furthermore the system must keep track of every data item's current home. There are many techniques for doing this including broadcast,

forwarding pointers, name services, etc. A number of these techniques will be discussed in a future lecture on naming.

Migrating Primary (multiple reader/single writer) An improvement on the migration protocol is to allow read operations to be performed on local replicas and to migrate the primary copy only on writes. This improves on the write performance of primary-backup (only if nonblocking writes are used), and avoids some of the thrashing of the migration approach. It is also good for (mobile) clients operating in disconnected mode. Before disconnecting from the network the client becomes the primary allowing it to perform updates locally. When the client reconnects to the network it updates all the backups.

Active Replication The active replication protocol is a replicated write protocol. In this protocol write operations are propagated to all replicas, while reads are performed locally. The writes can be propagated using either point-to-point communication or multicast. The benefit of this approach is that all replicas receive all operations at the same time (and in the same order), and it is not necessary to track a primary, or send all operations to a single server. However it does require atomic multicast or a centralised sequencer, neither of which are scalable approaches.

Quorum-Base Protocols With quorum based protocols write operations are executed at a subset of all replicas. When performing read operations clients must also contact a subset of replicas to find out the newest version of the data. In this protocol all data items are associated with a version number. Every time a data item is modified its version number is increased.

This protocol defines a *write quorum* and a *read quorum*, which specify the number of replicas that must be contacted for writes and reads respectively. The write quorum must be greater than half of the total replicas, while the sum of the read quorum and the write quorum must be greater than the total number of replicas. In this way a client performing a read operation is guaranteed to contact at least one replica that has the newest version of the data item. The choice of quorum sizes depends on the expected read-write ratio and the cost of group communication.

Update Propagation

Another important aspect of implementing replication and consistency protocols is the question of how updates are propagated to other replicas. There are three approaches to this: send the data, send the operation, and send an invalidation. In the first approach the updated data item is simply sent to the other replicas. In the second approach the operation that caused an update to the data item is sent to all replicas. This operation is then performed by the remote replicas, updating their local store. Finally sending an invalidation involves notifying the replicas that the copy of the data item that they hold is no longer valid. It is then up to each replica to contact the sender of the invalidation to retrieve the new state of the data item. Which approach to use largely depends on the context.

The benefits of sending the updated data are that the replicas do not need to perform any actions other than simply replacing their copy of the appropriate data item. Sending the data is a good approach if the data items are small, or if few updates are performed. The benefits of propagating the update operation are that the messages may be significantly smaller than the actual data items, this is useful when bandwidth is limited. Likewise replicas can store logs of operations, and it may be possible to resolve write conflicts if the updates affect different parts of the data item. Invalidation is a useful approach if data items are large, and many updates occur.

Push vs Pull

Besides deciding what to send in an update message, it is also important to decide whether updates are pushed to all replicas when they occur or pulled from replicas when they are needed. The push model is a useful approach when a high degree of freshness is required (i.e., clients always want to access the newest data), as well as when there are few updates and many reads. A drawback of

this approach, however, is that the writer must keep track of all replicas. This is not a problem when the set of replicas is small and stable, but does become a problem when there are many replicas (e.g., web browser caches) and they are unstable (e.g., browsers purging their caches, being stopped and started, etc.)

On the other hand, when there are many updates and few reads it is more efficient to have the reader pull the update, that is, send a request for the newest version whenever a read is performed. It is also efficient to do this when the server does not want to keep track of all replicas. The drawback of this approach is that it may incur a polling delay, meaning that replicas must check for the most up to date version every time a read request is made. It is possible to avoid the poll delay by periodically checking the freshness of the replicated data, however, this means that replicas may contain stale data (as happens in the Web).

Leases Because the push approach is inefficient if a replica has no interested clients, the concept of timed leases can be used to keep track of and push to interested replicas only. When a replica is interested in receiving updates for a particular data item it acquires a lease for that item. Whenever updates occur they are then propagated to that replica. When the lease expires the replica no longer receives updates. It is up to each replica to renew its lease if it is still interested in updates.

To cut down on the costs of constantly renewing leases and to prevent sending unnecessary updates, it is possible to base the length of leases on characteristics of the replicas and data items. Lease length can be based on the age of the data item, on the renewal frequency of a replica, and on the overhead that lease management incurs. With age based leases, data items that were recently modified will receive shorter leases. This is based on the expectation that they will be modified again soon and will therefore generate many update messages. As such it is important to make sure that the receivers of the updates are still interested in them. For data items that are not expected to be modified soon the lease age can be longer, as they will not generate many update message. With renewal-frequency based leases, replicas that often request to have their copy of a data item updated will receive longer leases than those that do so infrequently. Finally state-space-overhead based leases base the lease length on available resources for storing and processing lease state as well as propagating updates. When available resources are low, lease lengths will also be low.

Multicast vs Unicast

As has been mentioned previously, multicast communication can often be used to propagate updates to other replicas. This is particularly useful when updates have to be propagated to many replicas. Furthermore atomic multicast is very useful for maintaining operation order. In atomic multicast a message is guaranteed to be delivered to all recipients or none at all. Also all messages are delivered in the same order to all recipients. Unfortunately it is difficult to implement atomic multicast in a scalable way.

In the situation where all replicas are on the same LAN, broadcast can also be used. This is obviously not geographically scalable, but in many situations (e.g., a cluster of replicated servers) such scalability is not required. Unicast (or point-to-point communication) is more useful when updates are pulled by clients, or when multicast cannot be implemented efficiently. A further consideration is that unicast communication mechanisms are also more readily available to programmers.

Replica Placement

A final issue with regards to the implementation of replication is the question of where to place replicas, how many replicas to create, who is responsible for creating and maintaining them, and how clients find the most appropriate replicas to connect to. Replicas can be categorised into *permanent replicas*, *server-initiated replicas*, and *client-initiated replicas* as shown in Figure 10.

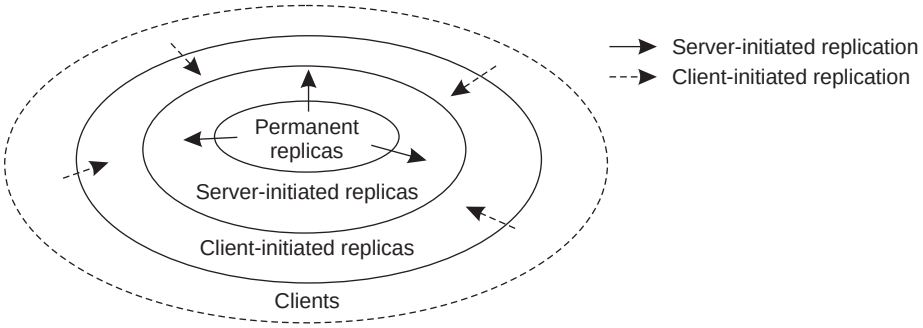

Figure 10: Different kinds of replicas

Permanent replicas are created by the data store owner and function as permanent storage for the data. Often this is a single server, but it may also be a cluster or group of mirrors maintained by the data store owner. This category also includes replicas that are created for fault tolerance (availability) reasons. Writes are usually only performed by clients directly connected to permanent replicas. Server-initiated replicas are replicas created in order to enhance the performance of the system. They are created at the request of the data store owner but are often placed on servers maintained by others. The replicas are not as long lived as the permanent replicas, although exceptions where server-initiated replicas exist for as long as the permanent replicas are possible. In order to improve performance these replicas are placed close to large concentrations of clients. Finally, client-initiated replicas are temporary copies that the data store owner is not generally aware of. They are created by clients to improve their own performance and access to the data. A typical example of client-initiated replicas are Web browser caches and proxy caches.

Dynamic Replication

In many cases the patterns of use that a distributed system will experience will change over time. For example, the number of clients accessing the system can change (grow or shrink), the amount of data that the system contains will tend to grow, the access characteristics may change (i.e., the R/W ratio may change), etc. The changes can be steady or bursty. Bursty changes are often characterised by sudden, heavy, increases in usage, followed by sharp declines.

In order to adapt to these changes many systems apply *dynamic replica placement*. With dynamic replica placement the decisions about where to place replicas and when to create new ones is made automatically by the system. This kind of automatic replication requires a specific infrastructure which allows the collection of usage pattern data and the migration of replicas to and from other servers. It generally also requires the availability of a supporting network of servers willing to host replicas.

An example of a dynamic replica placement strategy comes from the RaDaR Web hosting service [RA99]. In that system clients send all requests to a nearest server, where they are forwarded on to a server that contains the actual replica. All servers keep track of where requests for replicated data originated. The system defines a number of thresholds: replication, migration and deletion. These threshold are used to determine what should happen with replicas - whether new ones should be created, existing ones destroyed, or existing ones moved to different servers. For example, when the number of requests at a particular replica exceeds the replication threshold a new replica will be created.

Request Routing

So far it has been assumed that clients always connect to the most appropriate replica. Determining where a client's most appropriate replica is, or even what is most appropriate for a client is a difficult problem. Most notably it is difficult to integrate replication solutions into existing distributed systems (e.g., the Web, FTP, etc.) precisely because of this problem. Ideally a client would transparently connect to its most appropriate replica, without user intervention, or without a noticeable detour to a redirection service. The details of identifying and finding replicas and other resources in a distributed system will be discussed in a future lecture on naming in distributed systems.

References

[GL02] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SigAct News*, June 2002.

[RA99] Michael Rabinovich and Amit Aggarwal. RaDaR: a scalable architecture for a global Web hosting service. *Computer Networks*, 31(11–16):1545–1561, 1999.

[Vog08] Werner Vogels. Eventual consistency. *ACM Queue*, October 2008.