Mobile and Ubiquitous Computing 2022-23
MEIC/METI - Alameda & Tagus

Replication & Consistency Models

Definitions: Device Connectivity

- System models:
  - how data is accessed, where it is stored, who is allowed to update it, how updated data propagates between devices, and what consistency is maintained

- A mobile system comprises a number of devices:
  - with computing, storage, and communication capabilities
  - can communicate with each other over a spectrum of networking technologies

- Two devices are *connected* if they can send messages to each other, either over a wireless or wired network

- *Weakly connected* devices can communicate, but only using a low-bandwidth, high latency connection

- A device is said to be *disconnected* if it cannot currently communicate with any other device

- Devices may experience *intermittent connectivity* characterized by alternating periods in which they are connected and disconnected

Definitions: items and collections

- An *item* is the basic unit of information that can be managed, manipulated, and replicated by devices

- Items include:
  - photos, songs, playlists, appointments, e-mail messages, files, videos, contacts, tasks, documents, and any other data objects of interest to mobile users

- Each item can be named by some form of *globally unique identifier*

- A *collection* is a set of related items, generally of the same type and belonging to the same person:
  - e.g., “Joe’s e-mail” is a collection of e-mail messages, etc.
  - it is an abstract entity that is not tied to any particular device or location or physical storage representation
  - each collection has a globally unique identifier so devices can refer to specific collections in replication protocols

- Collections can be shared and replicated among devices

Definitions: Full and Partial Replicas

- A *replica* is a copy of items from a collection that is stored on a given device
- A replica is a *full replica* if it contains all of the items in a collection
- As new items are added to a collection, copies of these items automatically appear in every full replica of the collection
- A *partial replica* contains a subset of the items in a collection
- Devices maintain their replicas in local, persistent storage, called data stores, so that the replicated items survive device crashes and restarts

Definitions: Operations

- Software applications running on a device can access:
  - the device's locally stored replicas, and
  - possibly replicas residing on other connected devices

- Such applications can perform four basic classes of operations on a replica:
  - CRUD: create, read, update, delete

- A *read* operation returns the contents of one or more items from a replica:
  - read operations include retrieving an item by its globally unique identifier, as in a conventional file system read operation, as well as querying items by content

- A *create* operation generates a new item with fresh contents and adds it to a collection:
  - this item is first created in the replica on which the create operation is performed, usually the device's local replica
  - it is then replicated to all other replicas for the same collection

Definitions: Operations

- An *update* operation changes the contents of an item (or set of items) in a replica, producing a new version of that item:
  - a file system write operation is an example of one that modifies an item
  - a SQL update statement on a relational database is also a modify operation

- A *delete* operation directly removes an item from a replica and the associated collection:
  - because the item is permanently deleted from its collection, it will be removed from all replicas of that collection
  - by contrast, a device holding a partial replica may choose to discard an item from its replica to save space without causing that item to be deleted from the collection

Summary

Replication Requirements for Basic Data-Oriented Systems

	REMOTE ACCESS	DEVICE–MASTER	PEER-TO-PEER	PUB–SUB
Continuous connectivity	√√			√
Update anywhere		√√	√√	
Consistency		√√	√√	√√
Topology independence			√√	
Conflict handling		√√	√√	
Partial replication		√	√	√√

Summary

	DEVICES	DATA	READS	UPDATES
Remote access	Server plus mobile devices	Web pages, files, databases, etc.	Performed at server	Performed at server
Device caching	Server plus mobile devices	Web pages, files, databases, etc.	Performed on local cache; performed at server for cache misses	Performed at server and (optionally) in local cache
Device–master replication	Master replica plus mobile devices	Calendars, e-mail, address book, files, music, etc.	Performed on local replica	Performed on local replica; sent to master when connected
Peer-to-peer replication	Any mobile or stationary device	Calendars, e-mail, address book, files, music, etc.	Performed on local replica	Performed on local replica; synchronized with others lazily
Pub–sub	Publisher plus mobile devices	Events, weather, news, sports, etc.	Performed at subscriber's local replica	Performed at publisher; disseminated to subscribers

Data Consistency

Strong Consistency

- Consistency provided by a replicated system:
  - it is an indication of the extent to which users must be aware of the replication process and policies

- Systems that provide **strong consistency** try to exhibit **identical behavior to a non replicated system**:
  - this property is often referred to as **one-copy serializability**
  - it means that an application program, when performing a **read operation**, receives the data resulting from the **most recent update operation(s)**
  - update operations are performed at each device in some well-defined order, at least conceptually
  - maintaining strong consistency **requires substantial coordination** among devices that replicate a data collection which is not compatible with intermittent connectivity.
  - typically, **all or most replicas must be updated atomically** using multiple rounds of messages, such as a two-phase commit protocol

Weak Consistency

- Relaxed/optimistic consistency models have become popular for replicated systems:
  - due to their **tolerance of network** and **replica failures** and their **ability to scale** to large numbers of replicas

- Especially important in **mobile environments**:
  - rather than remaining mutually consistent at all times, replicas are allowed to **temporarily diverge** by accepting updates to local data items
  - such **updates propagate lazily** to other replicas

- Read operations performed on a device's local replica may return data that does not reflect recent updates made by other devices:
  - users and applications must be **able to tolerate potentially stale information**

- Mobile systems generally strive for **eventual consistency**:
  - guaranteeing that **each replica eventually receives the latest update** for each replicated item

- Other stronger (and weaker) consistency guarantees are possible

Eventual Consistency

- A system providing **eventual consistency** guarantees that:
  - replicas **eventually converge to a mutually consistent state**, i.e., to identical contents, if **update activity ceased**

- Ongoing updates may prevent replicas from ever reaching identical states:
  - especially in a mobile system where **communication delays between replicas can be large** due to intermittent connectivity
  - thus, a more pragmatic definition of eventual consistency is desired

- Practically, a mobile system provides eventual consistency if:
  - each update operation is **eventually received** by each device,
  - **noncommutative updates** are performed in the same order at each replica, and
  - the **outcome** of applying a sequence of updates is the same at each replica

- Eventually consistent systems make **no guarantees whatsoever about the freshness of data** returned by a read operation:
  - readers are simply assured of **receiving items that result from a valid update operation** performed sometime in the past
  - e.g. a person **might update a phone number** from her **cell phone** and then be **presented with the old phone number** when querying the address book on **her laptop**

Causal Consistency

- In a system providing causal consistency:
  - a user **may read stale data** but is at least guaranteed to observe states of replicas in which **causally related update operations** have been performed in the proper order

- Suppose an **update originates at some device that had already received and incorporated a number of other updates** into its local replica:
  - this new update is said to **causally follow** all of the previously received updates
  - a causally consistent replicated system ensures that

if update U2 follows update U1, then a user is never allowed to observe a replica that has performed update U2 without also performing update U1

- If two **updates are performed concurrently**, that is, without knowledge of each other:
  - they **can be incorporated into different devices in different observable orders**

Session Consistency

- One **potential problem** faced by users who **access data from multiple devices** is that they **may <u>observe data that fluctuates in its staleness</u>**:
  - e.g. a user may update a phone number on her cell phone and then read the new phone number from her tablet but later read the old phone number from her laptop

- Session guarantees have been devised to:
  - provide a **<u>user</u>** (or application) with a **view of a replicated database** that is **consistent with respect to the set of read and update operations** performed by that user **while still allowing temporary divergence** among replicas

- **Unlike most consistency models** which provide system wide guarantees, **session guarantees are individually selectable by each user or application.**

- **Application designers can choose the set of session guarantees** that they desire based on the **semantics of the data that they manage** and the expected access patterns

Definition of Session

- A **session** is an abstraction for the **sequence of read and write operations** performed during the execution of an application

- Sessions are **not intended to correspond to atomic transactions** that ensure atomicity and serializability

- Instead, the intent is to:
  - present individual applications with a **view of the database that is consistent with their own actions**,
  - even if they **read and write from various, potentially inconsistent servers**

- We want the **results of operations performed in a session to be consistent** with:
  - the model of a **single centralized server**,
  - possibly being **read and updated concurrently by multiple clients**

Session Guarantees

- Session guarantees can be **easily implemented on mobile devices**:
  - provided some **small state** can be **carried with the user** as she **switches between devices**

- More practically:
  - this state can be embedded in applications that access data from mobile devices

- However, systems providing session guarantees on top of an eventually consistent replication protocol may need to:
  - **occasionally prevent access** to some device's replica
  - i.e., **availability may be reduced to enforce the desired consistency properties**, which could adversely affect mobile users.

- One practical alternative is for the system to simply:
  - **inform the user** (or application) when an operation violates a session guarantee, but
  - allow that operation to **continue with weaker consistency**

Session Guarantees (from weakest to strongest)

- Read Your Writes:
  - **read** operations reflect previous **writes**

Violation Example: after changing his password, a user would occasionally type the new password and receive an “invalid password” response

- Monotonic Reads:
  - successive **reads** reflect a non decreasing set of **writes**

Violation Example: on a calendar recently added (or deleted) meetings may seem to appear and disappear

- Writes Follow Reads:
  - **writes** are propagated after **reads** on which they depend

Violation Example: shared bibliographic database to which users contribute entries: a user reads some entry, discovers that it is inaccurate, and then issues a Write to update the entry and it’s not in the DB

- Monotonic Writes:
  - **writes** are propagated after **writes** that logically precede them

Violation Example: version N is written to some server, and version N+1 to a different server, and on some site version N+1 is applied before version N

Session Guarantees

- These properties are "**guaranteed**" in the sense that:
  - either the storage system **ensures them for each read and write operation belonging to a session**, or
  - it **informs the calling application that the guarantee cannot be met**

- The guarantees can easily be layered **on top of a weakly-consistent** replicated data system:
  - **each read or write operation is performed at a single server**, and
  - the **writes are propagated to other servers in a lazy fashion**

- To ensure that the guarantees are met:
  - **the servers at which an operation can be performed must be restricted to a subset of available servers** that are **sufficiently up-to-date**

System Model

- Basic assumption:
  - a **weakly consistent replicated storage system** to which the guarantees will be added
  - it consists of **a number of servers that each hold a full copy of some replicated database** and **clients that run applications desiring access to the database**

- The session guarantees are applicable to systems in which **clients and servers may reside on separate machines** and a **client accesses different servers over time**:
  - e.g, a mobile client may choose servers based on which ones are available in its region and can be accessed most cheaply

System Model

- Definition and implementation of session guarantees:
  - it is **unaffected by whether Writes are simple database updates or more complicated atomic transactions**

- Each **Write** has a **globally unique identifier**:
  - it is called a "**WID**"
  - the server that first accepts the Write, for instance, might be responsible for assigning its WID

- **Read** and **Write** operations may be **performed at any server or set of servers**

- The guarantees are presented assuming that **each Read or Write is executed against a single server's copy of the database**:
  - i.e. for the most part, we discuss variants of a **read-any/write-any** replication scheme
  - however, the guarantees could also be used in systems that read or write multiple copies, such as all of the available servers in a partition

Terminology

- We define **DB(S,t)** to be:
  - the ordered **sequence of Writes that have been received by server S at or before time t**
  - if it is known to be the current time, then it may be omitted leaving DB(S) to represent the current contents of the server's database

- Conceptually:
  - **server S creates its copy of the database**,
  - **it uses it to answer Read requests**,
  - it **starts with an empty database** and **applies each Write in DB(S) in the given order**

- In practice, a **server is allowed to process the Writes in a different order as long as they are commutative.**

- The **order of Writes in DB(S) does not necessarily correspond to the order in which server S first received the Writes.**

Terminology

- Weak consistency permits database copies at different servers to vary:
  - **DB(S1,t) is not necessarily equivalent to DB(S2,t) for two servers S1 and S2**

- Practical systems generally desire eventual consistency:
  - servers converge towards identical database copies in the absence of updates
  - thus relying on two properties: **total propagation** and **consistent ordering**

- We **assume** that the **replicated system provides eventual consistency** and thus includes mechanisms to ensure these two properties as follows:
  - **Writes are propagated among servers** by a process called **anti-entropy**.
  - anti-entropy ensures that **each Write is eventually received by each server**
  - i.e, **for each Write W** there exists a **time t** such that **W is in DB(S,t) for each server S**

- There are no other assumptions about:
  - the anti-entropy protocol,
  - the frequency with which it happens,
  - the policy by which servers choose anti-entropy partners, or
  - other characteristics of the anti-entropy process

Terminology

- **All servers apply non-commutative Writes** to their databases in **the same order**:
  - let **WriteOrder(W1,W2)** be a boolean predicate indicating whether Write W1 should be ordered before Write W2
  - the system ensures that **if WriteOrder(W1,W2) then W1 is ordered before W2 in DB(S) for any server S that has received both W1 and W2**

- In a **strongly consistent** system:
  - **WriteOrder would reflect the order in which individual Writes or transactions are committed**

- In an **eventually consistent** system:
  - **servers could use any of a variety of techniques to agree upon the order of Writes**
  - e.g. **using timestamps to determine the Write order does not imply that servers have synchronized clocks** since **there is no requirement that Writes be ordered by the actual time at which they were performed**

- We make **no assumption about**:
  - **how servers agree on the ordering of Writes**, or
  - about **how servers make their copies of the database conform to this ordering**

Providing the Guarantees

- The implementations require only minor cooperation from the servers that process Read and Write operations

- Specifically, a server must be willing to return information about the:
  - **unique identifier** (WID) assigned to a **new Write**,
  - the **set of WIDs for Writes** that are **relevant to a given Read**, and
  - the **set of WIDs for all Writes** in its database

- The burden of providing the guarantees lies primarily with the **session manager on the client**:
  - through which all of a session's Read and Write operations are **serialized**
  - it is a **component of the client stub** that mediates communication with available servers

- For each session, the session manager it maintains two sets of WIDs:
  - **write-set** = set of WIDs for those Writes performed in the session
  - **read-set** = set of WIDs for the Writes that are relevant to session Reads

Session Guarantees – Read Your Writes

- The **Read Your Writes** guarantee is motivated by the fact that:
  - users and applications find it particularly confusing **if they update a database and then immediately read from the database only to discover that the update appears to be missing**

- This guarantee ensures that:
  - the **effects of any Writes made within a session are visible to Reads within that session**
  - thus, **Reads are restricted to copies of the database that include all previous Writes in this session**

- RYW-guarantee:
  - **if Read R follows Write W in a session**, and
  - **R is performed at server S at time t**, then
  - **W is included in DB(S,t)**

- Applications are **not guaranteed** that:
  - a **Read following a Write to the same data item will return the previously written value**
  - in particular, **Reads within the session may see other Writes that are performed outside the session**

Read Your Writes – example 1

- After **changing his password**, a user would occasionally type the new password and **receive an "invalid password" response**:
  - this annoying problem would arise because **the login process contacted a server to which the new password had not yet propagated**
  - the problem **can occur in any weakly consistent system** that manages passwords

- It **can be solved cleanly by having a session per user in which the RYW-guarantee is provided**:
  - such a session should be created for each new user and must exist for the lifetime of the user's account
  - by performing updates to the user's password as well as checks of this password within the session, users can use a new password without regard for the extent of its propagation

- The **RYW-guarantee ensures that the login process will always read the most recent password**

- Notice that this application requires a session to persist across logouts and machine reboots

Read Your Writes – example 2

- Consider a user whose **electronic mail is managed in a weakly consistent replicated database:**
  - as the **user reads and deletes messages**, those messages are **removed from the displayed "new mail"** folder
  - if the **user stops reading mail and returns sometime later**, she **should not see deleted messages** reappear simply because the mail reader refreshed its display from a different copy of the database

- The **RYW-guarantee can be requested within a session used by the mail reader**:
  - to ensure that the effects of any actions taken, such as deleting a message or moving a message to another folder, remain visible

Providing Read Your Writes

- It ensures that:
  - the effects of any Writes made within a session are visible to Reads within that session
  - thus, **Reads are restricted to copies of the database that include all previous Writes in this session**

- It involves two basic steps:
  - whenever a Write is accepted by a server, its assigned WID is added to the session's write-set
  - before each Read to server S at time t, **the session manager must check that the write-set is a subset of DB(S,t)**

- This check could be done:
  - **on the server** by passing the **write-set** to it, or
  - **on the client** by retrieving the **server's list of WIDs**

- The session manager can continue trying available servers until it discovers one for which the check succeeds:
  - if it cannot find a suitable server, then it reports that the guarantee cannot be provided

read-set = set of WIDs for the Writes that are relevant to session Reads
write-set = set of WIDs for those Writes performed in the session

Session Guarantees – Monotonic Reads

- The **Monotonic Reads** guarantee permits users to **observe a database that is increasingly up-to-date over time:**
  - it ensures that **Read operations are made only to database copies containing all Writes whose effects were seen by previous Reads within the session**

- Intuitively, **a set of Writes completely determines the result of a Read if**:
  - **the set includes "enough" of the database's Writes**
  - so that the **result of executing the Read against this set is the same as executing it against the whole database.**
  - **These are the relevant Writes for that read R1: RelevantWrites(S1, t1, R1)**

- Specifically, we say a Write set WS is complete for Read R and DB(S,t) if and only if:
  - WS is a subset of DB(S,t) and
  - for any set WS2 that contains WS and is also a subset of DB(S,t),
  - the result of R applied to WS2 is the same as the result of R applied to DB(S,t)

- Monotonic Reads Guarantee:
  - **if Read R1 occurs before R2 in a session**, and
  - **R1 accesses server S1 at time t1 and R2 accesses server S2 at time t2**, then
  - **RelevantWrites(S1,t1,R1) is a subset of DB(S2,t2)**

Monotonic Reads – example 1

- A **user's appointment calendar is stored online in a replicated database**:
  - where **it can be updated by both the user and automatic meeting schedulers**
- The **user's calendar program periodically refreshes its display** by reading all of today's calendar appointments from the database
  - if it **accesses servers with inconsistent copies of the database, recently added (or deleted) meetings may appear to come and go**
- The **MR-guarantee can effectively prevent this** since:
  - it disallows access to copies of the database that are less current than the previously read copy

Monotonic Reads – example 2

- Consider a **replicated electronic mail database**
- The mail reader issues a **query to retrieve all new mail messages** and **displays summaries** of these to the user
- When the user **issues a request to display one of these messages**, the mail reader **issues another Read** to retrieve the message's contents
- The MR-guarantee can be used by the mail reader to ensure that:
  - the **second Read is issued to a server that holds a copy of the message**
- Otherwise, the user, upon trying to display the message, might **incorrectly be informed that the message does not exist**

Providing Monotonic Reads

- It ensures that:
  - Read operations are made only to **database copies containing all Writes whose effects were seen by previous Reads within the session**

- It involves two basic steps:
  - before each Read to server S at time t, the session manager must **ensure that the read-set is a subset of DB(S,t)**
  - after each Read R to server S, the **WIDs for each Write in RelevantWrites(S,t,R) should be added to the session's read-set**

- This assumes that the server can compute the relevant Writes and return this information along with the Read result

read-set = set of WIDs for the Writes that are relevant to session Reads
write-set = set of WIDs for those Writes performed in the session

Session Guarantees – Writes Follows Reads

- The Writes Follow Reads guarantee ensures that **traditional Write/Read dependencies are preserved** in the ordering of Writes at all servers:
  - in every copy of the database, **Writes made during the session are ordered after any Writes whose effects were seen by previous Reads in the session**

- WFR-guarantee:
  - if Read R1 precedes Write W2 in a session and
  - R1 is performed at server S1 at time t1, then,
  - for any server S2, if W2 is in DB(S2) then any W1 in RelevantWrites(S1,t1,R1) is also in DB(S2) and WriteOrder(W1,W2)

- This guarantee is different in nature from the previous two guarantees in that **it affects users outside the session**:
  - not only does the session **observe that the Writes it performs occur after any Writes it had previously seen**, but
  - also **all other clients will see the same ordering** of these Writes regardless of whether they request session guarantees

Writes Follows Reads – example

- Imagine a **shared bibliographic database** to which users contribute entries describing published papers:
  - suppose that a user **reads some entry**, discovers that it is inaccurate, and then **issues a Write to update the entry**
  - e.g., the person might discover that the page numbers for a paper are wrong and then correct them with a Write such as "UPDATE bibdb SET pages = '45-53' WHERE bibid = 'Jones93'."

- The WFR-guarantee can ensure that:
  - the **new Write updates the previous bibliographic entry at all servers**

- The WFR-guarantee, as defined, associates two constraints on Write operations:
  - a constraint on Write order ensures that **a Write properly follows previous relevant Writes in the global ordering** that **all database replicas will eventually reflect**
  - a constraint on propagation ensures that **all servers (and hence all clients) only see a Write after they have seen all the previous Writes on which it depends**
- This example requires both these properties

Providing WFR and MW

- Providing the Writes Follow Reads and Monotonic Writes guarantees requires:
  - two additional, but reasonable, constraints on the servers' behavior

- C1
  - when a server S accepts a new Write W2 at time t, it ensures that WriteOrder(W1,W2) is true for any W1 already in DB(S,t)
  - that is, new Writes are ordered after Writes that are already known to a server

- C2
  - anti-entropy is performed such that if W2 is propagated from server S1 to server S2 at time t then any W1 in DB(S1,t) such that WriteOrder(W1,W2) is also propagated to S2

- Strictly speaking, the two conditions discussed above must hold for any Write W1 in the session's read-set or write-set rather than for any Write in DB(S,t):
  - this subtle distinction is not likely to have a practical consequence since the weaker requirements would require a server to keep track of clients' read-sets and write-sets.
  - the stronger requirements allow a server's behavior to be independent of the session state maintained by clients

read-set = set of WIDs for the Writes that are relevant to session Reads
write-set = set of WIDs for those Writes performed in the session

Providing Writes Follows Reads

- It ensures that traditional Write/Read dependencies are preserved in the ordering of Writes at all servers:
  - in every copy of the database, **Writes made during the session are ordered after any Writes whose effects were seen by previous Reads in the session**

- It involves two basic steps:
  - **each Read R to server S** at time t results in **RelevantWrites(S,t,R) being added to the session's read-set**
  - **before each Write to server S** at time t, the session manager c**hecks that this read-set is a subset of DB(S,t)**

read-set = set of WIDs for the Writes that are relevant to session Reads
write-set = set of WIDs for those Writes performed in the session

Monotonic Writes

- The Monotonic Writes guarantee says that **Writes must follow previous Writes within the session**

- In other words:
  - a **Write is only incorporated** into a server's database copy **if the copy includes all previous session Writes**
  - the Write is ordered after the previous Writes

- MW-guarantee:
  - if Write W1 precedes Write W2 in a session, then,
  - for any server S2, if W2 in DB(S2) then W1 is also in DB(S2) and WriteOrder(W1,W2)

- This guarantee provides **assurances that are relevant both to the user of a session as well as to users outside the session**

Monotonic Writes – example 1

- The MW-guarantee could be used by a text editor when editing replicated files to ensure that:
  - if the user saves version N of the file and later saves version N+1 then version N+1 will replace version N at all servers
- In particular, it avoids the situation in which:
  - version N is written to some server, and
  - version N+1 to a different server, and
  - the versions get propagated such that version N is applied before N+1

Monotonic Writes – example 2

- Consider a **replicated database containing software source code**

- Suppose that a programmer **updates a library to add functionality in an upward compatible way**:
  - this **new library can be propagated to other servers in a lazy fashion since it will not cause any existing client software to break**
  - however, suppose that the programmer **also updates an application to make use of the new library** functionality
  - **if the new application code gets written to servers that have not yet received the new library, then the code will not compile successfully**

- To avoid this potential problem, the programmer can:
  - **create a new session that provides the MW-guarantee**, and
  - issue the Writes containing new versions of both the library and application code within this session

Providing Monotonic Writes

- It requires that:
  - a Write is only incorporated into a server's database copy **if the copy includes all previous session Writes**

- It involves two basic steps:
  - in order for a server S to accept a Write at time t, **the server's database, DB(S,t), must include the session's write-set**
  - also, whenever a **Write is accepted by a server, its assigned WID is added to the write-set**

read-set = set of WIDs for the Writes that are relevant to session Reads
write-set = set of WIDs for those Writes performed in the session

Read / Write Guarantees

- Operations on which a session is updated or checked

Guarantee	session state updated on	session state checked on
Read Your Writes	Write	Read
Monotonic Reads	Read	Read
Writes Follow Reads	Read	Write
Monotonic Writes	Write	Write

Version Vectors

- A **version vector is a sequence of <server, clock> pairs, one for each server**
- The **server portion is simply a unique identifier** for a particular copy of the replicated database
- The **clock is a value from the given server's monotonically increasing logical clock**
- The only constraint on this **logical clock** is that it **must increase for each Write accepted by the server:**
  - a Lamport clock
  - a real-time clock or
  - simply a counter
- **A <server,clock> pair serves nicely as a WID**, and we assume that **WIDs are assigned in this manner by the server that first accepts the Write**

Version Vectors at Each Server

- **Each server maintains its own version vector** with the following invariant:
  - **if a server has <S,c>** in its version vector, then **it has received all Writes that were assigned a WID by server S before or at logical time c on S's clock**

- For this invariant to hold, during anti-entropy:
  - **servers must transfer Writes in the order of their assigned WIDs**

- A **server's version vector is updated** as part of the anti-entropy process so that:
  - it precisely **specifies the set of Writes in its database**

- Assuming the use of version vectors by servers:
  - more practical implementations of the guarantees are possible in which **the sets of WIDs are replaced by version vectors** (next slide)

Replacing set of WIDs by Version Vectors

- To obtain a **version vector V providing a representation for a set of WIDs**, Ws:
  - set V[S] = the time of the latest WID assigned by server S in Ws (or 0 if no Writes are from S)

- To obtain a version vector V that represents the **union of two sets of WIDs**, Ws1 and Ws2:
  - first obtain V1 from Ws1 and V2 from Ws2 as above
  - then, set V[S] = MAX(V1[S], V2[S]) for all S

- To check **if one set of WIDs, WS1, is a subset of another, WS2**:
  - first obtain V1 from WS1 and V2 from WS2 as above.
  - then, check that V2 “dominates” V1, where dominance is defined as one vector being greater or equal to the other in all components

- With these rules, **the state maintained for each session compacts into two version vectors**:
  - one to **record the session’s Writes**, and
  - one to **record the session’s Reads** (actually the Writes that are relevant to the session’s Reads)

Finding a Server

- To find an acceptable server:
  - the **session manager must check that one or both of these session vectors are dominated by the server's version vector**

- Which session vectors are checked depends on the operation being performed and the guarantees being provided within the session

- **Servers return a version vector along with Read results to indicate the relevant Writes**:
  - in practice, servers may have difficulty computing the set of relevant Writes
  - 1) determining the relevant Writes for a complex query, such as one written in SQL, may be costly
  - 2) it may require servers to maintain substantial bookkeeping of which Writes produced or deleted which database items

- In real systems, servers typically do not remember deleted database entries:
  - they just store a copy of the database along with a version vector
  - for such systems, **a server is allowed to return its current version vector as a gross estimation of the relevant Writes**
  - this **may cause the session manager to be overly conservative when choosing acceptable servers**

Performance Improvement

- Checks for a suitable server can be amortized over many operations within a session:
  - in particular, the **previously contacted server** is always an acceptable choice for the server at which to perform the next Read or Write operation
  - if the session manager "latches on" to a given server, then the **checks can be skipped**
  - only when the session manager **switches to a different server**, like when the previous server becomes unavailable, must a server's current version vector be compared to the session's vectors

- To facilitate finding a server that is sufficiently up-to-date, **the session manager can cache the version vectors of various servers**

- Since **a server's database can only grow over time in terms of the numbers of Writes it has received and incorporated:**
  - **cached version vectors represent a lower bound on a server's knowledge**