# Programming Ubiquitous Things 2023/24 - Spring 2024 - UiO

## Replication

Prof. Paulo Ferreira
paulofe@ifi.uio.no
UiO/IFI/PT

1

1

## Contents

- Introduction (3-10):
  - Motivation, current scenario, and some examples
- System models (11-38):
  - Basic definitions, device-master replication, P2P, pub-sub
- Data consistency (39-49):
  - Strong, weak, best effort, eventual, causal, bounded, VFC, session guarantees
- Session Guarantees (50-75):
  - Read Your Writes, Monotonic Reads, Writes Follow Reads, Monotonic Writes
- Providing the Session Guarantees (76-89):
  - Supporting each guarantee (read-set, write-set), version-vectors, WIDs, finding a server
- Other issues (90-95):
  - Protocols, partial replication, conflicts management, examples

2

2

# Introduction





## Current Scenario

- **Mobility has become increasingly important for both business and casual users of computing technology**
- **DEVICES** - With the widespread adoption of portable computing devices, such as laptops, PDAs, tablet computers, music players, and smartphones, **people can have almost constant access to their personal data as well as to information that is shared with others**:
  - a **user drinking coffee in a cybercafé in India can access e-mail residing on a mail server in Seattle**
  - a **doctor in New York can monitor the health of patients in remote parts of Africa**
  - a **mother waiting to pick up her children after school can be instantly notified that her daughter's soccer practice has been moved to a new location**
  - **teenagers congregating at the mall can use their cell phones to locate not only their buddies but also the hottest sales**
- **NETWORK** - Advances in wireless technology, such as WiFi:
  - **allow people to communicate from their computers with friends, colleagues, and services located around the world**
  - however, **providing users anytime, anywhere access to contextually relevant information presents substantial challenges to designers of mobile computing systems**

# Specific Mobile Aspects

- In (traditional wired) distributed systems:
  - **powerful computers are connected over a fixed networking infrastructure**
- The ubiquitous Internet, mobile computing environments differ in a number of fundamental ways
- Specifically, **mobile computing systems** must accommodate **three novel aspects**:
  - portable devices with **limited resources** (e.g., displays, CPU resources, storage, battery life, and security)
  - intermittent, low-bandwidth, high-latency **network connections**
  - **changing** environmental conditions and contexts
- Techniques that have been developed specifically for mobile computing systems include:
  - **replication** and **caching** of data for off-line access,
  - **remotely** accessing data that resides on other machines,
  - **offloading** computation onto servers or surrogate PCs, and
  - **adapting** system policies and mechanisms to users' changing context and hence changing information needs

5

5

# History

| YEAR | SYSTEM/ TECHNOLOGY | SIGNIFICANCE |
|---|---|---|
| 1981 | Grapevine | Showed that practical systems could use weak consistency replication |
| 1983 | Locus | Devised version vectors for conflict detection |
| 1987 | Clearinghouse | Commercial product relying on epidemic algorithms for update propagation |
| 1987 | Laptops | Provided a truly mobile platform for serious computing |
| 1989 | Grove | Group editor using operation transformation |
| 1989 | Lotus Notes | Commercial product for document replication via periodic bidirectional data exchanges |
| 1990 | Coda | First distributed file system to support disconnected operation; later explored weakly connected operation and automatic conflict resolution |
| 1991 | GSM | Second-generation cellular telephone network launched in Finland |
| 1991 | Ubiquitous computing | Vision for mobile computing pioneered at Xerox PARC |
| 1993 | Apple Newton | First commercial PDA |
| 1993 | Ficus | Peer-to-peer replicated file system with conflict resolution |
| 1994 | Bayou | Replicated database with application-specific conflict management and session guarantees |
| 1994 | Bluetooth | Industry standard short-range wireless protocol developed, although devices did not hit the market until several years later |
| 1996 | Palm Pilot | First widely adopted PDA with sync capability |
| 1997 | WiFi | High-speed wireless local-area networking standard |
| 1997 | WAP | Forum established to standardize wireless Web access |
| 1998 | Roam | Introduced ward model for scalable peer-to-peer replication |
| 1999 | BlackBerry | Commercial cell phone popularizing mobile e-mail access |
| 2001 | IceCube | Allowed application-provided ordering constraints on operations |
| 2002 | TACT | Explored alternative, bounded consistency models |
| 2004 | PRACTI | Separated invalidation notifications from updates in a log-based replication system |

6

6

# Some Examples 1/4

- offline weather app: AccuWeather
  - there is no such thing as a truly offline weather app
  - **you need to use AccuWeather online at some point**
  - however, **AccuWeather provides an accurate 15-day forecast**, which means that even if you are without internet for two weeks, you should still have some indication of whether you need a sombrero or a ski mask for your trip outside
  - what's more, it does so in an intuitive package which takes just seconds to get to grips with
- offline eBook reader app: Amazon Kindle
  - eBook reader apps make excellent **offline apps** because they can keep you occupied for hours without needing to reconnect to the internet
  - **Kindle gives you quick access to thousands of digital books**, and it comes with all of the options you need for an excellent reading experience
  - buy a book (or pick up a free one), download it to your device, and then you can happily read it without ever connecting it to the internet again

7

7

# Some Examples 2/4

- offline travel app: TripAdvisor Hotels Flights
  - TripAdvisor is the king of travel apps
  - supported by a thriving community, it offers reviews, photos and feedback from fellow travelers, then ranks attractions and activities based on what those people say
  - TripAdvisor used to have dedicated **City Guides** which could be **downloaded externally**, but now all of this functionality is baked into the one app, including **offline access to reviews, maps and photos of more than 300 cities**
- offline documents app: Google Drive/One Drive/Dropbox
  - Google Drive lets you **download files and documents** to your device.
  - you can then **work on these files offline**, and they **sync straight back up into the cloud when you get internet again**
  - to do this, tap the 'i' or Options icon of a file in Google Drive, then tap the switch next to Keep on Device
  - you can do this to **as many files as you like**, and **Google Drive will let you work on them away from the cloud**

8

8

# Some Examples 3/4

- offline app for saving things for later: Pocket
  - Pocket is one of the most popular **offline reading apps** on the Play Store
  - you can use it to **download articles, videos, and other content** you find online to your device, then read it offline later
  - you simply click on the share button on the article you want to save and select Pocket to **read it later**
  - it has a beautifully designed interface and is a great way to make sure you don't miss out on content that you didn't manage to finish reading or watching the first time

- offline dictionary app: Offline Dictionaries
  - if you're in a foreign country and don't speak the language, it's crucial that you have a **means of communicating with locals**
  - Offline Dictionaries is a free Android app that sets itself apart from the others thanks to its large database of synonyms and support for more than 50 languages
  - upon launching the app, you **download all the languages you'll want to refer to**, then refer to the app freely without having to worry about internet connectivity



# Some Examples 4/4

- offline translation app: Google Translate
  - Google Translate is one of the easiest-to-use and most effective translators out there
  - you can speak or type into Google Translate to get things translated into more than 90 languages
  - these **key features are available offline**, so long as you **download the languages you're looking to translate between**
  - you can **save your translations as well**, so you can refer back to them later

- offline map app: Google Maps
  - using the old version of Google Maps **offline was a little awkward, but since its most recent update, this functionality is better than ever**
  - it's easy to **download a by visiting the *Offline areas*** tab in the settings menu
  - from there you can **download full city maps**, including Google's excellent navigation system, for use without internet
  - the best part of all, **any maps downloaded in your offline areas will be automatically removed after 30 days,** so there's no need to worry about unused apps taking up storage space unnecessarily

# System models

11

11

## Basic Definitions – connected device

- 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

12

12

# Basic 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

13

13

# Basic Definitions – replicas, partial and full

- 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**

14

14

# Basic 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 several basic classes of operations on a replica:
  - **read, create, modify, delete, update**
- 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

15

15

# Basic Definitions - operations

- A ***modify*** 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
- An ***update*** is a generic term for a **create, modify, or delete operation**
- Thus, the operations can be said to include:
  - **reads, and**
  - **updates**

16

16

# Basic Definitions - updates

- Replication protocols are mainly concerned with **propagating updates between replicas**
- When an update is made directly to an item in a device's replica, that device is said to have updated the item
- Not all operations can necessarily be performed on all replicas:
  - e.g., a **read-only replica** residing on a device might allow read operations but prevent update operations
- In some system models, items are created but never modified:
  - in this case, replicas contain **read-only items**

17

17

# Models

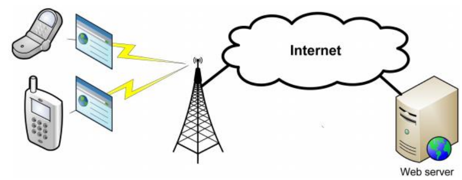
- Remote data access:
  - **information on a server machine** on which it can be remotely invoked by mobile and wireless devices (thin-client)
- Master replication:
  - **authoritative copy resides on a master site**
  - caching or replication is used
- P2P
  - **all devices holding a replica play (nearly) identical roles**, i.e., there is no master replica
- Pub-Sub:
  - small snippets of information, such as news articles, weather reports, and event notifications, are **broadcast from a central site, the publisher, to a number of subscribers**

18

18

# Remote Data Access

- The most basic model for providing anytime, anywhere access to shared information is:
  - store such information on a server machine from which it can be remotely fetched by mobile and wireless devices
  - this model is sometimes called **thin-client**
- Key benefits:
  - support for **arbitrary types of information** and **arbitrary data management systems**
  - since the **data is centrally maintained**, access controls governing who is allowed to read and write various information items can be readily enforced
  - **data consistency is not an issue** since all updates are performed directly at the server; devices that fetch data directly from the server always get the most recently written version




- Main drawback:
  - **data is inaccessible** if a network connection to the server cannot be established or if the server is temporarily unavailable
  - **access time to the data is limited** by the round-trip communication latency between the mobile device and the storage server
  - such communication **consumes valuable battery life** on the mobile device and may incur network charges

19

19

# Master Replication – master site

- Commonly, portable devices store full or partial replicas of data collections whose **authoritative copy resides on a master site**
- The **master source** may be:
  - a shared server, such as a mail server, or a private computer to which the portable device is at least occasionally connected
- Laptops, PDAs, music players, and even cell phones have **enough storage capacity** to replicate significant amounts of data from various sources
- Even if a device has **continuous connectivity to the master**:
  - **entirely replicating databases**, such as a person's address book and calendar, **guarantees instant access to frequently used information and allows local searching**
- For mobile devices that have only occasional connectivity to information sources, **replication is essential**:
  - e.g., a person's iPod may download music from the home PC (or indirectly from the Internet) only when connected by a USB cable
  - without the ability to store music locally, the iPod would be useless

20

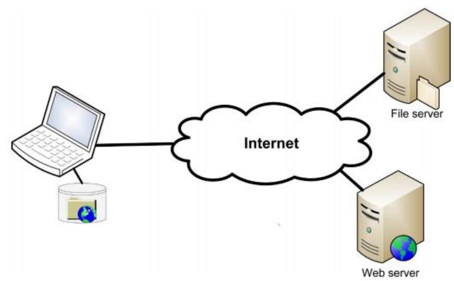
20

# Master Replication – caching and replication

- Two broad approaches have been taken to ensure that a **mobile device has ready access to critical data obtained from the master**:
  - devices **cache** recently accessed data in local storage as a simple extension to the remote data access model, or
  - devices maintain an actively managed, user-visible **replica** of the master's data

- By caching data, mobile devices:
  - can **amortize the cost of retrieving that data over several read operations**, and
  - can retain access to that data when the **device becomes disconnected** from the master storage server

- Example:
  - a **laptop may cache files** that have been fetched from a file server along with the set of recently browsed Web pages




21

21

# Master Replication – caching (1/2)

- Data is generally fetched into a device's cache on-demand:
  - when the user tries to access a file or Web page, the **device's cache is first consulted to see if the data is already available**
  - if the desired data is not cached (or if the cached copy is determined to be **out-of-date** and the user desires the most recent data), then the device may contact the appropriate server to fetch the data
  - in this case, the **fetched data is stored in the cache** for future access

- Devices can control the size of their caches and shrink or grow the cache based on their available storage:
  - items may need to be **discarded to free up space** according to some cache replacement policy, such as removing the items that have been used least recently

- **For well-connected** devices:
  - a **small cache may be sufficient** to hold their working set of frequently accessed data and
  - provide **substantial performance benefits** by avoiding much (but not all) communication with the server

22

22

# Master Replication – caching (2/2)

- Drawback of **on-demand caching**:
  - information requested by a user **will not be available if it is not cached** and the **server is not reachable**
  - such cache misses will occur for **data that has not been recently accessed**, or perhaps that has **never been accessed**
- To minimize cache misses:
  - **hoarding (or stashing)** can be used during periods of server connectivity
  - **preemptively load data objects** into a device's cache in anticipation of future use
- Hoard profiles indicate **which files or data objects a device is likely to access in the near future**, perhaps while disconnected:
  - such **profiles can be specified by users** based on their anticipated needs, or
  - **automatically generated** from observations of past user behavior





# Master Replication – replication (1/5)

- Replicating data on a mobile device is **similar to caching** in that the device stores data whose master copy lives elsewhere
- However, the **replication model differs from device-side caching** in a number of key aspects:
  - in replication, a **whole or partial data collection is copied onto a device at one time**, rather than as individual objects are accessed, and **explicitly refreshed periodically through a synchronization protocol**
  - in replication, attempts to **read a data object fail if the data is not resident** on the accessing device, rather than resulting in a cache miss and a remote access to the master
  - in replication, data **objects are implicitly added to a device's replica** when new objects become part of the replicated data collection
  - in replication, when a device deletes a replicated object, **that object is removed from the data collection and all of its other replicas**, rather than simply being discarded from the device's local storage

# Master Replication – replication (2/5)

- A **wide variety of replication techniques** have been developed for non mobile devices:
  - **quorums and other techniques that provide strong mutual consistency guarantees**
- **Most of such solutions for distributed systems are not applicable for mobile/ubiquitous computing**
- To permit access to data replicated on disconnected devices:
  - mobile systems rely on **weaker consistency guarantees**
  - users are typically permitted to read and write any data that is replicated on their devices **without coordinating** with other devices that may be sharing the same data
- This **read-anywhere, write-anywhere replication model** is:
  - **well suited to mobile devices** with **high-capacity storage**,
  - but **intermittent or weak connectivity** and **limited battery** life
- It is **widely used for both consumer and enterprise applications**

25

25

# Master Replication – replication (3/5)

- Updates originating at a mobile device to a cached item are generally:
  - **not only written to the cached copy**, but also
  - **written directly to the master server**, so that
  - **the updated item is immediately available to other devices.**
- If a connection to the server is not currently possible, then:
  - **updates may be performed locally and queued for later transmission**
- **Updates made by other devices are not necessarily reflected immediately in a device's local cache**:
  - **methods can be used for ensuring that caching clients always read the data that was most recently written**, but
  - these **techniques do not work for intermittently connected devices**
- Therefore, in mobile settings, a device usually is permitted to:
  - **read old items from its cache, and thus the user may see stale information**
- Of course, **if a user accesses data that only he updates**, such as his personal calendar, then **consistency is not an issue**

26

26

# Master Replication – replication (4/5)

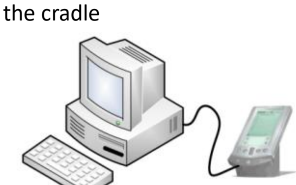
- Data:
  - updated on a device are **uploaded to the master site**, and
  - updates made on other devices are **downloaded from the master site**
- Thus:
  - all updates are done locally and sent to the **master site**,
  - which then **distributes them to other devices** holding replicas of the information
- The process of communicating with the master to upload and download updated data objects is called **synchronization**
- The term **reconciliation** or **reintegration** is also sometimes used for this process





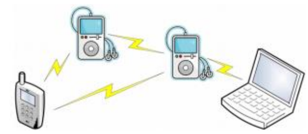
# Master Replication – replication (5/5)

- **Synchronization** takes place as connectivity allows and policy dictates
- Device **without wireless networking** hardware:
  - only means of communicating with a PC is through a wired sync cradle
  - it synchronizes with the attached PC whenever the device is placed in the cradle
- Device **with wireless connectivity** to the master site:
  - e.g., a cell phone that synchronizes e-mail with a mail server
  - it may synchronize its data periodically, say, every 5 minutes or when explicitly requested by a user



- Consequence of a **write-anywhere replication model** is:
  - two users may independently update the same data item on different devices, thereby introducing **conflicting updates**
  - even concurrent updates to different objects may conflict if, taken together, they violate **some invariant** that should hold on the data
- In a master replication model:
  - the **master** is responsible for **detecting when two devices produce conflicting updates**
  - in some cases, the master may be able to **automatically resolve conflicts that arise**, whereas in other cases, such conflicts may **require human attention**

# P2P Replication (1/5)



- All devices holding a replica play (nearly) identical roles, i.e., **there is no master replica**
- Updates are propagated via **pairwise synchronization operations**:
  - relying only on **communication between pairs of devices**
  - P2P replication can effectively **deal with varying connectivity between peers**
- Devices form an **overlay network** of arbitrary topology:
  - **neighbors periodically synchronize** with each other to propagate updates
  - **each node in the overlay network is a fixed or mobile device**, and
  - **each edge represents a synchronization partnership** between two devices with at least **occasional network connectivity**
- Updated data objects flow between devices via the **overlay network**
- Compared with the master replication model:
  - P2P replication over arbitrary overlay topologies requires **more complicated synchronization protocols** but offers a number of key advantages

29

29

# P2P Replication (2/5)

- A device that belongs to a community of replicas can **invite others to join** the community simply by establishing local synchronization partnerships
- The overlay topology can grow **organically** without informing other devices
- Users need **not even be aware of the full set of devices** that are sharing data
- Synchronization **partnerships can come and go as long** as the overlay network of replicas remains well-connected
- If a mobile device **opportunistically** encounters another device that has data in common:
  - these two devices can synchronize with each other without any prior arrangement or synchronization history

30

30

# P2P Replication (3/5)

- The peer-to-peer replication process is **tolerant of failed devices and network outages**
- In the master replication model:
  - if the **master is temporarily unavailable**, devices cannot propagate new updates among themselves until the master recovers or reconnects
- In P2P model:
  - the **loss of a single device does not prevent updates** from propagating along different paths
  - it allows updates to propagate among devices that have internal connectivity but no connection to the Internet at large
- Example:
  - colleagues are holding an off-site meeting at a remote location without an Internet connection but want to collaboratively edit a document and share their edits between their laptops
  - the laptops may be connected by a local WiFi network or use point-to-point Bluetooth or infrared connectivity to exchange new versions of the document

31

31

# P2P Replication (4/5)

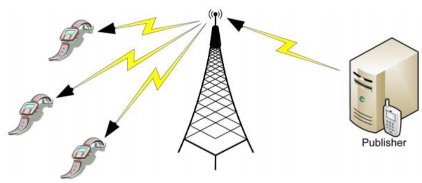
- Even when mobile devices are **well-connected**:
  - nontechnical (e.g., political) concerns may lead organizations to favor configurations that do not rely on a master replica
- Specifically, using peer-to-peer replication, also known as **multi-master replication**:
  - puts all participants on an even footing
  - various relief organizations that need to share emergency information wish to be viewed as equal partners.
- P2P replication model supports collaborators operating as peers when managing shared data
- Principal **cost of P2P replication**:
  - it requires **more complex protocols** for ensuring that updated data objects reach each replica while efficiently using bandwidth
  - **update conflicts may be more prevalent** than in the device–master model
  - **conflicts** may be detected during synchronization between devices that did not introduce the conflicting updates
  - overall, mobile users must deal with a **more complex model** resulting from the absence of a master replica, the lack of knowledge about the full replication topology, and decentralized conflict handling

32

32

# Publish-Subscribe Systems (5/5)

- Characterized primarily by their **pattern of information dissemination**:
  - small snippets of information, such as news articles, weather reports, and event notifications, are broadcast from a central site, the publisher, to a number of subscribers
- Information is grouped into **topic-based channels** allowing devices to subscribe to items of interest




- The information may reach subscribers directly or via other subscribers
- Subscribers may be organized in a **tree topology** with the publisher at the root
- The **publisher and subscribers may be either mobile or fixed devices** with wireless or wired communication capabilities

33

33

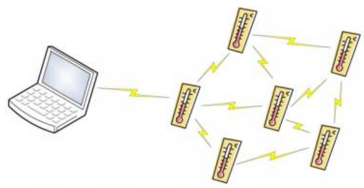
# Publish-Subscribe Systems

- From the perspective of mobile data management:
  - a common and increasingly important scenario is a fixed publisher broadcasting information via wireless networks to mobile devices
  - e.g., users may receive sport scores on their cell phones
- **Cellular providers** offer such information services to attract customers and provide additional revue streams
- Once a **user subscribes to a channel**, such as news or weather:
  - new items published to that channel are automatically replicated to the user’s device(s)
  - such items are treated as read-only and created only at the publisher
- Data replicated to mobile devices via a pub–sub system may be only of **ephemeral interest**:
  - the data is often discarded once they have been read by the user

34

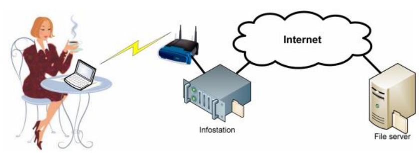
34

# Related Technologies and Models



Ad Hoc Wireless Sensors Networks

Delay-Tolerant Networking




Infostations

35

35

# Summary

- Continuous connectivity:
  - does the system only operate when devices are well-connected?
- Update anywhere:
  - can any device update data items that then must be propagated to other replicas?
- Consistency:
  - does the system require mechanisms to enforce consistency guarantees, such as eventual consistency?
- Topology independence:
  - is the connectivity between devices that replicate data unconstrained, i.e., defined by an arbitrary graph?
- Conflict handling:
  - may devices perform conflicting updates that need to be detected and resolved
- Partial replication:
  - do devices wish to replicate some portion of a data collection?

36

36

# 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; dissemi-nated to subscribers |
| Sensor network | Sensor nodes | Environment data, e.g., temperature | Performed at nodes that accumulate and aggregate data | Real-time data stream at each node; routed to others over ad hoc network |
| Delay-tolerant networking | Any mobile devices | Messages (including files, Web pages, etc.) | Performed by message recipient | Message created by sender; routed to destination through intermediaries |
| Infostations | Publishers, infostations, plus mobile devices | News, messages, advertisements, events, music, etc. | Performed at infostation or on local replica | Performed at publisher; relayed to infostations; picked up by nearby devices |

# Data consistency





## Consistency Algorithms

- Strong vs weak
- Best effort
- Eventual
- Causal
- Bounded
  - VFC
  - Session

# 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
  - typically, **all or most replicas must be updated atomically** using multiple rounds of messages, such as a two-phase commit protocol

41

41

# 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

42

42

# Best Effort Consistency

- Simply **make a best effort** to deliver updates to all replicas:
  - **read** operations performed at different replicas **may return different answers**
- Replicas **may not converge to a mutually consistent state** for any of number of reasons:
  - all updates may not make it to all replicas (e.g., if updates are sent over a mostly, but not totally, reliable communication channel)
- Despite reliable delivery, replicas will not converge if:
  - **updates are performed differently** at different replicas (i.e., the application of an update is not deterministic)
  - updates are **applied in different orders** at different replicas and are not commutable
  - replicas have **different conflict resolution** policies
  - metadata, such as **deletion tombstones**, are discarded too early
  - replicas **lose or corrupt data**, such as when a replica is restored from an old backup
  - the system is improperly configured, such as when the **synchronization topology** is not a well-connected graph

43

43

# Eventual Consistency

- A system providing **eventual consistency** guarantees that:
  - replicas would **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
- 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**

44

44

# 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

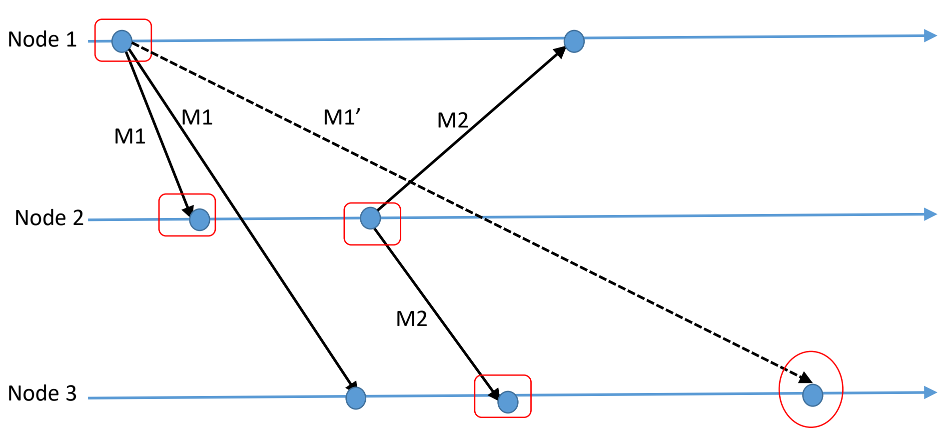
***if update U2 follows update U1, then a user is never allowed to observe a replica that has performed update U2 without also performing update U1***

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





# Causality (in a network with 3 nodes)



# Bounded Consistency

- In some cases, **bounds can be placed on the timeliness or inaccuracy of items** that are read from a device's local replica, providing bounded inconsistency
- For example:
  - an application may desire to **read data that is no more than an hour old**
  - in this case, the system would **guarantee that any updates made more than an hour ago have been incorporated** into the device's replica before allowing a local read operation
- Similarly, a system may **enforce bounds on numerical error or order error**
- This **requires replicas to know about updates made elsewhere** and generally relies on regular connectivity between replicas
- Thus, techniques for ensuring bounded inconsistency **may not be applicable to all mobile environments**

47

47

# Vector Field Consistency

- It unifies:
  - i) several forms of consistency enforcement and a multidimensional criteria (**time**, **sequence**, and **value**) to limit replica divergence with
  - ii) techniques based on **locality-awareness** (w.r.t. players position)
- **Selectively and dynamically** strengthens/weakens replica consistency based on the ongoing app state while elegantly managing:
  - **how the consistency degree changes throughout application (e.g., game) execution w.r.t. each user**, and
  - **how the consistency requirements are specified**


48

48

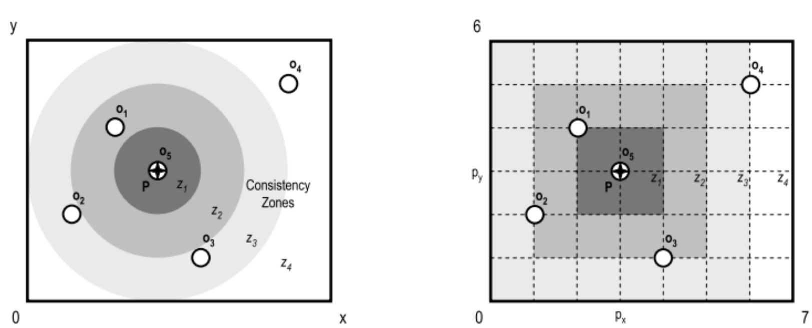
# Vector Field Consistency – observation points

- By employing **locality-awareness** techniques VFC considers that throughout the game execution:
  - there are certain **"observation points"** that we call **pivots** (e.g., the player's position) around which the consistency is required to be strong and weakens as the distance from the pivot increases
  - since **pivots** can change with time (e.g., if the player moves), objects' **consistency needs can also change with time**
- It provides a three-dimensional vector for specifying consistency degrees, where each dimension bounds the replica divergence in:
  - **time** (delay),
  - **sequence** (number of operations), and
  - **value** (magnitude of modifications) constraints
- **Programmers (or even app/game designers)** can parameterize VFC by specifying both the **pivots** and the **consistency degrees** according to game logic





# Vector Field Consistency




**a.** Conceptual consistency zones. **b.** Simplified consistency zones.

# Session Guarantees

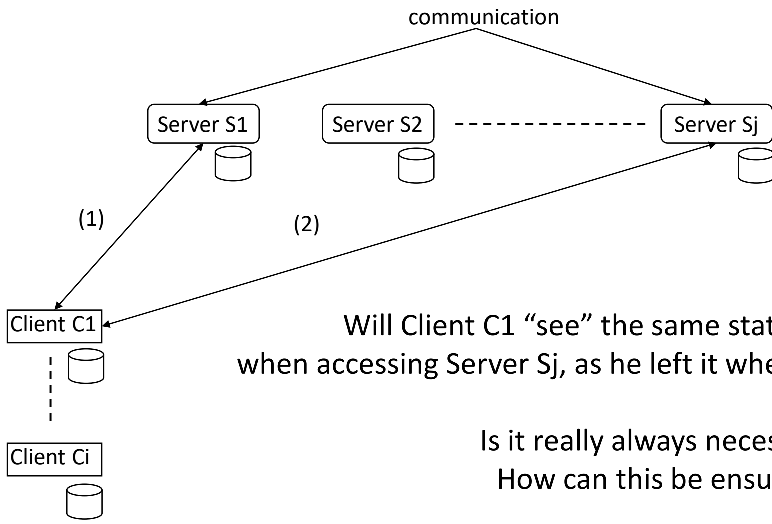




## Session Consistency

- One **potential problem** faced by users who **access data from multiple devices** is they **may observe data that fluctuates in its staleness**:
  - 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 user (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 causal consistency**, which is a system wide property, **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

# System Global View




Would like to have the system behaving as if there was a single server and database (e.g., master replication with a single server)

Will Client C1 “see” the same state of the session when accessing Server Sj, as he left it when he acessed Server S1?

Is it really always necessary?
How can this be ensured?

53



# 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**

54

# Session Guarantees (1/3)

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

55

55

# Session Guarantees (2/3)

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

avoid: 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**

avoid: on a calendar recently added (or deleted) meetings may appear to come and go

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

avoid: 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

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

avoid: 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

56

56

# Session Guarantees (3/3)

- 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**

57

57

# Terminology (1/7)

- Basic assumption:
  - a **weakly consistent replicated storage system** to which the guarantees will be added
  - it consists of **a number of servers each holding 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
- The term "**database**" is not meant to:
  - **imply any particular data model or organization**, nor
  - are the techniques specific to any data model
- A **database** is simply:
  - a **set of data items**,
  - a data item **can be anything** from a conventional file to a tuple in a relational database

58

58

# Terminology (2/7)

- Two main operations on a database are considered:
  - **Read** and **Write**
- **Read** operation represents a **query** over the contents of the database:
  - a Read could be a simple retrieval operation such as "return the contents of file foo" or a complicated query such as "return the names of all employees who live in Oslo"
- The **Write** operation **updates the database**:
  - a Write may involve creating, modifying, or deleting data items
  - it may also represent a transaction that atomically updates multiple items in a server's database

59

59

# Terminology (3/7)

- 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

60

60

# Terminology (4/7)

- 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 t 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 their effect on the database is unchanged**
- The **order of Writes in DB(S) does not necessarily correspond to the order in which server S first received the Writes**

61

61

# Terminology (5/7)

- 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**, also referred to in some papers as rumor mongering, lazy propagation, or update dissemination
  - 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

62

62

# Terminology (6/7)

- **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., the Grapevine system orders Writes by their origination timestamp
  - **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**

63

63

# Terminology (7/7)

- We only assume that:
  - the **system has some means by which Writes are ordered consistently at every server**,
  - as required for eventual consistency, and
  - uses the **WriteOrder predicate to represent this ordering**
- **Weakly consistent systems often allow conflicting Writes to occur**:
  - i.e., two clients may make concurrent and incompatible updates to the same data item
- Existing systems **resolve conflicting Writes in different ways**:
  - in some systems the **Write order may determine which Write "wins",**
  - while other systems rely on **humans to resolve detected conflicts**
- **How the system detects and resolves Write conflicts** is important to its users but **has no impact on the session guarantees**

64

64

RYW: **read** operations reflect previous **writes**

# 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**

65

65

RYW: **read** operations reflect previous **writes**

# 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

66

66

RYW: **read** operations reflect previous **writes**

# 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

67

67

MR: successive **reads** reflect a non decreasing set of **writes**

# 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**
- 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)
- MR-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)**

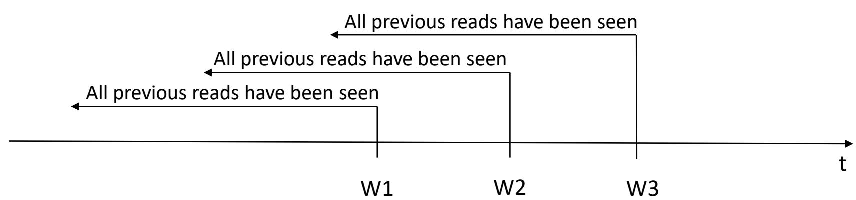
68

68

MR: successive **reads** reflect a non decreasing set of **writes**

# 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




69

69

MR: successive **reads** reflect a non decreasing set of **writes**

# 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**

70

70

WFR: **writes** are propagated after **reads** on which they depend

# 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

71

71

WFR: **writes** are propagated after **reads** on which they depend

# 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 of these properties

72

72

MW: **writes** are propagated after **writes** that logically precede them

# 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**

73

73

MW: **writes** are propagated after **writes** that logically precede them

# 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 after N+1**

74

74

MW: **writes** are propagated after **writes** that logically precede them

# 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

75

75

# Session Guarantees - summary

- **Read Your Writes**:
  - the effects of any Writes made within a session are visible to Reads within that session
- **Monotonic Reads**:
  - Read operations are made only to database copies containing all Writes whose effects were seen by previous Reads within the session
- **Writes Follow Reads**:
  - 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
- **Monotonic Writes**:
  - a Write is only incorporated into a server's database copy if the copy includes all previous session Writes

76

76

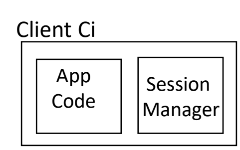
# Providing the Session Guarantees





## 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**:
  - it is a **component of the client stub** that mediates communication with available servers
  - through which all of a session's Read and Write operations are serialized
- For each session, the session manager maintains two sets of WIDs:
  - **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 Read Your Writes

RYW: **read** operations reflect previous **writes**

Ex: it avoids that after changing his password, a user would occasionally type the new password and receive an "invalid password" response

- 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**

| Guarantee | session state updated on | session state checked on |
|---|---|---|
| *Read Your Writes* | Write | Read |

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

79

79

# Providing Monotonic Reads

MR: successive **reads** reflect a non decreasing set of **writes**

Ex.: it avoids that on a calendar recently added (or deleted) meetings may appear to come and go

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

| Guarantee | session state updated on | session state checked on |
|---|---|---|
| *Monotonic Reads* | Read | Read |

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

80

80

# Providing WFR and MW

WFR: **writes** are propagated after **reads** on which they depend

MW: **writes** are propagated after **writes** that logically precede them

- Providing the Writes Follow Reads and Monotonic Writes guarantees requires:
  - two additional, but reasonable, constraints (C1 and C2) on the servers' behavior
- Constraint 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**
- Constraint C2:
  - anti-entropy (i.e., communication between servers) 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 constraints 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

WFR: **writes** are propagated after **reads** on which they depend

Ex: consider a 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

- 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 checks that this read-set is a subset of DB(S,t)**

| Guarantee | session state updated on | session state checked on |
|---|---|---|
| *Writes Follow Reads* | Read | Write |

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

MW: **writes** are propagated after **writes** that logically precede them

# Providing Monotonic Writes

Ex: it avoids that 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

- It says 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**

| Guarantee | session state updated on | session state checked on |
|---|---|---|
| *Monotonic Writes* | Write | Write |

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 |

- **Read Your Writes**:
  - the effects of any Writes made within a session are visible to Reads within that session
- **Monotonic Reads**:
  - Read operations are made only to database copies containing all Writes whose effects were seen by previous Reads within the session
- **Writes Follow Reads**:
  - 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
- **Monotonic Writes**:
  - a Write is only incorporated into a server's database copy if the copy includes all previous session Writes

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

# 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)

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



# 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 (1/2)

- 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

89

89

# Performance Improvement (2/2)

- **Caching of data at clients can also be used to improve overall performance and data availability**
- Data may be available in the cache but:
  - cannot be read by an application because it does not meet the **application's session guarantees**
  - such a situation can arise when applications with **different consistency requirements** are sharing the cache
- Example:
  - suppose a client machine is executing two applications, a **mail reader** and a **program that collects statistics** on the mail messages that the user receives
  - the **statistics gathering program has no consistency requirements** and hence requests no session guarantees
  - the **mail reader requires the Monotonic Reads and Read Your Writes guarantees**
  - if at some point the **statistics program reads from a server that holds an outdated copy of the user's mail database**, thereby filling the cache with old data
  - when the **mail reader executes**, allowing it to **retrieve data from the cache would likely violate its Monotonic Reads guarantee**
- This type of scenario can occur for any weakly consistent replicated system with client caching, regardless of the existence of mobile clients

90

90

# Other Issues



91

## Relevant Issues

- The designer of a replication protocol must deal with:
  - Consistency: What consistency guarantees are desired and how are they provided?
  - Update format: Do replicas exchange data items or update operations?
  - Change tracking: How do devices record updates that need to be propagated to other devices?
  - Metadata: What metadata is stored and communicated about replicated items?
  - Sync state: What state is maintained at a device for each synchronization partnership?
  - Change enumeration: How do devices determine which updates still need to be sent to which other devices?
  - Communication: What transport protocols are used for sending updates between devices?
  - Ordering: How do devices decide on the order in which received updates should be applied?
  - Filtering: How are the contents of a partial replica specified and managed?
  - Conflicts: How are conflicting updates detected and resolved?

representing, recording, sending, and ordering updates



92

# Other Issues – Partial Replication

- The replication protocols and techniques previously described ignore issues that arise with partial replicas:
  - How do devices decide which items to retain in their partial replicas, i.e., specify the items of interest?
  - When and where are filters applied to items propagating between replicas?
  - What happens when a device changes its interests?
  - What happens when items are updated causing them to move in or out of a device's interest set?
  - What constraints must be placed on synchronization topologies that include partial replicas?

access-based caching, policy-based hoarding, topic-based channels, hierarchical sub-collections, content-based filters, context-based filters





# Other Issues – Conflicts Management

- What is a conflict?
- Conflict detection:
  - no detection, version histories, previous versions, version vectors, made-with-knowledge, read-sets, operation conflict tables, integrity constraints, dependency checks
- Conflict resolution:
  - how conflicts can be resolved, where are conflicts resolved, who/what resolves the conflicts

# Examples

| SYSTEM | DATES | DATA | MODEL | CONSISTENCY | PROTOCOL | CONFLICTS |
|---|---|---|---|---|---|---|
| Coda (CMU) | 1987– | Files | Client–server | Weak while disconnected, isolation-only transactions | multiRPC to servers, log reintegration after disconnection | Per-file timestamps, automatic conflict resolvers |
| Ficus, Rumor, Roam (UCLA) | 1990–1998 | Files | Peer-to-peer | Weak | Best-effort multicast plus state-based anti-entropy | Per-file version vectors, automatic conflict resolvers |
| Bayou (Xerox PARC) | 1992–1997 | Databases | Peer-to-peer | Eventual, session guarantees | Pairwise log reconciliation | Per-update application-specific dependency checks and merge procedures |
| Sybase iAnywhere | ~1995– | Databases | Client–server | Weak while disconnected | Timestamp-based row exchange | Per-row previous version, automatic conflict resolvers |
| Microsoft Sync Framework (MSF) | 2001– | XML, databases, files | Peer-to-peer | Eventual | Pairwise state-based reconciliation | Per-replica version vectors, manual or automatic resolution |

95

95

# Conclusions

- Seamless computing requires that people have ready access to their data at any time from anywhere
- Centralized approach to data management is infeasible:
  - non-uniform network connectivity and latencies as well as device limitations and regulatory restrictions
- “optimistic” or “update-anywhere” replication:
  - which replicas are allowed to behave autonomously
- System models:
  - basic definitions, device-master replication, P2P, pub-sub
- Data consistency
  - strong, weak, best effort, eventual, causal, bounded, VFC, session guarantees
- Other issues:
  - protocols, partial replication, conflicts management, examples

96

96