Sistemas de Computação Móvel e Ubíqua

2021/2022

Data Management

2021/2022

Context

Mobile devices tend to have:

- Constrained resources (battery, storage)
- Variable network conditions
- Limited bandwidth
- High latency
- Periods of disconnection

Challenges for data management:

- Provide high availability
- Provide responsive data access

3

Disconnected operation

Disconnection

- Expected
  - Due to communication costs, energy saving, location changes, etc.
- Unexpected
  - Due to network unavailability, network congestion, server failures, etc.

Disconnected operation consists in allowing applications to continue operating during periods of disconnection

- Necessary to prepare for disconnection while connected
- Design applications for Offline-first or Local-first

Ultimate goal: **hide disconnection** from users

- Can be provided by applications or by the system.

4

Challenges #1

How to ensure that clients can access data despite variable connectivity (and even periods of disconnection)?

Clients need to keep copies of the data locally.

Challenges #2

How to ensure that clients have responsive data access and minimize resource consumption?

Clients need to rely on the local data copies even while connected. Need to keep local data copies up-to-date.

Challenges #3

Data needed in mobile devices is often location and context-dependent.

- E.g. In a navigation application, users are interested in the areas close to the users.

How to access the relevant data?

Caching/replication algorithms need to take location and context into consideration.

7

Caching and Replication

Key technique to improve performance and availability (and provide support for disconnected operation).

Caching (and replication) consist in creating and maintaining several distributed copies of data used by applications.

The goal is to attempt to guarantee that most data requests are serviced from near data replicas for…

- Providing availability even for disconnected nodes.
- Improving performance by servicing requests closer to the clients.

8

Caching vs. Replication

It is common for systems to have two levels of replication.

First-class replicas: **replica**
- Data replicas are long-lived
- Durability depends on first-class replicas

Second-class replicas: **cache**
- Cache replicas are short-lived – created and destroyed when necessary
- Data durability is guaranteed by storing data in (cloud) servers

9

Challenges

What to replicate to ensure low latency?

How to maintain cache/replicas consistency with server or between replicas?

What to replicate to ensure high data availability in the presence of disconnections?

How to manage location-dependent data in the cache?

How to enable cooperation between multiple peer caches?

10

Caching in Mobile Computing

There are many challenges:

- Where to cache?
- What data to cache, when do we cache and for how long?
- How to keep cache up-to-date?
- How to handle concurrent updates?

Where to cache?

Server

- Cache for popular items to improve performance
- minimize cost of retrieving from storage (or remote servers) the same information multiple times

Mobile Device

- Cache data to improve performance and availability
- Provide support for disconnected operation

Proxy Server

- Cache data to improve performance and availability
- Remove storage burden from mobile device
- Device vs Proxy: access pattern, communication cost, update rate, available resources at the device

What to cache?

Caching is used extensively in memory, file systems and distributed systems

Ultimate goal:

- Cache all needed data item
- Cache no additional data item

Traditional caching algorithms:

- **LRU – least recently used**

Cache data items as they are accessed; when needing to free some space, delete the least recently accessed item

Why isn't this enough for supporting mobile devices?

13

Disconnected Operation

Whenever some information is better than no information

When availability is more important than consistency

14

What to cache? Pre-fetching (hoarding)

Pre-fetching consists in the process of populating the cache with data before it is necessary

The term **hoarding** is often used for naming the pre-fetch process executed by mobile nodes prior to disconnection.

15

Pre-fetching (hoarding)

Main issues:

- What data items to hoard?
- When and how often do we perform hoarding?
- How to deal with cache misses?
- How reconcile the cache version of the data item with the version at the server.

16

Pre-fetching: what to pre-fetch?

User defined: the user explicitly specifies the data that should be cached

Automatic process: the system automatically infers the data that needs to be cached for each user from context

- Based on prior access history, access patterns and known activity
  - Recently used objects
  - Groups of objects that are usually accessed together
- Having some help from the application
  - E.g. a delivery person needs the information related with her deliveries – the application can request this information at the beginning of day

17

Pre-fetching: when to pre-fetch?

Data must be pre-fetched during the connected period

For supporting unexpected disconnection, mobile nodes need to maintain the cache populated at all times

- Add/remove items as users interests change
- Keep replicas synchronized

18

Data Dissemination: Interaction Models

Pull-model: (request/reply) each client actively requests data from server

- The most common model
- Responses can be cached
- Pooling to get updates (Consumes energy and burdens servers)

Push-model: servers push data that clients need (or not)

- Interesting for scenarios where there is a broadcast channel that can be shared by multiple clients: Publish/subscribe
  - For example, we can subscribe to a stock ticker, and whenever the stock information is updated, it will be sent to us.
- Asynchronous: decouples clients from servers in time
- Information broadcasting when it is available (can be resource efficient)

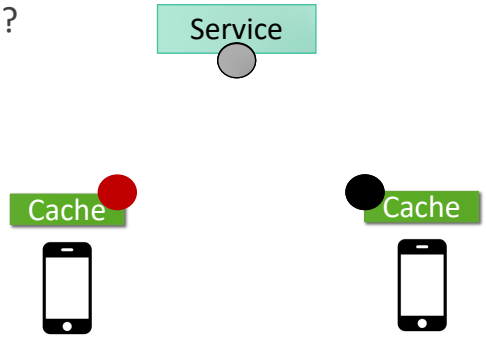
19

Data Maintenance

Data must be kept synchronized with official versions (maintained in servers or other mobile computers)

- Not a problem for read-only data ☺

What about read-write data?



20

Data consistency

Data replication algorithms are typically classified as:

- Strong consistency – (informally) if the system gives the illusion that there is a single replica
- Weak consistency – otherwise, meaning that different copies may (temporarily) present different states

- Weak consistency… by updating the cache more frequently we can minimize the degree of inconsistency.

Cache maintenance in Mobile Computing

The resources of mobile environments, coupled with the need to support seamless mobility, make maintaining a consistent cache at the mobile client a challenging task.

Reasons:

- The underlying cache maintenance protocol should not overburden wireless spectrum and the mobile device.
- Protocols should be energy-efficient, tolerant of disconnections, and adaptive to varying the QoS provided by the wireless network.

Cache consistency requirements:

- Strong consistency: Cache is always up-to-date
- Weak consistency: Cache may not always be up-to-date

Strong consistency precludes disconnected execution and introduces latency on data access

Cache maintenance: client-based

Polling

- On every data access, client polls the server to assess if data has changed (no cache?)
- Data items can have an associated timestamp
  - Realtime clock, logical clock or version vector
- TTL (Time To Live)
  - When caching, the server (or client) assigns a time-to-live validity interval.
  - When the TTL expires, the client polls the server to assess if data has changed (immediately or on the next data access)

Polling for strong consistency, must be always connected, generates more network traffic and introduces latency on data access.

TTL introduces weak consistency

23

Cache maintenance: server-based

Servers initiates the cache consistency verification and push invalidation reports when data has been modified

- **Stateless Approach** – server does not maintain info about the cache contents of the clients
  - Invalidation reports published on data modification or periodically (batch)
  - Clients may subscribe some invalidation reports, depending on the data they cache
- **Stateful Approach** – server keeps track of the cache contents of its clients (invalidation reports tailored for each client)
  - Stateful Asynchronous approach - use invalidation reports (callbacks)
    - Using notification services
  - Stateful Synchronous approach - periodic broadcast of invalidation reports (server keeps track of objects that are recently updated and broadcast information to clients periodically)

24

Publish-subscribe system

Publish-subscribe system model:

- Publishers publish events (messages);
- Subscribers subscribe to events – a subscriber will receive all events published that match its subscription.

Different approach for matching:

- Events are published to channels – subscribers can subscribe to a channel;
- Events have tags – subscribers can specify a tag filter to identify which events it is interested on.

25

Invalidation using Publish-subscribe system

How to do it?

- Servers publish an invalidation event whenever there is a change in an object;
- Mobile devices subscribe to the events associated with the object identifiers they cache.

Challenges?

- While disconnected there might be multiple messages regarding the same object.

26

E.g. Disconnected Execution Asynchronous Stateful Invalidation

- *Frank Adelstein, et. al. Fundamentals of Mobile and Pervasive Computing. McGraw-Hill. 2005. sec. 3.4.5*

Asynchronous invalidation reports when data changes.

Addresses the problem of how to manage invalidation reports when the mobile device is disconnected

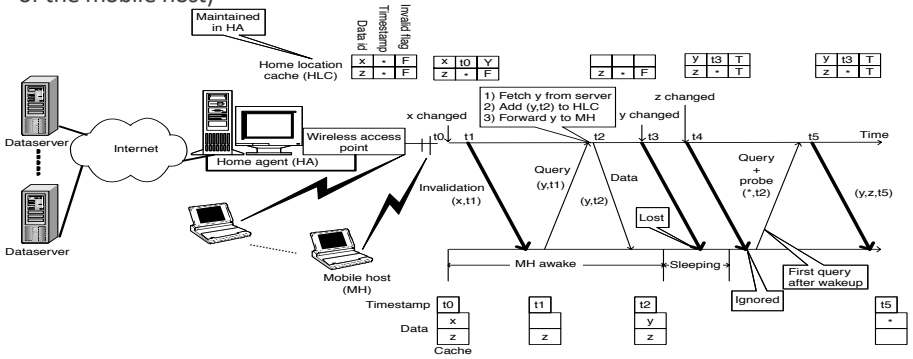
Components:
- Home Agent (HA)
  - Can be maintained at any thrusted static host
  - Closer to the mobile host for improving performance
- Home Location Cache (HLC)
  - Keeps the last invalidation timestamp of every object in cache of a mobile
- Mobile host (MH)
  - Two modes: awake and asleep (connected/disconnected)

For each mobile host a HLC is maintained by its HA

27

E.g. Home location cache

Home Agent (HA)
- Can be viewed as a Proxy
- Pass all messages between the Mobile Host and Data server to assist with handling disconnections
- Keep track of what data have been locally cached at its mobile hosts (cache state info of the mobile host)

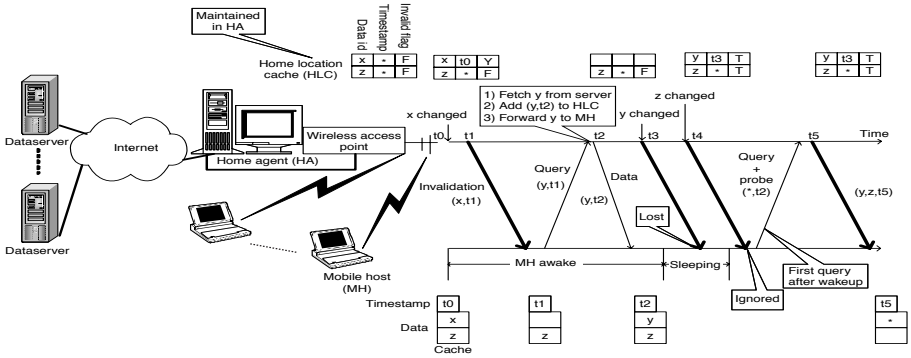


28

E.g. Home location cache (cont.)

Home Location Cache (HLC) for each MH

- Buffers invalidation messages
- Keeps the last invalidation timestamp of every object in MH cache

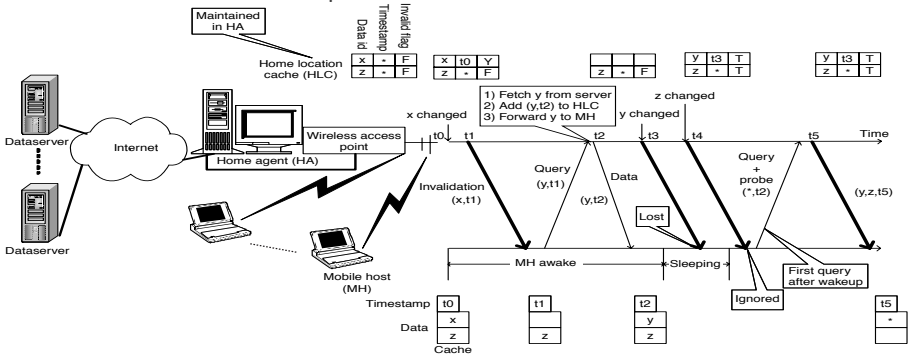


E.g. Home location cache (cont.)

When the data server updates any data items, it sends an Invalidation Message to all Home Agents

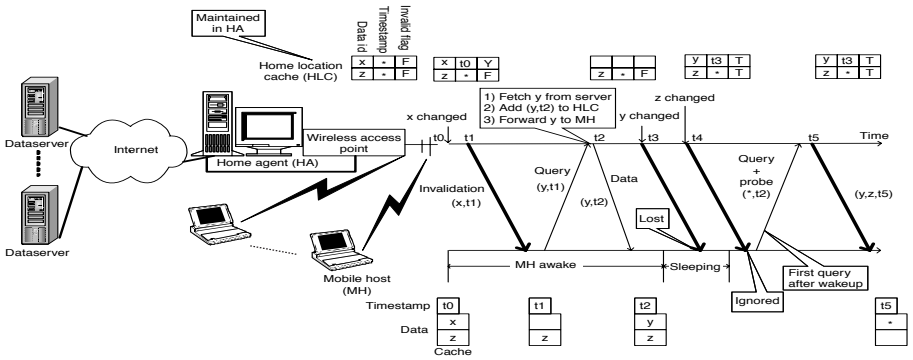
When HA receives an invalidation report

- updates HLC info regarding the data item with invalidation tag specifying whether an invalidation report has been sent to the mobile device or not.



E.g. Home location cache (cont.)

After a disconnection, a mobile device need to ask its HLC whether there is any missing invalidation report or not.



E.g. Home location cache (cont.)

An Example Scenario

MH Cache with timestamp t0 and two data items:

- IDs x and z (x updated at t0)

The HA has received an Invalidation Message (IM) notifying that data item x changed at the server

- At t1 adds IM to HLC and forwards it to mobile host with ID and timestamp (x; t1)
- MH updates its cache with timestamp t1, and deletes x from cache

MH wants to access y it sends a data request (y; t1) to the HLC

- HLC fetches y, and sends it (y; t2)
- MH updates its timestamp to t2, adds y to the cache

E.g. Home location cache (cont.)

MH get disconnected – e.g. sleep mode

T3: y changed, and IM for y is lost

MH Wakeup

- Z changed, and invalidation message for z is sent and ignored
- Ignore all invalidation messages until the 1st query
  - To keep data in sync
- Query + Probe (*, t2) asking for all invalidation messages
- Reply at t5 with batch of Invalidate message (y, z; t5)
- Update cache to t5 and invalidate y and z

33

Cache and local updates

For being responsive and supporting disconnection, mobile system must accept updates in the clients

- Whenever some information is better than no information
- When availability is more important than consistency
- Cache as a secondary replica

Client updates must be sent to the servers (and other clients) asynchronously

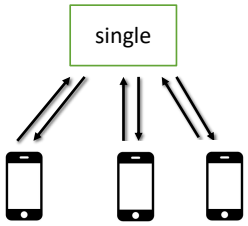
This leads to the need to handle concurrent updates

34

Data management solutions

Single replica service

- Only needs cache management

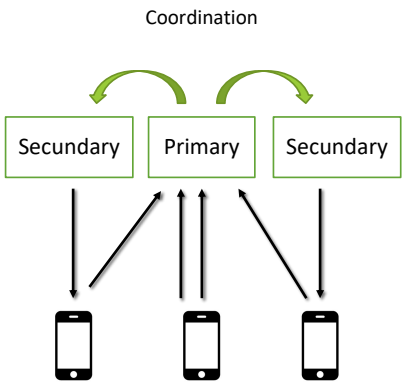


35

Data management solutions

Multiple replicas – pessimistic approach (single master)

- Updates are coordinated; only one update at a time on the master
- Any host may contact any replica , but writes to master
- Simple synchronization process

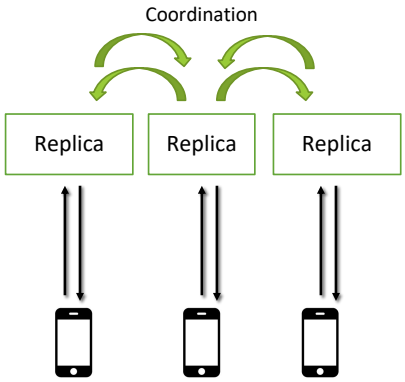


36

Data management solutions

Multiple replicas – optimistic approach

- Uncoordinated updates; users can execute update at any time, any replica
- Good scalability
- May lead to replica divergence due to concurrent updates
- **Need mechanism to detect conflicts and solve conflicts (reconciliation)**



37

Data management: Reconciliation

Desirable properties

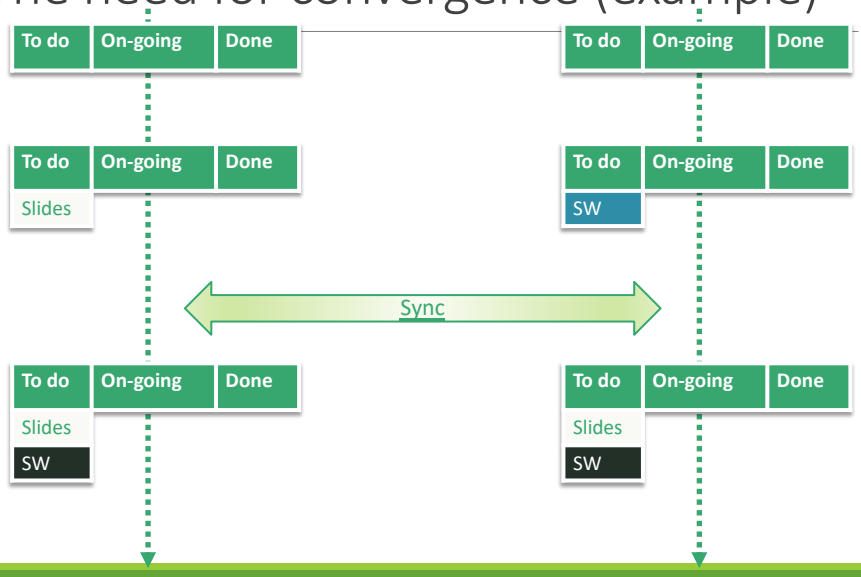
- Eventual convergence
  - When the system is idle all copies converge to the same state.
- Integration of all contributions
- Intention preservation (whenever possible)
  - The effects of the operations must be preserved

Approaches

- State-based
  - Reconciliation mechanism uses different data version as the basis to create a reconciled version
- Operation-based
  - When users execute updates, the system creates a log of operations; these logs are used to merge concurrent updates

38

The need for convergence (example)



39

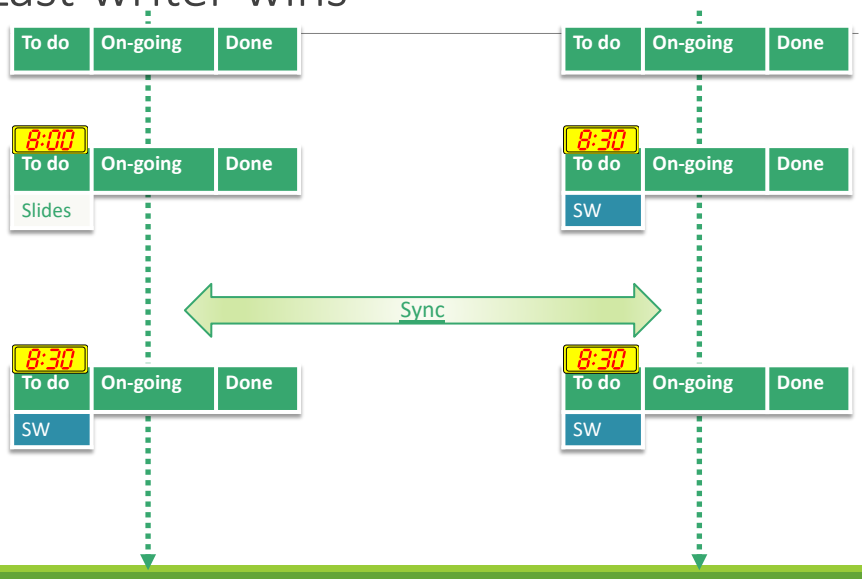
Convergence

Last-writer wins

- Timestamp every update. On concurrent updates, the last update prevails.
- What to version?

40

Last-writer-wins



41

Convergence

Last-writer wins

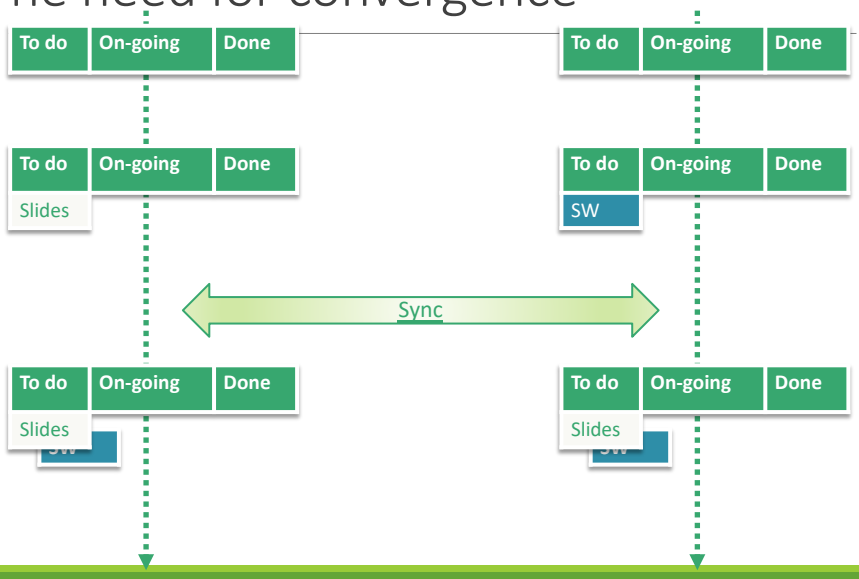
- Timestamp every update. On concurrent updates, the last update prevails.
- Leads to **lost-updates**.
- Variants: *first-writer-wins*, *boss-wins*, *wife-wins*, etc.

Keep multiple versions

- Need to manage and merge multiple version in the application code.

42

The need for convergence



43

Convergence

Semantic merge

- Merge updates taking into consideration the data type.

Alternatives:

- Application-defined conflict resolution policy
  - Allow application to define the conflict resolution policy for each data object
- Operational transformation
  - Used (mostly) for text documents – arrays of characters
- Conflict-free replication data types
  - Used for general data types

44

Operational transformation

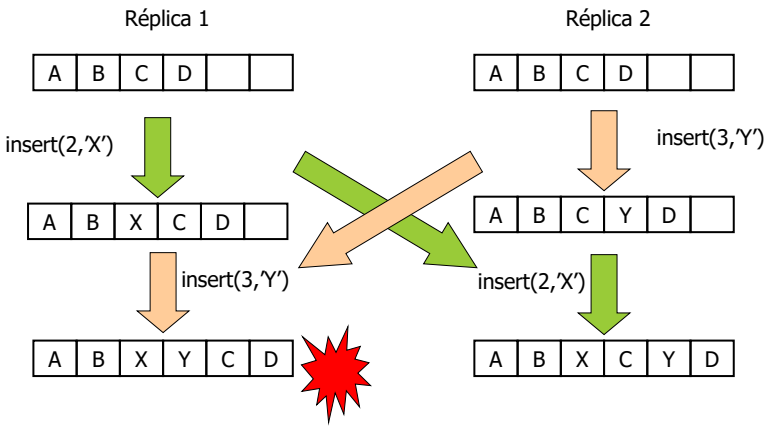
Example: Used in Google Docs

Data model
- Sequence of atoms

Operations
- Insert (atom, pos)
- Remove (pos)

45

Operational Transformation: key idea



46

Operational transformation

Transformation algorithm

- Defines how operations are processed
- To be correct:
  - **Convergence**
  - **Causality**
  - **Intention preservation**

Transformation function

- Defines how an operation is transformed against other operations
- Transformation functions are notoriously hard to design for algorithms supporting peer-to-peer interaction

47

Transformation function examples

Transformations for document editor

T_{ii} (Insert(p1, c1, u1), Insert(p2, c2, u2)) :

```
if p1 <p2 or (p1 = p2 and u1 < u2)
        return Insert(p1, c1, u1 )
else
        return Insert(p1+1,c1,u1)
```

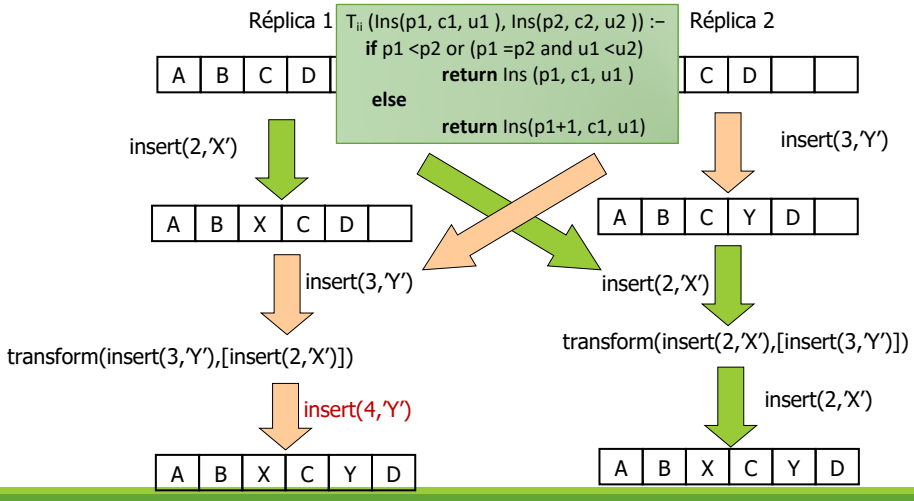
p_i — position
c_i — character
u_i — user

T_{id} (Insert (p1, c1, u1), Delete(p2, u2)) :

```
if p1 ≤ p2
        return Insert(p1, c1, u1 )
else
        return Insert(p1 − 1, c1, u1 )
```

48

Operational transformation: key idea



49

Conflict-free Replicated Data Types

Data type
- Well defined interface.
- Interface as the sequential abstract data type.

Replicated data type
- Objects can be replicated at multiple nodes.

Conflict-free replicated data types
- Replicas can be modified without coordination but...
- Eventual convergence guaranteed in a decentralized way.
  - All replicas converge to the same state after receiving the same set of updates.
- Well defined concurrency semantics.

50

CRDTs for application developers

Key aspect is to define the semantics of the data type

With no concurrent updates, the semantics should be the same of the sequential data type.

With concurrent updates, it is necessary to define the **concurrency semantics**.

- When operation commute, it is possible to provide the same semantics of the sequential data type.
- Often, this is not possible and there is a need for **arbitrate** the result of concurrent executions.
  - Causal-order, total-order, total-causal-order, …

  (for total-order can use some clock, last-writer-wins, etc…)

51

Example CRDT: Register

Interface

- Assign(val)

Concurrency semantics

- Some operations can be ordered, some are concurrent
  - Eventually all replicas have the same value
- Last-writer-wins
  - Keep the value with the highest timestamp (total order)
- Multi-value register
  - Keep all values written concurrently (causal-order)
  - (read returns a list of possible values)

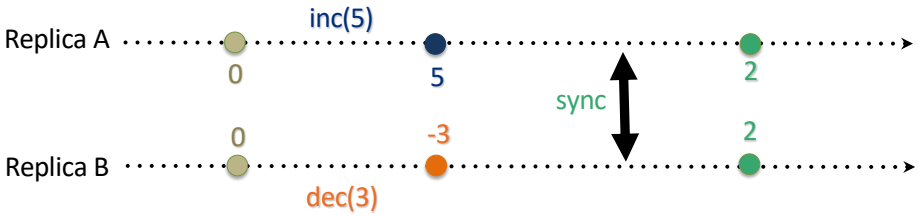
52

Example CRDT: Counter

Starts with 0

Interface:
- inc(n)
- dec(n)

Concurrency semantics
- inc and dec commute.
- Combine all updates
  - adding all increments and subtracting all decrements

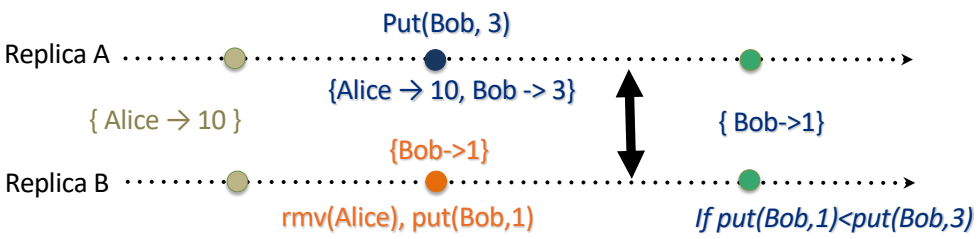


53

Example CRDT: Map

Interface
- put(k, o)
- rmv(k)

Concurrency semantics
- put and rmv for different k commute
- For concurrent on the same k must arbitrate or use multivalue (versions)
- E.g: last writer wins



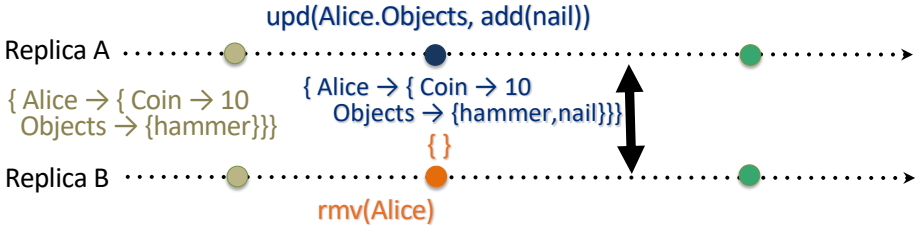
54

Example CRDT: Map of CRDTs

Interface
- put(k, o)
- rmv(k)
- upd(k, *op*) *update internal CRDT*

Concurrency semantics
- Updating a value, use semantics of internal CRDT
- Diferent value types for same k?
  - Eg. k with values of each type (key is k+valueType)

Eg. Map of map of counter and set



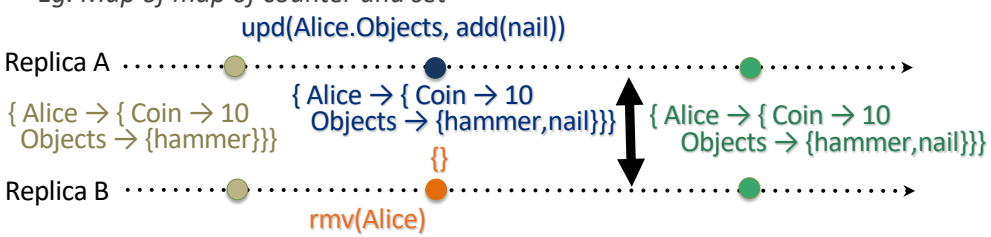
55

Example CRDT: Map of CRDTs

Interface
- put(k, o)
- rmv(k)
- upd(k, *op*) *update internal CRDT*

Concurrency semantics
- Update wins

Eg. Map of map of counter and set



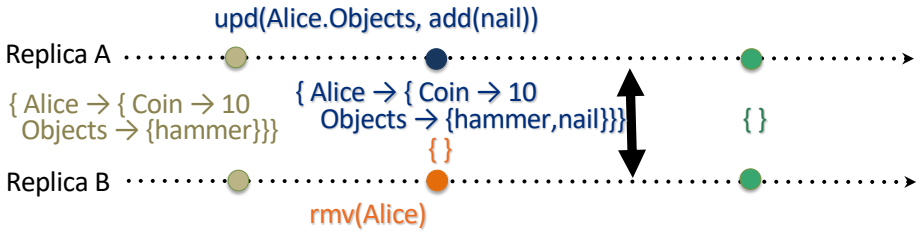
56

Example CRDT: Map of CRDTs

Interface
- put(k, o)
- rmv(k)
- upd(k, *op*) *update internal CRDT*

Concurrency semantics
- ~~Update-wins~~
- Remove-wins

Eg. Map of map of counter and set



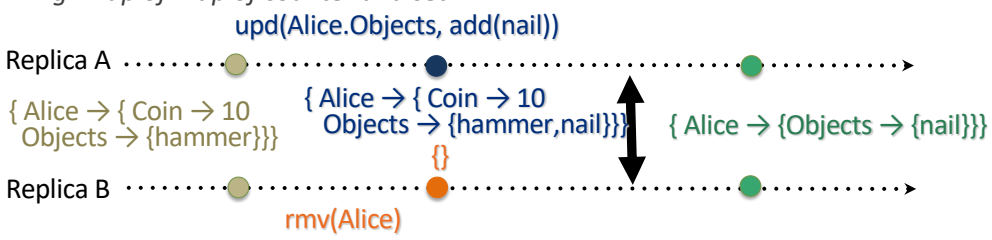
57

Map

Interface
- put(k, o)
- rmv(k)
- upd(k, *op*) *update internal CRDT*

Concurrency semantics
- ~~Update-wins~~
- ~~Remove-wins~~
- Remove as recursive reset
  - Each CRDT supports a reset operation

Eg. Map of map of counter and set



58

Example - Designing a CRDT: Set

Sequential specification of Set:

*S.*add(*e*) *{e ∈ S}*

*S.*rmv(*e*) *{e ∉ S}*

Commutative operations (*e ≠ f*):

*S.*add(*e*) || *S.* add(*e*) *{e ∈ S}*

S. rmv(*e*) || *S.* rmv(*e*) *{e ∉ S}*

S. add(*e*) || *S.* add(*f*) *{e,f ∈ S}*

S. rmv(*e*) || *S.* rmv(*f*) *{e, f ∉ S}*

S. add(*e*) || *S.* rmv(*f*) *{e ∈ S, f ∉ S}*

Non-commutative operations:

S. add(*e*) || *S.* rmv(*e*) *{????}*

59

Design alternatives for *add(e) || rmv(e)*

S. add(*e*) || *S.* rmv(*e*) *{????}*

add wins: *{e ∈ S}*

remove wins: *{e ∉ S}*

error state: *{⊥e ∈ S}*

last writer wins: *{ add(e) < rmv(e) ⇒ e ∉ S,*
rmv(e) < add(e) ⇒ e ∈ S }

60

Not Sequentially Consistent

Consider **Add-wins set** such that:

*S.*add(*e*) *{e ∈ S}*
S. remove(*e*) *{e ∉ S}*
S. add(*e*) || *S.* remove(*e*) *{e ∈ S}*

But…

(add(e); remove(*f*)) || (add(*f*); remove(e)) *{e, f ∈ S}*

Not sequentially consistent!

61

Takeaway lesson

Most operations do not commute.

Concurrency semantics defines the outcome in the presence of concurrent updates.

- Multiple alternatives exist for each data type.

Right concurrency semantics depends on the application.

62

Libraries for building mobile applications

Multiple libraries exist for managing CRDT data of mobile applications, usually supporting peer-to-peer synchronization.

Examples of these libraries are:

- Automerge – supports synchronizing JSON object among multiple clients.
- YJS – supports synchronizing multiple object types among multiple clients.
- Legion – supports synchronizing multiple object types among multiple clients.

Data Management in current mobile phones

Mobile OSs support

- SQL databases
- Key-value pairs
- Files

HTML 5 supports

- Session storage – key/value pairs maintained during one application session
- Local storage – key/value pairs maintained for an application across sessions
- IndexedDB – key/value object-oriented database for Web browsers
  - stores JSON objects with indexes
  - limited support on latter versions of most known Web browsers

Databases Services for mobile applications

Cloud's Mobile platform as a service often include databases with support for caching in mobile device.

- Example: Firebase.

A number of (independent) databases exist supporting mobile applications.

- Example: Couchbase / Pouchdb

Bibliography

Books:

- Frank Adelstein, et. al. Fundamentals of Mobile and Pervasive Computing. McGraw-Hill. 2005. Chap 3. (outdated)

Papers

- Operational Transform **(just the concept):** Imine A., Molli P., Oster G., and Rusinowitch M. (2003) Proving correctness of transformation functions in real-time groupware. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work* (ECSCW'03), 277-293.
- Nuno Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. arXiv:1806.10254 (June 2018). Retrieved January 11, 2021 from http://arxiv.org/abs/1806.10254