IN.5022 — Concurrent and Distributed Computing

Replicated data structures and CRDTs

Prof. P. Felber

pascal.felber@unine.ch

Agenda

- Shared data and the CAP theorem
- Eventual consistency
- CRDTs: properties and definitions
- Examples of CRDTs
- Use and deployment of CRDTs

Shared data

- Distributed applications manage shared state
  - Transient data in memory
  - Persistent data in files
  - Data stored in a database
- Databases are preferred for sizeable and complex data structures with consistency/durability requirements
  - Traditional relational databases
  - New-generation “NoSQL” databases

Strong consistency

- Strong consistency requires operations by multiple clients on some shared data to be strictly sequential
  - All participants see the same order
- Strong consistency in a single-node system is relatively easy…
  - Single copy of the data
  - Supported by traditional RDMS

…but a single-node system is also way more likely to fail

Availability and scalability

- Shared data should be **available**
  - Application components can access (read/write) the data at any time without blocking
  - The data store must **scale** to sustain the client load
- The only way to achieve high-availability is with a distributed system (using **replication**)
  - Replication provides fault tolerance (failover to a replica)
  - Replication provides scalability (share load among replicas)

Replication and consistency

- Replicated data must be maintained **consistent**
  - All replicas *agree* on the latest version of the data
  - This typically requires updates to be propagated to all replicas to maintain consistency (clients should only see up-to-date state)
- There are many consistency models that differ in the guarantees they provide w.r.t. keeping data synchronised
  - Weak consistency models allow the state to (temporarily) diverge
  - Strong consistency models keep all replicas up-to-date at all times
- Strong consistency in a distributed system can be obtained through consensus protocols (e.g., Paxos, Raft, …)
  - Consensus algorithms are costly (low throughput, high latency)

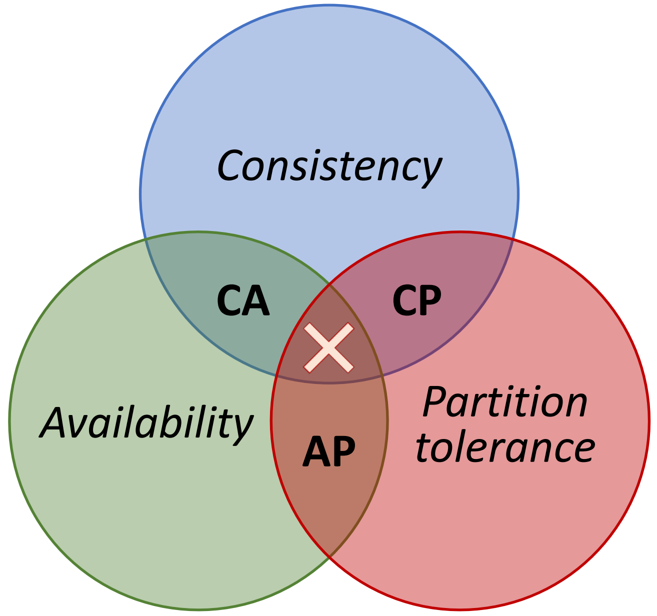
CAP theorem [Eric Brewer (Berkeley) — Seth Gilbert, Nancy Lynch (MIT)]

Three properties of a DB system

1. **Consistency —** all database clients see the same data, even with concurrent updates
2. **Availability —** all database clients are able to access some version of the data
3. **Partition tolerance —** the database can be split over multiple servers

Pick two!

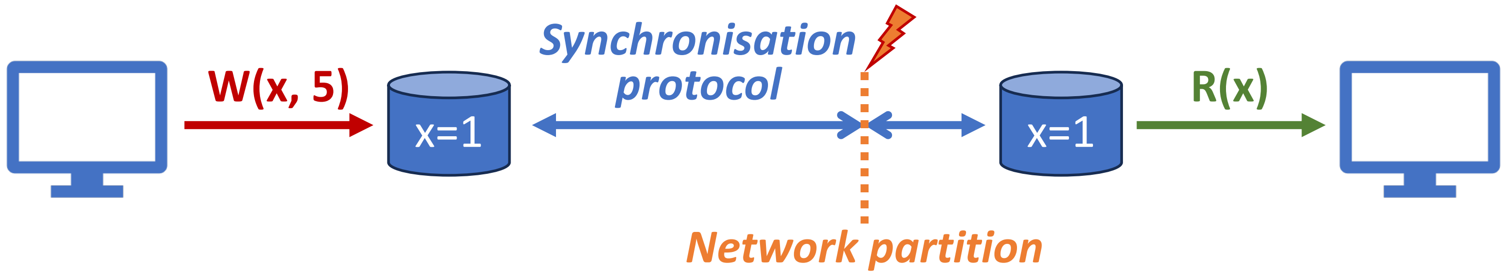
CA: Relational DBMS (MySQL, Postgress, …)
CP: NoSQL (HBase, MongoDB, Redis, …)
AP: NoSQL (Cassandra, CouchDB, Dynamo, Riak, …)



CAP theorem (cont'd)

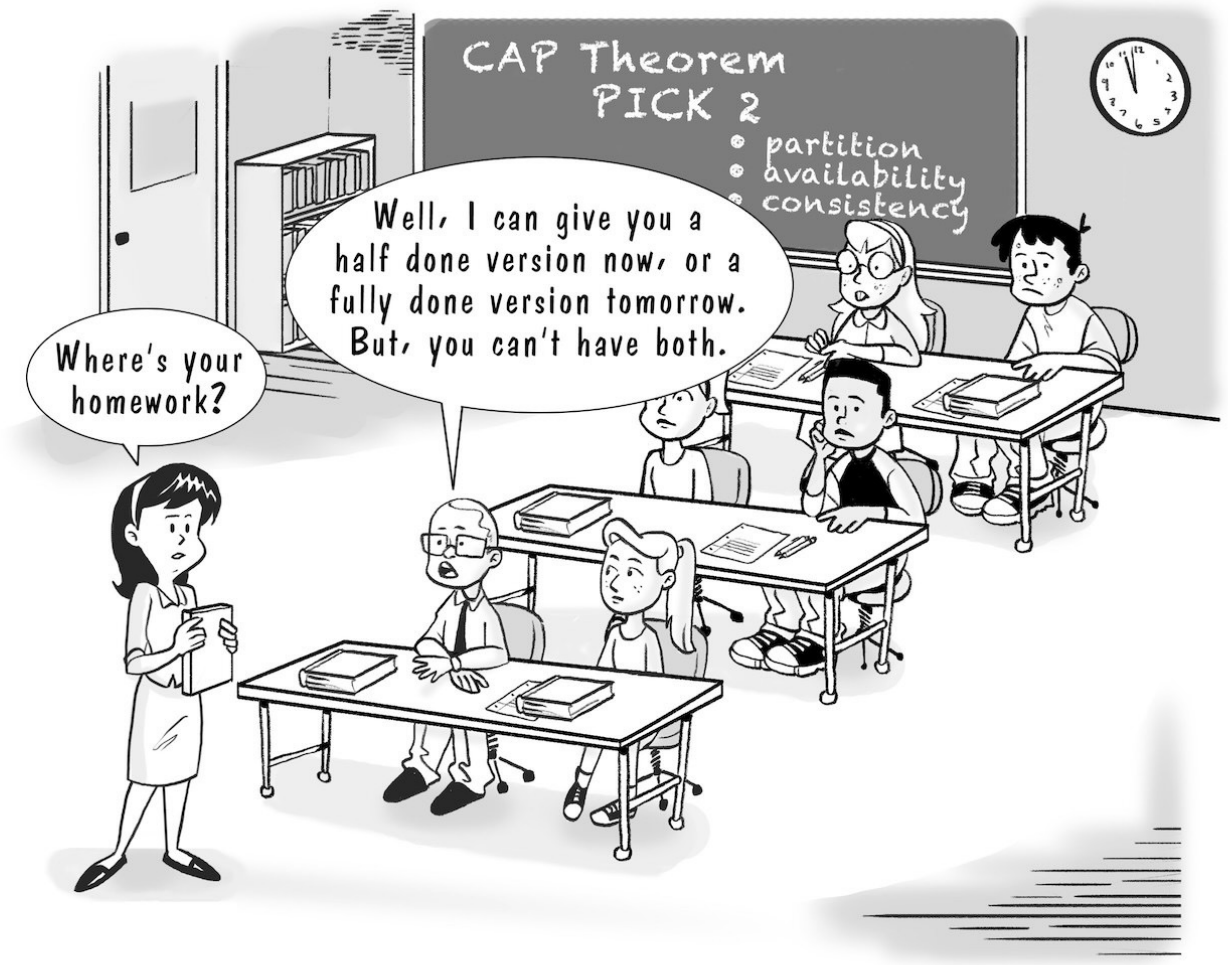
"You can have at most 2 of these 3 properties for any shared-data system"

To scale, you have to partition (replicate data)
⇒ *must choose between consistency or availability*



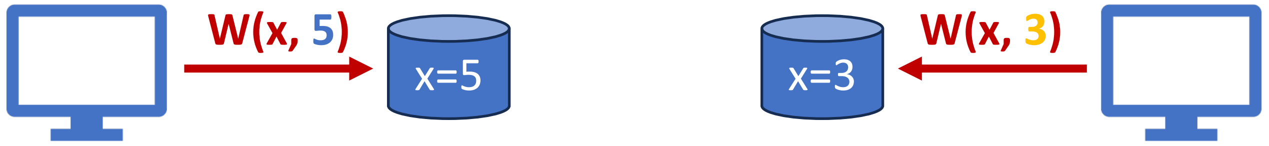
You **can** write ⇒ the system is **available** but (maybe) **inconsistent**
You **cannot** write ⇒ the system is **consistent** but **not available**

CAP theorem (cont'd)

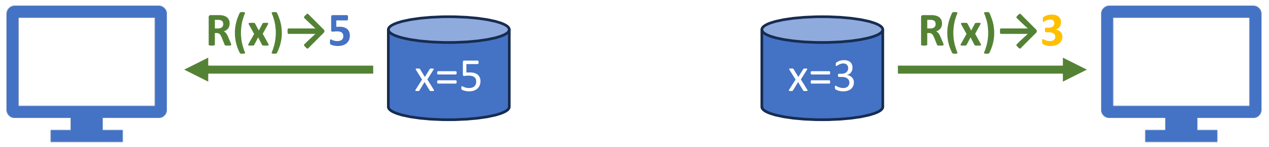

CAP Theorem
PICK 2
partition
availability
consistency
Where's your homework?
Well, I can give you a half done version now, or a fully done version tomorrow. But, you can't have both.

Eventual consistency

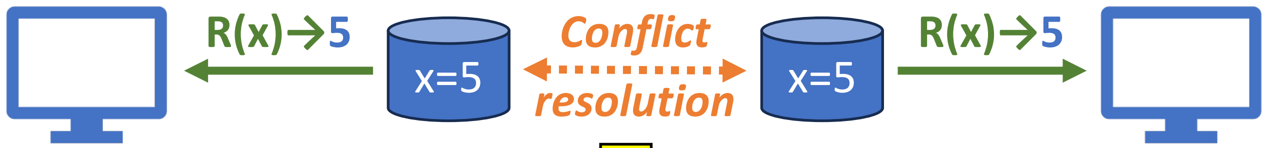
- Consistency is a continuum (weak to strong) with trade-offs
- **Eventual consistency** is a widely used "practical" model
  1. Updates performed locally *without coordination*



  2. Nodes may get *approximative* answers (observe different values)



  3. Changes eventually propagated across the system *(convergence)*



Eventual consistency (cont'd)

- *Conflict resolution* policy is used for state reconciliation
  - Exchange of possible values between nodes
  - Reconciliation to choose the final value between available values
    - No universal approach (e.g., using vector clocks)
  - Can be done asynchronously or before read/write if inconsistency is found

- Eventual consistency guarantees that…

…"when no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent"

ACID vs. BASE

- Two classical models for distributed data stores

ACID	**BASE**
Atomicity	Basic Availability
Consistency	Soft-state
Isolation	Eventual consistency
Durability	

- ACID prefers consistency (traditional RDMBS model)
  - Strong consistency (real-time consensus), vertical scalability
- BASE prefers availability (NoSQL databases)
  - Eventual consistency (deferred consensus), horizontal scalability

From eventual consistency to CRDTs

- *Strong* eventual consistency states that…

  …"any two nodes that have received the same (unordered) set of updates will be in the same state"

- Therefore, reconciliation operations should ideally be…
  - Commutative
  - Associative
  - Idempotent (in case it is applied multiple times)

- **Strong eventual consistency**

  = *eventual consistency*

  + *automatic conflict resolution* "that always works"

CRDTs

- **Conflict-free replicated data types (CRDTs)** are a class of data structures for which it is always possible to merge or resolve concurrent updates on different replicas of the data structure without conflicts
  - Concurrent updates are allowed to go through, possibly creating inconsistencies that are “resolved” later
  - They provide a conflict resolution function that guarantees data convergence
  - They support strong eventual consistency ⇒ optimistic replication
  - They were initially motivated by collaborative text editing and later generalised to support replicated databases

Two types of CRDTs

- State-based CRDTs: **CvRDTs** (convergent replicated data types)
  - Replicas send their full local state to other replicas
  - Messages are delivered at least once (but in any order and possibly multiple times)
  - States are merged by a function which must be commutative, associative and idempotent
- Operation-based CRDT: **CmRDTs** (commutative replicated data types)
  - Replicas propagate state by transmitting only the update operation

State-based CRDTs

- A CvRDT is a tuple `(S, s0, q, u, m)`

`S`	— set of possible states for the CRDT
`s0`	— initial state of the CRDT
`q`	— query function that lets a client read the current object state
`u`	— update function that lets a client alter the object state
`m`	— binary merge function

- Update function `u` must monotonically increase the internal state according to partial order on states (semilattice)
- Merge function `m` needs to be

associative	— `∀x, y, z : m(x, (m(y, z))) = m(m(x, y), z)`
commutative	— `∀x, y : m(x, y) = m(y, x)`
Idempotent	— `∀x : m(x, x) = x`

Operation-based CRDTs (not covered in this lecture)

- With CmRDTs, modifications are described by operations
  - Replicas propagate state by transmitting only the update operation
  - Replicas receive the updates and apply them locally
- Concurrent operations are…

  Commutative, i.e., they can be applied in any order

  Not necessarily **idempotent**

  ⇒ Must ensure that all operations on a replica are delivered to the other replicas without duplication, but in any order

  Theorem: ***"any state-based object can be emulated by an operation-based object of a corresponding interface"*** (and vice versa)

Example 1: G-set

- Grow-only set
  - Clients can add and lookup elements
  - Upon partition, the state of the replicas might diverge
    - Client receive approximate responses (not necessarily up-to-date)
  - When connection goes back, states merge

Alice	*G-Set*		*G-Set*	*Bob*	
|	{}	<---->	{}	|	*1*
add(A)	{A}	<---->	{A}	|	*2*
lookup(A):T	{A}	<---->	{A}	|	*3*
|	{A}	<-//->	{A}	|	*4*
add(B)	{A,B}	<-//->	{A,C}	add(C)	*5*
lookup(A):T	{A,B}	<-//->	{A,C}	lookup(C):T	*6*
lookup(C):F	{A,B}	<-//->	{A,C}	lookup(B):F	*7*
|	{A,B,C}	<---->	{A,B,C}	|	*8*
lookup(C):T	{A,B,C}	<---->	{A,B,C}	lookup(B):T	*9*

Example 1: G-set (cont'd)

https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

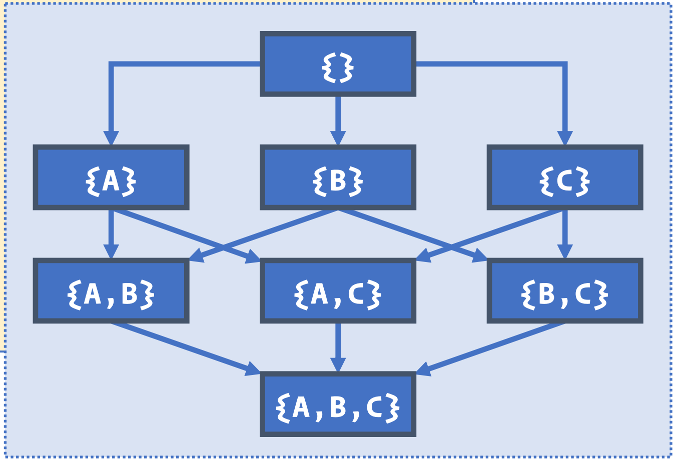
```
payload set A
    initial ∅

update add(element e)              Monotonically increases state (w.r.t. `compare` function)
    A := A ∪ {e}

query lookup(element e) : boolean b
    let b = (e ∈ A)

merge (S, T) : payload U
    let U.A = S.A ∪ T.A

compare (S, T) : boolean b         Partial order on states
    let b = (S.A ⊆ T.A)
```


$$\begin{tikzcd}
 & \{\} \arrow[dl] \arrow[d] \arrow[dr] & \\
\{A\} \arrow[d] \arrow[dr] & \{B\} \arrow[dl] \arrow[dr] & \{C\} \arrow[dl] \arrow[d] \\
\{A,B\} \arrow[dr] & \{A,C\} \arrow[d] & \{B,C\} \arrow[dl] \\
 & \{A,B,C\} &
\end{tikzcd}$$

Example 2: G-counter

- Grow-only counter
  - Counter for a cluster of *n* nodes implemented with an array
  - Each node is assigned an identifier from 0 to *n* – 1 and has its own slot in the array **P**, which it increments locally
  - Updates are propagated in the background and merged by taking the maximum of every element in **P**
  - The value of the counter is the sum of all entries in the array

Example 2: G-counter (cont'd)

https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

```
payload integer[n] P
    initial [0,0,...,0]

update increment()                    Monotonically increases state (w.r.t. `compare` function)
    let g = myId()
    P[g] := P[g] + 1

query value() : integer v
    let v = sum(P)

merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])

compare (X, Y) : boolean b                                          Partial order on states
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i])
```

Example 3: PN-counter

- How can we implement a counter that can grow and shrink?
  - By combining two G-counters
- Positive-negative counter
  - One G-counter for increments (P)
  - One G-counter for decrements (N)
  - The value of the counter is the difference of the two G-counters
  - Merge is handled by separately merging the two P and N counters
  - Note that the CRDT’s internal state still increases monotonically, (even though the external state exposed through query returns increasing and decreasing values)

Example 3: PN-counter (cont'd)

https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

```
payload integer[n] P, integer[n] N
    initial [0,0,...,0], [0,0,...,0]

update increment()            Monotonically increases state (w.r.t. `compare` function)
    let g = myId()
    P[g] := P[g] + 1

update decrement()            Monotonically increases state (w.r.t. `compare` function)
    let g = myId()
    N[g] := N[g] + 1

query value() : integer v
    let v = sum(P) - sum(N)

merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
    let ∀i ∈ [0, n - 1] : Z.N[i] = max(X.N[i], Y.N[i])

compare (X, Y) : boolean b                          Partial order on states
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i] ∧
             ∀i ∈ [0, n - 1] : X.N[i] ≤ Y.N[i])
```

Example 4: 2P-set

- How can we implement a set supporting removals?
  - By combining two G-sets
- Two-phase set
  - One G-set for elements added (A)
  - One G-set for elements removed (R), also called “tombstone” set
  - Elements must be added before being removed
  - Once removed, elements cannot be re-added, i.e., the query will never return `true` again for those elements (“remove-wins" semantics)
  - Merge is handled by separately merging the two A and R sets

Example 4: 2P-set (cont'd)

https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

```
payload set A, set R
    initial ∅, ∅

update add(element e)              Monotonically increases state (w.r.t. `compare` function)
    A := A ∪ {e}

update remove(element e)           Monotonically increases state (w.r.t. `compare` function)
    pre lookup(e)
    R := R ∪ {e}

query lookup(element e) : boolean b
    let b = (e ∈ A ∧ e ∉ R)

merge (S, T) : payload U
    let U.A = S.A ∪ T.A
    let U.R = S.R ∪ T.R

compare (S, T) : boolean b                                  Partial order on states
    let b = (S.A ⊆ T.A ∧ S.R ⊆ T.R)
```

More useful CRDTs

- By adding timestamps to elements, one can support more complex CRDTs
  - LWW-element-set supports re-insertion of removed elements
- One can generalise the composition approach to create **sequence CRDTs** (sequence, list, ordered set, …)
  - Sequence CRDTs can be used to build collaborative real-time editors where several users concurrently modify a shared text
- Many CRDT libraries are available
  - Yjs [https://yjs.dev]
  - Automerge [https://automerge.org]

    …

Use of CRDTs [wikipedia]

- CRDTs are used by many distributed databases and web frameworks
  - Redis: globally distributed, highly available and scalable database
  - Riak: distributed NoSQL key-value data store written in Erlang (used for instance by LoL for its in-game chat system)
  - Phoenix: web framework written in Elixir (used for information sharing)
- CRDTs are used by companies for industrial applications
  - @ Facebook: in the *Apollo* database, in the *FlightTracker* system (for managing the Facebook graph internally)
  - @ TomTom: to synchronise navigation data between the devices of a user
  - @ Apple: in the *Notes* application for syncing offline edits between multiple devices

Summary

- Distributed applications usually manage shared state
- Scalability is key for dealing with large numbers of clients
  - Requires data to be replicated (and geographically distributed)
  - Data stores should be **consistent**, **available** and **partition-tolerant**
- All 3 properties cannot be guaranteed at once (CAP theorem)
  - RDBMs usually focus on CA, NoSQL databases on CP/AP
- By weakening consistency, one can ensure all 3 properties
  - Eventual consistency is a common model: *"when no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent"*

Summary (cont’d)

- A CRDT is a data structure that is replicated across multiple computers in a network, with the following features [wikipedia]
  1. The application can update any replica independently, concurrently and without coordinating with other replicas
  2. An algorithm (itself part of the data type) automatically resolves any inconsistencies that might occur
  3. Although replicas may have different states at any particular point in time, they are guaranteed to eventually converge
- Support strong eventual consistency ⇒ *optimistic replication*
- Two classes: state-based and operation-based

Summary (cont'd)

- CRDTs are useful building blocks
  - Simple structures: G-counter, G-set
  - Can be composed into more complex ones: PN-counter, 2P-set
  - Sequence CDRTs can be used for online applications (e.g., collaborative editing, databases, …)
  - Several CRDT libraries available, used in many open-source and industrial distributed systems