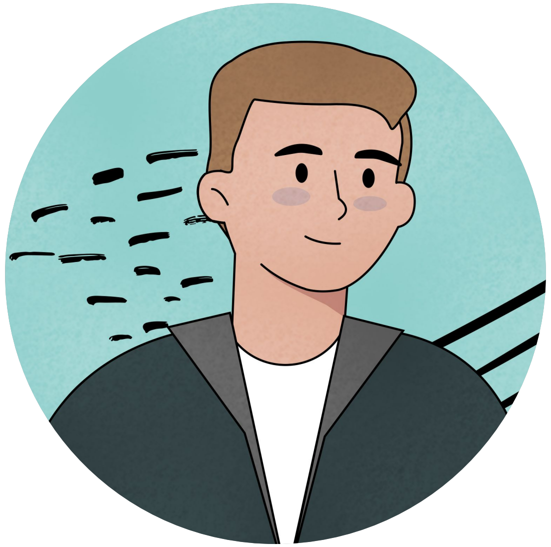
A Local-First Collaborative Modeling Approach with Replicated Data Types

Léo Olivier
Kirollos Morcos
Marcos Didonet Del Fabro
Sébastien Gérard

The Team



Léo Olivier

2^{nd} year PhD student

Kirollos Morcos

Master's intern

Marcos Didonet Del Fabro

CEA research engineer, PhD co-advisor

Sébastien Gérard

CEA research director, PhD director

1. Background & Motivation

Challenges of Collaborative Modeling

- Lack of support for collaboration features in existing tools[(2)], **despite growing modeling practitioners needs**.
- Shift in user behavior.
  - More nomads, use multiple devices…
- Is the separation between **asynchronous** and **synchronous** collaboration still relevant?

12 Popular modeling tools

Only **6/12** Collaboration support

Only **3/12** Automatic conflict resolution

Feature	Feature group	%
Model browsing/search	Model manipulation and query	100
Multi-view modeling	Models and languages	98
Collaboration at model level	Models and languages	98
Visual editors	Editors and modeling environments	95
Model validation	Model manipulation and query	95
History	Versioning	95
Real-time collaboration	Collaboration dynamics	95
Model merging	Versioning	95
Role-based access control	Stakeholder management	93
Screen sharing	Synchronous communication	93

The most needed techniques for collaborative modeling[(1)].

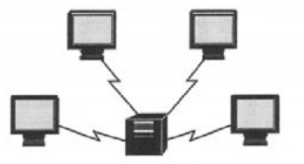
(1) Collaborative Model-Driven Software Engineering—A systematic survey of practices and needs in industry, David et al. (2023)

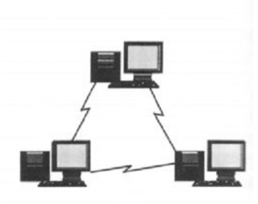
(2) Real-time Collaborative Multi-Level Modeling by Conflict-Free Replicated Data Types, David, Syriani (2022)

The Case for Hybrid Collaboration Workflows

- A company that develops **wind turbines** operates divisions in Europe and America.
- Teams use models to coordinate.
- Real-time collaboration *within* a division.
- Asynchronous collaboration *between* divisions due to time zones.

Traditional Collaboration Architectures





	<u>**Central server**</u>	<u>**Git-like solution**</u>
Pros	• Simple architecture: only need an API to query & update the model. • Server linearizes all updates: no conflicts management needed.	• Supports asynchronous collaboration. • Each member keeps a local copy of the shared model. • Changes can be merged later.
Cons	• **No support** for asynchronous collaboration. • High latency degrades user experience (**rollbacks**). • **Not fault-tolerant** (server is a single point of failure). • **Data breach** could expose intellectual property.	• No support for real-time collaboration. • **Conflicts** must be resolved **manually**, usually at the syntax level (not semantic). • **Merge** process can be **slow** and **error-prone** (Higher cognitive load for users).

2. Proposal

Local-First Collaborative Modeling

- **Local-First[1] collaborative modeling:**
  - Models are stored locally on user device
    - Always available.
  - Automatic model merging
    - Seamless collaboration.
  - Preservation of contributions
    - No work is lost in case of conflicts during a merge.
  - Support for hybrid collaboration workflow
    - Users can work online or offline.

Local-First principles

Availability

Multi-device

Offline capabilities

Multiple collaboration workflows

Data preservation

Privacy

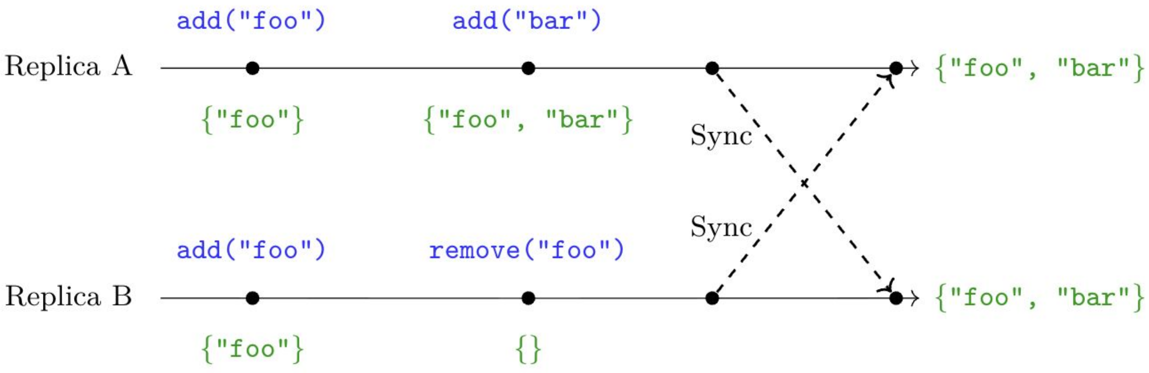
Data ownership

(1) *Local-First Software: You Own Your Data, in spite of the Cloud, Kleppmann et al. (2019)*

Conflict-free Replicated Data Type (CRDT) (1)

$$\text{conflict}((t,o),(t',o')) = t \parallel t' \wedge \neg\text{commute}(o,o')$$

Two concurrent operations that do not commute are in conflict.



Time diagram of a set CRDT execution. The "*Add-Wins*" policy states that in case of conflict between a concurrent `add` and `remove` on the same element, the remove has no effect and the element remains in the set.

- Replicated objects for asynchronous distributed systems.
- Encapsulate a **conflict resolution policy**.
  - Not really "conflict-free"…
- **Automatic merge** of concurrent updates.
- Replicas automatically **converge** towards a **common state** without coordination.

(1) Conflict-free Replicated Data Types, Shapiro et al. (2011)

Achieving Local-First Collaborative Modeling

- Requirement for a CRDT model adapted to the needs of Local-First collaborative modeling:
  - **Easy to specify and implement** new CRDTs tailored to modeling needs.
    - Impossible in existing libraries.
  - Allow the specification of **semantic conflict resolution policies** suitable for asynchronous collaboration.
    - No “Last-Writers-Wins”!
  - Support **composition** and **nesting** pf CRDTs for complex models.

An Extensible Model for Domain-specific CRDTs

- Operations are stored in a **generic partially-ordered log** (PO-Log).
  - The log *is* the CRDT state.
- Operations are delivered to replicas in **causal order** using a RCB component.
- A data type-specific `eval(query,state)` function **interprets the PO-Log** to produce the actual CRDT value.

Pure operation-based model[1]

Genericity

Extensibility

(1) *Pure Operation-Based Replicated Data Types*, Baquero et al. (2017)

An Extensible Model for Domain-specific CRDTs

- **Prune redundant operations**: R_{self} and R_{by} relations.
  - R_{self}: already redundant upon delivery ;
  - R_{by} : old op is made redundant by the delivery of a new op.
- New CRDT = specifying:
  - Set of operations ;
  - `eval(q, s)` function ;
  - Redundancy relations.

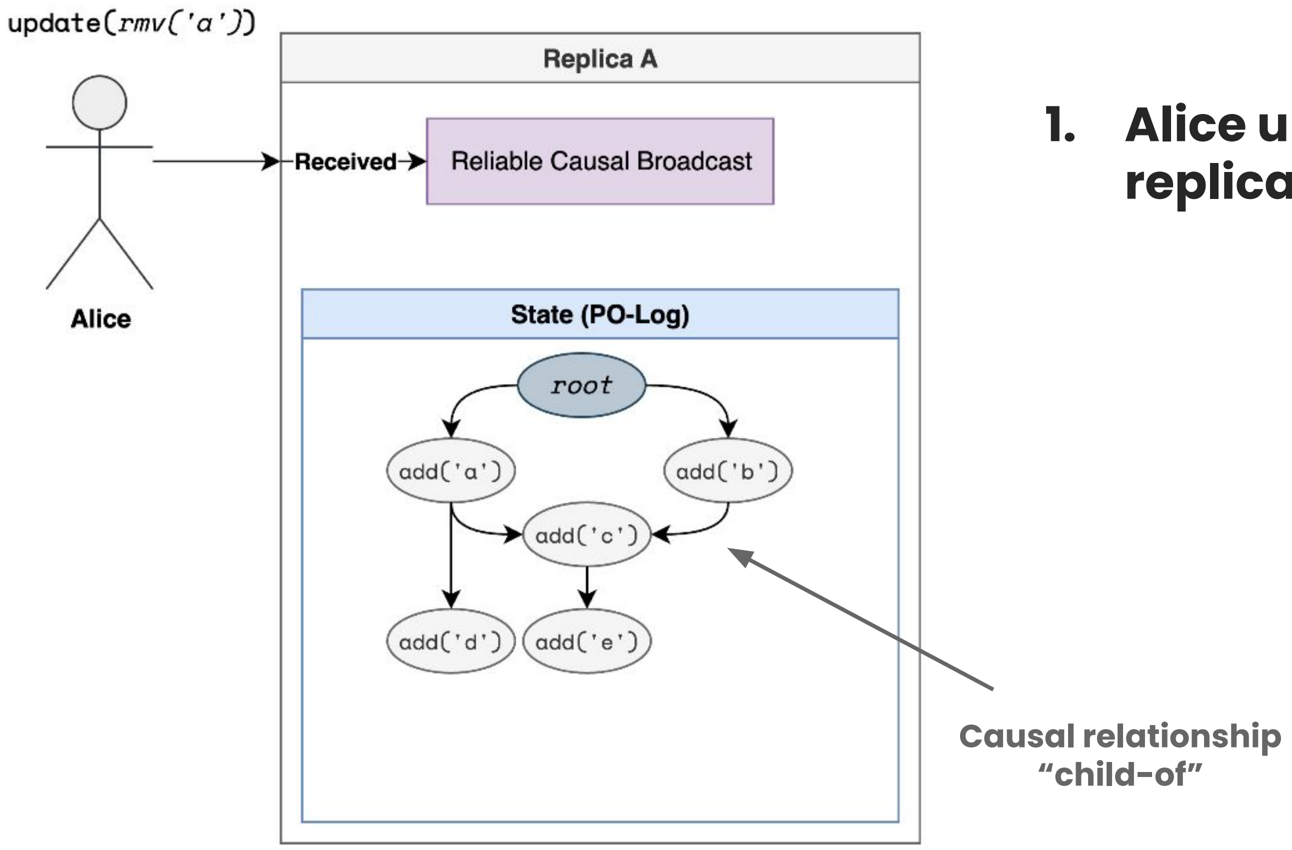
Algorithm 1: Distributed algorithm for a replica i showing the interaction between the RCB middleware and the Pure CRDT model

state: $s_i \leftarrow \emptyset$; // PO-Log

// Triggered by the local user
upon event operation(o)**:**
  if *o is a query* **then**
    eval$_i(o, s_i)$;
  else
    broadcast$_i(o)$;

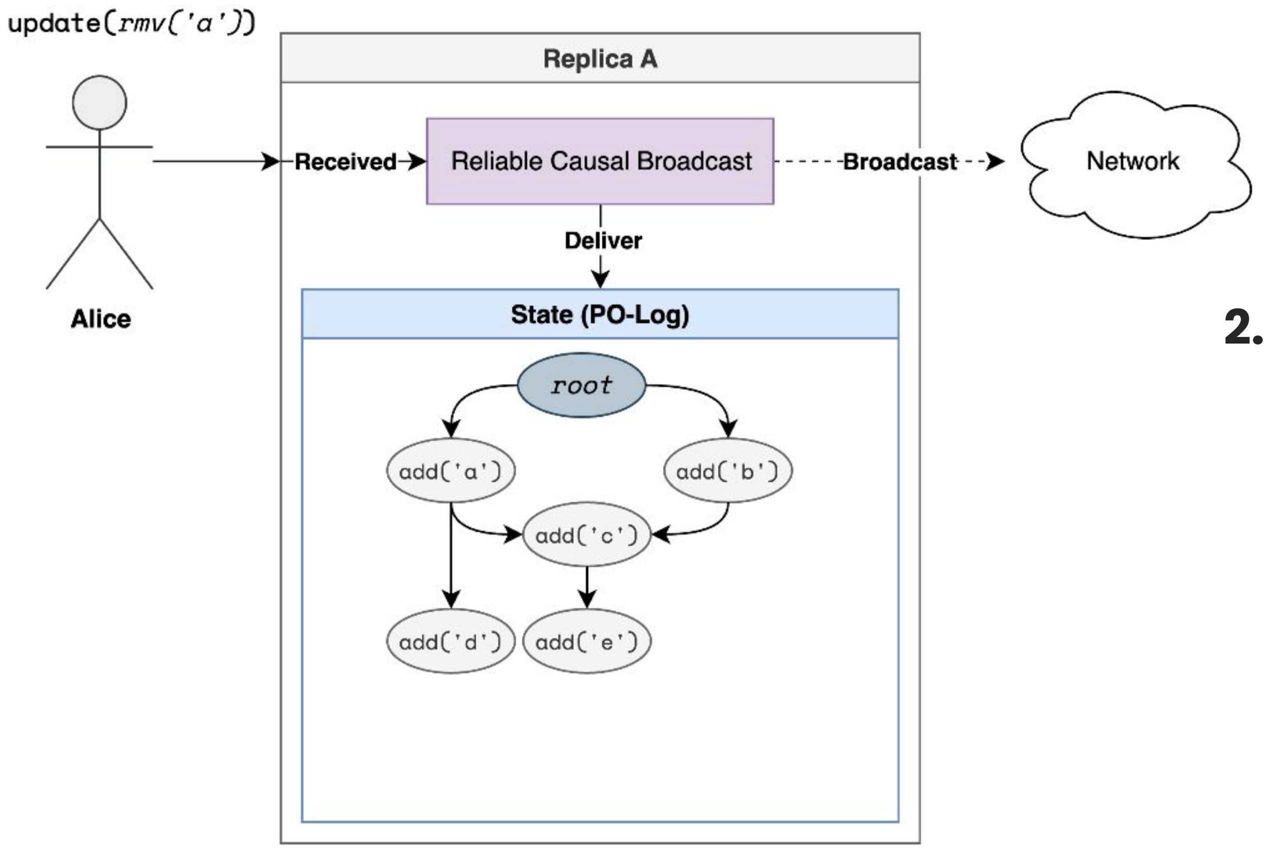
// Triggered by the RCB
upon event deliver(t, o)**:**
  // Effect of the operation
  $s_i \leftarrow \{(t,o) \mid (t,o) \not R_{self}\ s_i\} \cup s_i \setminus \{(t',o') \mid \forall (t',o') \in s_i : (t',o')\ R_{by}\ (t,o)\}$;

An Extensible Model for Domain-specific CRDTs



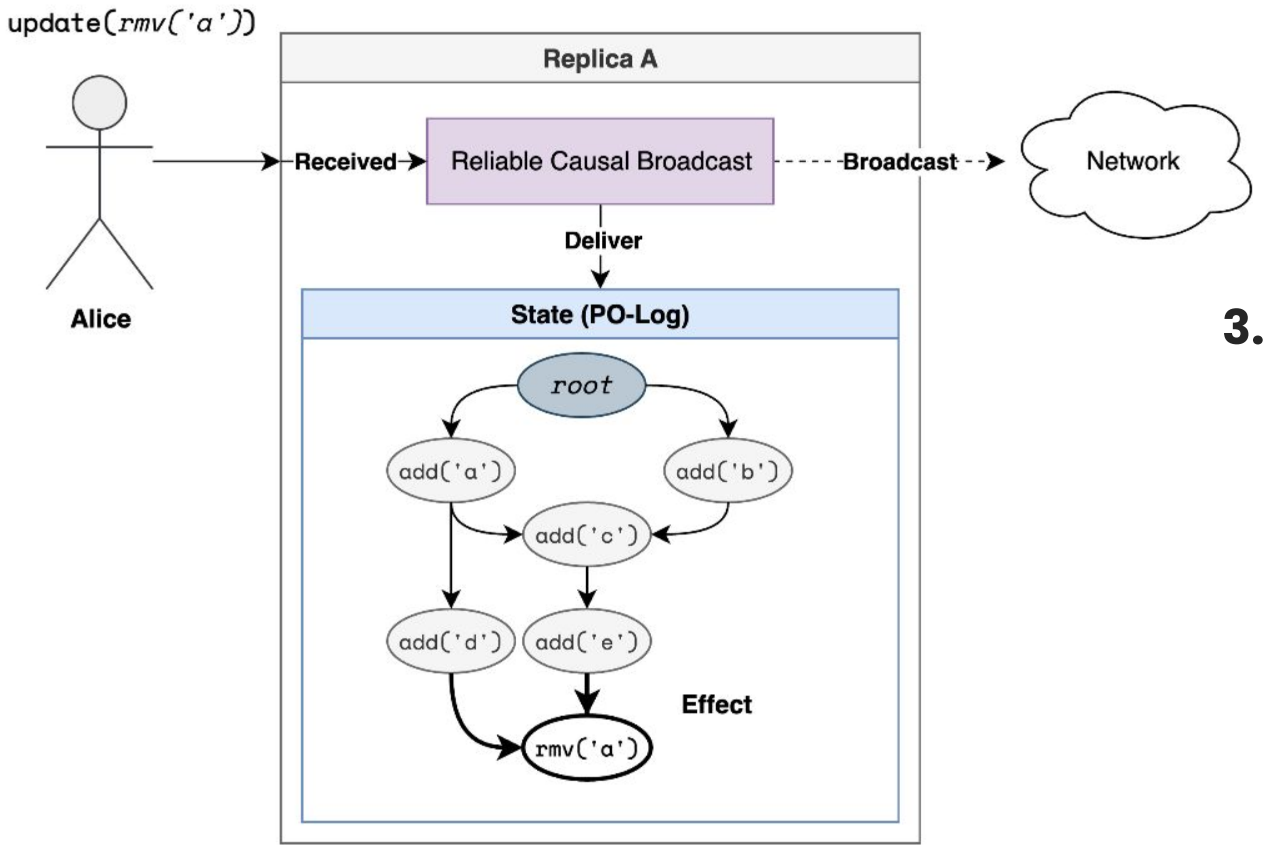
1. **Alice updates the replica**

An Extensible Model for Domain-specific CRDTs



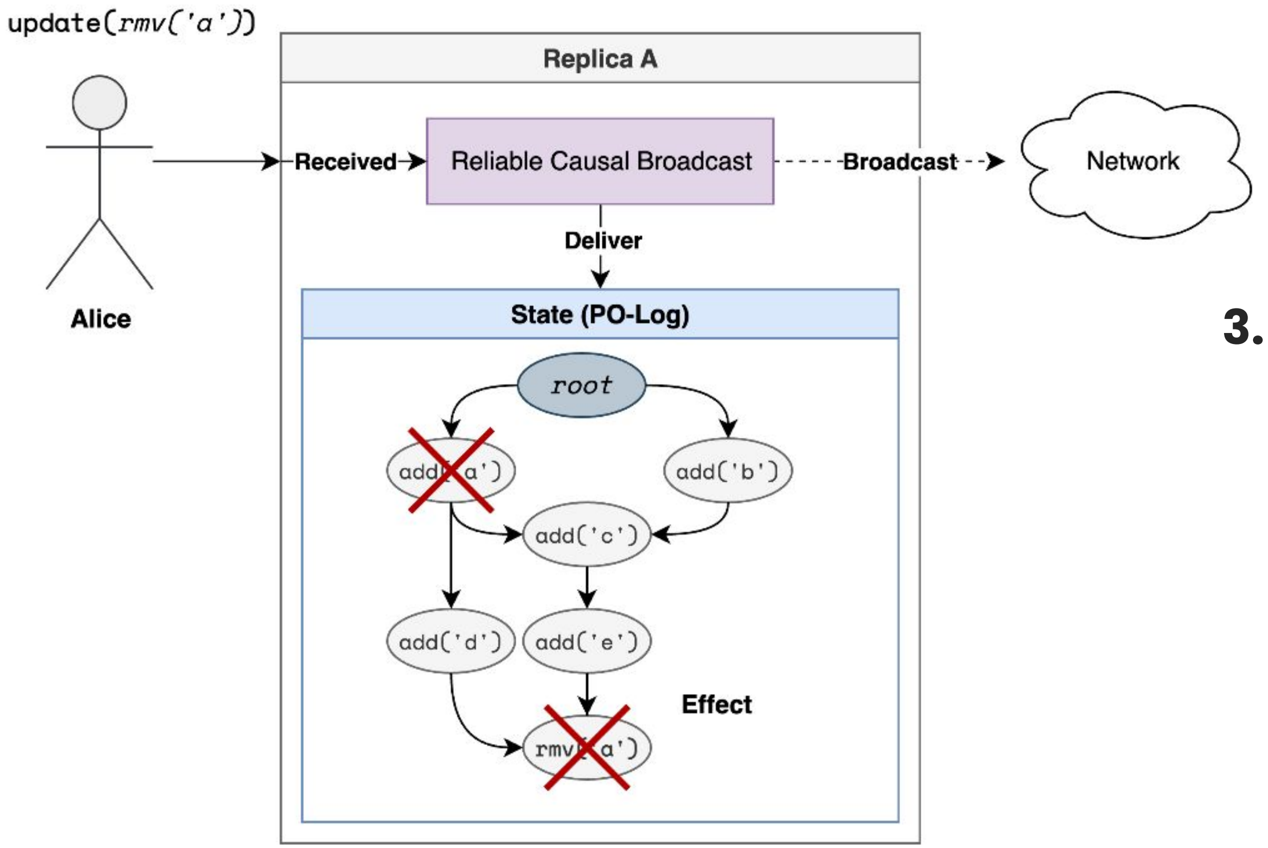
2. **Operation is delivered locally and broadcasted to other participants**

An Extensible Model for Domain-specific CRDTs



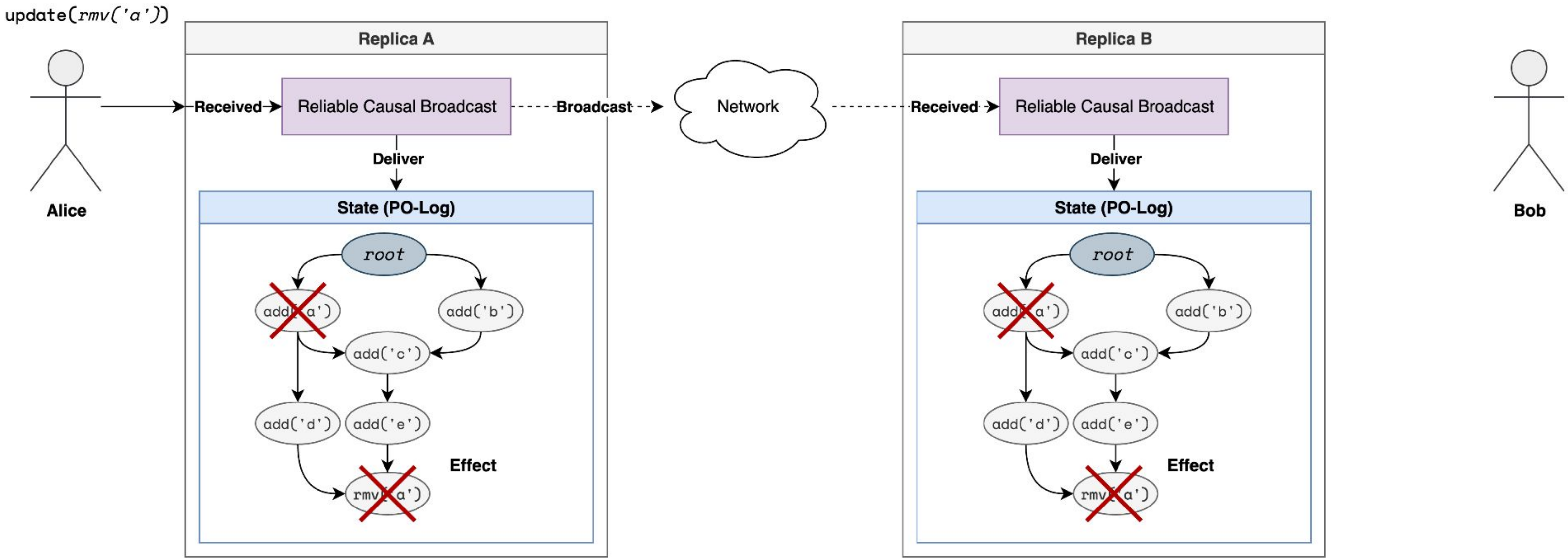
3. **<u>Effect phase:</u> the operation is appended to the log**

An Extensible Model for Domain-specific CRDTs



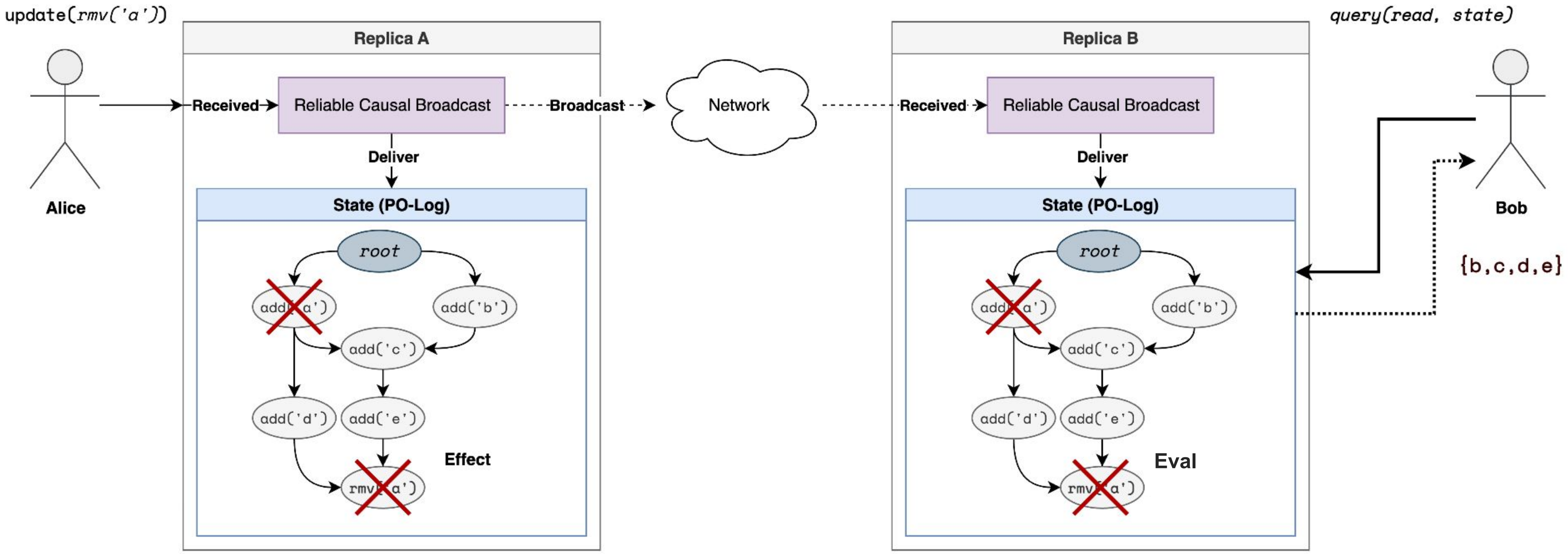
3. **<u>Effect phase:</u> redundancy relationships prune past operations from the log**

An Extensible Model for Domain-specific CRDTs



4. **Operation is received by Bob**

An Extensible Model for Domain-specific CRDTs

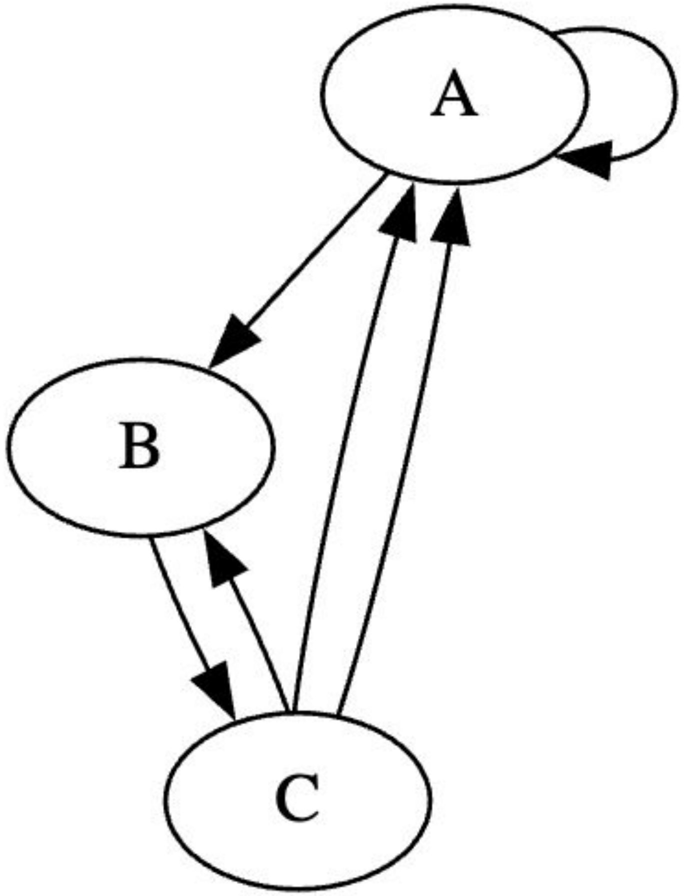


5. **Bob query the state**

2. Replicated Data Types for Collaborative Modeling

Directed Multigraph Definition

- Many models are based on **directed multigraph** structures.
- No CRDT specification with semantic conflict resolution exists for this data type.
- Graph `G = (V, A, E)`
  - V: vertices id ;
  - A: arcs id ;
  - `E ⊆ V x V x A`: directed edges.
  - Dangling edges not allowed.



Example of a directed multidigraph.

An Add-Wins Directed Multigraph CRDT

$$
\begin{aligned}
U &= \{\mathsf{addVertex}(v), \mathsf{removeVertex}(v) \mid v \in V\} \\
&\quad \cup \{\mathsf{addArc}(v, v', a), \mathsf{removeArc}(v, v', a) \mid v, v' \in V, a \in A\} \\
Q &= \{\mathsf{read}\}
\end{aligned}
$$

Update operations (U) and query operations (Q).

Concurrent operations		**Conflict resolution**
`addVertex(v)`	`removeVertex(v)`	Keep the vertex
`addArc(v,v',a)`	`removeArc(v,v',a)`	Keep the arc
`addArc(v,v',a)`	`removeVertex(v)`	Arc insertion is recorded but stays invisible until the vertex is reinserted.

Conflicts that may arise.

An Add-Wins Directed Multigraph CRDT

Specification for the redundancy relations (*left*) and evaluation function (*right*).

$$(t, o)\ \mathsf{R}_{self}\ s = (\mathsf{op}(o) = (\mathsf{removeVertex} \vee \mathsf{removeArc}))$$

$$\begin{aligned}(t', o')\ \mathsf{R}_{by}\ (t, o) = t' \prec t \\ \wedge (o' = \mathsf{addArc}(v'_1, v'_2, a') \wedge o = \mathsf{removeVertex}(v) \\ \wedge v'_1 = v \vee v'_2 = v) \\ \vee (o' = \mathsf{addVertex}(v') \wedge (o = \mathsf{addVertex}(v) \\ \vee o = \mathsf{removeVertex}(v)) \wedge v' = v) \\ \vee (o' = \mathsf{addArc}(v'_1, v'_2, a') \wedge (o = \mathsf{addArc}(v_1, v_2, a) \\ \vee o = \mathsf{removeArc}(v_1, v_2, a)) \\ \wedge v'_1 = v_1 \wedge v'_2 = v_2 \wedge a' = a)\end{aligned}$$

Algorithm 2: Evaluation of the local state s_i to reconstruct the multidigraph G

Function eval(read, s_i):
- $V \leftarrow \emptyset, A \leftarrow \emptyset, E \leftarrow \emptyset$;
- **foreach** $(t, o) \in s_i$ **do**
  - **if** $o = \mathsf{addVertex}(v)$ **then**
    - $V \leftarrow V \cup \{v\}$;
  - **else if** $o = \mathsf{addArc}(v_1, v_2, a) \wedge v_1, v_2 \in V$ **then**
    - $A \leftarrow A \cup \{a\}$;
    - $E \leftarrow E \cup \{(v_1, v_2, a)\}$;
- **return** $G = (V, A, E)$

Supporting Nested CRDTs in the Vertices and Edges

- Extend the AW Multidigraph to support nested CRDTs in its vertices and edges.
- Flat PO-Log → **hierarchy of nested logs**[1].
  - Leaf nodes: classical PO-Log ;
  - Internal nodes: map identifiers to child logs ;
  - Defines a path in the nested hierarchy.
- Introduces a new parent–child relation to express hierarchical reset: $R_{byParent}$.

$$\mathsf{Log}_0 := T \hookrightarrow O$$

$$\forall n \geq 0 : \mathsf{Log}_{n+1} := K \hookrightarrow \mathsf{Log}_n$$

$$\mathsf{Log} := \bigcup_{n \geq 0} \mathsf{Log}_n$$

Recursive definition of nested logs.

(1) Nested Pure Operation-Based CRDTs, Bauwens et al. (2023)

An Update-Wins Directed Multigraph CRDT

- G = $(V, A, E, \lambda_V, \lambda_E)$
  - $\lambda_{V'}, \lambda_E$: assign vertex/arc identifiers to its corresponding label ;
  - Label → child CRDT.

$$
\begin{aligned}
U = \{&\mathsf{updateVertex}(v, \ell_v), \mathsf{removeVertex}(v) \mid v \in V, \ell_v \in \mathsf{Log}\} \\
&\cup \{\mathsf{updateArc}(v, v', a, \ell_e), \mathsf{removeArc}(v, v', a) \mid \\
&\quad v, v' \in V, a \in A, \ell_e \in \mathsf{Log}\}
\end{aligned}
$$

Update operations (U) and query operations (Q).

$$
\begin{aligned}
(k, child)\ \mathsf{R}_{byParent}\ (t, o) = \\
&(o = \mathsf{removeVertex}(v) \wedge (k = v \\
&\quad \vee k = (v_1, v_2, a) \wedge (v = v_1 \vee v = v_2))) \\
&\vee (o = \mathsf{removeArc}(v_1, v_2, a) \wedge k = (v_1', v_2', a') \\
&\quad \wedge (v_1 = v_1' \wedge v_2 = v_2' \wedge a = a'))
\end{aligned}
$$

Nested multidigraph CRDT specification for relation $R_{byParent}$

$$
\begin{aligned}
M_V &: V \hookrightarrow \mathsf{Log}, \\
M_E &: (V \times V \times A) \hookrightarrow \mathsf{Log}, \\
\mathsf{Multidigraph} &:= (M_V, M_E)
\end{aligned}
$$

Internal state for the nested multidigraph.

An Update-Wins Directed Multigraph CRDT

Algorithm 3: Update-Wins Multidigraph CRDT hierarchical reset policy

```
// During the effect phase
if (k, child) R_byParent (t, o) then
    // The reset does not apply to child operations
       concurrent with t
    conc ← false;
    reset(child, t, conc);

Function reset(log, t, conc):
    if log ∈ (T ↪ O) then
        log ← log \ {(t′, o′) | ∀(t′, o′) ∈ log : (t′ ≺ t) ∨ conc};
    else if log ∈ (K ↪ Log) then
        // The reset function is called recursively in
           all children
        foreach child ∈ log do
            reset(child, t, conc);
```

Algorithm 4: Recursive evaluation of the nested multidigraph G

```
Function eval(read, (M_V, M_E) ∈ Multidigraph_i):
    V ← ∅, A ← ∅, E ← ∅;
    λ_V ← ∅, λ_E ← ∅;
    foreach (v, child) ∈ M_V do
        V ← V ∪ {v};
        λ_V ← λ_V ∪ {(v, eval(read, child))}
    foreach (v1, v2, a, child) ∈ M_E do
        if v1, v2 ∈ V then
            A ← A ∪ {a};
            E ← E ∪ {(v1, v2, a)};
            λ_E ← λ_E ∪ {(v1, v2, a, eval(read, child))}
    return G = (V, A, E, λ_V, λ_E)
```

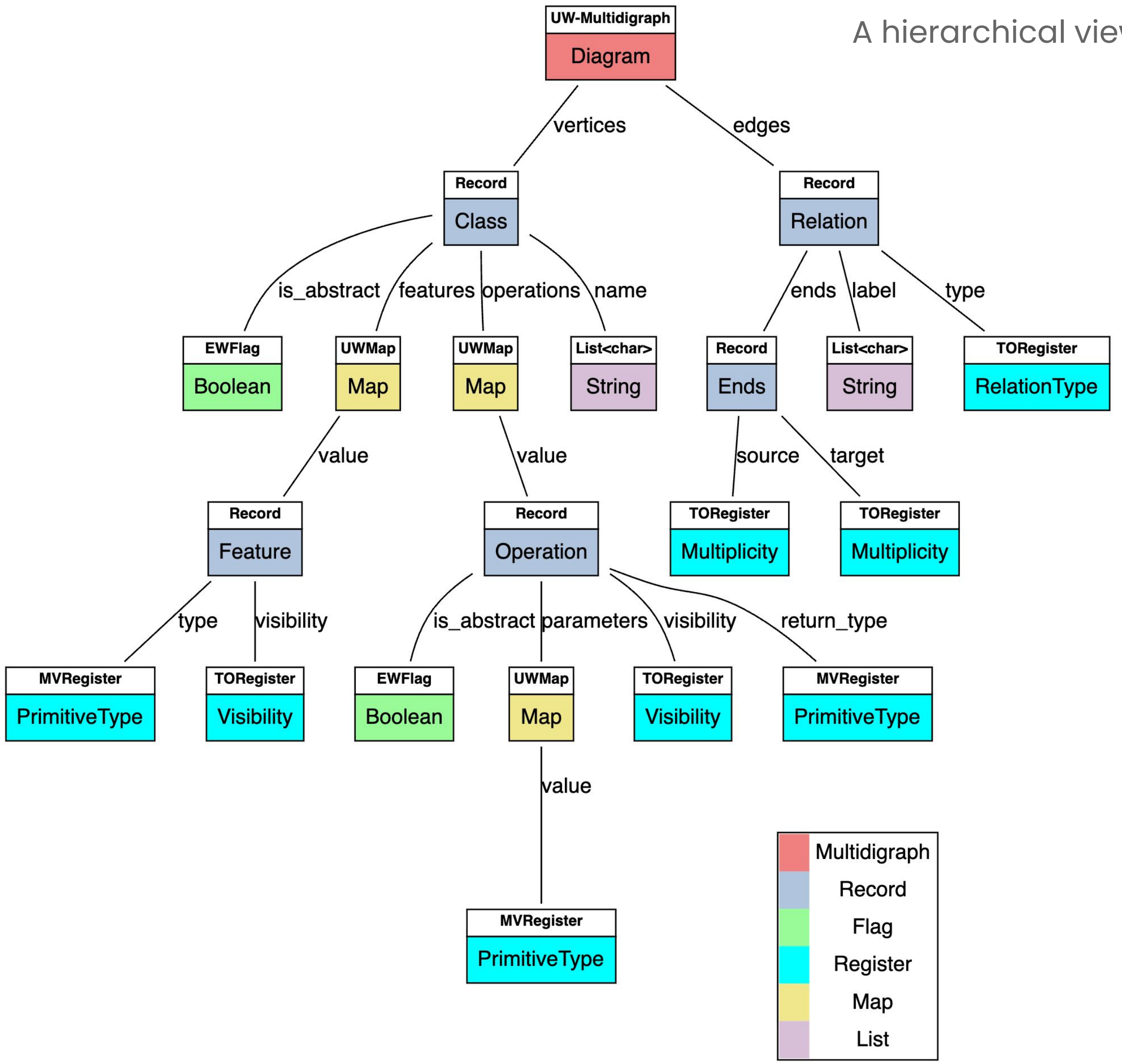
A UML Class Diagram CRDT

- Local-First **UML Class Diagram** with CRDTs.
- Compose multiple CRDTs within the `UW Multidigraph`.
  - User-friendly conflict resolution policy.
  - Vertices hold `Class` CRDTs.
  - Edges hold `Relations` CRDTs.
  - Both represented as record-like CRDTs.
- Support essential features of a UML Class Diagram.
  - (Abstract) classes, operations, features, relations, multiplicities, etc.

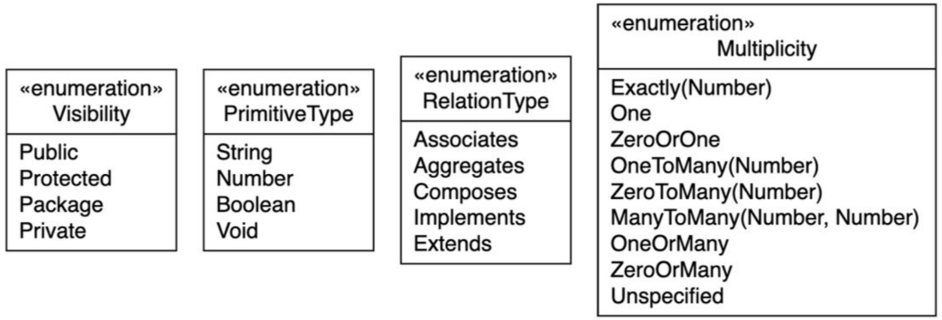
Name	Evaluated value type	Policy
EWFlag	Boolean	"Enable-Wins"
Class, Relation, Feature, Operation, Multiplicity	Record	None (no conflict)
MVRegister⟨T⟩	Set⟨T⟩	Keep all concurrent values
UWMap⟨K, Log⟩	Map⟨K, LogValue⟩	"Update-Wins"
UWMultidigraph⟨V, A⟩	Graph⟨V, A⟩	"Update-Wins"
TORegister⟨T⟩	T	User-defined total order

Summary of the CRDTs used in the Class Diagram CRDT.

A Class Diagram CRDT



A hierarchical view of the UML Class Diagram CRDT.

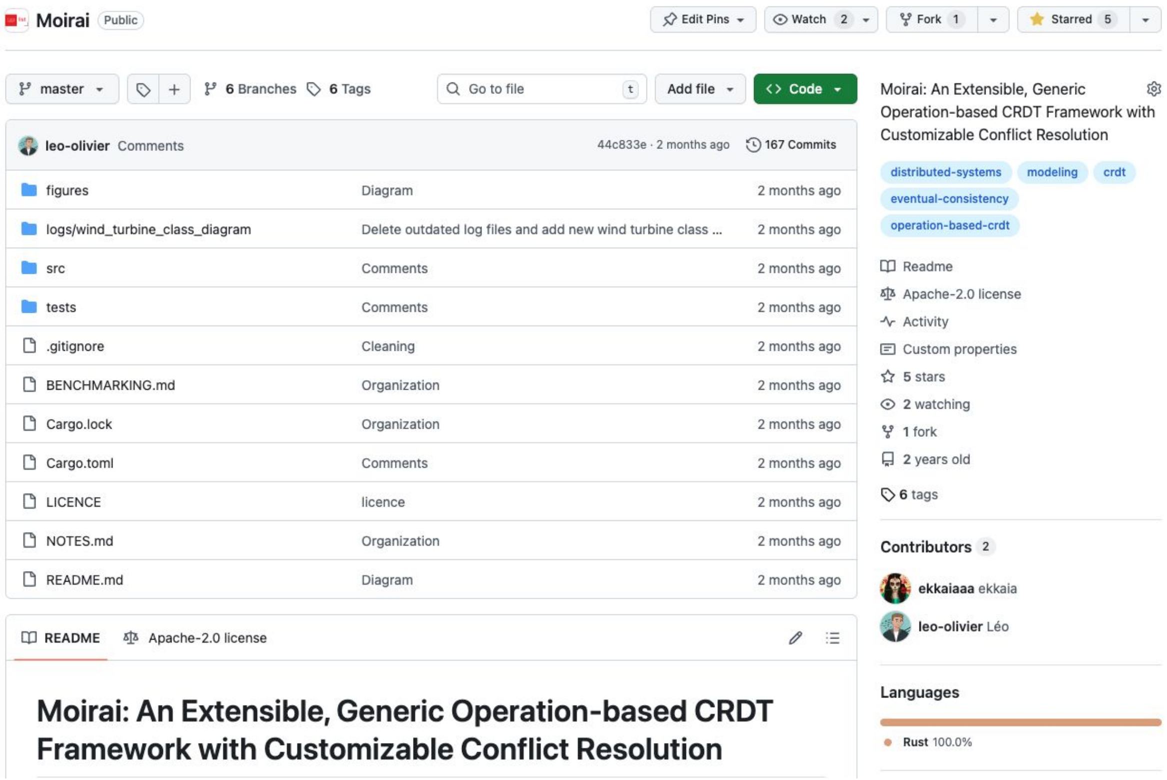


Enumerations used in the Class Diagram CRDT.

4. Evaluation

Moirai Framework

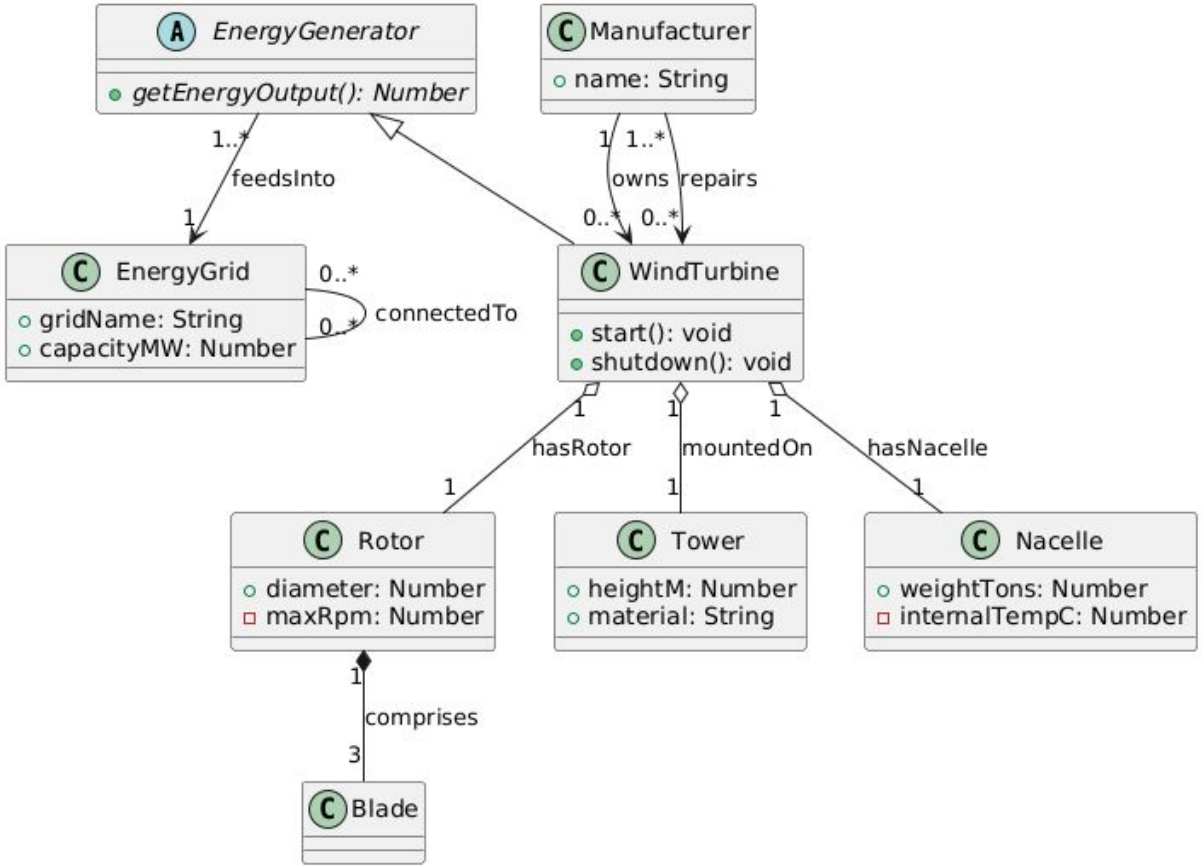
- Pure operation-based CRDT implementation in Rust.
- Designed for extensibility and composability.
- Why Rust:
  - Deterministic performance ;
  - Memory safety ;
  - Cross-platform support (e.g., WebAssembly).



Moirai repository: https://github.com/CEA-LIST/Moirai

Expressiveness

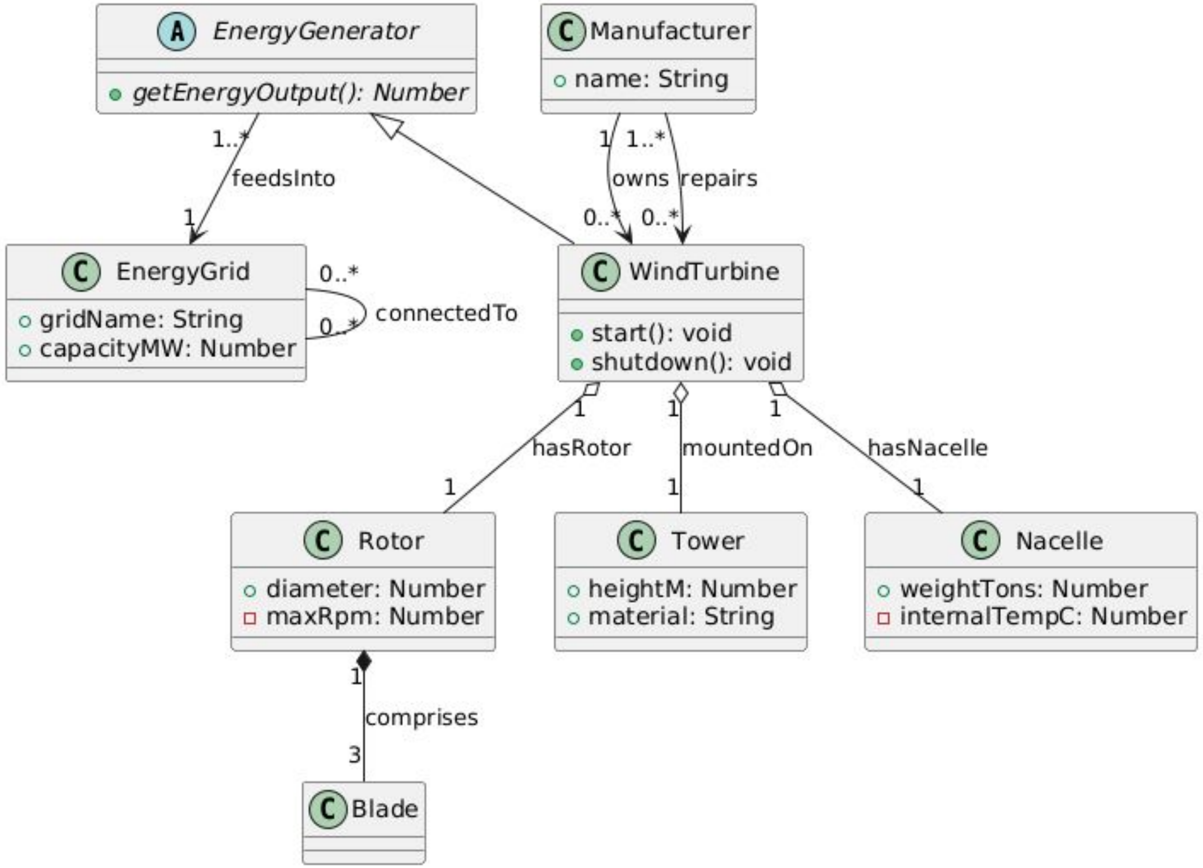
- **Does the approach allow for defining rich and complex models?**
- Design of a *wind turbine model* within the Moirai framework using the Class Diagram CRDT.
- Small number of elements, but showcases a rich set of class diagram features.



A UML class diagram of a wind turbine model, generated with PlantUML.

Asynchronous collaboration (1/3)

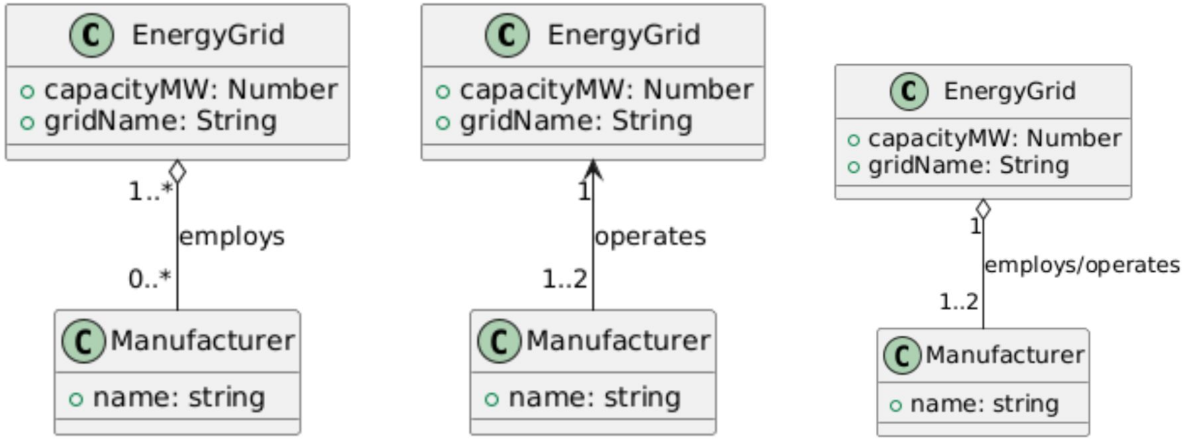
- **Is the approach suitable for asynchronous collaboration?**
  - High risk of conflicts!
- Class diagram CRDT conflict resolution policy:
  - No rollbacks.
  - Every operation has a visible effect.
  - Retains all conflicting values to prevent data loss when necessary.



A UML class diagram of a wind turbine model, generated with PlantUML.

Asynchronous collaboration (2/3)

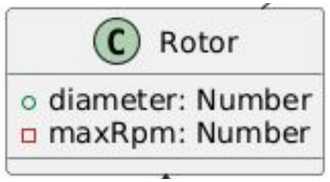
Alice's operations	Bob's operations
Graph.updateArc("manufacturer", "energy_grid", "rel", . . .)	
Relation.type(TORegister.write("Aggregates"))	Relation.type(TORegister.write("Associates"))
Relation.label(MVRegister.write("employs"))	Relation.label(MVRegister.write("operates"))
On the relation ends	
Ends.source(TORegister.write(ZeroOrMany)))	Ends.source(TORegister.write(OneToMany(2))))
Ends.target(TORegister.write(OneOrMany)))	Ends.target(TORegister.write(One)))



(a) Alice's version. (b) Bob's version. (c) Merge result.

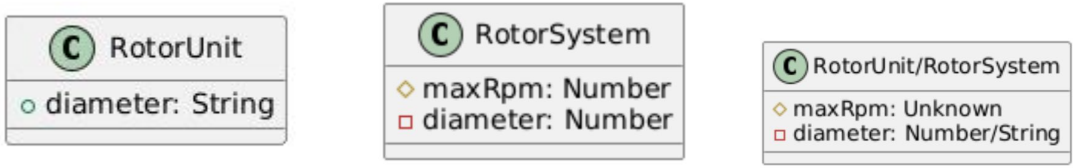
Concurrent update of relation label, multiplicity, and type.

Asynchronous collaboration (3/3)



Before Alice and Bob modifications.

Alice's operations	Bob's operations
On maxRpm feature	
Feature.visibility(TORegister.write("Public")) Feature.type(MVRegister.write("String"))	Feature.visibility(TORegister.write("Protected")) Feature.type(MVRegister.write("Number"))
On Rotor class	
Class.name(MVRegister.write("RotorUnit"))	Class.name(MVRegister.write("RotorSystem"))
On diameter feature	
Map.remove("maxRpm")	Map.update("diameter", Feature.visibility(TORegister.write("Private")))



(a) Alice's version. (b) Bob's version. (c) Merge result.

Concurrent update of feature visibility, type, and class label.

Robustness

- Formal Verification with **VeriFx**[1]:
  - SMT theorem prover checks for counterexamples to required properties ;
  - Ensures the specification is strongly eventually consistent.
- Fuzzer tool:
  - Generates executions with varying levels of concurrency ;
  - Checks that all replicas converge.
  - Measure throughput of operations.

```
object GraphOps {
enum GraphOps[V, E] {
    AddVertex(id: V) | RemoveVertex(id: V) |
    AddArc(from: V, to: V, id: E) | RemoveArc(from: V, to: V, id: E)
  }
}

class PureAWMultidigraph[V, E](polog: Set[TaggedOp[GraphOps[V, E]]])
  extends PureOpBasedCRDT[GraphOps[V, E], PureMultidigraph[V, E]] {

  def copy(newPolog: Set[TaggedOp[GraphOps[V, E]]]) =⋯
  // Remove operations are self-redundant
  def selfRedundant(op: TaggedOp[GraphOps[V, E]]): Boolean = op.o match {⋯
  }

  // Check if `x` is redundant given `y`
  def redundantBy(x: TaggedOp[GraphOps[V, E]], y: TaggedOp[GraphOps[V, E]]): Boolean = {⋯
  }

  // Check if the PO-Log contains a vertex or edge
  def contains(v: V, e: E): Boolean =⋯
}

object PureMultidigraph extends PureCRDTProof2[GraphOps, PureMultidigraph]
```

VeriFx proof code.

Operations	Model size	Replicas	Ops/sec (approx.)
100,000	150 elems	4	50,000
		8	30,000
		16	15,000

Throughput of operations on the Class Diagram CRDT measured on different numbers of replicas.

(1) *VeriFx: Correct Replicated Data Types for the Masses*, De Porre et al. (2023)

4. Conclusion

Limitations

- Beyond data replication, collaborative modeling needs:
    - Undo/redo ;
    - History browsing ;
    - Access control.
- Applicability to other metamodels remains to be explored.
    - Current limitation: no user interface, only Graphviz export available.

Future work

- Automatic generation of collaborative Local-First DSL
- Transactions
- Integration of Moirai as a replication layer for an existing modeling tool
  - *e.g.*, Syson (SysML v2).

Thanks for your attention!

Léo OLIVIER

leo.olivier@cea.fr

Moirai: https://github.com/CEA-LIST/Moirai

Mastering Complexity

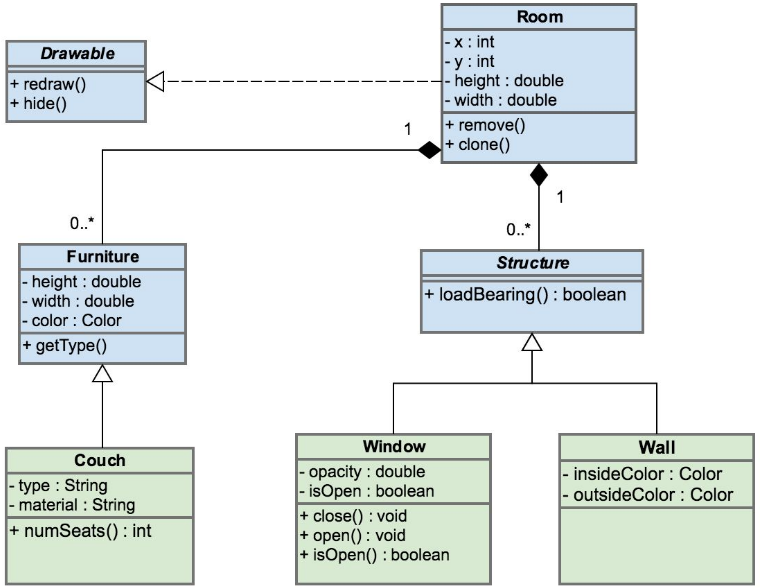
- **Contemporary challenges** require complex technological solutions.
  - Adaptation to global warming ;
  - Green industry ;
  - Sustainable energy production.
- Mastering complexity by harnessing **collective intelligence**[(1)].
  - Large, international, and multidisciplinary teams.

Illustration coming soon!

(1) *Collective Intelligence and Group Performance*, Woolley et al. (2015)

Sharing Domain Knowledge with Models

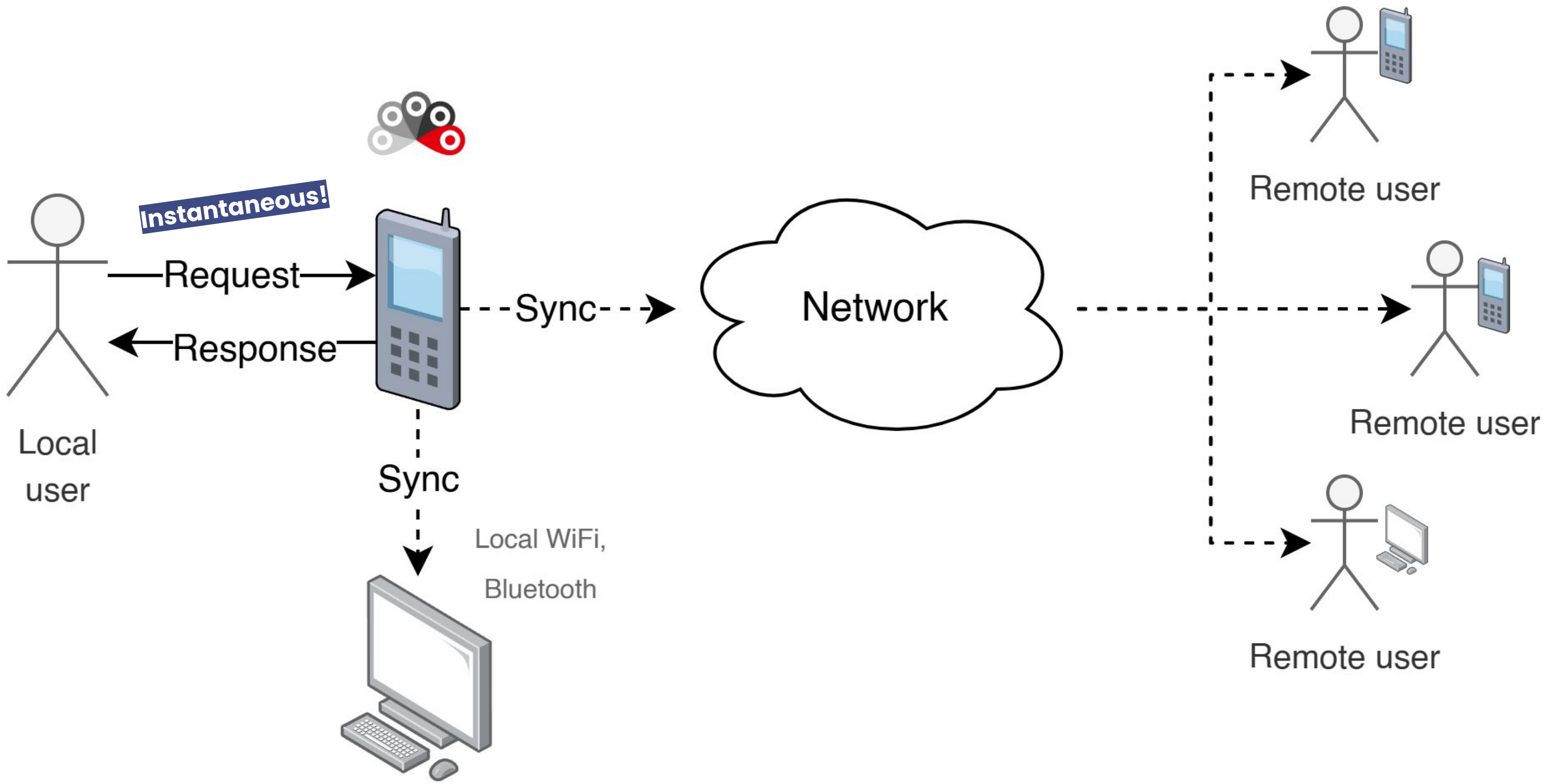
- Model-Based System Engineering (**MBSE**).
- Useful for **designing** complex systems and **coordinating** among team members in engineering projects.
  - *Example*: software engineers often use UML class diagrams to represent software architecture.
- Recent successful applications of MBSE:
  - Simulation of Smart Grids[(1)] ;
  - Sustainable factories[(2)].
- What about **collaborative modeling** solutions?



Example of a class diagram associated with a UML model.

(1) Model-based Systems Engineering for Sustainable Factory Design, Bataleblu et al. (2024)
(2) The Smart Grid Simulation Framework: Model-driven engineering applied to Cyber-Physical Systems, Oudart et al. (2020)

Local-First Workflow



Interaction between the user and the Local-First software showing the different data synchronization options.