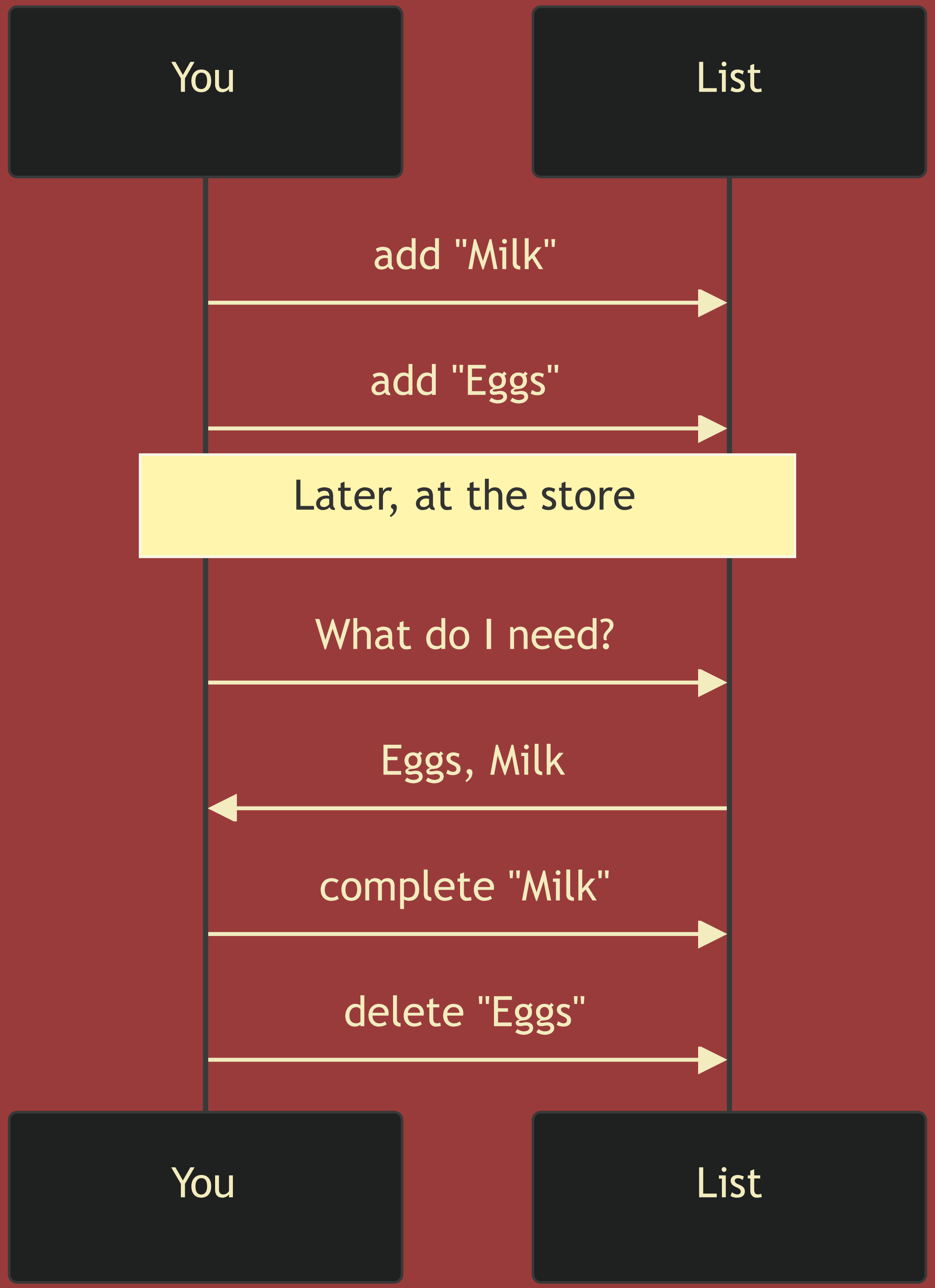
CRDTs in Rust

👋 Hello!

The Situation:
to-do list



You

List

add "Milk"

add "Eggs"

Later, at the store

What do I need?

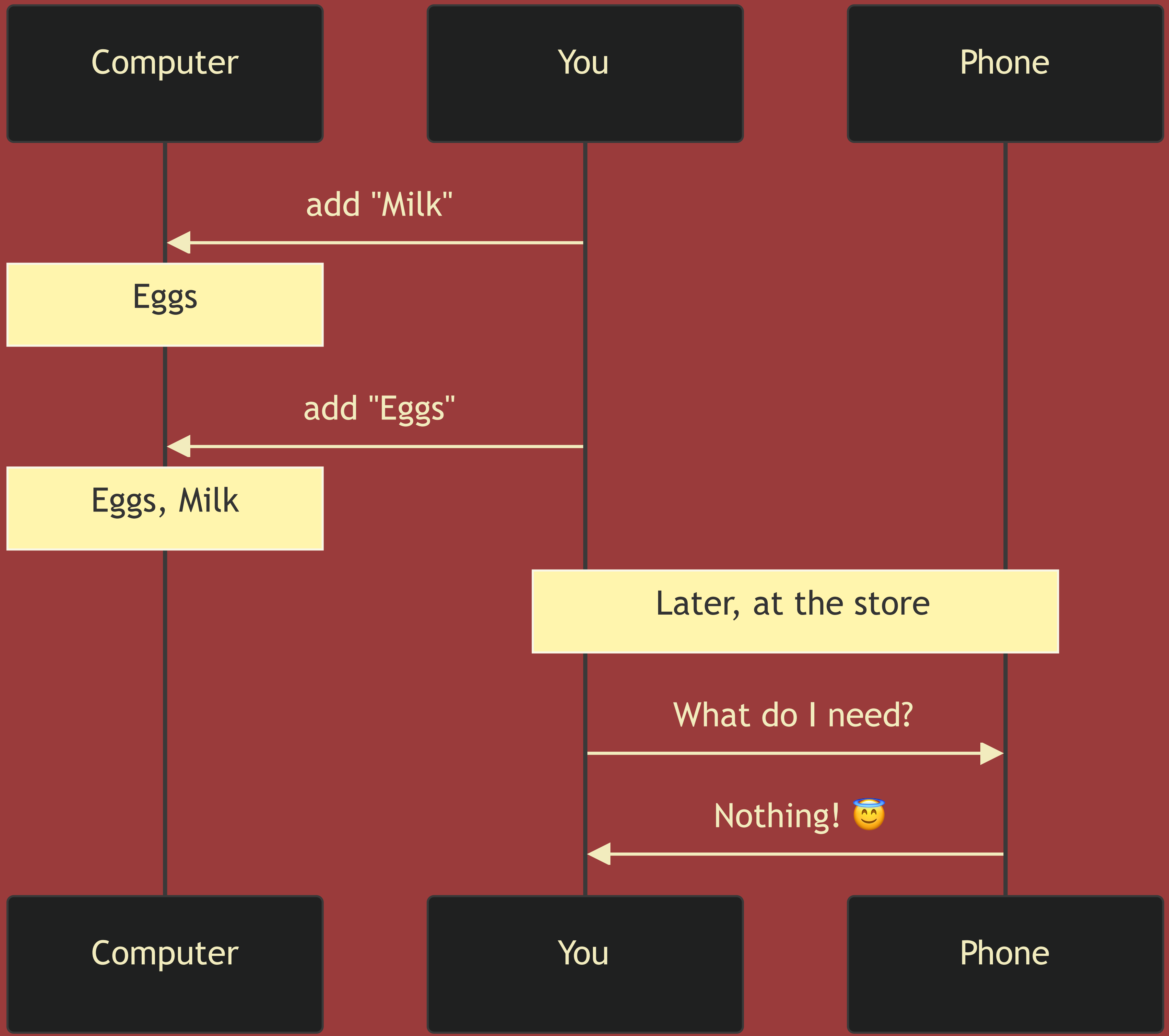
Eggs, Milk

complete "Milk"

delete "Eggs"

You

List



Computer

You

Phone

add "Milk"

Eggs

add "Eggs"

Eggs, Milk

Later, at the store

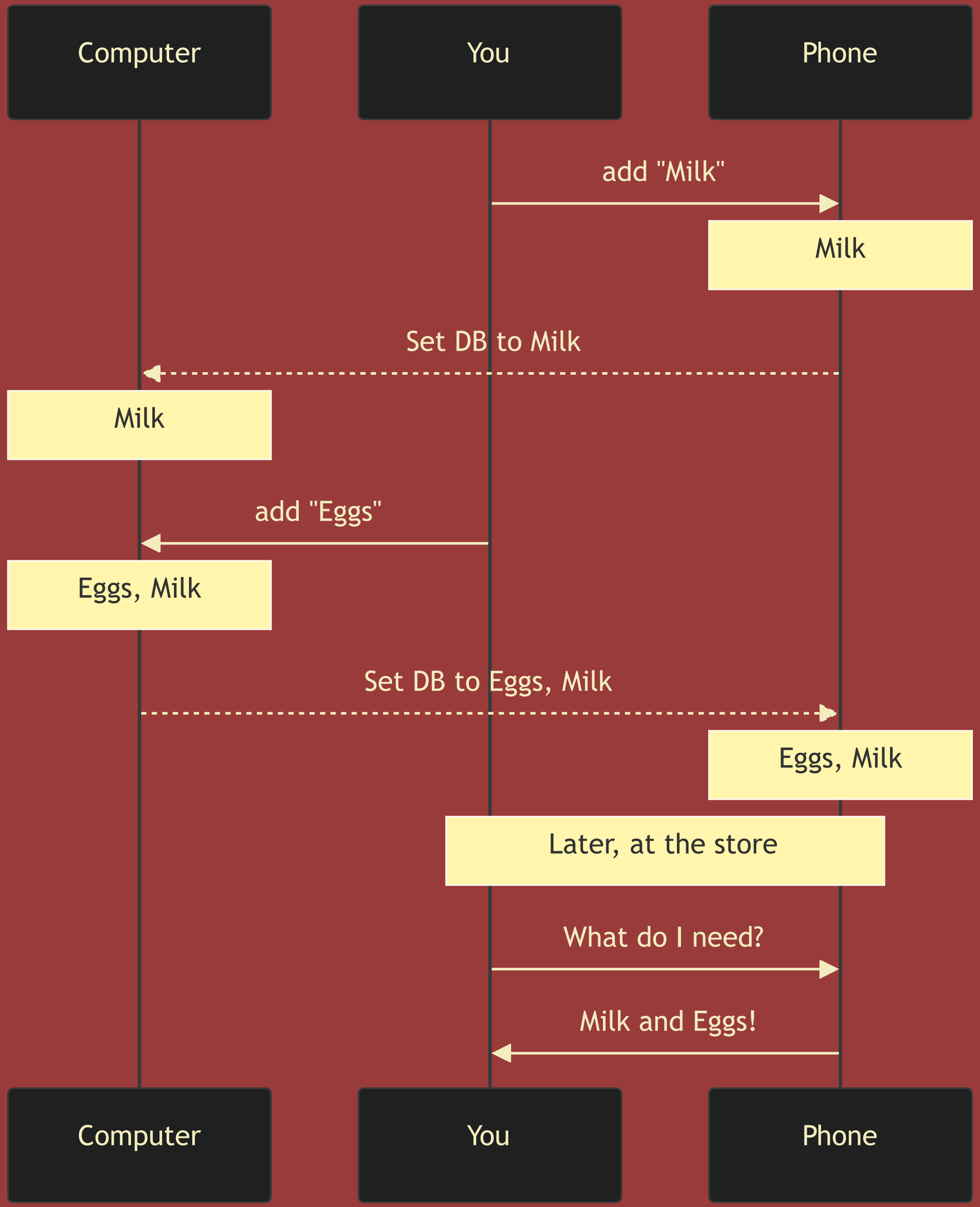
What do I need?

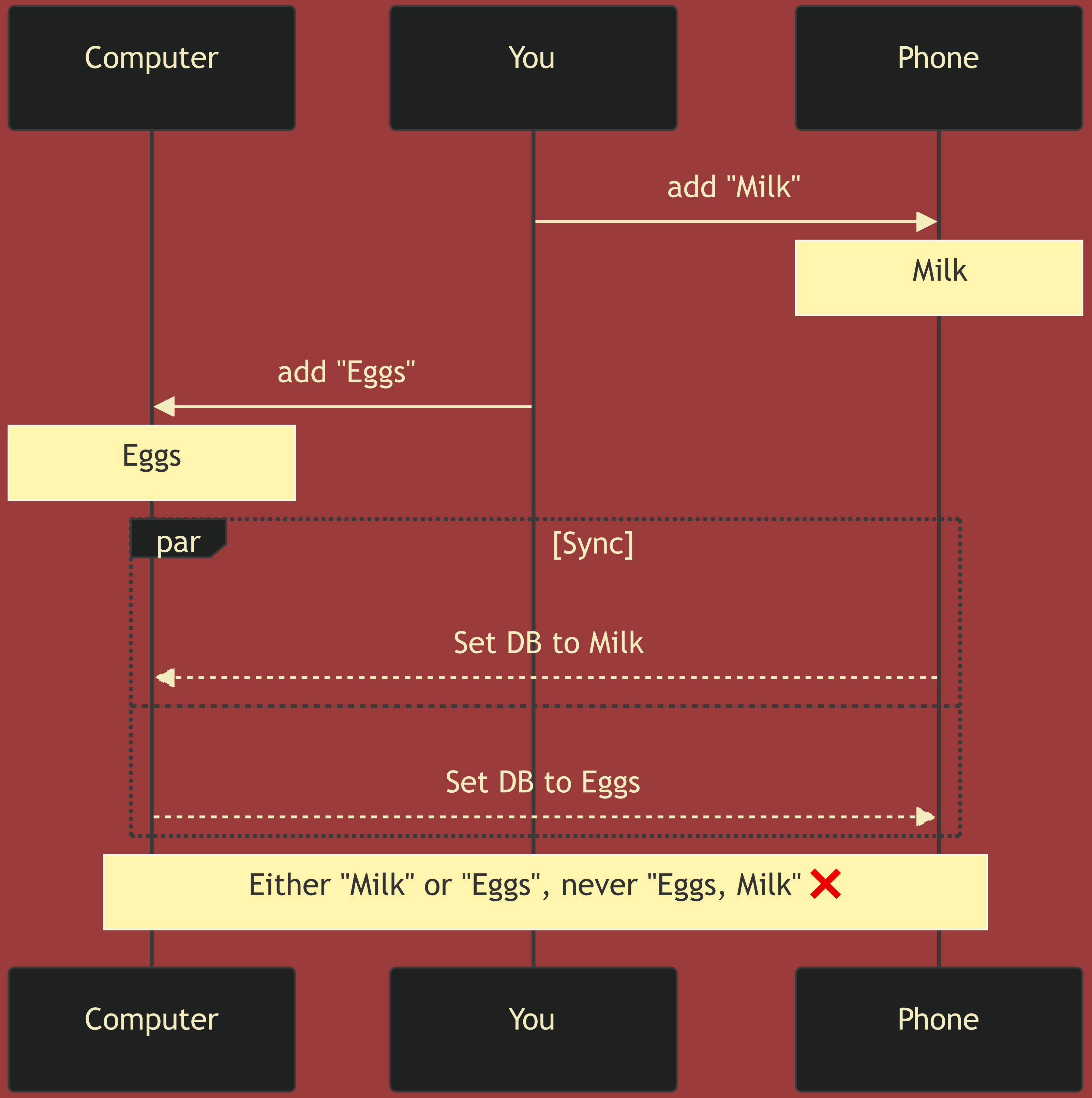
Nothing! 😇

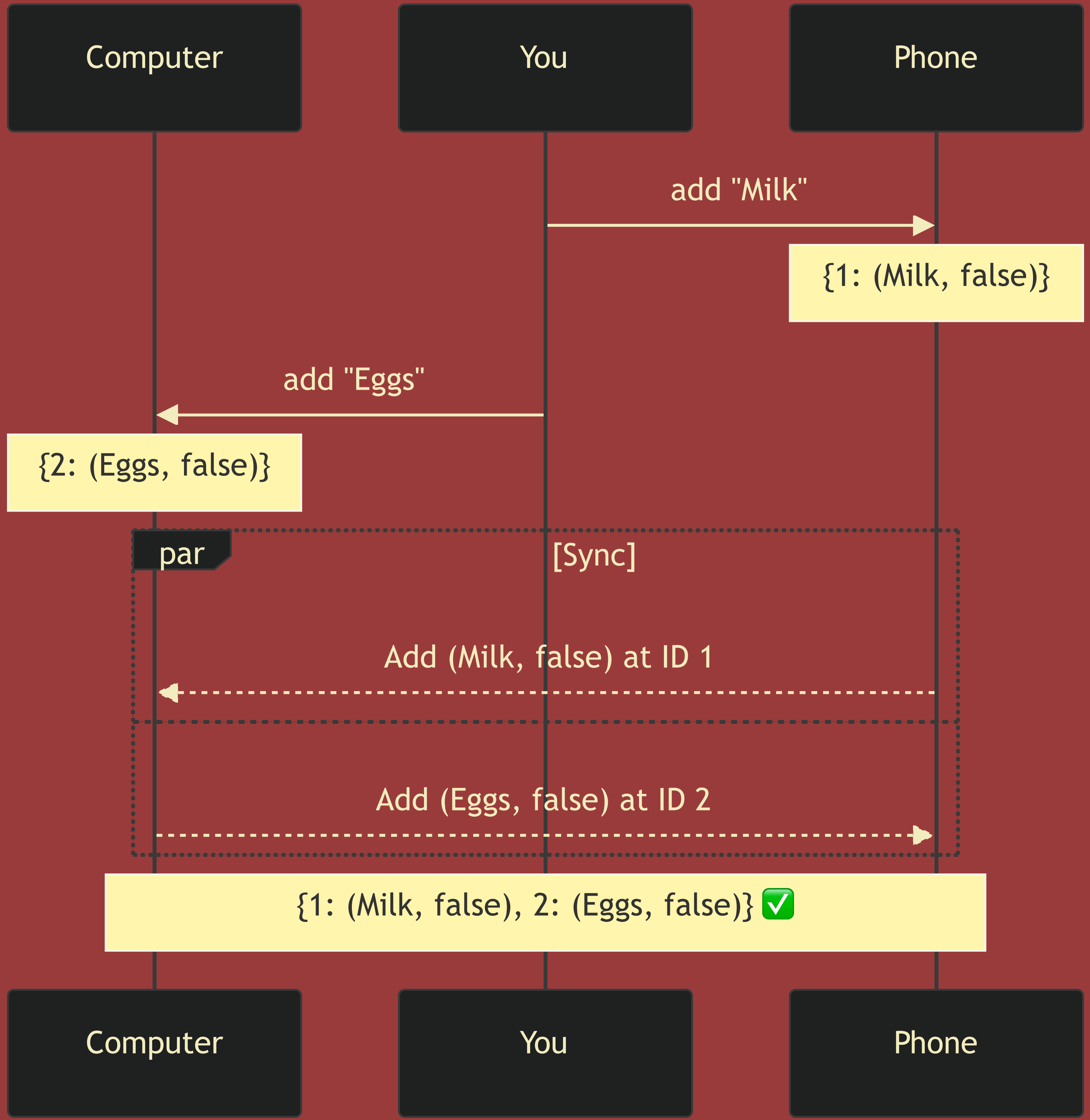
Computer

You

Phone


Computer
You
Phone
add "Milk"
Milk
Set DB to Milk
Milk
add "Eggs"
Eggs, Milk
Set DB to Eggs, Milk
Eggs, Milk
Later, at the store
What do I need?
Milk and Eggs!
Computer
You
Phone


Computer
You
Phone
add "Milk"
Milk
add "Eggs"
Eggs
par
[Sync]
Set DB to Milk
Set DB to Eggs
Either "Milk" or "Eggs", never "Eggs, Milk" ❌
Computer
You
Phone



Computer

You

Phone

add "Milk"

{1: (Milk, false)}

add "Eggs"

{2: (Eggs, false)}

par

[Sync]

Add (Milk, false) at ID 1

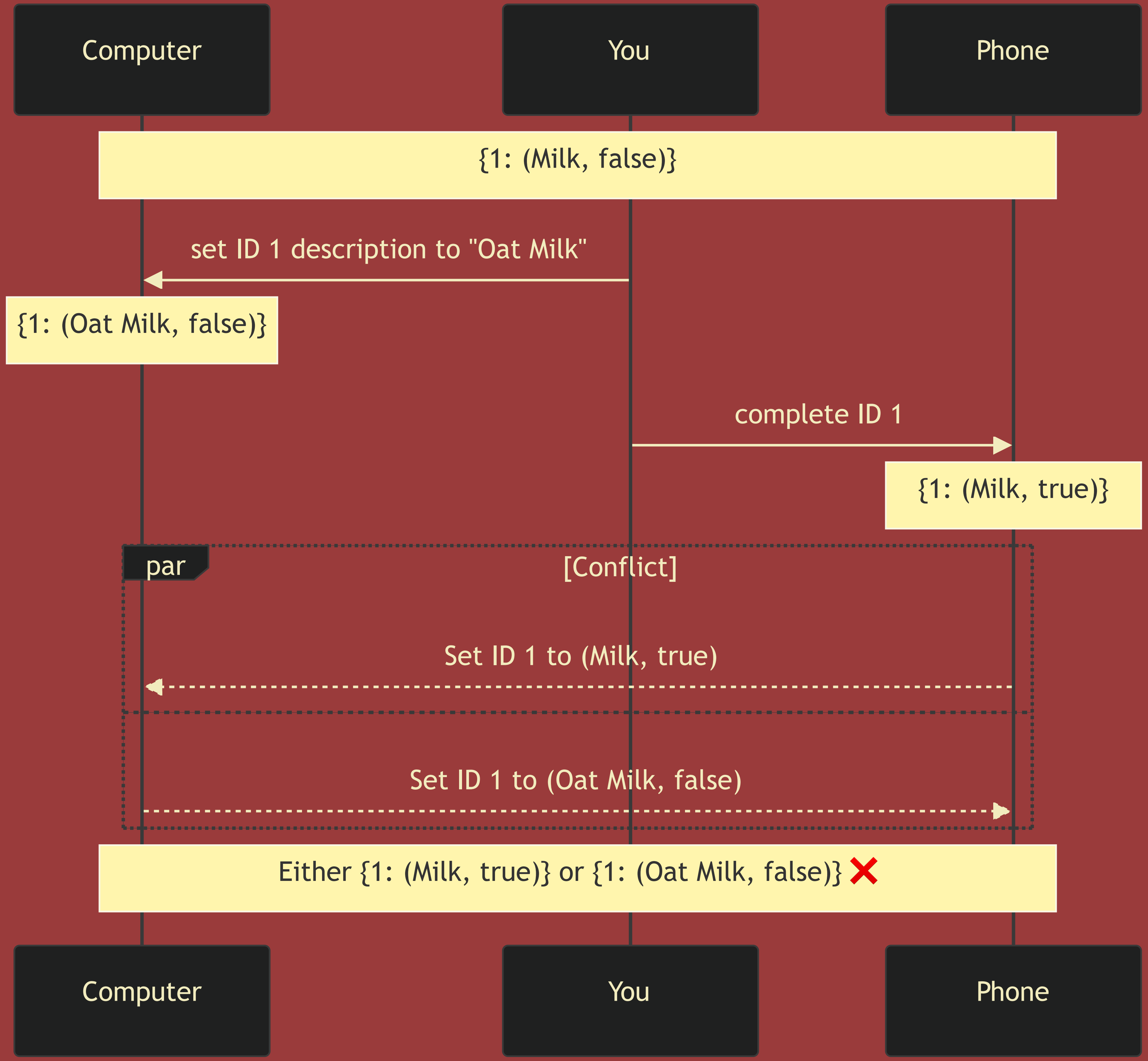
Add (Eggs, false) at ID 2

{1: (Milk, false), 2: (Eggs, false)} ✅

Computer

You

Phone



Computer

You

Phone

{1: (Milk, false)}

set ID 1 description to "Oat Milk"

{1: (Oat Milk, false)}

complete ID 1

{1: (Milk, true)}

par

[Conflict]

Set ID 1 to (Milk, true)

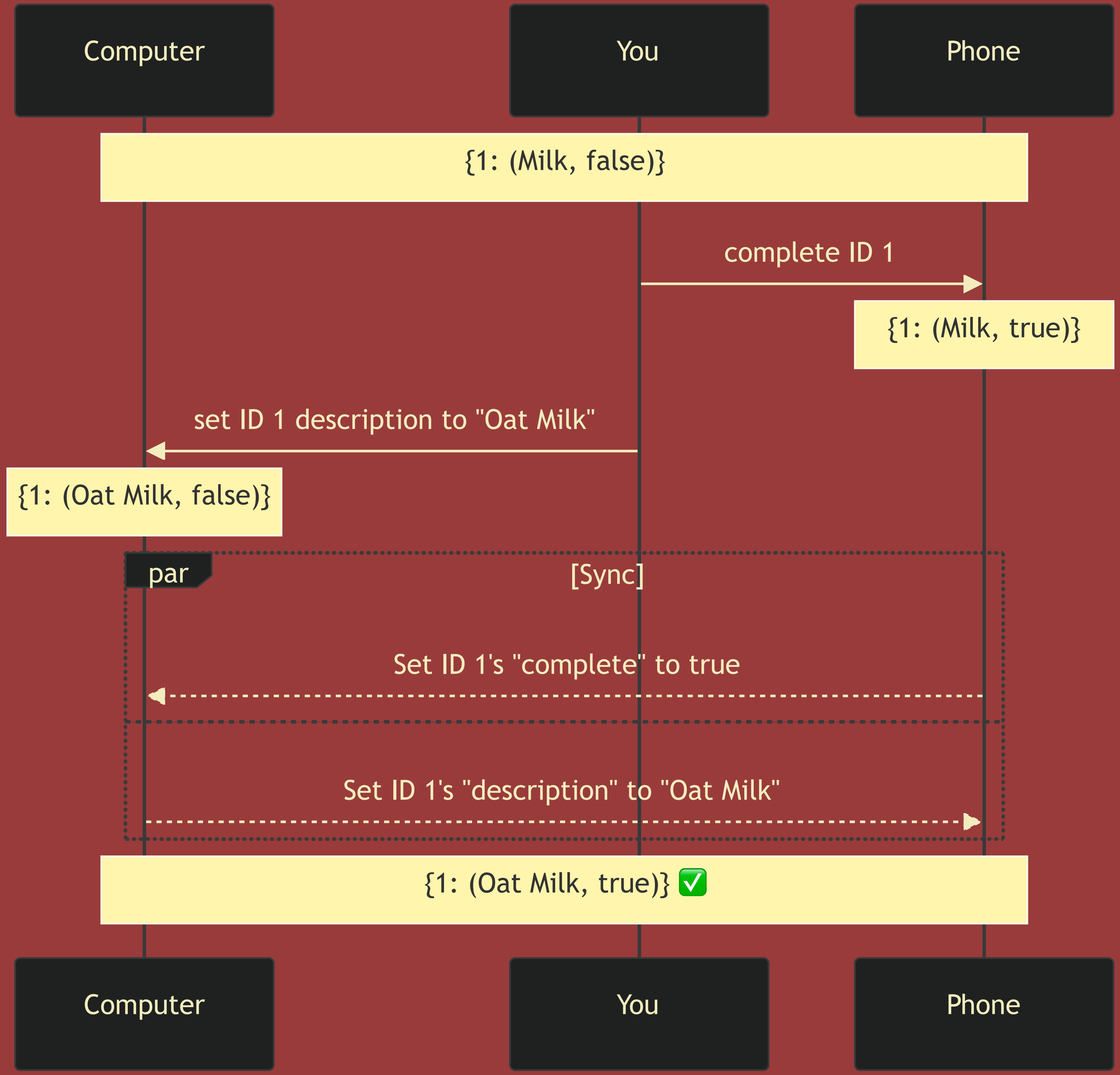
Set ID 1 to (Oat Milk, false)

Either {1: (Milk, true)} or {1: (Oat Milk, false)} ❌

Computer

You

Phone



Computer

You

Phone

{1: (Milk, false)}

complete ID 1

{1: (Milk, true)}

set ID 1 description to "Oat Milk"

{1: (Oat Milk, false)}

par

[Sync]

Set ID 1's "complete" to true

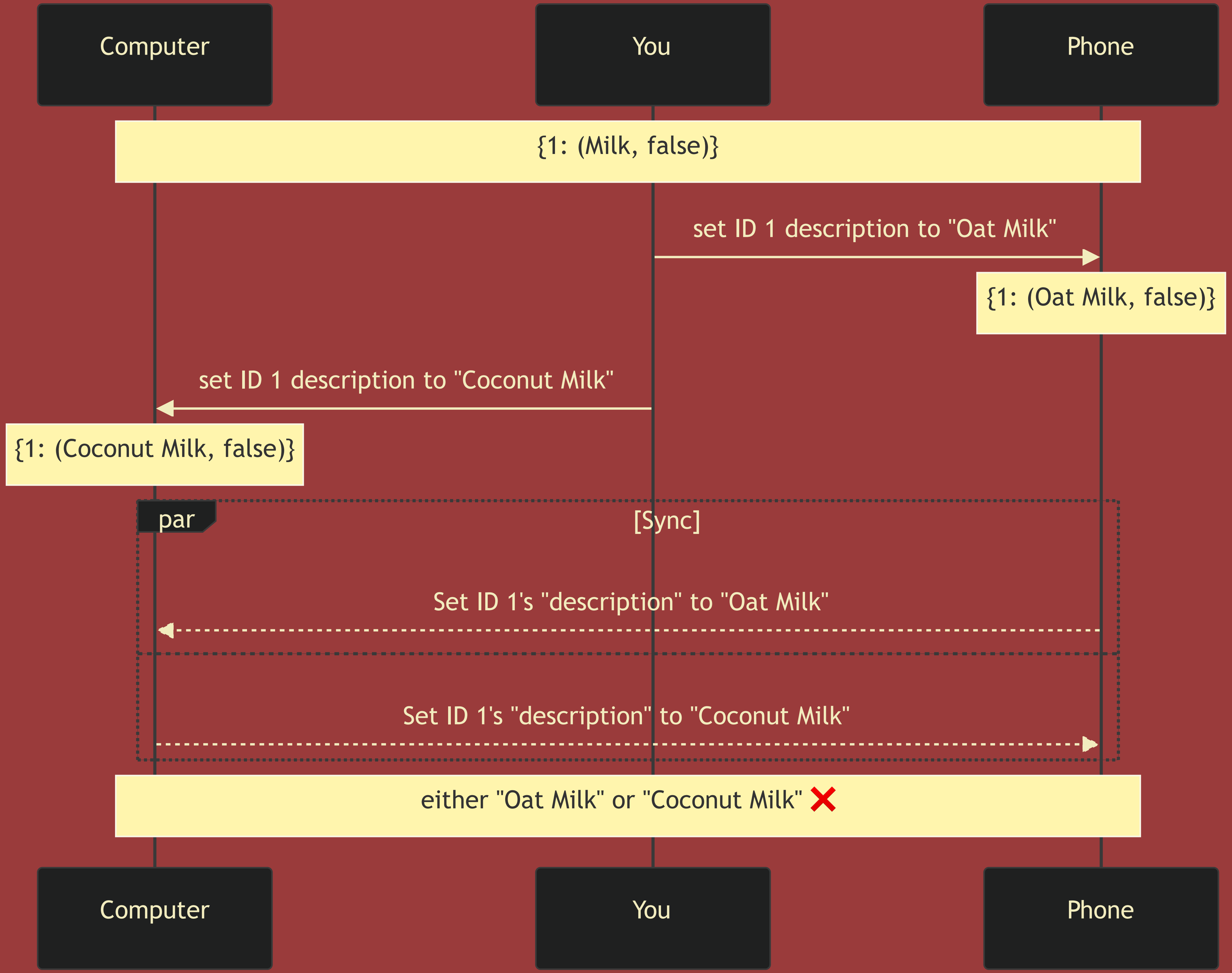
Set ID 1's "description" to "Oat Milk"

{1: (Oat Milk, true)} ✅

Computer

You

Phone



Computer

You

Phone

{1: (Milk, false)}

set ID 1 description to "Oat Milk"

{1: (Oat Milk, false)}

set ID 1 description to "Coconut Milk"

{1: (Coconut Milk, false)}

par

[Sync]

Set ID 1's "description" to "Oat Milk"

Set ID 1's "description" to "Coconut Milk"

either "Oat Milk" or "Coconut Milk" ❌

Computer

You

Phone

The Promise of CRDTs:
Merge any two values consistently, no matter when or where they were created.

But first of all... what are they?

- Conflict-Free (sometimes Convergent) Replicated Data Types
- Collaboration: Yjs, Apple Notes
- Databases: Redis, Riak, Antidote
- Local-first apps

Making a to-do list

We need to:

1. Make a state-based CRDT that can…
   - Keep a set of tasks by ID
   - Resolve concurrent edits in any fields
   - Eventually archive (delete) completed tasks
2. Decide on a sync protocol
3. ✨ Demo ✨

Building a (state-based) CRDT

You need two things:

1. A data structure (your state)
2. A `merge` function, which is:
    - Idempotent ($a \bullet a = a$) so merging identical replicas is OK.
    - Commutative ($a \bullet b = b \bullet a$) so merging in any direction is OK.
    - Associative ($a \bullet (b \bullet c) = (a \bullet b) \bullet c$) so merging in any order is OK.

Merge

```rust
pub trait Merge {
    fn merge(self, other: Self) -> Self;
}
```

Merge

```
pub trait Merge {
    fn merge_mut(&mut self, other: Self);
}
```

The simplest CRDT

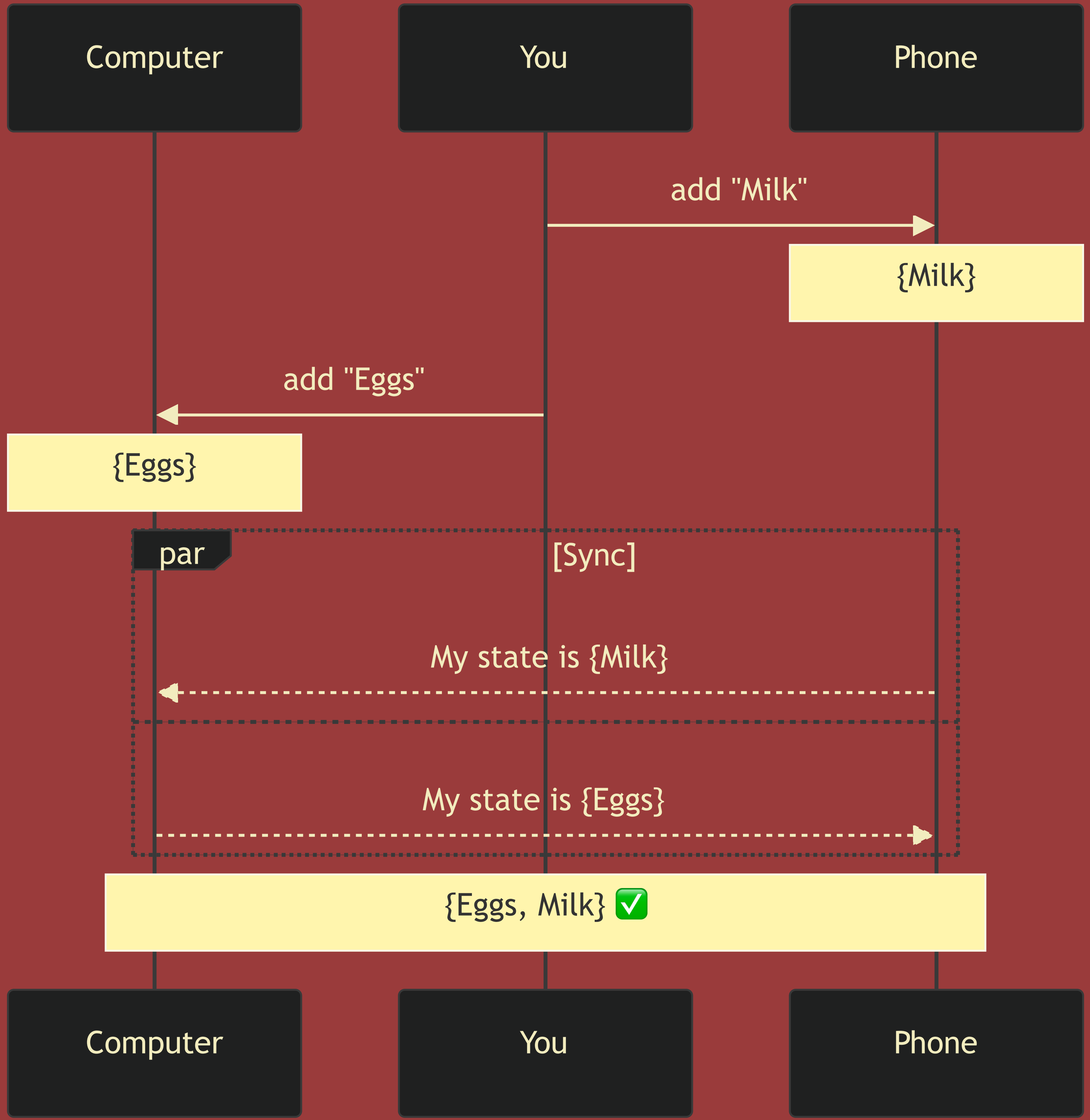
The "grow-only set", or G-Set. The `merge` function is set union.

The simplest CRDT

The "grow-only set", or G-Set. The `merge` function is set union.

```
pub struct GSet<T: Eq + Ord>(BTreeSet<T>);

impl<T: Eq + Ord> Merge for GSet<T> {
    fn merge_mut(&mut self, mut other: Self) {
        self.0.append(&mut other.0);
    }
}
```



Computer

You

Phone

add "Milk"

{Milk}

add "Eggs"

{Eggs}

par

[Sync]

My state is {Milk}

My state is {Eggs}

{Eggs, Milk} ✅

Computer

You

Phone

Sync Protocol

1. Send your whole state to every peer.
2. When you get a state, `merge` it with yours.

- Duplicate/redundant data is a problem
- Delta-state CRDTs help with this

Keeping Fields

The "last write wins register", or LWW-Register. Keep any value, merge by taking the last written value.

- … but what does "last write" mean?
- We can't use wall time because clocks drift.
- We could use a logical clock but having physical time is nice.

Hybrid Logical Clock

```
pub struct HybridLogicalClock {
    timestamp: DateTime<Utc>,
    counter: u16,
    node_id: Uuid,
}

impl HybridLogicalClock {
    pub fn tick(&mut self) {
        let now = Utc::now();
        if now > self.timestamp {
            self.timestamp = now;
            self.counter = 0;
        } else {
            self.counter += 1;
        }
    }
}
```

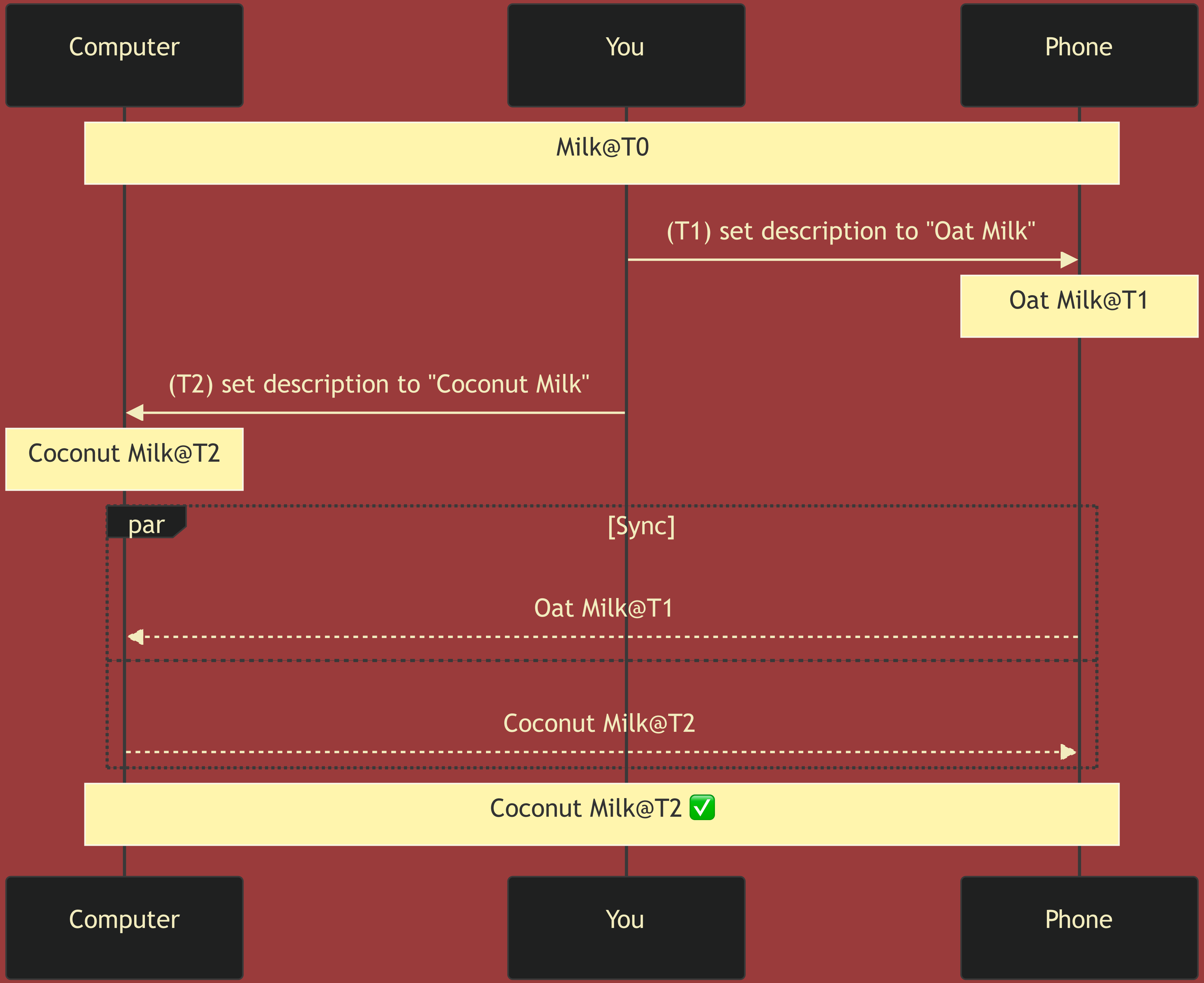
Hybrid Logical Clock

```rust
impl Ord for HybridLogicalClock {
    fn cmp(&self, other: &Self) -> Ordering {
        self.timestamp.cmp(&other.timestamp)
            .then(self.counter.cmp(&other.counter))
            .then(self.node_id.cmp(&other.node_id))
    }
}
```

LWW Register

```
pub struct LWWRegister<T> {
    value: T,
    clock: HybridLogicalClock,
}

impl<T> Merge for LWWRegister<T> {
    fn merge_mut(&mut self, other: Self) {
        if other.clock > self.clock {
            self.value = other.value;
            self.clock = other.clock;
        }
    }
}
```


Computer
You
Phone
Milk@T0
(T1) set description to "Oat Milk"
Oat Milk@T1
(T2) set description to "Coconut Milk"
Coconut Milk@T2
par
[Sync]
Oat Milk@T1
Coconut Milk@T2
Coconut Milk@T2 ✅
Computer
You
Phone

Hold on, wasn't that a conflict?

- Yeah, semantically.
- CRDTs only guarantee that values converge, not that they make sense.
- One way to solve this: multi-value register that allows the user to pick what they wanted.
- Another way: use a sequence CRDT to edit text. Resolve to something like "Coconut Oat Milk"
- Choosing the right CRDTs for your application is about choosing the constraints you can live with.

Grow-Only Map

```rust
pub struct GMap<K: Hash + Ord, V: Merge>(BTreeMap<K, V>);
```

Grow-Only Map

```
impl<K: Hash + Ord, V: Merge> GMap<K, V> {
    pub fn insert(&mut self, key: K, value: V) {
        match self.0.entry(key) {
            Entry::Occupied(mut existing) => {
                existing.get_mut().merge_mut(value);
            }
            Entry::Vacant(vacant) => {
                vacant.insert(value);
            }
        }
    }
}
```

Grow-Only Map

```rust
impl<K: Hash + Ord, V: Merge> Merge for GMap<K, V> {
    fn merge_mut(&mut self, other: Self) {
        for (key, value) in other.0 {
            self.insert(key, value);
        }
    }
}
```

We now have enough to make our to-do list!

Document, V1

```
pub struct Document {
    tasks: GMap<Uuid, Task>,
}

pub struct Task {
    added: LWWRegister<DateTime<Utc>>,
    description: LWWRegister<String>,
    completed: LWWRegister<bool>,
}
```

Document, V1

```
impl Merge for Document {
    fn merge_mut(&mut self, other: Self) {
        self.tasks.merge_mut(other.tasks);
    }
}

impl Merge for Task {
    fn merge_mut(&mut self, other: Self) {
        self.added.merge_mut(other.added);
        self.complete.merge_mut(other.complete);
        self.description.merge_mut(other.description);
    }
}
```

Speaking of constraints…

How about removing values?

A "two-phase map", or 2P-Map

```
pub struct TwoPMap<K: Ord + Clone, V: Merge> {
    adds: BTreeMap<K, V>,
    removes: BTreeSet<K>,
}
```

How about removing values?

```rust
impl<K: Ord + Clone, V: Merge> TwoPMap<K, V> {
    pub fn insert(&mut self, key: K, value: V) {
        if self.removes.contains(&key) {
            return;
        }

        // (snip) same code as GMap::insert
    }

    pub fn remove(&mut self, key: K) {
        self.adds.remove(&key);
        self.removes.insert(key);
    }
}
```

How about removing values?

```
impl<K: Ord + Clone, V: Merge> Merge for TwoPMap<K, V> {
    fn merge_mut(&mut self, mut other: Self) {
        // same as GSet
        self.removes.append(&mut other.removes);

        // same as GMap
        for (key, value) in other.adds {
            self.insert(key, value);
        }

        // drop unnecessary values
        self.adds.retain(|k, _| !self.removes.contains(k))
    }
}
```

Metadata Overhead

In general, this is a problem.

- 2P-Sets and 2P-Maps keep tombstones forever.
- G-Sets and G-Maps can never shrink.
- LWW Registers keep an HLC (ours are 34 bytes each.)
- Sequence CRDTs have a per-item overhead. (So "Hello" would have 5 characters worth)
- Compression helps.
- Optimizations in industrial CRDTs (e.g. ORSWOT) mostly go towards avoiding this problem.

Archiving Completed Tasks

```diff
 pub struct Document {
-    tasks: GMap<Uuid, Task>,
+    tasks: TwoPMap<Uuid, Task>,
 }
```

Bookkeeping

```rust
pub struct Replica {
    id: Uuid,
    clock: HybridLogicalClock,
    document: Document,
}

impl Replica {
    pub fn receive(&mut self, other: Replica) {
        self.document.merge_mut(other.document);
        self.clock = self.clock.max(other.clock).claim(self.id);
    }
}
```

Demo!

Thanks 👋

- bytes.zone
- github.com/BrianHicks/rust-crdt-talk

Implementations:

- automerge.org
- loro.dev
- crates.io/crates/crdts