BRB
& BRBDT’s
& Anti-Entropy
& BRB Membership

Byzantine Reliable Broadcast

Byzantine Reliable Broadcast (BRB)

An algorithm that guarantees all honest procs apply BRBDT Operations in Source Order.

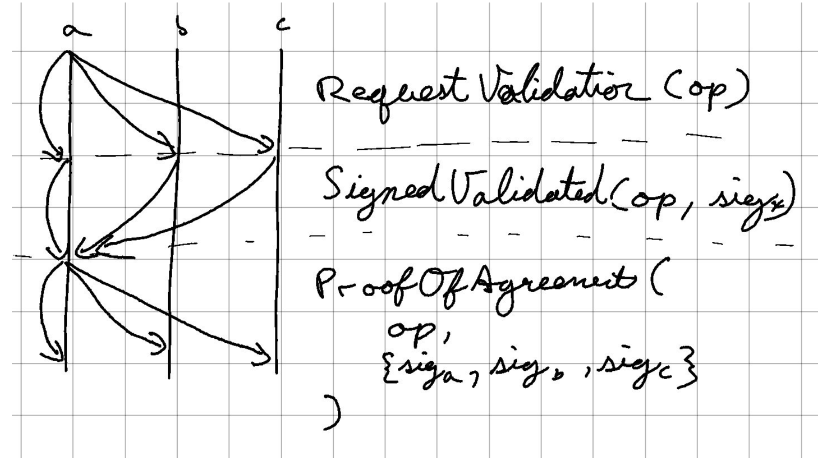
BRBDT: Byzantine Reliable Broadcast Data Types

- These are data types that are protected by BRB. e.g. AT2, ORSWOT, CRDT’s

Source Order: Op’s are ordered in the same order that a source produced them, but no ordering guarantees are made between Op’s from different sources.

BRB Algorithm (high level)

1. An Op is initiated by broadcasting a request for validation to the network.
2. Network members validate the Op and sign it and respond back to the source with their signature.
3. The source collects these signatures
4. Once it receives a super majority of signatures, the source broadcasts out a proof of agreement.
5. Once members receive the proof, they apply the Op to the BRBDT.



See the next slide for a worked example.

BRB - Sequence Diagram

```
A = B = C = Proc {
  pending_proof: {},
  peers: {A, B, C},
  received: {A0, B0, C0}
  delivered: {A0, B0,
C0}
}
```

```
A = B = C = Proc {
  pending_proof: {},
  peers: {A, B, C},
  received: {A1, B0, C0}
  delivered: {A0, B0,
C0}
}
```

```
A = Proc {
  pending_proof: {M: {Asig, Bsig,
Csig}},
  peers: {A, B, C},
  received: {A1, B0, C0}
  delivered: {A0, B0, C0}
}
```

```
B = C = Proc {
  pending_proof: {},
  peers: {A, B, C},
  received: {A1, B0, C0}
  delivered: {A0, B0,
C0}
}
```

```
A = B= C = Proc {
  pending_proof: {},
  peers: {A, B, C},
  received: {A1, B0, C0}
  delivered: {A1, B0,
C0}
}
```

1. **A** initiates an operation on the BRBDT and wraps it in a Msg:

```
m = Msg { op,  A1
}
```

2. **A** Broadcasts request for message validation:

```
RequestValidation(m)
```

3. procs validate msg and inc. **A**'s receive clock

4. All members send **A** signed msg as proof of their validation:

```
SignedValidated(m, sig)
```

5. Once **A** has collected a quorum of signatures, **A** Broadcasts these signatures as proof of agreement

```
ProofOfAgreement(
  msg,
  {Asig, Bsig, Csig}
)
```

6. Procs apply the op to the BRBDT after verifying the proof

1. 2a 2b 2c 3a 3b 3c 4c 4b 4a 5a 5b 5c 6a 6b 6c

A **B** **C**

BRBDataTypes (BRBDT)

BRB is agnostic to the underlying data type it protects. We refer to this data type as BRBDT (a throwback to CRDT's).

BRB will provide a source ordering guarantee. If a BRBDT requires stricter ordering guarantees, it must provide them through the `validate(..)` method by refusing to validate an Op if it does not satisfy the stricter ordering requirements.

```rust
 8 pub trait BRBDataType: Debug {
 9     type Op: Debug + Clone + Hash + Eq + Serialize;
10 
11     /// initialize a new replica of this datatype
12     fn new(actor: Actor) -> Self;
13 
14     /// Protection against Byzantines
15     fn validate(&self, from: &Actor, op: &Self::Op) -> bool;
16 
17     /// Executed once an op has been validated
18     fn apply(&mut self, op: Self::Op);
19 }
```

Called when an actor is first instantiated.

Called when we've been requested to validate an operation.

Called once BRB has seen network agreement over this operation.

BRB Anti-Entropy

Anti-Entropy is a process used to keep network members up-to-date and to ensure totality over Op delivery, we also use it for onboarding new members or creating read-only replicas.

Totality: if one honest process applies an op, eventually all other honest processes will apply the same op or leave the network.

Motivating example: Say we run through the BRB algorithm up to the phase where the source has collected a super-majority of signatures. The source can then send this proof to just one honest member and then halt. The rest of the network will not learn of this proof of agreement and never apply it. This violates totality.

BRB Anti-Entropy

Honest procs periodically initiate anti-entropy with each voting members.

- Initiate Anti-Entropy by sending an “Anti-Entropy packet” containing your current generation and your current delivered VClock.

```
AntiEntropy {
    generation: brb_membership::Generation,
    delivered: crdts::VClock<Actor>,
},
```

- A proc receiving these Anti-Entropy packets responds by comparing the generation and delivered VClock with their own and sending back any new Op’s or Membership changes they have seen.
- **Onboarding** is handled by the new proc initiating Anti-Entropy with:

```
AntiEntropy { generation: 0, delivered: VClock::new() }
```

BRB Membership

A consensus algorithm for dynamically reconfiguring network membership.

Each membership reconfiguration is marked by a corresponding generation clock increment.

Each generation, any member may propose at most 1 Reconfig.

The BRB Membership algorithm will try to reach consensus over the set of reconfigurations we will apply to take us to the next generation

```
pub struct Vote {
    gen: Generation,
    ballot: Ballot,
    voter: Actor,
    sig: Sig,
}
pub enum Ballot {
    Propose(Reconfig),
    Merge(BTreeSet<Vote>),
    SuperMajority(BTreeSet<Vote>),
}

pub enum Reconfig {
    Join(Actor),
    Leave(Actor),
}
```

BRB Membership | Algorithm

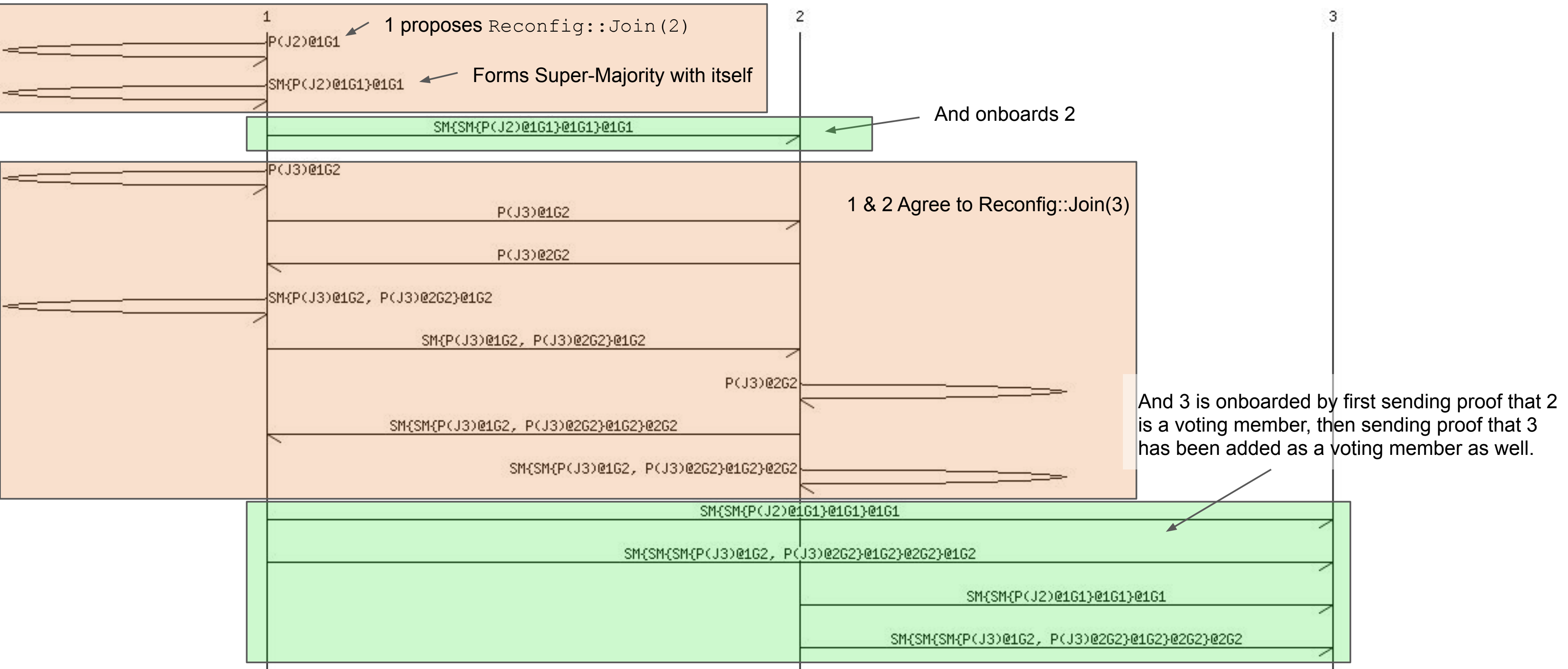
- A member broadcasts a vote with `Ballot::Propose(reconfig)`.
- When a member receives a vote, it logs the vote and then considers the following cases
- Have we voted already?
  - If no, then adopt this ballot as our own and broadcast it as our vote.
- Is it still possible to form super majority over a set of reconfigs?
  - If not, we have a split vote, change our vote to a `Ballot::Merge(..)` containing all votes we've seen so far
- Do we have a super majority over super majorities?
  - If yes, the algorithm terminates and we advance the generation and apply the reconfigurations
- Do we have a supermajority?
  - If yes, then change our vote to Ballot::SuperMajority containing the votes we have seen and broadcast

See the next few slides for some worked examples of the network agreeing on network reconfigurations.

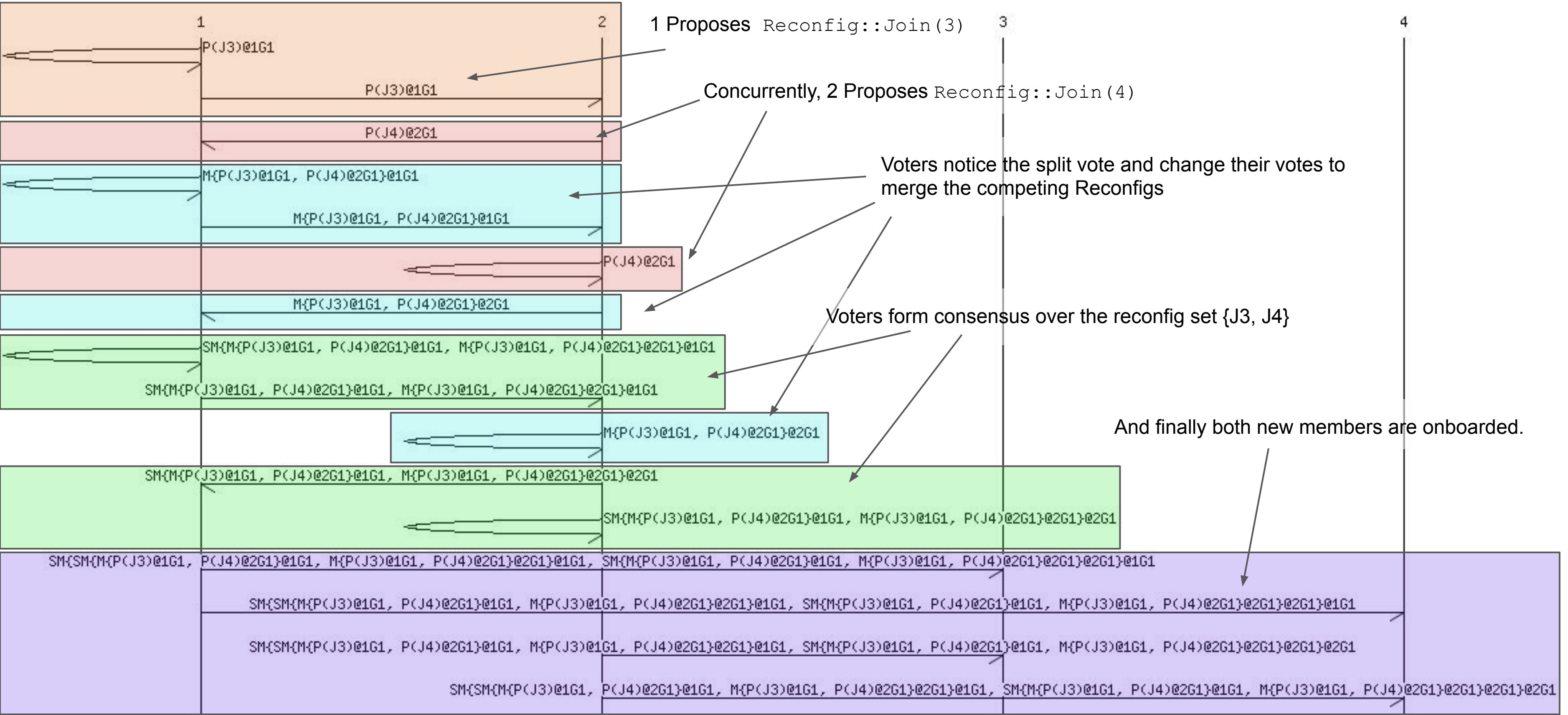
```
pub struct Vote {
    gen: Generation,
    ballot: Ballot,
    voter: Actor,
    sig: Sig,
}
pub enum Ballot {
    Propose(Reconfig),
    Merge(BTreeSet<Vote>),
    SuperMajority(BTreeSet<Vote>),
}

pub enum Reconfig {
    Join(Actor),
    Leave(Actor),
}
```

Onboarding 2 new Procs:



Split Vote | Procs 1 & 2 concurrently propose reconfigs



BRB Membership | Forced Reconfigs

Forced reconfigurations are used to forcibly reconfigure network membership.

This occurs when a network is first created (genesis proc), or when the network suffers a catastrophic failure where the network can no longer form super-majority.

Any new process who is onboarded will need to manually apply these forced reconfigurations or onboarding will fail since the new process can not verify these forced reconfigs from the history alone.