Fast and Reliable P2P Without Breaking the Memory Budget

Ognjen Maric, DFINITY Foundation

Joint work with:
Manu Drijvers, Tim Gretler, Yotam Harchol, Tobias Klenze, Stefan Neamtu, Yvonne-Anne Pignolet, Rostislav Rumenov, Daniel Sharifi, Victor Shoup

P2P Broadcast in (Blockchain) Consensus

Validators can broadcast a *vote* message containing together with their heights h(s) and h(t). We requi the vote is considered invalid. If th invalid. Together with the sig

Casper the Friendly Fi

2. At time $t_{r,v}$, if v is the primary $E_{r-1,v}$. If they have finalised it,

3. If v is a voter for the prevote of round r, v waits until either it is at least time $t_{r,v} + 2T$ or round r is completable, then broadcasts a prevote. They prevote for the head of the best chain containing

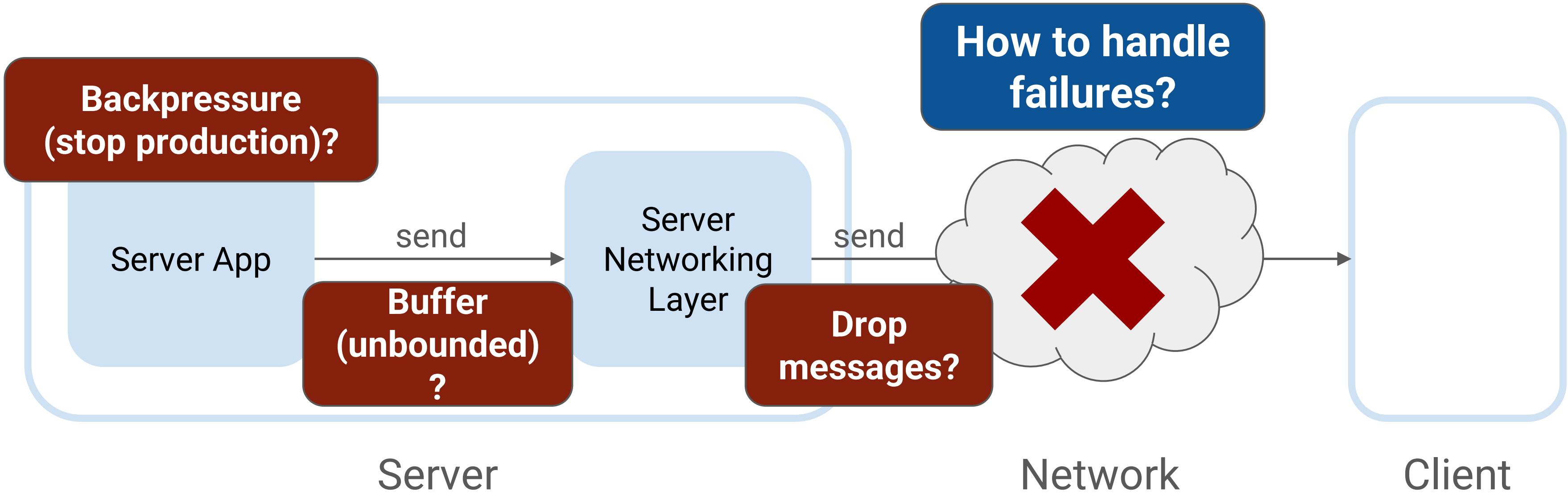
// *Finish*
combine
broadca
done ←
if $\mathcal{N} \subseteq$ {
(b) not *prop*
// *Pro*

Textbook stuff, solved problem?

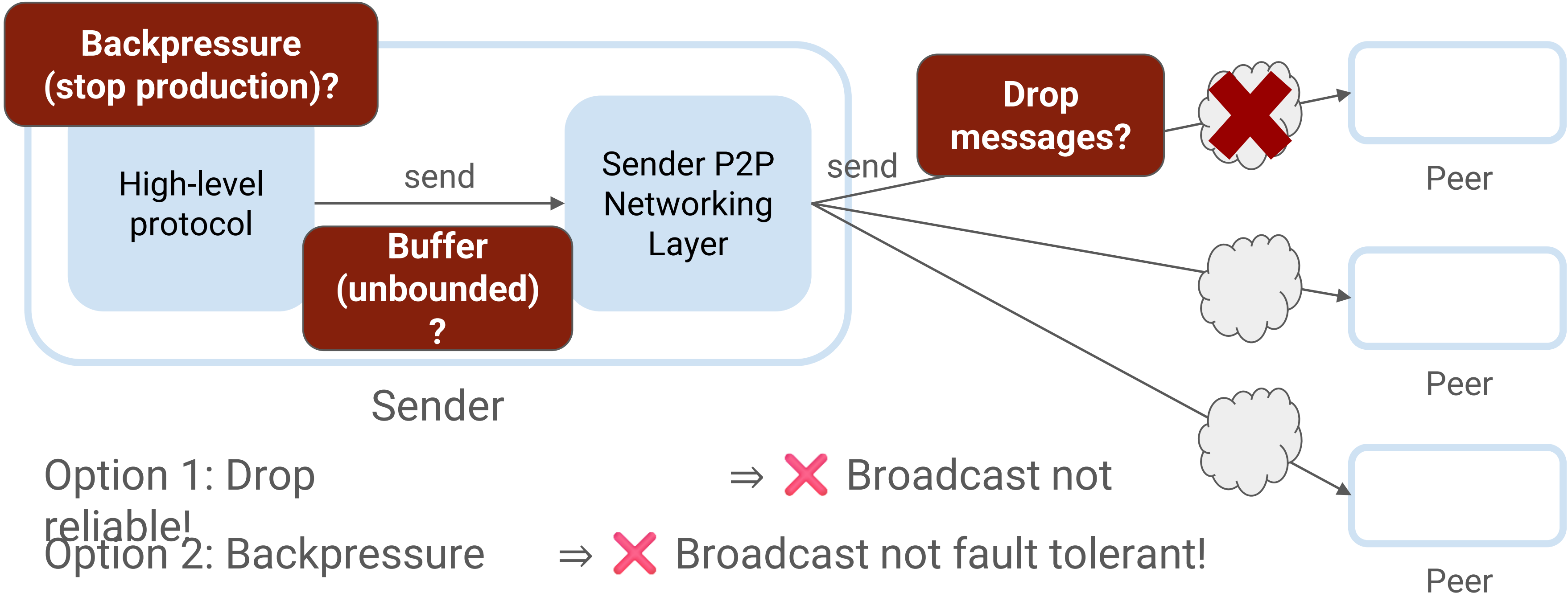
Simple problems sometimes not so simple: Reliable, fault-tolerant broadcast w/ bounded memory

...

Handling Failures: Client-Server Scenario



Handling Failures: P2P Broadcast Scenario

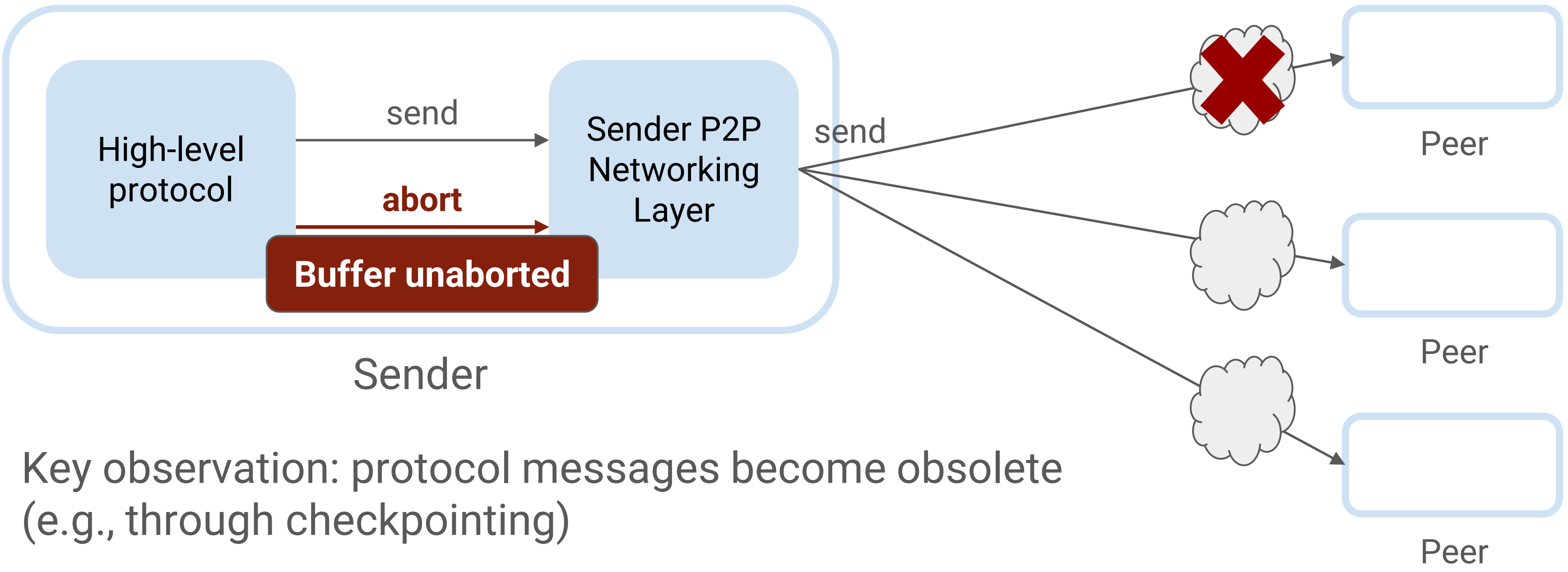


Option 1: Drop ⇒ ❌ Broadcast not reliable!

Option 2: Backpressure ⇒ ❌ Broadcast not fault tolerant!

Option 3: Buffer ⇒ ❌ Eventually out-of-memory (DoS)!

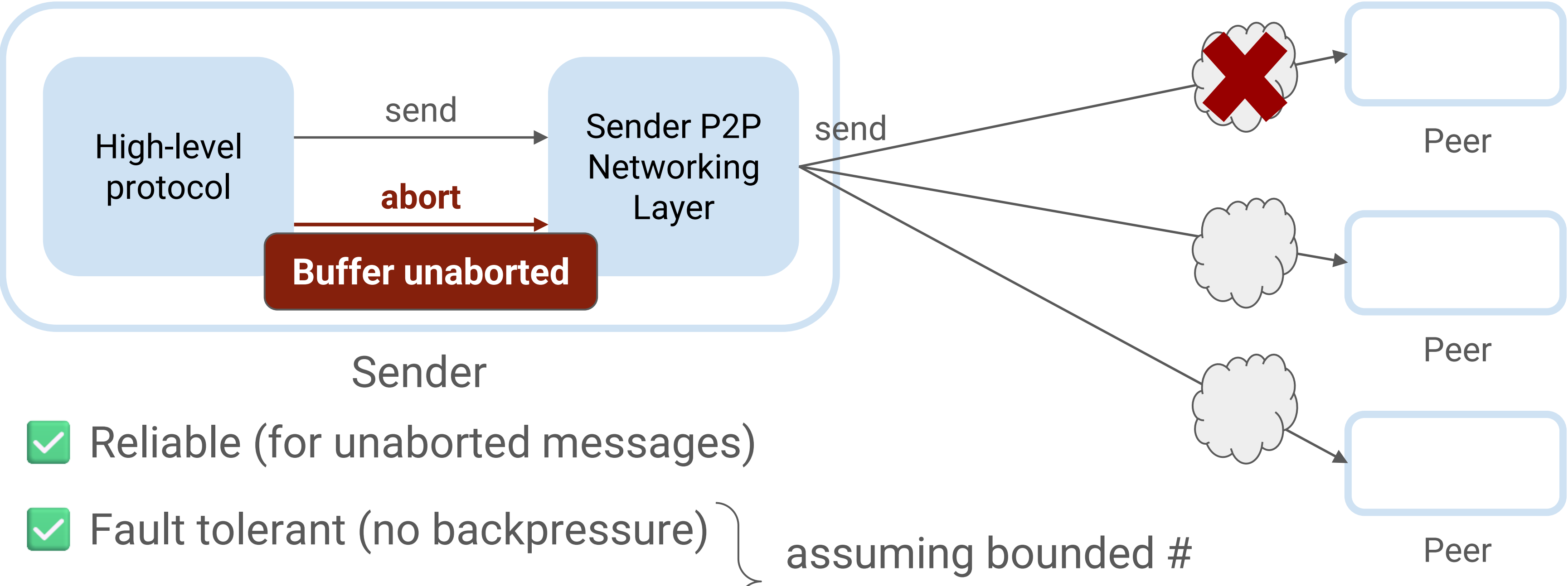
Our Solution: *Abortable* Broadcast



Key observation: protocol messages become obsolete (e.g., through checkpointing)

Idea: add explicit abort call for obsolete messages; buffer only unaborted

Our Solution: *Abortable* Broadcast



- ✅ Reliable (for unaborted messages)
- ✅ Fault tolerant (no backpressure)
- ✅ Bounded memory usage

} assuming bounded # of unaborted messages

Talk Outline

1. Abortable broadcast: interface, assumptions and guarantees

1. Our implementation of abortable broadcast

1. Evaluation & related work

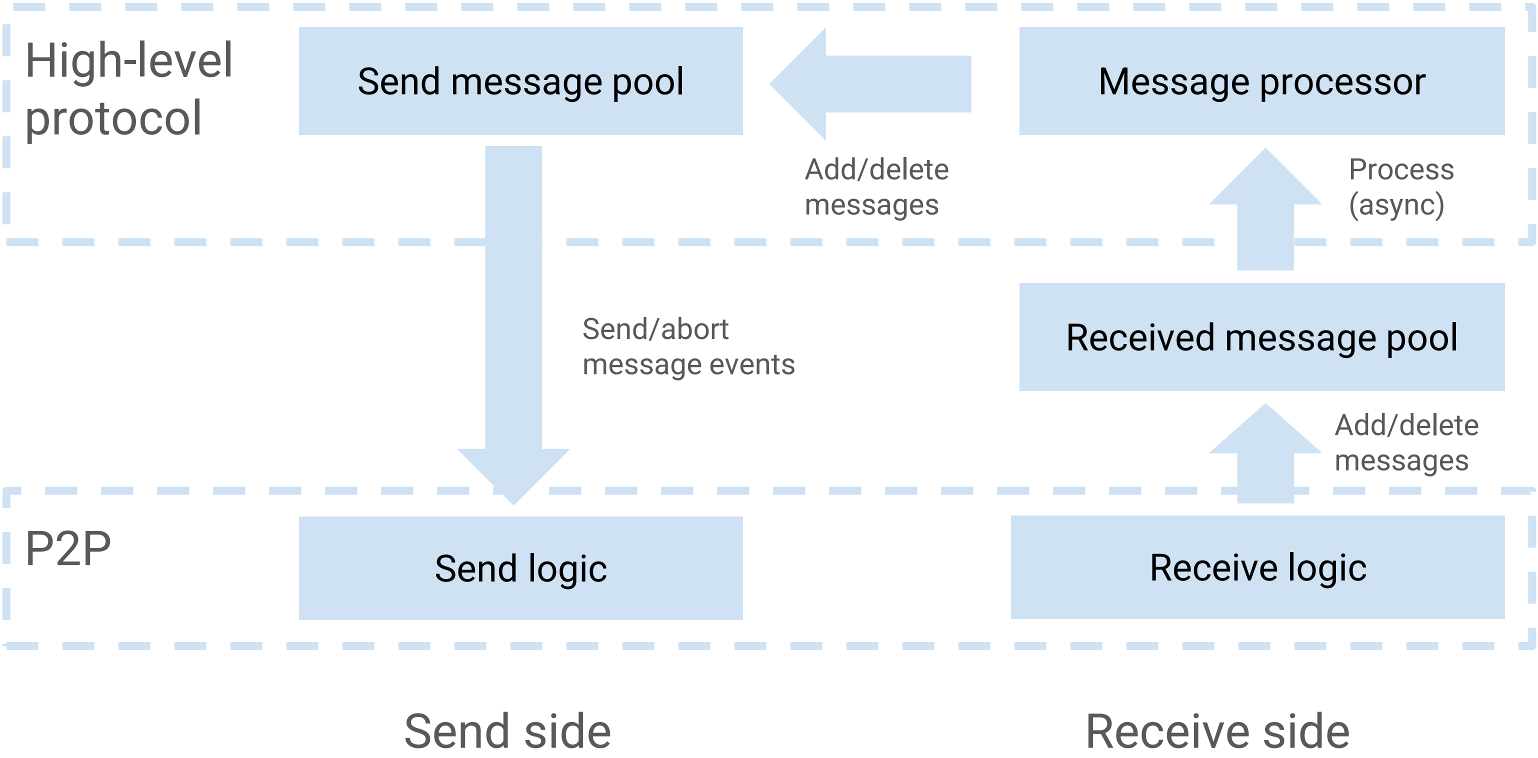
Talk Outline

1. Abortable broadcast: interface, assumptions and guarantees

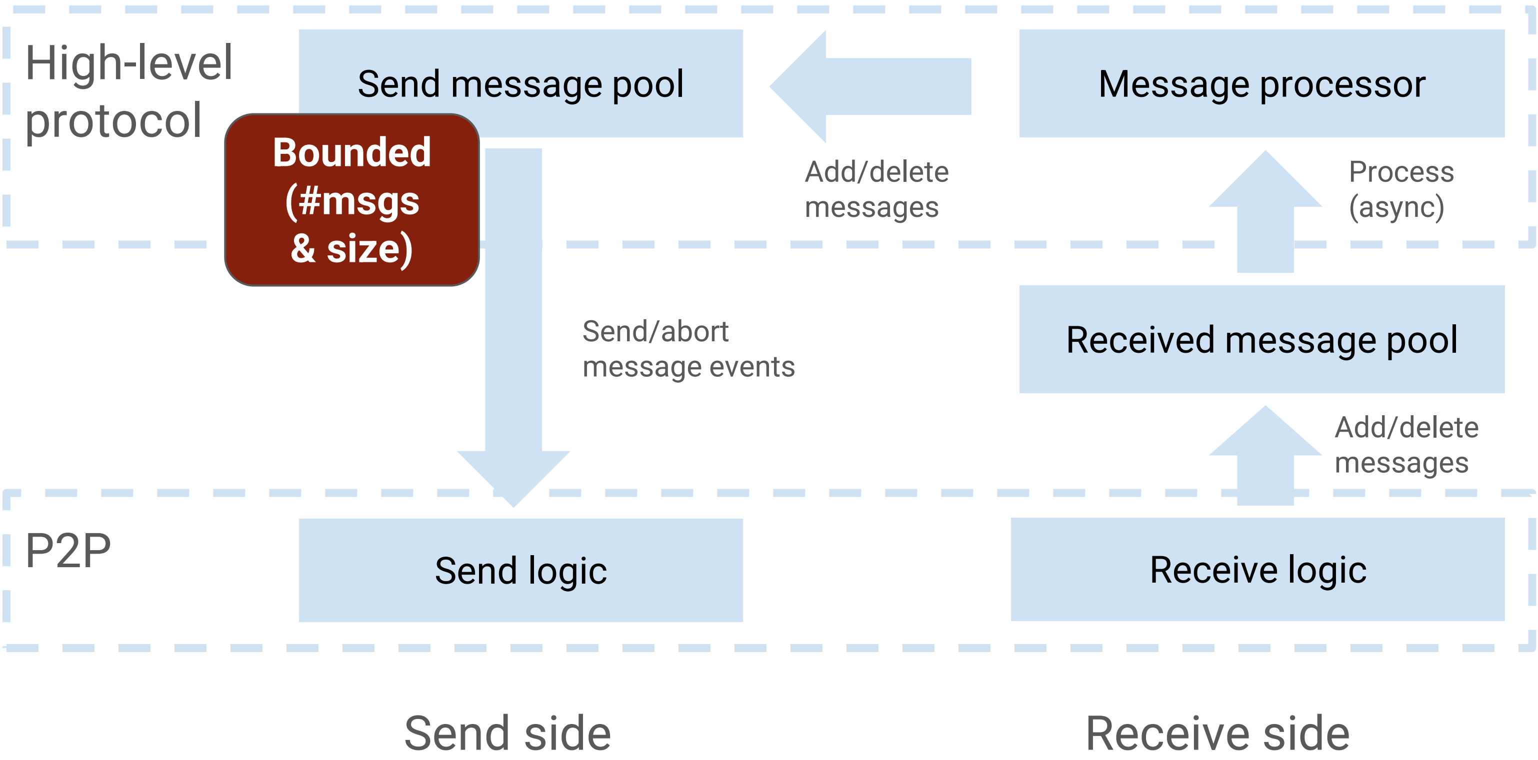
1. Our implementation of abortable broadcast

1. Evaluation & related work

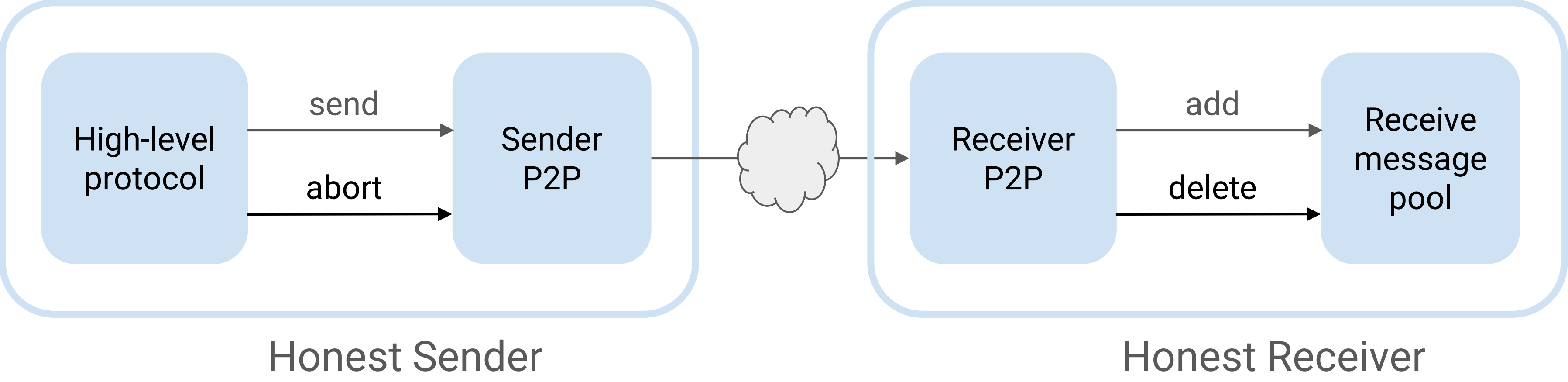
Abortable Broadcast: Interface



Abortable Broadcast: *Assumption*



Abortable Broadcast: *Guarantees*



G1: Sent & not aborted messages *eventually* received

G2: Sent & not aborted messages received *timely*, when network behaves

G3: P2P and receive pool use *bounded* memory

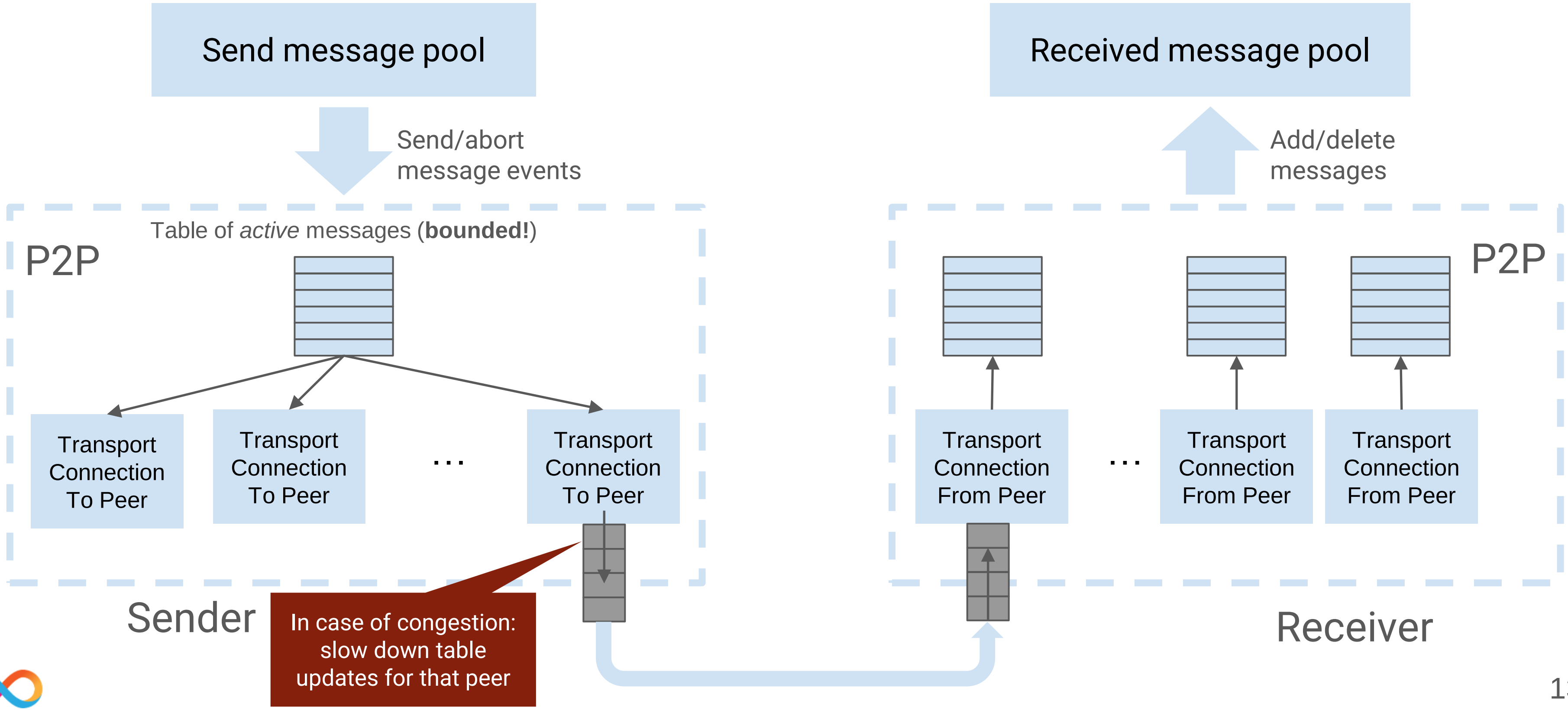
Talk Outline

1. Abortable broadcast: interface, assumptions and guarantees

1. Our implementation of abortable broadcast

1. Evaluation & related work

Abortable Broadcast: Implementation (Conceptual)



The “Slot Table” data structure

0	content: A, version: 2
1	content: none, version: 3
2	content: C, version: 1
3	content: D, version: 2
4	content: E, version: 4
5	content: none, version: 0

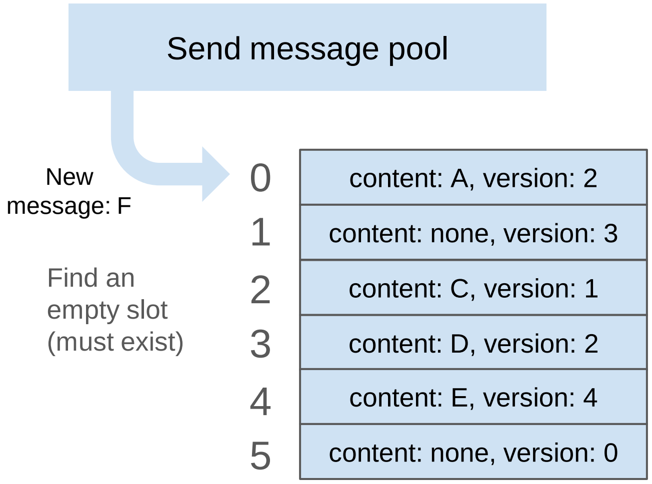
- Numbered slots, bounded
- Each slot has *content* and a *version*
  - Slots may be empty
  - Version is increased with every change to the table

The “Slot Table” data structure

0	content: A, version: 2
1	content: none, version: 3
2	content: C, version: 1
3	content: D, version: 2
4	content: E, version: 4
5	content: none, version: 0

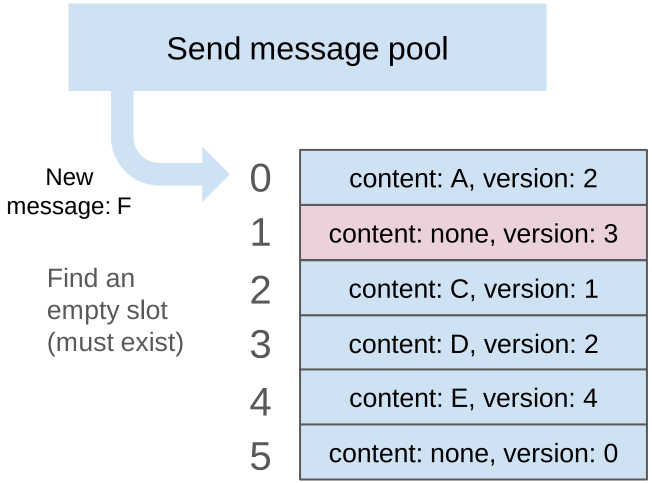
- Numbered slots, **bounded (G3)**
- Each slot has *content* and a *version*
  - Slots may be empty
  - Version is increased with every change to the table

The “Slot Table” data structure



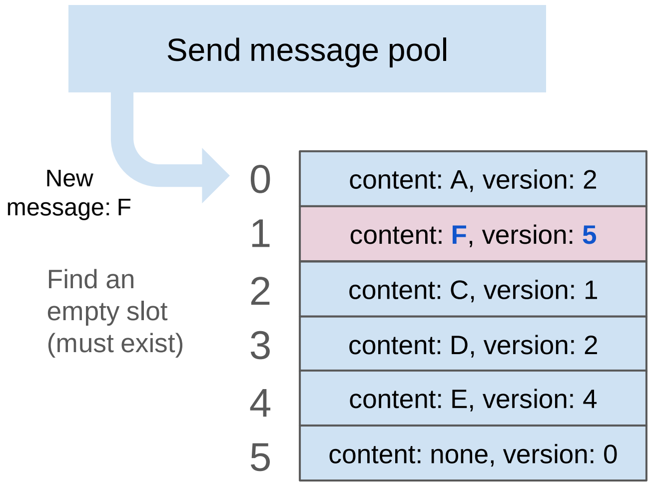
- Numbered slots, bounded
- Each slot has *content* and a *version*
  - Slots may be empty
  - Version is increased with every change to the table

The “Slot Table” data structure



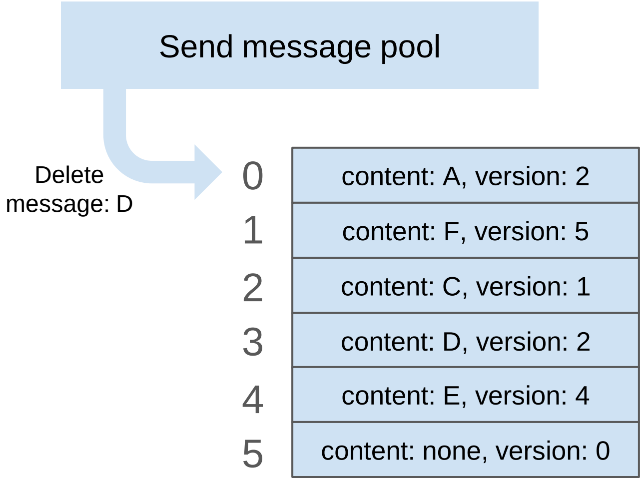
- Numbered slots, bounded
- Each slot has *content* and a *version*
  - Slots may be empty
  - Version is increased with every change to the table

The “Slot Table” data structure



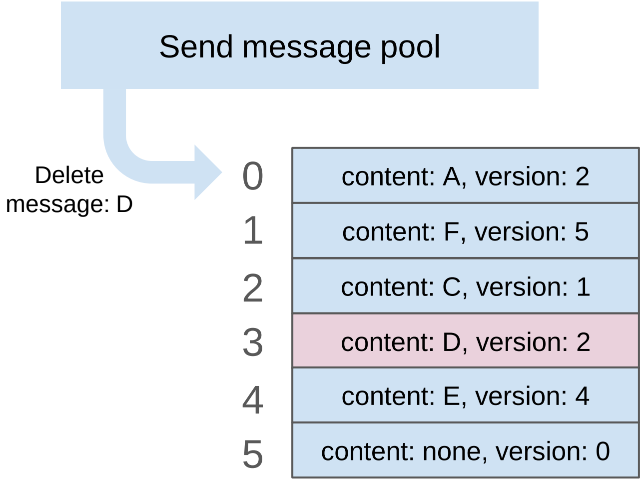
- Numbered slots, bounded
- Each slot has *content* and a *version*
  - Slots may be empty
  - Version is increased with every change to the table

The “Slot Table” data structure



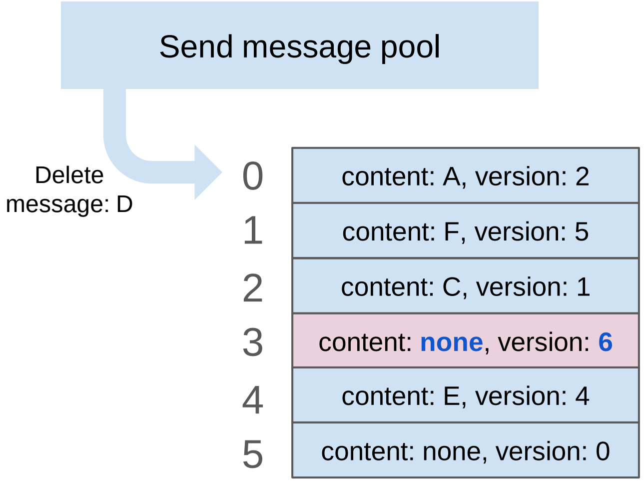
- Numbered slots, bounded
- Each slot has *content* and a *version*
    - Slots may be empty
    - Version is increased with every change to the table

The “Slot Table” data structure



- Numbered slots, bounded
- Each slot has *content* and a *version*
    - Slots may be empty
    - Version is increased with every change to the table

The “Slot Table” data structure



- Numbered slots, bounded
- Each slot has *content* and a *version*
  - Slots may be empty
  - Version is increased with every change to the table

The “Slot Table” data structure

Send message pool

Events:
1. Add A (0)
2. Add B (1)
3. Add C (2)
4. Add D (3)
5. Add E (4)
6. Add F (5)
7. Remove B (1)
8. Remove E (4)
9. Remove F (5)
10. Add G (5)

0	content: A, version: 1
1	content: none, version: 7
2	content: C, version: 3
3	content: D, version: 4
4	content: none, version: 8
5	content: G, version: 10

Sender

Sync protocol

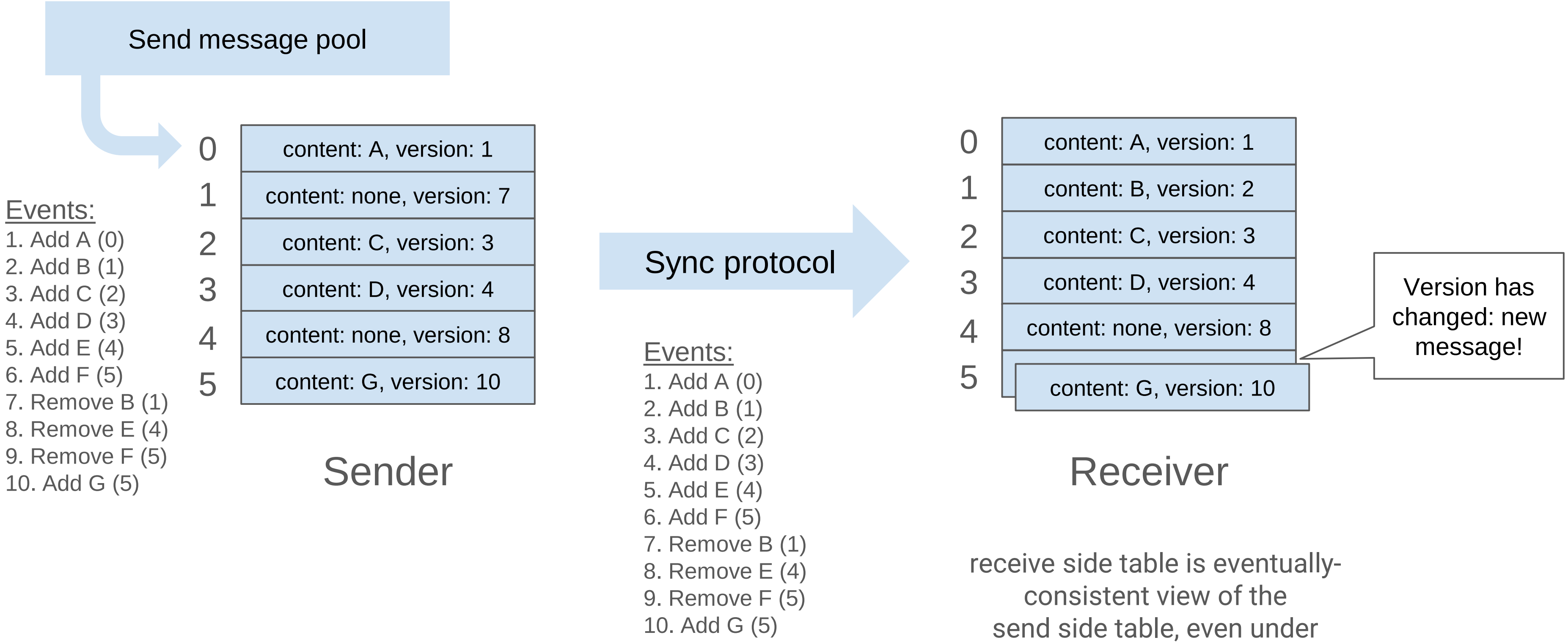
Events:
1. Add A (0)
2. Add B (1)
3. Add C (2)
4. Add D (3)
5. Add E (4)
6. Add F (5)
7. Remove B (1)
8. Remove E (4)
9. Remove F (5)
10. Add G (5)

0	content: A, version: 1
1	content: B, version: 2
2	content: C, version: 3
3	content: D, version: 4
4	content: none, version: 8
5	content: F, version: 6

Receiver

receive side table is eventually-consistent view of the send side table, even under congestion

The “Slot Table” data structure



receive side table is eventually-consistent view of the send side table, even under congestion

The “Slot Table” data structure

Send message pool

Events:

1. Add A (0)
2. Add B (1)
3. Add C (2)
4. Add D (3)
5. Add E (4)
6. Add F (5)
7. Remove B (1)
8. Remove E (4)
9. Remove F (5)
10. Add G (5)

0	content: A, version: 1
1	content: none, version: 7
2	content: C, version: 3
3	content: D, version: 4
4	content: none, version: 8
5	content: G, version: 10

Sender

Sync protocol

Events:

1. Add A (0)
2. ~~Add B (1)~~
3. Add C (2)
4. Add D (3)
5. ~~Add E (4)~~
6. ~~Add F (5)~~
7. Remove B (1)
8. Remove E (4)
9. ~~Remove F (5)~~
10. Add G (5)

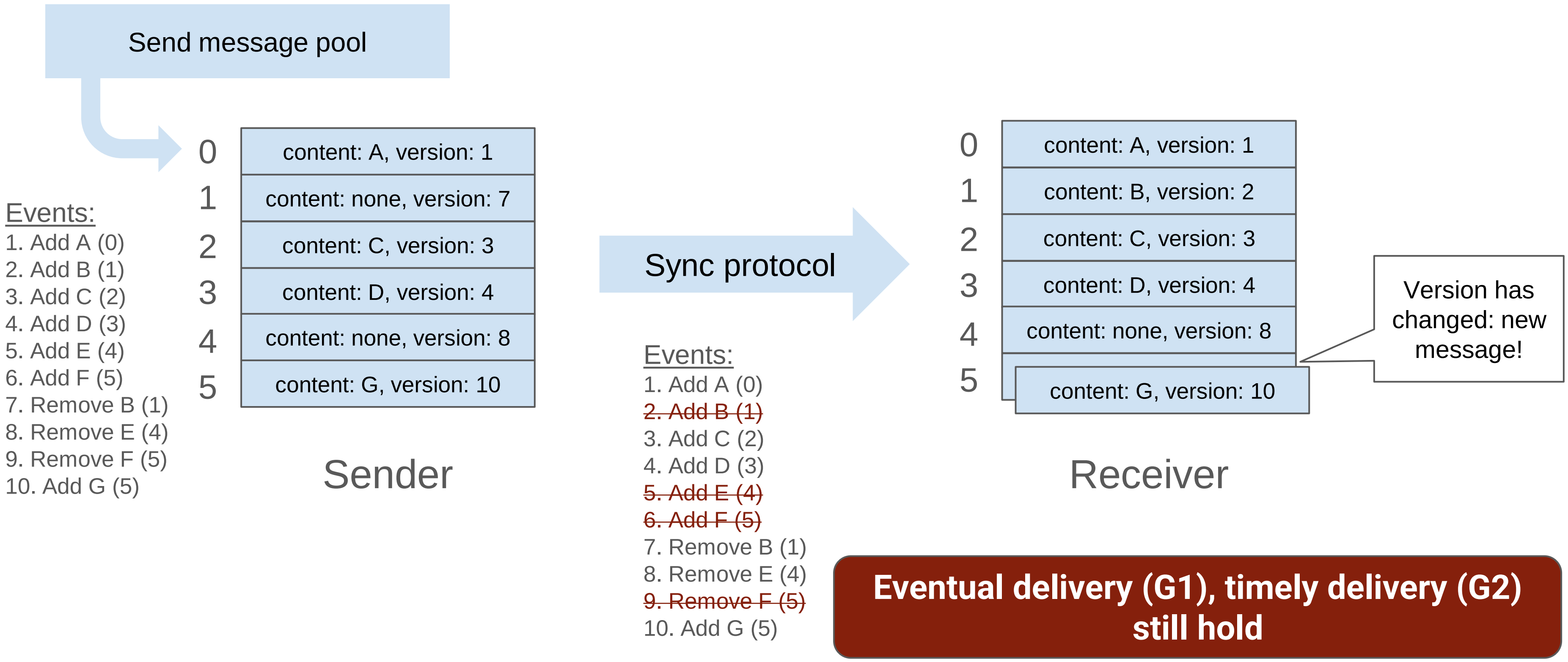
0	content: A, version: 1
1	content: B, version: 2
2	content: C, version: 3
3	content: D, version: 4
4	content: none, version: 8
5	content: G, version: 10

Version has changed: new message!

Receiver

Obsolete slot events dropped on network congestion

The “Slot Table” data structure



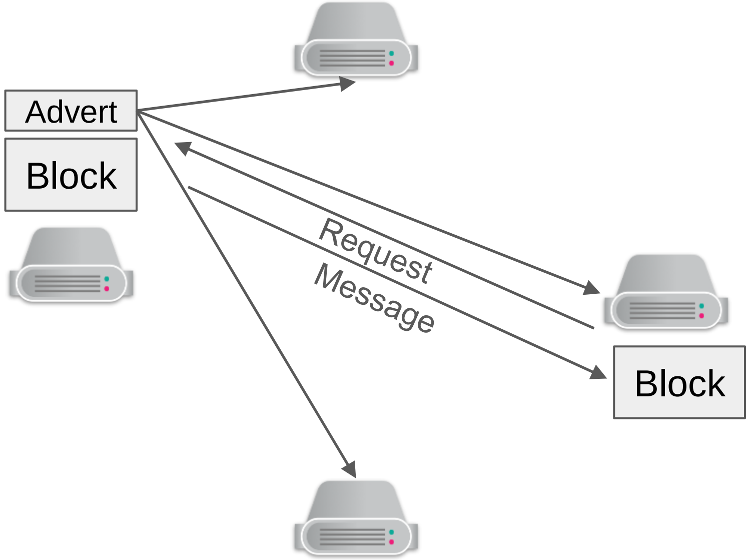
Bandwidth Optimization

For large messages, nodes broadcast just their *adverts*

Receivers request the full messages they are interested in

- Many messages are relayed; no need to receive them from all peers
- Some messages may not be interesting, or may only become interesting later

Decreases latency, saves bandwidth, increases throughput



Talk Outline

1. Abortable broadcast: interface, assumptions and guarantees

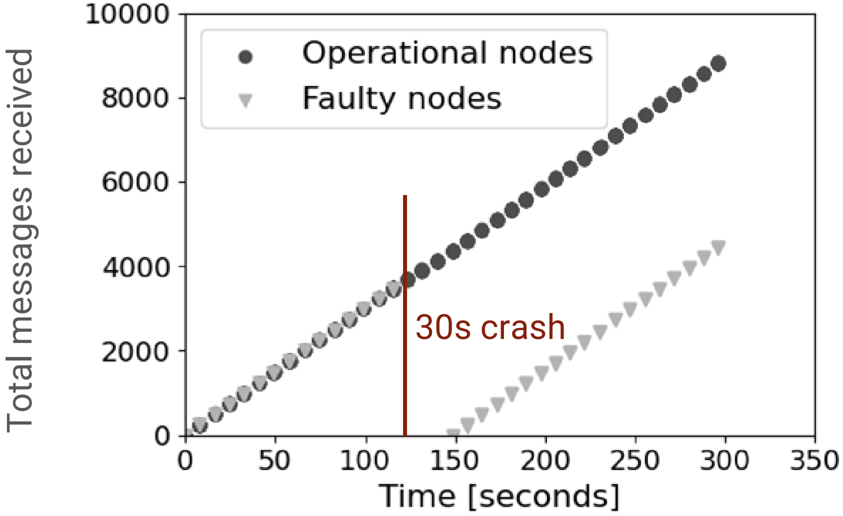
1. Our implementation of abortable broadcast

1. Evaluation & related work

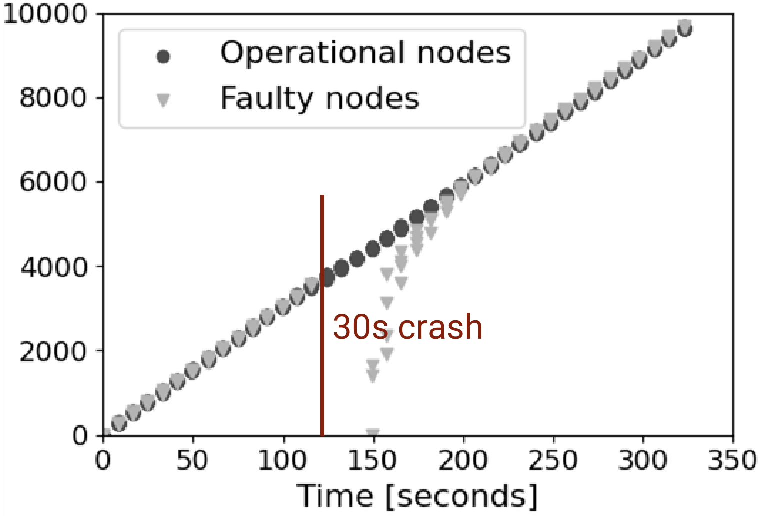
Related Work

- Little in terms of guaranteed message delivery with bounded memory
  - PBFT includes a bespoke retransmission mechanism to keep memory bounded
- Bitcoin, ETH1.0: no checkpointing, so unbounded memory
  - Bounded in practice by low throughput
  - ~600GB state for Bitcoin
- GossipSub (libp2p):
  - Used by ETH2.0, Polkadot, Polygon, Mina, …
  - Bounded memory
  - No delivery guarantees; clients must implement bespoke retransmission

Comparison to GossipSub: Delivery Guarantees



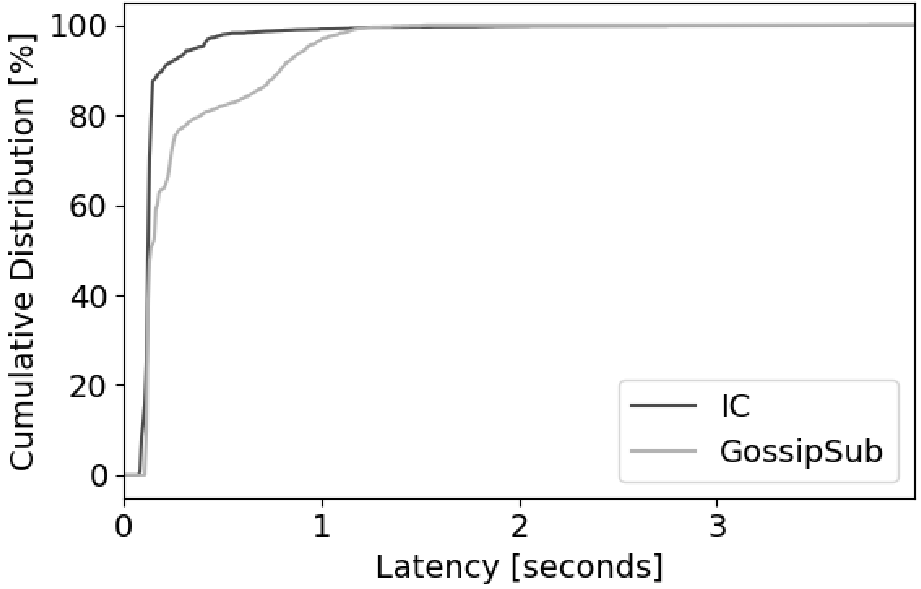
GossipSub



Our implementation

31 nodes, crash 4/31 for 30 seconds

Comparison to GossipSub: Latency



31 nodes, send rate up to 4 Gbps (12.5 Gbps links)

Conclusion & Future Work

Takeaways

- True (Byzantine) fault tolerance requires bounding memory
- Reliability not so simple when bounded
- Our solution achieves all three

Future work

- Better bandwidth utilization
  - More peers: overlay networks, ECCs?
  - Better handling of input messages
- Better resilience to volumetric attacks

https://dfinity.org/grants

Appendix

Bounding the receive pools

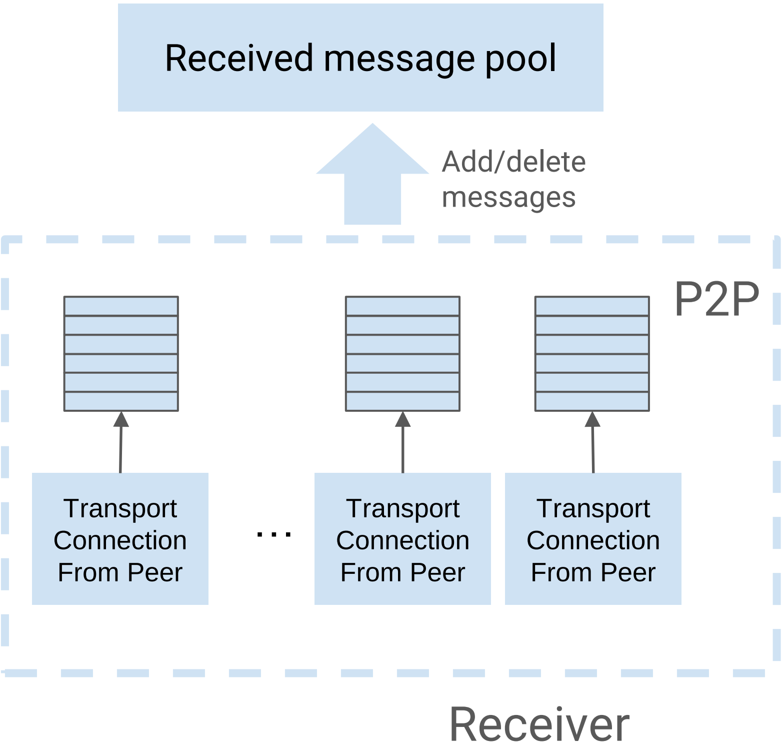
⇒ If a message is aborted by all senders,
it is no longer needed

→ can be deleted from the receive pool

The receive pool is bounded using the same bound on the slot tables

(More specifically, |pool| < C*n, for n peers and a bound C)

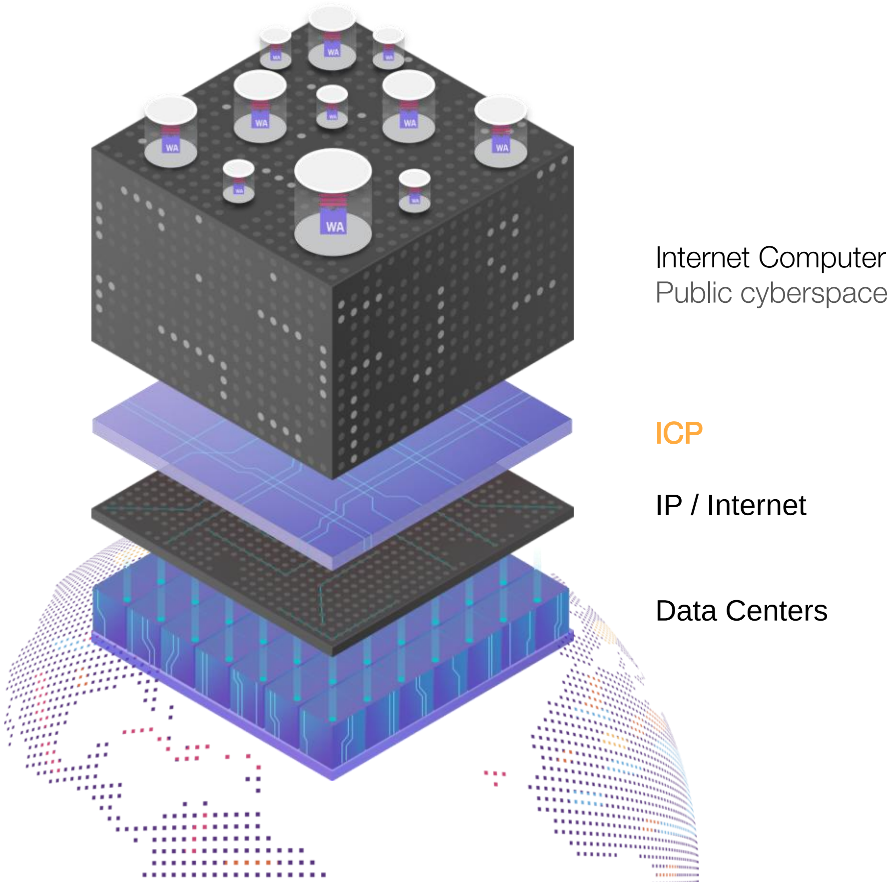
Bounded memory guarantee (G3) fulfilled!



Internet Computer Protocol (ICP)

Coordination of nodes in independent data centers, jointly performing any computation for anyone

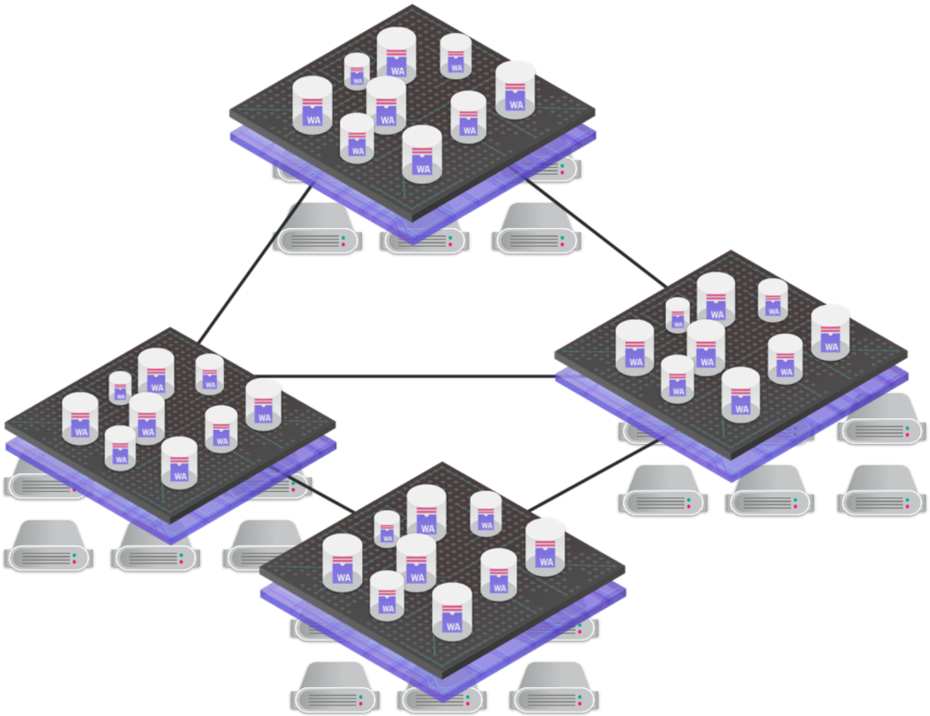
- Create Internet Computer blockchains
- Guarantee safety and liveness of smart contract execution despite Byzantine participants



Scalability: Nodes and Subnets

Nodes are partitioned into subnets

Canister smart contracts are assigned to different subnets



Scalability: Nodes and Subnets

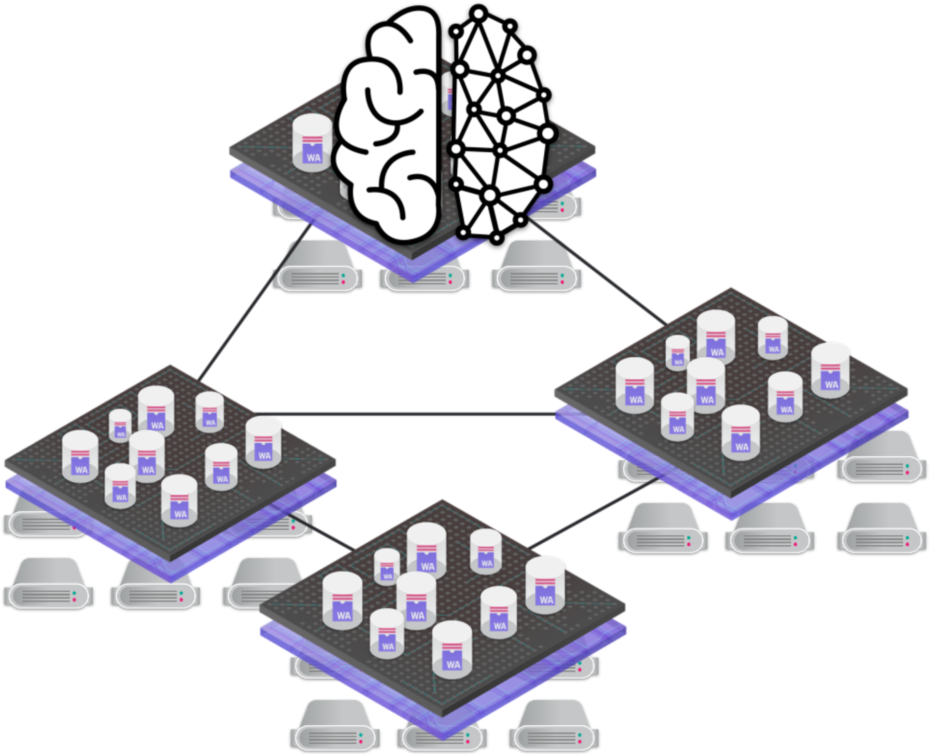
Nodes are partitioned into subnets

Canister smart contracts are assigned to different subnets

One subnet is special: it host the **Network Nervous System (NNS)** canisters which govern the IC

ICP token holders vote on

- Creation of new subnets
- Upgrades to new protocol version
- Replacement of nodes
- …



Comparison* with other Blockchain Systems

Newer comparison by DFINITY

coincodex

Layer-1 Performance Comparison

	Ethereum	Cardano	Solana	Avalanche	Algorand	Internet Computer
Transaction Speed	15-20 TPS	2 TPS	2,000-3,000 TPS	4,500 TPS	20 TPS	11,500 TPS 250,000 QPS
Transaction Finality	14 minutes	10-60 minutes	21-46 seconds	2-3 seconds	4-5 seconds	1 second
Scalability	Not very scalable	Not very scalable	Not very scalable	Not very scalable	More scalability	Indefinite scalability
Node Count	6,000 nodes	3,173 nodes	1,603 nodes	1,243 nodes	1,997 nodes	443 nodes
Storage Costs	$73,000,000 / GB	Inadequate data storage	$1,000,000 / GB	$988,000 / GB	IPFS off-chain storage	$5 / GB
Cloud Service Dependency	70% of nodes run on AWS	Unclear how many are cloud	Most nodes run on cloud	Unclear how many are cloud	Most nodes run on cloud	Independent data centers

https://coincodex.com/article/14198/layer-1-performance-comparing-6-leading-blockchains/

* a bit old and somewhat outdated