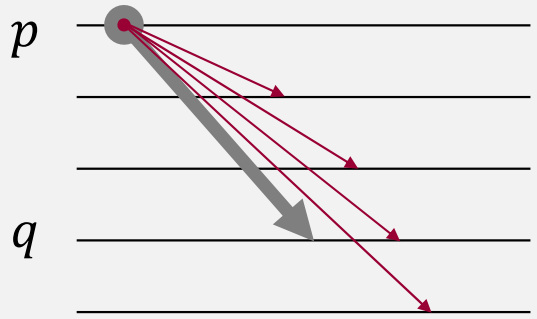
Using Minicasts for Efficient Asynchronous Causal Unicast and Byzantine Tolerance

Laine Rumreich and Dr. Paul Sivilotti

The Ohio State University
College of Engineering

Overview

1. Efficient asynchronous causal unicast
2. Byzantine-tolerant asynchronous causal unicast

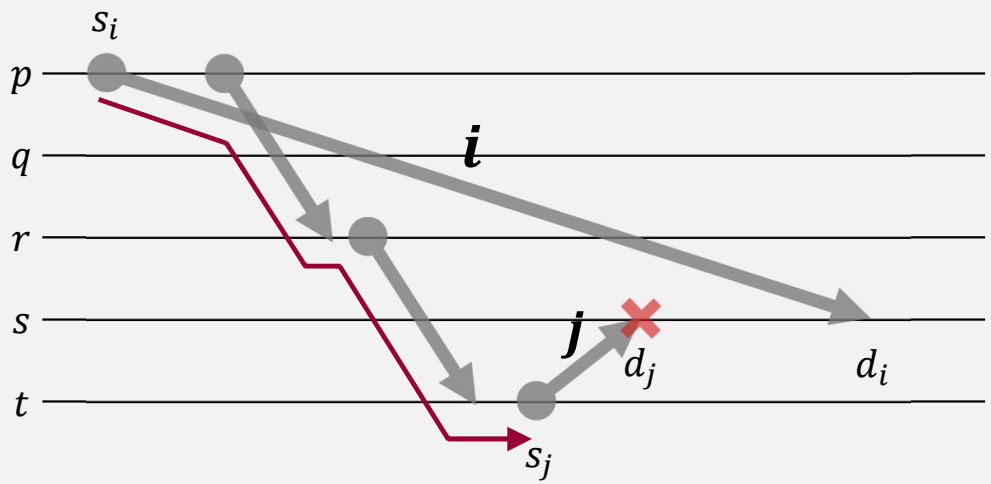


Efficient Asynchronous Causal Unicast

Background

Causal Message Ordering

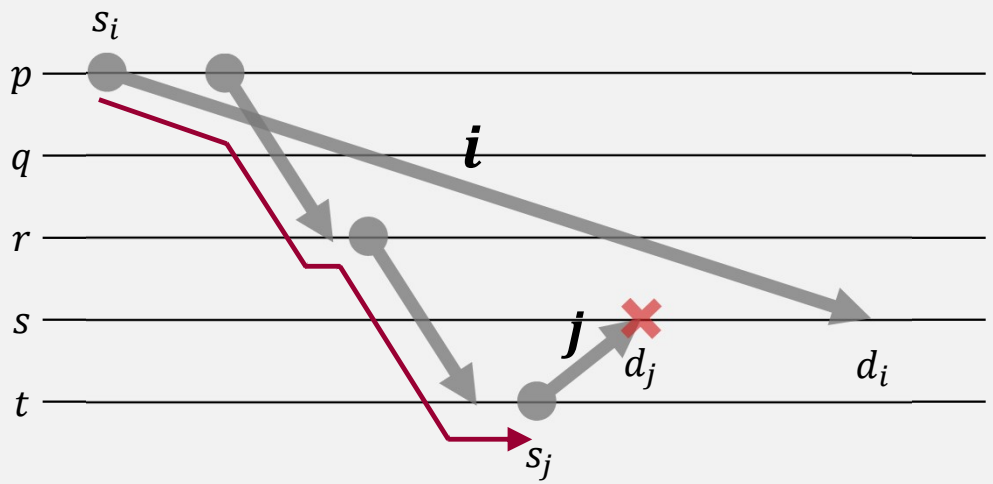
Let i and j be two messages and denote their corresponding sends and deliveries as s_i, s_j, d_i, and d_j. **For messages i and j with the same destination, if $s_i \rightarrow s_j$, then $d_i \rightarrow d_j$.**



Background

Causally Ordered Unicast Algorithms

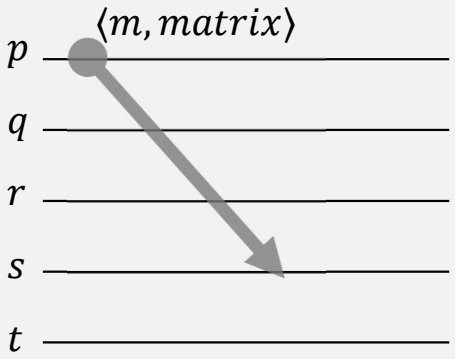
- When messages arrive out of order, they are buffered (i.e. *received*) until they can be safely *delivered*



Background

Causally Ordered Unicast Algorithms

- To enforce causal ordering, messages are augmented with a matrix of the entire history of messages
- This information is propagated to all nodes through other messages



q has sent 2 messages to p

	p	q	r	s
p	0	1	4	1
q	2	0	2	2
r	1	1	0	4
s	1	2	3	0

System Model

Asynchronous System

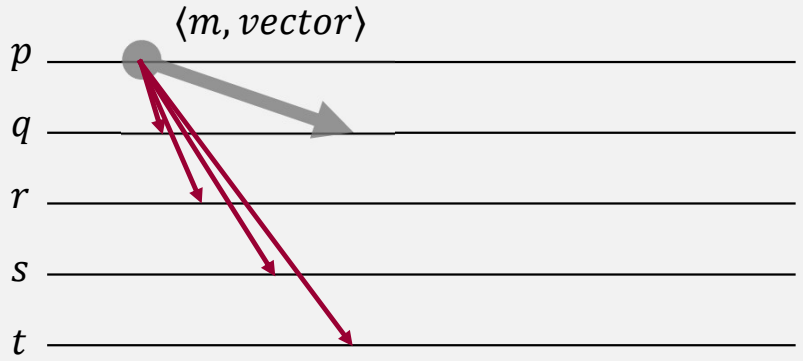
- *No bounds on message delays. i.e., no way to distinguish between a "slow" message and a message that was never sent*

FIFO Channels

- *Assume a set of FIFO channels connecting each process to every other process*

Algorithm Intuition

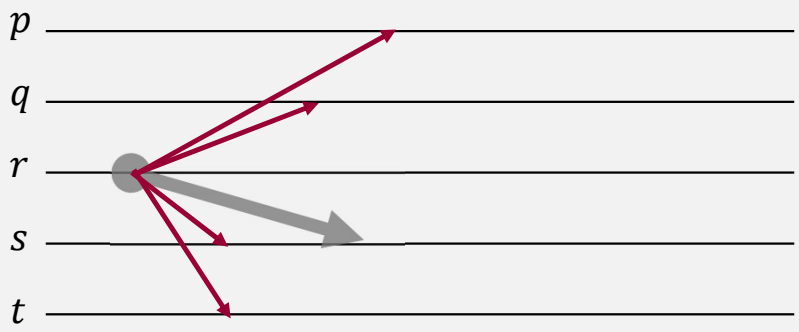
- *Minicast is defined here as a **small** (O(1)) **broadcast message***
  - *i.e. destination and message number*
- *Information is sent directly to every other process, not propagated*
- *Only a vector is required (rather than a matrix) to maintain causal ordering*



Algorithm – Data Structures

deliveredMCs

- *Number of minicasts delivered from every other process*



p.deliveredMCs

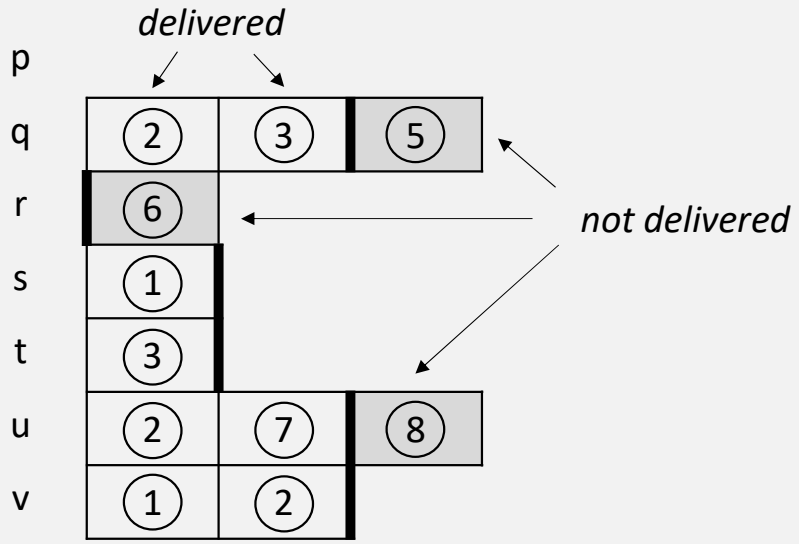
p	4
q	4
r	5
s	2
t	3
u	7
v	3

Algorithm – Data Structures

`receivedUnicasts`

- *Unicast messages that have been received but not yet delivered*

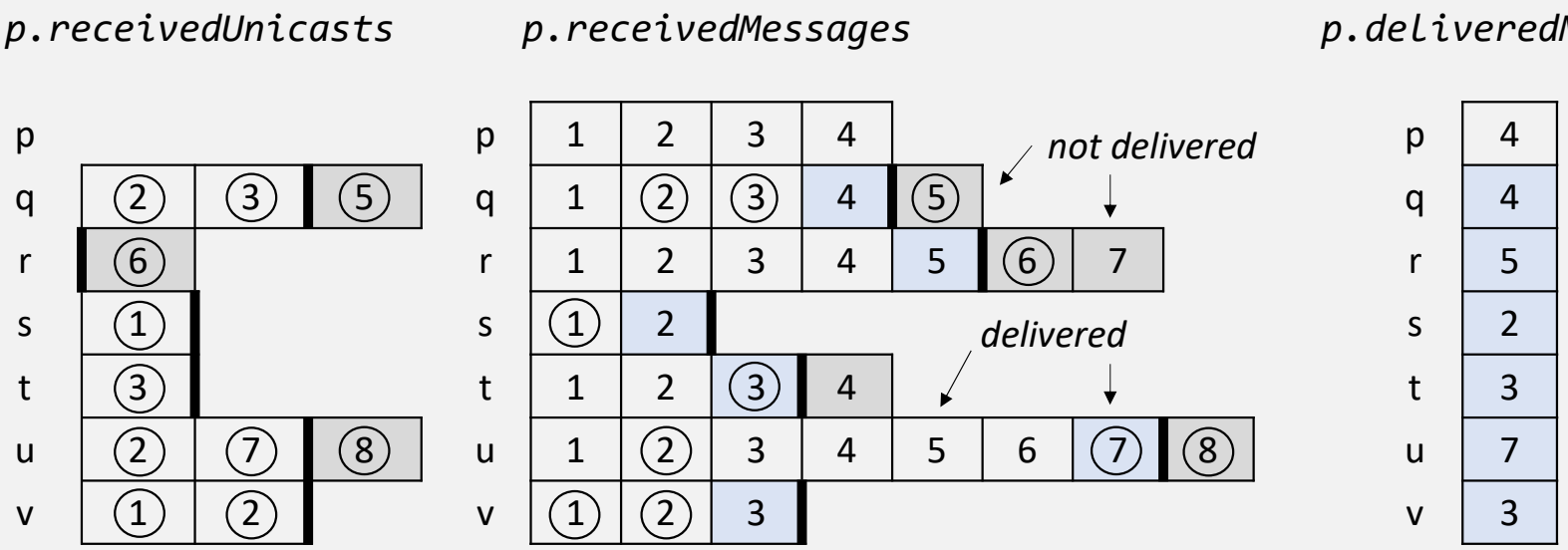
`p.receivedUnicasts`



Algorithm – Data Structures

receivedMessages

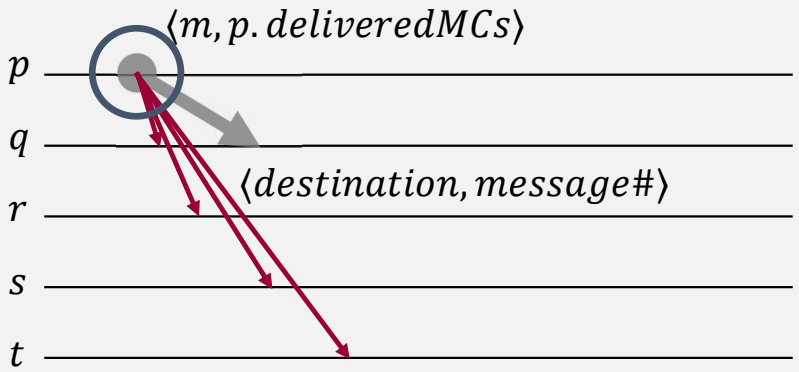
- *Queues of minicast and unicast messages merged together to preserve order*



Algorithm

Message Send

5: send message m to process $q \longrightarrow$
6: **Send**($\langle m, deliveredMCs \rangle$) to q
7: **Broadcast**($\langle q, deliveredMCs[p] \rangle$)
8: $deliveredMCs[p]$ += 1



p.deliveredMCs

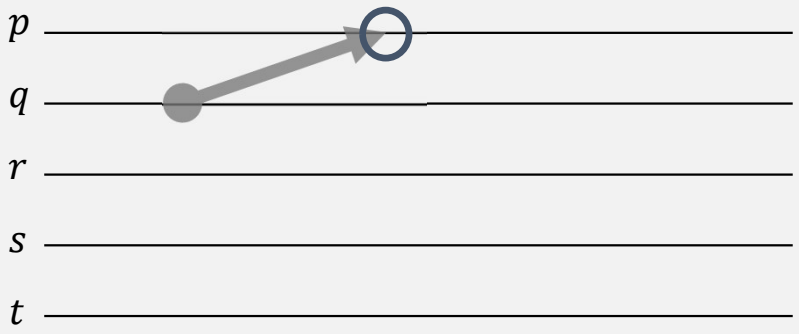
p	5
q	4
r	5
s	2
t	3
u	7
v	3

Algorithm

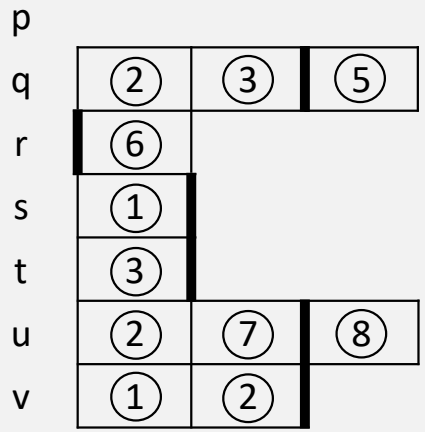
Unicast Arrives

9: $\langle m, MCs \rangle$ arrives from process $q \longrightarrow$

10: $receivedUnicasts[q].enqueue(\langle m, MCs \rangle)$



p.receivedUnicasts

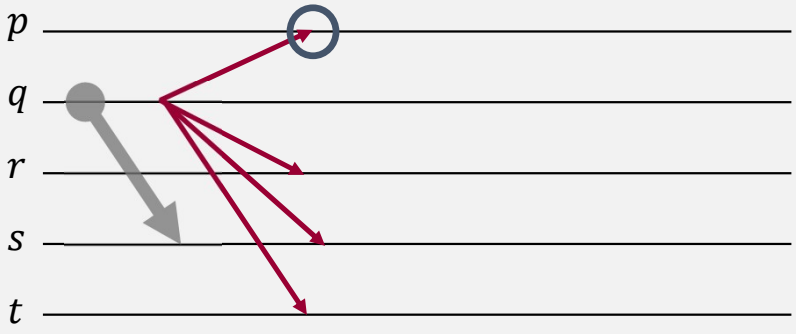


Algorithm

Minicast Arrives

11: $\langle r, msgNum \rangle$ arrives from process $q \longrightarrow$

12: $receivedMessages[q].enqueue(\langle r, msgNum \rangle)$



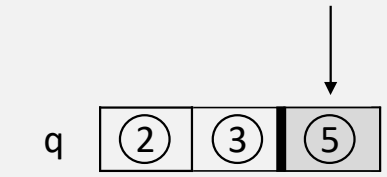
p.receivedMessages

p	1	2	3	4				
q	1	②	③	4	⑤	6		
r	1	2	3	4	5	⑥	7	
s	①	2						
t	1	2	③	4				
u	1	②	3	4	5	6	⑦	⑧
v	①	②	3					

Algorithm

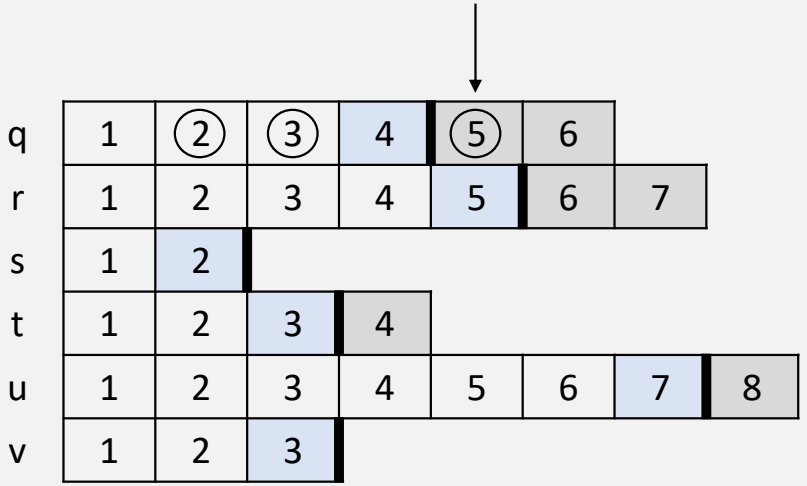
Order Unicast in receivedMessages

13: $receivedMessages[q].front = \langle p, msgNum \rangle$
14: $\wedge\ receivedUnicasts[q] \neq \epsilon \longrightarrow$
15: $next \leftarrow receivedUnicasts[q].dequeue()$
16: **if** $next.MCs[q] = msgNum$ **then**
17: $receivedMessages[q].replaceFront(next)$
18: **end if**



p.receivedUnicasts

p.receivedMessages



Algorithm

Unicast is Delivered

19: $receivedMessages[q].front = \langle m, MCs \rangle$
20: $\wedge\ MCs \leq_q deliveredMCs \longrightarrow$
21: $\quad next \leftarrow receivedMessages[q].dequeue()$
22: $\quad$ **Deliver**$(next)$
23: $\quad deliveredMCs[q] \leftarrow next.MCs[q]$

p.receivedMessages

p	1	2	3	4				
q	1	(2)	(3)	4	(5)	6		
r	1	2	3	4	5	(6)	7	
s	(1)	2						
t	1	2	(3)	4				
u	1	(2)	3	4	5	6	(7)	(8)
v	(1)	(2)	3					

q.deliveredMCs at time of send

p	3
q	5
r	5
s	1
t	3
u	6
v	3

$\leq_q$

p.deliveredMCs

p	4
q	5
r	5
s	2
t	3
u	7
v	3

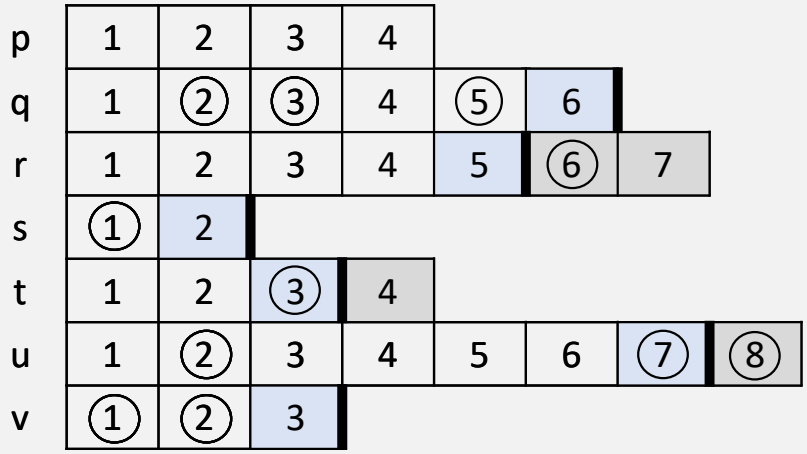
(5) is delivered

Algorithm

Minicast is Delivered

24: $receivedMessages[q].front = \langle r, msgNum \rangle$
25: $\wedge\ r \neq p \longrightarrow$
26: $\quad receivedMessages[q].dequeue()$
27: $\quad deliveredMCs[q] \leftarrow msgNum$

p.receivedMessages



Proof Intuition

1. Safety

Theorem 1. *For sends s_i and s_j with the same destination, if $s_i \rightarrow s_j$, then $d_i \rightarrow d_j$. That is, delivery events are causally ordered.*

Lemma 2. *For events e_i and e_j, if $e_i \rightarrow e_j$, then $deliveredMCs(e_i) < deliveredMCs(e_j)$*

Proof Intuition

2. Liveness

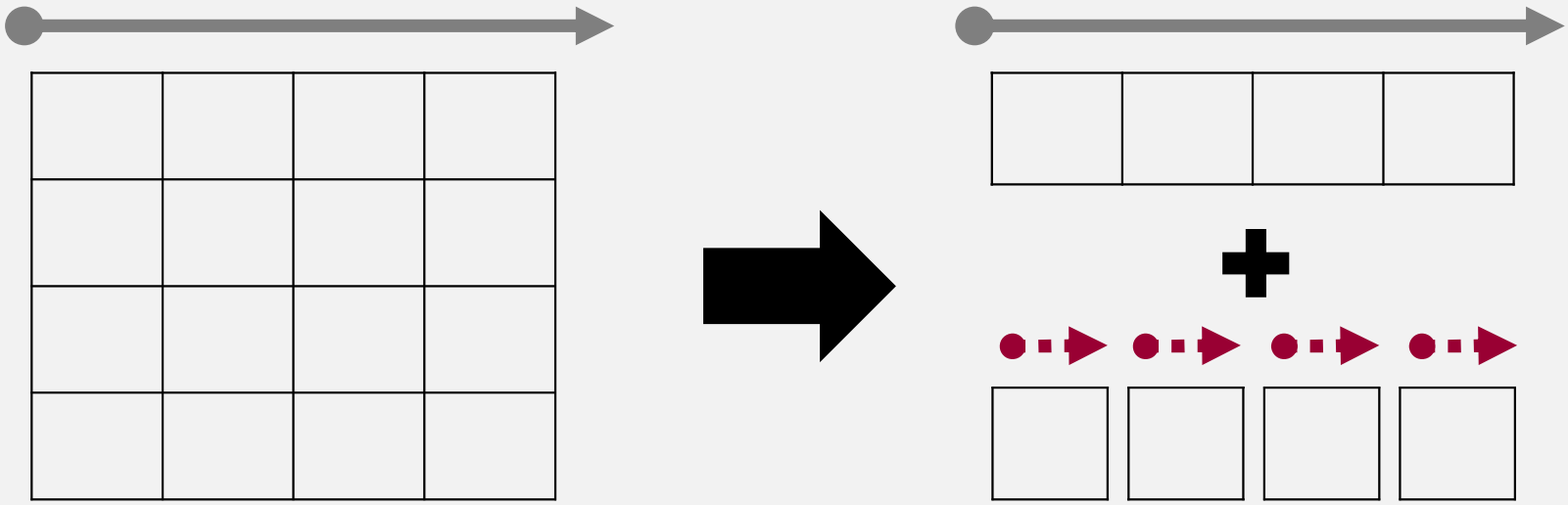
Theorem 2. *Every message must eventually be delivered.*

There must exist at least one global minimal message m such that all sends that happened before s_m have been delivered. We show that once this message is received, it is eventually delivered.

We use induction to show that every message must eventually be the minimal message and thus must eventually be delivered.

Results

- Asynchronous causal unicast using only O(n) space in overhead compared to O(n^2) previously

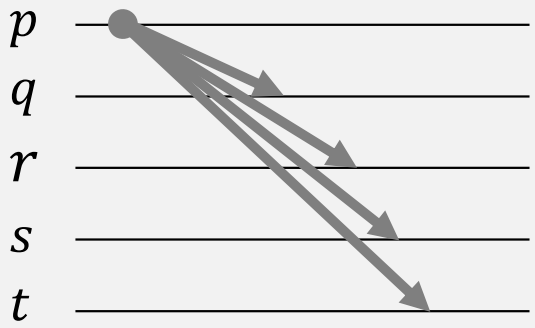


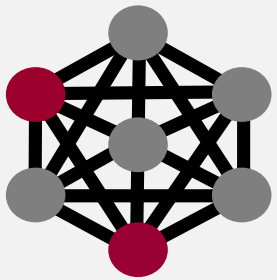
Byzantine Tolerant Asynchronous Causal Unicast

Background

Byzantine Reliable Broadcast

- **p is correct:** all *correct* processes accept and agree on the value of the message
- **p is faulty:** either all correct processes accept and agree on the same value of the message or none of them accept the message

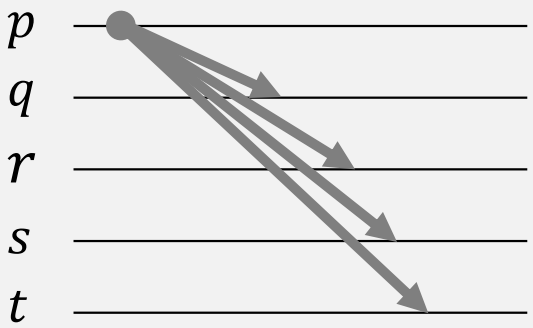


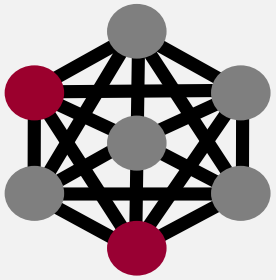


Background

Bracha's Byzantine Reliable Broadcast

- Send *initial, echo,* and *ready* messages
- When a process receives enough *ready* messages, it accepts the message
- All other correct processes are bound to accept the message, regardless of whether they received an *initial* message

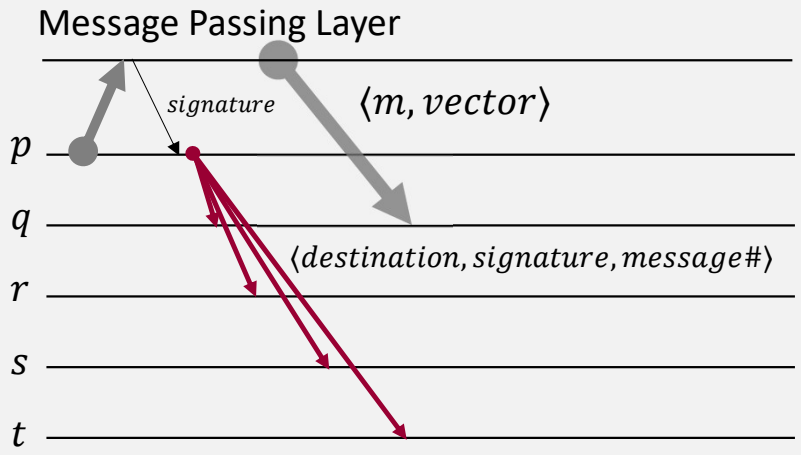




Algorithm

Message Send

1: send message m to process $q \longrightarrow$
2: $signature \leftarrow$ **Send_MPLayer**$(q, \langle m, deliveredMCs \rangle)$
3: **BCCH_Broadcast**$(\langle q, signature, deliveredMCs[p] \rangle)$
4: $deliveredMCs[p]$ += 1



p.deliveredMCs

p	4
q	4
r	5
s	2
t	3
u	7
v	3

Arrival of a minicast means a unicast message must have been sent because of the cryptographic signature

Algorithm

Minicast Arrives

5: $\langle r, signature, msgNum \rangle$ arrives from process q
6: $\wedge$ **Verify**$(q, r, signature) \longrightarrow$
7: $receivedMessages[q].enqueue(\langle r, msgNum \rangle)$

Minicasts are verified based on a cryptographic signature, so only valid minicasts are accepted

Byzantine Tolerant Proof Intuition

1. A byzantine node could send a minicast without a unicast

   Arrival of minicast means a unicast message must have been sent because of the cryptographic signature

2. A byzantine node could send a unicast without a minicast

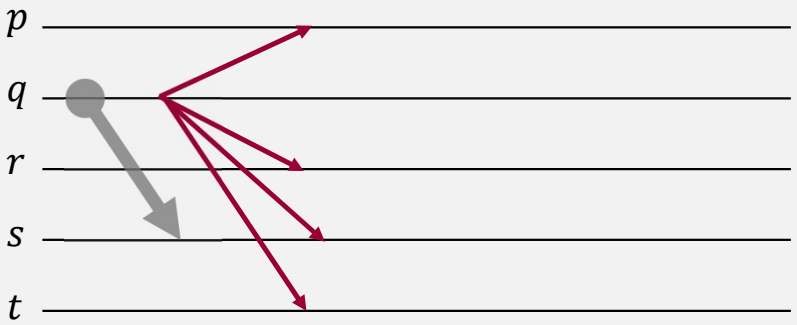
   The unicast is thrown away once another message arrives. No other nodes received a minicast, so they are not impacted.

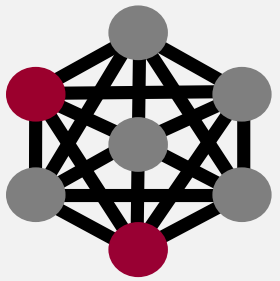
3. The minicast is not sent to all processes

   BCCH guarantees every process accepts/rejects the message and agrees on the content

Results

- First algorithm to implement byzantine-tolerant asynchronous causal unicast
- Requires O($nlogn$) overhead space
- Requires additional latency





Questions

Laine Rumreich *rumreich.1@osu.edu*
Paul Sivilotti *paolo@cse.ohio-state.edu*