Distributed Computing

Slides last updated: 17.11.2024

Computer Systems
Distributed Systems

Exercise Session 9 · HS 2024

Distributed Computing

Agenda

- Assignment Review
- Lecture Recap
    - Byzantine Agreement
        - King’s Algorithm
        - Ben-Or’s Algorithm
    - (Efficient-) Reliable Broadcast
- Quiz
- Assignment Preview

Distributed Computing

Assignment Review

1.1 An Asynchronous Riddle

A hangman summons his 100 prisoners, announcing that they may meet to plan a strategy, but will then be put in isolated cells, with no communication. He explains that he has set up a switch room that contains a single switch. Also, the switch is not connected to anything, but a prisoner entering the room may see whether the switch is on or off (because the switch is up or down).

Every once in a while the hangman will let one arbitrary prisoner into the switch room. The prisoner may throw the switch (on to off, or vice versa), or leave the switch unchanged. Nobody but the prisoners will ever enter the switch room. The hangman promises to let any prisoner enter the room from time to time, arbitrarily often. That is, eventually, each prisoner has been in the room at least once, twice, a thousand times or any number you want.

At any time, any prisoner may declare: ”We have all visited the switch room at least once.” If the claim is correct, all prisoners will be released, otherwise, the hangman will execute his job (on all the prisoners). Which strategy would you choose...

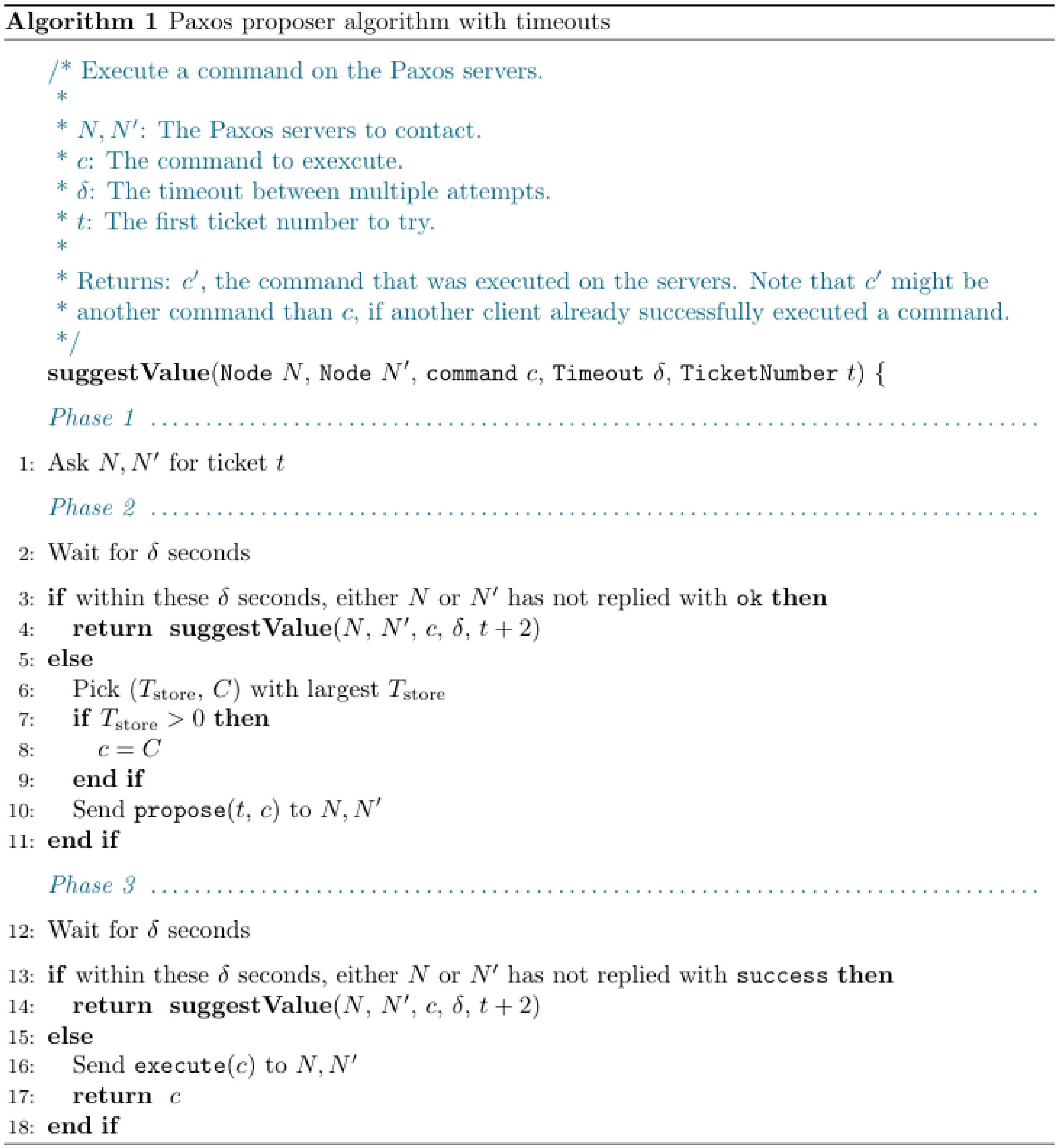
a) ...if the hangman tells them, that the switch is off at the beginning?

b) ...if they don’t know anything about the initial state of the switch?

1.2 Paxos

You decide to use Paxos for a system with 3 servers (acceptors), which we call N1, N2, N3. There are two clients (proposers) A and B. The implementation of the acceptors is exactly as defined in the script, see Algorithm 15.13. We extended the code of the proposers, such that they now use explicit timeouts. The algorithm is described below, note in particular Lines 2-4 and 12-14.

a) Assume that two users each try to execute a command. One user calls $suggestValue(N1, N2, a, 1, 1)$ on A at time T_0, and a second user calls $suggestValue(N2, N3, b, 2, 2)$ on B at time $T_0 + 0.5\text{sec}$. Draw a timeline containing all transmitted messages! We assume that processing times on nodes can be neglected (i.e. is zero), and that all messages arrive within less than 0.5sec .



Algorithm 1 Paxos proposer algorithm with timeouts

```
    /* Execute a command on the Paxos servers.
     *
     * $N, N'$: The Paxos servers to contact.
     * $c$: The command to exexcute.
     * $\delta$: The timeout between multiple attempts.
     * $t$: The first ticket number to try.
     *
     * Returns: $c'$, the command that was executed on the servers. Note that $c'$ might be
     * another command than $c$, if another client already successfully executed a command.
     */
    suggestValue(Node $N$, Node $N'$, command $c$, Timeout $\delta$, TicketNumber $t$) {
    Phase 1 ...........................................................
 1: Ask $N, N'$ for ticket $t$
    Phase 2 ...........................................................
 2: Wait for $\delta$ seconds
 3: if within these $\delta$ seconds, either $N$ or $N'$ has not replied with ok then
 4:   return suggestValue($N$, $N'$, $c$, $\delta$, $t + 2$)
 5: else
 6:   Pick $(T_{\text{store}}, C)$ with largest $T_{\text{store}}$
 7:   if $T_{\text{store}} > 0$ then
 8:     $c = C$
 9:   end if
10:   Send propose($t$, $c$) to $N, N'$
11: end if
    Phase 3 ...........................................................
12: Wait for $\delta$ seconds
13: if within these $\delta$ seconds, either $N$ or $N'$ has not replied with success then
14:   return suggestValue($N$, $N'$, $c$, $\delta$, $t + 2$)
15: else
16:   Send execute($c$) to $N, N'$
17:   return $c$
18: end if
```

Distributed Computing

1.2 Paxos

You decide to use Paxos for a system with 3 servers (acceptors), which we call N1, N2, N3. There are two clients (proposers) A and B. The implementation of the acceptors is exactly as defined in the script, see Algorithm 15.13. We extended the code of the proposers, such that they now use explicit timeouts. The algorithm is described below, note in particular Lines 2-4 and 12-14.

b) In a) we chose artificial initial ticket numbers and timeout values, and we saw that Paxos terminates rather quickly. Let us look at another selection of these values: The two clients start with initial ticket numbers $t_A = t_0$ and $t_B = t_0 + 1$ for some value t_0, and both clients have the same timeout $\delta_A = \delta_B$. Assume that both clients start at T_0. What will happen?

Algorithm 1 Paxos proposer algorithm with timeouts

```
    /* Execute a command on the Paxos servers.
     *
     * $N, N'$: The Paxos servers to contact.
     * $c$: The command to exexcute.
     * $\delta$: The timeout between multiple attempts.
     * $t$: The first ticket number to try.
     *
     * Returns: $c'$, the command that was executed on the servers. Note that $c'$ might be
     * another command than $c$, if another client already successfully executed a command.
     */
    suggestValue(Node $N$, Node $N'$, command $c$, Timeout $\delta$, TicketNumber $t$) {

    Phase 1 ..................................................
 1: Ask $N, N'$ for ticket $t$
    Phase 2 ..................................................
 2: Wait for $\delta$ seconds
 3: if within these $\delta$ seconds, either $N$ or $N'$ has not replied with ok then
 4:    return suggestValue($N$, $N'$, $c$, $\delta$, $t+2$)
 5: else
 6:    Pick $(T_{store}, C)$ with largest $T_{store}$
 7:    if $T_{store} > 0$ then
 8:       $c = C$
 9:    end if
10:    Send propose($t$, $c$) to $N, N'$
11: end if
    Phase 3 ..................................................
12: Wait for $\delta$ seconds
13: if within these $\delta$ seconds, either $N$ or $N'$ has not replied with success then
14:    return suggestValue($N$, $N'$, $c$, $\delta$, $t+2$)
15: else
16:    Send execute($c$) to $N, N'$
17:    return $c$
18: end if
```

1.3 Improving Paxos

We are not happy with the runtime of the Paxos algorithm of Exercise 1.2. Hence, we study some approaches which might improve the runtime.

The point in time when clients start sending messages cannot be controlled, since this will be determined by the application that uses Paxos. It might help to use different initial ticket numbers. However, if a client with a very high ticket number crashes early, all other clients need to iterate through all ticket numbers. This problem can easily be fixed: Every time a client sends an ask(t) message with $t \leq T_{\max}$, the server can reply with an explicit $nack(T_{\max})$ in Phase 1, instead of just ignoring the $ask(t)$ message.

a) Assume you added the explicit nack message. Do different initial ticket numbers solve runtime issues of Paxos, or can you think of a scenario which is still slow?

b) Instead of changing the parameters, we add a waiting time between sending two consecutive ask messages. Sketch an idea of how you could improve the expected runtime in a scenario where multiple clients are trying to execute a command by manipulating this waiting time!
Extra challenge: Try not to slow down an individual client if it is alone!

2.1 Consensus with Edge Failures

In the lecture we only discussed node failures, but we always assumed that edges (links) never fail. Let us now study the opposite case: Assume that all nodes work correctly, but up to f edges may fail.

Analogously to node failures, edges may fail at any point during the execution. We say that a failed edge does not forward any message anymore, and remains failed until the algorithm terminates. Assume that an edge always simultaneously fails completely, i.e., no message can be exchanged over that edge anymore in either direction.

We assume that the network is initially fully connected, i.e., there is an edge between every pair of nodes. Our goal is to solve consensus in such a way, that all nodes know the decision.

a) What is the smallest f such that consensus might become impossible? (Which edges fail in the worst-case)

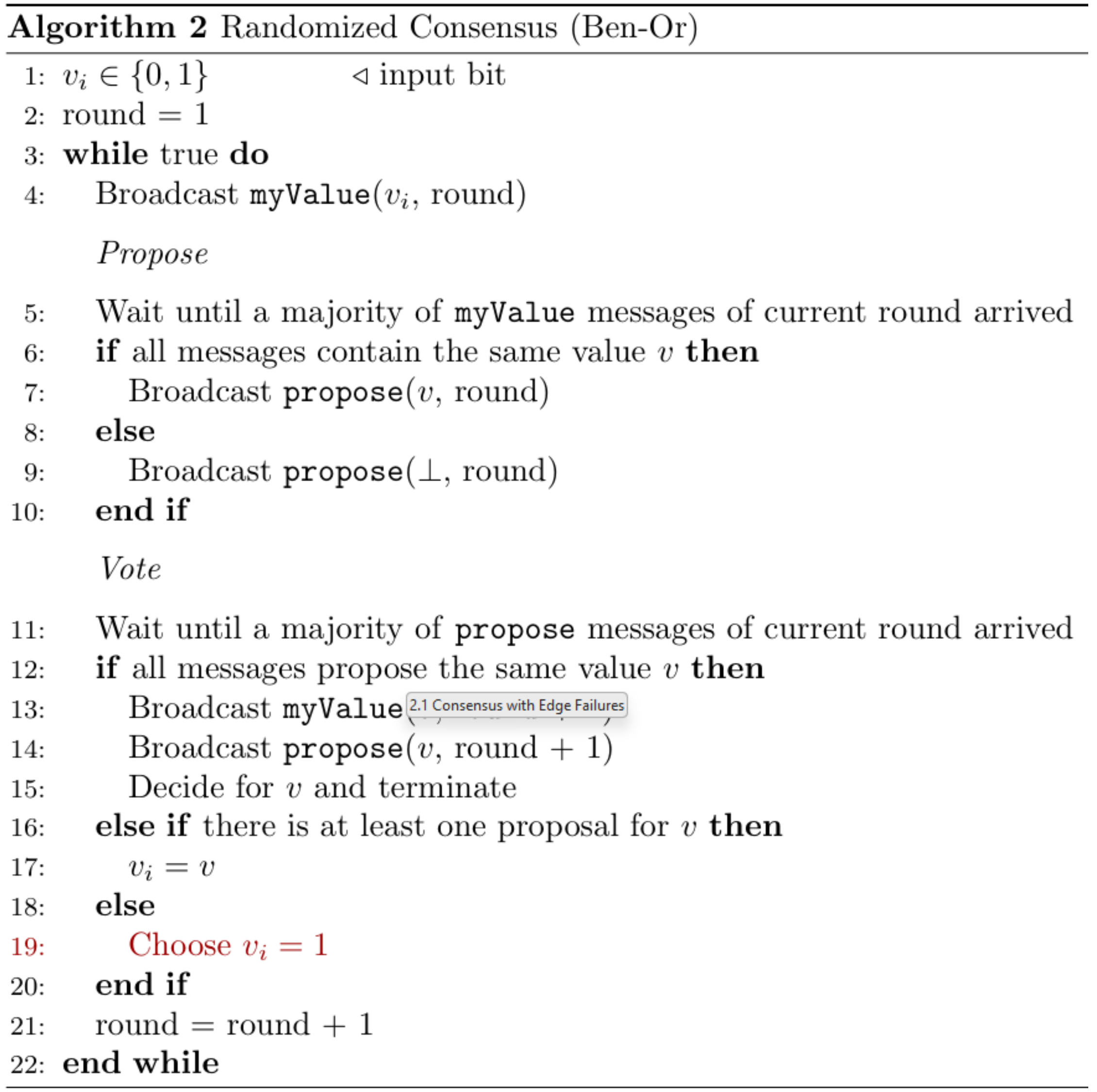
b) What is the largest f such that consensus might still be possible? (Which edges fail in the best-case)

c) Assume that you have a setup which guarantees you that the nodes always remain con nected, but possibly many edges might fail. A very simple algorithm for consensus is the following: Every node learns the initial value of all nodes, and then decides locally. How much time might this algorithm require? Assume that a message takes at most 1 time unit from one node to a direct neighbor.

2.2 Deterministic Random Consensus?!

Algorithm 16.15 from the lecture notes solves consensus in the asynchronous time model. It seems that this algorithm would be faster, if nodes picked a value deterministically instead of randomly in Line 23. However, a remark in the lecture notes claims that such a deterministic selection of a value will not work. We did it anyway! (The only change is on Line 23).

Show that this algorithm does not solve consensus! Start by choosing initial values for all nodes and show that the algorithm below does not terminate.



Algorithm 2 Randomized Consensus (Ben-Or)

```
 1: $v_i \in \{0, 1\}$                ◁ input bit
 2: round = 1
 3: while true do
 4:    Broadcast myValue($v_i$, round)
       Propose
 5:    Wait until a majority of myValue messages of current round arrived
 6:    if all messages contain the same value $v$ then
 7:       Broadcast propose($v$, round)
 8:    else
 9:       Broadcast propose($\perp$, round)
10:    end if
       Vote
11:    Wait until a majority of propose messages of current round arrived
12:    if all messages propose the same value $v$ then
13:       Broadcast myValue 2.1 Consensus with Edge Failures
14:       Broadcast propose($v$, round + 1)
15:       Decide for $v$ and terminate
16:    else if there is at least one proposal for $v$ then
17:       $v_i = v$
18:    else
19:       Choose $v_i = 1$
20:    end if
21:    round = round + 1
22: end while
```

2.3 Consensus with Bandwidth Limitations

Consensus with no failures, a fully connected network and unlimited bandwidth is trivial: First, every node sends its value to all other nodes. Second, every node waits for all values, and then decides.

So far, we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 through n.

We assume that all messages are transmitted reliably and arrive exactly after one time unit. The bandwidth limitation is as follows: Assume that every node can only send one message (containing one value) to one neighbor per time unit. For example, at time 0, u1 can send a message to u2, at time 1 a message to u3, and so on. However, u1 cannot send a message to both u2 and u3 at the same time! Also, a node cannot send multiple values in the same message.

a) Develop an algorithm that solves consensus in this scenario. Optimize your algorithm for runtime!

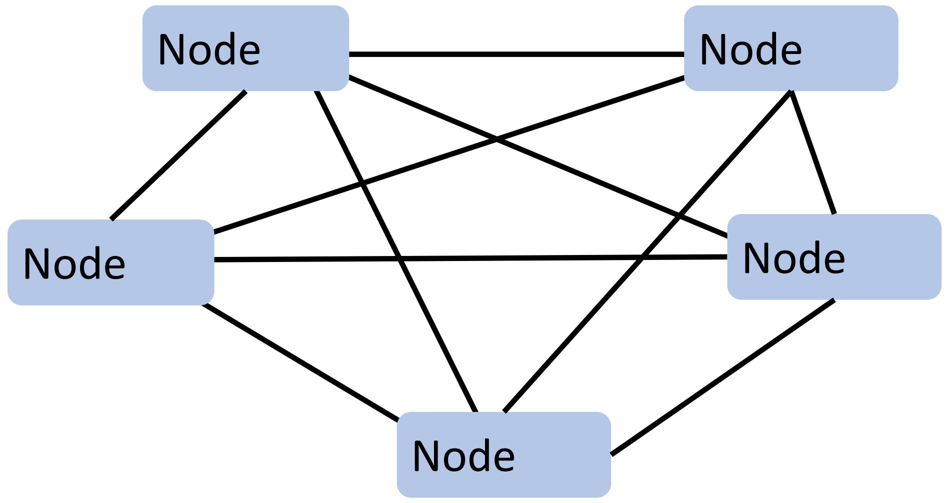
b) What is the runtime of your algorithm?

c) Assume that you not only need to solve consensus, but the more challenging task that every node must learn the input values of all nodes. Show that this problem requires at least $n - 1$ time units!

Distributed Computing

Byzantine Agreement

Set-Up



Node: single actor in a distributed system

Previous Challenges in Consensus

- Messages can get lost
- Nodes may crash
- Messages can have various delay

New Challenge in Byzantine Agreement

- *Byzantine nodes = Nodes can have arbitrary behaviour*

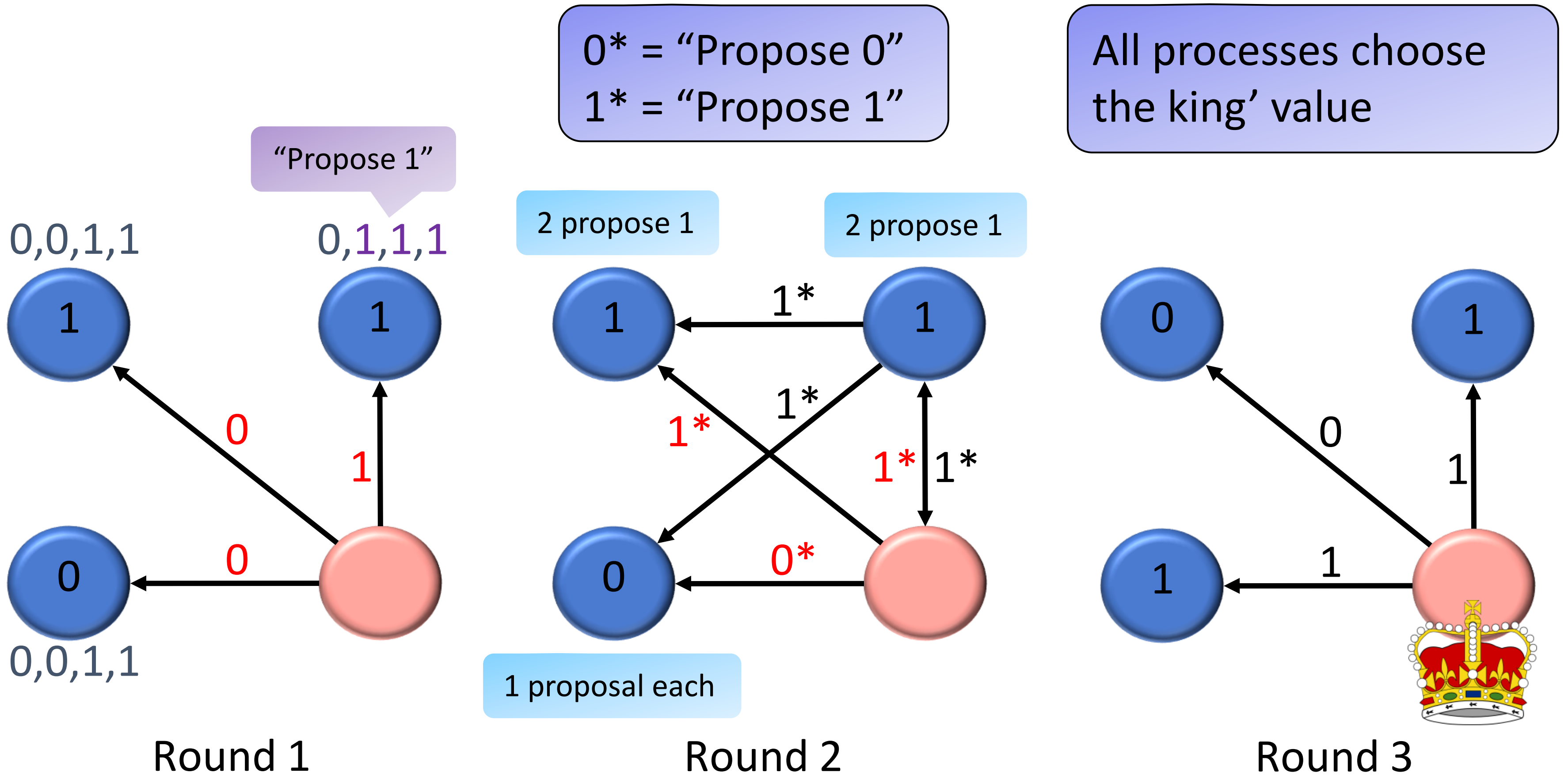
Byzantine Agreement

We want:

1. Agreement:
   All (correct) nodes decide on the same value.
2. Termination:
   All (correct) nodes terminate.
3. All-same Validity:
   If all correct nodes start with the same value v, the decision value must be v.

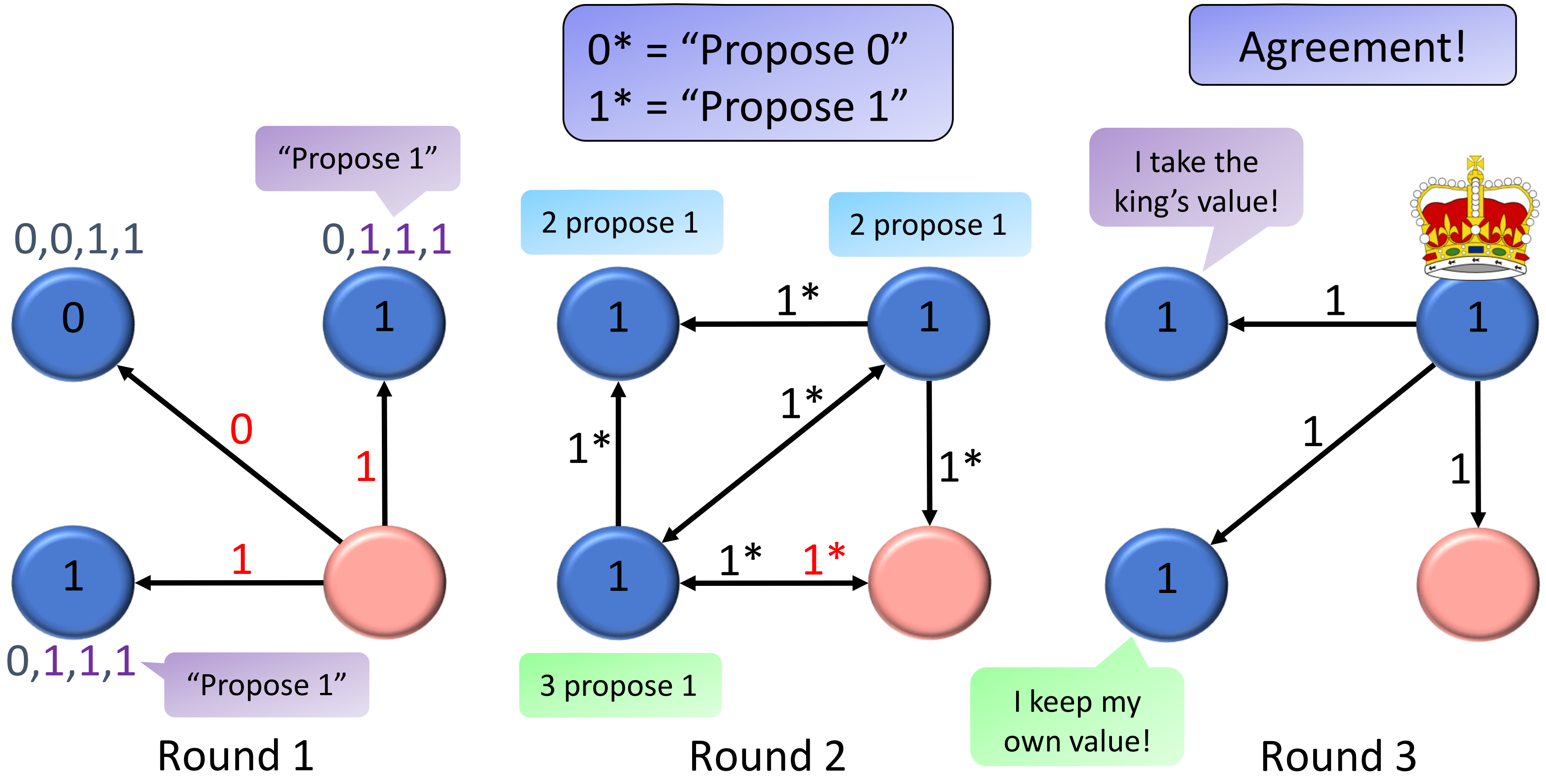
(Synchronous) King's Algorithm

- Example: *n* = 4, *f*=1
- Phase 1:



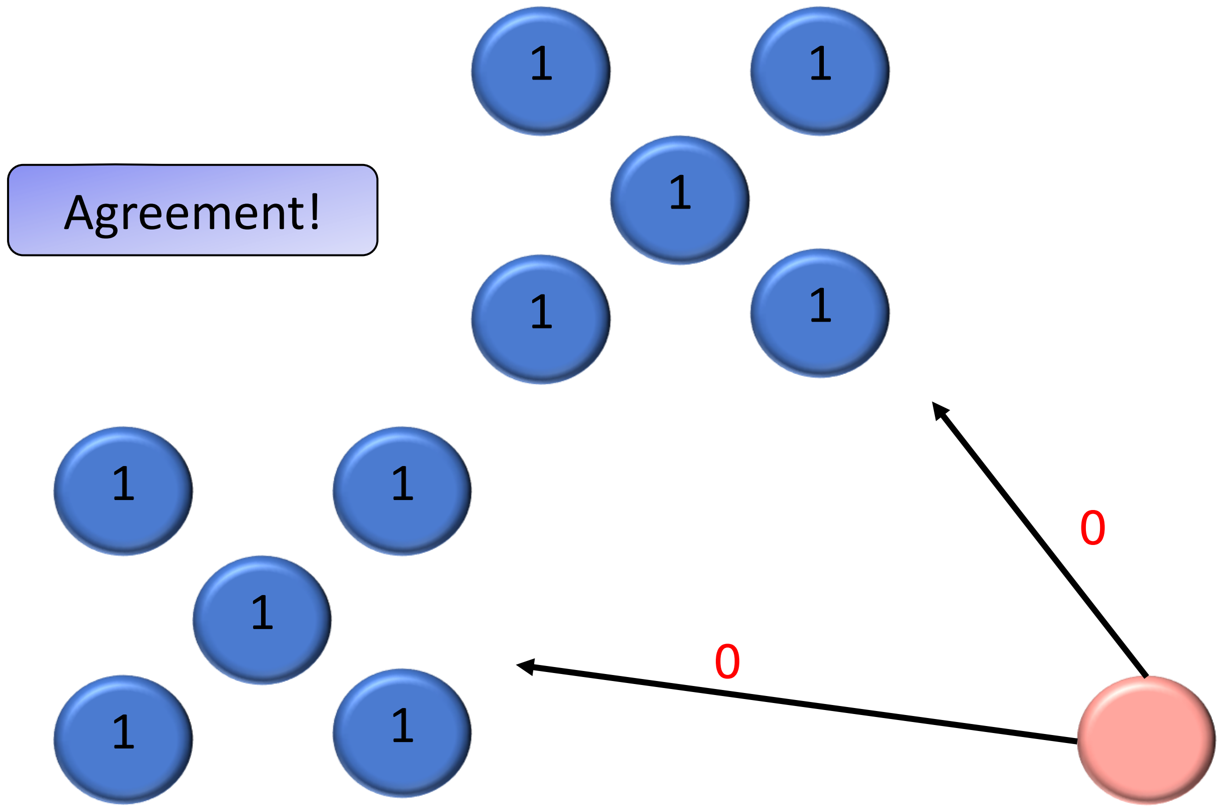
(Synchronous) King's Algorithm

- Example: $n = 4$, f=1
- Phase 2:



(Asynchronous) Ben-Or's Algorithm

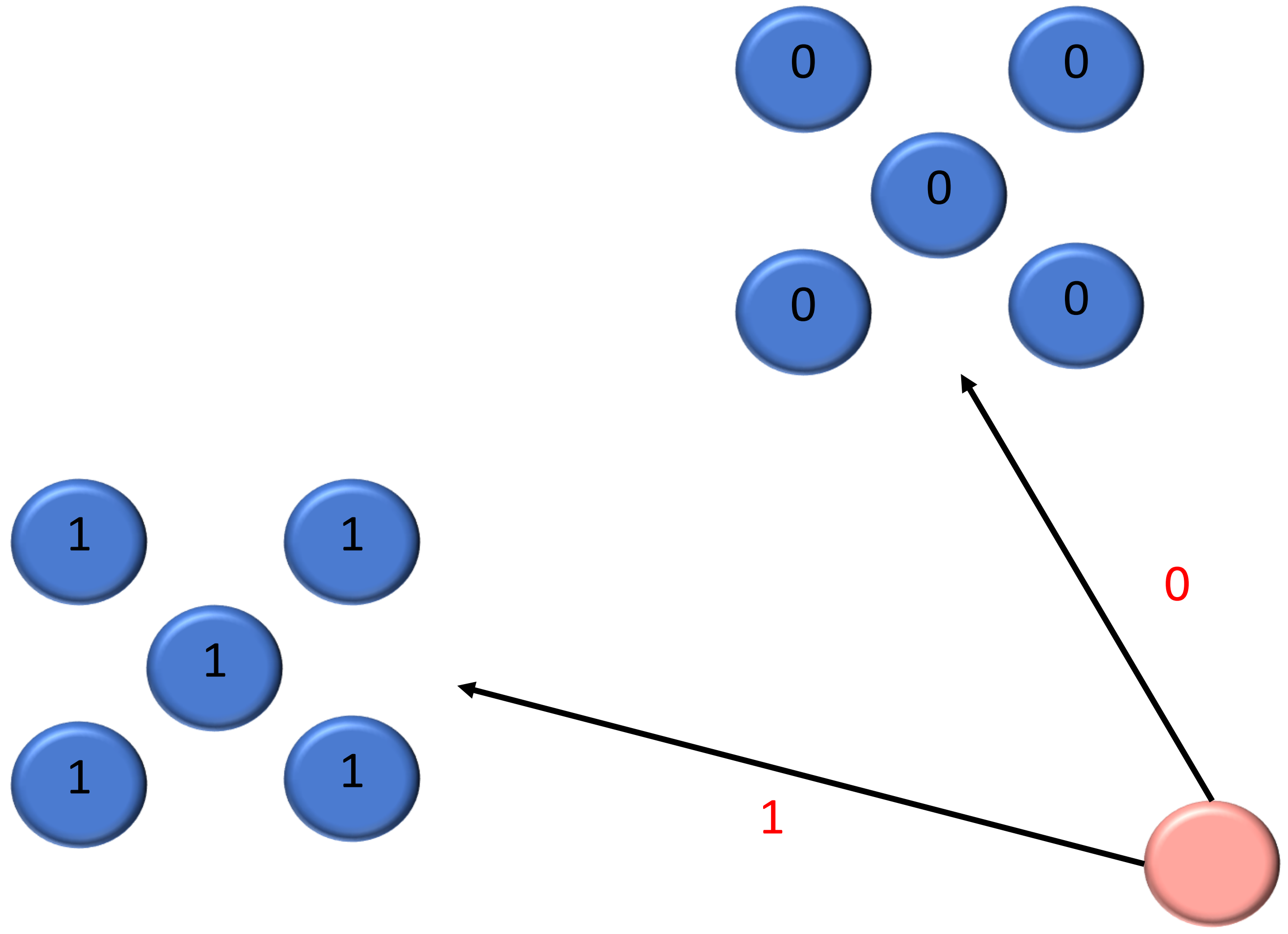
- Example: $n = 11, f=1$
- Byzantine node has no power



ETH zürich Systems@ETH zürich Distributed Computing

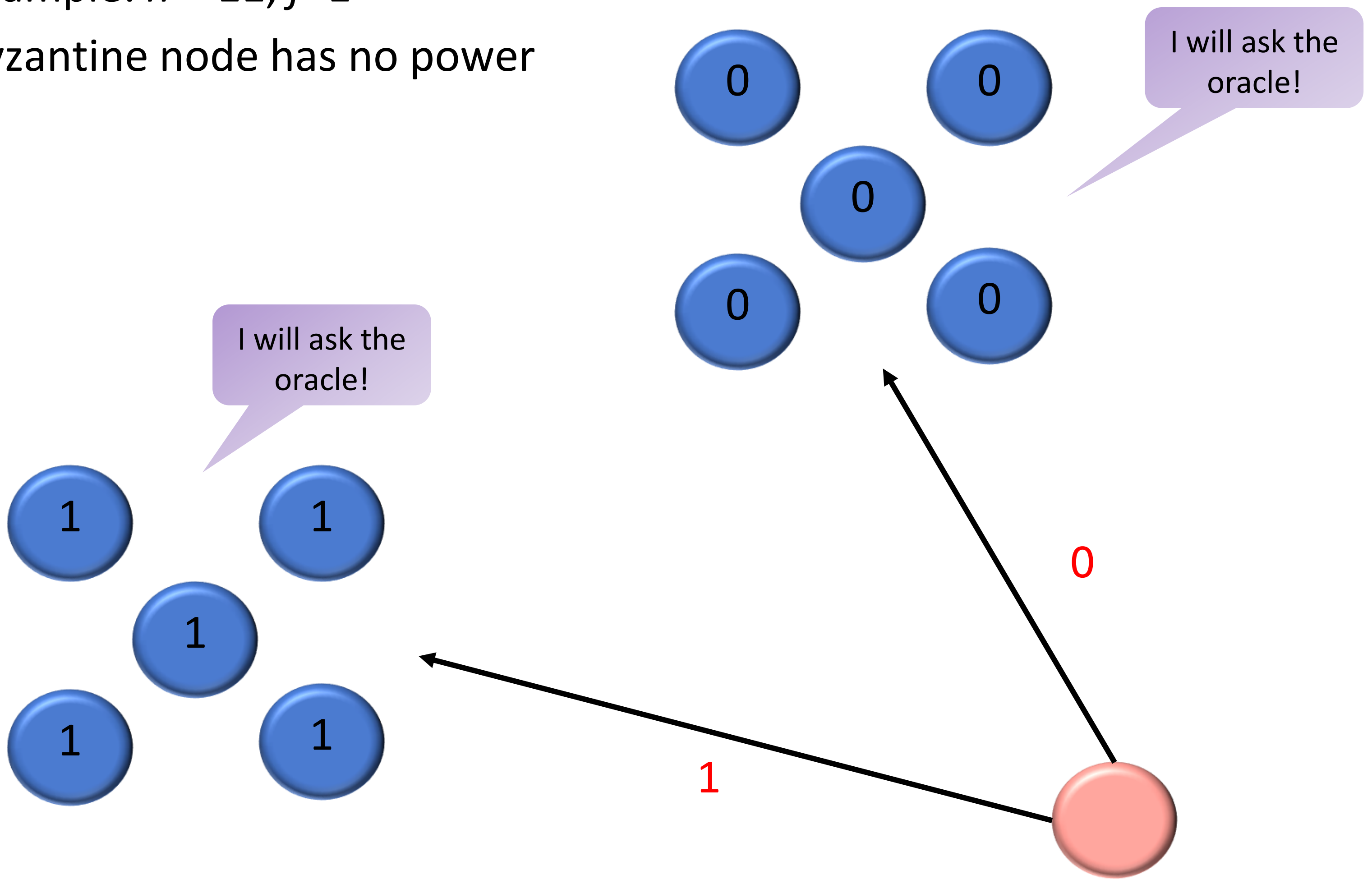
(Asynchronous) Ben-Or's Algorithm

- Example: n = 11, f=1



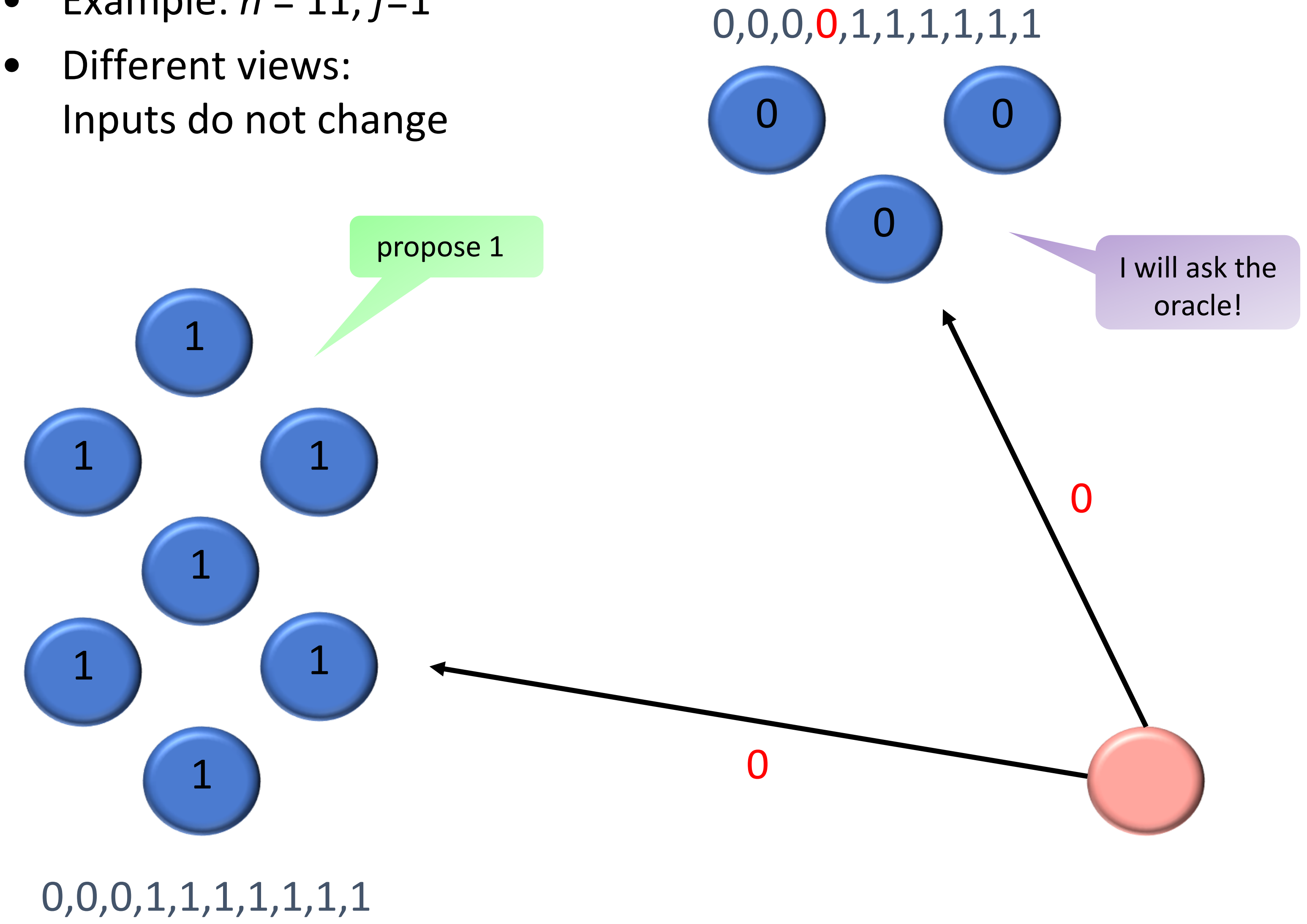
(Asynchronous) Ben-Or’s Algorithm

- Example: n = 11, f=1
- Byzantine node has no power



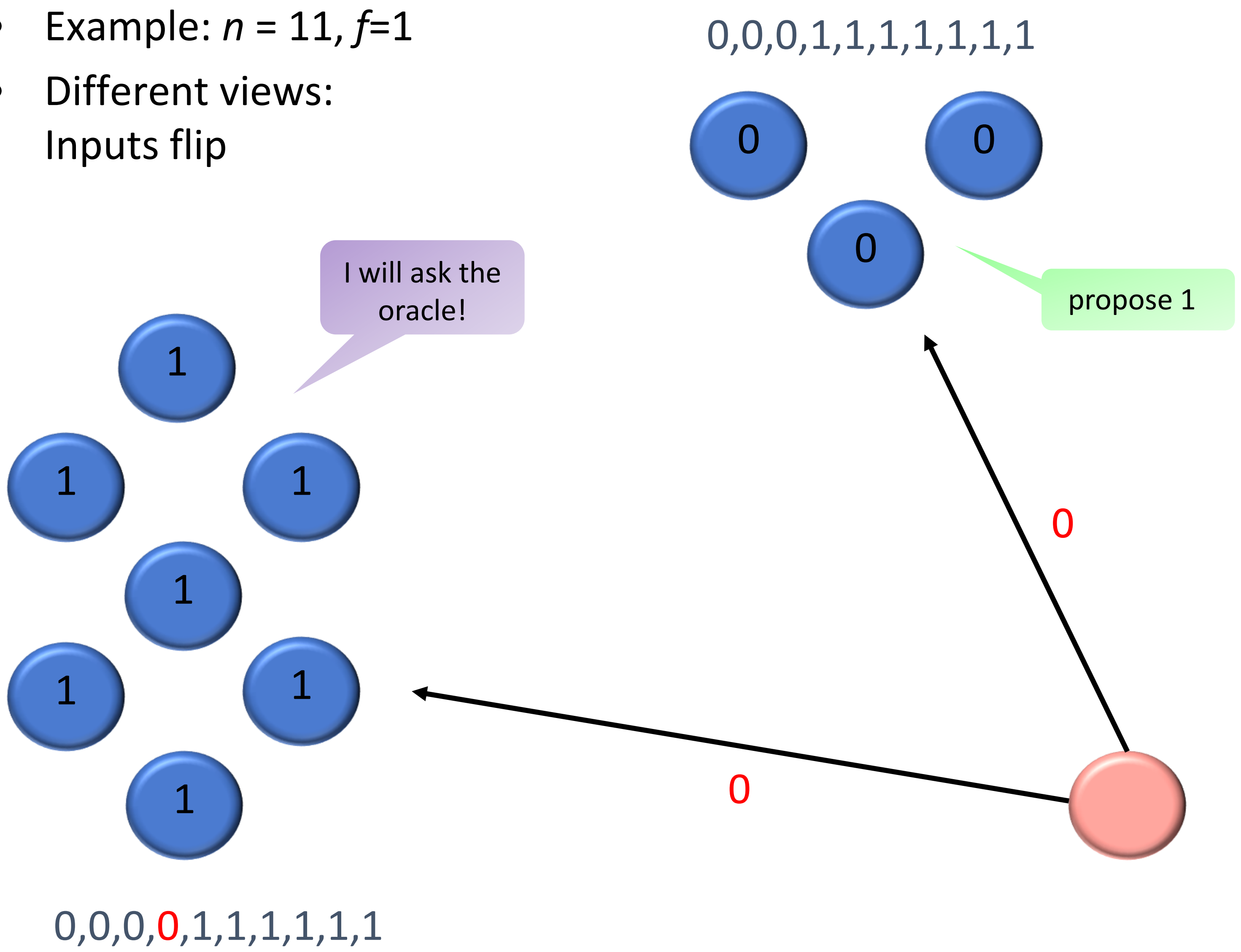
(Asynchronous) Ben-Or's Algorithm

- Example: $n = 11$, $f=1$
- Different views:
  Inputs do not change



(Asynchronous) Ben-Or's Algorithm

- Example: *n* = 11, *f*=1
- Different views:
  Inputs flip



ETH zürich Systems@ETH zürich Distributed Computing

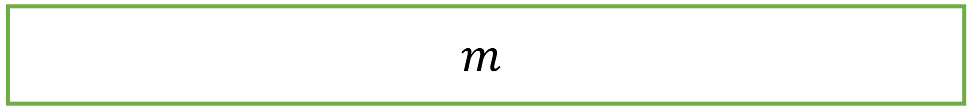
Broadcast

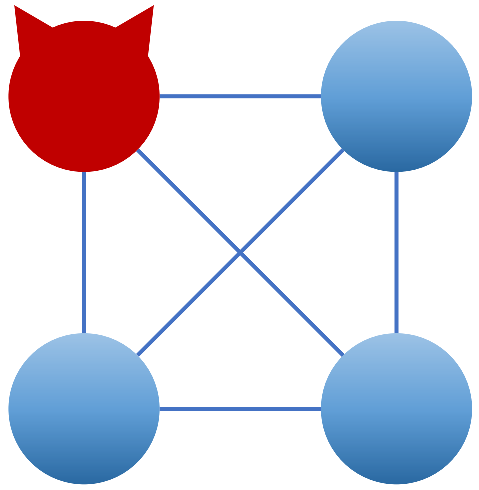
Broadcast

- **Validity:** If a correct node broadcasts a message, it will eventually be accepted by every other correct node.
- **Weak Integrity:** If a correct node v has not broadcast a message, no message of v will be accepted by any other correct node.
- **Integrity:** Every correct node delivers at most one message. The delivered message must have been broadcast by a node.
- **Totality:** If a correct node accepts a message m, m will eventually be accepted by every other correct node.
- **Agreement:** If correct nodes v and v' accept messages m and m', respectively, then $m = m'$.

Efficient Reliable Broadcast

- Use fact that usually, the size of the message is way larger than the number of nodes in the system.
- Split message m into n fragments $f_1, f_2, \ldots, f_n$ using (n, k) erasure code.

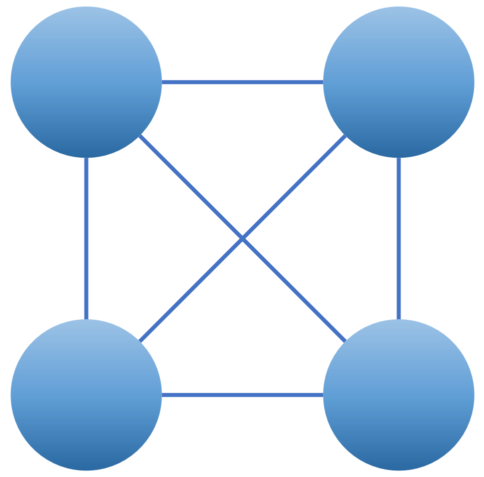




Efficient Reliable Broadcast

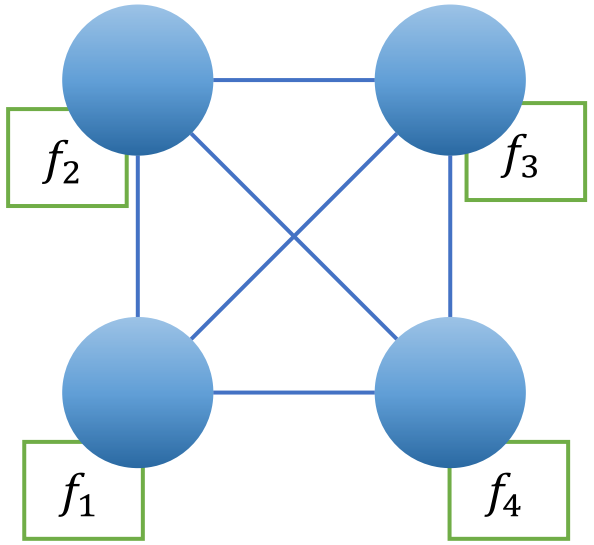
- Use fact that usually, the size of the message is way larger than the number of nodes in the system.
- Split message m into n fragments $f_1, f_2, \dots, f_n$ using (n, k) erasure code.
- Assume optimal erasure code, any k fragments allow to restore the message m.
- What values do we choose for n and k, and why?

$\boxed{f_1}$ $\boxed{f_2}$ $\boxed{f_3}$ $\boxed{f_4}$



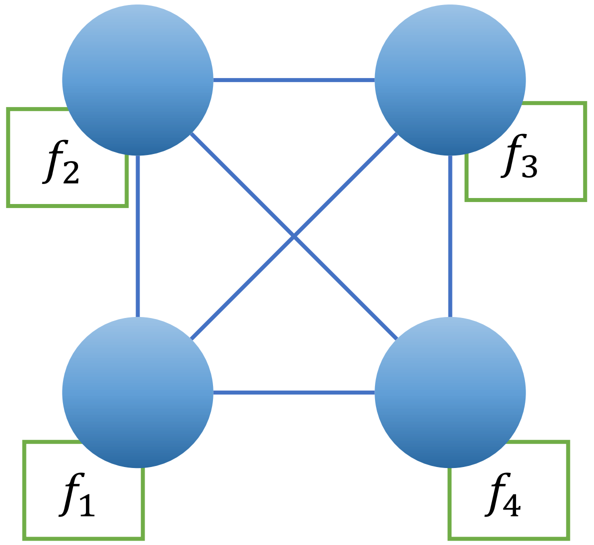
Efficient Reliable Broadcast

- Use fact that usually, the size of the message is way larger than the number of nodes in the system.
- Split message m into n fragments $f_1, f_2, \dots, f_n$ using (n, k) erasure code.
- Assume optimal erasure code, any k fragments allow to restore the message m.
- What values do we choose for n and k, and why?
- What are we still missing?



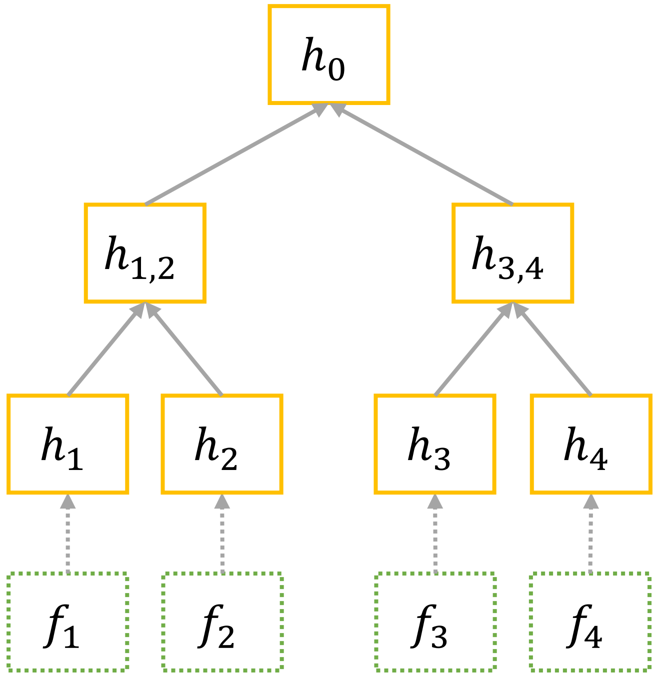
Efficient Reliable Broadcast

- Use fact that usually, the size of the message is way larger than the number of nodes in the system.
- Split message m into n fragments $f_1, f_2, \dots, f_n$ using (n, k) erasure code.
- Assume optimal erasure code, any k fragments allow to restore the message m.
- What values do we choose for n and k, and why?
- What are we still missing?



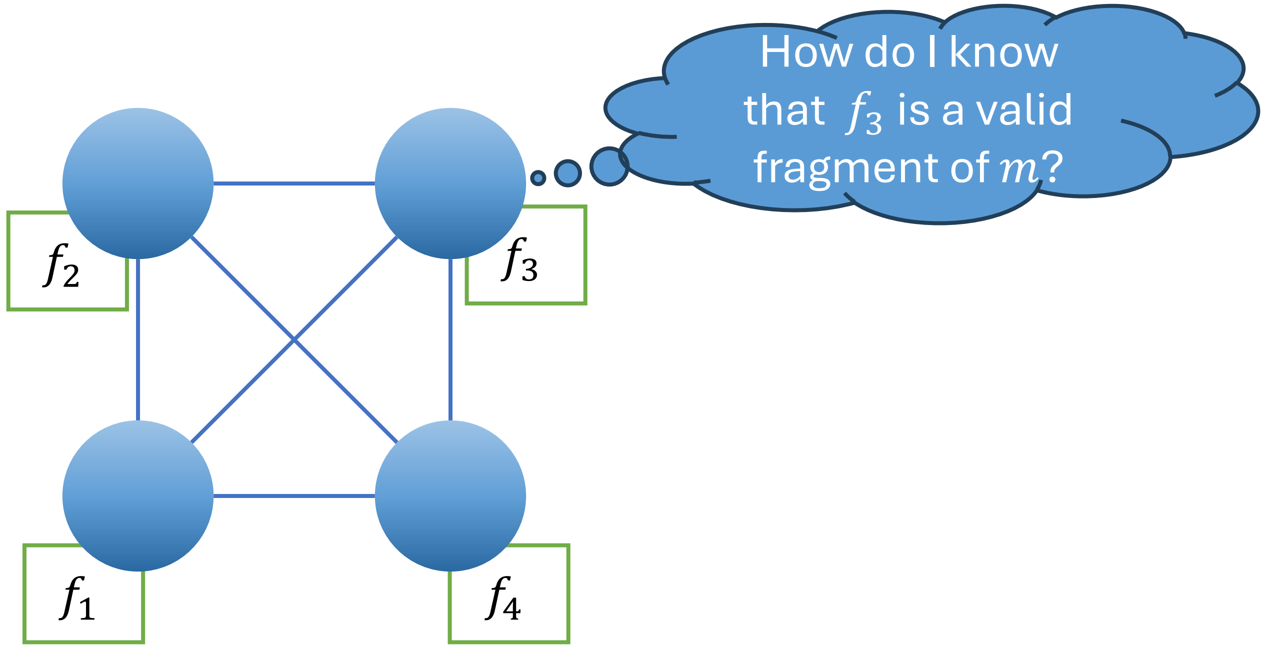
Merkle Trees

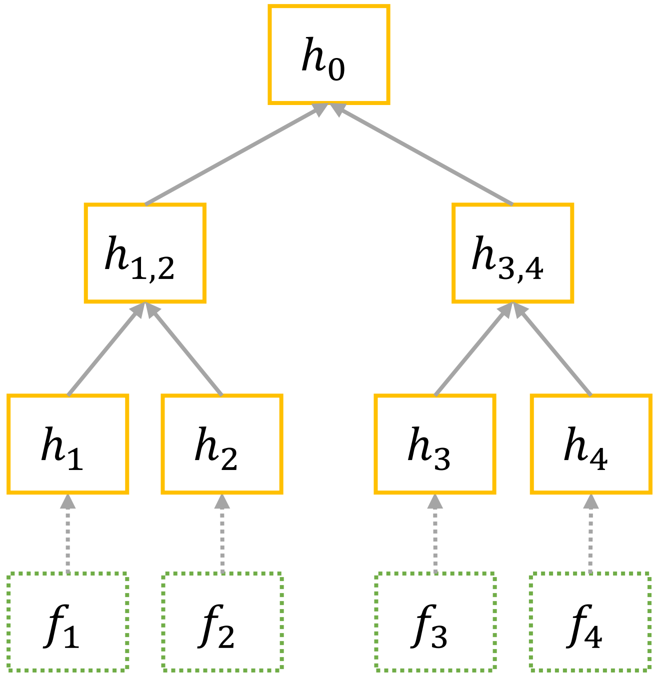
- A tree in which every leaf is labeled with the hash of a data block, and every inner node (including the root) is labeled with the hash of its children.



ETH zürich Systems@ETH zürich Distributed Computing

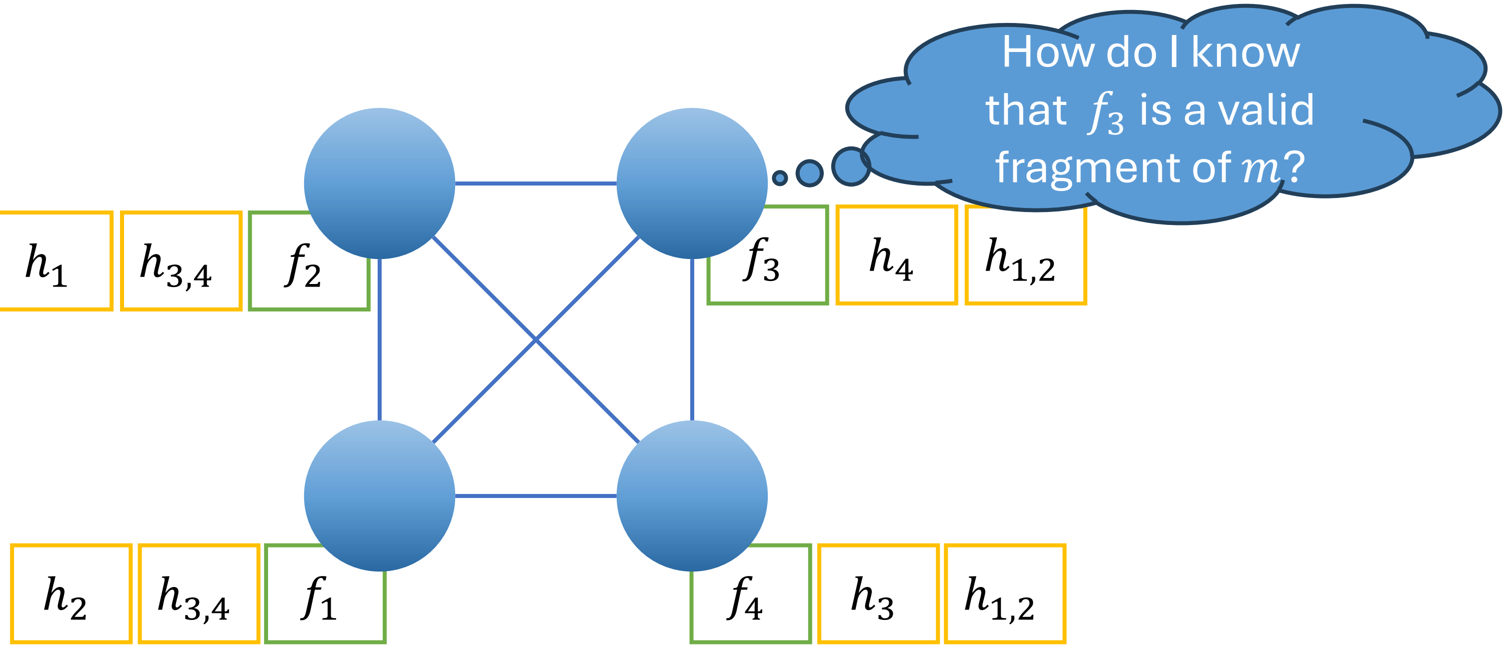
Merkle Proof

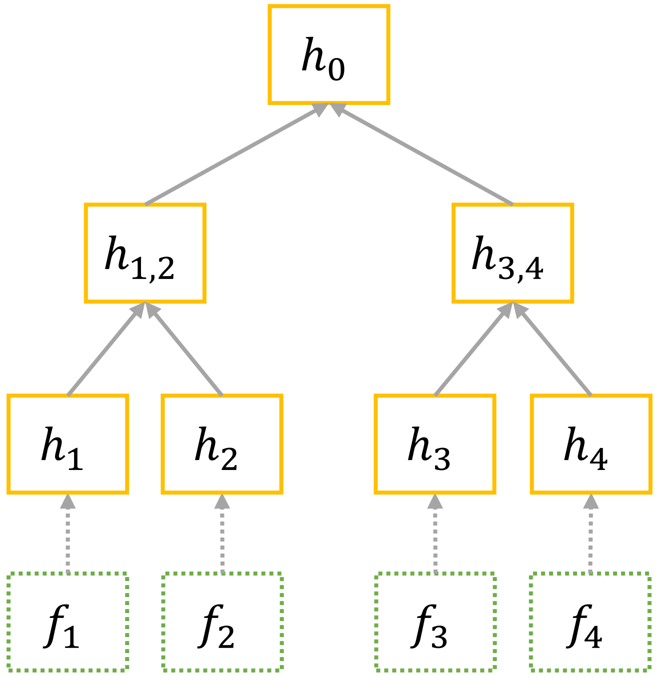




Merkle Proof

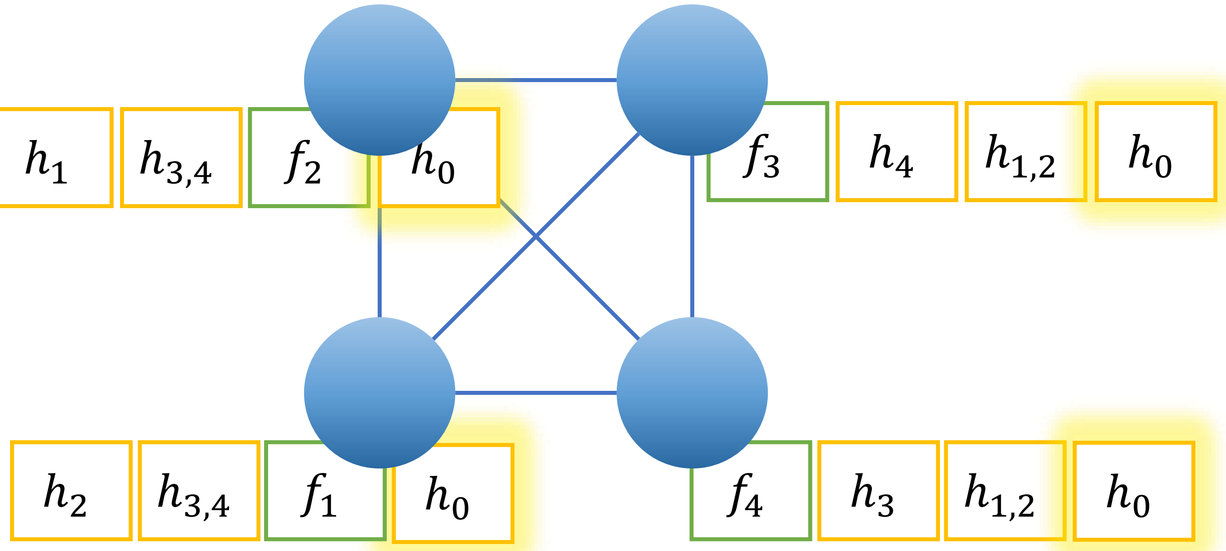
- All hashes required to recompute the root has h_0 given a fragment f_i.
- Can you find a closed form expression for the size of a Merkle Proof for n fragments?

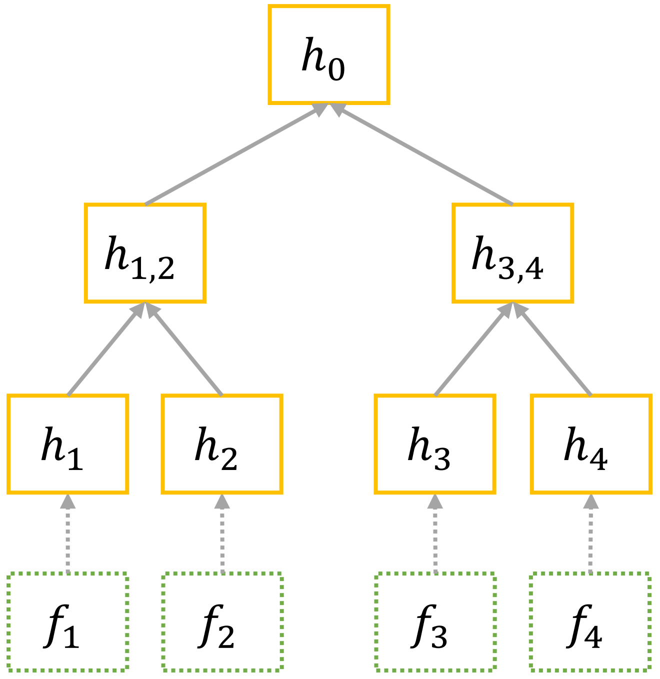




Merkle Proof

- All hashes required to recompute the root has h_0 given a fragment f_i.
- Can you find a closed form expression for the size of a Merkle Proof for n fragments?
- Still need a "correct" root hash to compare computed root hash to. Any idea on how to distribute it?

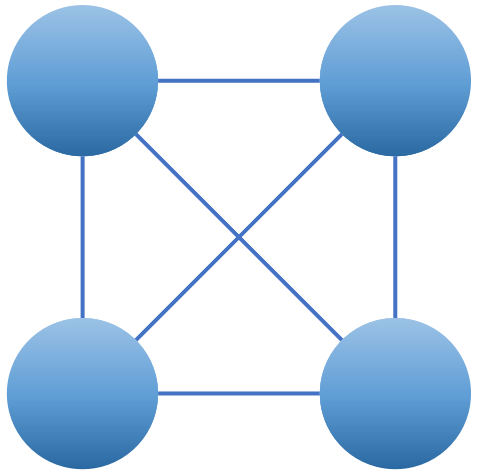




Distributed Computing

Efficient Reliable Broadcast

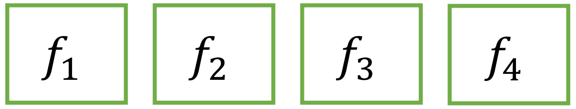
m

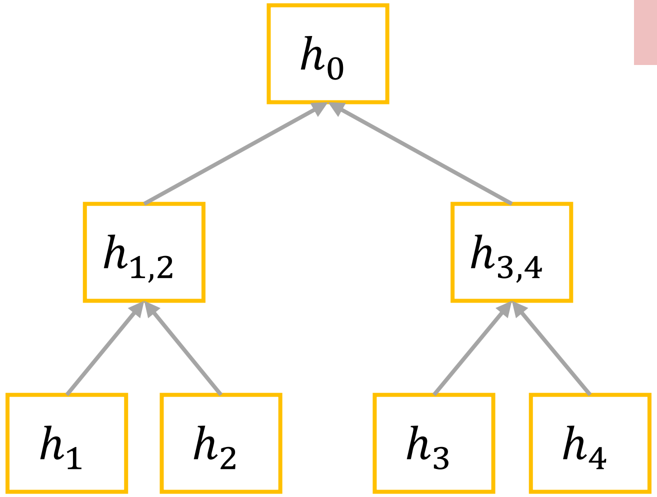


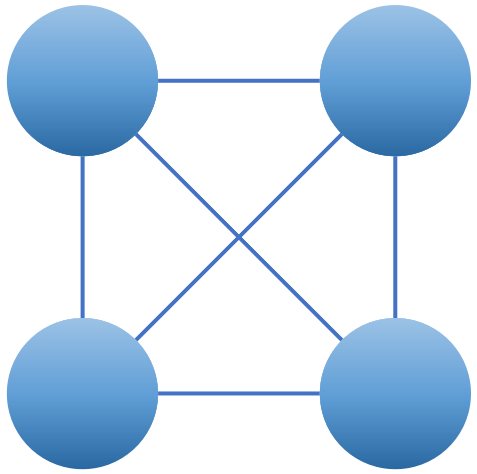
Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \bot$, $F = \{\}$, and $m = \bot$.

```
$(f_1, \ldots, f_n) :=$ get_fragments$(m)$
$h_0 :=$ get_merkle_root_hash$((f_1, \ldots, f_n))$
Execute Algorithm $\boxed{18.11}$ for message $root\_hash := h_0$
for $v_j \in V$ do
    $P_j :=$ get_merkle_proof$((f_1, \ldots, f_n), j)$
    Send $(f_j, P_j)$ to $v_j$
end for

upon receiving $(f_j, P_j)$ and $root\_hash \neq \bot$:
    if valid$(f_j, P_j, root\_hash)$ then
        $F := F \cup \{f_j\}$
        if $i = j$ and not broadcast $(f_i, P_i)$ before then
            Broadcast $(f_i, P_i)$
        end if
    end if
end upon

if $|F| = f + 1$ and $m = \bot$ then
    $m :=$ recover_message$(F)$
    $(f_1, \ldots, f_n) :=$ get_fragments$(m)$
    $h_0 :=$ get_merkle_root_hash$((f_1, \ldots, f_n))$
    if $h_0 = root\_hash$ then
        if not broadcast $(f_i, P_i)$ before then
            $P_i :=$ get_merkle_path$((f_1, \ldots, f_n), i)$
            Broadcast $(f_i, P_i)$
        end if
    else
        $root\_hash := \bot$
    end if
end if

if $|F| = n - f$ and $m \neq \bot$ and $root\_hash \neq \bot$ and not delivered then
    deliver$(m)$
end if
```

Efficient Reliable Broadcast





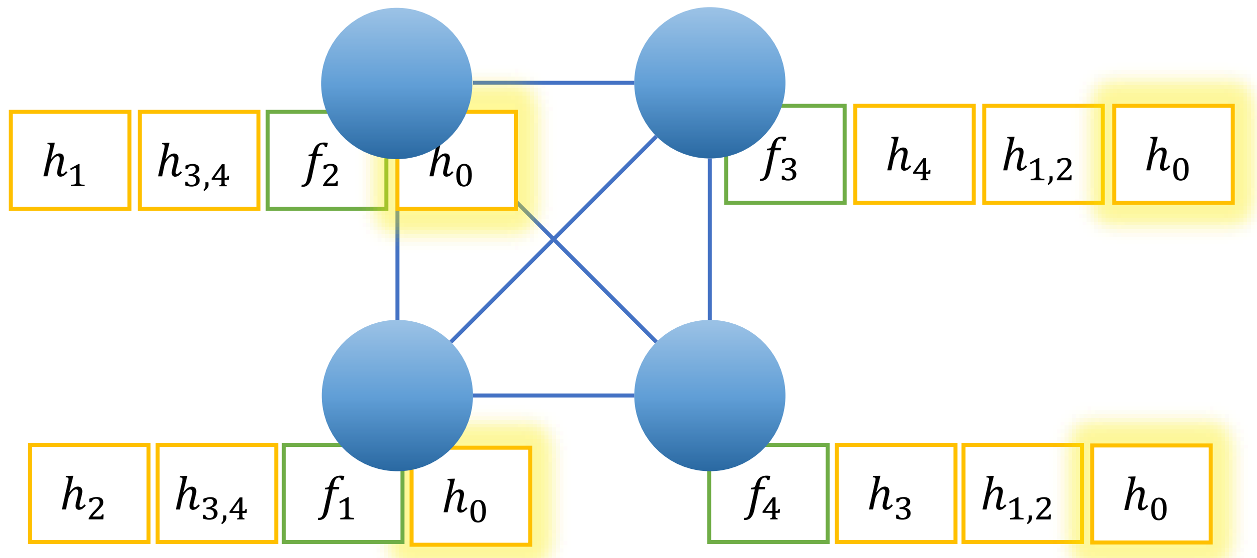


Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \bot$, $F = \{\}$, and $m = \bot$.

```
$(f_1, \ldots, f_n)$ := get_fragments($m$)
$h_0$ := get_merkle_root_hash($(f_1, \ldots, f_n)$)
Execute Algorithm $\boxed{18.11}$ for message $root\_hash$ :=$h_0$
for $v_j \in V$ do
    $P_j$ := get_merkle_proof($(f_1, \ldots, f_n), j$)
    Send $(f_j, P_j)$ to $v_j$
end for

upon receiving $(f_j, P_j)$ and $root\_hash \neq \bot$:
    if valid($f_j, P_j, root\_hash$) then
        $F := F \cup \{f_j\}$
        if $i = j$ and not broadcast $(f_i, P_i)$ before then
            Broadcast $(f_i, P_i)$
        end if
    end if
end upon

if $|F| = f + 1$ and $m = \bot$ then
    $m$ := recover_message($F$)
    $(f_1, \ldots, f_n)$ := get_fragments($m$)
    $h_0$ := get_merkle_root_hash($(f_1, \ldots, f_n)$)
    if $h_0 = root\_hash$ then
        if not broadcast $(f_i, P_i)$ before then
            $P_i$ := get_merkle_path($(f_1, \ldots, f_n), i$)
            Broadcast $(f_i, P_i)$
        end if
    else
        $root\_hash := \bot$
    end if
end if

if $|F| = n - f$ and $m \neq \bot$ and $root\_hash \neq \bot$ and not delivered then
    deliver($m$)
end if
```

Distributed Computing

Efficient Reliable Broadcast

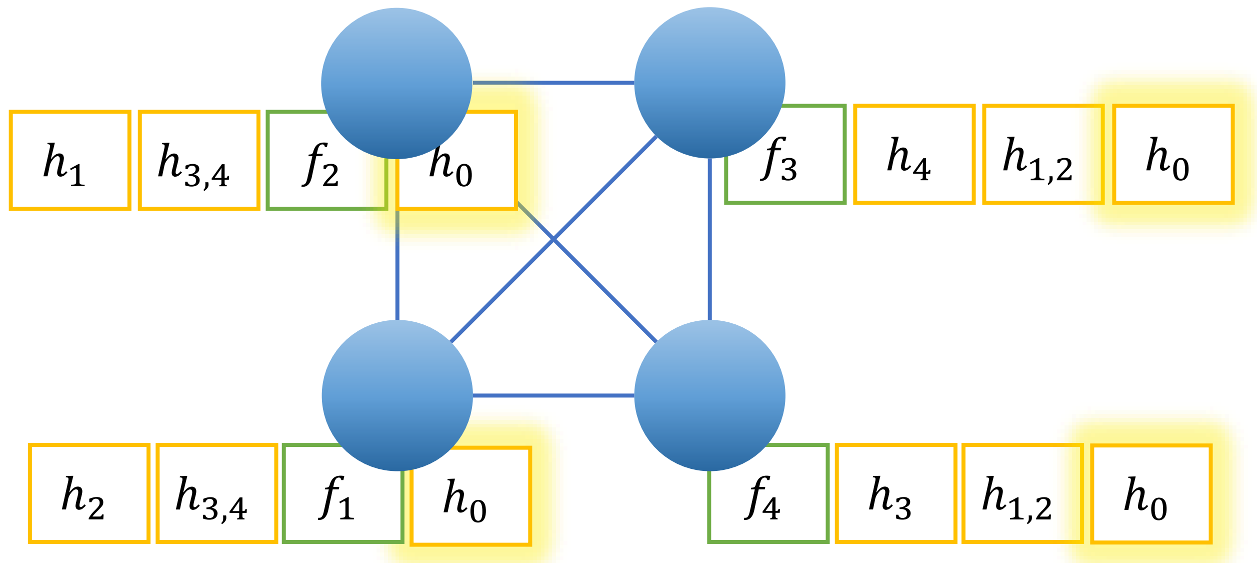


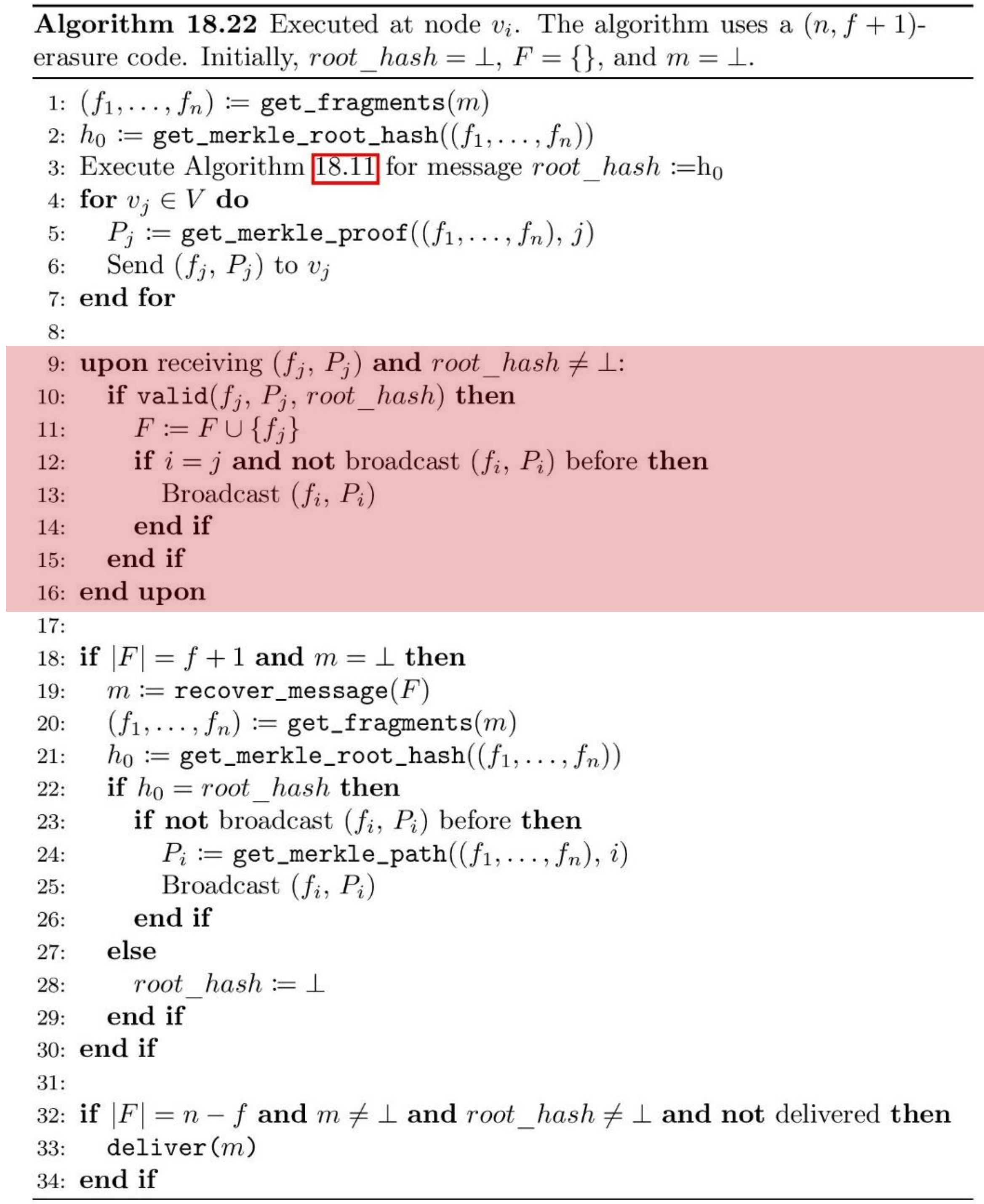
Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \perp$, $F = \{\}$, and $m = \perp$.

```
(f_1, ..., f_n) := get_fragments(m)
h_0 := get_merkle_root_hash((f_1, ..., f_n))
Execute Algorithm 18.11 for message root_hash :=h_0
for v_j ∈ V do
    P_j := get_merkle_proof((f_1, ..., f_n), j)
    Send (f_j, P_j) to v_j
end for

upon receiving (f_j, P_j) and root_hash ≠ ⊥:
    if valid(f_j, P_j, root_hash) then
        F := F ∪ {f_j}
        if i = j and not broadcast (f_i, P_i) before then
            Broadcast (f_i, P_i)
        end if
    end if
end upon

if |F| = f + 1 and m = ⊥ then
    m := recover_message(F)
    (f_1, ..., f_n) := get_fragments(m)
    h_0 := get_merkle_root_hash((f_1, ..., f_n))
    if h_0 = root_hash then
        if not broadcast (f_i, P_i) before then
            P_i := get_merkle_path((f_1, ..., f_n), i)
            Broadcast (f_i, P_i)
        end if
    else
        root_hash := ⊥
    end if
end if

if |F| = n − f and m ≠ ⊥ and root_hash ≠ ⊥ and not delivered then
    deliver(m)
end if
```

Efficient Reliable Broadcast

- Upon receiving a fragment, we can use the Merkle Proof to establish authenticity of the fragment data.

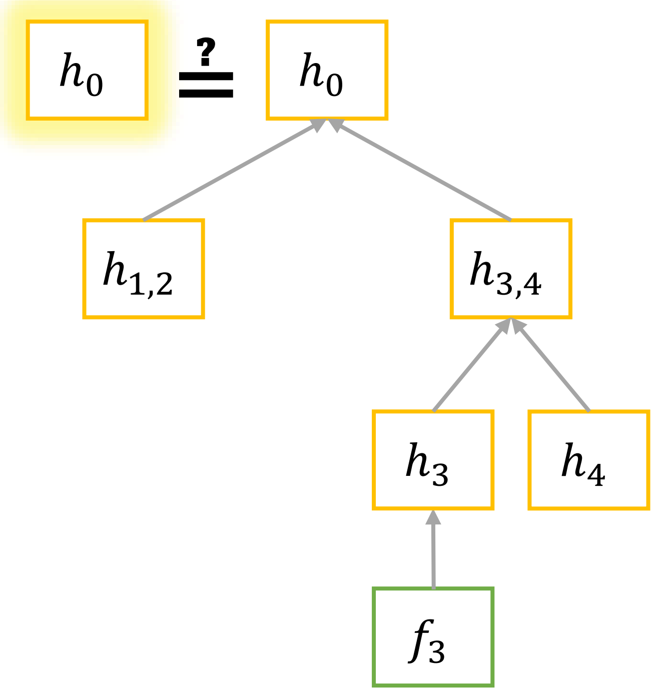


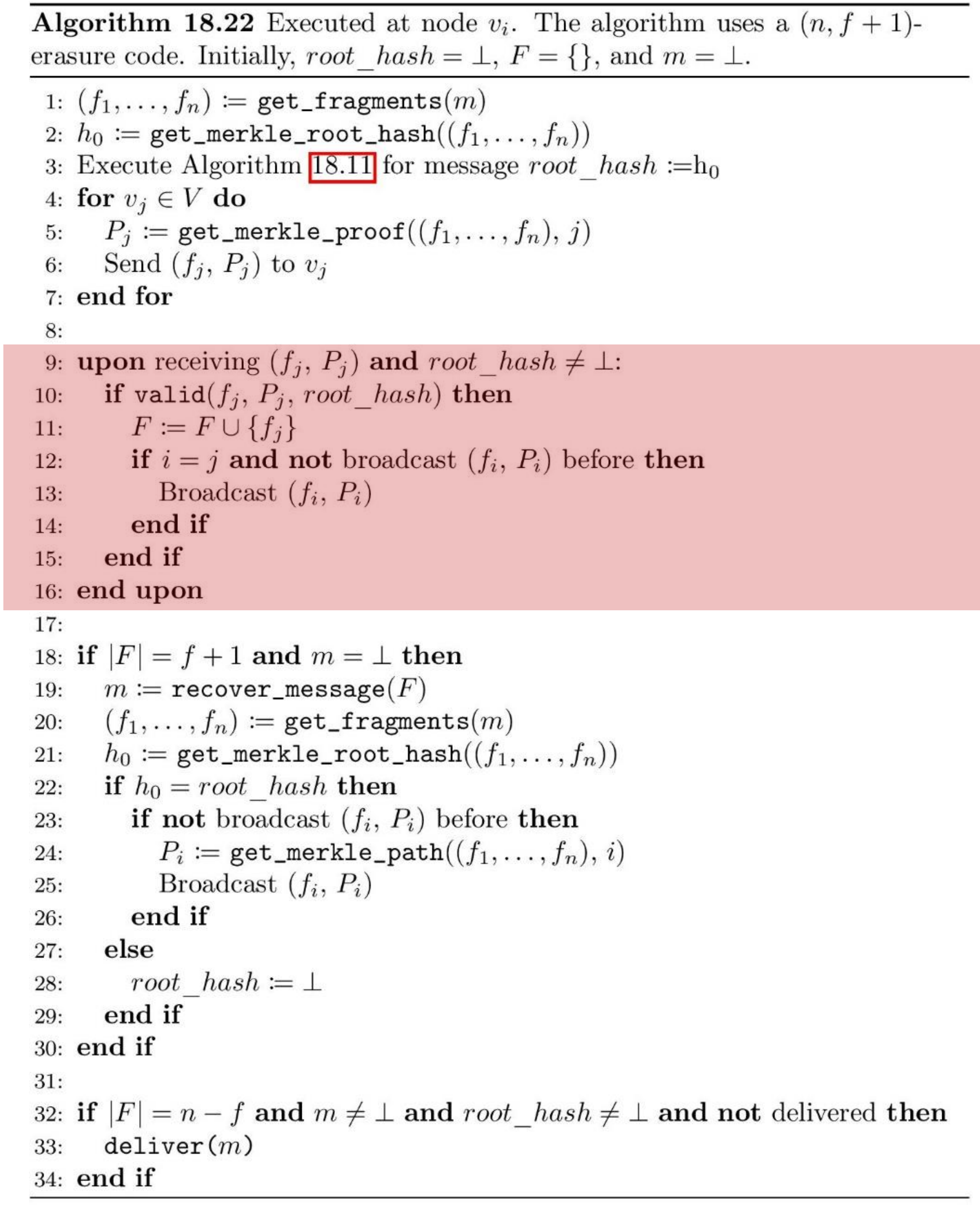

Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \perp$, $F = \{\}$, and $m = \perp$.

```
$(f_1, \ldots, f_n) :=$ get_fragments($m$)
$h_0 :=$ get_merkle_root_hash($(f_1, \ldots, f_n)$)
Execute Algorithm $\boxed{18.11}$ for message $root\_hash := h_0$
for $v_j \in V$ do
    $P_j :=$ get_merkle_proof($(f_1, \ldots, f_n)$, $j$)
    Send $(f_j, P_j)$ to $v_j$
end for

upon receiving $(f_j, P_j)$ and $root\_hash \neq \perp$:
    if valid($f_j$, $P_j$, $root\_hash$) then
        $F := F \cup \{f_j\}$
        if $i = j$ and not broadcast $(f_i, P_i)$ before then
            Broadcast $(f_i, P_i)$
        end if
    end if
end upon

if $|F| = f + 1$ and $m = \perp$ then
    $m :=$ recover_message($F$)
    $(f_1, \ldots, f_n) :=$ get_fragments($m$)
    $h_0 :=$ get_merkle_root_hash($(f_1, \ldots, f_n)$)
    if $h_0 = root\_hash$ then
        if not broadcast $(f_i, P_i)$ before then
            $P_i :=$ get_merkle_path($(f_1, \ldots, f_n)$, $i$)
            Broadcast $(f_i, P_i)$
        end if
    else
        $root\_hash := \perp$
    end if
end if

if $|F| = n - f$ and $m \neq \perp$ and $root\_hash \neq \perp$ and not delivered then
    deliver($m$)
end if
```

Efficient Reliable Broadcast

- Upon receiving a fragment, we can use the Merkle Proof to establish authenticity of the fragment data.



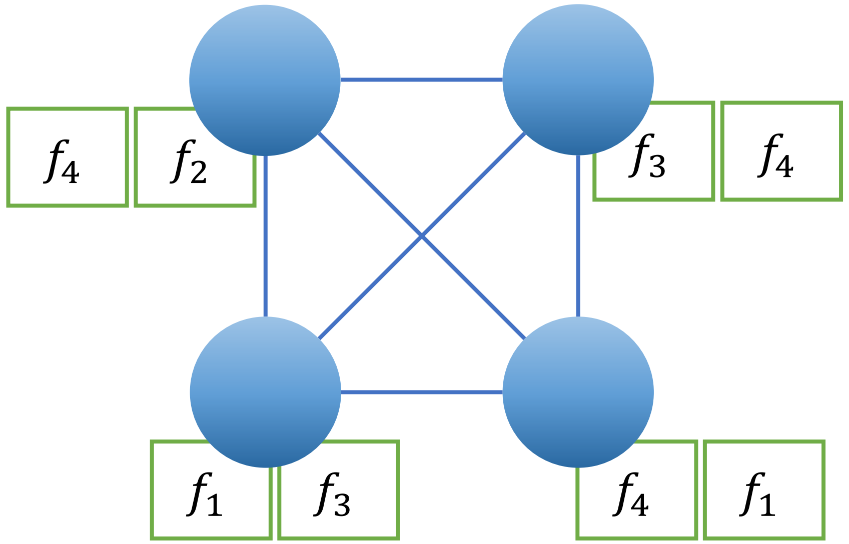


Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \perp$, $F = \{\}$, and $m = \perp$.

```
$(f_1, \ldots, f_n)$ := get_fragments($m$)
$h_0$ := get_merkle_root_hash($(f_1, \ldots, f_n)$)
Execute Algorithm $\boxed{18.11}$ for message $root\_hash$ :=$h_0$
for $v_j \in V$ do
    $P_j$ := get_merkle_proof($(f_1, \ldots, f_n), j$)
    Send $(f_j, P_j)$ to $v_j$
end for

upon receiving $(f_j, P_j)$ and $root\_hash \neq \perp$:
    if valid($f_j, P_j, root\_hash$) then
        $F := F \cup \{f_j\}$
        if $i = j$ and not broadcast $(f_i, P_i)$ before then
            Broadcast $(f_i, P_i)$
        end if
    end if
end upon

if $|F| = f + 1$ and $m = \perp$ then
    $m$ := recover_message($F$)
    $(f_1, \ldots, f_n)$ := get_fragments($m$)
    $h_0$ := get_merkle_root_hash($(f_1, \ldots, f_n)$)
    if $h_0 = root\_hash$ then
        if not broadcast $(f_i, P_i)$ before then
            $P_i$ := get_merkle_path($(f_1, \ldots, f_n), i$)
            Broadcast $(f_i, P_i)$
        end if
    else
        $root\_hash := \perp$
    end if
end if

if $|F| = n - f$ and $m \neq \perp$ and $root\_hash \neq \perp$ and not delivered then
    deliver($m$)
end if
```

Distributed Computing

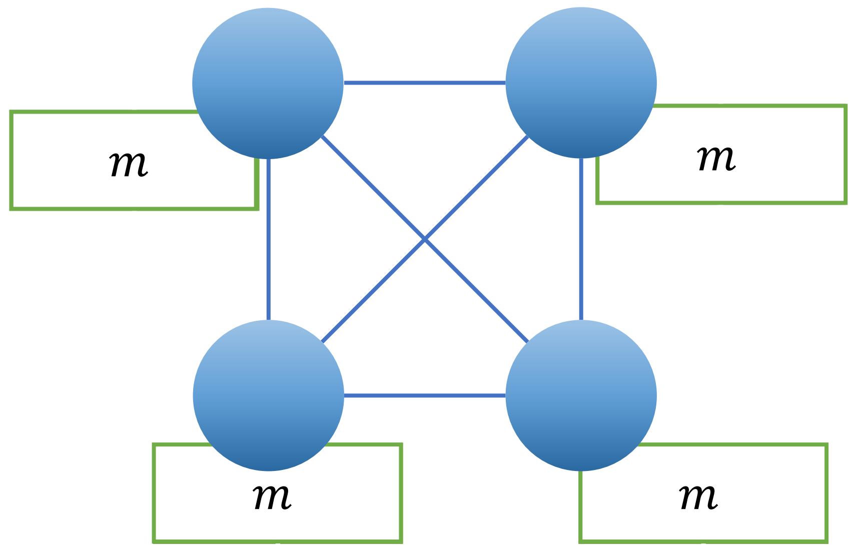
Efficient Reliable Broadcast



Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \perp$, $F = \{\}$, and $m = \perp$.

```
$(f_1, \dots, f_n) := \texttt{get\_fragments}(m)$
$h_0 := \texttt{get\_merkle\_root\_hash}((f_1, \dots, f_n))$
Execute Algorithm $\boxed{18.11}$ for message $root\_hash$ :=h$_0$
for $v_j \in V$ do
    $P_j := \texttt{get\_merkle\_proof}((f_1, \dots, f_n), j)$
    Send $(f_j, P_j)$ to $v_j$
end for

upon receiving $(f_j, P_j)$ and $root\_hash \neq \perp$:
    if $\texttt{valid}(f_j, P_j, root\_hash)$ then
        $F := F \cup \{f_j\}$
        if $i = j$ and not broadcast $(f_i, P_i)$ before then
            Broadcast $(f_i, P_i)$
        end if
    end if
end upon

if $|F| = f + 1$ and $m = \perp$ then
    $m := \texttt{recover\_message}(F)$
    $(f_1, \dots, f_n) := \texttt{get\_fragments}(m)$
    $h_0 := \texttt{get\_merkle\_root\_hash}((f_1, \dots, f_n))$
    if $h_0 = root\_hash$ then
        if not broadcast $(f_i, P_i)$ before then
            $P_i := \texttt{get\_merkle\_path}((f_1, \dots, f_n), i)$
            Broadcast $(f_i, P_i)$
        end if
    else
        $root\_hash := \perp$
    end if
end if

if $|F| = n - f$ and $m \neq \perp$ and $root\_hash \neq \perp$ and not delivered then
    $\texttt{deliver}(m)$
end if
```

Efficient Reliable Broadcast

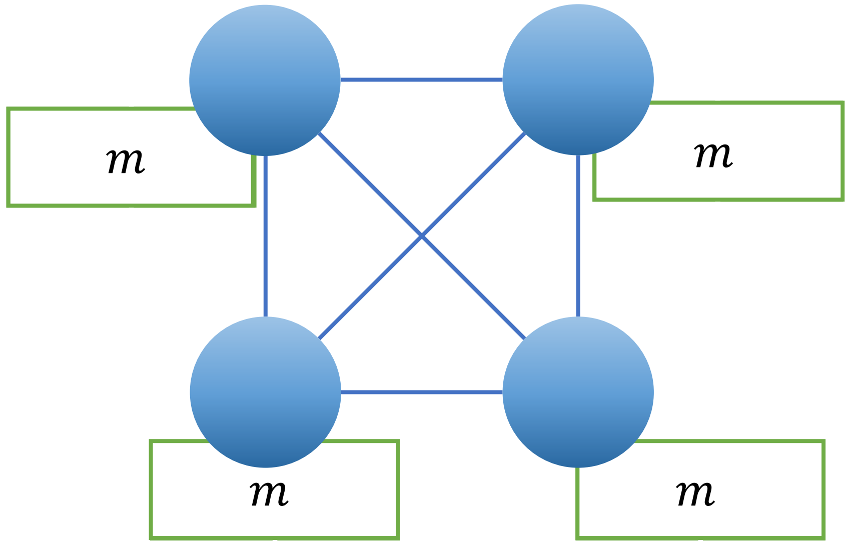


Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \perp$, $F = \{\}$, and $m = \perp$.

```
$(f_1, \dots, f_n) :=$ get_fragments$(m)$
$h_0 :=$ get_merkle_root_hash$((f_1, \dots, f_n))$
Execute Algorithm $\boxed{18.11}$ for message $root\_hash := h_0$
for $v_j \in V$ do
    $P_j :=$ get_merkle_proof$((f_1, \dots, f_n), j)$
    Send $(f_j, P_j)$ to $v_j$
end for

upon receiving $(f_j, P_j)$ and $root\_hash \neq \perp$:
    if valid$(f_j, P_j, root\_hash)$ then
        $F := F \cup \{f_j\}$
        if $i = j$ and not broadcast $(f_i, P_i)$ before then
            Broadcast $(f_i, P_i)$
        end if
    end if
end upon

if $|F| = f + 1$ and $m = \perp$ then
    $m :=$ recover_message$(F)$
    $(f_1, \dots, f_n) :=$ get_fragments$(m)$
    $h_0 :=$ get_merkle_root_hash$((f_1, \dots, f_n))$
    if $h_0 = root\_hash$ then
        if not broadcast $(f_i, P_i)$ before then
            $P_i :=$ get_merkle_path$((f_1, \dots, f_n), i)$
            Broadcast $(f_i, P_i)$
        end if
    else
        $root\_hash := \perp$
    end if
end if

if $|F| = n - f$ and $m \neq \perp$ and $root\_hash \neq \perp$ and not delivered then
    deliver$(m)$
end if
```

ETH zürich Systems@ETH zürich Distributed Computing

Efficient Reliable Broadcast



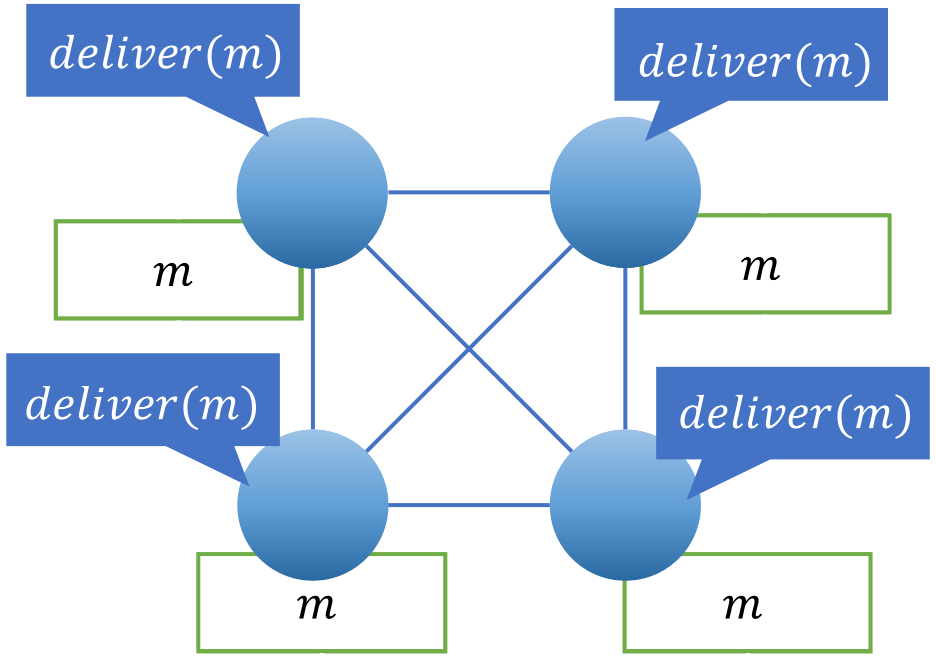
Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \perp$, $F = \{\}$, and $m = \perp$.

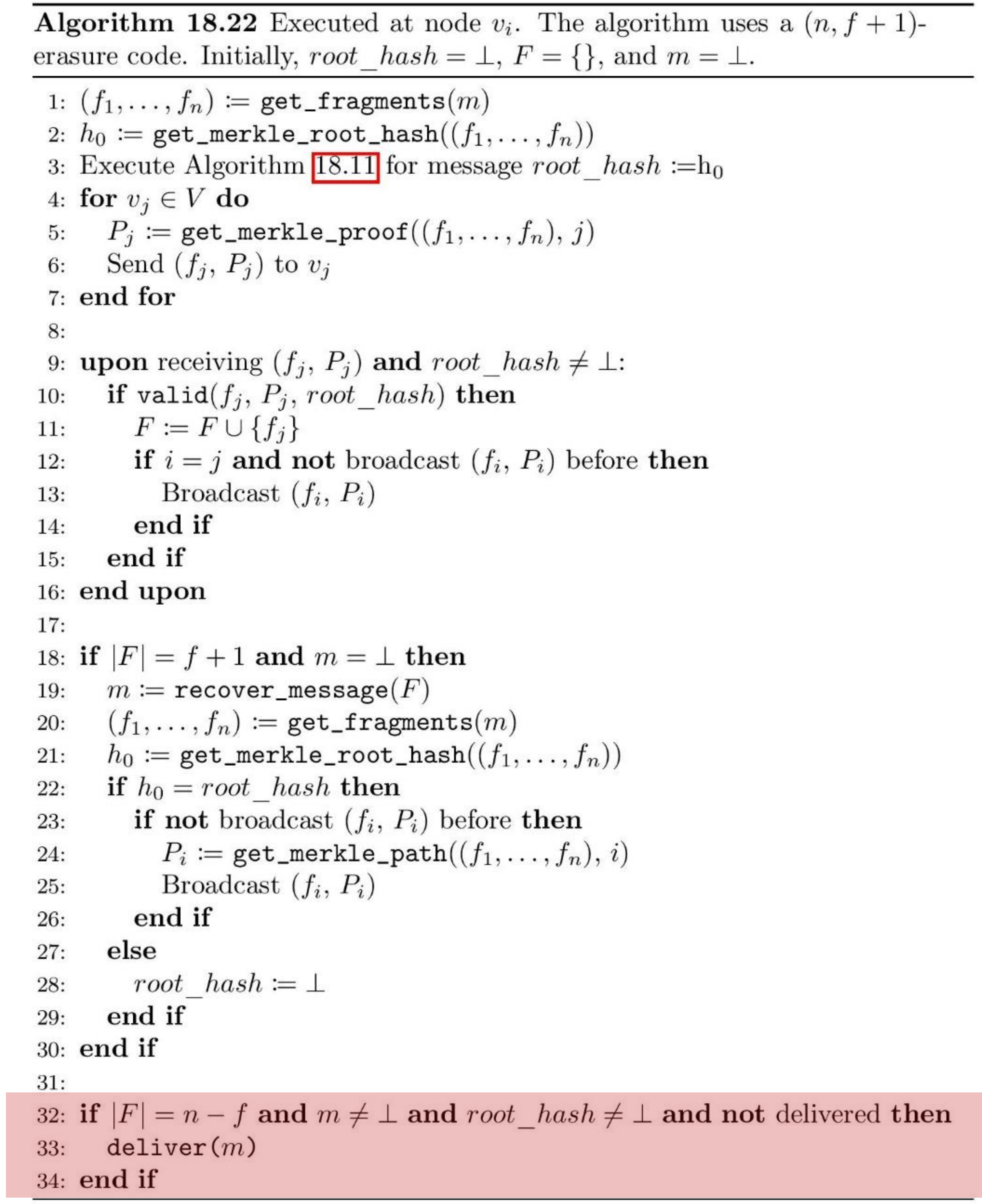
```
1: $(f_1, \dots, f_n) :=$ get_fragments$(m)$
2: $h_0 :=$ get_merkle_root_hash$((f_1, \dots, f_n))$
3: Execute Algorithm $\boxed{18.11}$ for message $root\_hash$ :=h$_0$
4: for $v_j \in V$ do
5:     $P_j :=$ get_merkle_proof$((f_1, \dots, f_n), j)$
6:     Send $(f_j, P_j)$ to $v_j$
7: end for
8:
9: upon receiving $(f_j, P_j)$ and $root\_hash \neq \perp$:
10:    if valid$(f_j, P_j, root\_hash)$ then
11:        $F := F \cup \{f_j\}$
12:        if $i = j$ and not broadcast $(f_i, P_i)$ before then
13:            Broadcast $(f_i, P_i)$
14:        end if
15:    end if
16: end upon
17:
18: if $|F| = f + 1$ and $m = \perp$ then
19:    $m :=$ recover_message$(F)$
20:    $(f_1, \dots, f_n) :=$ get_fragments$(m)$
21:    $h_0 :=$ get_merkle_root_hash$((f_1, \dots, f_n))$
22:    if $h_0 = root\_hash$ then
23:        if not broadcast $(f_i, P_i)$ before then
24:            $P_i :=$ get_merkle_path$((f_1, \dots, f_n), i)$
25:            Broadcast $(f_i, P_i)$
26:        end if
27:    else
28:        $root\_hash := \perp$
29:    end if
30: end if
31:
32: if $|F| = n - f$ and $m \neq \perp$ and $root\_hash \neq \perp$ and not delivered then
33:    deliver$(m)$
34: end if
```

ETH zürich Systems@ETH zürich Distributed Computing

Efficient Reliable Broadcast

- What did we gain?
- Communication complexity of $3|m|n + O(n^2 \log(n)\, H)$ in the asynchronous model





Algorithm 18.22 Executed at node v_i. The algorithm uses a $(n, f+1)$-erasure code. Initially, $root\_hash = \perp$, $F = \{\}$, and $m = \perp$.

```
$(f_1, \dots, f_n) :=$ get_fragments$(m)$
$h_0 :=$ get_merkle_root_hash$((f_1, \dots, f_n))$
Execute Algorithm $\boxed{18.11}$ for message $root\_hash := h_0$
for $v_j \in V$ do
    $P_j :=$ get_merkle_proof$((f_1, \dots, f_n), j)$
    Send $(f_j, P_j)$ to $v_j$
end for

upon receiving $(f_j, P_j)$ and $root\_hash \neq \perp$:
    if valid$(f_j, P_j, root\_hash)$ then
        $F := F \cup \{f_j\}$
        if $i = j$ and not broadcast $(f_i, P_i)$ before then
            Broadcast $(f_i, P_i)$
        end if
    end if
end upon

if $|F| = f + 1$ and $m = \perp$ then
    $m :=$ recover_message$(F)$
    $(f_1, \dots, f_n) :=$ get_fragments$(m)$
    $h_0 :=$ get_merkle_root_hash$((f_1, \dots, f_n))$
    if $h_0 = root\_hash$ then
        if not broadcast $(f_i, P_i)$ before then
            $P_i :=$ get_merkle_path$((f_1, \dots, f_n), i)$
            Broadcast $(f_i, P_i)$
        end if
    else
        $root\_hash := \perp$
    end if
end if

if $|F| = n - f$ and $m \neq \perp$ and $root\_hash \neq \perp$ and not delivered then
    deliver$(m)$
end if
```

Distributed Computing

Quiz

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f.

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model.

3. If in the first phase of the King algorithm the first king is honest, then:
    1. Every honest node will decide for the initial input of the king.
    2. An honest node will propose a value in the first phase.
    3. The king himself can change its own value in the first phase.
    4. The nodes might change the value in future phases if there are other honest kings later.

4. In reliable broadcast:
    1. All honest nodes accept at most one message.
    2. All honest nodes accept at least one message.
    3. All honest nodes echo at least one message.
    4. It might happen that no honest node accepts a message.

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model.

3. If in the first phase of the King algorithm the first king is honest, then:
   1. Every honest node will decide for the initial input of the king.
   2. An honest node will propose a value in the first phase.
   3. The king himself can change its own value in the first phase.
   4. The nodes might change the value in future phases if there are other honest kings later.

4. In reliable broadcast:
   1. All honest nodes accept at most one message.
   2. All honest nodes accept at least one message.
   3. All honest nodes echo at least one message.
   4. It might happen that no honest node accepts a message.

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
    1. Every honest node will decide for the initial input of the king.
    2. An honest node will propose a value in the first phase.
    3. The king himself can change its own value in the first phase.
    4. The nodes might change the value in future phases if there are other honest kings later.

4. In reliable broadcast:
    1. All honest nodes accept at most one message.
    2. All honest nodes accept at least one message.
    3. All honest nodes echo at least one message.
    4. It might happen that no honest node accepts a message.

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
    1. Every honest node will decide for the initial input of the king. False
    2. An honest node will propose a value in the first phase.
    3. The king himself can change its own value in the first phase.
    4. The nodes might change the value in future phases if there are other honest kings later.

4. In reliable broadcast:
    1. All honest nodes accept at most one message.
    2. All honest nodes accept at least one message.
    3. All honest nodes echo at least one message.
    4. It might happen that no honest node accepts a message.

ETH zürich Systems@ETH zürich Distributed Computing

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
   1. Every honest node will decide for the initial input of the king. False
   2. An honest node will propose a value in the first phase. False
   3. The king himself can change its own value in the first phase.
   4. The nodes might change the value in future phases if there are other honest kings later.

4. In reliable broadcast:
   1. All honest nodes accept at most one message.
   2. All honest nodes accept at least one message.
   3. All honest nodes echo at least one message.
   4. It might happen that no honest node accepts a message.

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
   1. Every honest node will decide for the initial input of the king. False
   2. An honest node will propose a value in the first phase. False
   3. The king himself can change its own value in the first phase. True
   4. The nodes might change the value in future phases if there are other honest kings later.

4. In reliable broadcast:
   1. All honest nodes accept at most one message.
   2. All honest nodes accept at least one message.
   3. All honest nodes echo at least one message.
   4. It might happen that no honest node accepts a message.

ETH zürich Systems@ETH zürich Distributed Computing

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
   1. Every honest node will decide for the initial input of the king. False
   2. An honest node will propose a value in the first phase. False
   3. The king himself can change its own value in the first phase. True
   4. The nodes might change the value in future phases if there are other honest kings later. False

4. In reliable broadcast:
   1. All honest nodes accept at most one message.
   2. All honest nodes accept at least one message.
   3. All honest nodes echo at least one message.
   4. It might happen that no honest node accepts a message.

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
   1. Every honest node will decide for the initial input of the king. False
   2. An honest node will propose a value in the first phase. False
   3. The king himself can change its own value in the first phase. True
   4. The nodes might change the value in future phases if there are other honest kings later. False

4. In reliable broadcast:
   1. All honest nodes accept at most one message. False
   2. All honest nodes accept at least one message.
   3. All honest nodes echo at least one message.
   4. It might happen that no honest node accepts a message.

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
   1. Every honest node will decide for the initial input of the king. False
   2. An honest node will propose a value in the first phase. False
   3. The king himself can change its own value in the first phase. True
   4. The nodes might change the value in future phases if there are other honest kings later. False

4. In reliable broadcast:
   1. All honest nodes accept at most one message. False
   2. All honest nodes accept at least one message. False
   3. All honest nodes echo at least one message.
   4. It might happen that no honest node accepts a message.

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
   1. Every honest node will decide for the initial input of the king. False
   2. An honest node will propose a value in the first phase. False
   3. The king himself can change its own value in the first phase. True
   4. The nodes might change the value in future phases if there are other honest kings later. False

4. In reliable broadcast:
   1. All honest nodes accept at most one message. False
   2. All honest nodes accept at least one message. False
   3. All honest nodes echo at least one message. False
   4. It might happen that no honest node accepts a message.

Quiz questions (choose the right answer)

1. No algorithm satisfies byzantine agreement with n = 3f. True

2. A single byzantine node can prevent any deterministic algorithm from reaching agreement in asynchronous model. True

3. If in the first phase of the King algorithm the first king is honest, then:
   1. Every honest node will decide for the initial input of the king. False
   2. An honest node will propose a value in the first phase. False
   3. The king himself can change its own value in the first phase. True
   4. The nodes might change the value in future phases if there are other honest kings later. False

4. In reliable broadcast:
   1. All honest nodes accept at most one message. False
   2. All honest nodes accept at least one message. False
   3. All honest nodes echo at least one message. False
   4. It might happen that no honest node accepts a message. True

Distributed Computing

Assignment Preview

Assignment Preview

1 Synchronous Model

Quiz

1.1 Synchronous Consensus in a Grid

In the lecture you learned how to reach consensus in a fully connected network where every process can communicate directly with every other process. Now consider a network that is organized as a 2-dimensional grid such that every process has up to 4 neighbors. The width of the grid is w, the height is h. Width and height are defined in terms of edges: A 2×2 grid contains 9 nodes! The grid is big, meaning that $w + h$ is much smaller than $w \cdot h$. We use the synchronous time model; i.e., in every round every process may send a message to each of its neighbors, and the size of the message is not limited.

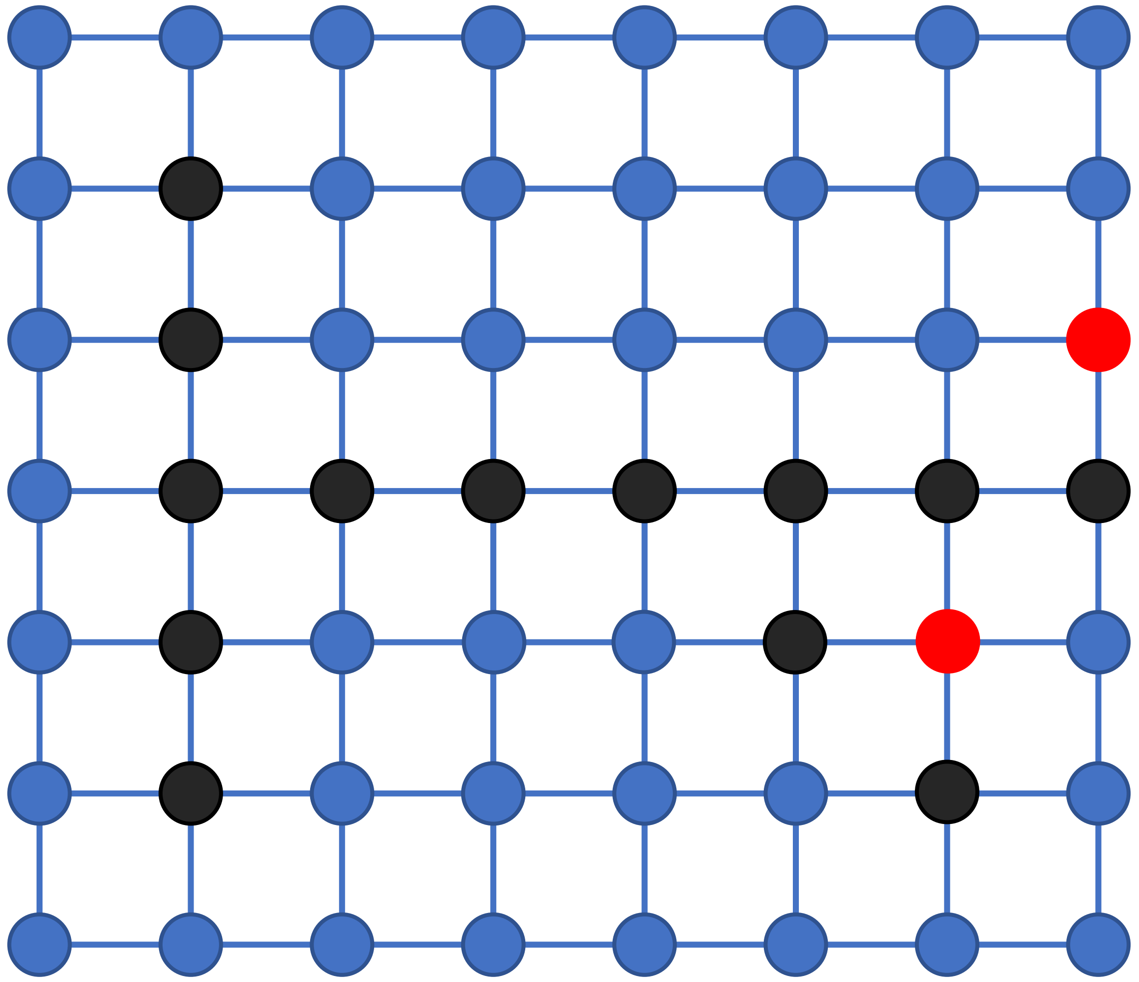
a) Assume every node knows w and h. Write a short protocol to reach consensus.

b) From now on the nodes do not know the size of the grid. Write a protocol to reach consensus and optimize it according to runtime.

c) How many rounds does your protocol from **b)** require?

Assume there are Byzantine nodes and that you are the adversary who can select which nodes are Byzantine.

d) What is the smallest number of Byzantine nodes that you need to prevent the system from reaching agreement, and where would you place them?



ETH zürich Systems@ETH zürich Distributed Computing

Assignment Preview

1.2 Synchronous Consensus in a Grid - Crash Failures

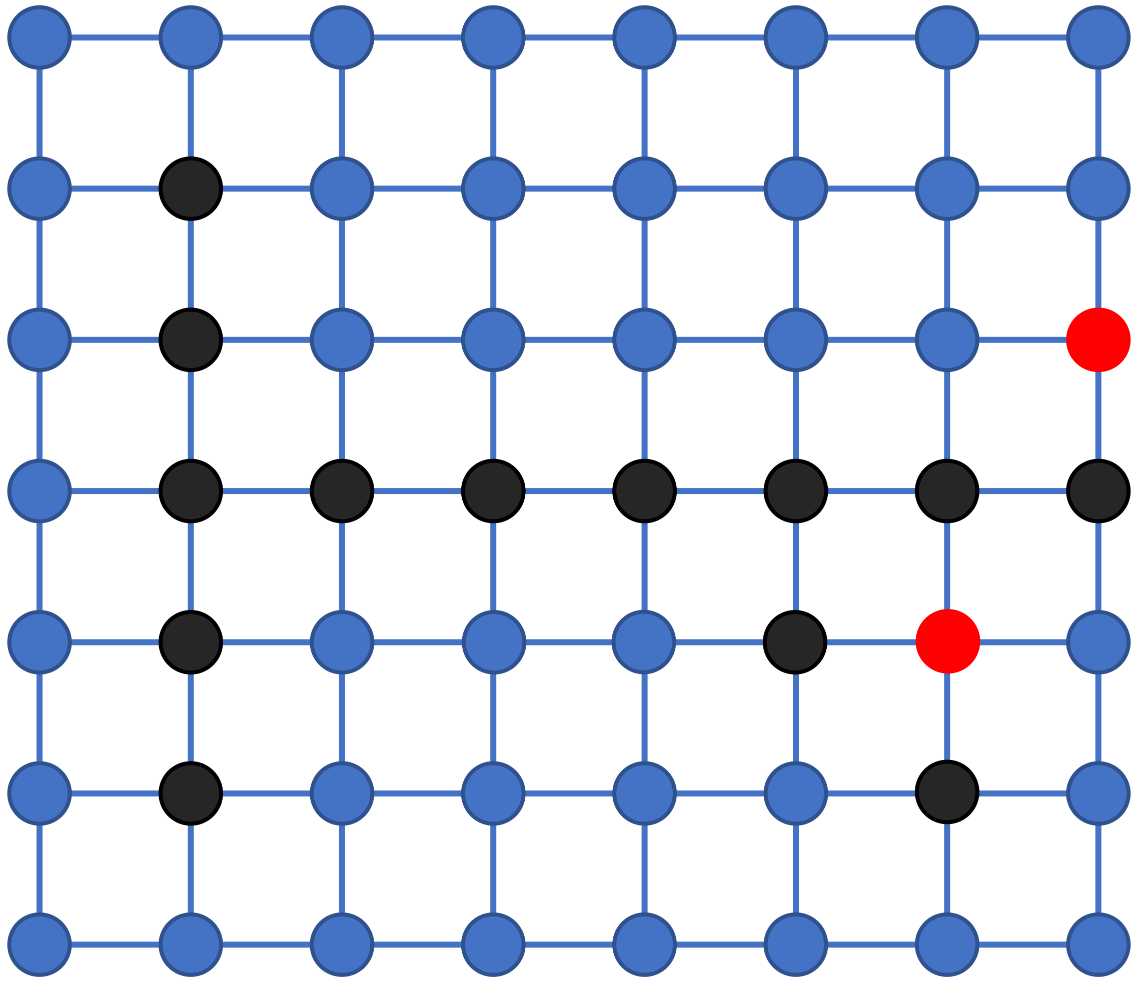
Consider the same network as in Question 1.1. Assume that some of the nodes crashed at the beginning of the algorithm such that any two correct processes are still connected through at least one path of correct processes.

Let l be the length of the longest shortest path between any pair of nodes in the grid; i.e., l is the number of edges between those two nodes which are "farthest away" from each other. If there are no failures, l is the distance between two corners, i.e. $l = w + h$.

a) Modify the algorithm from 1.1**b)** to solve consensus in $l + 1$ rounds with this special type of crash failures. Show that your algorithm works correctly; i.e., a node does not terminate before it learned the initial value of all nodes.

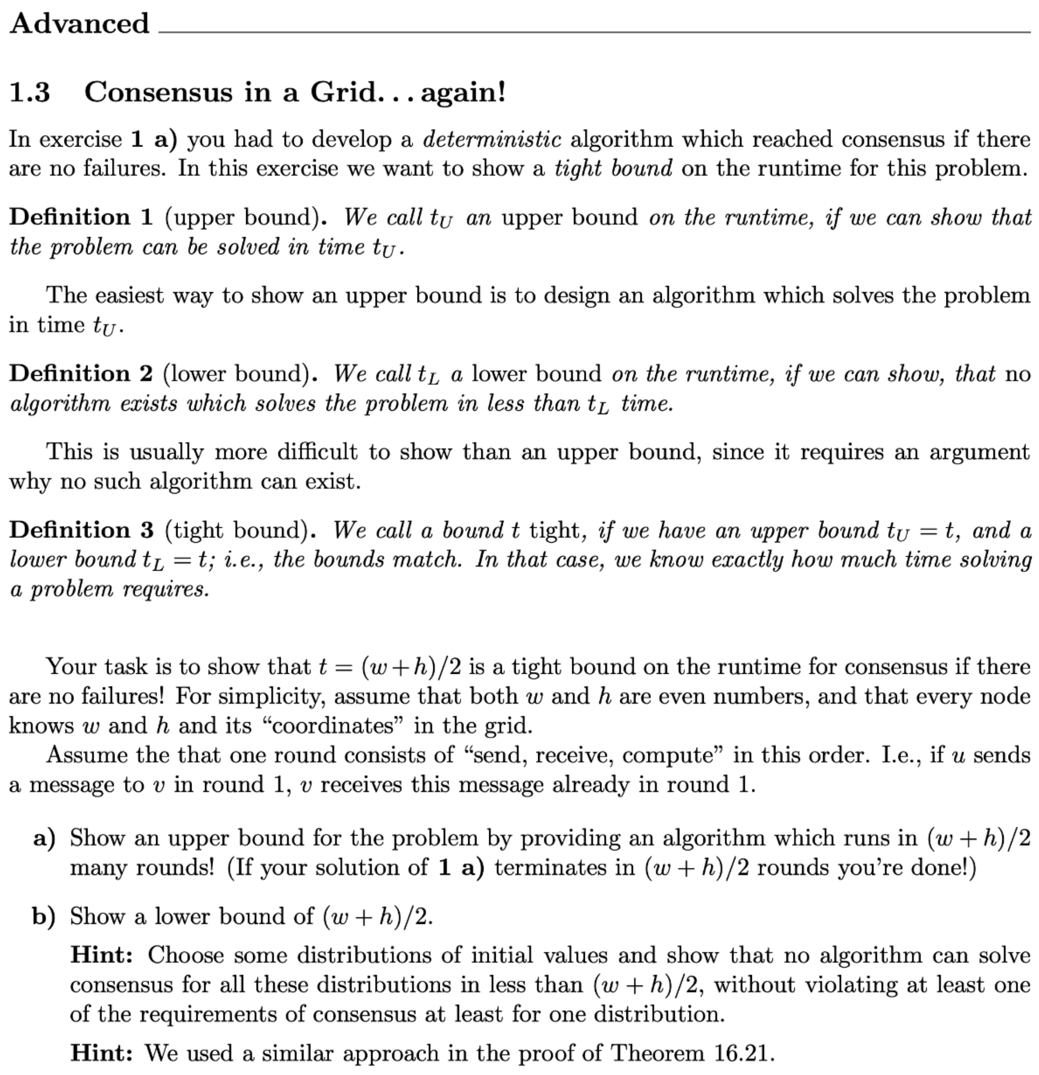
b) As an adversary you are allowed to crash up to $w + h$ many nodes at the beginning of the algorithm. Let $w = 7, h = 6$. What is the largest l you can achieve?

c) Assume that you run the algorithm with any type of crash failures; i.e, nodes can crash at any time during the execution. Show that with such failures the algorithm does not always work correctly anymore, by giving an execution and a failure pattern in which some nodes terminate too early!



ETH zürich Systems@ETH zürich Distributed Computing

Assignment Preview



Advanced

1.3 Consensus in a Grid. . . again!

In exercise **1 a)** you had to develop a *deterministic* algorithm which reached consensus if there are no failures. In this exercise we want to show a *tight bound* on the runtime for this problem.

Definition 1 (upper bound)**.** *We call t_U an* upper bound *on the runtime, if we can show that the problem can be solved in time t_U.*

The easiest way to show an upper bound is to design an algorithm which solves the problem in time t_U.

Definition 2 (lower bound)**.** *We call t_L a* lower bound *on the runtime, if we can show, that* no *algorithm exists which solves the problem in less than t_L time.*

This is usually more difficult to show than an upper bound, since it requires an argument why no such algorithm can exist.

Definition 3 (tight bound)**.** *We call a bound t* tight*, if we have an upper bound $t_U = t$, and a lower bound $t_L = t$; i.e., the bounds match. In that case, we know exactly how much time solving a problem requires.*

Your task is to show that $t = (w + h)/2$ is a tight bound on the runtime for consensus if there are no failures! For simplicity, assume that both w and h are even numbers, and that every node knows w and h and its "coordinates" in the grid.

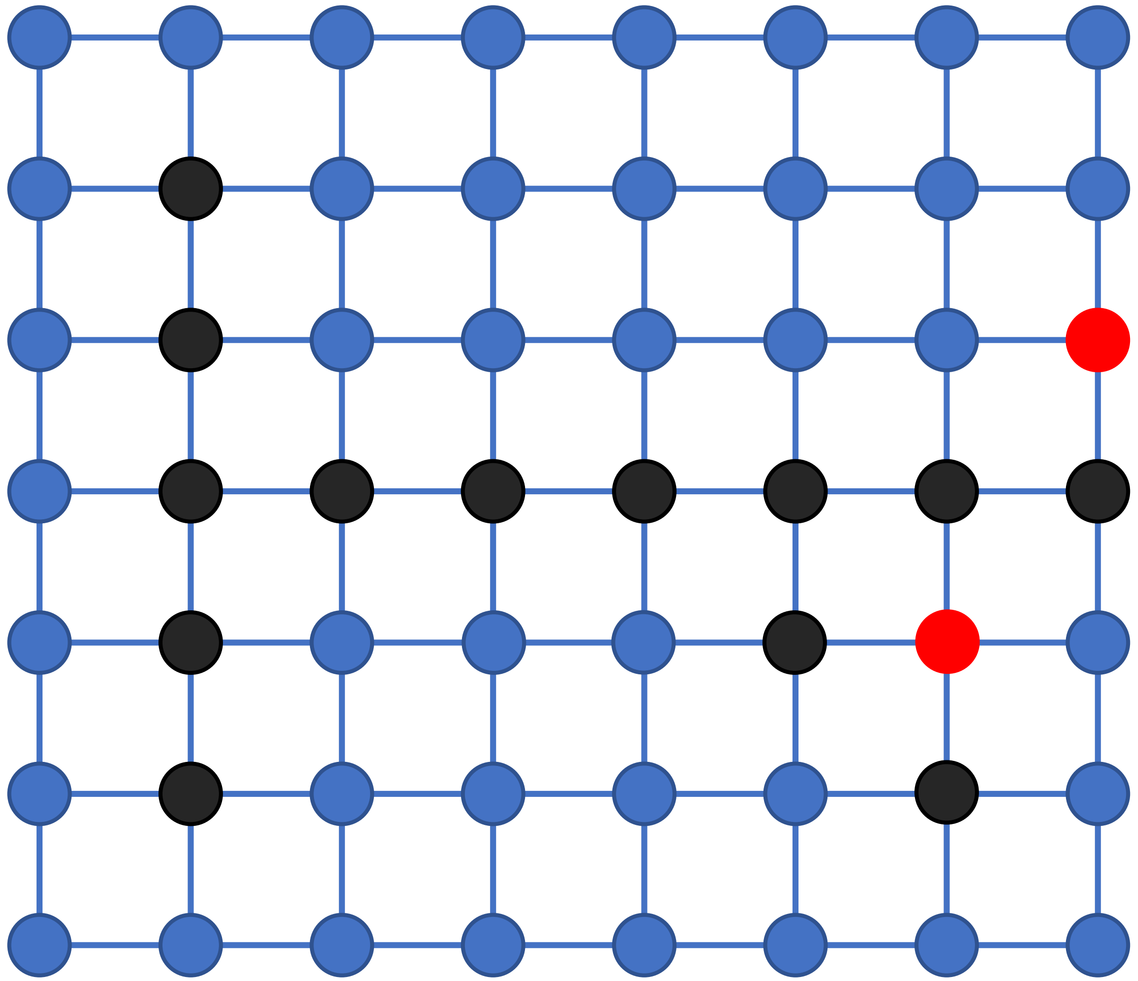
Assume the that one round consists of "send, receive, compute" in this order. I.e., if u sends a message to v in round 1, v receives this message already in round 1.

a) Show an upper bound for the problem by providing an algorithm which runs in $(w + h)/2$ many rounds! (If your solution of **1 a)** terminates in $(w + h)/2$ rounds you're done!)

b) Show a lower bound of $(w + h)/2$.

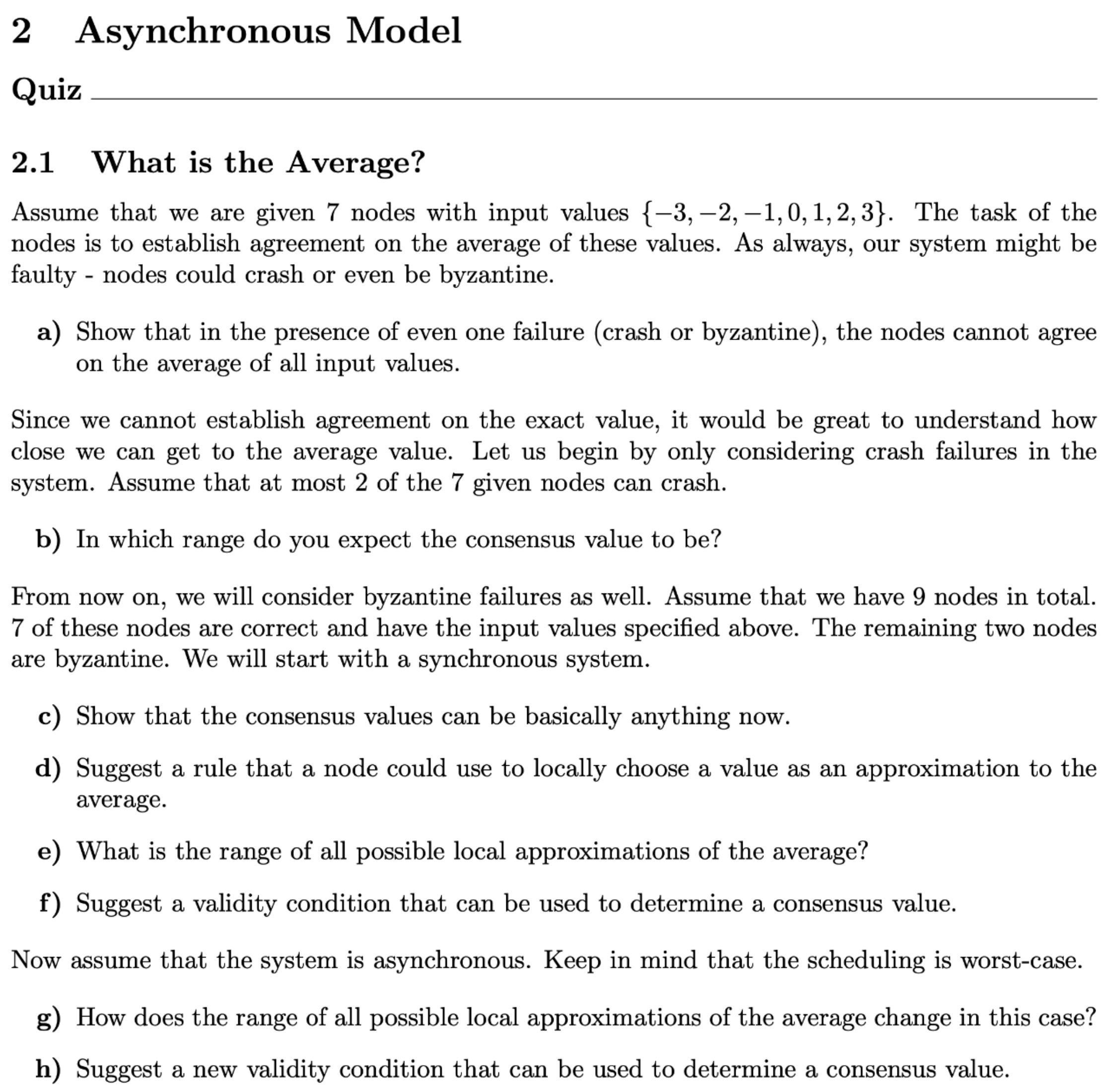
Hint: Choose some distributions of initial values and show that no algorithm can solve consensus for all these distributions in less than $(w + h)/2$, without violating at least one of the requirements of consensus at least for one distribution.

Hint: We used a similar approach in the proof of Theorem 16.21.



ETH zürich Systems@ETH zürich Distributed Computing

Assignment Preview



2 Asynchronous Model

Quiz

2.1 What is the Average?

Assume that we are given 7 nodes with input values $\{-3,-2,-1,0,1,2,3\}$. The task of the nodes is to establish agreement on the average of these values. As always, our system might be faulty - nodes could crash or even be byzantine.

a) Show that in the presence of even one failure (crash or byzantine), the nodes cannot agree on the average of all input values.

Since we cannot establish agreement on the exact value, it would be great to understand how close we can get to the average value. Let us begin by only considering crash failures in the system. Assume that at most 2 of the 7 given nodes can crash.

b) In which range do you expect the consensus value to be?

From now on, we will consider byzantine failures as well. Assume that we have 9 nodes in total. 7 of these nodes are correct and have the input values specified above. The remaining two nodes are byzantine. We will start with a synchronous system.

c) Show that the consensus values can be basically anything now.

d) Suggest a rule that a node could use to locally choose a value as an approximation to the average.

e) What is the range of all possible local approximations of the average?

f) Suggest a validity condition that can be used to determine a consensus value.

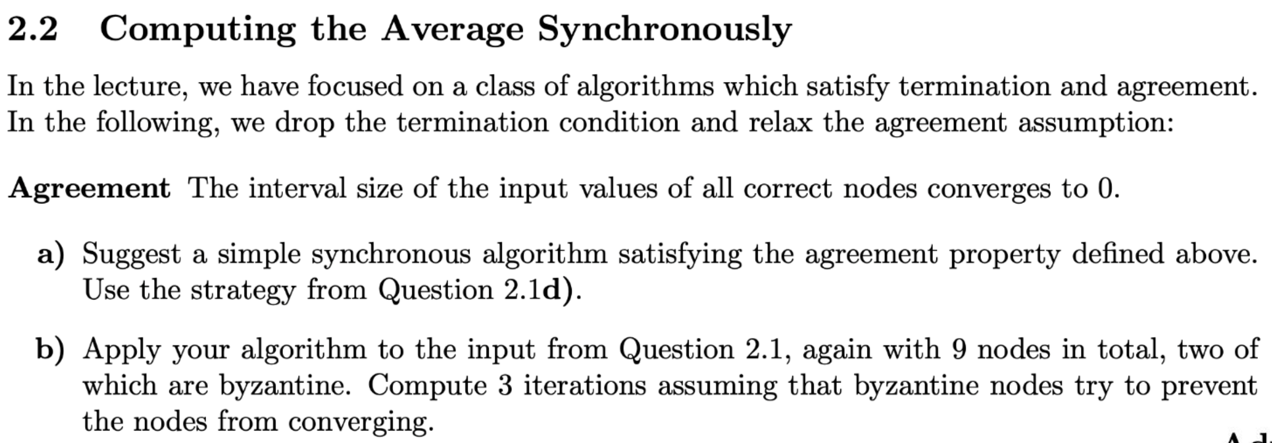
Now assume that the system is asynchronous. Keep in mind that the scheduling is worst-case.

g) How does the range of all possible local approximations of the average change in this case?

h) Suggest a new validity condition that can be used to determine a consensus value.

ETHzürich Systems@ETHzürich Distributed Computing

Assignment Preview



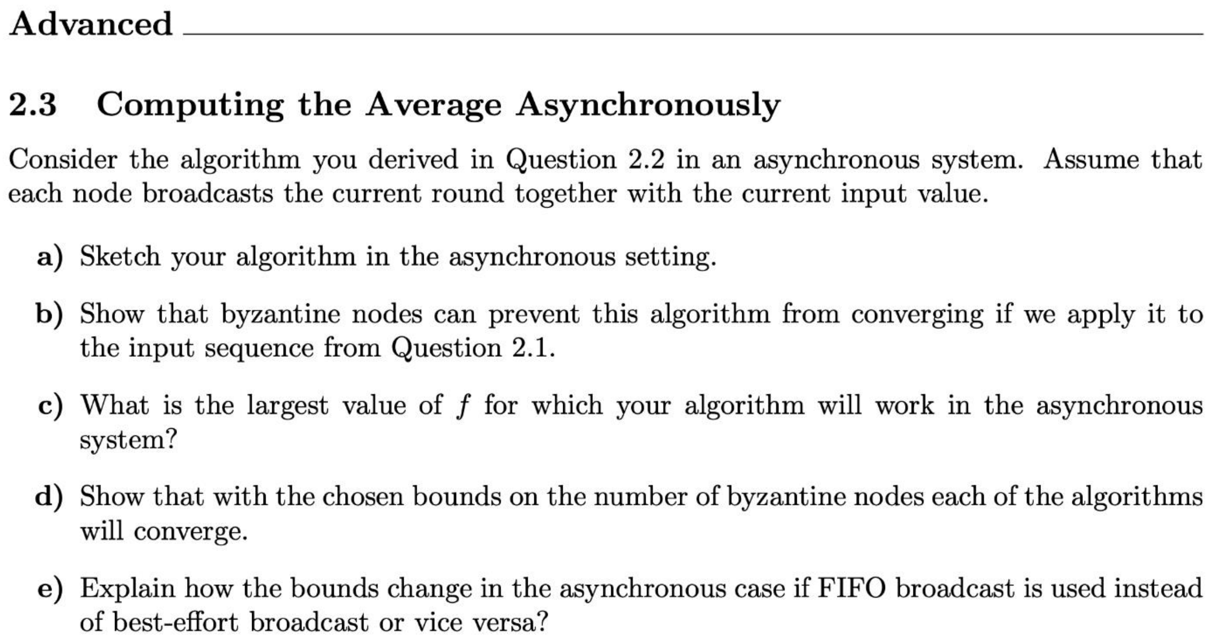
2.2 Computing the Average Synchronously

In the lecture, we have focused on a class of algorithms which satisfy termination and agreement. In the following, we drop the termination condition and relax the agreement assumption:

Agreement The interval size of the input values of all correct nodes converges to 0.

a) Suggest a simple synchronous algorithm satisfying the agreement property defined above. Use the strategy from Question 2.1**d)**.

b) Apply your algorithm to the input from Question 2.1, again with 9 nodes in total, two of which are byzantine. Compute 3 iterations assuming that byzantine nodes try to prevent the nodes from converging.



Advanced

2.3 Computing the Average Asynchronously

Consider the algorithm you derived in Question 2.2 in an asynchronous system. Assume that each node broadcasts the current round together with the current input value.

a) Sketch your algorithm in the asynchronous setting.

b) Show that byzantine nodes can prevent this algorithm from converging if we apply it to the input sequence from Question 2.1.

c) What is the largest value of f for which your algorithm will work in the asynchronous system?

d) Show that with the chosen bounds on the number of byzantine nodes each of the algorithms will converge.

e) Explain how the bounds change in the asynchronous case if FIFO broadcast is used instead of best-effort broadcast or vice versa?