Distributed Systems

Exercise Session 1

Selma Steinhoff: selmas@ethz.ch

Disclaimer: these are not official slides, there might be mistakes, treat them this way

Last exercise

Both clients start with the same initial ticket numbers T_A = T_B and timeouts A = B . Assume that both clients start at T_0 . What will happen?

Last exercise

Possible worst-case scenario:

- *all clients start their attempt to execute a command (approximately) at the same time, use the same timeout and the same initial ticket number.*
- *Possible that two clients always invalidate each others tickets, no client ever succeeds*

Last exercise

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 $\texttt{ask}(\mathrm{t})$ message with $t \leq T_{\max}$, the server can reply with an explicit $\texttt{nack}(T_{\max})$ in Phase 1, instead of just ignoring the $\texttt{ask}(t)$ 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!

Last exercise

Answers:

a) *No, it is not beneficial since same scenario can occur: two clients get nack(100) and then both at the same time try 101 etc.*

b) *Exponential backoff*

Last exercise

2.2 Deterministic Random Consensus?!

Algorithm 8.15 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! (See algorithm below, 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: decided = false

4: Broadcast `myValue`(v_i, round)

5: **while** true **do**

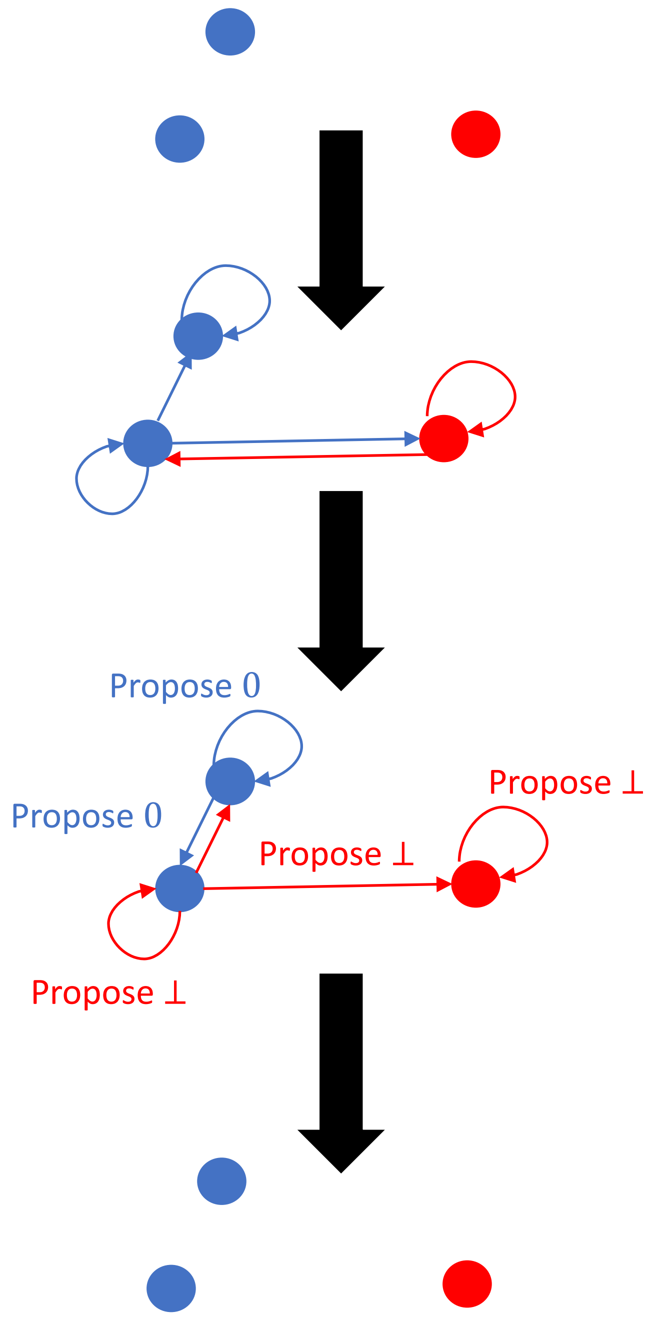
Propose

6: Wait until a majority of `myValue` messages of current round arrived
7: **if** all messages contain the same value v **then**
8: Broadcast `propose`(v, round)
9: **else**
10: Broadcast `propose`($\perp$, round)
11: **end if**

12: **if** decided **then**
13: Broadcast `myValue`(v_i, round+1)
14: Decide for v_i and terminate
15: **end if**

Adapt

16: Wait until a majority of `propose` messages of current round arrived
17: **if** all messages propose the same value v **then**
18: $v_i = v$
19: decided = true
20: **else if** there is at least one proposal for v **then**
21: $v_i = v$
22: **else**
23: Choose $v_i = 1$
24: **end if**
25: round = round + 1
26: Broadcast `myValue`(v_i, round)
27: **end while**



Last exercise

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.

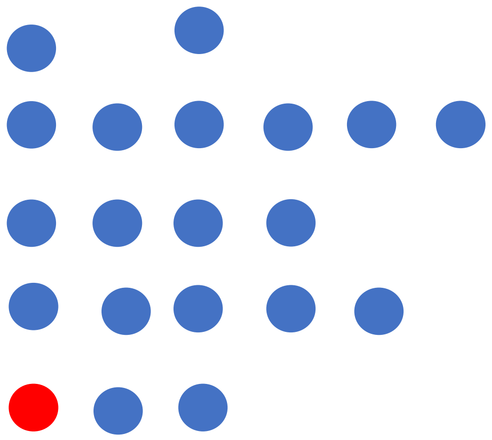
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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. E.g., at time 0, u_1 can send a message to u_2, at time 1 a message to u_3, and so on. However, u_1 cannot send a message to both u_2 and u_3 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!



Last exercise

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.

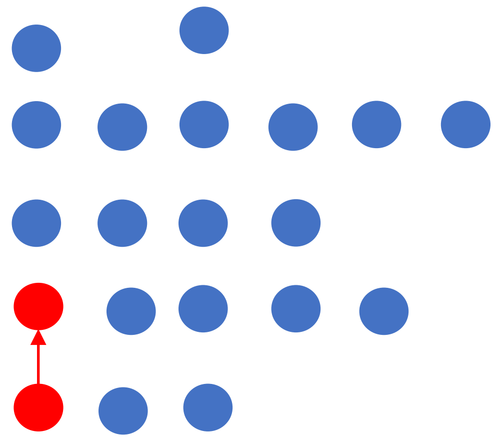
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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. E.g., at time 0, u_1 can send a message to u_2, at time 1 a message to u_3, and so on. However, u_1 cannot send a message to both u_2 and u_3 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!



Last exercise

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.

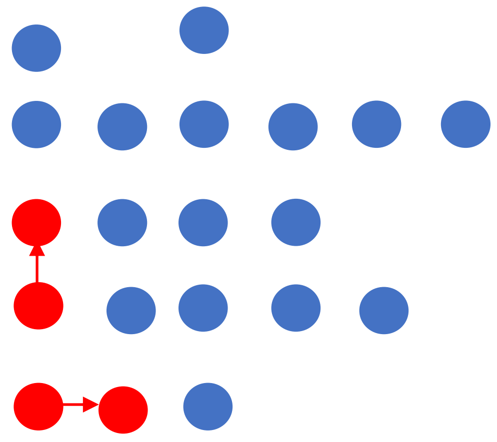
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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. E.g., at time 0, u_1 can send a message to u_2, at time 1 a message to u_3, and so on. However, u_1 cannot send a message to both u_2 and u_3 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!



Last exercise

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.

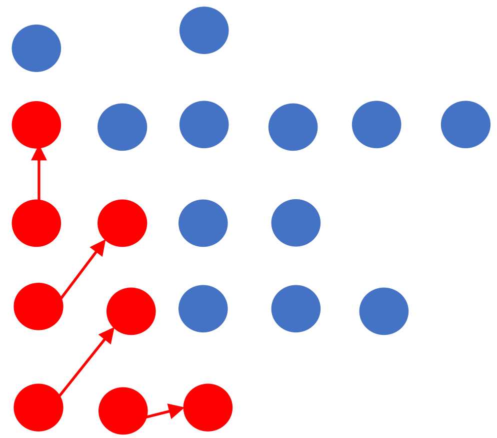
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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. E.g., at time 0, u_1 can send a message to u_2, at time 1 a message to u_3, and so on. However, u_1 cannot send a message to both u_2 and u_3 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!



Last exercise

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.

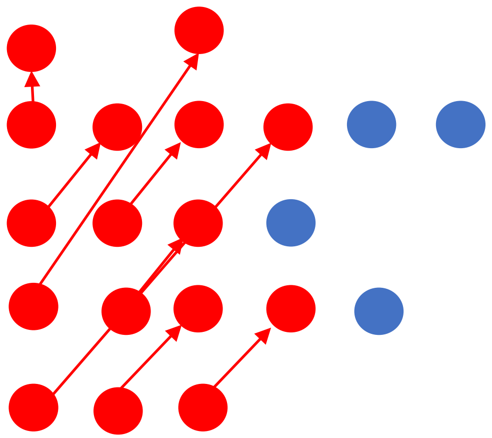
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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. E.g., at time 0, u_1 can send a message to u_2, at time 1 a message to u_3, and so on. However, u_1 cannot send a message to both u_2 and u_3 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!



Last exercise

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.

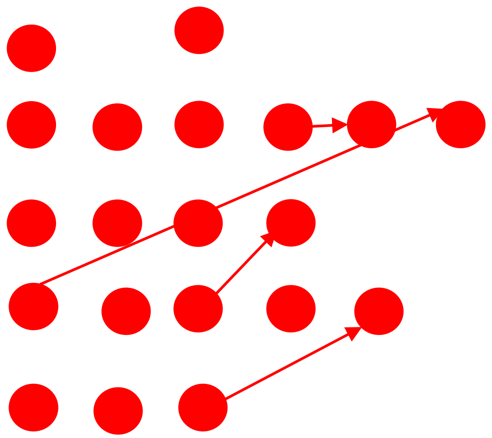
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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. E.g., at time 0, u_1 can send a message to u_2, at time 1 a message to u_3, and so on. However, u_1 cannot send a message to both u_2 and u_3 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!



Byzantine nodes

- Node which has arbitrary behavior
- So it can:
    - Not send messages
    - Sending different messages to different nodes
    - Sending wrong messages
    - Lie about input value
- If an algorithm works with f byzantine nodes, it is f-resilient

Consensus with byzantine nodes?

- Termination
- Agreement
- Validity ?

Different Validities

- Any-input validity:
  - The decision value must be input of any node
  - That includes byzantine nodes who might lie about their input values
- Correct-input validity:
  - The decision value must be input of a correct node
  - Difficult because byzantine node following the protocol are indistinguishable
- All-same validity:
  - if all correct nodes start with the same value, the decision must be that value
- Median validity:
  - If input values are orderable, byzantine outliers can be prevented by agreeing on a value close to the median value of the correct nodes

King Algorithm
(synchronous byzantine agreement)

Idea:
If not all correct input nodes have the same value, decide on value of one correct input node. Ensure this by doing f+1 rounds, since there must be at least one correct input node.

Algorithm 11.14 King Algorithm (for $f < n/3$)

x = my input value
for phase $= 1$ to $f + 1$ **do**
Round 1
  Broadcast `value`(x)
Round 2
  if some `value`(y) received at least $n - f$ times **then**
    Broadcast `propose`(y)
  end if
  if some `propose`(z) received more than f times **then**
    $x = z$
  end if
Round 3
  Let node v_i be the predefined king of this phase i
  The king v_i broadcasts its current value w
  if received strictly less than $n - f$ `propose`(y) **then**
    $x = w$
  end if
end for

Do until at least one correct input node

Send out own value

If some value received from all nodes but byzantine ones (or at least ((n – f)- f) correct ones), propose that value

If some value proposed by at least one correct node, set your value to that value

King of this phase broadcasts its value

If didn’t get propose from all nodes but byzantine ones (or at least ((n – f)- f) correct ones), set your value to value of king

King Algorithm (synchronous byzantine agreement)

Algorithm 11.14 King Algorithm (for $f < n/3$)

x = my input value
for phase = 1 to $f + 1$ **do**

Round 1

Broadcast `value`(x)

Round 2

if some `value`(y) received at least $n - f$ times **then**
Broadcast `propose`(y)
end if
if some `propose`(z) received more than f times **then**
$x = z$
end if

Round 3

Let node v_i be the predefined king of this phase i
The king v_i broadcasts its current value w
if received strictly less than $n - f$ `propose`(y) **then**
$x = w$
end if
end for

King Algorithm

(synchronous byzantine agreement)

Algorithm 11.14 King Algorithm (for $f < n/3$)

```
x = my input value
for phase = 1 to f + 1 do
    Round 1
    Broadcast value(x)
    Round 2
    if some value(y) received at least n − f times then
        Broadcast propose(y)
    end if
    if some propose(z) received more than f times then
        x = z
    end if
    Round 3
    Let node v_i be the predefined king of this phase i
    The king v_i broadcasts its current value w
    if received strictly less than n − f propose(y) then
        x = w
    end if
end for
```

Why f+1?

- Because there are f byzantine nodes, at least one of the kings will be a correct node

King Algorithm
(synchronous byzantine agreement)

Why n-f?

- Because there are n-f correct nodes, so we can’t wait for more.
- Ensures only one proposal:
  - If one node sees n-f values v, then every other node sees at least n-2f times v.
  - Because n- (n-2f) = 2f < n-f, there can be no proposal for another value.

Algorithm 11.14 King Algorithm (for $f < n/3$)

```
x = my input value
for phase = 1 to f + 1 do
    Round 1
    Broadcast value(x)
    Round 2
    if some value(y) received at least n − f times then
        Broadcast propose(y)
    end if
    if some propose(z) received more than f times then
        x = z
    end if
    Round 3
    Let node v_i be the predefined king of this phase i
    The king v_i broadcasts its current value w
    if received strictly less than n − f propose(y) then
        x = w
    end if
end for
```

King Algorithm
(synchronous byzantine agreement)

Why n-f propose messages?

- Similar as for n-f broadcast messages. We can wait for at most n-f ones because those are the correct nodes, and we have to wait for at least f+1 ones.

After a correct king, the correct nodes will not change their values anymore! Why?

- If all of them have less than n-f propose messages, all correct nodes will have the king value and then “all same validity” holds.
- If one does not adapt, this means that it got n-f propose messages. This means, every other message got at least n-f-f > f propose messages, so it adapted its value to the propose. So the king also adapted it’s value and again all nodes have the same value.

Algorithm 11.14 King Algorithm (for $f < n/3$)

x = my input value
for phase $= 1$ to $f + 1$ **do**
  Round 1
  Broadcast `value`(x)
  Round 2
  if some `value`(y) received at least $n - f$ times **then**
    Broadcast `propose`(y)
  end if
  if some `propose`(z) received more than f times **then**
    $x = z$
  end if
  Round 3
  Let node v_i be the predefined king of this phase i
  The king v_i broadcasts its current value w
  if received strictly less than $n - f$ `propose`(y) **then**
    $x = w$
  end if
end for

King Algorithm

(synchronous byzantine agreement)

- Does it solve byzantine agreement?
  - Validity: All same validity!
  - Agreement: They agree at least after the first correct king.
  - Termination: After (f+1)*3 rounds

Algorithm 11.14 King Algorithm (for $f < n/3$)

x = my input value
for phase $= 1$ to $f+1$ **do**
  Round 1
  Broadcast value(x)
  Round 2
  if some value(y) received at least $n - f$ times **then**
    Broadcast propose(y)
  end if
  if some propose(z) received more than f times **then**
    $x = z$
  end if
  Round 3
  Let node v_i be the predefined king of this phase i
  The king v_i broadcasts its current value w
  if received strictly less than $n - f$ propose(y) **then**
    $x = w$
  end if
end for

Asynchronous Byzantine Agreement

- Assumption: Messages do not need to arrive at the same time anymore. They have variable delays.

-> Also works, but is a lot more complicated.

-> Algorithm in script is proof of concept, so don't worry about it too much.

->Asynchrony changes messages you have to wait for, but not principle

- Problem: slow! (exponential runtime)

Algorithm 11.21 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/10$)

$x_i \in \{0, 1\}$ ◁ input bit

r = 1 ◁ round

Broadcast own value

Do until converged

Wait for enough messages

If big enough amount agrees, decide and terminate

If some agree, adapt your value but don't decide yet

If no popular value, decide randomly

Broadcast own value

until decided (see Line 8)

decision = x_i

Asynchronous Byzantine Agreement with oracle

- Now, if no popular value, all correct nodes will decide on same oracle value.
- Constant runtime
- Problem: oracle does not exist

Algorithm 11.21 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/10$)

1: $x_i \in \{0, 1\}$ $\triangleleft$ input bit
2: r = 1 $\triangleleft$ round

Broadcast own value

Do until converged

Wait for enough messages

If big enough amount agrees, decide and terminate

If some agree, adapt your value but don't decide yet

If no popular value, ask oracle

Broadcast own value

16: **until** decided (see Line 8)
17: decision = x_i

Asynchronous Byzantine Agreement with random bitstring

- New idea: generate a random bitstring and take next value of bitstring instead of asking oracle
- Problem: byzantine nodes know "random" value and can adapt their behavior

Algorithm 11.21 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/10$)

$x_i \in \{0, 1\}$ ◁ input bit

r = 1 ◁ round

Broadcast own value

Do until converged

Wait for enough messages

If big enough amount agrees, decide and terminate

If some agree, adapt your value but don't decide yet

If no popular value, ask look at bitstring

Broadcast own value

until decided (see Line 8)

decision = x_i

Reliable Broadcast

Best effort broadcast	• Best effort broadcast ensures that a message that is sent from a correct node v to another correct node w will be received and accepted by w
Reliable broadcast	• Reliable broadcast ensures that the nodes eventually agree on all accepted messages. That is, if a correct node v considers message m as accepted, then every other node will eventually consider message m as accepted.
FIFO (reliable) broadcast	• The FIFO (reliable) broadcast defines an order in which the messages are accepted in the system. If a node u broadcasts message m1 before m2, then any node v will accept the message m1 first.
Atomic broadcast	• Atomic broadcast makes sure that all messages are always received in the same order. So for two random nodes u1 and u2 and two random messages m1 and m2, if u1 sees m1 first, u2 will also see m1 first.

Reliable Broadcast

Algorithm 4.15 Asynchronous Reliable Broadcast (code for node u)

Broadcast own value

If message received from node directly, broadcast it together with your own name

If you do not get message from node directly, but from a reasonable amount of others also broadcast with own name

If you get enough forwarded messages, accept message

Reliable Broadcast

Algorithm 4.15 Asynchronous Reliable Broadcast (code for node u)

Broadcast own message $\texttt{msg}(u)$
if received $\texttt{msg}(v)$ from node v **then**
    Broadcast echo$(u, \texttt{msg}(v))$
end if
if received echo$(w, \texttt{msg}(v))$ from $n - 2f$ nodes w but not $\texttt{msg}(v)$ **then**
    Broadcast echo$(u, \texttt{msg}(v))$
end if
if received echo$(w, \texttt{msg}(v))$ from $n - f$ nodes w **then**
    Accept$(\texttt{msg}(v))$
end if

Reliable Broadcast - Properties

correct node **broadcasts** -> eventually **all** correct node accept

correct node NOT broadcasts -> **not** accepted by correct node

correct node **accepts** message -> eventually **all** correct node accept

FIFO Broadcast

Algorithm 12.20 FIFO Reliable Broadcast (code for node u)

```
Broadcast own round r message msg(u, r)
if received first message msg(v, r) from node v for round r then
    Broadcast echo(u, msg(v, r))
end if
if not echoed any msg'(v, r) before then
    if received echo(w, msg(v, r)) from f + 1 nodes w but not msg(v, r) then
        Broadcast echo(u, msg(v, r))
    end if
end if
if received echo(w, msg(v, r)) from n − f nodes w then
    if accepted msg(v, r − 1) then
        Accept(msg(v, r))
    end if
end if
```

Quiz

- Can byzantine nodes collaborate?
  - *Yes*
- Can byzantine nodes forge a sender address?
  - *No, otherwise one could impersonate all correct ones.*
- In all-same validity, is the decision value restricted if not all nodes start with the same value?
  - *No*
- Can there be any algorithm that can solve synchronous byzantine agreement in less than f+1 rounds?
  - *No, because the node with the smallest value can propagate its value to one further node and then crash f times. So in the last round, one node knows about the new value but has no chance to propagate it.*

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?

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.