Computer Systems

Exercise Session 6

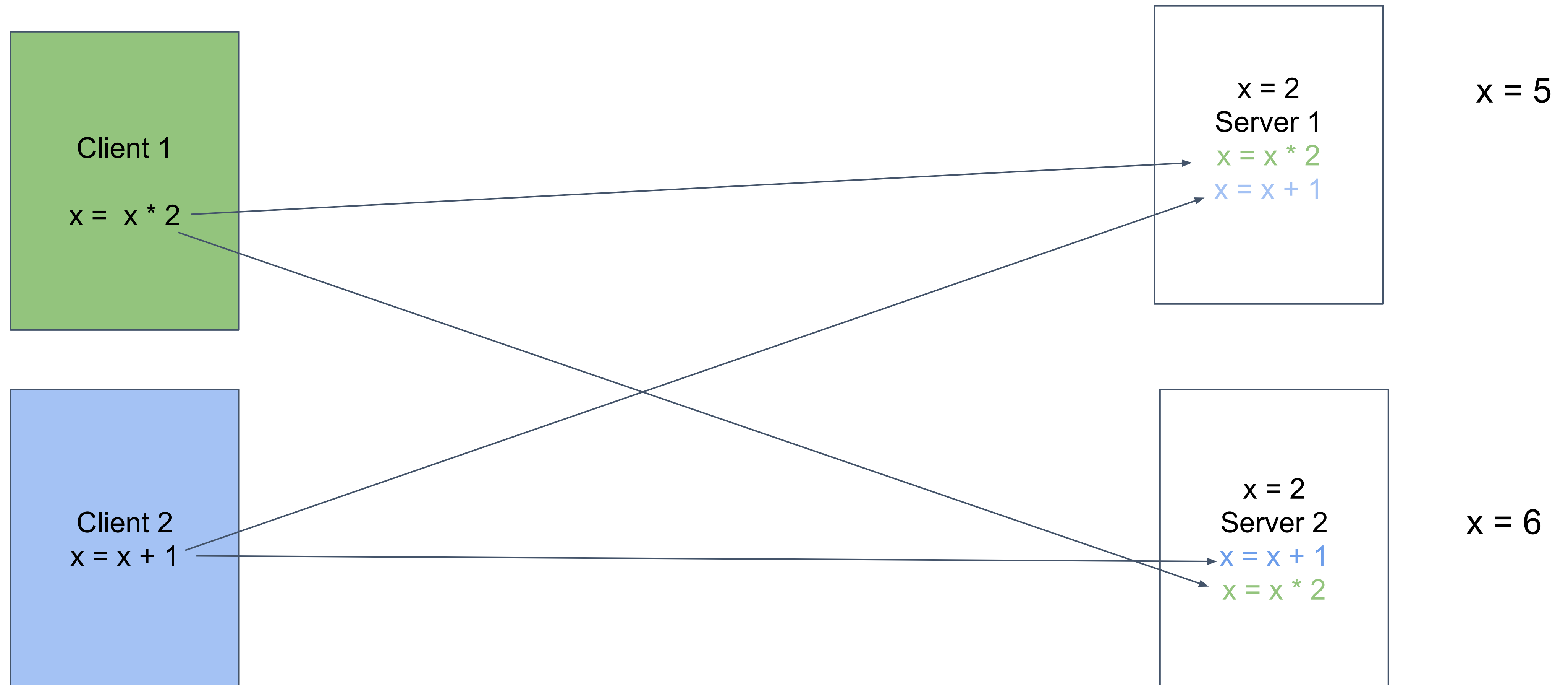
Fault-Tolerance in Distributed Systems

- Problem setup:
  - N distributed, trusted nodes
  - All-to-all asynchronous messages
  - Variable message transmission time
  - Messages may be lost
  - Some nodes may crash
- One fundamental goal: *state replication*
  - Same sequence of commands in the same order

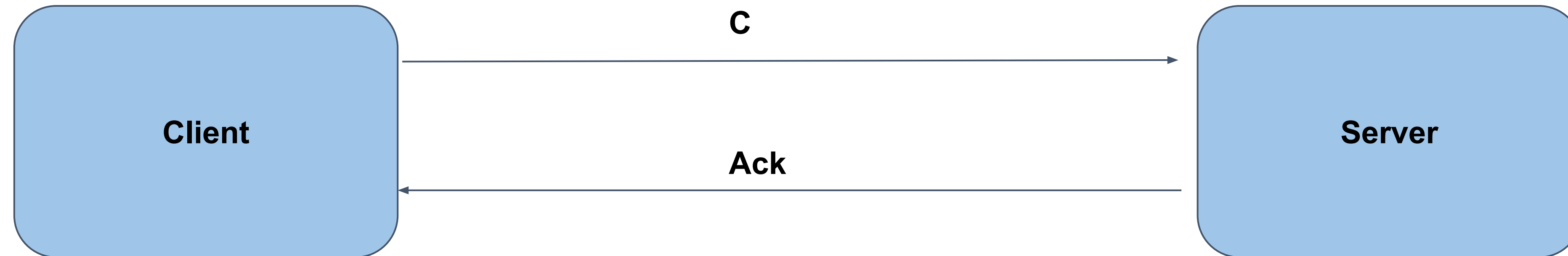
Fault-Tolerance in Distributed Systems

- One fundamental goal: *state replication*
  - Same sequence of commands in the same order
- For now, only one command
  - Repetitions of one command algorithms can solve full state replication

Fault-Tolerance in Distributed Systems

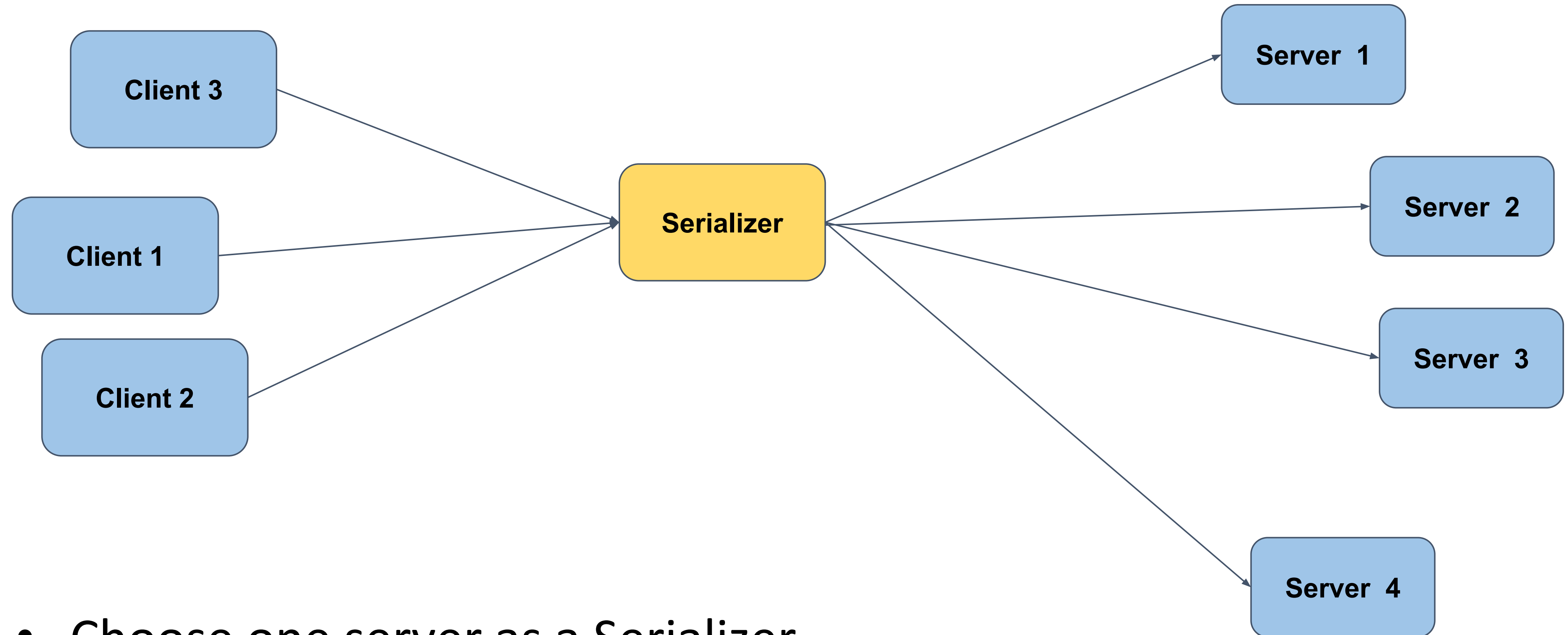


First approaches



- Server sends acknowledgement message
  - Reasonable with one client

First approaches



- Choose one server as a Serializer
  - Single point of failure

First approaches

Algorithm 15.10 Two-Phase Protocol

Phase 1

1: Client asks all servers for the lock

Phase 2

2: **if** client receives lock from every server **then**
3: Client sends command reliably to each server, and gives the lock back
4: **else**
5: Clients gives the received locks back
6: Client waits, and then starts with Phase 1 again
7: **end if**

Paxos – main ideas

- Servers hand out **tickets**
  - ”Weak lock”, which can be overwritten by a later ticket
- Only requires the **majority** of servers to agree
  - Already ensures that there is at most one accepted command
- Servers **notify clients** about their stored command
  - Client can then switch to supporting this stored command

Main steps of Paxos

- Clients asks for a specific ticket t
- Server only issues ticket t if t is the largest ticket requested so far
- If client receives majority of tickets, it proposes a command
- When a server receives a proposal, if the ticket of the client is still valid, the server stores the command and notifies the client
- If a majority of servers store the command, the client notifies all servers to execute the command

Algorithm 7.13 Paxos

Client (Proposer)	Server (Acceptor)
Initialization	
c ◁ *command to execute*	$T_{max} = 0$ ◁ *largest issued ticket*
$t = 0$ ◁ *ticket number to try*	
	$C = \bot$ ◁ *stored command*
	$T_{store} = 0$ ◁ *ticket used to store C*
Phase 1	
1: $t = t + 1$	
2: Ask all servers for ticket t	
	3: **if** $t > T_{max}$ **then**
	4: $T_{max} = t$
	5: Answer with ok(T_{store}, C)
	6: **end if**
Phase 2	
7: **if** a majority answers ok **then**	
8: Pick (T_{store}, C) with largest T_{store}	
9: **if** $T_{store} > 0$ **then**	
10: $c = C$	
11: **end if**	
12: Send propose(t, c) to same majority	
13: **end if**	
	14: **if** $t = T_{max}$ **then**
	15: $C = c$
	16: $T_{store} = t$
	17: Answer success
	18: **end if**
Phase 3	
19: **if** a majority answers success **then**	
20: Send execute(c) to every server	
21: **end if**	

Client ask for a specific ticket t

Server only issues ticket t if t is the largest ticket requested so far

If client receives majority of tickets, it proposes a command

When server receives proposal, if the ticket is still valid the server stores the command and notifies the client

If a majority of servers store the command the client notifies all servers to execute the command

Quiz

- At what point is a command c guaranteed to execute on every server?

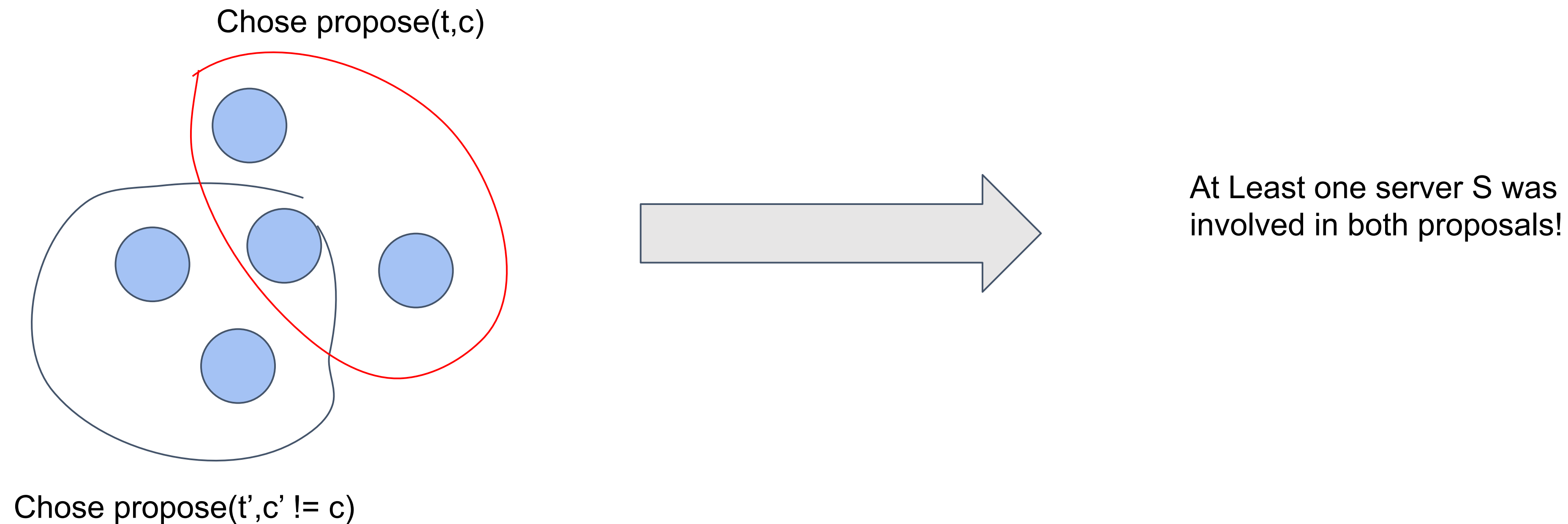
Quiz

- At what point is a command c guaranteed to execute on every server?

As soon as there is a first proposal(t,c) is chosen(stored on a majority of servers), c will be guaranteed to execute since every subsequent proposal will be for c.

Intuition behind proof:

Lemma 15.14. *We call a message* propose(*t*,*c*) *sent by clients on Line 12 a* ***proposal for (t,c)****. A proposal for (t,c) is* **chosen***, if it is stored by a majority of servers (Line 15). For every issued* propose(t',c') *with* $t' > t$ *holds that* $c' = c$*, if there was a chosen* propose(*t*,*c*).



Intuition behind proof:

Lemma 15.14. *We call a message* propose(t,c) *sent by clients on Line 12 a* ***proposal for (t,c)****. A proposal for (t,c) is* **chosen***, if it is stored by a majority of servers (Line 15). For every issued* propose(t',c') *with $t' > t$ holds that $c' = c$, if there was a chosen* propose(t,c).

Quiz

- How many Paxos nodes could crash so that Paxos still works?

Quiz

- How many Paxos nodes could crash so that Paxos still works?

Less than half due to the majority requirements

Quiz

- What do the client nodes need to know about the system for paxos to work?

Quiz

- What do the client nodes need to know about the system for paxos to work?

It needs to know the amount total amount of server nodes on the system to decide whether or not majority is achieved

Paxos

youtube for more intuition(with slight variation in terminology)

Consensus

We want:

- **Agreement:** all (correct) nodes decide for the same value
- **Termination:** all (correct) nodes terminate
- **Validity:** the decision value is the input value of at least one node

Impossibility:

- Consensus cannot be solved ***deterministically*** in the asynchronous model.

Question: Does paxos solve consensus?

Randomized Consensus

Easy cases:

- All inputs are equal (all 0 or 1)
- Almost all input values equal

Otherwise:

- Choose a ***random*** value locally. → expected time O(2^n) until all agree (once)

Algorithm 16.15 Randomized Consensus (assuming $f < n/2$)

```
1: v_i ∈ {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(⊥, 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(v, round + 1)
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 randomly, with Pr[v_i = 0] = Pr[v_i = 1] = 1/2
20:   end if
21:   round = round + 1
22: end while
```

Quiz

- How many node crashes can randomized consensus handle?

Quiz

- How many node crashes can randomized consensus handle?

$f < n/2$ due to majority

Shared coin

Algorithm 16.22 Shared Coin (code for node u)

1: Choose local coin $c_u = 0$ with probability $1/n$, else $c_u = 1$
2: Broadcast `myCoin`(c_u)
3: Wait for $n - f$ coins and store them in the local coin set C_u
4: Broadcast `mySet`(C_u)
5: Wait for $n - f$ coin sets
6: **if** at least one coin is 0 among all coins in the coin sets **then**
7:     return 0
8: **else**
9:     return 1
10: **end if**

Shared coin

- This algorithm implements shared coin for $f < n/3$, study proof in the script!
- Important tools for distributed proofs:
  - Global state : **Configuration** = per node state + in flight messages
- Step : **Transition** : i.e arrival of a message
- Execution: Full tree of all possible orderings of transitions
- A configuration is **bivalent** if the outcome is undecided, otherwise **univalent**
- A **critical configuration** is a configuration that is bivalent and all nodes are univalent

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. If the claim is wrong, 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?

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.

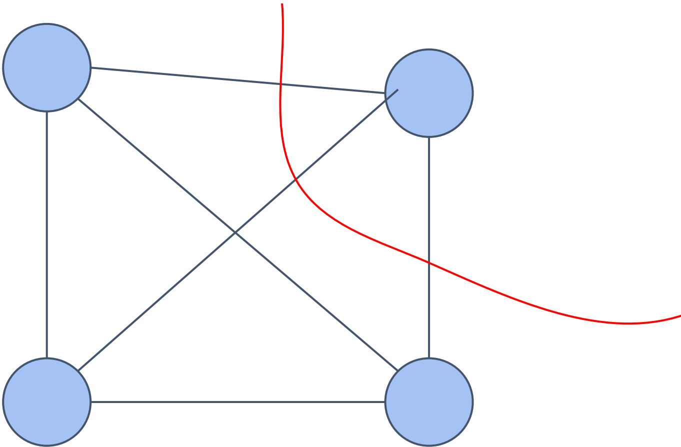
We assume that the network is initially fully connected, i.e., there is an edge between every pair of nodes. Our goal is to solve consensus in such a way, that *all* nodes know the decision.

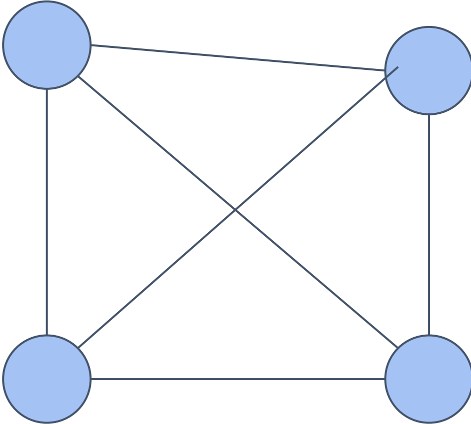
a) What is the smallest f such that consensus might become impossible? (Which edges fail in the worst-case)

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 connected, 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.





1.2 Paxos

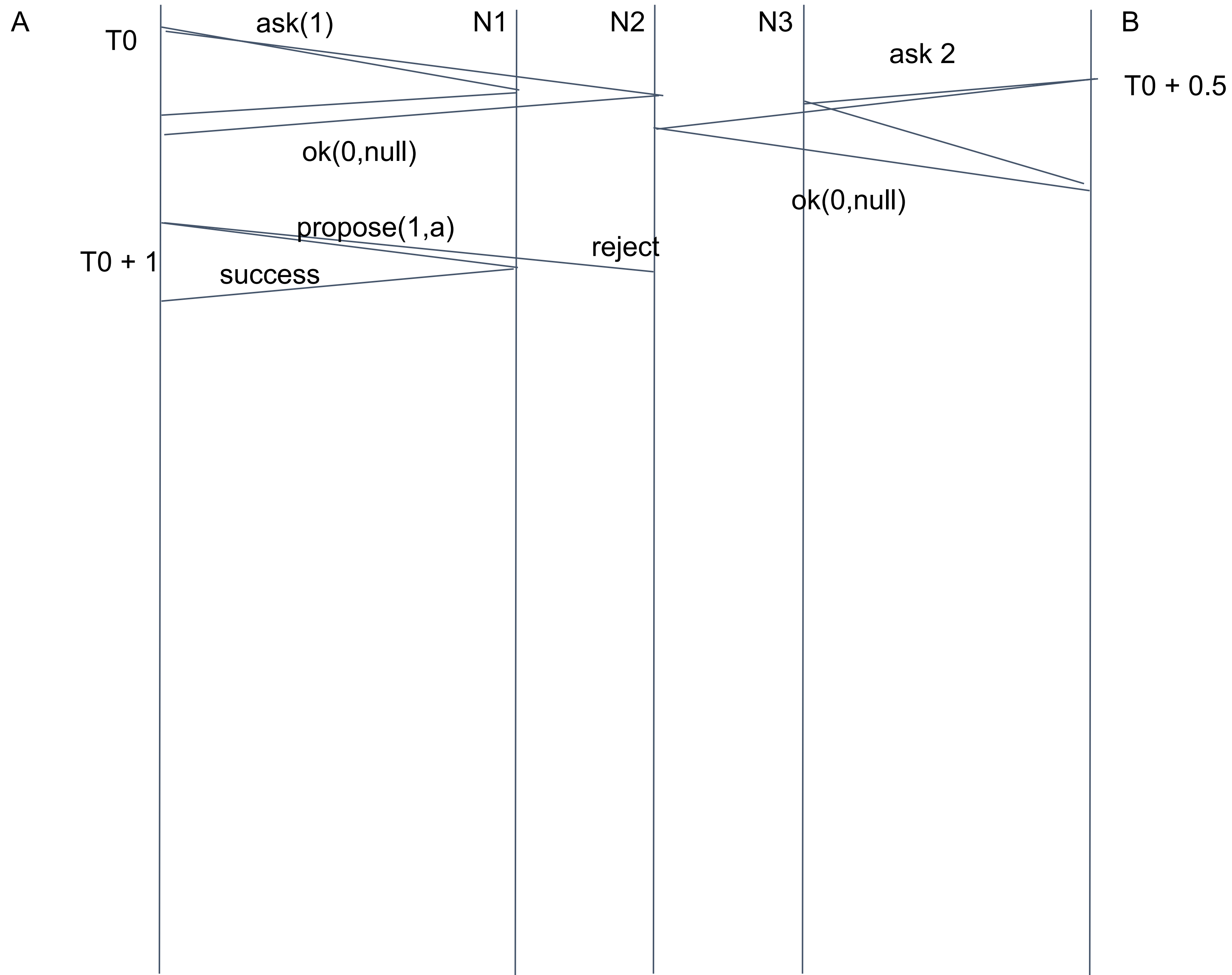
You decide to use Paxos for a system with 3 servers (acceptors), which we call N_1, N_2, N_3. There are two clients (proposers) A and B. The implementation of the acceptors is exactly as defined in the script, see Algorithm 7.13. We extended the code of the proposers, such that they now use explicit timeouts. The algorithm is described below, note in particular Lines 2-4 and 12-14.

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.
 * δ: 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 δ, TicketNumber t) {
Phase 1 ...............................................................
1:  Ask N, N' for ticket t
Phase 2 ...............................................................
2:  Wait for δ seconds
3:  if within these δ seconds, either N or N' has not replied with ok then
4:    return suggestValue(N, N', c, δ, 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 δ seconds
13: if within these δ seconds, either N or N' has not replied with success then
14:   return suggestValue(N, N', c, δ, t + 2)
15: else
16:   Send execute(c) to every server
17:   return c
18: end if
```

a) Assume that two users try to execute a command. One user calls **suggestValue(**N_1, N_2, a, 1, 1**)** on A at time T_0, and a second user calls **suggestValue(**N_2, N_3, b, 2, 2**)** on B at time $T_0 + 0.5sec$.

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 $0.5sec$.



A
N1
N2
N3
B
T0
ask(1)
ask 2
T0 + 0.5
ok(0,null)
ok(0,null)
propose(1,a)
reject
T0 + 1
success