Computer Systems

Exercise Session 3

Last exercise

Create a program which forks itself once. The parent process should output 100 times «I’m the parent and my child's PID is <pid>». The child should output 100 times «I'm the child and my PID is <pid>».
Do the PIDs match?

Answer: No, the PIDs are not matching.

Last exercise

What do you expect to happen here? Please explain what you think will happen.

```
int main(int argc, char **argv){
        while(1) {
                fork();
        }
}
```

Answer: If you execute this code, your computer will be (almost) dead. Every child forks new children in a loop and this new children fork new children in a loop as well. This consumes too many resources in a very short time. Not only memory, but also CPU cycles, page table entries, descriptors...

Last exercise

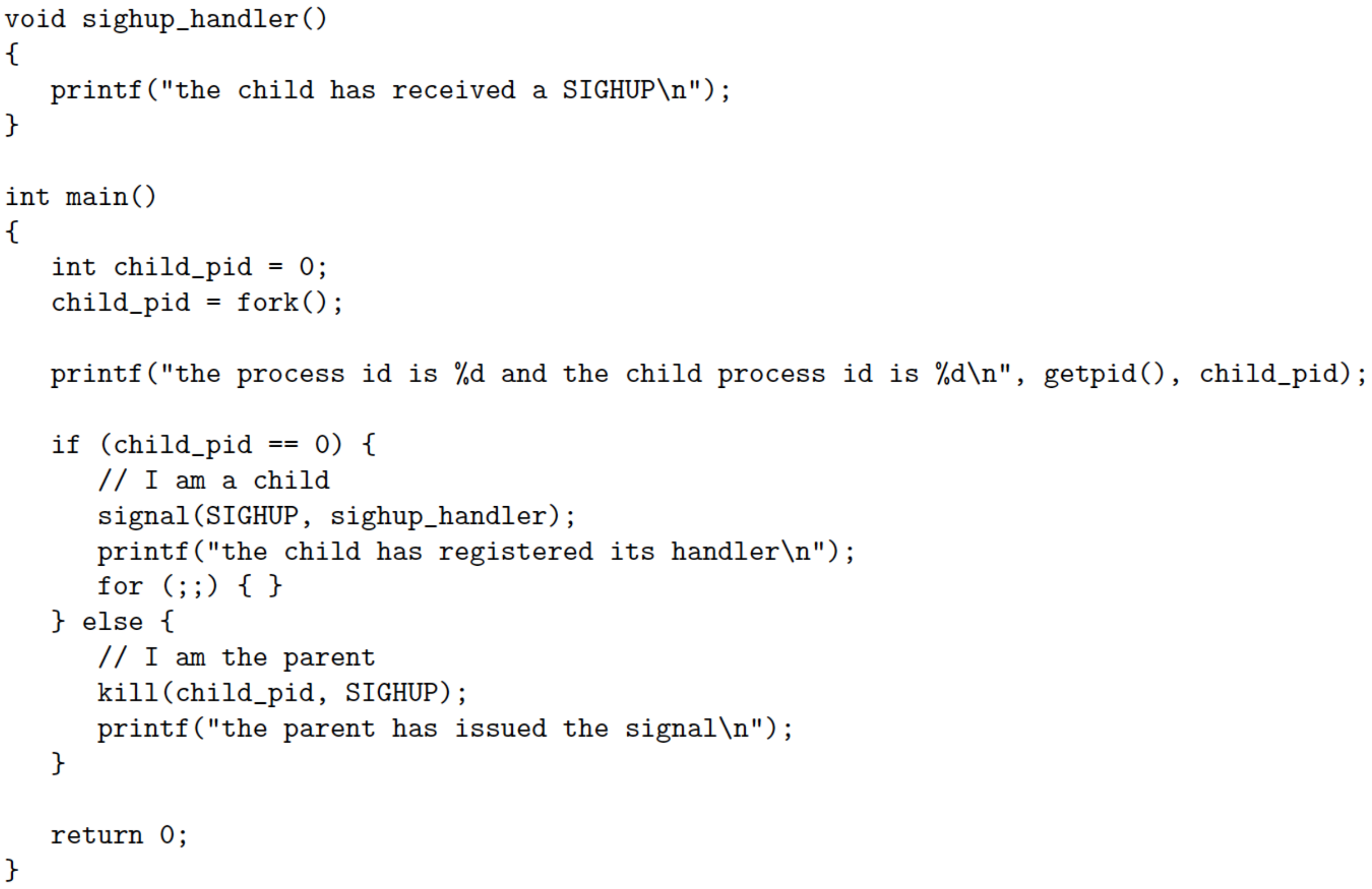
Write a simple program (main function) which executes the ls program. Lookup the manual page for the exec family (man 3 exec). After the exec call in you main function, have a printf which tells that you have called exec now. What do you notice?

Answer: The line after exec of your program will not be executed. exec replaces the currently executed program by a new one. It does not automatically fork a new process to execute ls -l.

How can you fix that?

Answer: First fork, one child does exec and the other waits for it and prints text.

Last exercise



```
void sighup_handler()
{
   printf("the child has received a SIGHUP\n");
}

int main()
{
   int child_pid = 0;
   child_pid = fork();

   printf("the process id is %d and the child process id is %d\n", getpid(), child_pid);

   if (child_pid == 0) {
      // I am a child
      signal(SIGHUP, sighup_handler);
      printf("the child has registered its handler\n");
      for (;;) { }
   } else {
      // I am the parent
      kill(child_pid, SIGHUP);
      printf("the parent has issued the signal\n");
   }

   return 0;
}
```

Last exercise

Write a program that registers a signal handler, forks 10 child processes which send signals to the parent and then terminate. In the parent, count the number of signals that you got and print them to the screen. Is this a valid implementation of a counter? Why not and which behavior of signals is problematic?

Answer: No, because signal handler does not count how many signals of the same type it receives, once it is invoked it discards further incoming signals.

Fault-Tolerance in Distributed Systems

- Various problems can occur in practice:
  - Messages may be lost
  - Nodes (client or server) may crash
  - Variable message transmission times

- One fundamental goal: *state replication*
  - Same sequence of commands in the same order

- First step to this: one command only
  - **Task:** make sure that all servers execute this same command

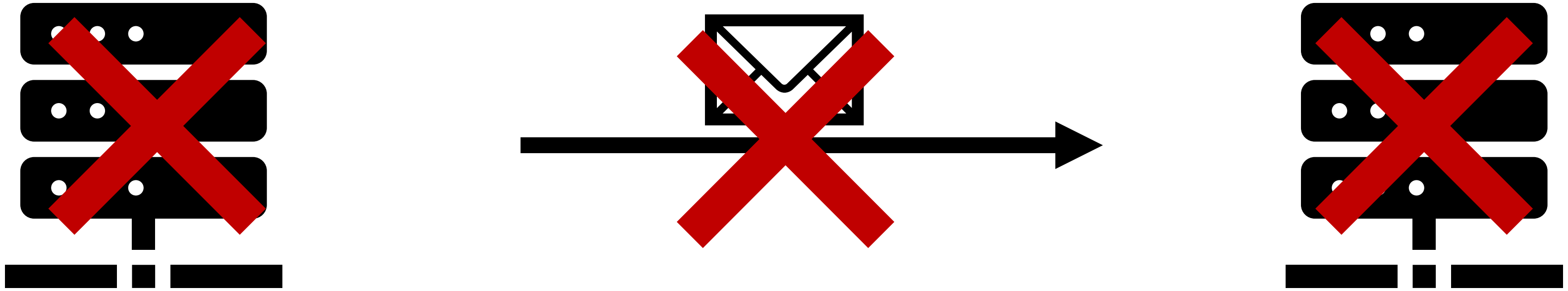
First approaches for state replication

- Server sends acknowledgement message
  - Reasonable with one client
  - Inconsistent state with multiple clients and servers

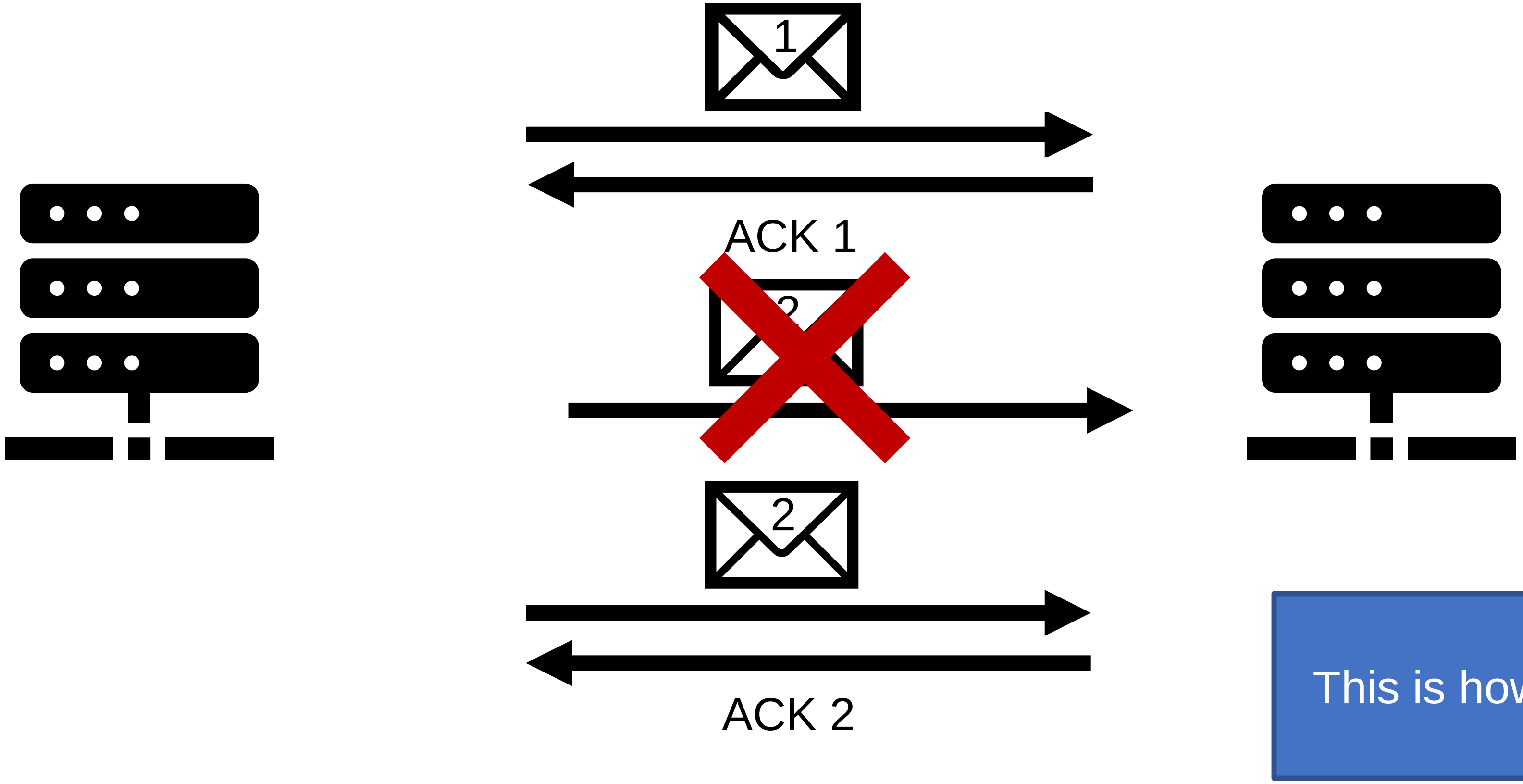
- Choose one server as a Serializer
  - Single point of failure

- Two-Phase Protocol and variants
  - How to handle server crashes?
  - How to avoid deadlock with locks?

Distributed Systems

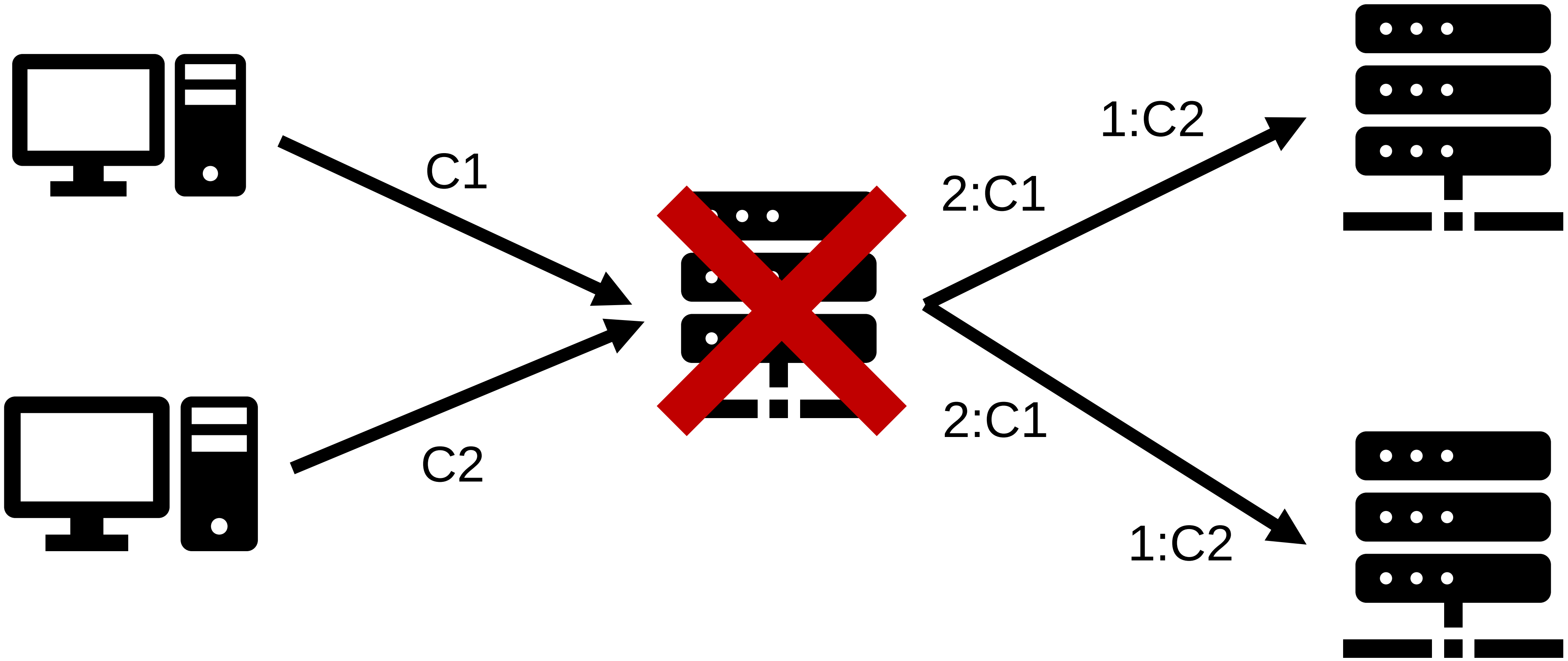


Message Loss

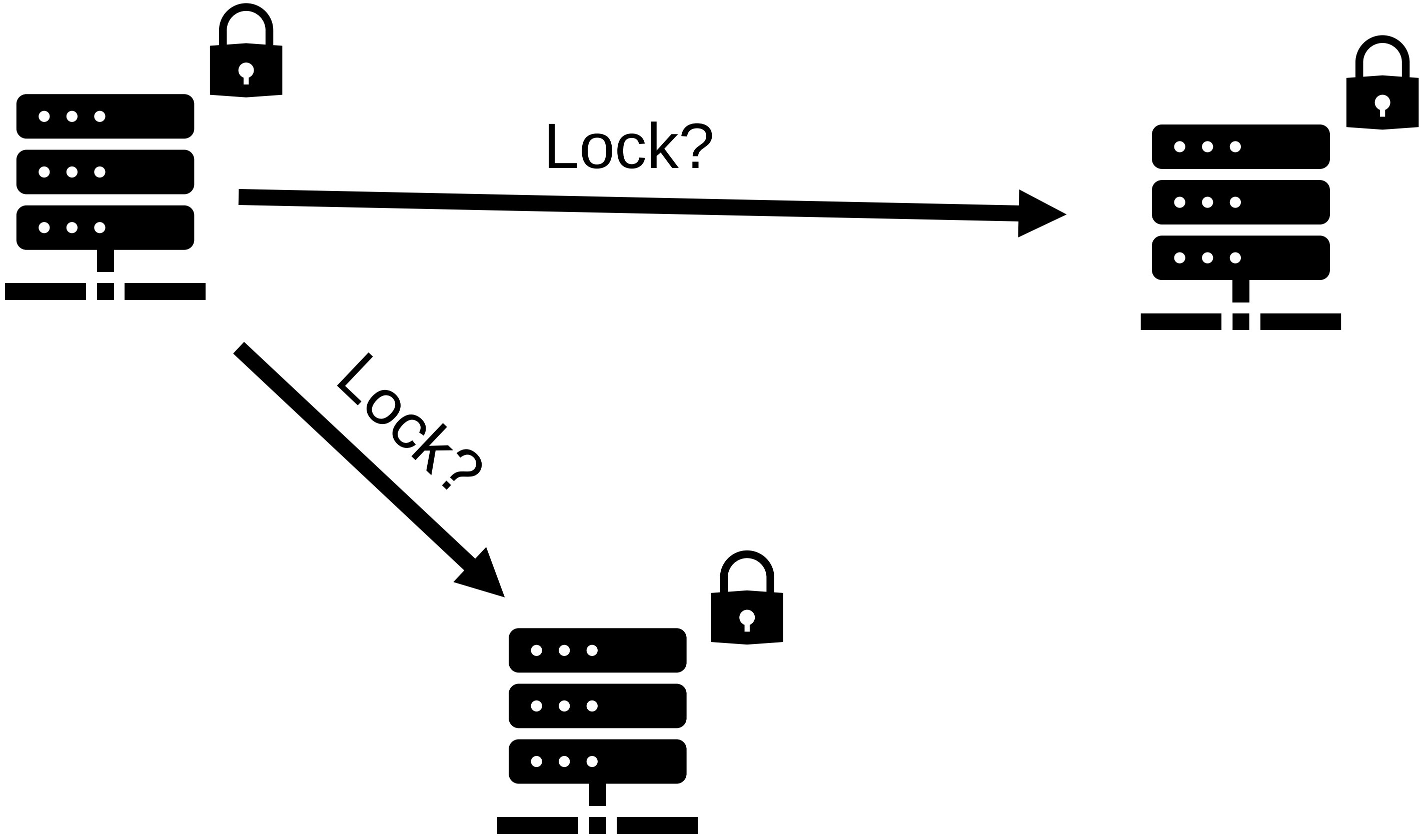


This is how TCP works

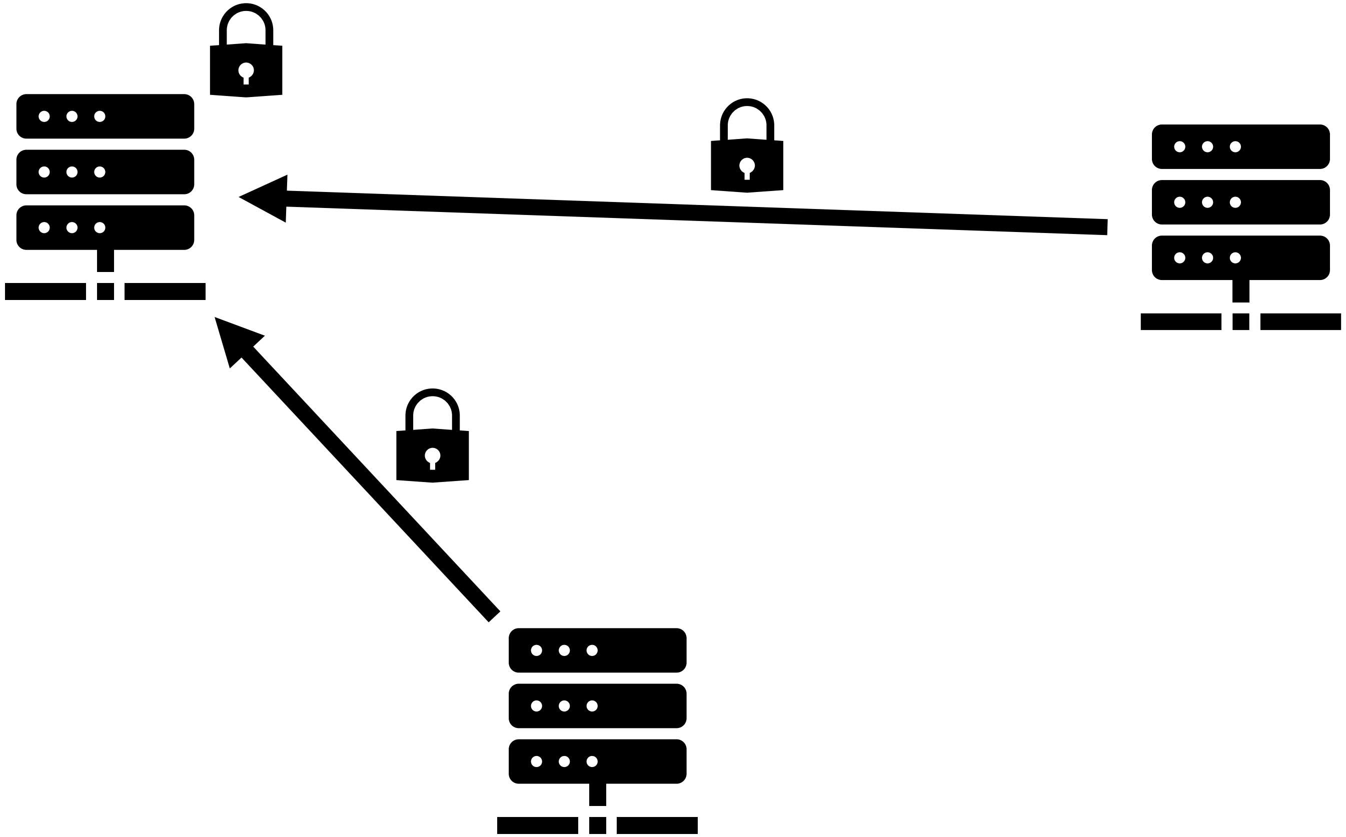
Serializer



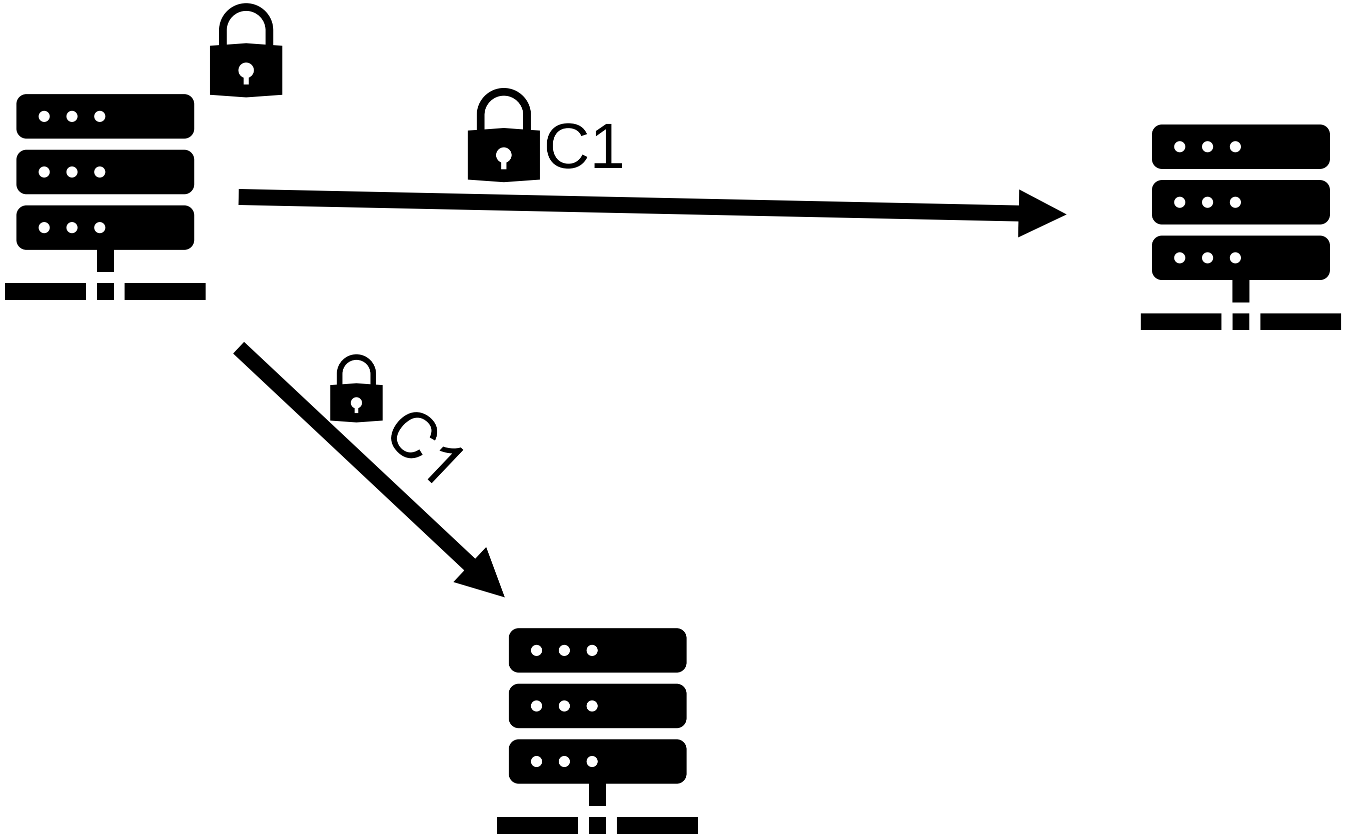
Using Locks?



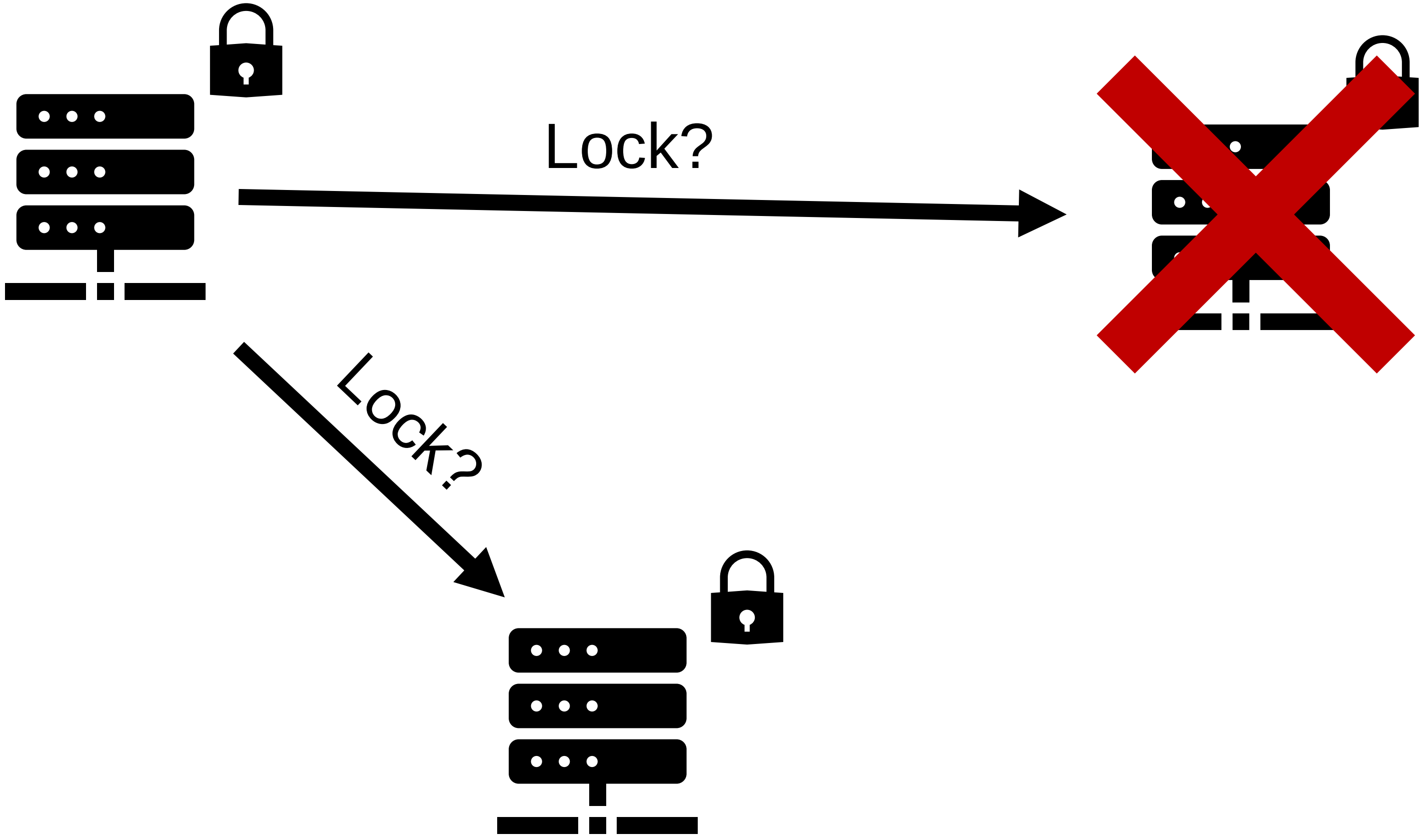
Using Locks?



Using Locks?

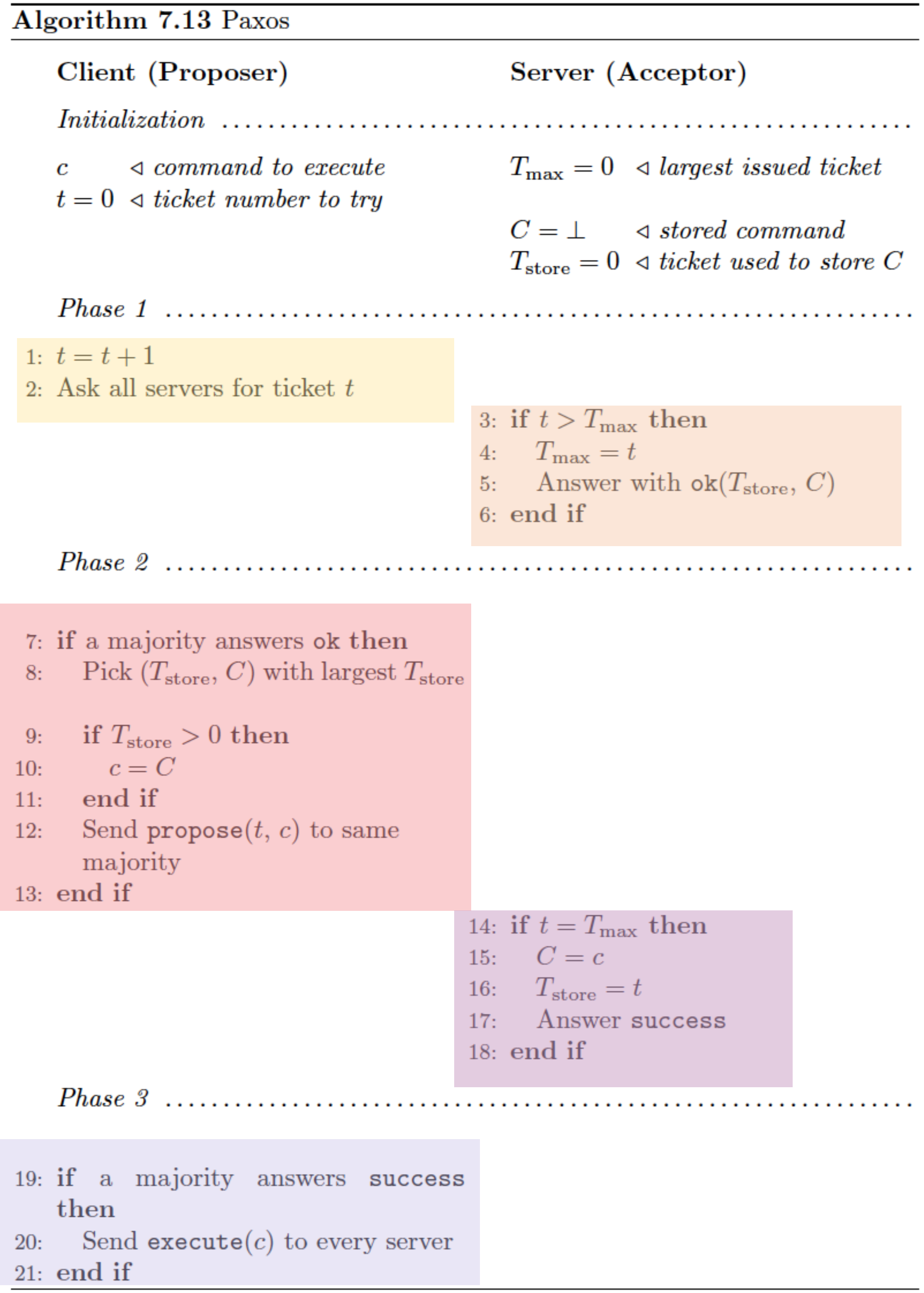


Not if nodes crash



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
- A video explaining paxos quite nicely with slightly different terminology
  - https://www.youtube.com/watch?v=d7nAGI_NZPk



```
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
                                           $T_{\text{store}} = 0$ ◁ ticket used to store $C$

Phase 1 ..................................................................
$t = t + 1$
Ask all servers for ticket $t$
                                           if $t > T_{\max}$ then
                                               $T_{\max} = t$
                                               Answer with ok($T_{\text{store}}$, $C$)
                                           end if

Phase 2 ..................................................................
if a majority answers ok then
    Pick ($T_{\text{store}}$, $C$) with largest $T_{\text{store}}$

    if $T_{\text{store}} > 0$ then
        $c = C$
    end if
    Send propose($t$, $c$) to same
    majority
end if
                                           if $t = T_{\max}$ then
                                               $C = c$
                                               $T_{\text{store}} = t$
                                               Answer success
                                           end if

Phase 3 ..................................................................
if a majority answers success
then
    Send execute($c$) to every server
end if
```

Clients asks for a specific ticket *t*

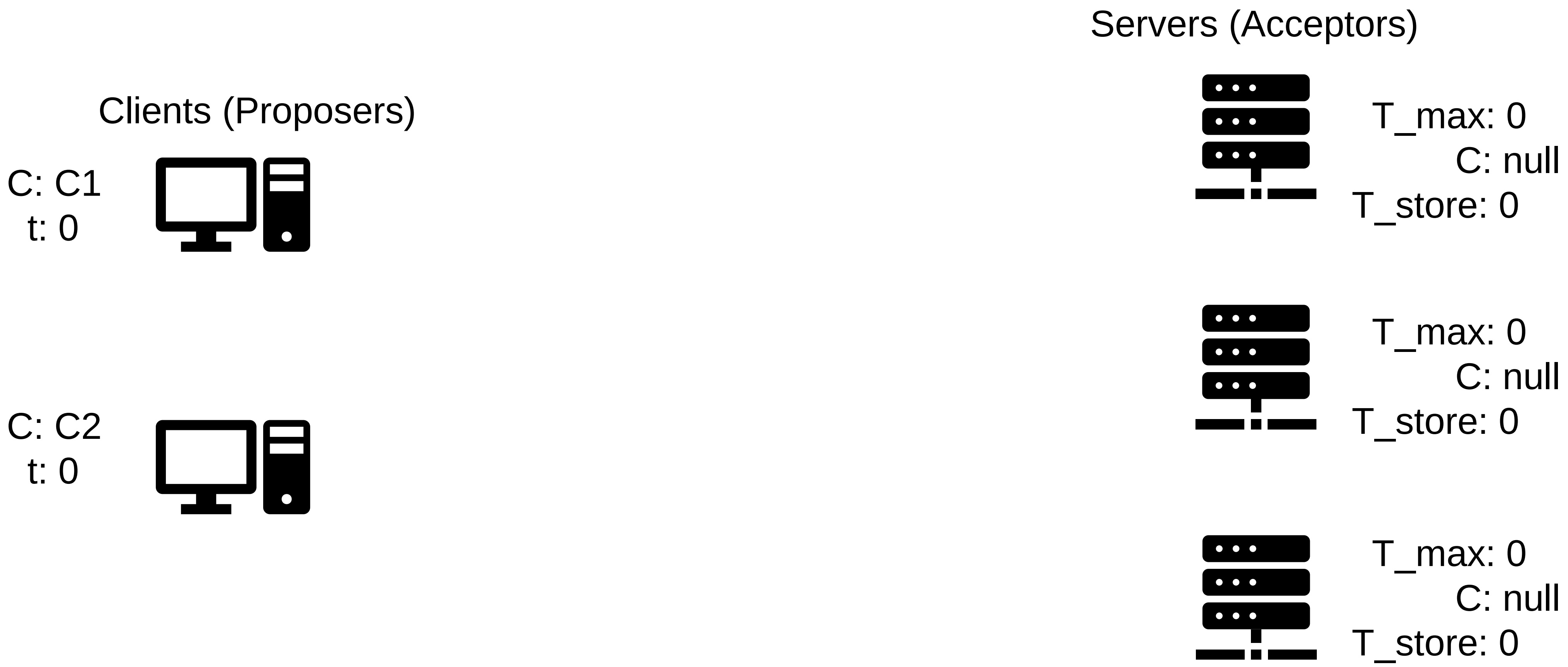
Server only issues ticket *t* if *t* is the largest ticket requested so far

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

Paxos



Servers (Acceptors)

Clients (Proposers)

C: C1
t: 0

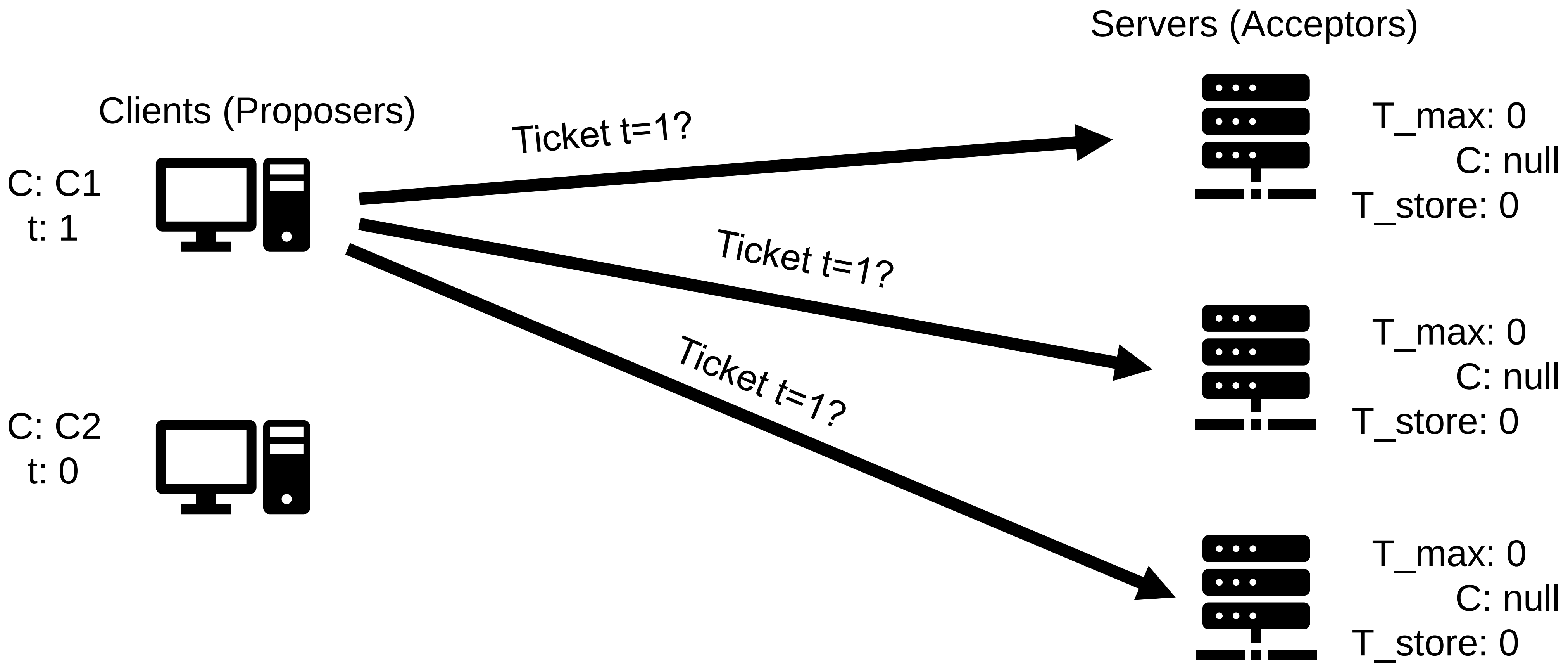
C: C2
t: 0

T_max: 0
C: null
T_store: 0

T_max: 0
C: null
T_store: 0

T_max: 0
C: null
T_store: 0

Paxos



Clients (Proposers)

C: C1
t: 1

C: C2
t: 0

Ticket t=1?

Ticket t=1?

Ticket t=1?

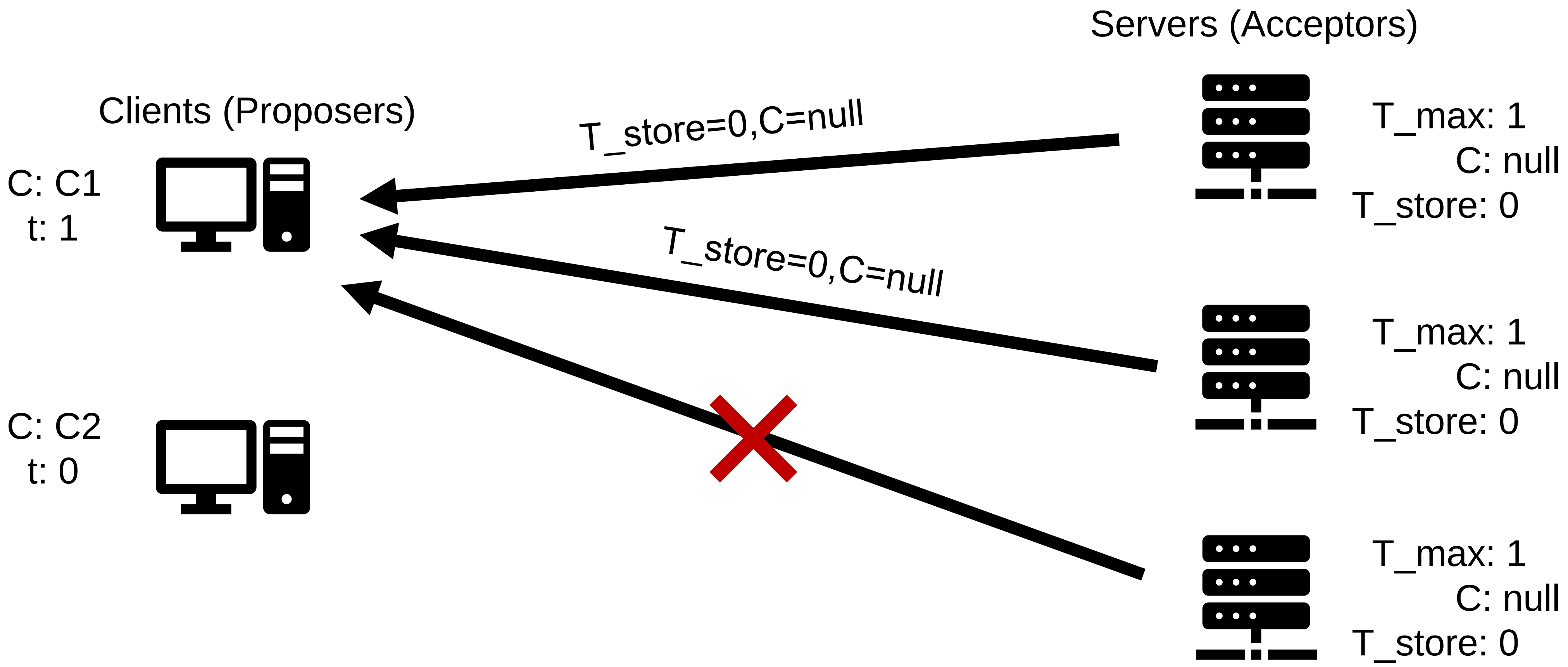
Servers (Acceptors)

T_max: 0
C: null
T_store: 0

T_max: 0
C: null
T_store: 0

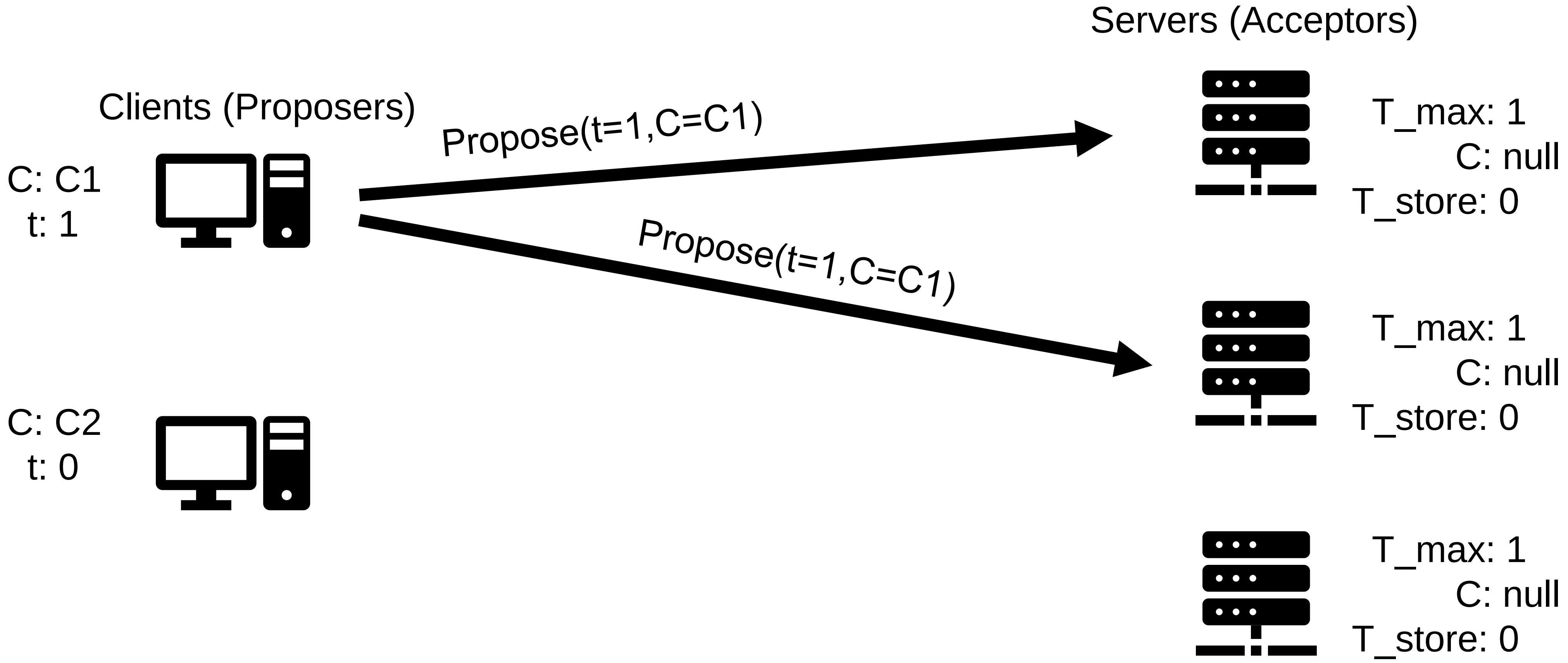
T_max: 0
C: null
T_store: 0

Paxos



Servers (Acceptors)
Clients (Proposers)
C: C1
t: 1
C: C2
t: 0
T_store=0,C=null
T_store=0,C=null
T_max: 1
C: null
T_store: 0
T_max: 1
C: null
T_store: 0
T_max: 1
C: null
T_store: 0

Paxos



Servers (Acceptors)

Clients (Proposers)

C: C1
t: 1

Propose(t=1,C=C1)

Propose(t=1,C=C1)

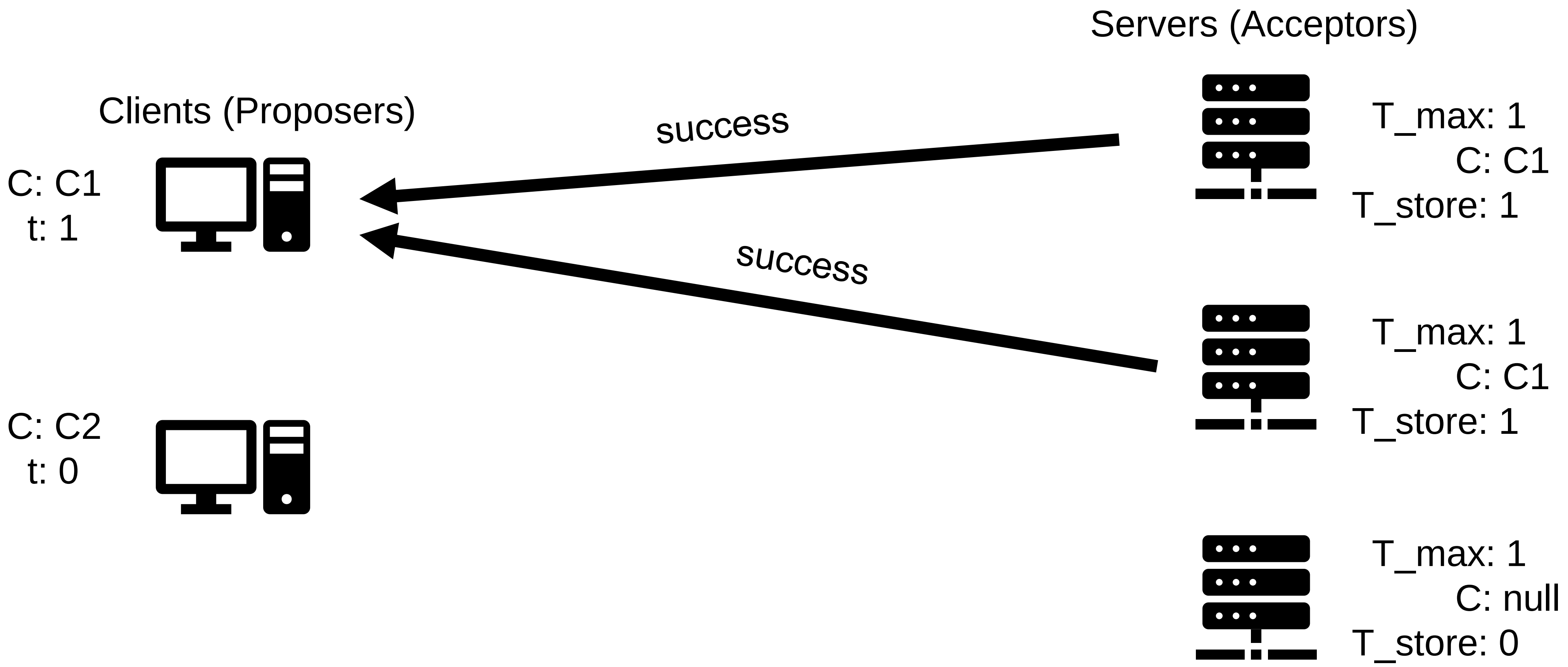
T_max: 1
C: null
T_store: 0

C: C2
t: 0

T_max: 1
C: null
T_store: 0

T_max: 1
C: null
T_store: 0

Paxos



Servers (Acceptors)

Clients (Proposers)

C: C1
t: 1

success

success

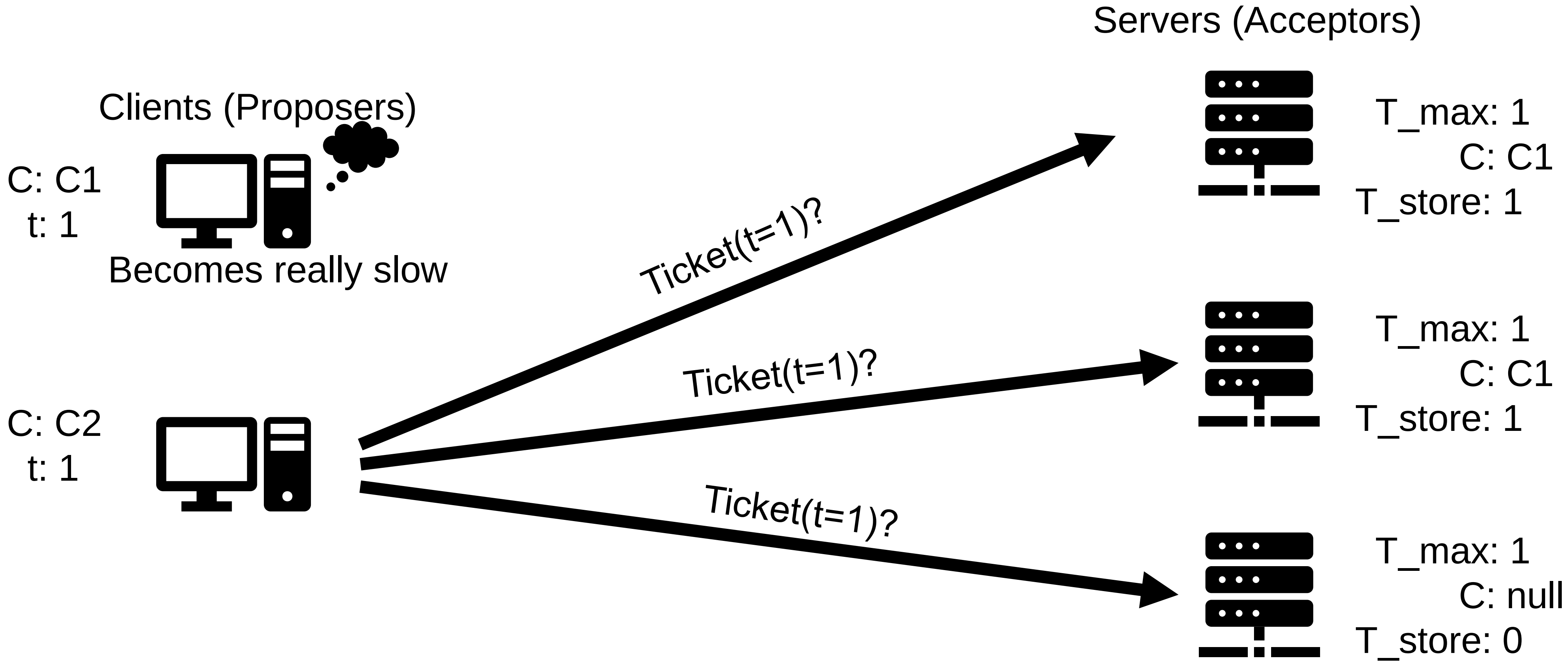
C: C2
t: 0

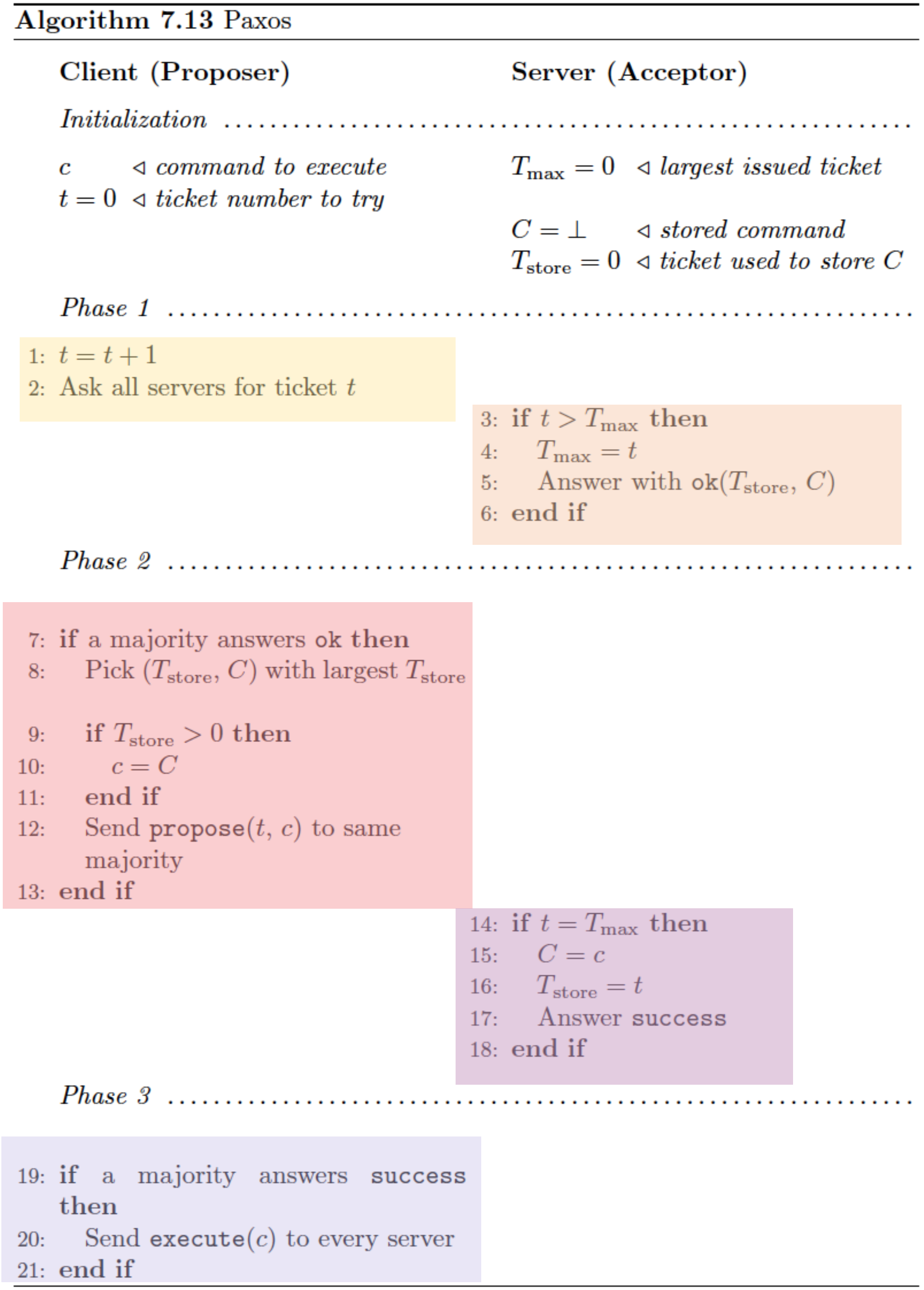
T_max: 1
C: C1
T_store: 1

T_max: 1
C: C1
T_store: 1

T_max: 1
C: null
T_store: 0

Paxos


Servers (Acceptors)
Clients (Proposers)
C: C1
t: 1
Becomes really slow
C: C2
t: 1
Ticket(t=1)?
Ticket(t=1)?
Ticket(t=1)?
T_max: 1
C: C1
T_store: 1
T_max: 1
C: C1
T_store: 1
T_max: 1
C: null
T_store: 0



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
                                           $C = \perp$     ◁ stored command
                                           $T_{store} = 0$ ◁ ticket used to store $C$

Phase 1 .................................................................
$t = t + 1$
Ask all servers for ticket $t$
                                           if $t > T_{max}$ then
                                               $T_{max} = t$
                                               Answer with ok($T_{store}$, $C$)
                                           end if
Phase 2 .................................................................
if a majority answers ok then
    Pick ($T_{store}$, $C$) with largest $T_{store}$

    if $T_{store} > 0$ then
        $c = C$
    end if
    Send propose($t$, $c$) to same
    majority
end if
                                           if $t = T_{max}$ then
                                               $C = c$
                                               $T_{store} = t$
                                               Answer success
                                           end if
Phase 3 .................................................................
if a majority answers success
then
    Send execute($c$) to every server
end if
```

Clients asks for a specific ticket *t*

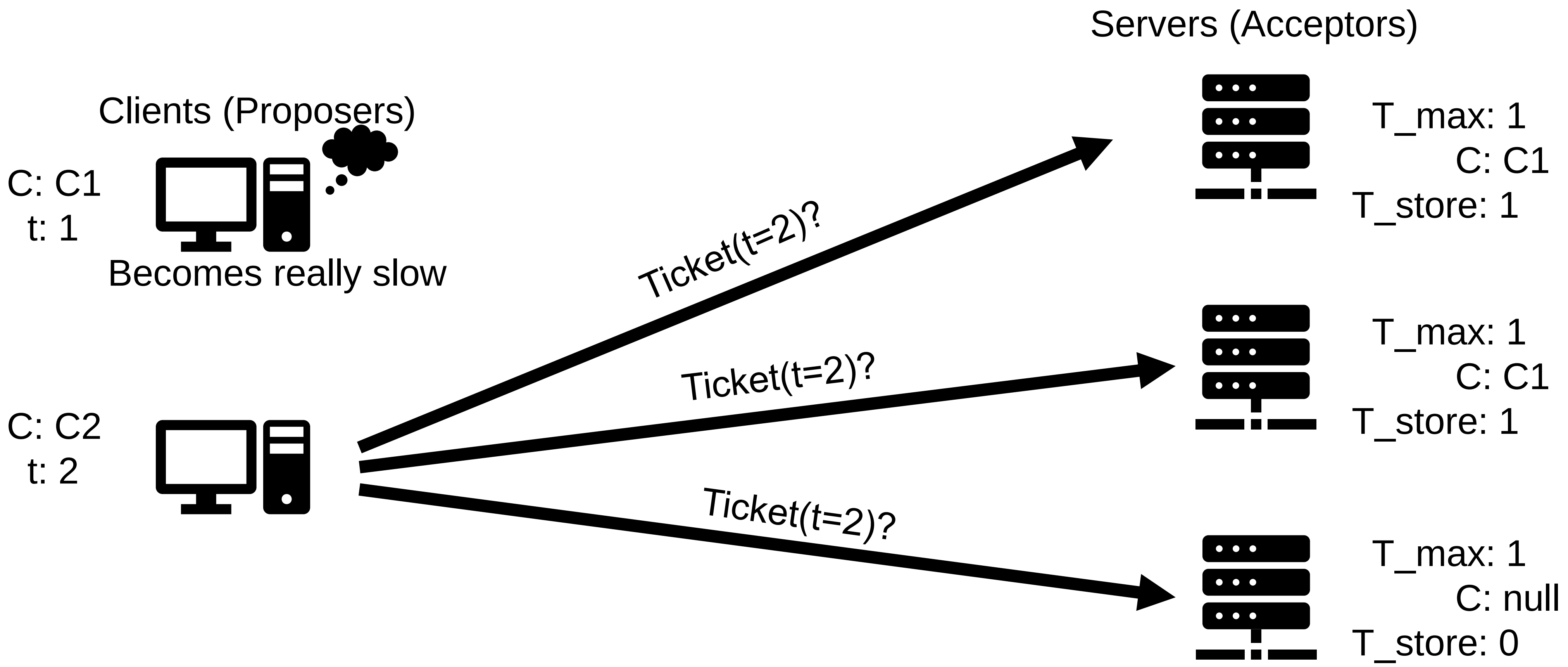
Server only issues ticket *t* if *t* is the largest ticket requested so far

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

Paxos



Servers (Acceptors)

Clients (Proposers)

C: C1
t: 1

Becomes really slow

C: C2
t: 2

Ticket(t=2)?

Ticket(t=2)?

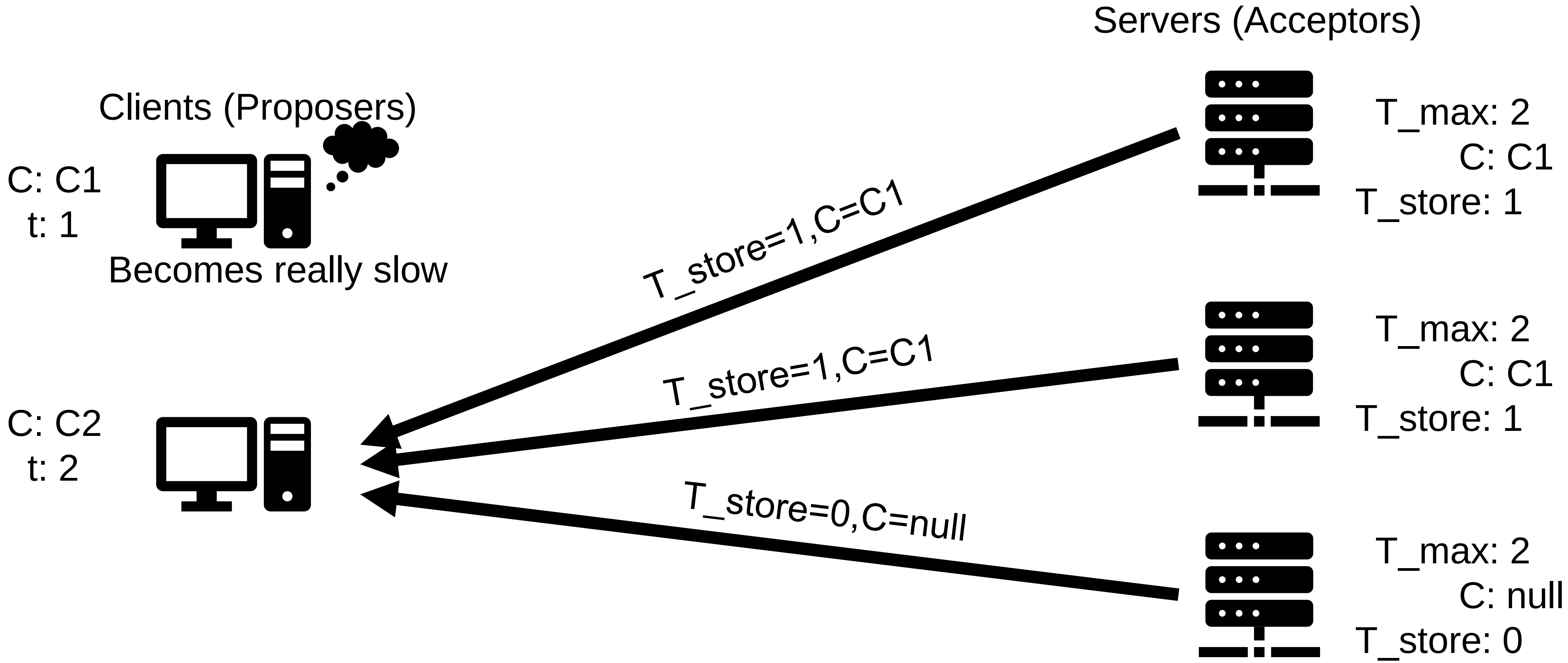
Ticket(t=2)?

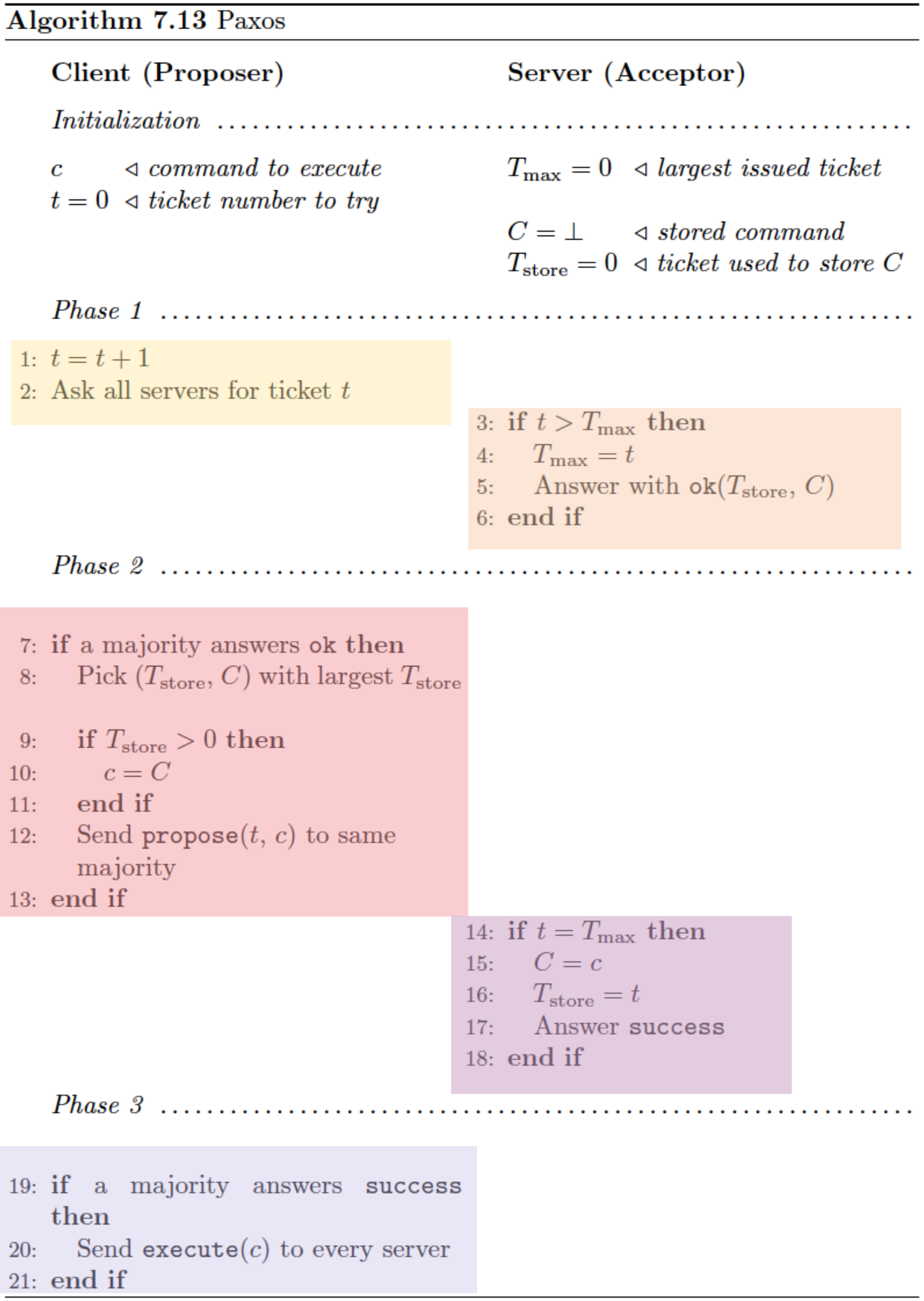
T_max: 1
C: C1
T_store: 1

T_max: 1
C: C1
T_store: 1

T_max: 1
C: null
T_store: 0

Paxos


Servers (Acceptors)
Clients (Proposers)
C: C1
t: 1
Becomes really slow
C: C2
t: 2
T_store=1,C=C1
T_store=1,C=C1
T_store=0,C=null
T_max: 2
C: C1
T_store: 1
T_max: 2
C: C1
T_store: 1
T_max: 2
C: null
T_store: 0



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
                                           $C = \perp$         ◁ stored command
                                           $T_{\text{store}} = 0$ ◁ ticket used to store $C$

Phase 1 .................................................................
$t = t + 1$
Ask all servers for ticket $t$
                                           if $t > T_{\max}$ then
                                               $T_{\max} = t$
                                               Answer with ok($T_{\text{store}}$, $C$)
                                           end if

Phase 2 .................................................................
if a majority answers ok then
    Pick ($T_{\text{store}}$, $C$) with largest $T_{\text{store}}$

    if $T_{\text{store}} > 0$ then
        $c = C$
    end if
    Send propose($t$, $c$) to same
    majority
end if
                                           if $t = T_{\max}$ then
                                               $C = c$
                                               $T_{\text{store}} = t$
                                               Answer success
                                           end if

Phase 3 .................................................................
if a majority answers success
then
    Send execute($c$) to every server
end if
```

Clients asks for a specific ticket *t*

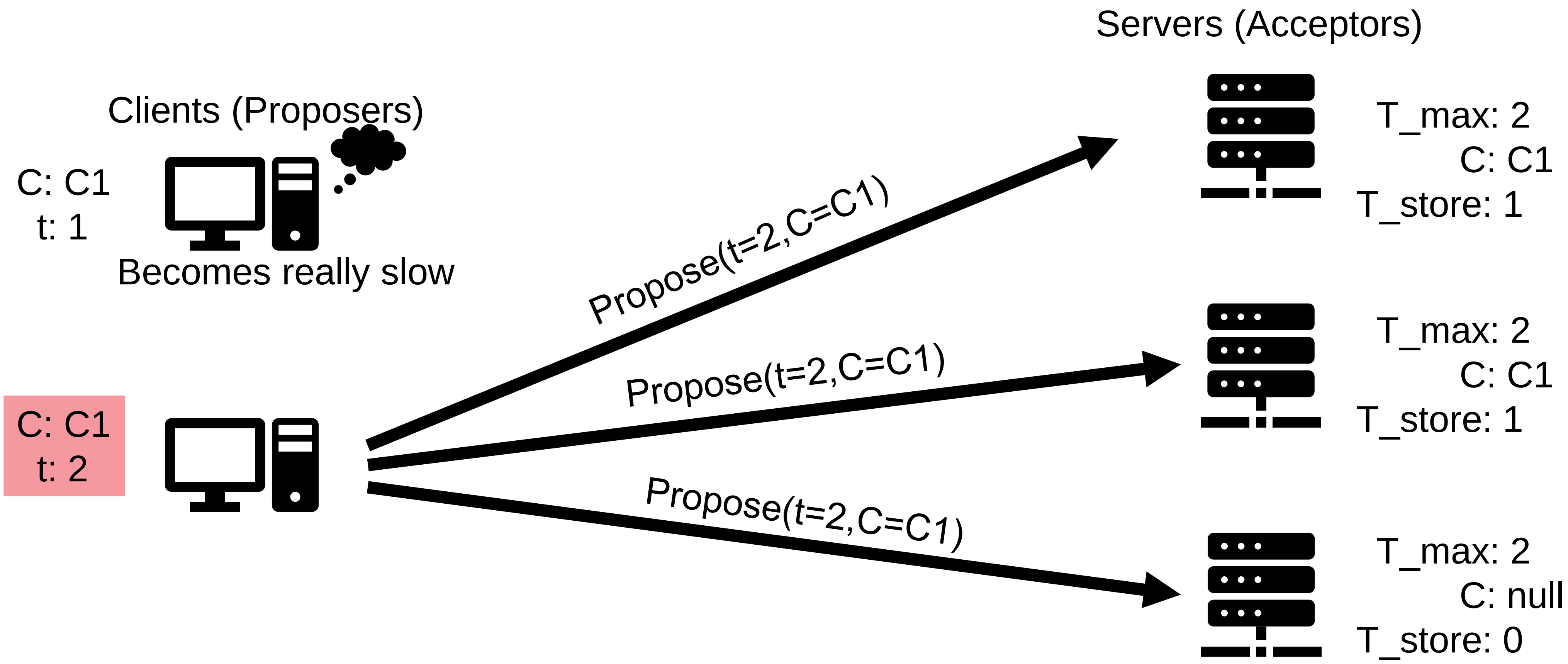
Server only issues ticket *t* if *t* is the largest ticket requested so far

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

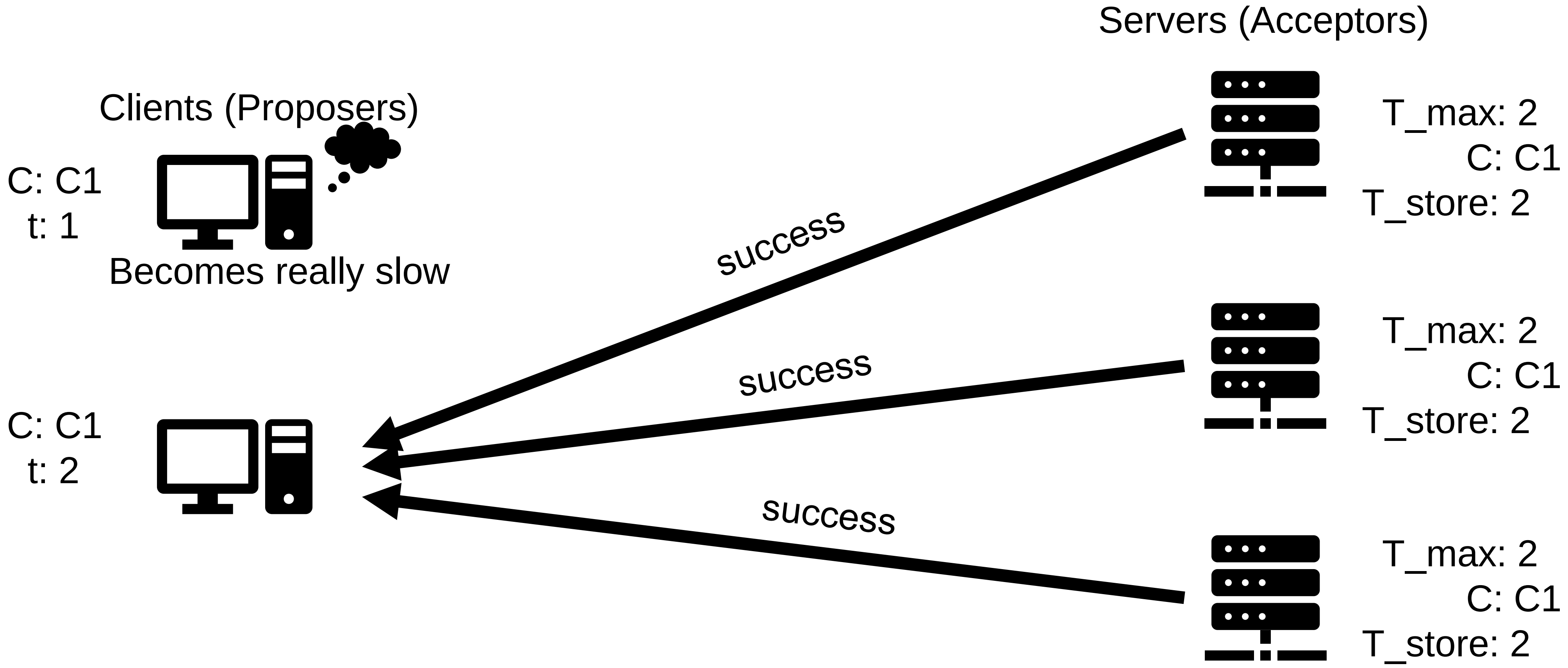
If a majority of servers store the command, the client notifies all servers to execute the command

Paxos



Servers (Acceptors)
Clients (Proposers)
C: C1
t: 1
Becomes really slow
C: C1
t: 2
Propose(t=2,C=C1)
Propose(t=2,C=C1)
Propose(t=2,C=C1)
T_max: 2
C: C1
T_store: 1
T_max: 2
C: C1
T_store: 1
T_max: 2
C: null
T_store: 0

Paxos



Servers (Acceptors)

Clients (Proposers)

C: C1
t: 1

Becomes really slow

C: C1
t: 2

success

success

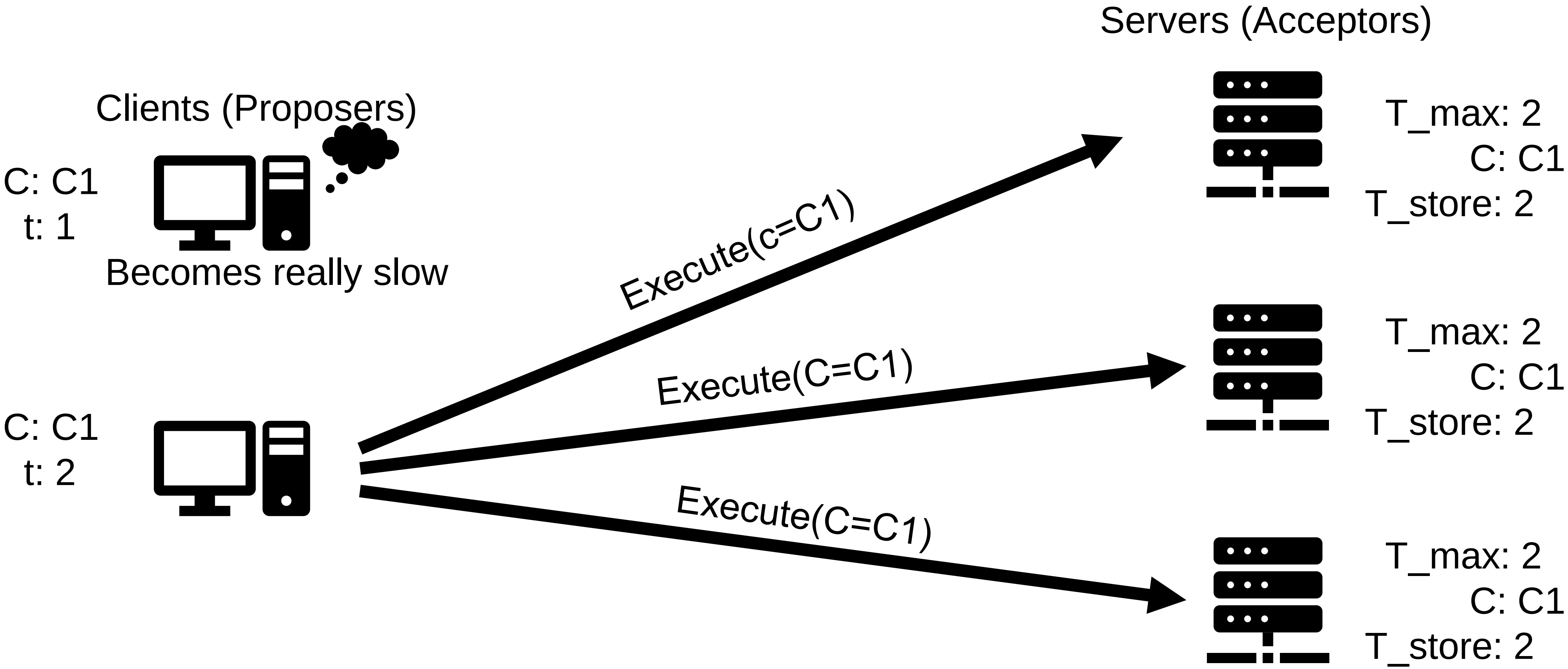
success

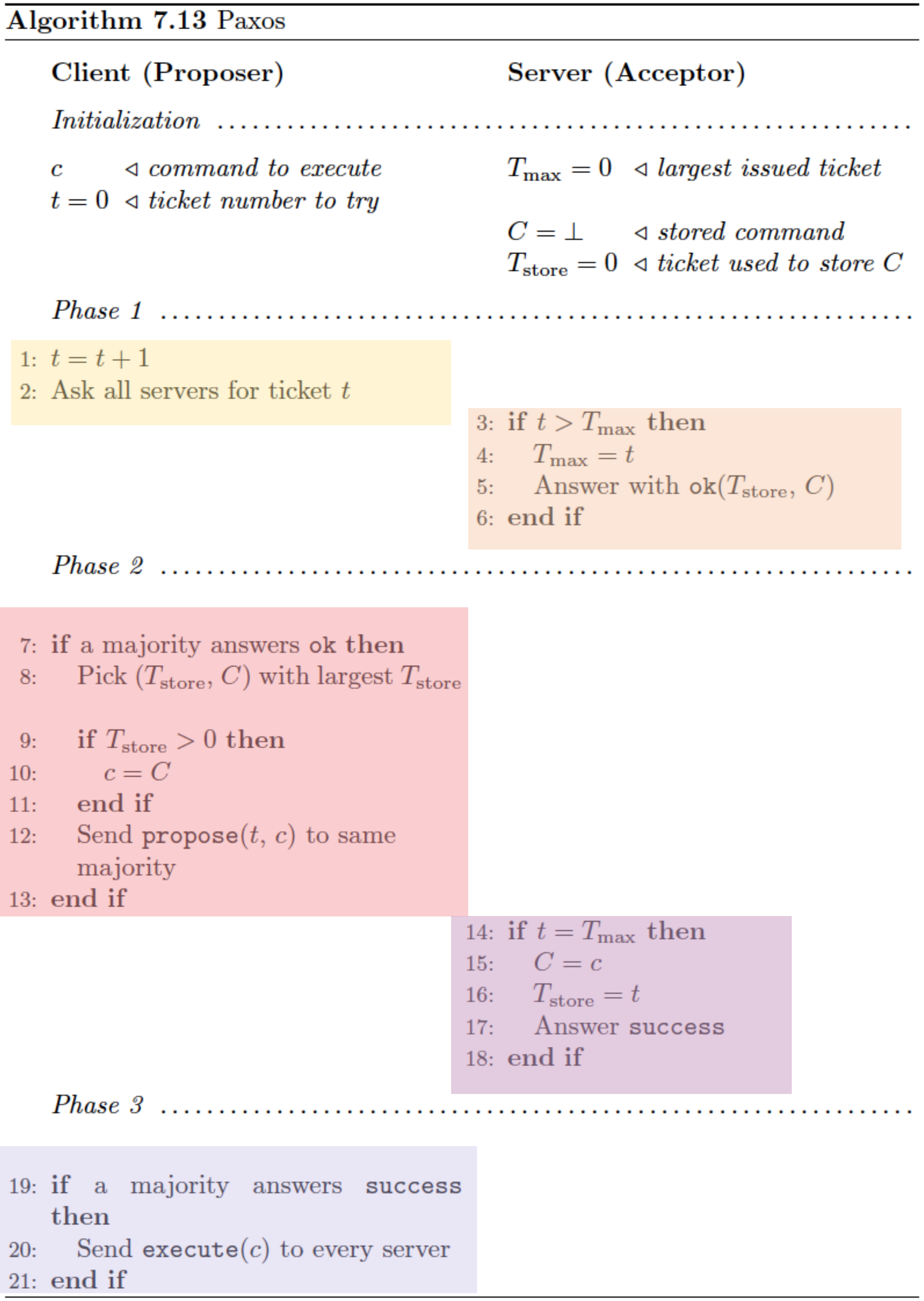
T_max: 2
C: C1
T_store: 2

T_max: 2
C: C1
T_store: 2

T_max: 2
C: C1
T_store: 2

Paxos


Servers (Acceptors)
Clients (Proposers)
C: C1
t: 1
Becomes really slow
C: C1
t: 2
Execute(c=C1)
Execute(C=C1)
Execute(C=C1)
T_max: 2
C: C1
T_store: 2
T_max: 2
C: C1
T_store: 2
T_max: 2
C: C1
T_store: 2



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 ...........................................................
$t = t + 1$
Ask all servers for ticket $t$
                                               if $t > T_{max}$ then
                                                   $T_{max} = t$
                                                   Answer with ok($T_{store}$, $C$)
                                               end if

Phase 2 ...........................................................
if a majority answers ok then
    Pick ($T_{store}$, $C$) with largest $T_{store}$

    if $T_{store} > 0$ then
        $c = C$
    end if
    Send propose($t$, $c$) to same
    majority
end if
                                               if $t = T_{max}$ then
                                                   $C = c$
                                                   $T_{store} = t$
                                                   Answer success
                                               end if

Phase 3 ...........................................................
if a majority answers success
then
    Send execute($c$) to every server
end if
```

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

Consensus

We want:

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

Impossibility:

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

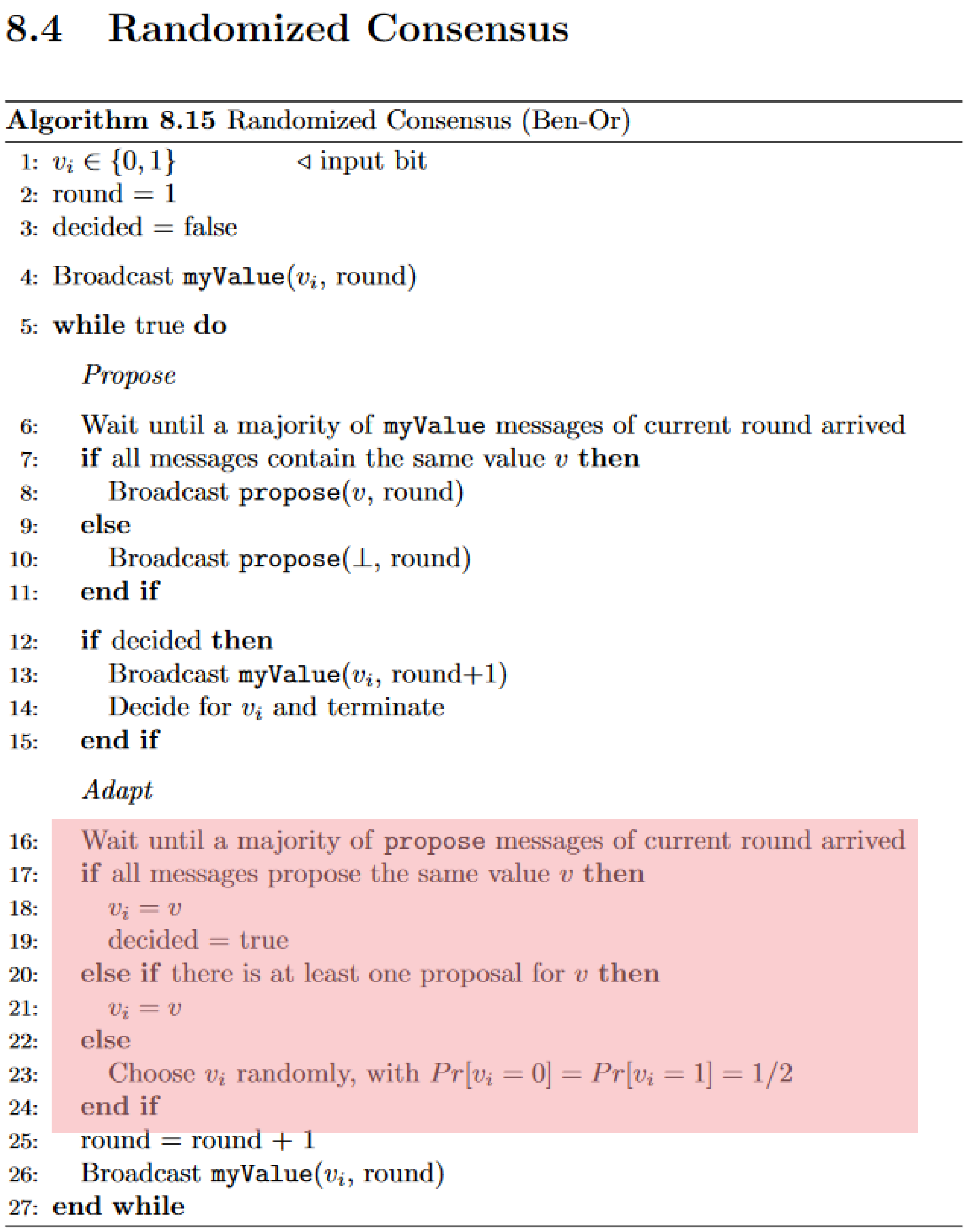
Randomized Consensus

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)
- Wouldn’t it be useful if the nodes could all toss the ***same*** coin? → Shared Coin.



8.4 Randomized Consensus

Algorithm 8.15 Randomized Consensus (Ben-Or)

```
$v_i \in \{0,1\}$            ◁ input bit
round = 1
decided = false
Broadcast myValue($v_i$, round)
while true do
  Propose
  Wait until a majority of myValue messages of current round arrived
  if all messages contain the same value $v$ then
    Broadcast propose($v$, round)
  else
    Broadcast propose($\perp$, round)
  end if
  if decided then
    Broadcast myValue($v_i$, round+1)
    Decide for $v_i$ and terminate
  end if
  Adapt
  Wait until a majority of propose messages of current round arrived
  if all messages propose the same value $v$ then
    $v_i = v$
    decided = true
  else if there is at least one proposal for $v$ then
    $v_i = v$
  else
    Choose $v_i$ randomly, with $Pr[v_i = 0] = Pr[v_i = 1] = 1/2$
  end if
  round = round + 1
  Broadcast myValue($v_i$, round)
end while
```

Majority has seen a majority ->

At least someone has seen majority ->

No majority seen ->

Randomized Consensus

Easy cases:

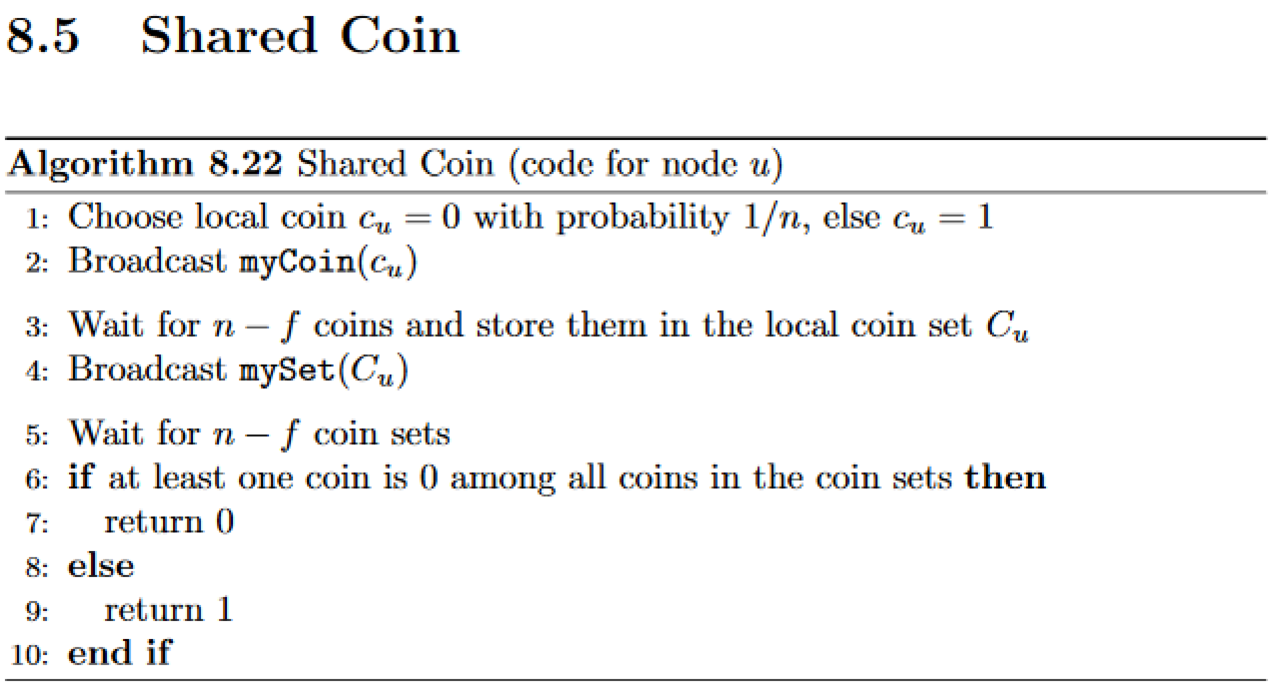
- All inputs are equal (all 0 or 1)
- Almost all input values equal

Otherwise:

- Choose a ***random*** value locally. → expected time O(2^n) until all agree (once)
- Wouldn’t it be useful if the nodes could all toss the ***same*** coin? → Shared Coin.

Shared Coin

- The algorithm stays exactly the same except the standard coin flip is replaced by a call to the shared coin algorithm
- Proofs for validity and agreement still hold (since it is still the same algorithm)
- The proof for termination has to be changed slightly to account for the changed probability that all coins will give the same result
- With a shared coin the runtime goes from exponential down to constant!
- But we can only tolerate $f < n/3$ crash failures as opposed to $f < n/2$



8.5 Shared Coin

Algorithm 8.22 Shared Coin (code for node u)

```
Choose local coin $c_u = 0$ with probability $1/n$, else $c_u = 1$
Broadcast myCoin($c_u$)
Wait for $n - f$ coins and store them in the local coin set $C_u$
Broadcast mySet($C_u$)
Wait for $n - f$ coin sets
if at least one coin is 0 among all coins in the coin sets then
    return 0
else
    return 1
end if
```

Quiz

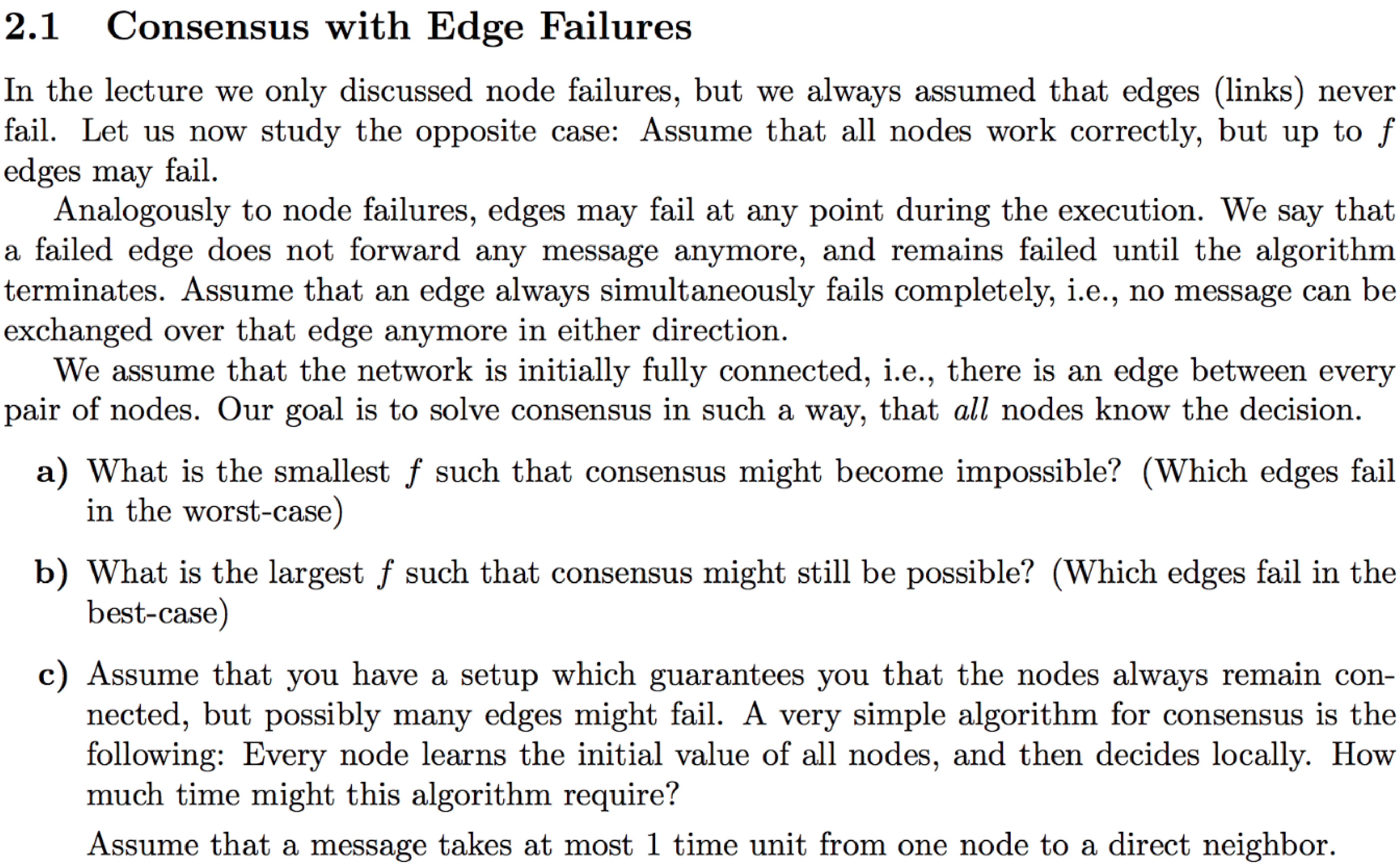
- **How does a node in Paxos know if a majority answered with ok?**
  - All nodes must know how many servers are in the system
- **Does the Paxos algorithm in the script achieve state replication?**
  - No it only shows one instance, for subsequent commands it would need to be restarted
  - be restarted
- **How many nodes could crash so that Paxos still works?**
  - less than half
- **Does Paxos solve consensus?**
  - No, termination is not given

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.