Computer Systems

Exercise Session 4

Agenda

- Bonus Task (Deadlines are anounced)
- Last Weeks Exercise (Paxos, Consensus)
- Scheduling
- I/O
  - Interrupts
  - DMA
- New exercise

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?

Algorithm 7.13 Paxos

Client (Proposer)	Server (Acceptor)
Initialization	
c ◁ *command to execute* $t = 0$ ◁ *ticket number to try*	$T_{\max} = 0$ ◁ *largest issued ticket* $C = \perp$ ◁ *stored command* $T_{\text{store}} = 0$ ◁ *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 $T_{\text{store}} > 0$ 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: $T_{\text{store}} = t$ 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	

Clients asks for a specific ticket *t*

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

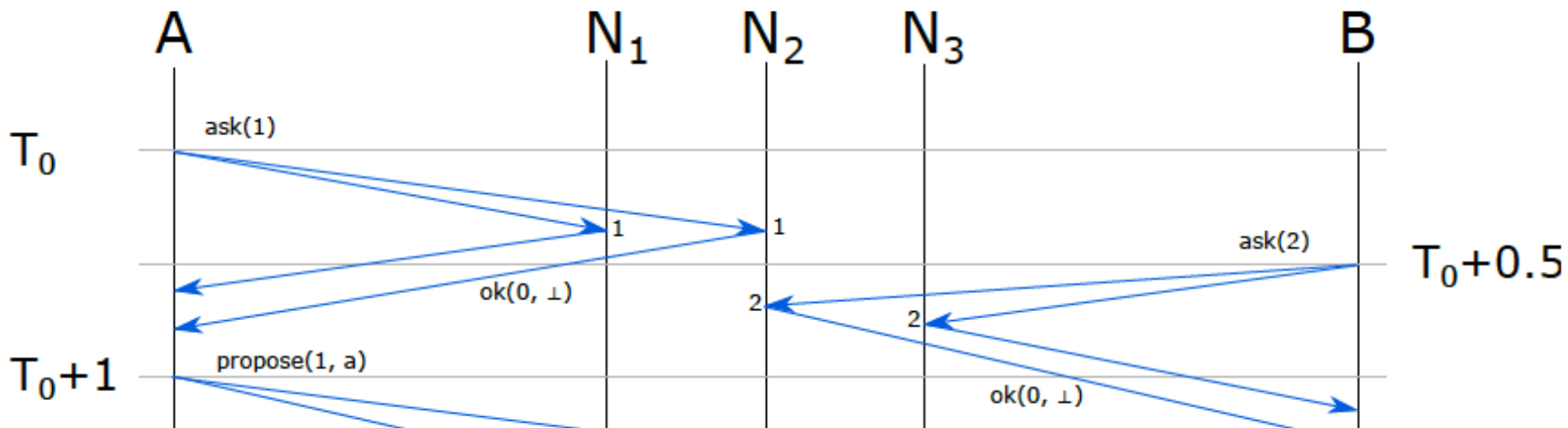
If client receives majority of tickets, it proposes a command

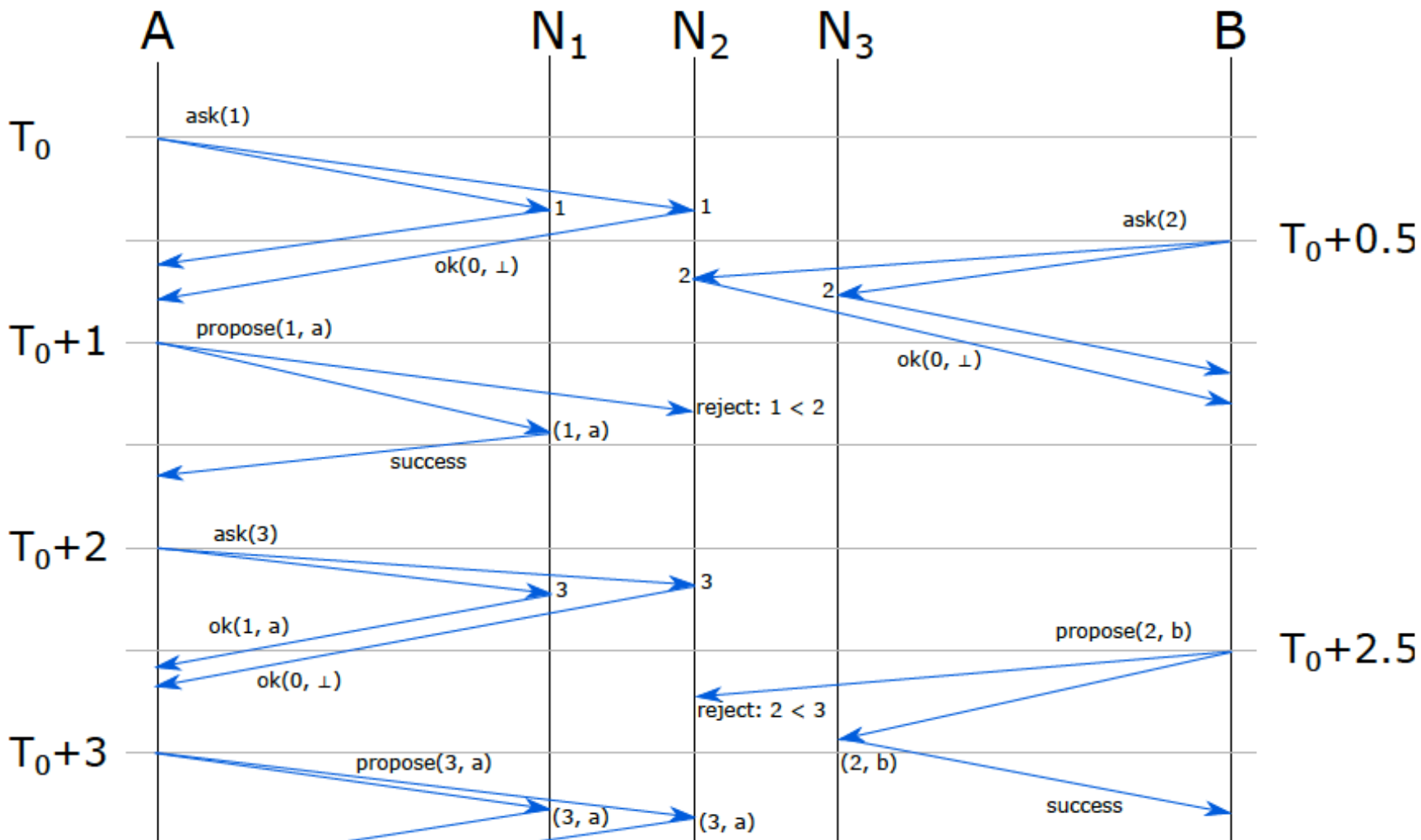
When a server receives a proposal, if the ticket of the client is still valid, the server stores the command and notifies the client

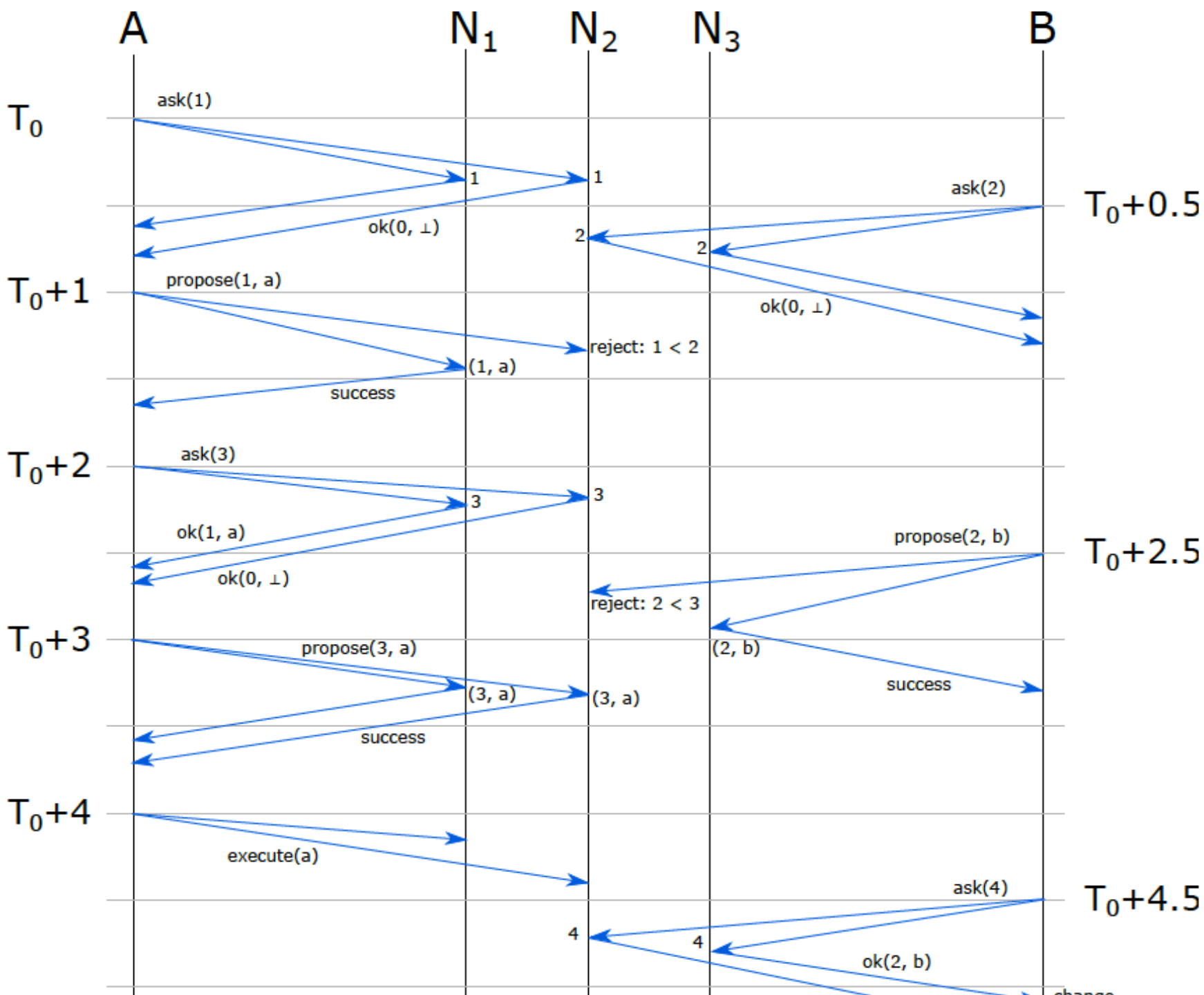
If a majority of servers store the command, the client notifies all servers to execute the command

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)
1
1
ask(2)
T0+0.5
ok(0, ⊥)
2
2
T0+1
propose(1, a)
ok(0, ⊥)
reject: 1 < 2
(1, a)
success
T0+2
ask(3)
3
3
ok(1, a)
propose(2, b)
T0+2.5
ok(0, ⊥)
reject: 2 < 3
T0+3
propose(3, a)
(2, b)
success
(3, a)
(3, a)


A
N1
N2
N3
B
T0
ask(1)
1
1
ok(0, ⊥)
T0+0.5
ask(2)
2
2
ok(0, ⊥)
T0+1
propose(1, a)
reject: 1 < 2
(1, a)
success
T0+2
ask(3)
3
3
ok(1, a)
ok(0, ⊥)
T0+2.5
propose(2, b)
reject: 2 < 3
(2, b)
success
T0+3
propose(3, a)
(3, a)
(3, a)
success
T0+4
execute(a)
T0+4.5
ask(4)
4
4
ok(2, b)

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?

Answer: A possible worst-case scenario is when all clients start their attempt to execute a command (approximately) at the same time, use the same timeout and the same initial ticket number.

In that case it can happen that two clients always invalidate each others tickets, and no client ever succeeds with finding a majority for its proposal messages.

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.

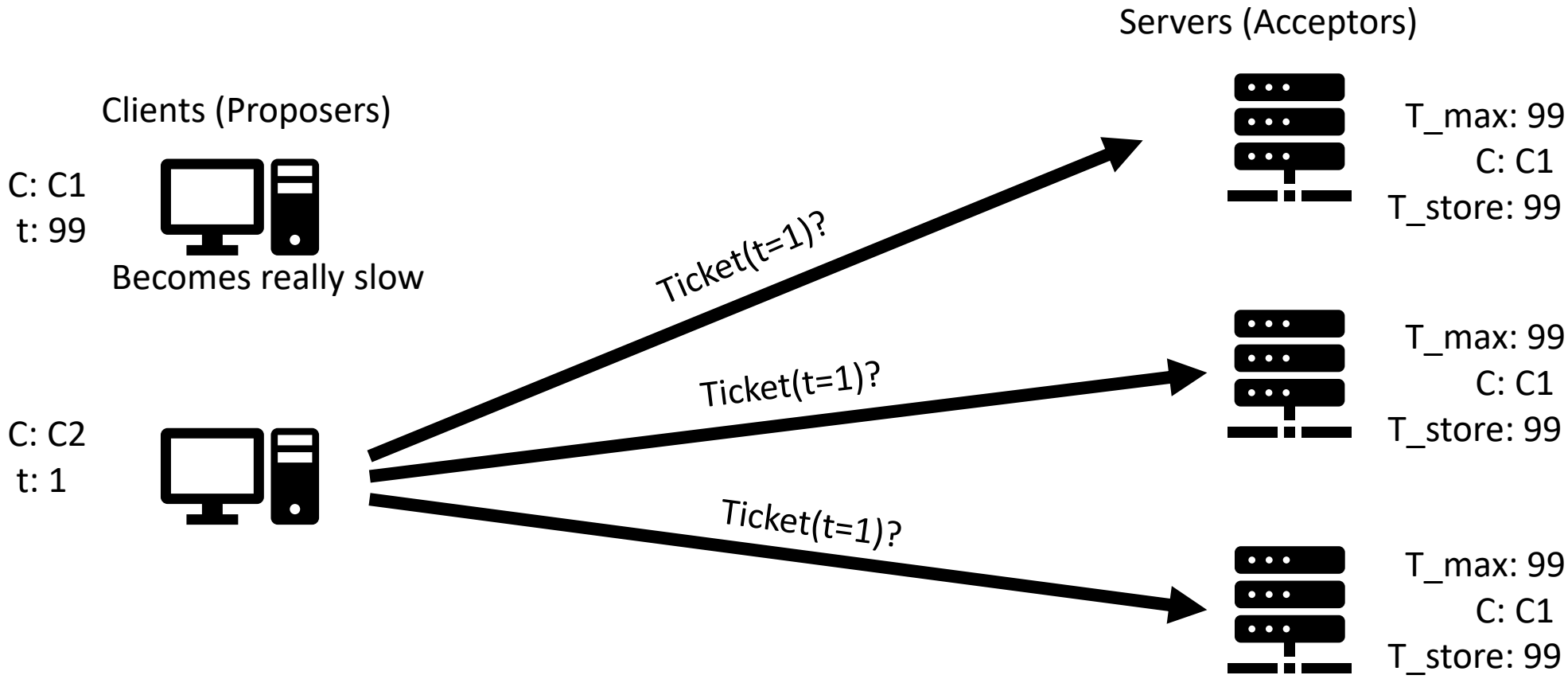
The point in time when clients start sending messages cannot be controlled, since this will be determined by the application that uses Paxos. It might help to use different initial ticket numbers. However, if a client with a very high ticket number crashes early, all other clients need to iterate through 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?

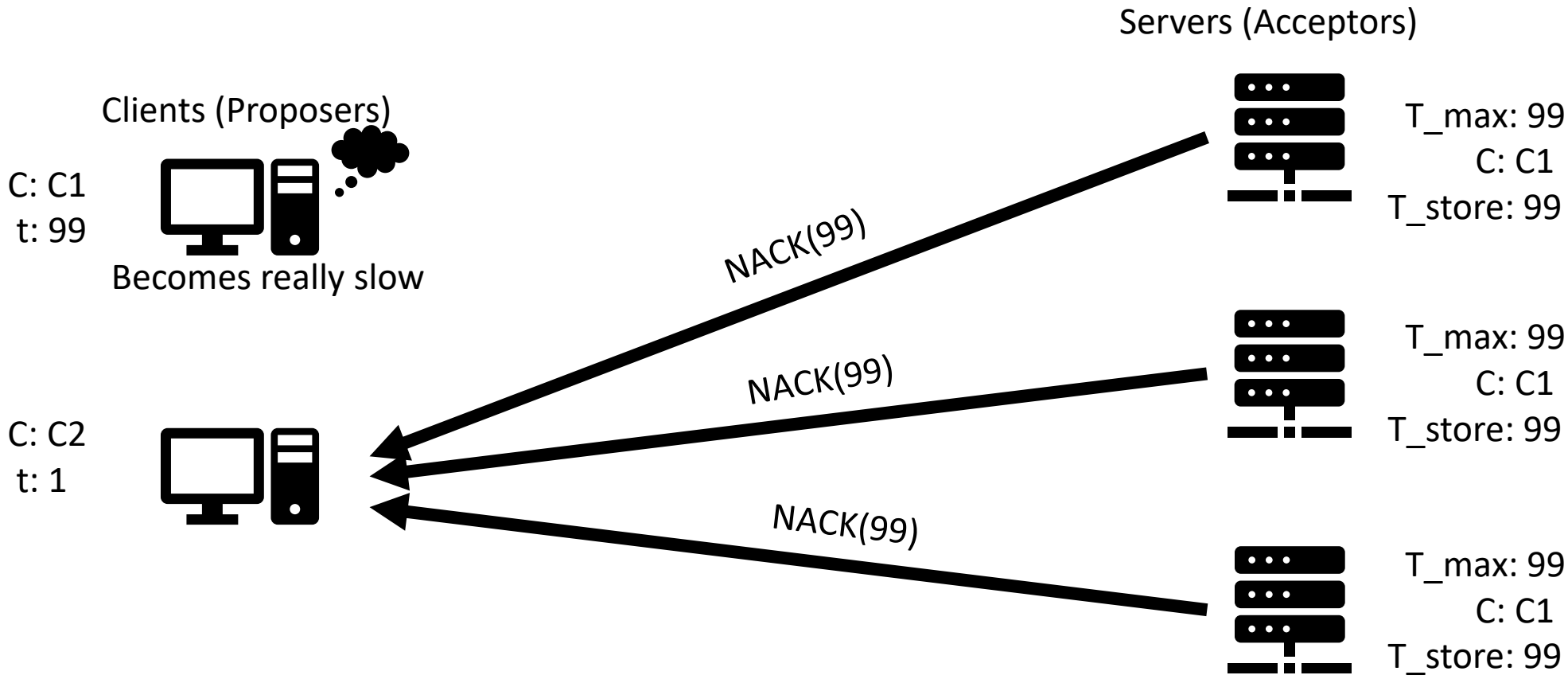
b) Instead of changing the parameters, we add a waiting time between sending two consecutive `ask` messages. Sketch an idea of how you could improve the expected runtime in a scenario where multiple clients are trying to execute a command by manipulating this waiting time!

Extra challenge: Try not to slow down an individual client if it is alone!

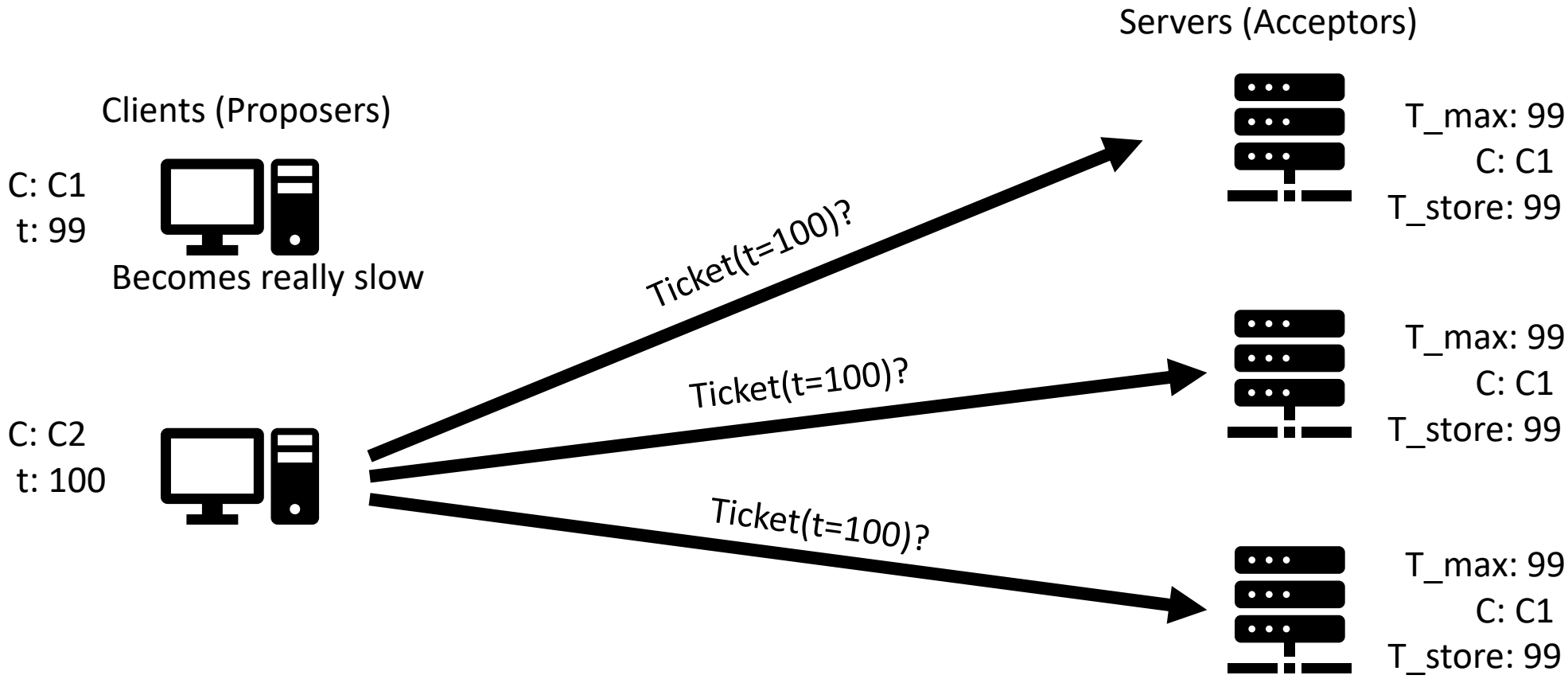
Paxos


Servers (Acceptors)
Clients (Proposers)
C: C1
t: 99
Becomes really slow
C: C2
t: 1
Ticket(t=1)?
Ticket(t=1)?
Ticket(t=1)?
T_max: 99
C: C1
T_store: 99
T_max: 99
C: C1
T_store: 99
T_max: 99
C: C1
T_store: 99

Paxos

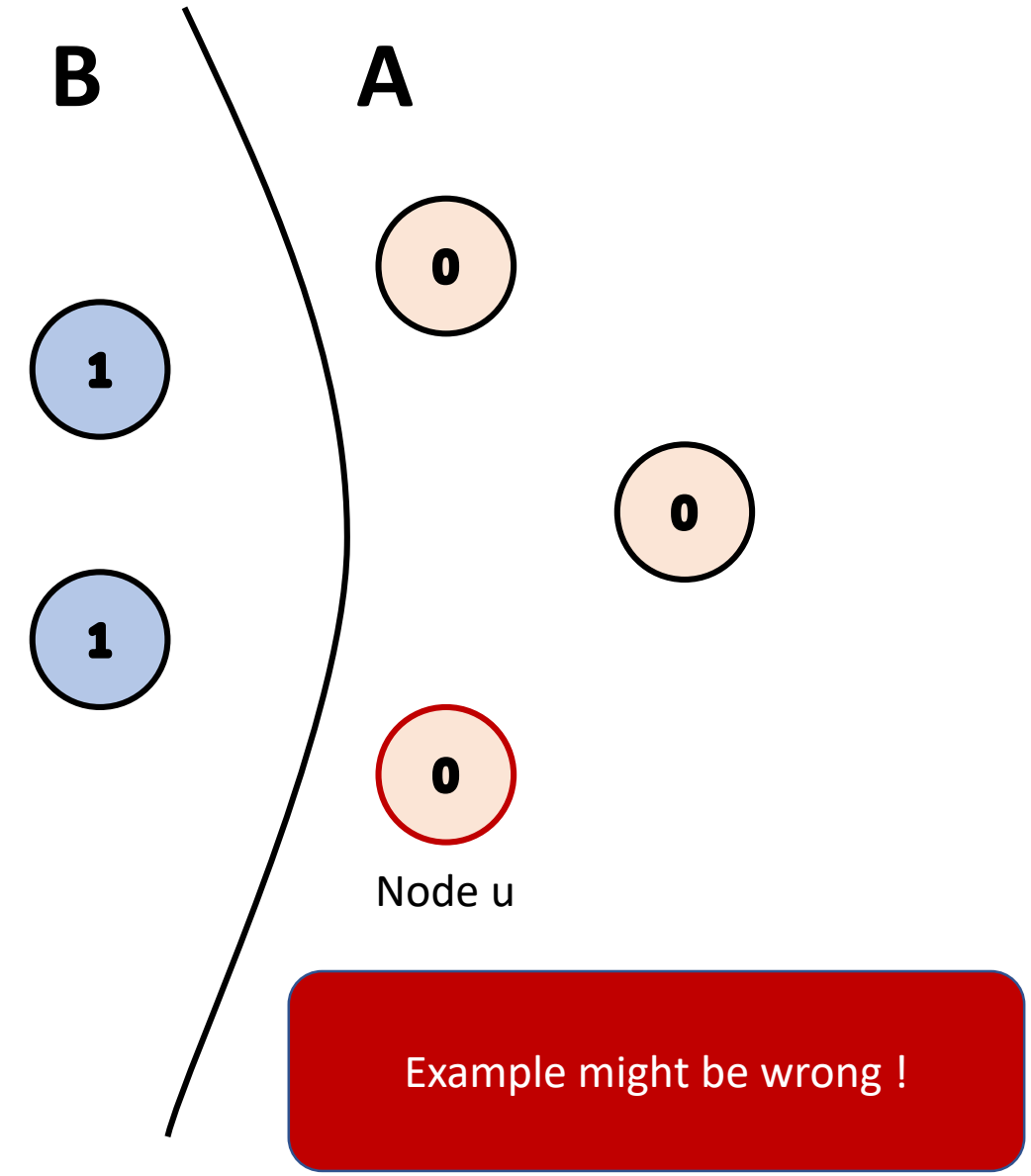

Servers (Acceptors)
Clients (Proposers)
C: C1
t: 99
Becomes really slow
C: C2
t: 1
NACK(99)
NACK(99)
NACK(99)
T_max: 99
C: C1
T_store: 99
T_max: 99
C: C1
T_store: 99
T_max: 99
C: C1
T_store: 99

Paxos


Servers (Acceptors)
Clients (Proposers)
C: C1
t: 99
Becomes really slow
C: C2
t: 100
Ticket(t=100)?
Ticket(t=100)?
Ticket(t=100)?
T_max: 99
C: C1
T_store: 99
T_max: 99
C: C1
T_store: 99
T_max: 99
C: C1
T_store: 99

Algorithm 2 Randomized Consensus (Ben-Or)

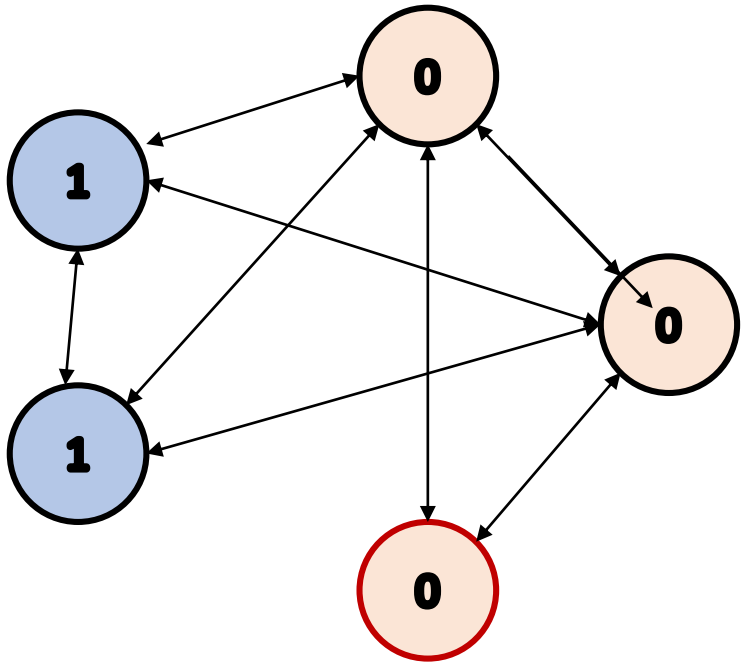
```
1:  v_i ∈ {0,1}                          ◁ input bit
2:  round = 1
3:  decided = false
4:  Broadcast myValue(v_i, round)
5:  while true do
        Propose
6:      Wait until a majority of myValue messages of current round arrived
7:      if all messages contain the same value v then
8:          Broadcast propose(v, round)
9:      else
10:         Broadcast propose(⊥, 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
```



Algorithm 2 Randomized Consensus (Ben-Or)

```
1:  v_i ∈ {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(⊥, 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
```

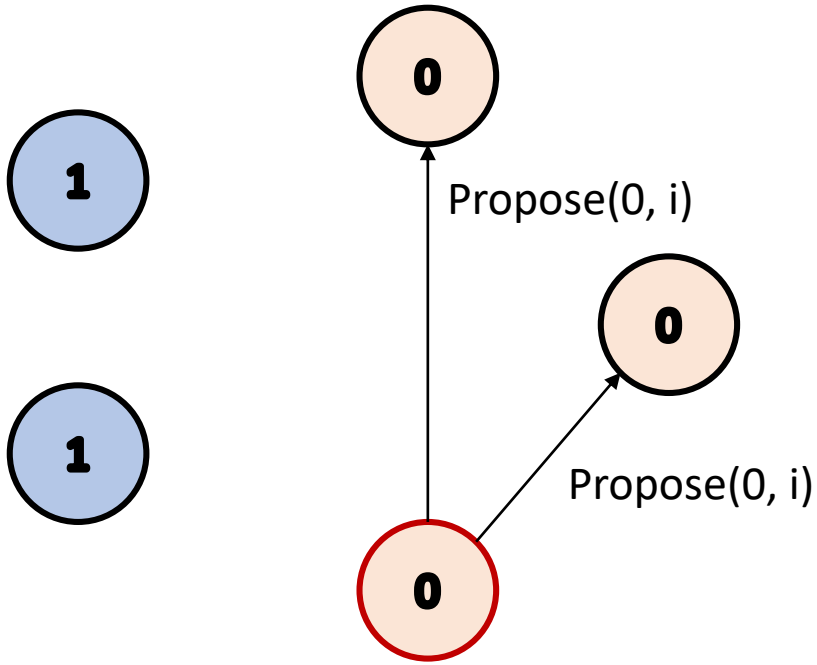
Example might be wrong !



Everyone broadcasts their value, node u receives only messages from region A

Algorithm 2 Randomized Consensus (Ben-Or)

```
1:  v_i ∈ {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(⊥, 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
```



Example might be wrong !

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.

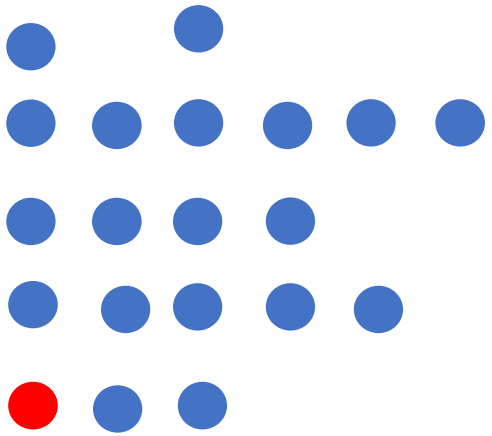
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

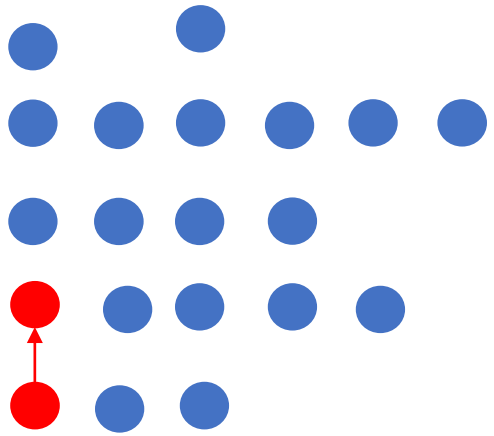
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

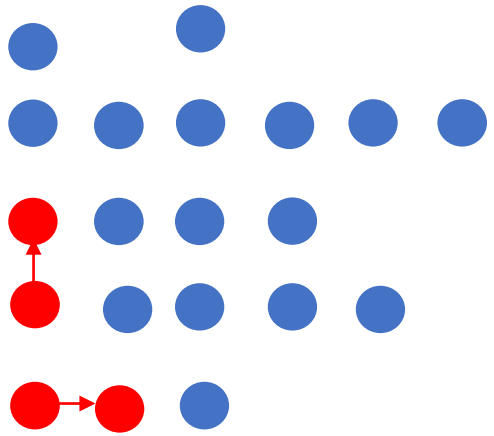
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

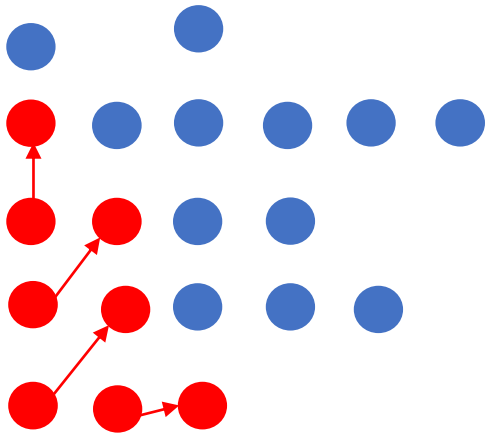
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

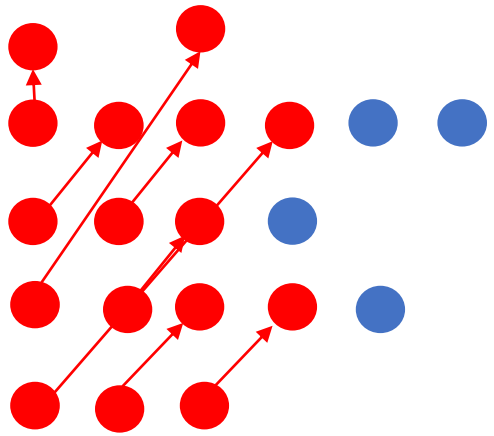
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

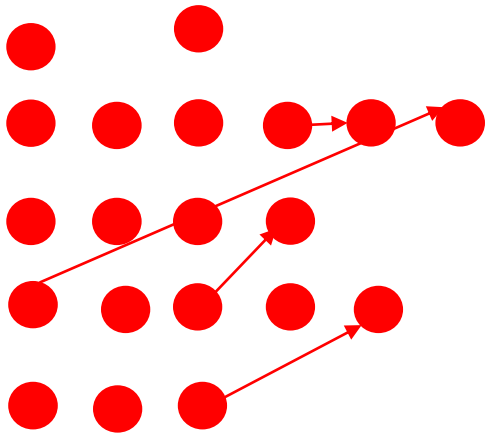
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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!



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?

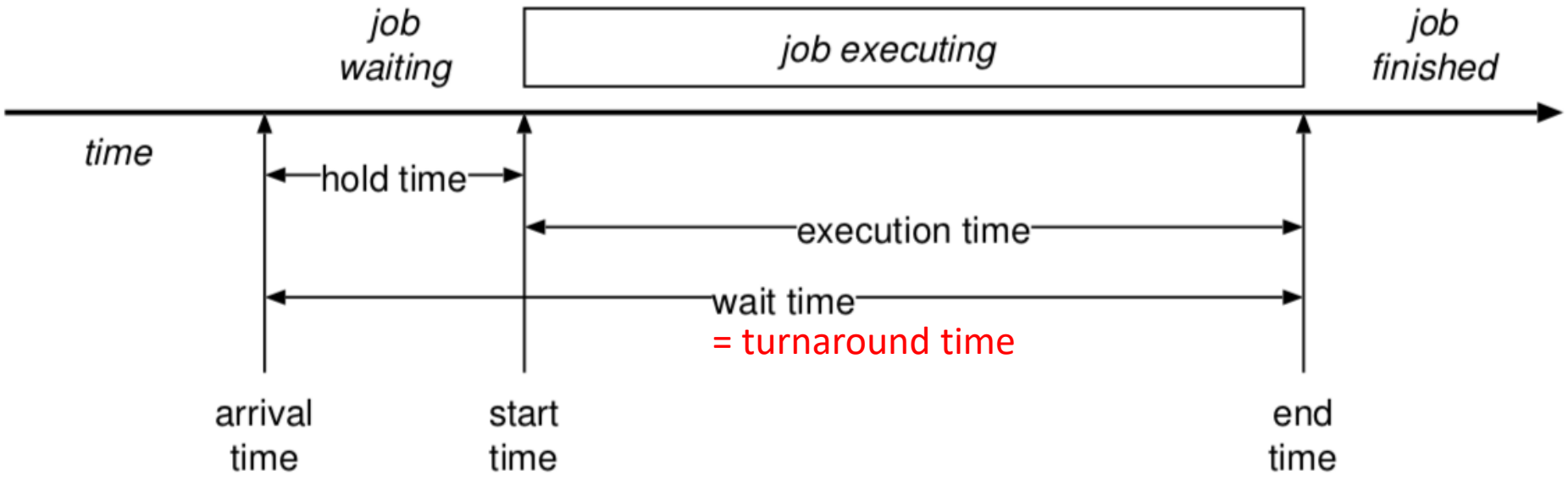
c) Assume that you not only need to solve consensus, but the more challenging task that every node must learn the input values of all nodes. Show that this problem requires at least $n - 1$ time units!

Answers:
b) log(n)
c) Each node needs to learn n-1 values,
so it needs to receive at least n-1 messages.
So in total there need to be n(n-1) messages
received. Because every received message
has been sent at some point, n(n-1) need to get sent
which requires n-1 rounds.

Scheduling

- Question: Deciding how to allocate a single resource among multiple clients.
  - In what order
  - How long

Scheduling Terminology



Response time: The time taken to respond to a request for service

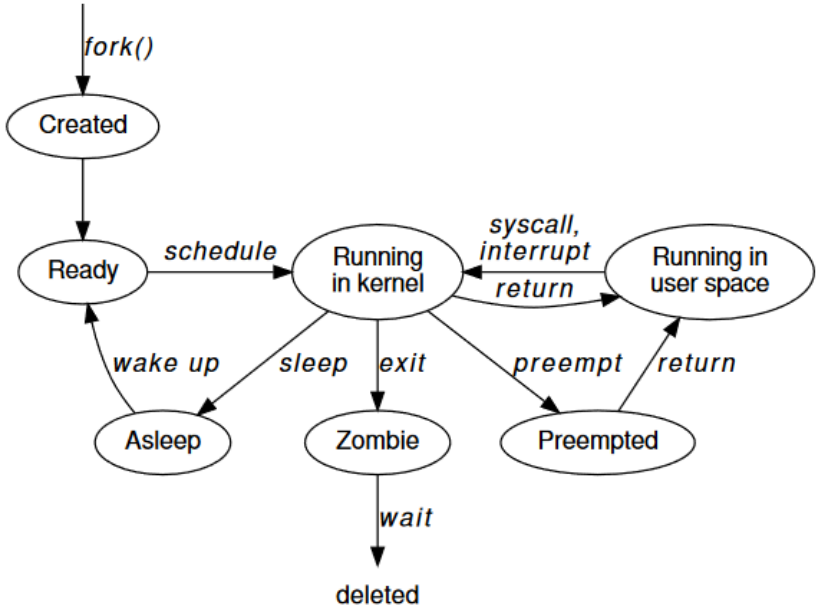
Different kinds of workloads

- **Batch workload**
  - “Run this job to completion and tell me when you’re done”
  - Main goals: throughput (jobs per hour), wait time, turnaround time, utilization
- **Interactive workloads**
  - “Wait for external events and react before the user gets annoyed”
  - Main goals: Response time, proportionality, fairness
- **Soft realtime workloads**
  - “This task must complete in less than 50ms” or “This program must get 10ms CPU every 50 ms”
  - Main goals: deadlines, guarantees, predictability
- **Hard realtime workloads**
  - “Ensure the plane’s control surface moves correctly in response to the pilot’s actions”
  - Embedded Systems

Preemptive vs non-preemptive

- Preemptive:
  - Processes dispatched and de-scheduled without warning
- Non-preemptive
  - Process explicitly has to give up the resource

- Soft-realtime systems are usually pre-emptive
- Hard-realtime systems are often not!



When to schedule

- A running process blocks (or calls yield)
- A blocked process unblocks
- A running or waiting process terminates
- An interrupt occurs (preemption)

FIFO (Batch scheduling)

- **Simplest algorithm!**

- **Example:**
  - Waiting times: 0, 24, 27
  - Avg. = (0+24+27)/3
    = 17

Task	Execution time
A	24
B	3
C	3

FIFO - Problem

- **Different arrival order**

- **Example:**
  - Waiting times: 6, 0, 3
  - Avg. = (0+3+6)/3 = 3

Task	Execution time
A	24
B	3
C	3

Short jobs queue up behind long-running jobs!

Shortest job first (Batch scheduling)

- **Always run process with the shortest execution time.**
- **Optimal: minimizes waiting time (and hence turnaround time)**

Task	Execution time
A	6
B	8
C	7
D	3

→ With preemption: shortest remaining time first

SJF & preemption

- Problem: jobs arrive all the time
- “Shortest remaining time next”
  - New, short jobs may preempt longer jobs already running
- Still not an ideal match for dynamic, unpredictable workloads
  - In particular, interactive ones.

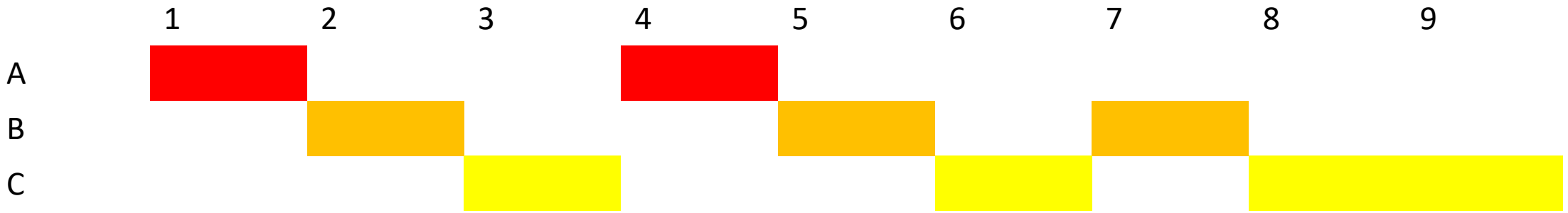
Execution time estimation

- Problem: what is the execution time?
  - For mainframes, could punt to user
  - And charge them more if they were wrong
- For non-batch workloads, use CPU burst times
  - Keep exponential average of prior bursts
- Or just use application information
  - Web pages: size of web page

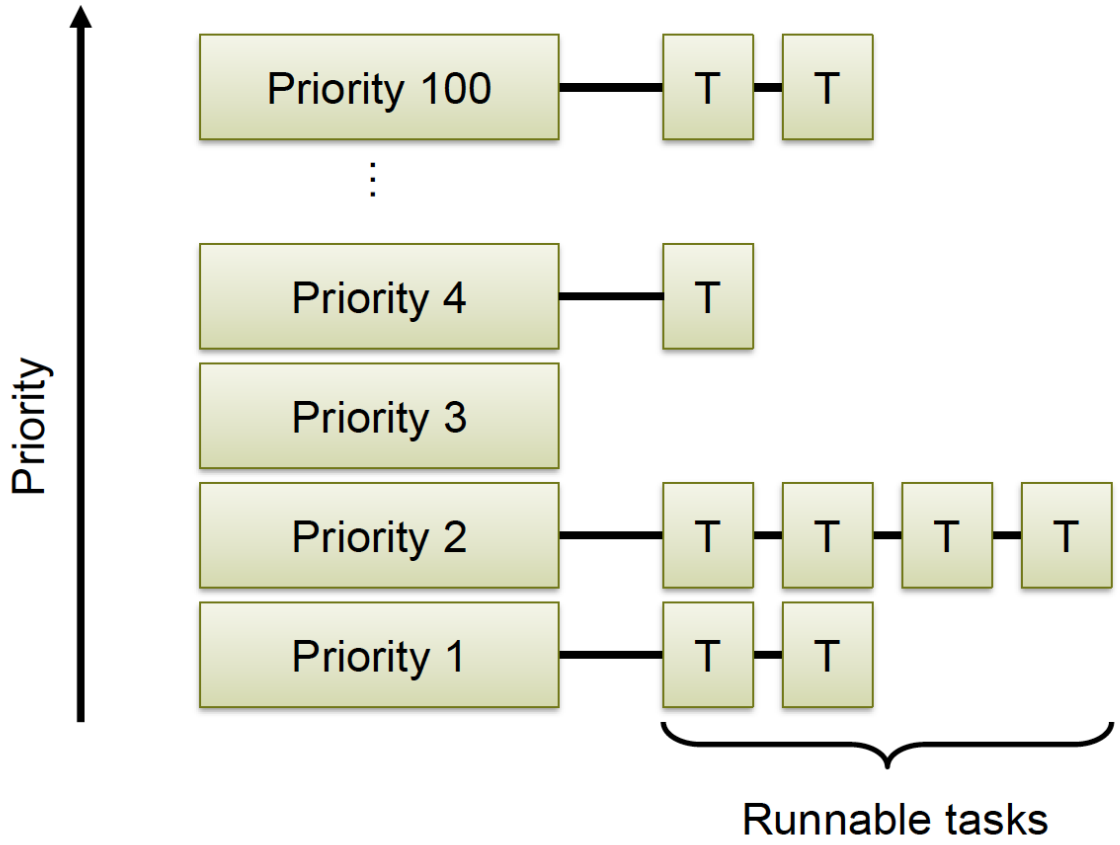
Round robin (interactive workloads)

- Maintains a queue of jobs and schedules them in order
- Higher turnaround than shortest job first, but better response time (goal of interactive workloads)

Process	Execution time (ms)
A	2
B	3
C	4



Priority Queues (interactive workloads)

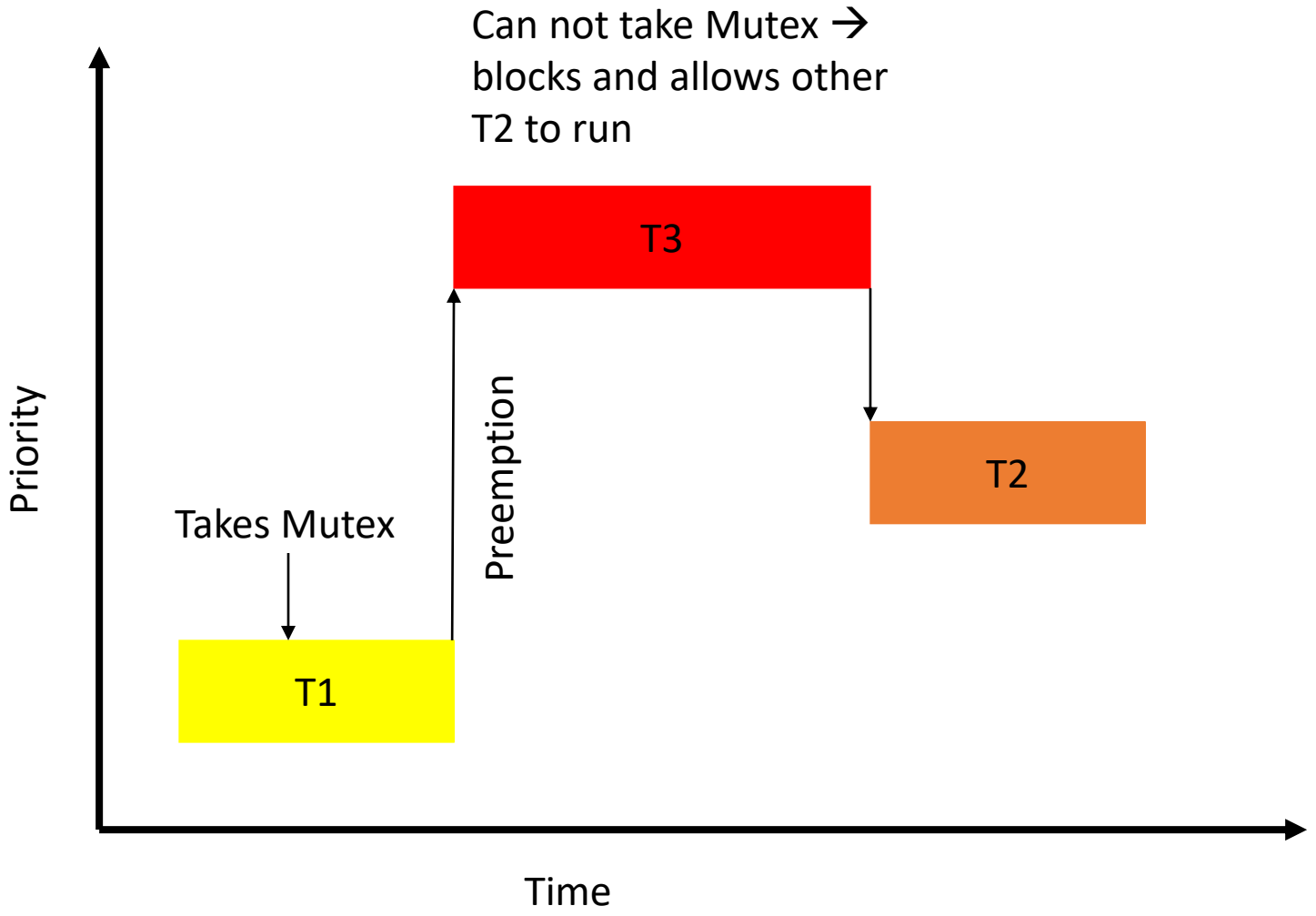


Can schedule differently on different levels, eg. round robin and shortest job first

Priority Queues – Problem

- Starvation– jobs with low priority never get scheduled
  - Solution: jobs gain priority with time, so called “ageing”
    - Reset original priority after being scheduled

Priority Inversion



Multilevel Feedback Queues (interactive workloads)

- Same idea as priority queues
- Idea: prioritize I/0 tasks
  - Solution: use priority queue and reduce priority of processes which use their entire quantum
  - I/0 bound processes tend to only use part of their quantum, so remain high priority

Rate monotonic scheduling (real time scheduling)

- Schedule periodic tasks by always running task with shortest period first.
- Will find schedule if:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

m tasks
C_i is the execution time of i'th task
P_i is the period of i'th task

Earliest Deadline First

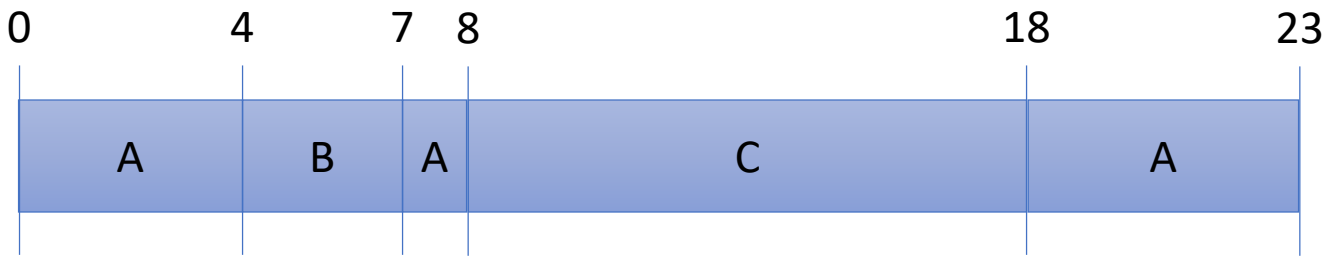
- Schedule task with earliest deadline first (duh..)
  - Dynamic, online.
  - Tasks don't *actually* have to be periodic…
  - More complex - o(n) – for scheduling decisions
- EDF will find a feasible schedule if:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- Which is very handy. Assuming zero context switch time…

Earliest Deadline First

Process	Arrival	Duration	Deadline
A	0	10	25
B	4	3	8
C	8	10	18



What is a device?

Specifically, to an OS programmer:

- Piece of hardware visible from software
- Occupies some location on a bus
- Set of registers
  - Memory mapped or I/O space
- Source of interrupts
- May initiate Direct Memory Access transfers

Algorithm 10.6 Programmed I/O input

```
1: inputs
2:     l: number of words to read from input
3:     d: buffer of size l
4: d ← empty buffer
5: while length(d) < l do
6:     repeat
7:         s ← read from status register
8:     until s indicates data ready
9:     w ← read from data register
10:    d.append(w)
11: end while
12: return
```

Algorithm 10.8 Interrupt-driven I/O cycle

1 Initiating (software)

1: Process A performs a blocking I/O operation
2: OS initiates an I/O operation with the device
3: Scheduler blocks Process A, runs another process

2 Processing (hardware)

4: Device performs the I/O operation
5: Raises device interrupt when complete, or an error occurs

3 Termination (software)

6: Currently running process is interrupted
7: Interrupt handler runs, processes any input data
8: Scheduler makes Process A runnable again

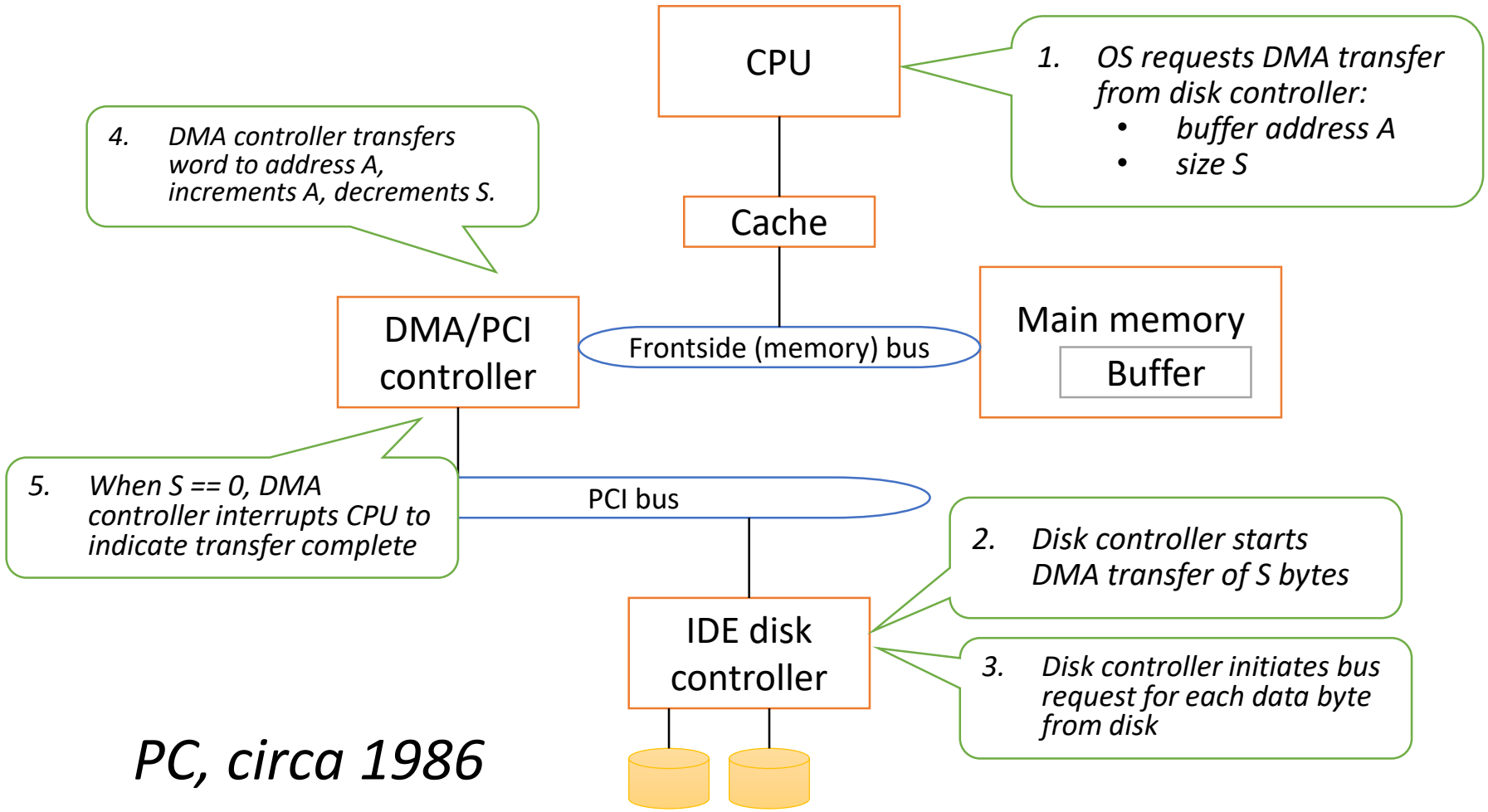
4 Resume (software)

9: Process A restarts execution.

Direct Memory Access (DMA)

- Avoid *programmed I/O* for lots of data (requires CPU)
- Device needs to be able to do DMA (*DMA controller)*
  - Generally built-in these days
- Bypasses CPU to transfer data directly between I/O device and memory
  - Doesn’t take up CPU time
  - Can save memory bandwidth
  - Only one interrupt per transfer

Very simple DMA transfer



PC, circa 1986

I/O Protection

- DMA operations can be dangerous to normal system operations because they directly access memory!
- $\Rightarrow$ need IOMMU (I/O Memory Management Unit) to ensure that devices only access memory that they're supposed to access
- Need to invalidate cache before transfer

Evolution of device I/O

1. Programmed I/O with polling.
   - Polling is too slow (CPU cycles, response latency)
   - $\Rightarrow$ Interrupts notify CPU device needs attention
2. Programmed I/O with interrupts.
   - CPU spends too much time copying data
   - $\Rightarrow$ DMA allows CPU and device to operate in parallel
3. DMA