Computer Systems

Exercise Session 4

Exam question

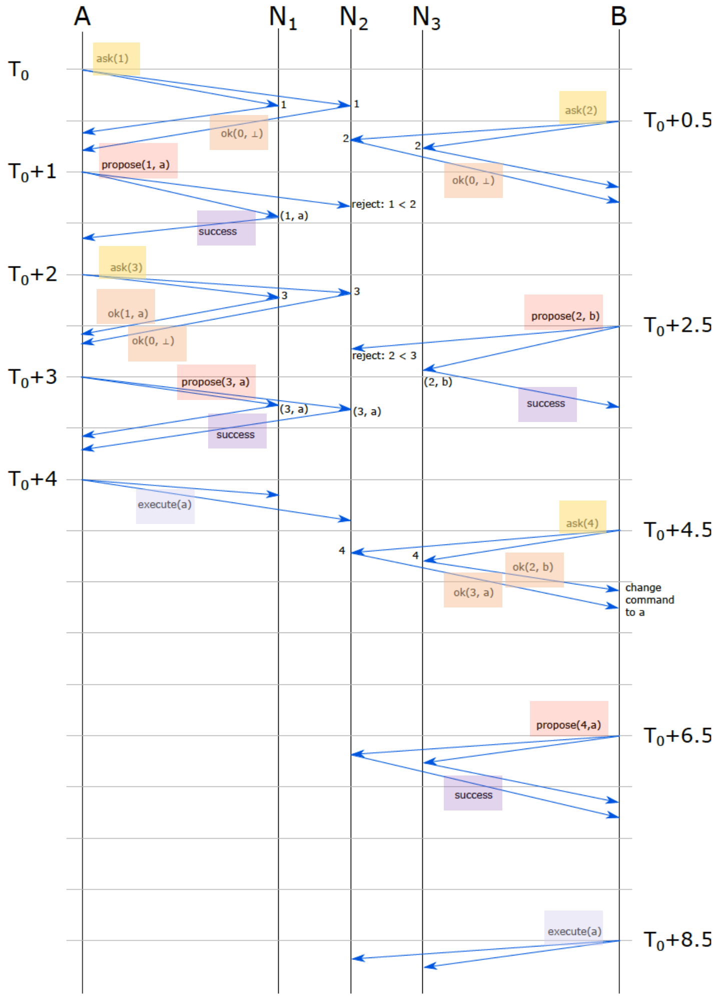
vs

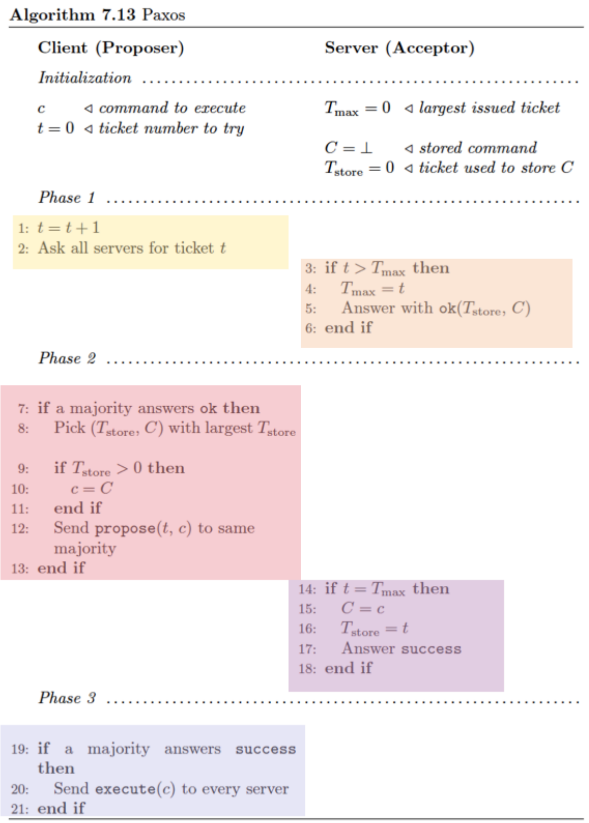
My email: ferraric@ethz.ch

https://hanslodge.com/clipart/8Txr4xdoc.htm
http://clipartmag.com/angel-cartoon-clipart

Last exercise

Exercise: 3 servers N1, N2, N3. Two clients A, B.
SuggestValue(N1, N2, a, 1, 1) on A at time T0,
suggestValue(N2, N3, b, 2, 2) on B at time T0 + 0.5sec





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

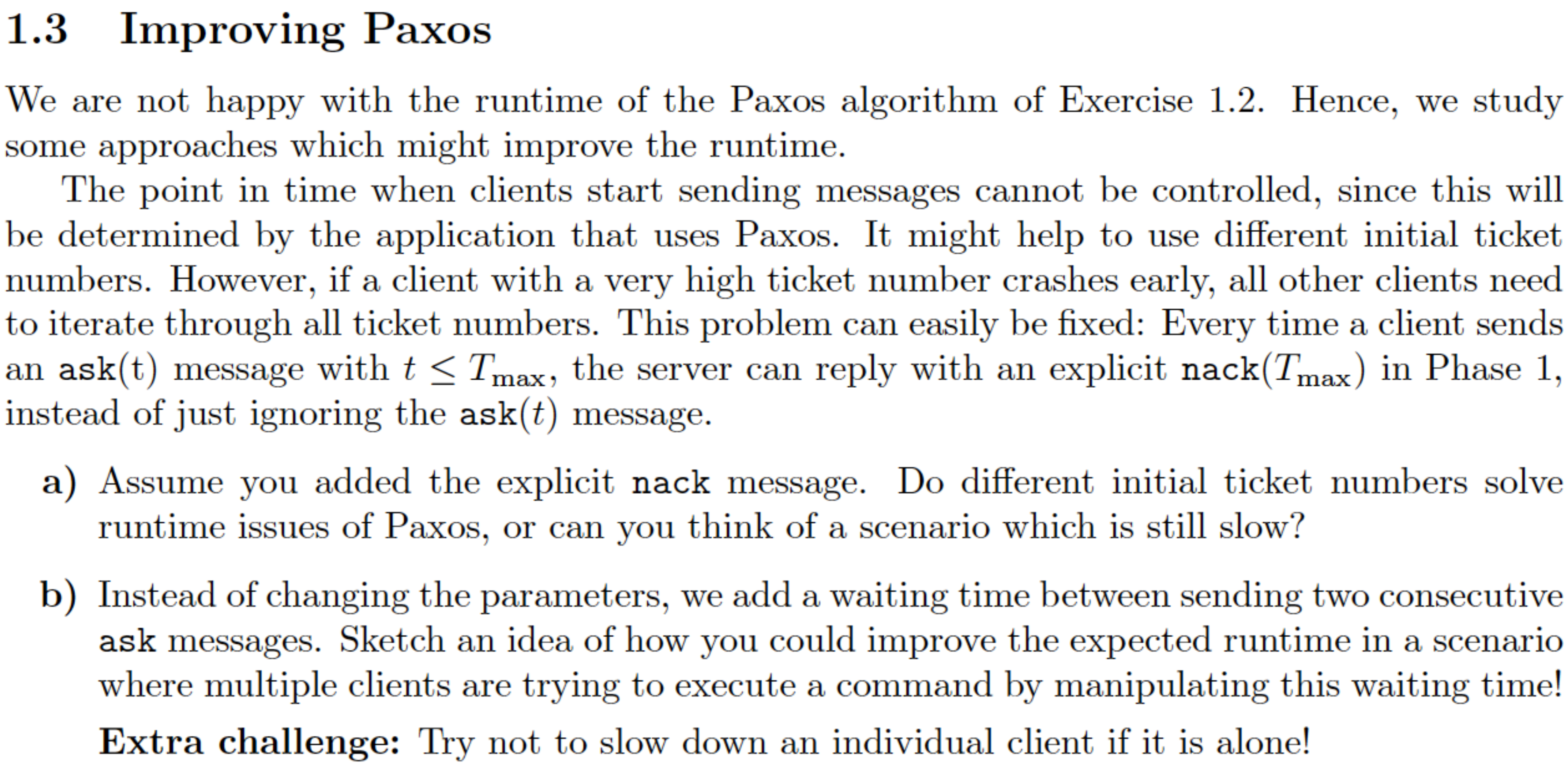
Last exercise

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.

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 `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!

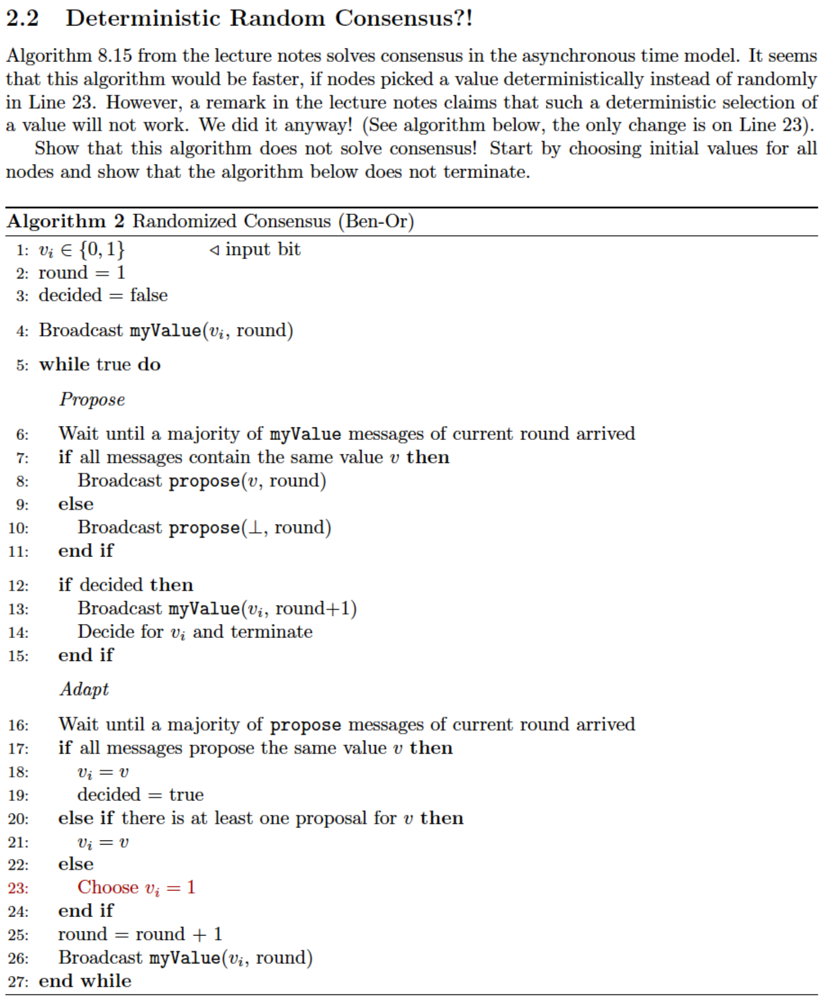
Extra challenge: Try not to slow down an individual client if it is alone!

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



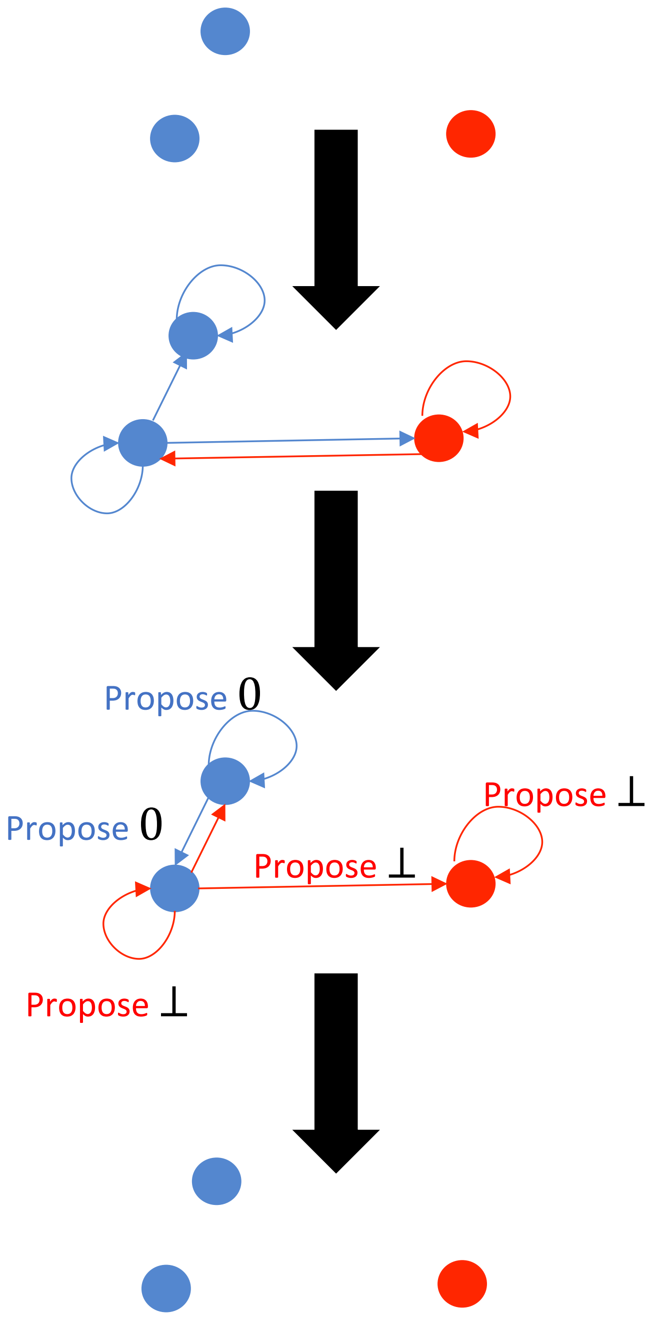
2.2 Deterministic Random Consensus?!

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).

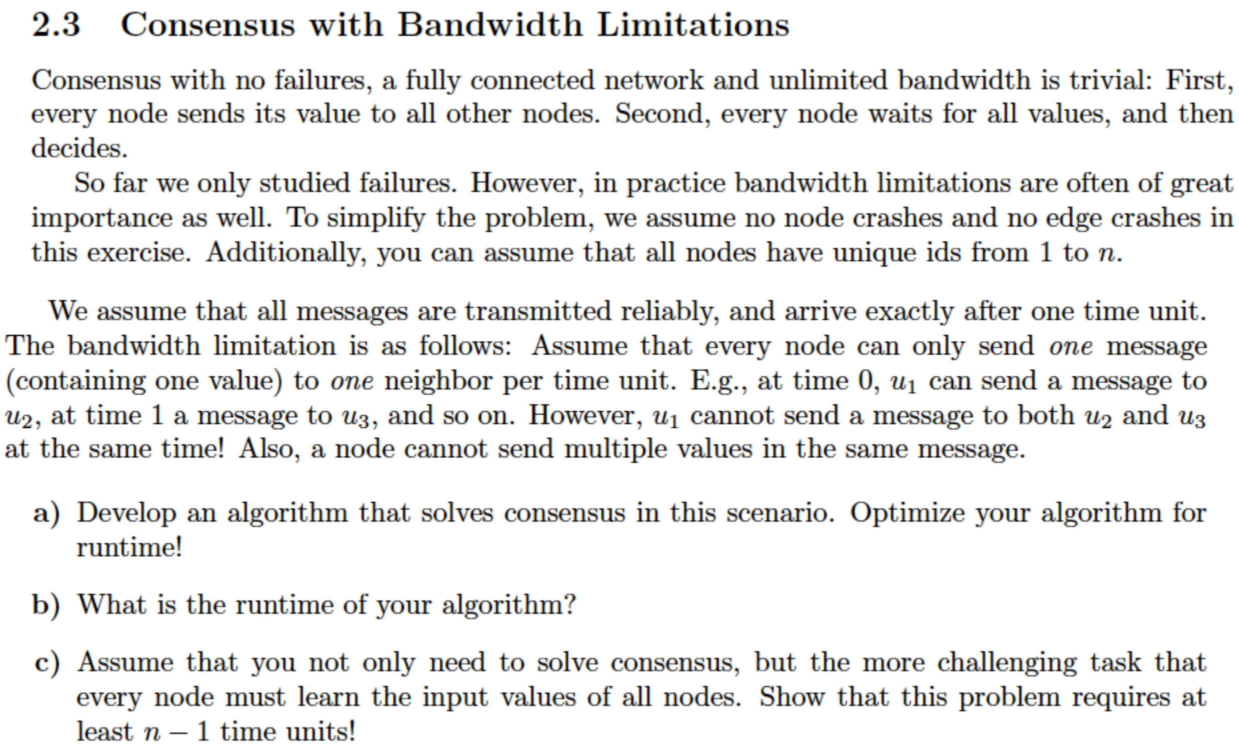
Show that this algorithm does not solve consensus! Start by choosing initial values for all nodes and show that the algorithm below does not terminate.

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.

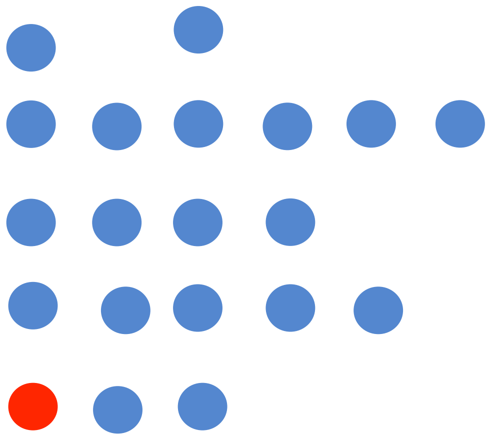
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

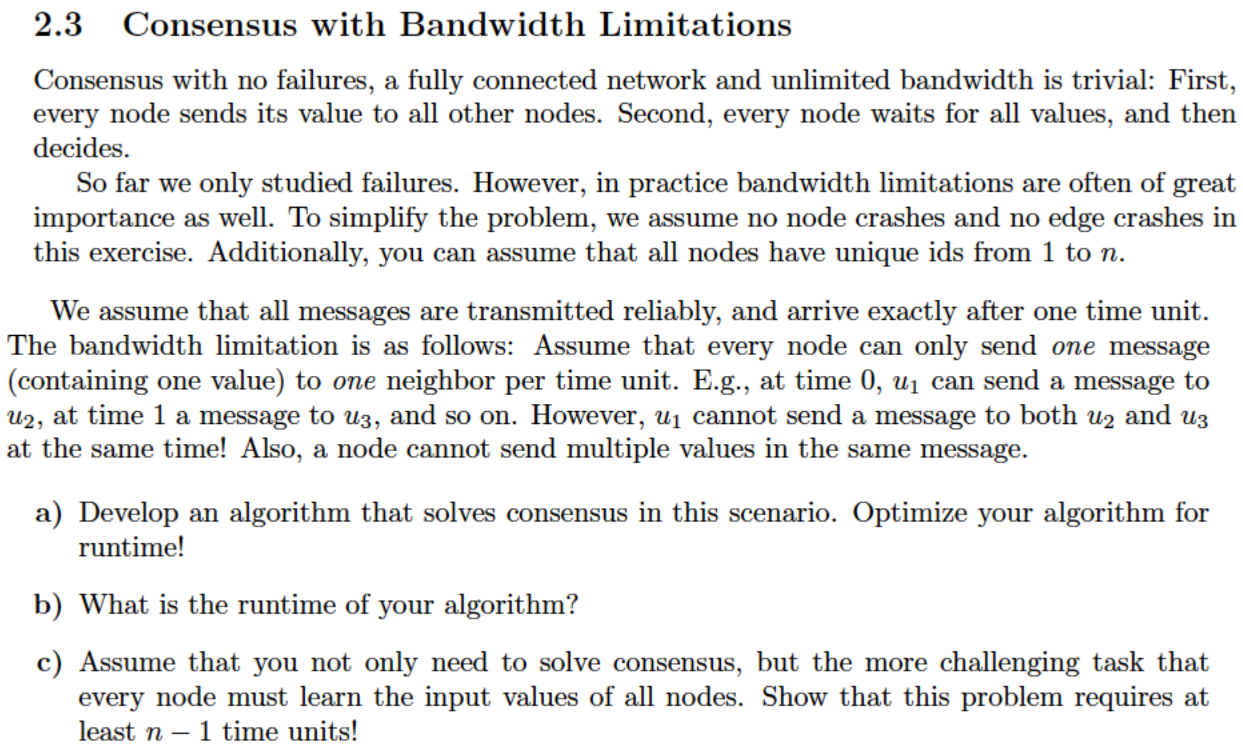
a) Develop an algorithm that solves consensus in this scenario. Optimize your algorithm for runtime!

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.

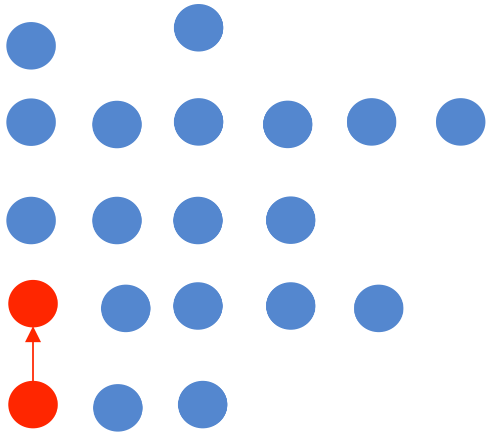
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

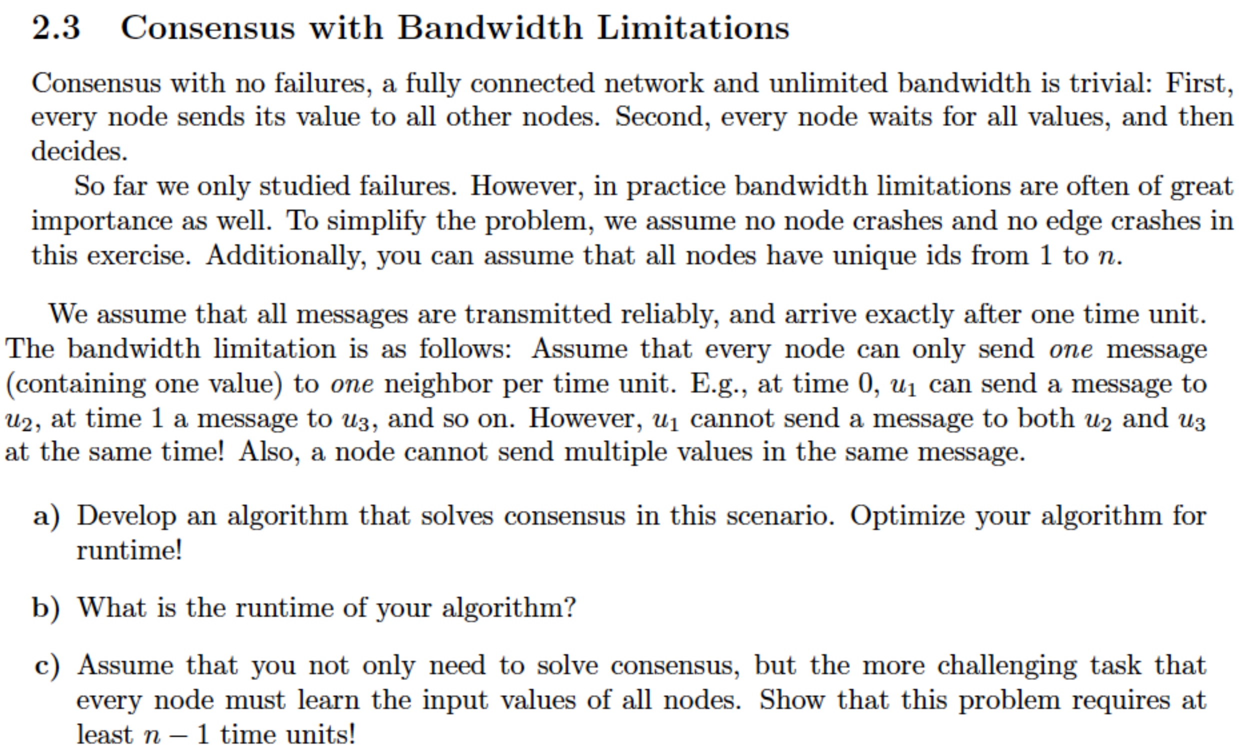
a) Develop an algorithm that solves consensus in this scenario. Optimize your algorithm for runtime!

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.

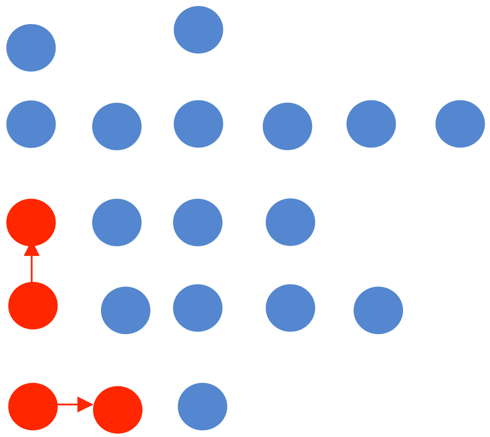
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

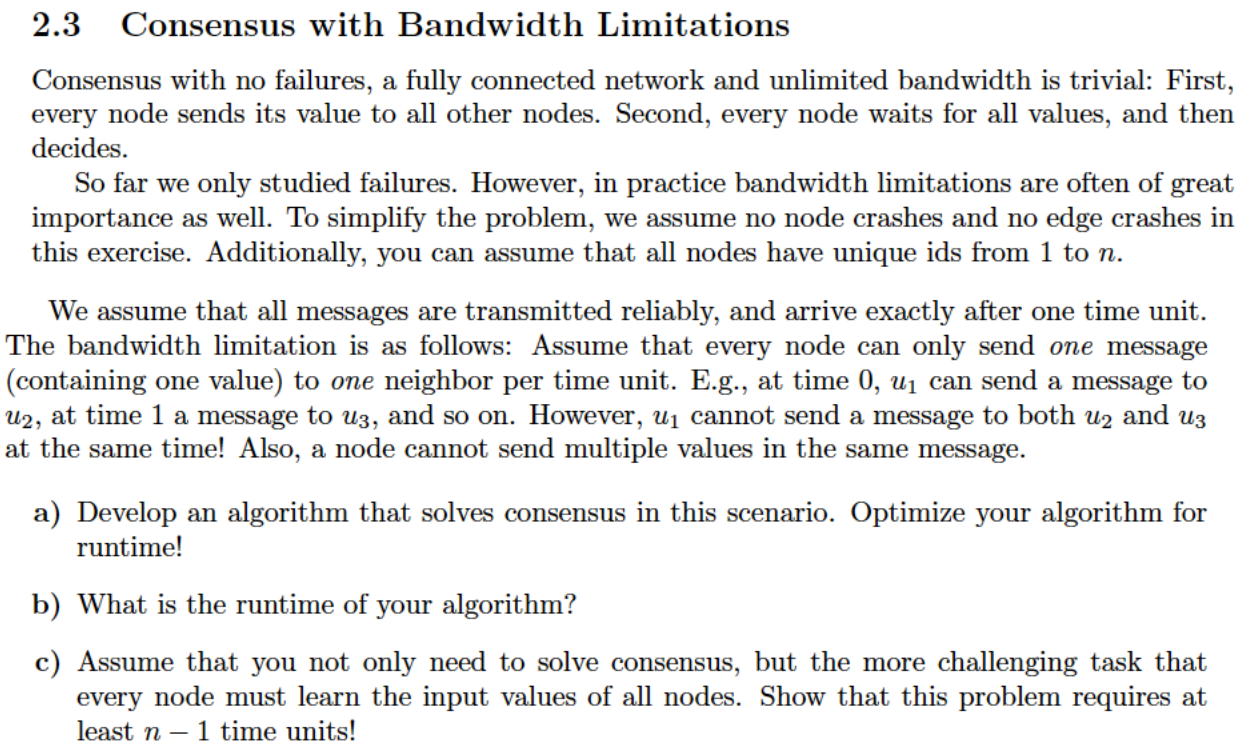
a) Develop an algorithm that solves consensus in this scenario. Optimize your algorithm for runtime!

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.

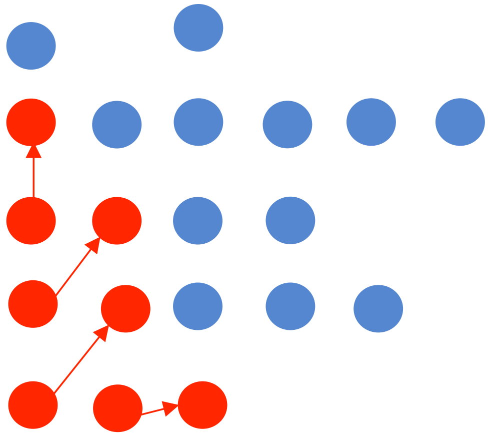
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

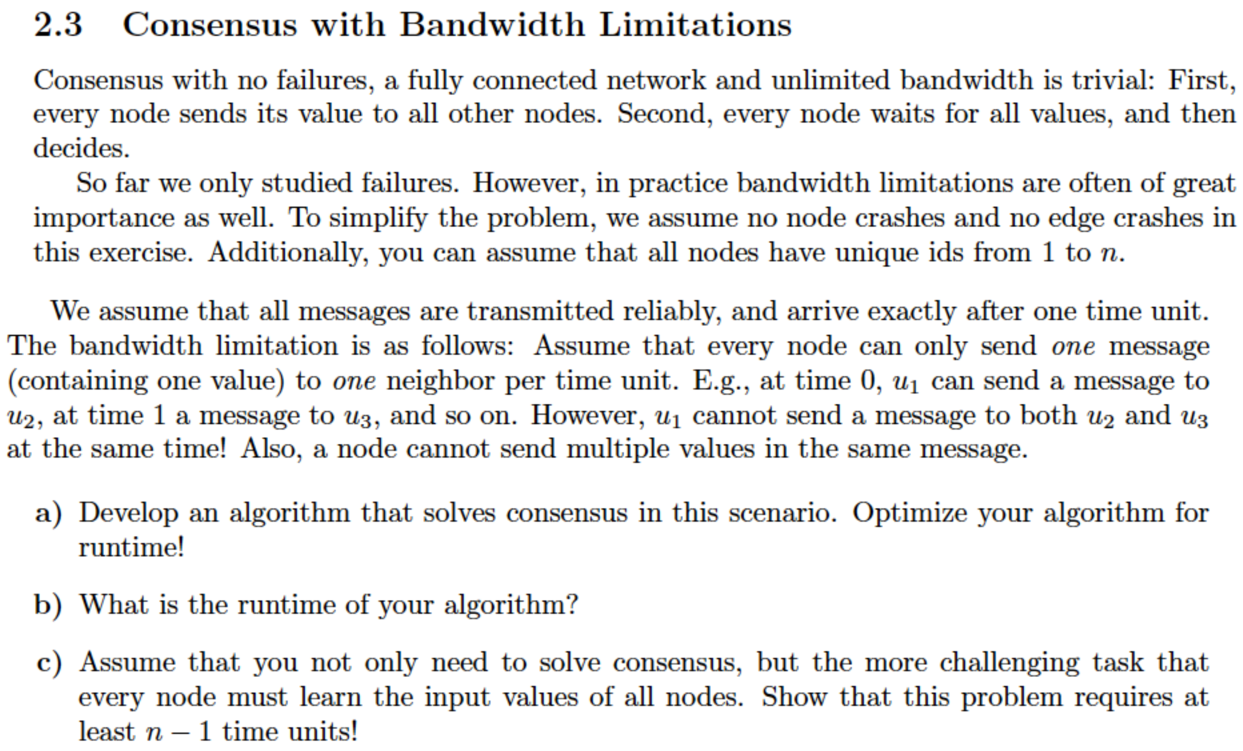
a) Develop an algorithm that solves consensus in this scenario. Optimize your algorithm for runtime!

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.

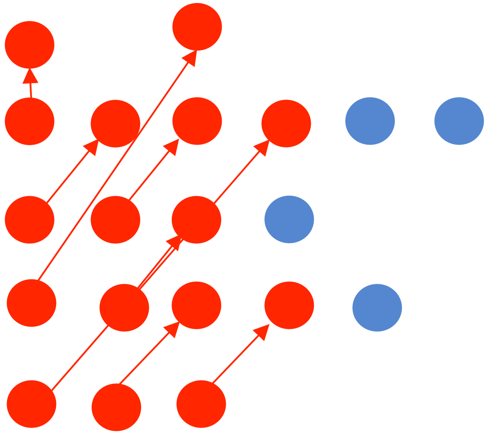
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

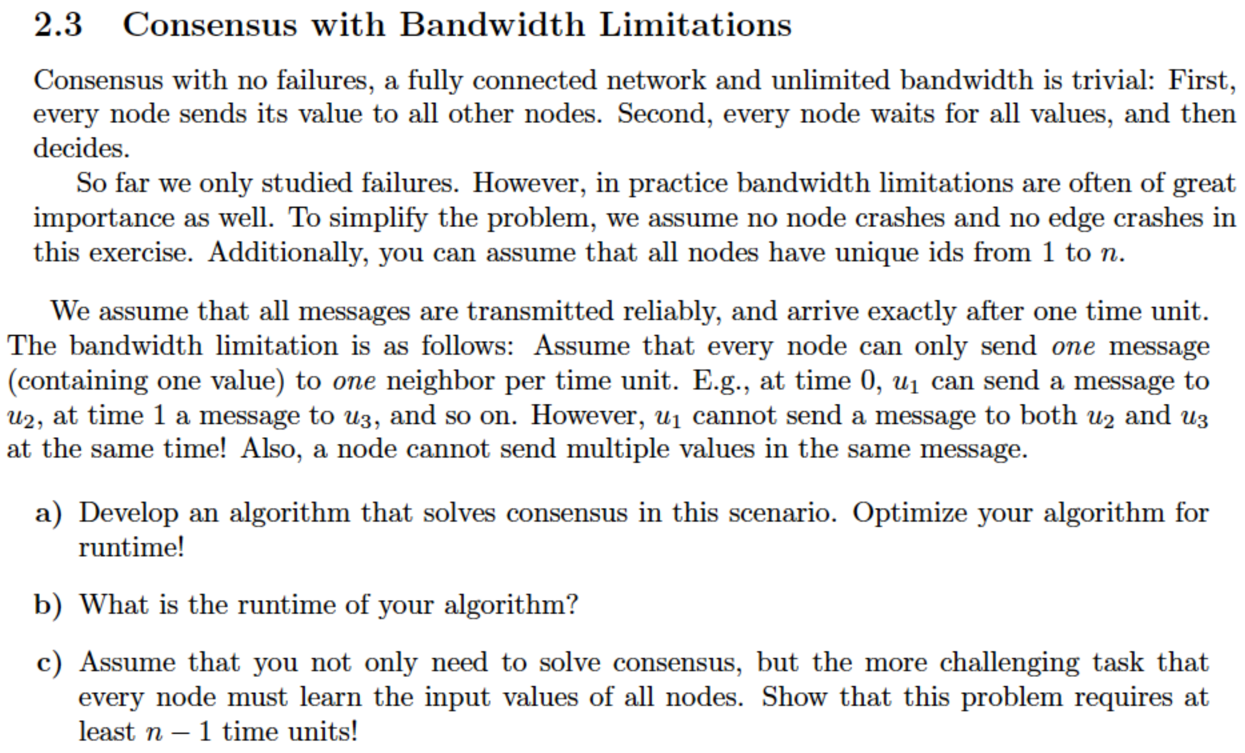
a) Develop an algorithm that solves consensus in this scenario. Optimize your algorithm for runtime!

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.

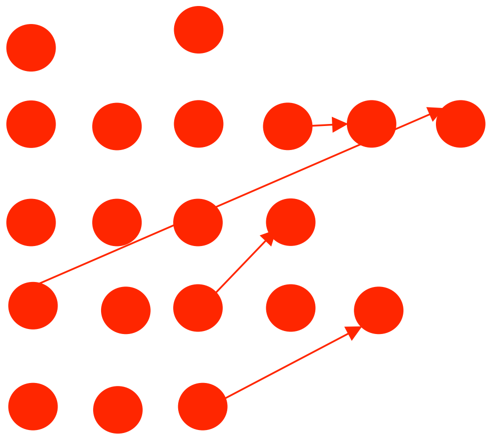
So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

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.

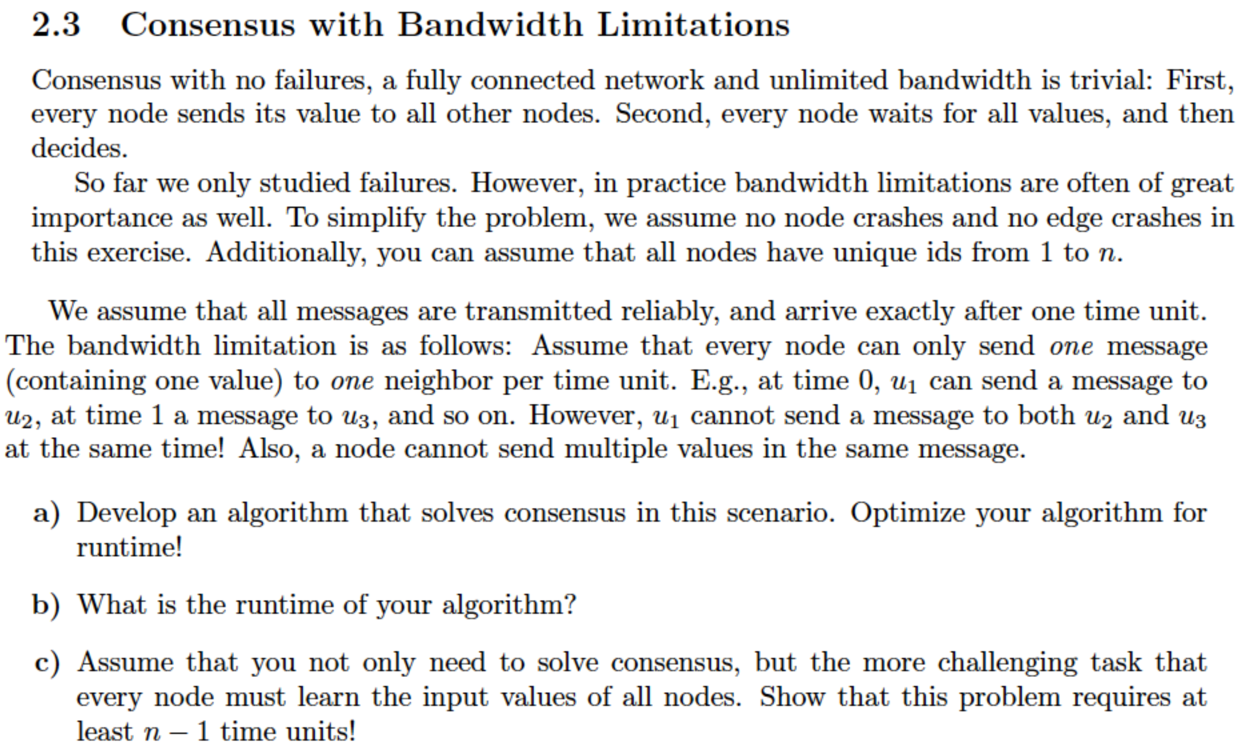
a) Develop an algorithm that solves consensus in this scenario. Optimize your algorithm for runtime!

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!



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?

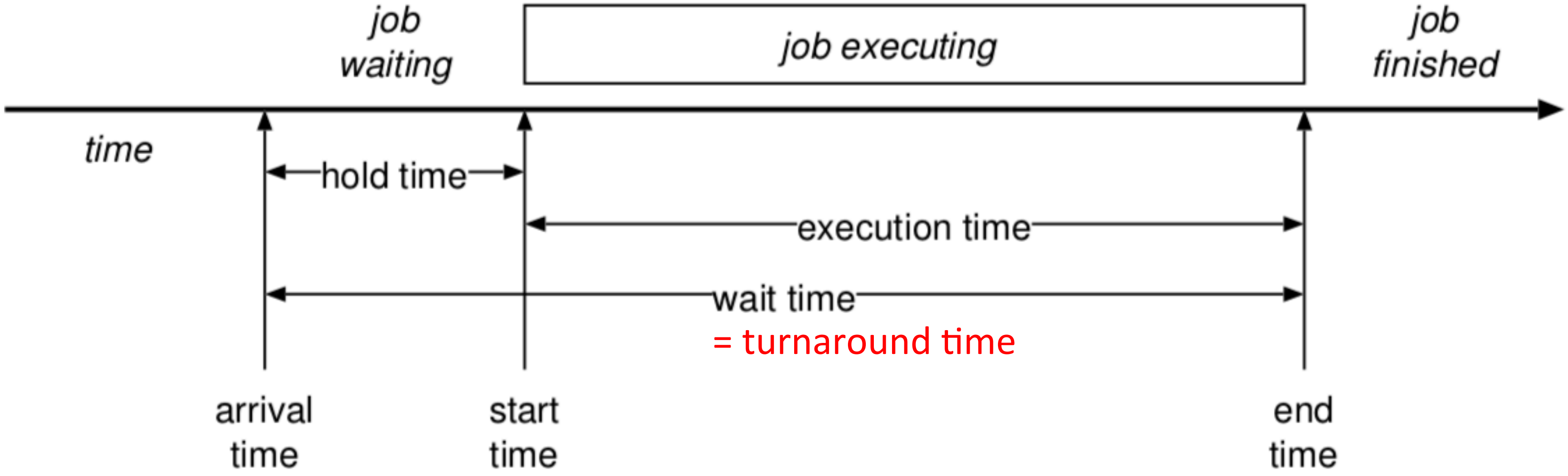
c) Assume that you not only need to solve consensus, but the more challenging task that every node must learn the input values of all nodes. Show that this problem requires at least $n - 1$ time units!

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

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

Scheduling Terminology



= turnaround time

Response time: worst case hold time (only used for interactive workloads)

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
- **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”
  - Difficult, not covered

Preemptive vs non-preemptive

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

When scheduling decisions are made

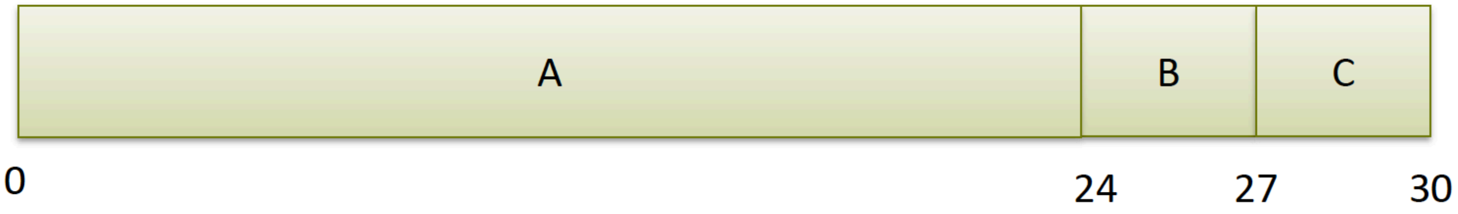
- A running process blocks (or calls yield)
- A blocked process unblocks
- A running or waiting process terminates
- An interrupt occurs (preemption)

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

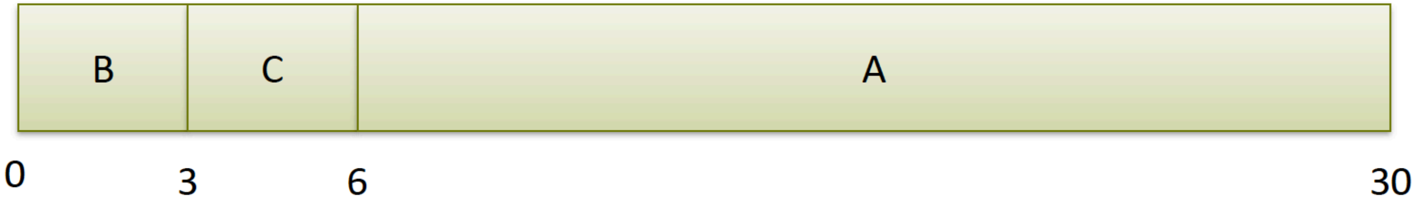


FIFO - Problem

- **Different arrival order**

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

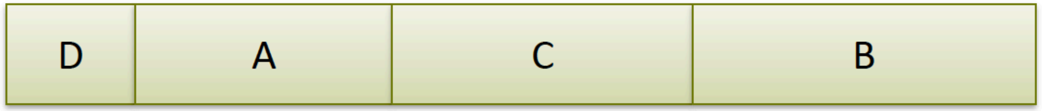
Task	Execution time
A	24
B	3
C	3



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

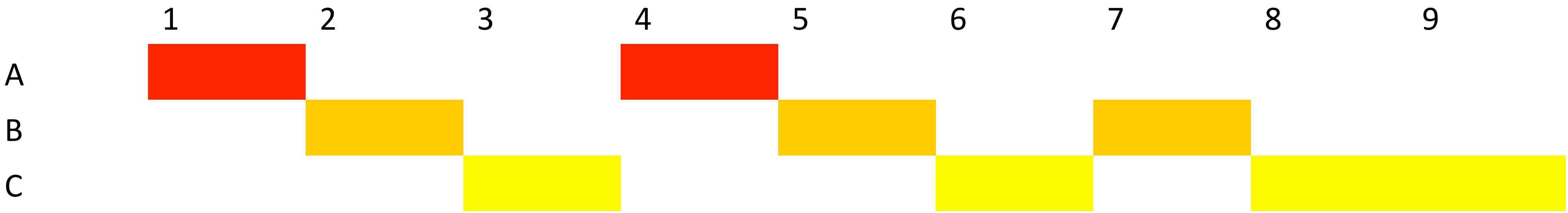


→ With preemption: shortest remaining time first

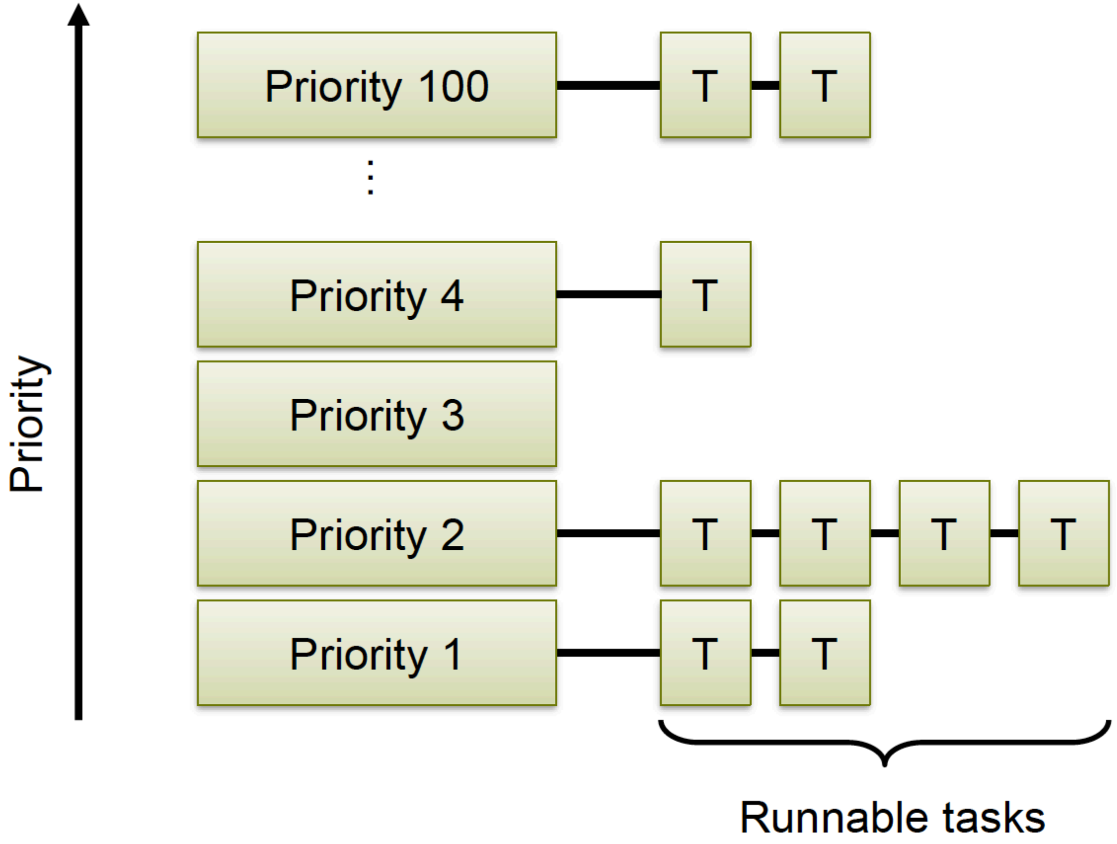
Round robin (interactive workloads)

- 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)

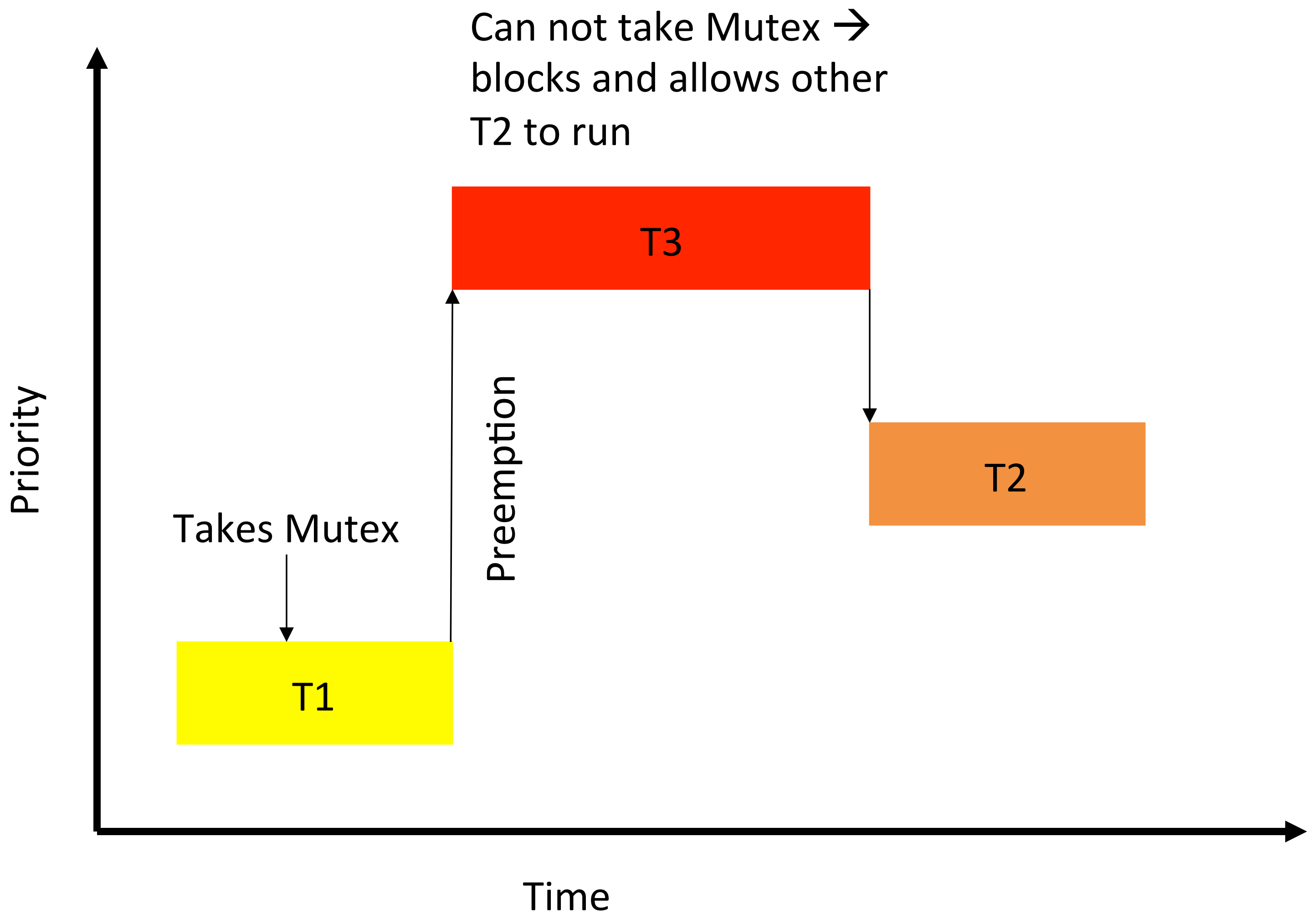


Can schedule differently on different levels, eg. round robin and shortest job first

Priority Queues – Problem

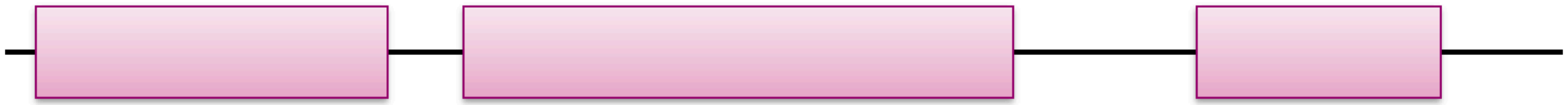
- Starvation– jobs with low priority never get scheduled
    - Solution: jobs gain priority with time, so called “ageing”
        - Reset original priority after being scheduled

Priority Inversion

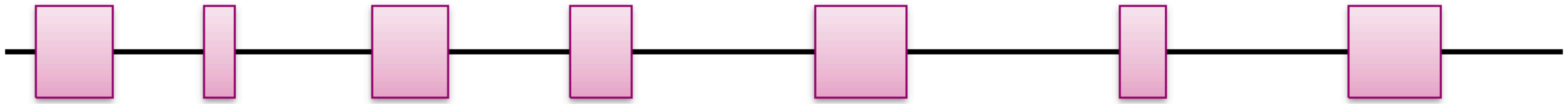


CPU and I/O bound tasks

CPU-bound task:



I/O-bound task:



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 (real time scheduling)

- Schedule tasks with earliest deadline first
- Will find schedule if:

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

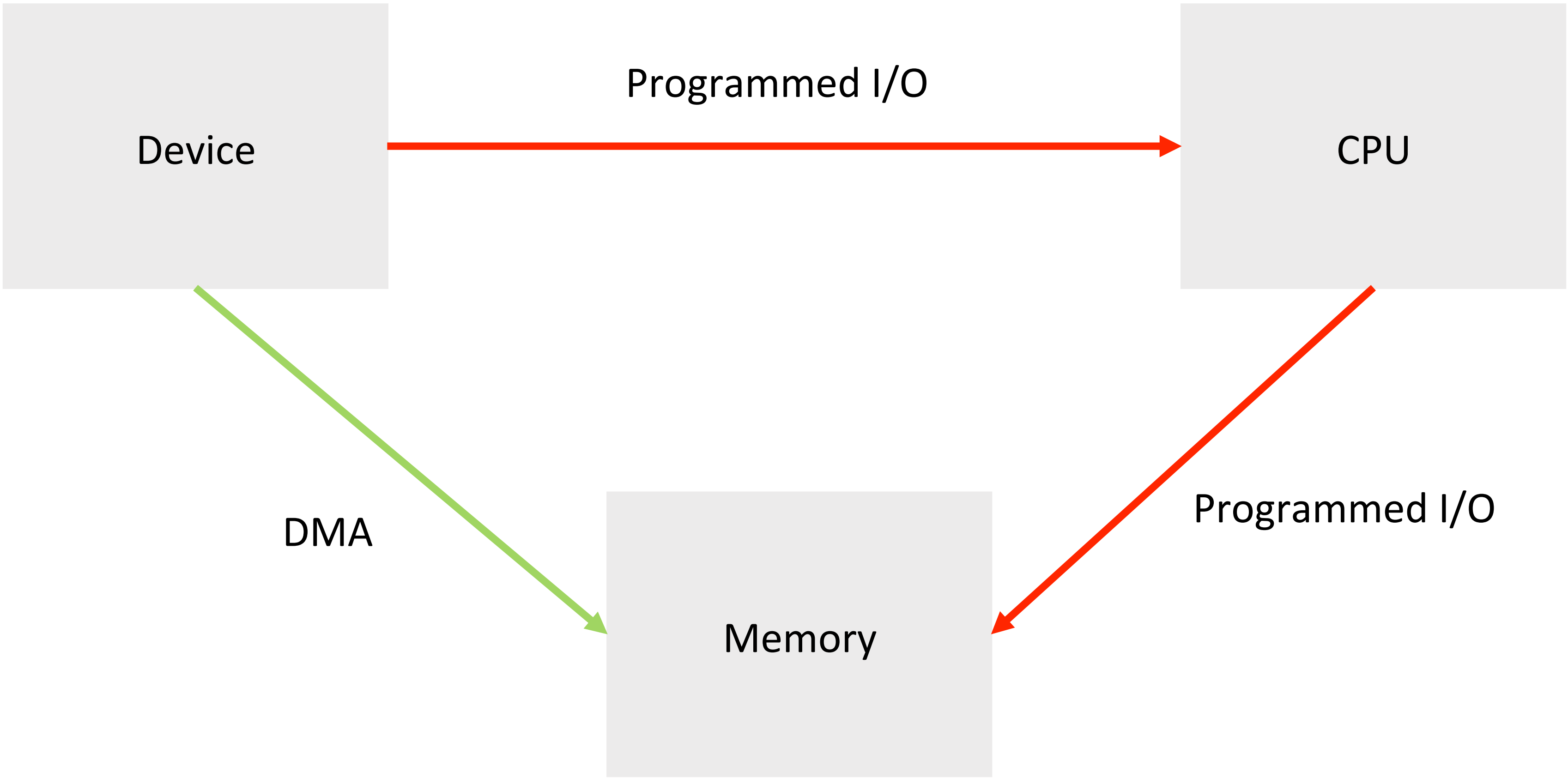
m tasks
C_i is the execution time of i’th task
P_i is the period of i’th task

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

Programmed I/O



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
  - Only one interrupt per transfer

I/O Protection

- DMA operations can be dangerous to normal system operations because they directly access memory!
- ⟹ 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)
   - ⟹ Interrupts notify CPU device needs attention
2. Programmed I/O with interrupts.
   - CPU spends too much time copying data
   - ⟹ DMA allows CPU and device to operate in parallel
3. DMA