Computer Systems

Exercise 7

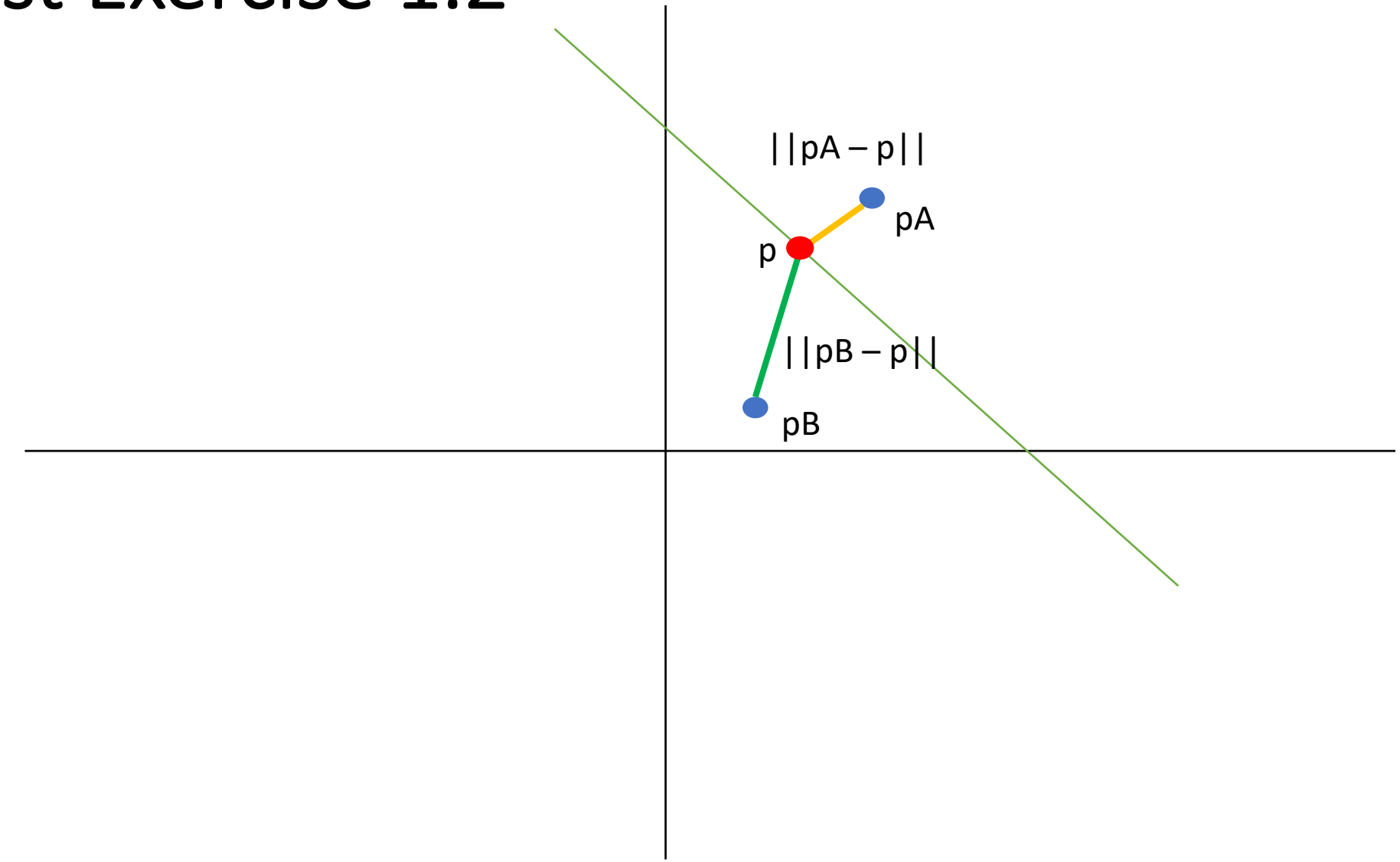
Last Exercise

1.2 Time Difference of Arrival

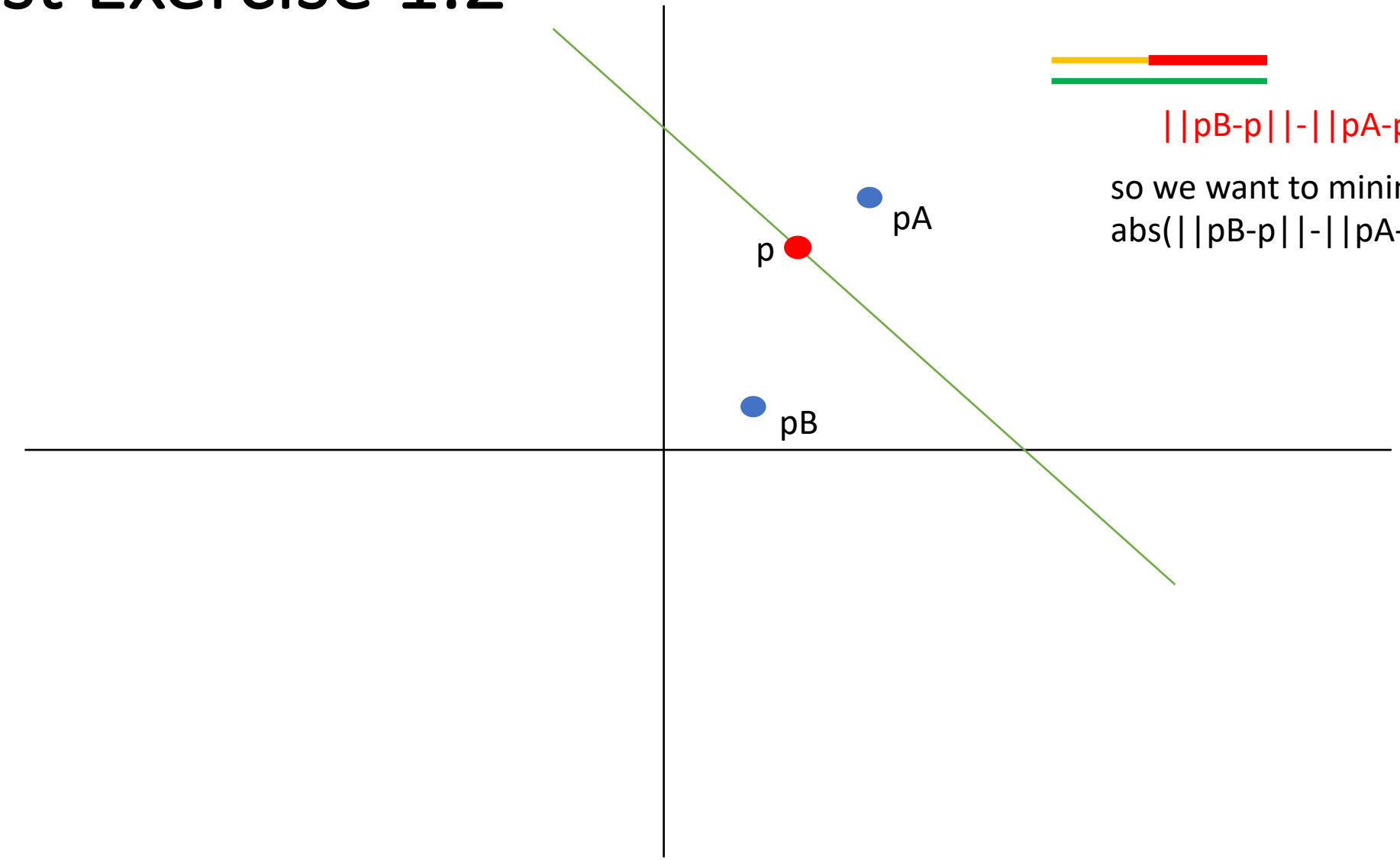
Assume you are located on a line $y = -x + 8$ km in the two dimensional plane. You receive the GPS signals from satellites A and B. Both signals are transmitted exactly at the same time t by both satellites. You receive the signal from satellite A 3.3 µs before the signal of satellite B. At time t, satellite A is located at $p_A = (6\text{ km}, 6\text{ km})$ and satellite B is located at $p_B = (2\text{ km}, 1\text{ km})$, in the plane.

a) Formulate the least squares problem to find your location.

Last Exercise 1.2



Last Exercise 1.2



||pB-p||-||pA-p|| = 1km

so we want to minimize:
abs(||pB-p||-||pA-p|| - 1)

Last Exercise

1.2 Time Difference of Arrival

Assume you are located on a line $y = -x + 8$ km in the two dimensional plane. You receive the GPS signals from satellites A and B. Both signals are transmitted exactly at the same time t by both satellites. You receive the signal from satellite A 3.3 µs before the signal of satellite B. At time t, satellite A is located at $p_A = (6\text{ km}, 6\text{ km})$ and satellite B is located at $p_B = (2\text{ km}, 1\text{ km})$, in the plane.

a) Formulate the least squares problem to find your location.

b) Are you more likely to be at position (2 km, 6 km) or (4 km, 4 km)?

Residual at (2,6): abs(||(2,1)-(2,6)||-||(6,6)-(2,6)||-1) = 5 – 4 – 1 = 0
Residual at (4,4): abs(||(2,1)-(4,4)||-||(6,6)-(4,4)||-1) = 3.6 – 2.8 – 1 = -0.2

Last Exercise

1.2 Time Difference of Arrival

Assume you are located on a line $y = -x + 8$ km in the two dimensional plane. You receive the GPS signals from satellites A and B. Both signals are transmitted exactly at the same time t by both satellites. You receive the signal from satellite A 3.3 µs before the signal of satellite B. At time t, satellite A is located at $p_A = (6\text{ km}, 6\text{ km})$ and satellite B is located at $p_B = (2\text{ km}, 1\text{ km})$, in the plane.

a) Formulate the least squares problem to find your location.

b) Are you more likely to be at position (2 km, 6 km) or (4 km, 4 km)?

c) What is the time when receiving the signal from satellite B?

Distance from (2,6) to (2,1) is: 5

Divide by speed of light: $\frac{5km}{3*10^8 m/s} = 16.7\ microseconds$

Time message is received: t + 16.7 microseconds

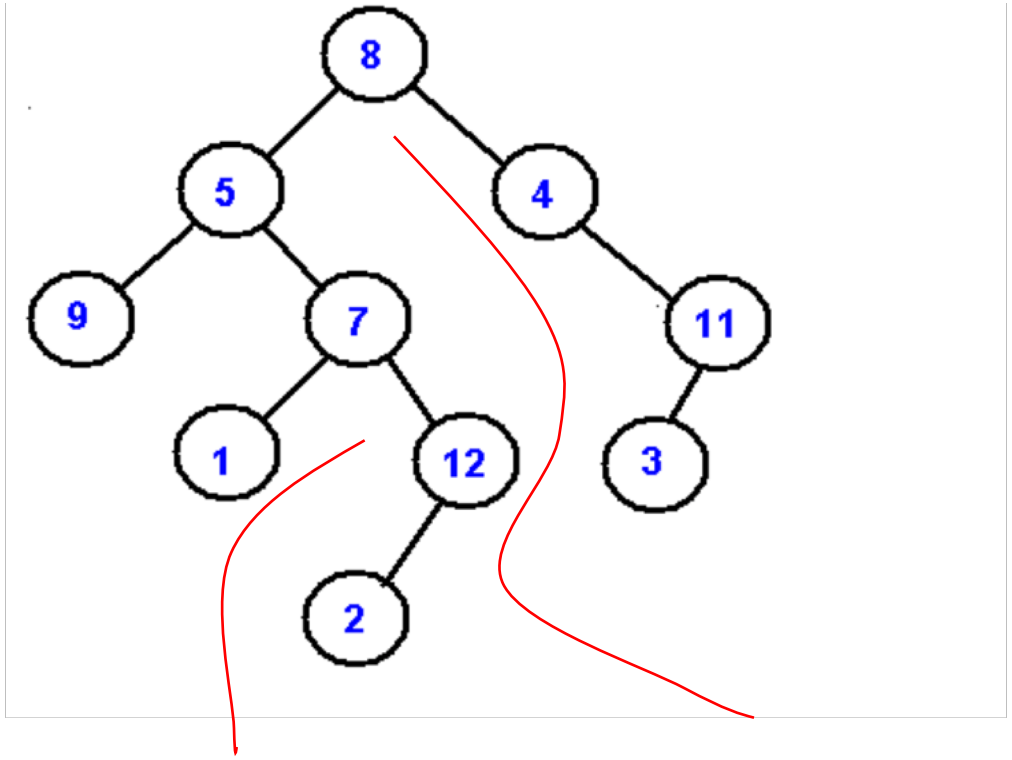
Last Exercise

1.3 Clock Synchronization: Spanning Tree

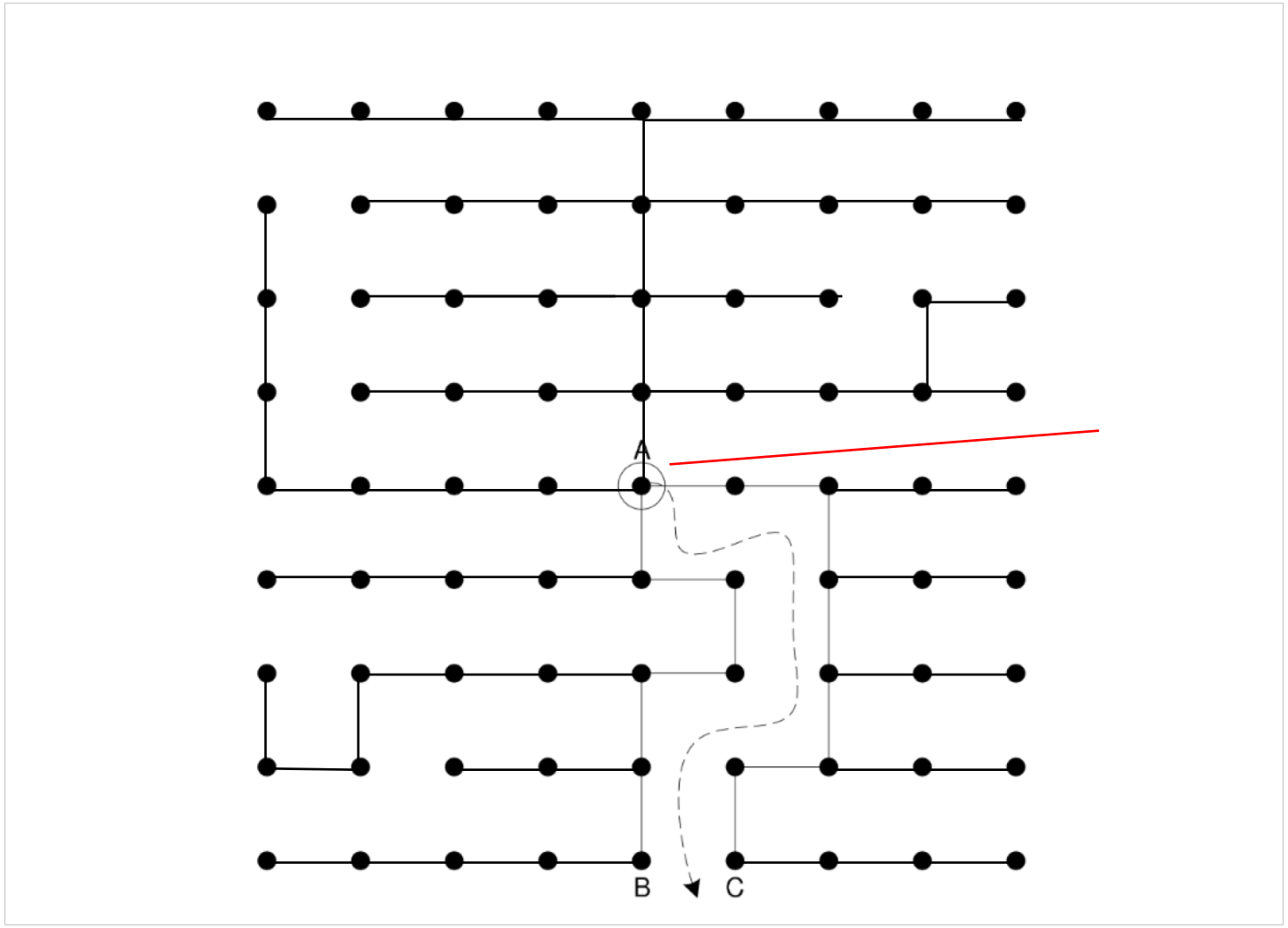
Common clock synchronization algorithms (e.g. TPSN, FTSP) rely on a spanning tree to perform clock synchronization. Finding a good spanning tree for clock synchronization is not trivial. Nodes which are neighbors in the network graph should also be close-by in the resulting tree. Show that in a grid of $n = m \times m$ nodes there exists at least a pair of nodes with a stretch of at least m. The stretch is defined as the hop distance in the tree divided by the distance in the grid.

Last exercise 1.3

General layout of spanning tree:



Last Exercise 1.3



Last Exercise

2.2 Measure of Concurrency from Vector Clocks

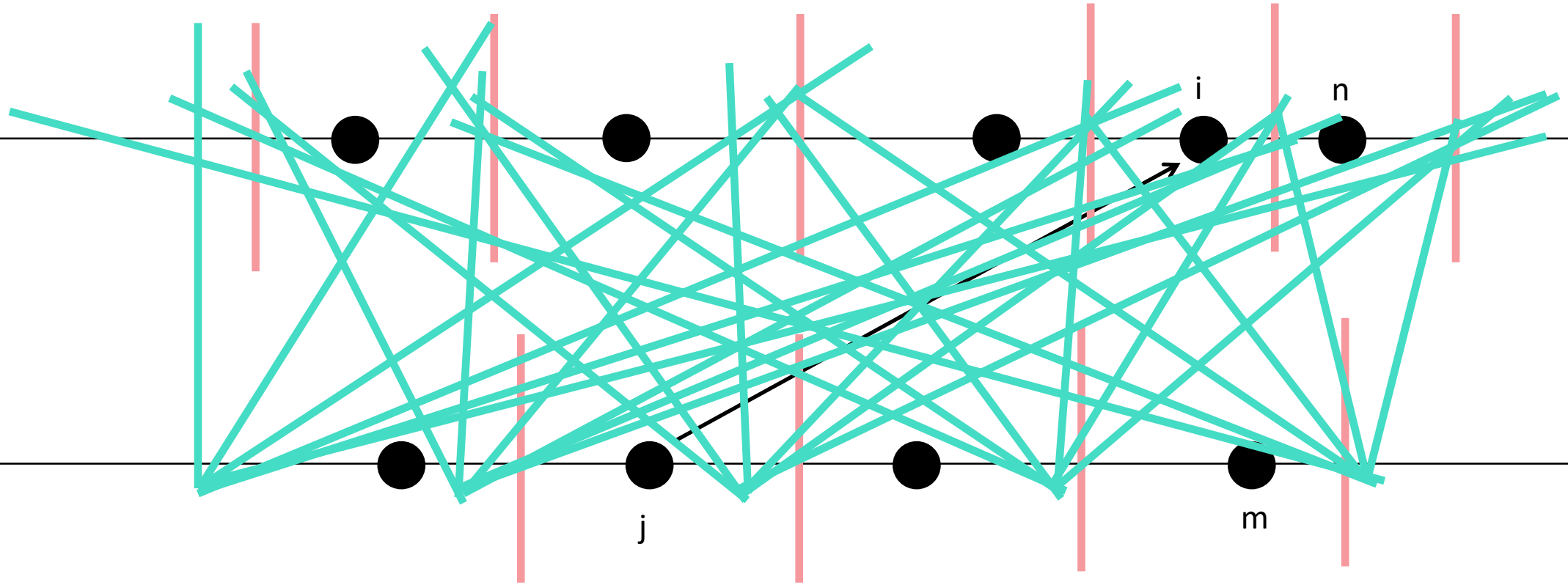
You are given two nodes that each have a vector logical clock that additionally logs the clock state upon receiving a message (see Algorithm 1).

Algorithm 1 Vector clocks with logging

1: (Code for node u)
2: Initialize $c_u[v] := 0$ for all other nodes v.
3: Upon local operation: Increment current local time $c_u[u] := c_u[u] + 1$.
4: Upon send operation: Increment $c_u[u] := c_u[u] + 1$ and include the whole vector c_u as d in message.
5: Upon receive operation: Extract vector d from message and update $c_u[v] := \max(d[v], c_u[v])$ for all entries v. Increment $c_u[u] := c_u[u] + 1$. Save the vector c_u to the log file of node u.

Assume that exactly one message gets send from one to the other node. Given the logs and current vector states of both nodes, write a short program that calculates the measure of concurrency as defined in the script (Definition 3.30). You can use your favorite programming language. The example solution will be in Python.

Last Exercise 2.2



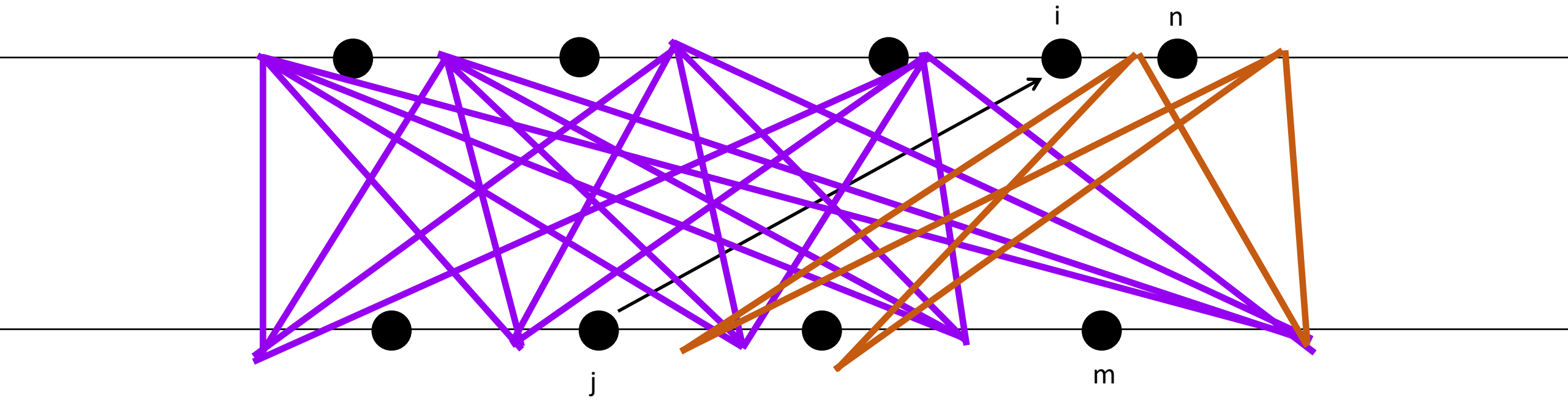
Measure of concurrency: $\frac{M_u - M_s}{M_c - M_s}$

M_s = Nr. sequential snapshots = $n + m + 1$

M_c = Nr. concurrent snapshots = $(m + 1)(n + 1)$

M_u = Nr. snapshots in our system =
$\big(i * (m + 1)\big) + ((n + 1 - i)(m + 1 - j))$

Last Exercise 2.2



Measure of concurrency: $\frac{M_u - M_s}{M_c - M_s}$

M_s = Nr. sequential snapshots = $n + m + 1$

M_c = Nr. concurrent snapshots = $(n + 1)(m + 1)$

M_u = Nr. snapshots in our system =
$\big(i * (m + 1)\big) + ((n + 1 - i)(m + 1 - j))$

Quorum Systems

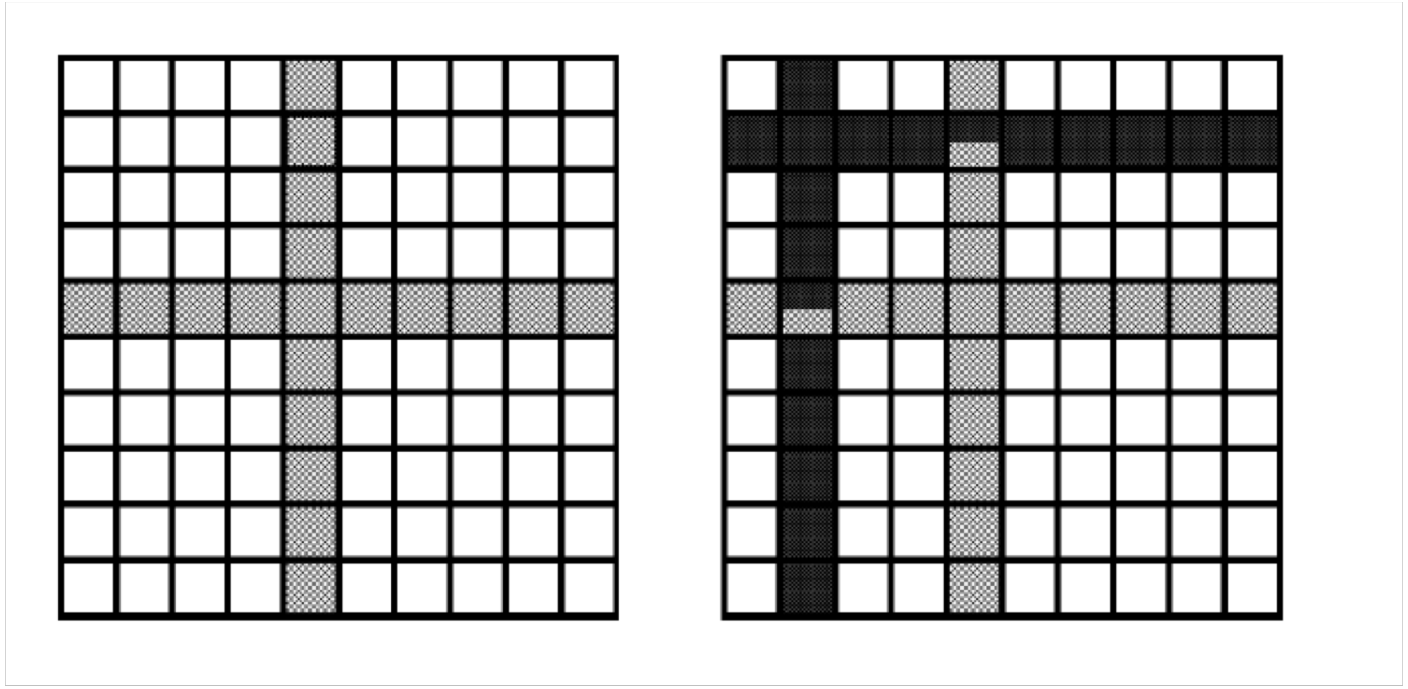
- Load ~ probability
  - Load of access strategy on node: Probability it gets accessed
  - Load on quorum system induced by access strategy: load of node with maximal load
  - Load of quorum system: load induced by access strategy with best access strategy
- Work ~ count
  - Work of quorum: number of nodes
  - Work induced by access strategy: expected number of nodes accessed
  - Work of quorum system: work induced by access strategy with best access strategy

Grid quorum system – Basic Grid

Problem:

- 2 quorums intersect

in two nodes -> deadlock



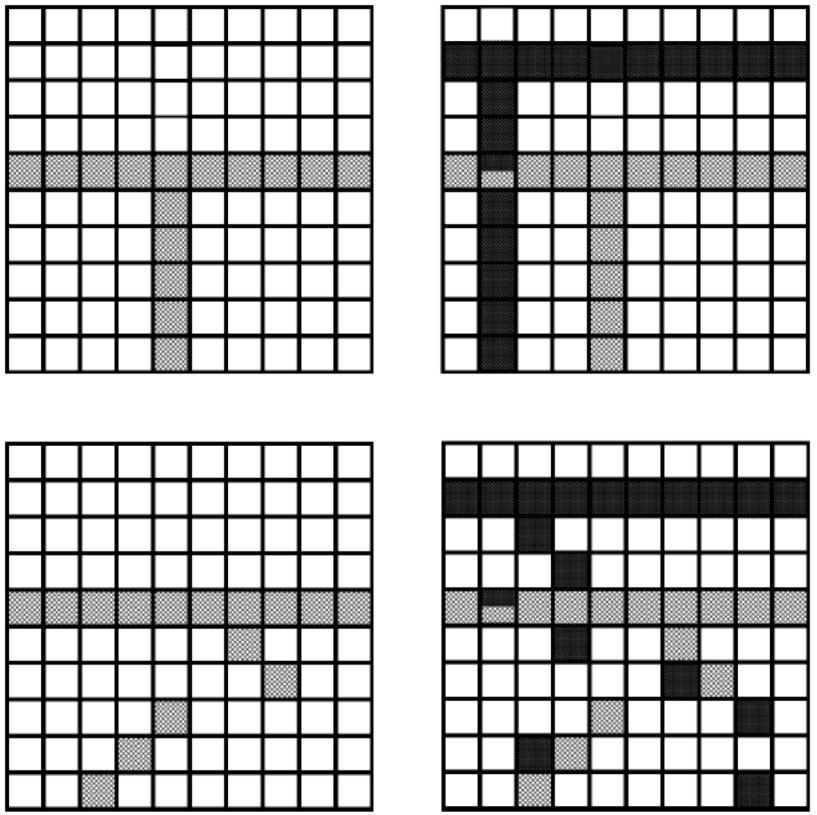
Grid quorum system – Another Grid

Better:

- Two quorums won't run into a deadlock anymore

Problem:

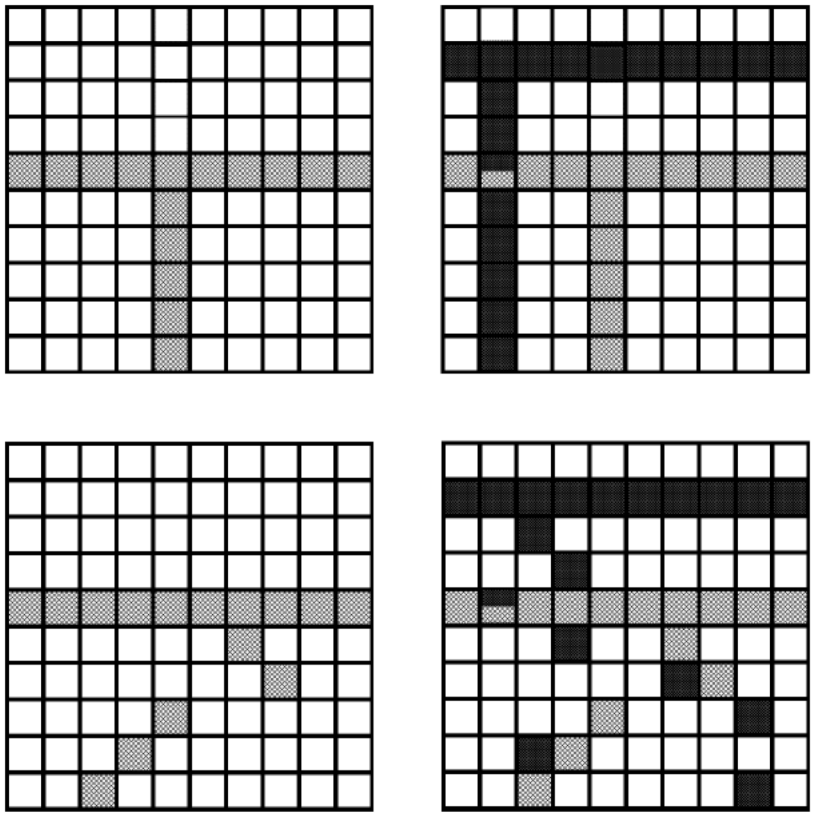
- It can still happen with three or more threads



Grid quorum system – Another Grid

Solution:
Try to get all locks in order (by id), if one is locked release all and start over.

-> at least one quorum will always make progress (the one with highest identifier locked currently)

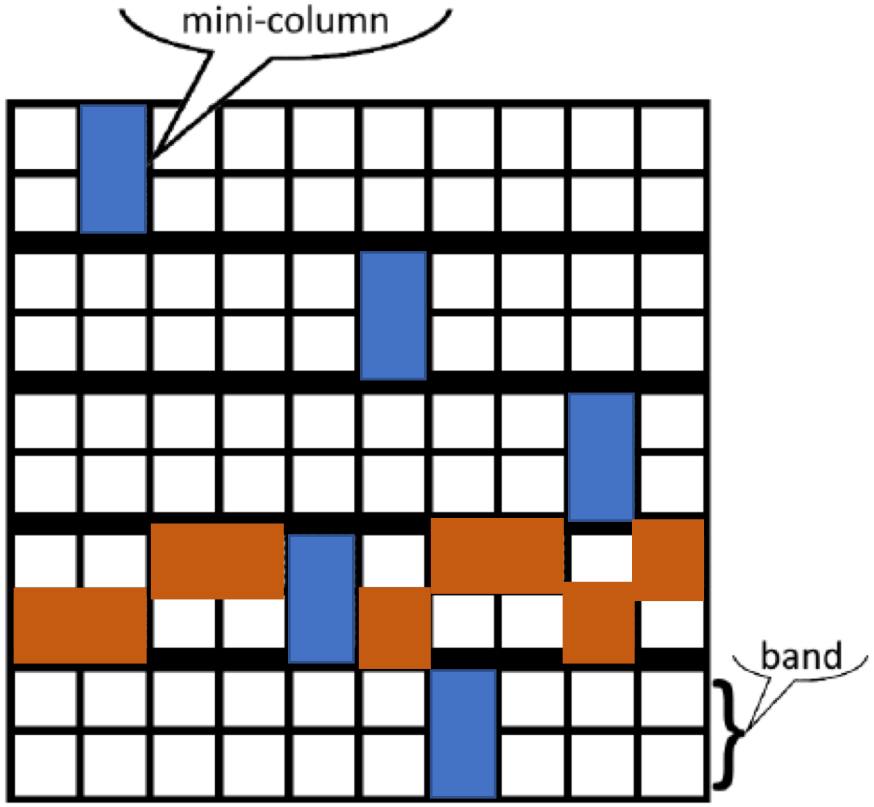


Fault tolerance

- f-resilient
  - any f nodes can fail and at least one quorum still exists
  - resilience: largest such f

B-grid quorum system

- Mini columns: one mini column in every band
- One band with at least one element per mini-column
- r = rows in a band, h = number of bands, d = count columns
- size of each quorum: h*r + d -1
- Has ideal properties:
  - work: $\theta(\sqrt{n})$
  - load: $\theta(\frac{1}{\sqrt{n}})$
  - asymptotic failure probability: 0

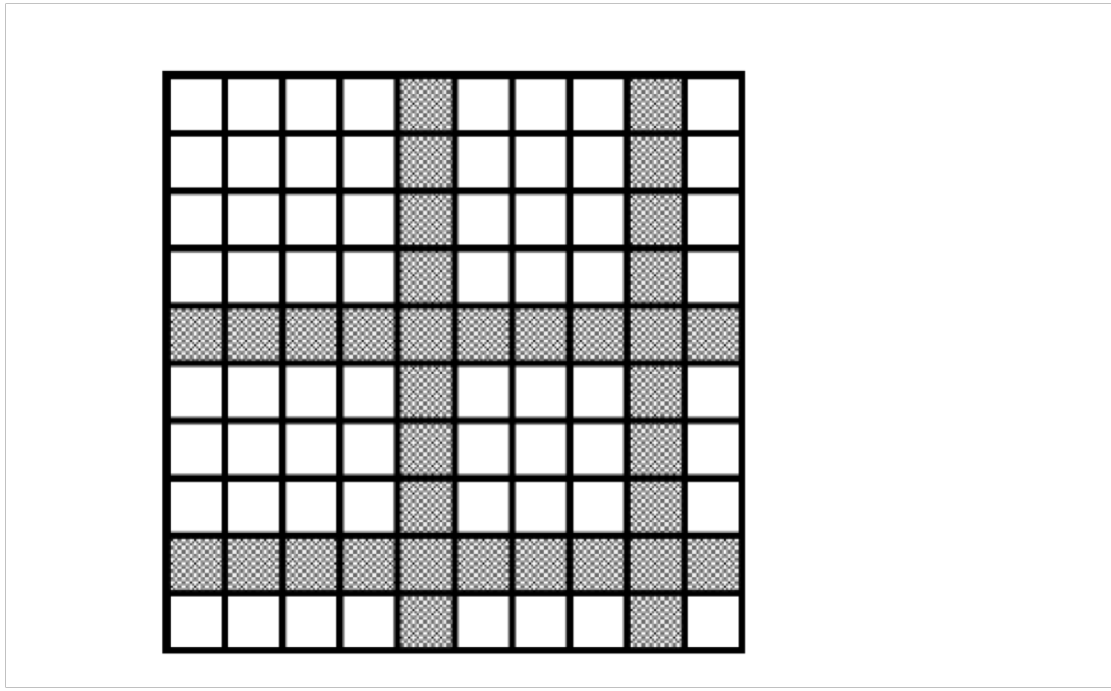


Byzantine quorum systems

- **f-disseminating**
  - if the intersection of two quorums always contains f+1 nodes
  - for any set of f byzantine nodes, there always is a quorum without byzantine nodes
  - *good model if data is self-authenticating, if not we need a stronger one*
- **f-masking**
  - if the intersection of two quorums always contains 2f+1 nodes
  - for any set of f byzantine nodes, there always is a quorum without byzantine nodes
  - *correct nodes will always be in majority*

Byzantine quorum systems – M grid

- $\sqrt{f+1}$ rows and $\sqrt{f+1}$ columns
  - $\sqrt{f+1}$* $\sqrt{f+1} = \mathrm{f}+1$ intersections
  - -> f disseminating



Opaque quorum systems

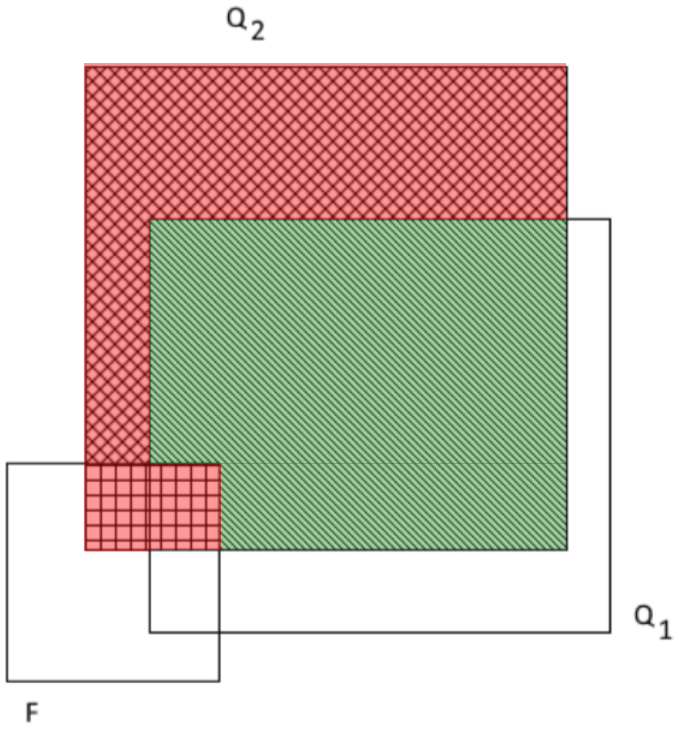
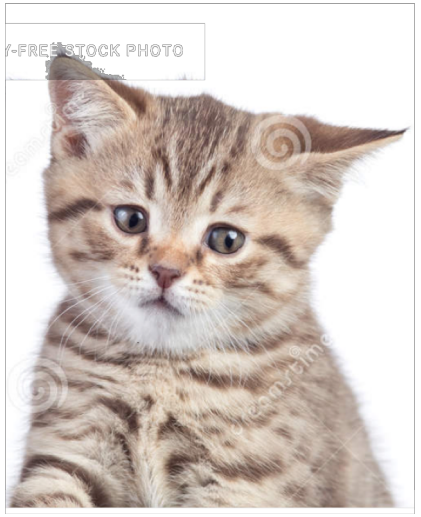


Figure 15.34: Intersection properties of an opaque quorum system. Equation (15.33.1) ensures that the set of non-byzantine nodes in the intersection of Q_1, Q_2 is larger than the set of out of date nodes, even if the byzantine nodes "team up" with those nodes. Thus, the correct up to date value can always be recognized by a majority voting.

Also we want at least one Quorum that contains no byzantine nodes

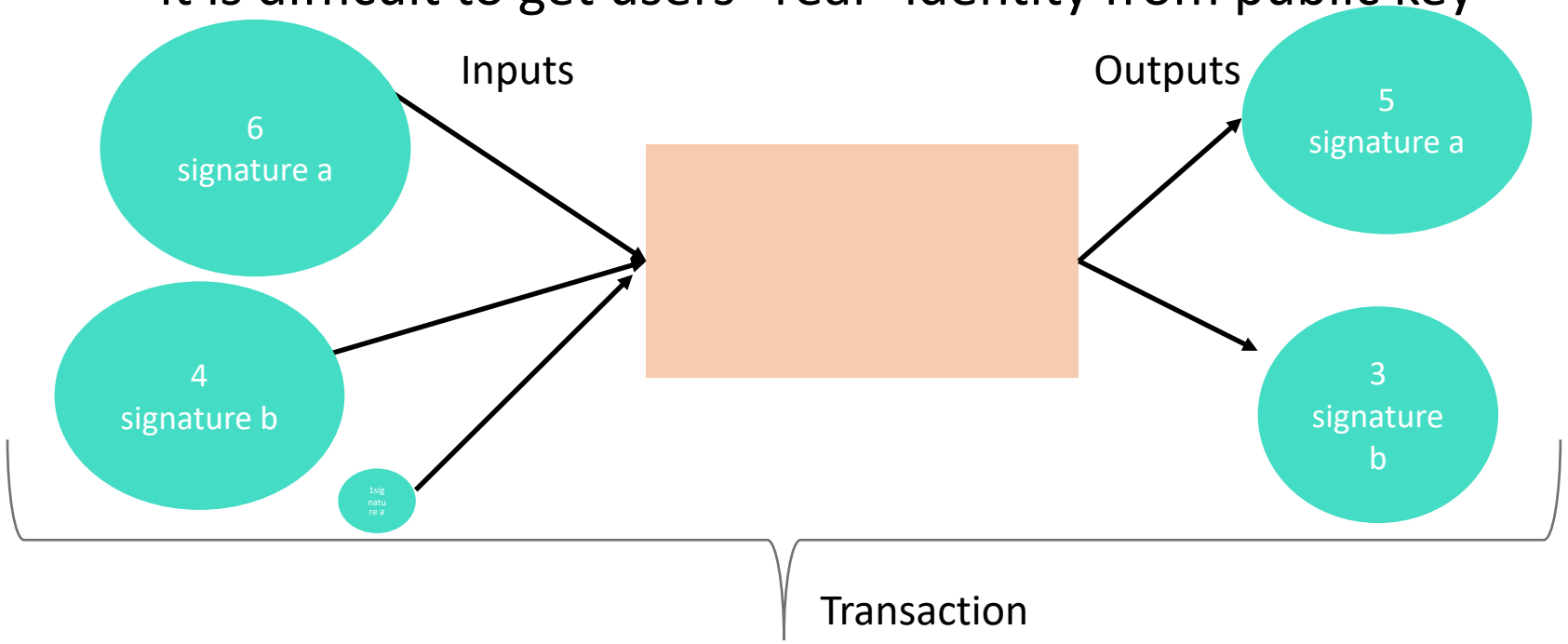
Consistency, Availability and Partition tolerance

- Consistency:
  - All nodes agree on the current state of the system
- Availability:
  - The system is operational and instantly processing incoming requests
- Partition tolerance:
  - Still works correctly if a network partition happens
- Good news:
  - achieving any two is very easy
- Bad news:
  - achieving three is impossible (CAP theorem)



Bitcoin

- decentralized network consisting of nodes
- users generate private/public key pair
  - address is generated from public key
  - it is difficult to get users “real” identity from public key



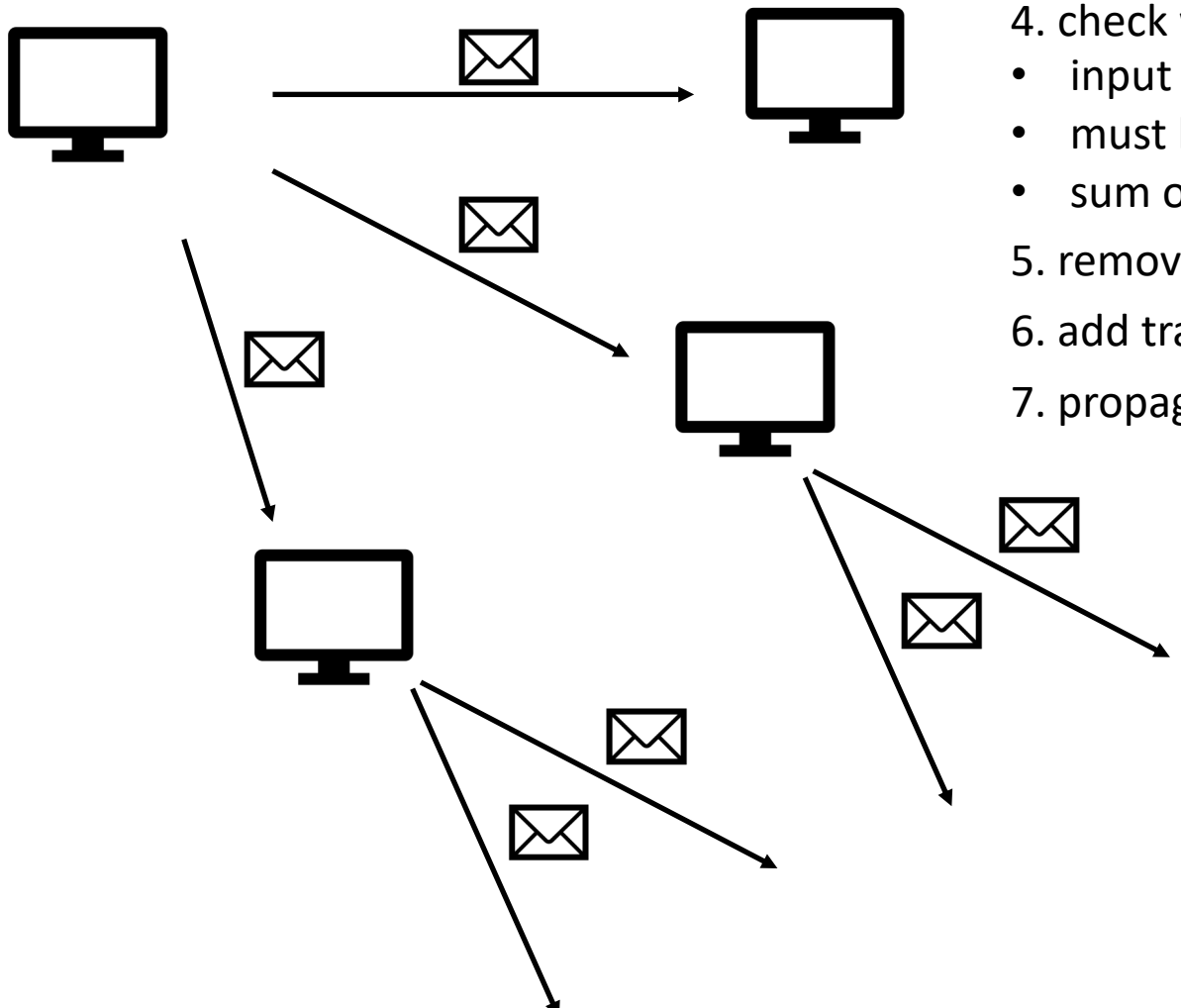
Bitcoin transactions

- Conditions:
  - Sum of inputs must always be at least the sum of outputs
    - unused part is used as transaction fee, gets paid to miner of block
  - An input must always be some whole output, no splitting allowed!
  - Money that a user “has” is defined as sum of unspent outputs

Blockchain

1. issue transaction
2. add transaction to local history

3. send transaction to other nodes in network



4. check whether transaction is valid
   - input of transaction must be in local UTXO
   - must have valid signature
   - sum of inputs >= sum of outputs
5. remove any input of transaction from local UTXO
6. add transaction to local history
7. propagate transaction further

Blockchain

- Right now we have infinitely growing memory pool and we can’t be sure that other nodes have the same pool
- Solution: Propagate memory pool through network and make sure everybody else will have same state
- Problem: How to avoid that everybody wants to propagate it’s own memory pool?
- Solution: Proof-of-Work
  - proof that you put a certain amount of work into propagating your memory pool

In Bitcoin

- A node is allowed to propagate block as soon as it has calculated a hash with some special property
  - no better method than iterating through different input values until the right hash appears
- Problem: What happens if two nodes calculate a correct hash at the same time? Some nodes will accept one block then, the other ones another block
- Solution: Nodes will always accept the block with the longest chain, so in order to keep up the split, the nodes would have to always calculate two hashes at the same time -> probability goes to 0

Bitcoin

- What does a node actually do when it receives a block:
  - node has current block $B_{max}$with height h_{max}
  - it connects received block in tree as child of its parent
  - if height of new block is bigger than h_{max} then
    - set $B_{max}$and h_{max}
    - compute new UTXO
    - clean up memory pool
- Result: all nodes that accept block will see same history of transactions

Smart Contracts

- Contract between two or more parties, encoded in such a way that correct execution is guaranteed by blockchain
  - Timelock: transaction will only get added to memory pool after some time has expired
    - Micropayment channel:
      - Idea: Two parties want to do multiple small transactions, but want to avoid fees. So they only submit first and last transaction to blockchain and privately do everything inbetween

Micropayment channel – setup transaction

Algorithm 16.26 Parties A and B create a 2-of-2 multisig output o

1: B sends a list I_B of inputs with c_B coins to A
2: A selects its own inputs I_A with c_A coins
3: A creates transaction $t_s\{[I_A, I_B], [o = c_A + c_B \rightarrow (A, B)]\}$
4: A creates timelocked transaction $t_r\{[o], [c_A \rightarrow A, c_B \rightarrow B]\}$ and signs it
5: A sends t_s and t_r to B
6: B signs both t_s and t_r and sends them to A
7: A signs t_s and broadcasts it to the Bitcoin network

A

B

cannot do anything with this, since no transaction has all required signatures

1) creates shared "account", does not sign it
2) creates timelocked transaction that unrolls shared account, signs it
3) sends them to B

4) signs both transactions

5) signs create transaction
6) broadcasts them to network

can't do anything with this, since unroll transaction is not valid without create transaction

Micropayment channel

Algorithm 16.27 Simple Micropayment Channel from S to R with capacity c

1: $c_S = c$, $c_R = 0$
2: S and R use Algorithm 16.26 to set up output o with value c from S
3: Create settlement transaction $t_f\{[o], [c_S \to S, c_R \to R]\}$
4: **while** channel open **and** $c_R < c$ **do**
5: In exchange for good with value δ
6: $c_R = c_R + \delta$
7: $c_S = c_S - \delta$
8: Update t_f with outputs $[c_R \to R, c_S \to S]$
9: S signs and sends t_f to R
10: **end while**
11: R signs last t_f and broadcasts it

set up shared account and unrolling

create settlement transaction

while buyer still has money and timelock not expired

exchange goods and adapt money

update settlement transactions with new values
S signs transaction and sends it to R

R signs last transaction and broadcasts it before timelock expires

Why does s sign it?

- like this, R always holds all fully signed transactions and can choose the last one (where he gets the most money)
- S cannot submit any transaction, so S cannot get the goods and later submit a transaction where S did not pay the money for it

Quiz

- What is the biggest number of nodes that can fail so that the majority quorum system still works?
  - n/2-1
- Are effects of blockchain transactions on a node deterministic?
  - yes
- Can a micropayment channel be used bidirectionally?
  - No, because then both parties would hold fully signed transactions and then submit the ones that are best for them

Quiz

1.1 The Resilience of a Quorum System

a) Does a quorum system exist, which can tolerate that all nodes of a specific quorum fail? Give an example or prove its nonexistence.

b) Consider the *nearly all* quorum system, which is made up of n different quorums, each containing $n - 1$ servers. What is the resilience of this quorum system?

c) Can you think of a quorum system that contains as many quorums as possible?
Note: the quorum system does not have to be minimal.

1.1 The Resilience of a Quorum System

a) No such quorum system exists. According to the definition of a quorum system, every two quorum of a quorum system intersect. So at least one server is part of both quorums. The fact that all servers of a particular quorum fail, implies that in each other quorum at least one server fails, namely the one which lies in the intersection. Therefore it is not possible to achieve a quorum anymore and the quorum system does not work anymore.

b) Just 1 - as soon as 2 servers fail, no quorum survives.

c) Imagine a quorum system in which all quorums overlap exactly in one single node. Each element of the powerset of the remaining $n-1$ nodes joined with this special node is a quorum. This gives 2^{n-1} quorums.
Can there be more? No! Consider a set from the powerset of n servers. Its complement cannot be a quorum as well, as they do not overlap. So, from each such couple, at most one set can be part of the quorum system. This gives an upper bound of $2^n/2 = 2^{n-1}$.

Quiz

2.1 Delayed Bitcoin

In the lecture we have seen that Bitcoin only has eventual consistency guarantees. The state of nodes may temporarily diverge as they accept different transactions and consistency will be re-estalished eventually by blocks confirming transactions. If, however, we consider a delayed state, i.e., the state as it was a given number Δ of blocks ago, then we can say that all nodes are consistent with high probability.

a) Can we say that the Δ-delayed state is strongly consistent for sufficiently large Δ?

b) Reward transactions make use of the increased consistency by allowing reward outputs to be spent after *maturing* for 100 blocks. What are the advantages of this maturation period?

2.1 Delayed Bitcoin

a) It is true that naturally occurring forks of length l decrease exponentially with l, however this covers naturally occuring blockchain forks only. As there is no information how much calculation power exists in total, it is always possible a large blockchain fork exists. This may be the result of a network partition or an attacker secretly running a large mining operation.

This is a general problem with all "open-membership" consensus systems, where the number of existing consensus nodes is unknown and new nodes may join at any time. As it is always possible a much larger unknown part of the network exists, it is impossible to have strong consistency.

In the Bitcoin world an attack where an attacker is secretly mining a second blockchain to later revert many blocks is called a 51% attack, because it was thought necessary to have a majority of the mining power to do so. However later research showed that by using other weaknesses in Bitcoin it is possible to do such attacks already with about a third of the mining power.

b) The delay in this case prevents coins from completely vanishing in the case of a fork. Newly mined coins only exist in the fork containing the block that created them. In case of a blockchain fork the coins would disappear and transactions spending them would become invalid as well. It would therefore be possible to taint any number of transactions that are valid in one fork and not valid in another. Waiting for maturation ensures that it is very improbable that the coins will later disappear accidentially.

Note that this is however only a protection against someone accidentially sending you money that disappears with a discontinued fork. The same thing can still happen, if someone with evil intent double spends the same coins on the other side of the fork. You will not be able to replay a transaction of a discontinued fork on the new active chain if the old owner spent them in a different transaction in the meantime. To prevent theft by such an attacker you need to wait enough time to regard the chance of forks continuing to exist to be small enough. A common value used is about one hour after a transaction entered a block ($\sim$6 blocks).