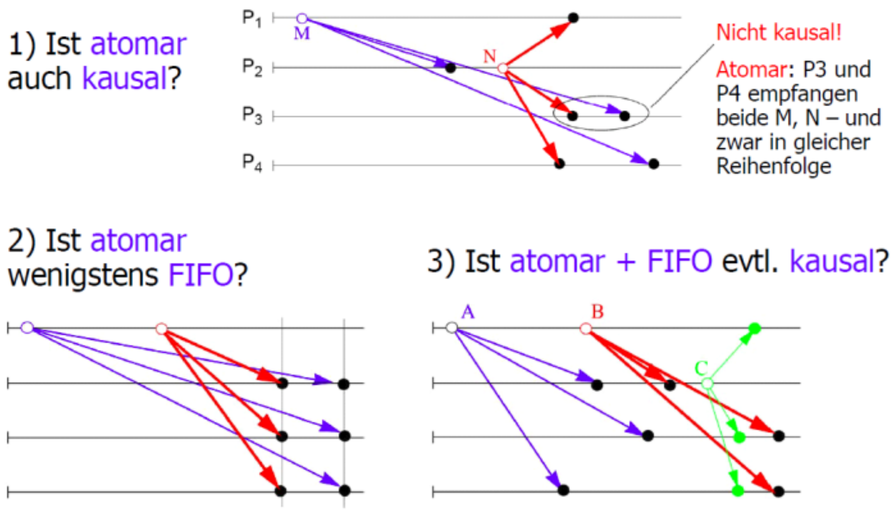
zusammenfassung

Samstag, 6. Januar 2018 14:02

atomar: wenn prozesse die zwei nachrichten empfangen, die in der selben reihenfolge empfangen
kausal: wenn ein pfad von links nach rechts sendeereignis von X zum sendeereignis von Y führt
FIFO: nur zwei prozesse betrachtend, kam die zuerst gesendete nachricht auch zuerst an?
    d.h. andere prozesse haben keinen einfluss darauf obs fifo ist oder nicht
atomar + FIFO ≠ kausal (simples beispiel: zwei unabhängige sendungen)
kausal + atomar := virtuell synchron
atomic


1) Ist atomar auch kausal?

Nicht kausal!

Atomar: P3 und P4 empfangen beide M, N – und zwar in gleicher Reihenfolge

2) Ist atomar wenigstens FIFO?

3) Ist atomar + FIFO evtl. kausal?

x "happened before" y iff
    x und y auf dem selben prozess und x vor y war, oder
    x eine nachricht M sendet und y die nachricht M empfängt, oder
    Transitivität

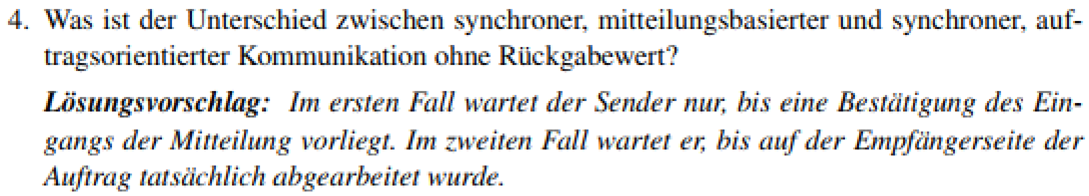
Uhrenbedingung: If happened before, then has smaller timestamp. Reverse is not always applicable.
    obviously geht es nicht rückwärts: nicht jedes vorheriges ereignis ist kausal abhängig weil nicht auf dem selben prozess.
    (Evt geht es bei Vector clocks)

- Initially all clocks are zero.
- Each time a process experiences an internal event, it increments its own logical clock in the vector by one.
- Each time for a process to send a message, it must increment its own clock (as in the bullet above) and then send a copy of its own vector.
- Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).

Aus <https://en.wikipedia.org/wiki/Vector_clock>

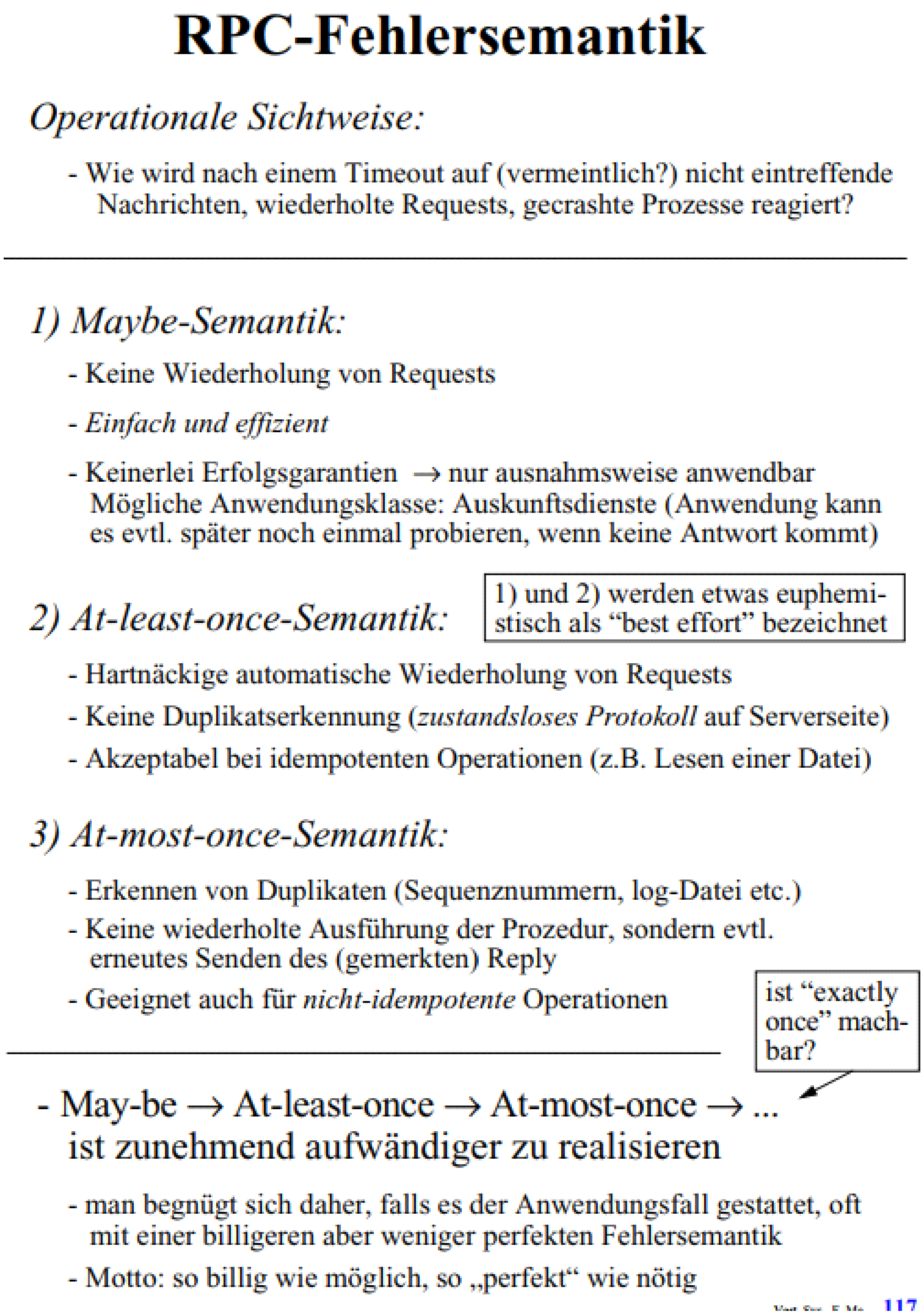
1. *What are the main advantages of using Vector Clocks over Lamport timestamps?*
Lamport guarantees that if B is causally dependent on A, then $L(A) < L(B)$.
However, the inverse does not hold: Just because something has a larger timestamp does not mean it depends on A.

Vector clocks allow us to find out whether messages A and B are causally related, because they provide a partial order instead of a total order. A Vector clock basically contains a Lamport timestamp for every thread. *If and only if* all those entries in A are smaller or equal and one of them is strictly smaller than the corresponding entries in B, the message A happened-before the message B.
This means we can also find out that two messages are independent.



4. Was ist der Unterschied zwischen synchroner, mitteilungsbasierter und synchroner, auftragsorientierter Kommunikation ohne Rückgabewert?

   Lösungsvorschlag: *Im ersten Fall wartet der Sender nur, bis eine Bestätigung des Eingangs der Mitteilung vorliegt. Im zweiten Fall wartet er, bis auf der Empfängerseite der Auftrag tatsächlich abgearbeitet wurde.*



RPC-Fehlersemantik

Operationale Sichtweise:

- Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Nachrichten, wiederholte Requests, gecrashte Prozesse reagiert?

---

1) Maybe-Semantik:

- Keine Wiederholung von Requests
- *Einfach und effizient*
- Keinerlei Erfolgsgarantien → nur ausnahmsweise anwendbar
  Mögliche Anwendungsklasse: Auskunftsdienste (Anwendung kann es evtl. später noch einmal probieren, wenn keine Antwort kommt)

2) At-least-once-Semantik:

1) und 2) werden etwas euphemistisch als "best effort" bezeichnet

- Hartnäckige automatische Wiederholung von Requests
- Keine Duplikatserkennung (*zustandsloses Protokoll* auf Serverseite)
- Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

3) At-most-once-Semantik:

- Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
- Keine wiederholte Ausführung der Prozedur, sondern evtl. erneutes Senden des (gemerkten) Reply
- Geeignet auch für *nicht-idempotente* Operationen

---

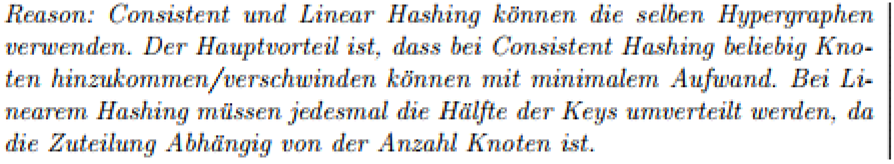
ist "exactly once" machbar?

- May-be → At-least-once → At-most-once → ...
  ist zunehmend aufwändiger zu realisieren
  - man begnügt sich daher, falls es der Anwendungsfall gestattet, oft mit einer billigeren aber weniger perfekten Fehlersemantik
  - Motto: so billig wie möglich, so „perfekt" wie nötig

Causal consistency vs Sequential Consistency vs Quiescent Consistency
Sequential is when there is one sequential order to which all processors view fits (doesn't mean that it must be run in that way). Causal is a partial order (like lamport clocks). Reads influence following writes. Writes might happen in any order though, if there is no causality.
Sequential implies causal.
Quiescent Consistency describes a completely clear program order - no overlaps are allowed.

Linearizability vs Serializability
Linearizability says that writes/reads appear to be instantaneous. After a read, any further reads will return the same or a newer value.
Serializability is a guarantee about transactions; It says that the code is equivalent to some serial execution.

Consistent Hashing
Stores the values with keys closest to the nodes ID.



Reason: Consistent und Linear Hashing können die selben Hypergraphen verwenden. Der Hauptvorteil ist, dass bei Consistent Hashing beliebig Knoten hinzukommen/verschwinden können mit minimalem Aufwand. Bei Linearem Hashing müssen jedesmal die Hälfte der Keys umverteilt werden, da die Zuteilung Abhängig von der Anzahl Knoten ist.

Durchmesser eines Graphen
"Durchmesser eines gittergraphen", "homogenität" als kriterium für bestimmte anwendungen, …
Längster kürzester pfad von vertex zu vertex. beinhaltet die vertices.

Game Theory
Price of Anarchy: Global maximum of welfare of all divided by the minimum in an equilibrium

Consensus

Validity
Der Entscheidungswert ist der Inputwert von einem Knoten
Termination
Alle korrekten Knoten terminieren in endlicher Zeit
Agreement
Alle korrekten Knoten entscheiden sich für den gleichen Wert

Quorum Systems
A Quorum System is a set of quorums such that every two quorums intersect.
The load induced by access strategy Z on a quorum system S is the maximal load induced by Z on any node in S.
(The load on a node is the sum of the load onto each quorum the node is in)

$$L_Z(v_i) = \sum_{Q \in S; v_i \in Q} P_Z(Q)$$

The load of a quorum system is the load induced by the 'best' access strategy, i.e. the minimum of these maxima.

The work of a quorum is the number of nodes in it.
The work induced by access strategy Z on a quorum system S is the expected number of nodes accessed (i.e. Probability of quorum times work of quorum, summed up for all quorums in the system)

$$W_Z(S) = \sum_{Q \in S} P_Z(Q) \cdot W(Q)$$

The work of a quorum system is the minimal work possible by changing the access strategy Z.

But the Access strategy Z must be the same for *both load and work*.
For *any* Quorum system, $L(S) \geq \frac{1}{\sqrt{n}}$

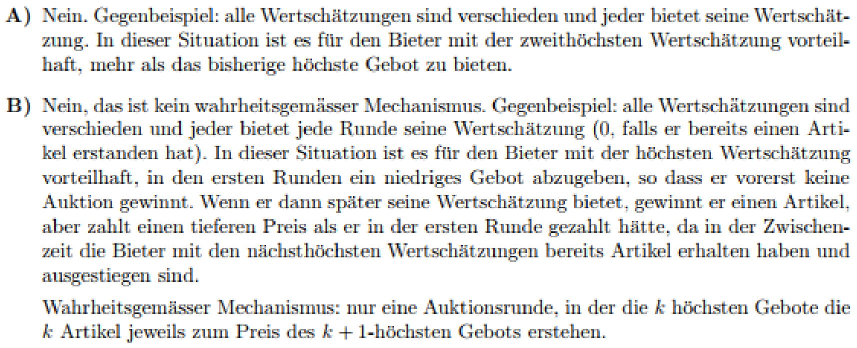
S is **f-resilient** if it can suffer f nodes dying and there is still at least one working quorum.

the failure-probability is the chance that at least one node of every quorum fails

A quorum System is called **minimal** if there is no quorum subset of an other quorum.

A quorum System is **f-disseminating** if every intersection of two quorums consists of at least f+1 nodes and for any set of f byzantine nodes, there is at least one quorum without byzantine nodes (minimum n=3f). If the data is self-verifying, then this is enough (e.g. through authentication with signed messages).
It is **f-masking** if the intersection always contains 2f+1 nodes and there is at least one quorum without byzantine nodes (minimum n=4f). That means that the correct nodes outvote the byzantines in the intersection and at least one quorum operates correctly.
It is **f-opaque** if for any two quroums, the number of correct nodes in the intersection is larger than the number of byzantine nodes in Q2 plus the number of nodes in Q1 but not Q2

Wahrheitsgemässe Auktion
Wenn es nie besser ist, über sein gebot zu lügen.
2nd-price auction ist truthful, repeated 2nd price für ununterscheidbare Artikel nicht (weil man warten kann und es billiger wird), 3rd price nicht. repeated 2nd-price für unterscheidbare Artikel schon.



A) Nein. Gegenbeispiel: alle Wertschätzungen sind verschieden und jeder bietet seine Wertschätzung. In dieser Situation ist es für den Bieter mit der zweithöchsten Wertschätzung vorteilhaft, mehr als das bisherige höchste Gebot zu bieten.

B) Nein, das ist kein wahrheitsgemässer Mechanismus. Gegenbeispiel: alle Wertschätzungen sind verschieden und jeder bietet jede Runde seine Wertschätzung (0, falls er bereits einen Artikel erstanden hat). In dieser Situation ist es für den Bieter mit der höchsten Wertschätzung vorteilhaft, in den ersten Runden ein niedriges Gebot abzugeben, so dass er vorerst keine Auktion gewinnt. Wenn er dann später seine Wertschätzung bietet, gewinnt er einen Artikel, aber zahlt einen tieferen Preis als er in der ersten Runde gezahlt hätte, da in der Zwischenzeit die Bieter mit den nächsthöchsten Wertschätzungen bereits Artikel erhalten haben und ausgestiegen sind.

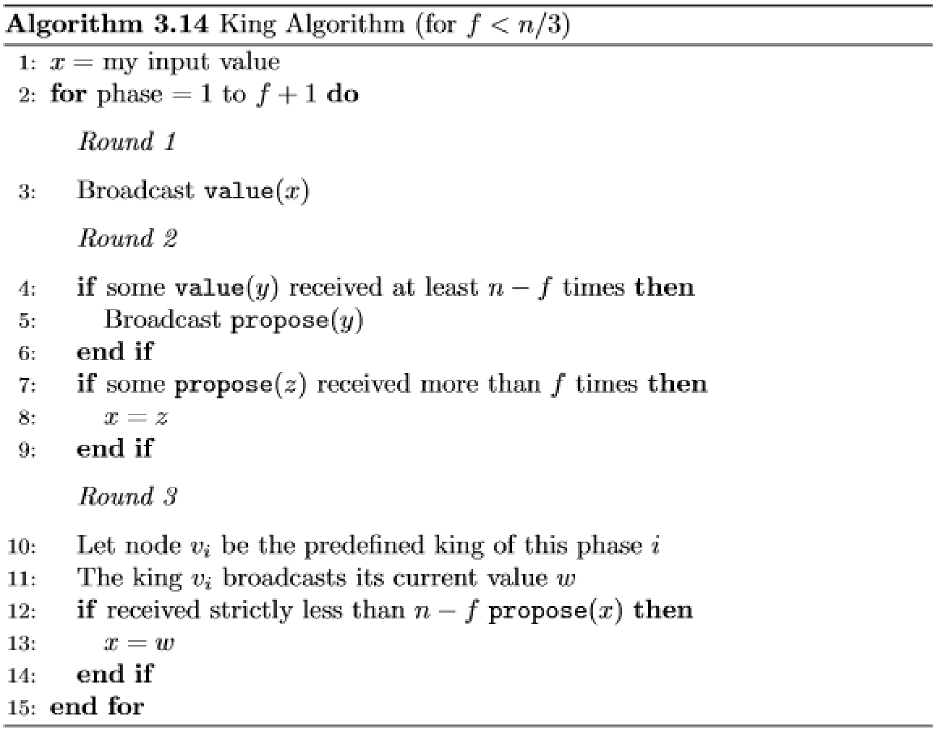
Wahrheitsgemässer Mechanismus: nur eine Auktionsrunde, in der die k höchsten Gebote die k Artikel jeweils zum Preis des $k+1$-höchsten Gebots erstehen.

Hashing
Locks

Configuration Tree
This model does not work for algorithms that use the message delay. All messages are transmitted in at most one timestep and any local computations are done instantly.

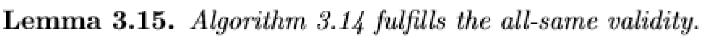
"v-valent" if a configuration already determines the value. e.g. if all input values are 0, the configuration is 0-valent.
C is **critical** if it is bivalent but all configurations that are direct children of it are univalent.

King Algorithm
n = 3f+1



Algorithm 3.14 King Algorithm (for $f < n/3$)

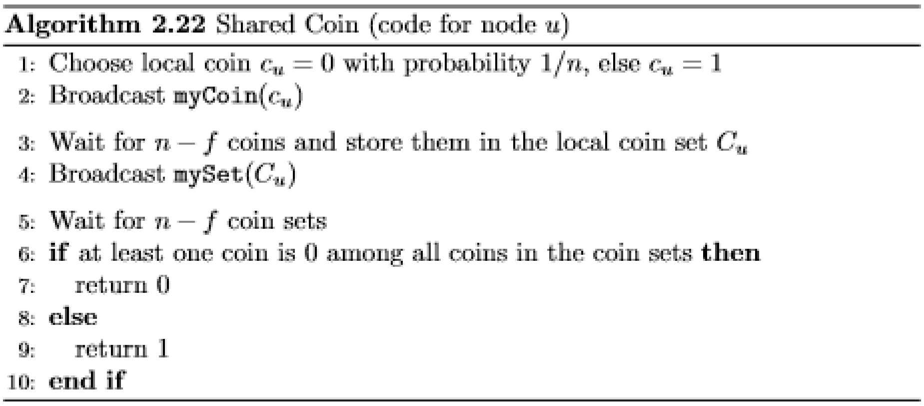
```
x = my input value
for phase = 1 to $f + 1$ do
    Round 1
    Broadcast value($x$)
    Round 2
    if some value($y$) received at least $n - f$ times then
        Broadcast propose($y$)
    end if
    if some propose($z$) received more than $f$ times then
        $x = z$
    end if
    Round 3
    Let node $v_i$ be the predefined king of this phase $i$
    The king $v_i$ broadcasts its current value $w$
    if received strictly less than $n - f$ propose($x$) then
        $x = w$
    end if
end for
```



Lemma 3.15. *Algorithm 3.14 fulfills the all-same validity.*

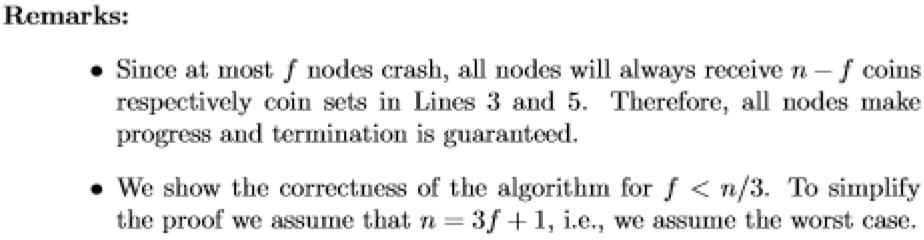
Jede node ist mal king. Wenn ein king etwas bestimmt, weil nicht $n - f$ nodes für dasselbe stimmten, dann wird das von nodes die auch wenig erhalten haben akzeptiert. D.h. es kann passieren dass der erste king byzantine ist und jede node eine andere input value hat - dann würde sich die byzantine meinung durchsetzen.

Shared Coin
(only about faulty nodes, not byzantine)



Algorithm 2.22 Shared Coin (code for node u)

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



Remarks:

- Since at most f nodes crash, all nodes will always receive $n - f$ coins respectively coin sets in Lines 3 and 5. Therefore, all nodes make progress and termination is guaranteed.
- We show the correctness of the algorithm for $f < n/3$. To simplify the proof we assume that $n = 3f + 1$, i.e., we assume the worst case.

=> Termination, Correct-input-validity, ~~Agreement~~

Inputs of possibly faulty nodes are not really considered to be the output and that's fine as long as some correct input is the output.

Lamport

12 Lamport-Zeit – Wechselseitiger Ausschluss

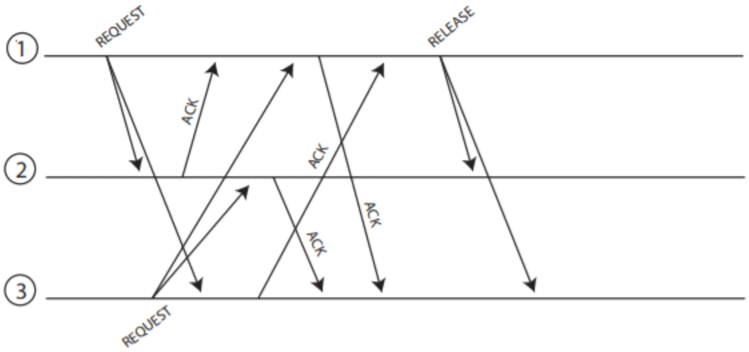
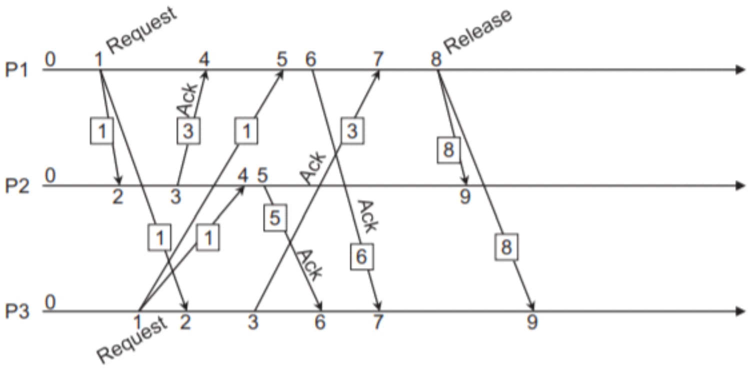


Abbildung 3: Wechselseitiger Ausschluss mit Lamport-Zeit

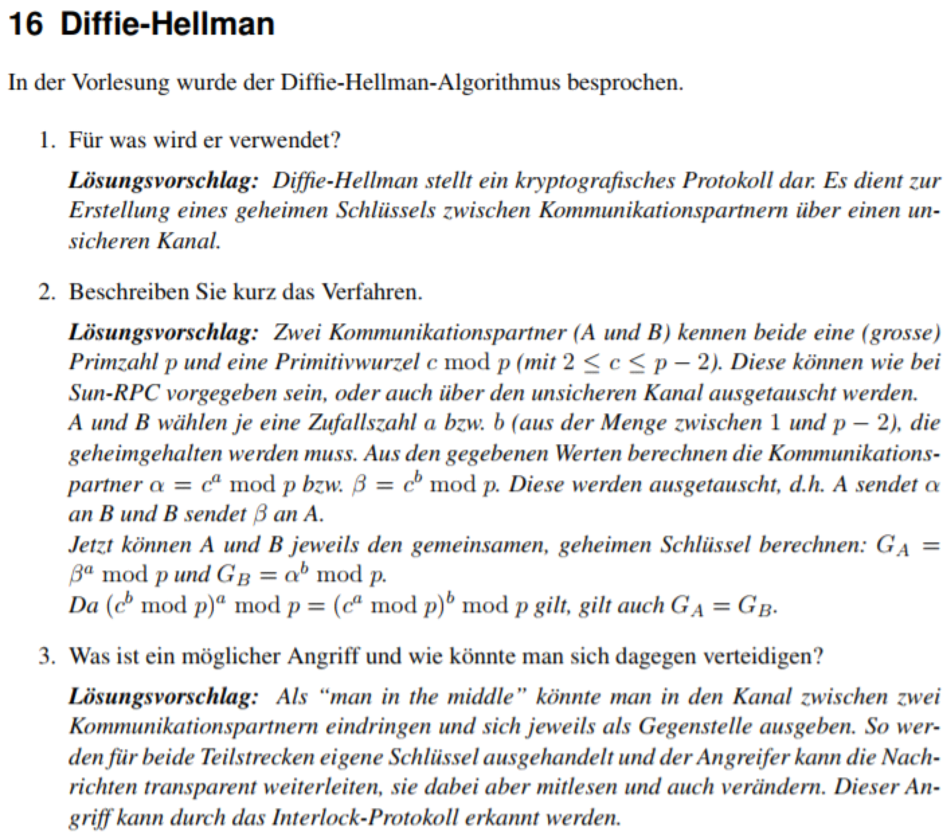
In Abb. 3 ist ein Zeitdiagramm dargestellt mit Nachrichten von drei Prozessen. Prozesse 1 und 3 bewerben sich um den exklusiven Zugriff auf eine gemeinsame Ressource. Die Prozesse wenden das aus der Vorlesung bekannte Verfahren zum wechselseitigen Ausschluss an, das Lamport-Zeit und verteilte Warteschlangen benutzt.

1. Geben Sie die Sende- und Empfangszeitstempel für jedes Ereignis an.

Lösungsvorschlag:



Increase counter when sending a message. When receiving, set counter to $\max(other + 1, mine + 1)$
where *other* is the timestamp of the other process, contained in its message.
Lamport requires **FIFO**
Zum en-queue-en die timestamps des senders verwenden, sonst ist es global nicht konsistent. (Noch ohne addition vom +1)
To actually enter the critical section, we need any message with a later timestamp than ours from every thread. That means that no thread can later come and say they have a lower number.
Additionally, the timestamp must obviously be the lowest in the local queue.



16 Diffie-Hellman

In der Vorlesung wurde der Diffie-Hellman-Algorithmus besprochen.

1. Für was wird er verwendet?

   Lösungsvorschlag: *Diffie-Hellman stellt ein kryptografisches Protokoll dar. Es dient zur Erstellung eines geheimen Schlüssels zwischen Kommunikationspartnern über einen unsicheren Kanal.*

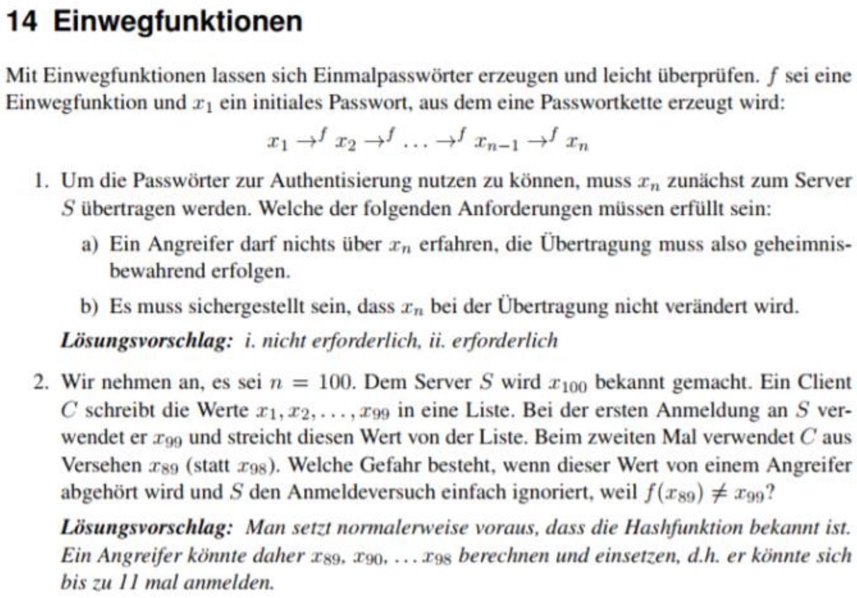
2. Beschreiben Sie kurz das Verfahren.

   Lösungsvorschlag: *Zwei Kommunikationspartner (A und B) kennen beide eine (grosse) Primzahl* p *und eine Primitivwurzel* $c \bmod p$ *(mit* $2 \leq c \leq p - 2$*). Diese können wie bei Sun-RPC vorgegeben sein, oder auch über den unsicheren Kanal ausgetauscht werden.*
   A und B wählen je eine Zufallszahl a *bzw.* b *(aus der Menge zwischen* 1 *und* $p - 2$*), die geheimgehalten werden muss. Aus den gegebenen Werten berechnen die Kommunikationspartner* $\alpha = c^a \bmod p$ *bzw.* $\beta = c^b \bmod p$*. Diese werden ausgetauscht, d.h. A sendet* α *an B und B sendet* β *an A.*
   Jetzt können A und B jeweils den gemeinsamen, geheimen Schlüssel berechnen: $G_A = \beta^a \bmod p$ *und* $G_B = \alpha^b \bmod p$*.*
   Da $(c^b \bmod p)^a \bmod p = (c^a \bmod p)^b \bmod p$ *gilt, gilt auch* $G_A = G_B$*.*

3. Was ist ein möglicher Angriff und wie könnte man sich dagegen verteidigen?

   Lösungsvorschlag: *Als "man in the middle" könnte man in den Kanal zwischen zwei Kommunikationspartnern eindringen und sich jeweils als Gegenstelle ausgeben. So werden für beide Teilstrecken eigene Schlüssel ausgehandelt und der Angreifer kann die Nachrichten transparent weiterleiten, sie dabei aber mitlesen und auch verändern. Dieser Angriff kann durch das Interlock-Protokoll erkannt werden.*

Hash Chains



14 Einwegfunktionen

Mit Einwegfunktionen lassen sich Einmalpasswörter erzeugen und leicht überprüfen. f sei eine Einwegfunktion und x_1 ein initiales Passwort, aus dem eine Passwortkette erzeugt wird:

$$x_1 \rightarrow^f x_2 \rightarrow^f \ldots \rightarrow^f x_{n-1} \rightarrow^f x_n$$

1. Um die Passwörter zur Authentisierung nutzen zu können, muss x_n zunächst zum Server S übertragen werden. Welche der folgenden Anforderungen müssen erfüllt sein:
   a) Ein Angreifer darf nichts über x_n erfahren, die Übertragung muss also geheimnisbewahrend erfolgen.
   b) Es muss sichergestellt sein, dass x_n bei der Übertragung nicht verändert wird.

   Lösungsvorschlag: *i. nicht erforderlich, ii. erforderlich*

2. Wir nehmen an, es sei $n = 100$. Dem Server S wird x_{100} bekannt gemacht. Ein Client C schreibt die Werte $x_1, x_2, \ldots, x_{99}$ in eine Liste. Bei der ersten Anmeldung an S verwendet er x_{99} und streicht diesen Wert von der Liste. Beim zweiten Mal verwendet C aus Versehen x_{89} (statt x_{98}). Welche Gefahr besteht, wenn dieser Wert von einem Angreifer abgehört wird und S den Anmeldeversuch einfach ignoriert, weil $f(x_{89}) \neq x_{99}$?

   Lösungsvorschlag: *Man setzt normalerweise voraus, dass die Hashfunktion bekannt ist. Ein Angreifer könnte daher* $x_{89}, x_{90}, \ldots x_{98}$ *berechnen und einsetzen, d.h. er könnte sich bis zu 11 mal anmelden.*

2PC vs 3PC

Donnerstag, 8. Juni 2017 07:17

2PC

Coordinator sends VOTE-REQ to all
Participants recieve that and vote YES or NO
Coordinator waits for all participants until first NO
  all YES => commit and sends COMMIT
  some NO => abort and sends ABORT to all which voted YES
    Those who voted NO have already aborted themselves
Participant recieves COMMIT or ABORT and does that, then stops

This Protocol meets the 5 AC rules:
AC1: Every processor decides the same
AC2: Any processor arrving at a decision stops => Cannot reverse its decision
AC3: Controller only decides COMMIT if nobody voted NO => No imposed COMMIT
AC4: If there are no failures and all processors voted YES, the decision will be COMMIT (nontriviality)
AC5: If all failures are repaired and no more failures occur for sufficiently long, then all processors will eventually reach a decision (liveness)

For AC5 we need to extend the protocol and ask around in case of timeout.

Uncertainity Period: When a participant times out waiting for a decision and everybody is in the same situation when asking around, all processors will block. This can happen if the coordinator fails after receiving all YES votes but before sending any COMMIT message
Why can't every participant then just ask everybody else? If one says no, abort, else say yes. Because the failed coordinator might want to abort.
There's also the possibility that the coordinator and a participant fail. In that case, it is impossible to say whether this participant has recieved the COMMIT and committed or whether we should abort because no COMMIT message was sent, so we have to wait.

Persistence through logging to node disk.
YES logs *before* sending, NO logs before or after. Because if it crashes in between and finds neither a YES nor a NO log record, it aborts unilaterally.
Same for the coordinator with COMMIT or ABORT.
Reason is probably that data to evaluate is then no longer in memory and cannot be reevaluated if not yet decided.
https://courses.cs.washington.edu/courses/csep552/13sp/lectures/4/2pc.pdf

Linear 2PC
Less messages by moving on in a daisychain. Total number of messages is not 3n but only 2n because a NO propagates in both directions and a COMMIT through the whole line. The coordinator seems to be the end of the chain.

3PC

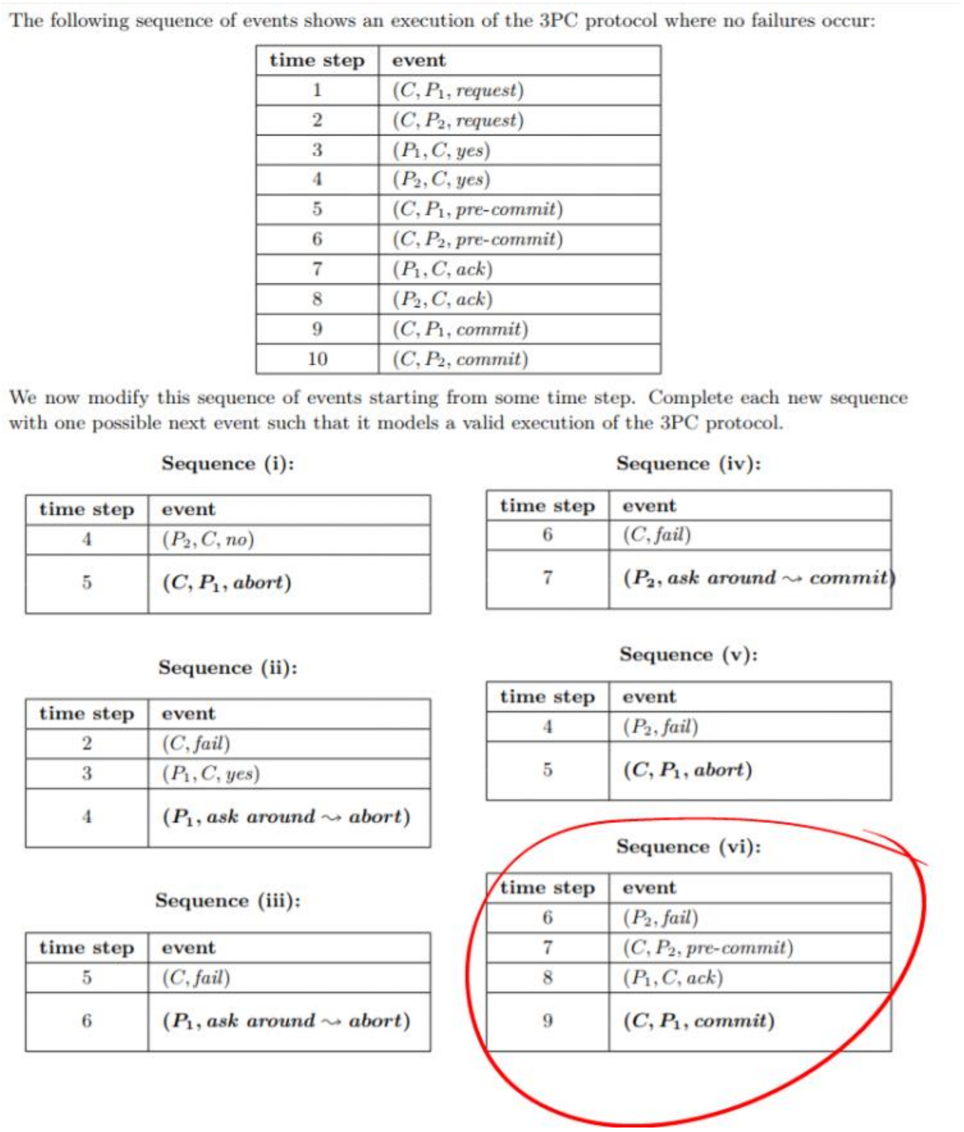
Doesn't block => liveness
AC1: every node decides the same
AC2: no node changes its decision
AC3: no imposed COMMIT
AC4: nontriviality: if there are no failures and everybody voted YES, then the decision will be COMMIT
AC5: If all failures are repaired and there aren't any more for sufficiently long, then the protocol will terminate with a decision (liveness)

Assuming no communication failures.
NB rule: Nobody can decide to commit as long as anybody is uncertain.

Difference to 2PC: PRE-COMMIT -> ACK -> COMMIT
So if the coordinator fails after VOTE-REQ and all processors vote YES, then they would all be in uncertainity in 2PC. In 3PC it is guaranteed that nobody has decided to commit while anybody is uncertain. So if everybody is uncertain, they can find that out and safely abort. To make sure, nobody is uncertain before deciding, the coordinator needs the PRE-COMMIT. Now if the coordinator crashes after sending PRE-COMMIT, participants know what is going to happen but have to ask around to make sure everybody is certain before committing.
If coordinator times out waiting for votes, ABORT. If coordinator times out waiting for ACKs, ignore those and send the others a commit. (**some also say to wait. The appended solutions say otherwise**) They can later ask around to find out that they should commit.
If a node fails after recieving PRE-COMMIT, it has to ask around to make sure nobody is uncertain.

Again, logging YES before sending because if crashes and no YES there, then abort. It seems like sending precommits is not logged, so if the coordinator crashes after starting 3PC but has no decision in its log, it has to ask around (maybe somebody already got a precommit)

Not used in practise because probability of blocking is small enough and 3PC is too expensive.



The following sequence of events shows an execution of the 3PC protocol where no failures occur:

time step	event
1	$(C, P_1, request)$
2	$(C, P_2, request)$
3	(P_1, C, yes)
4	(P_2, C, yes)
5	$(C, P_1, pre\text{-}commit)$
6	$(C, P_2, pre\text{-}commit)$
7	(P_1, C, ack)
8	(P_2, C, ack)
9	$(C, P_1, commit)$
10	$(C, P_2, commit)$

We now modify this sequence of events starting from some time step. Complete each new sequence with one possible next event such that it models a valid execution of the 3PC protocol.

Sequence (i):

time step	event
4	(P_2, C, no)
5	**$(C, P_1, abort)$**

Sequence (ii):

time step	event
2	$(C, fail)$
3	(P_1, C, yes)
4	**$(P_1, ask\ around \rightsquigarrow abort)$**

Sequence (iii):

time step	event
5	$(C, fail)$
6	**$(P_1, ask\ around \rightsquigarrow abort)$**

Sequence (iv):

time step	event
6	$(C, fail)$
7	**$(P_2, ask\ around \rightsquigarrow commit)$**

Sequence (v):

time step	event
4	$(P_2, fail)$
5	**$(C, P_1, abort)$**

Sequence (vi):

time step	event
6	$(P_2, fail)$
7	$(C, P_2, pre\text{-}commit)$
8	(P_1, C, ack)
9	**$(C, P_1, commit)$**

PBFT

Dienstag, 9. Januar 2018 10:30

Practical Byzantine Fault Tolerance

$n = 3f + 1$

- Messages are signed
- One node is considered primary (might change over time)
- Messages might not have their order preserved

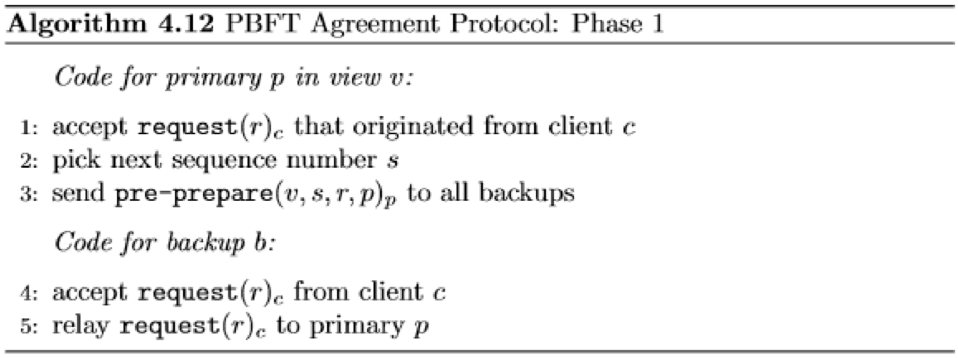
If Backup Nodes detect faulty primary node, they start a new *view* v where the next Node is now the Primary. ($primary = v \bmod n$)
The primary picks consecutive sequence numbers. Backup nodes verify through intercommunication that they all have received the same order.

No correct node will execute a request with the sequence number belonging to a different request.
Nodes will collect confirmation messages for a decision that a request should be executed by asking $2f + 1$ nodes, including itself.

If we have two sets of $2f + 1$ nodes, then there exists a correct node in their intersection.

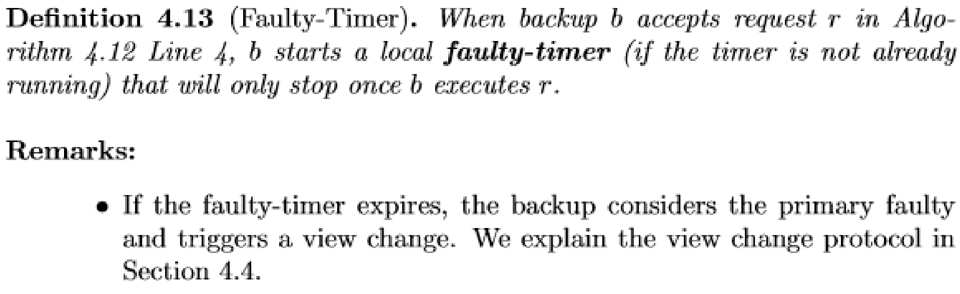
2f+1 because that's the majority of the correct nodes (n = 3f+1)

Agreeing on a unique order of requests within a view
1. primary sends *pre-prepare* message to all backups with a specified sequence number.
2. Backups send *prepare* messages to all nodes to state that they agree.
3. All nodes send *commit* messages to all nodes, execute the request and inform the client.



Algorithm 4.12 PBFT Agreement Protocol: Phase 1

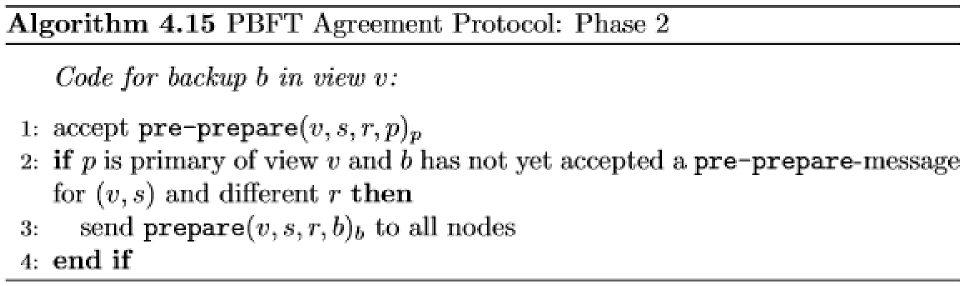
```
Code for primary p in view v:
1: accept request(r)_c that originated from client c
2: pick next sequence number s
3: send pre-prepare(v, s, r, p)_p to all backups
Code for backup b:
4: accept request(r)_c from client c
5: relay request(r)_c to primary p
```



Definition 4.13 (Faulty-Timer). *When backup b accepts request r in Algorithm 4.12 Line 4, b starts a local* ***faulty-timer*** *(if the timer is not already running) that will only stop once b executes r.*

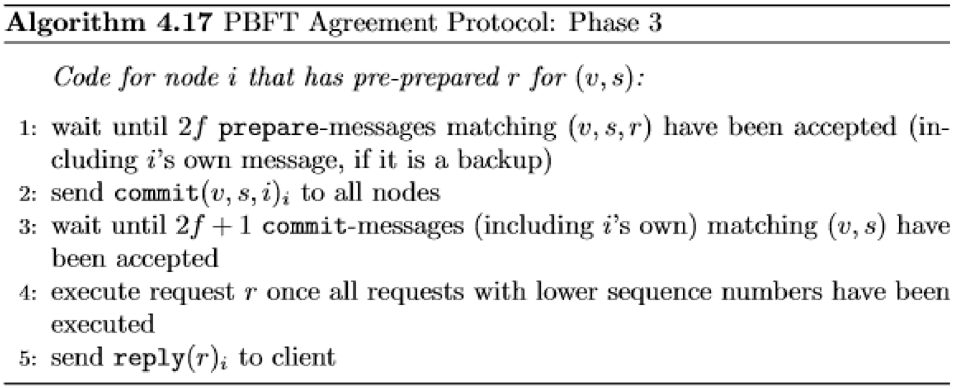
Remarks:

- If the faulty-timer expires, the backup considers the primary faulty and triggers a view change. We explain the view change protocol in Section 4.4.



Algorithm 4.15 PBFT Agreement Protocol: Phase 2

```
Code for backup b in view v:
1: accept pre-prepare(v, s, r, p)_p
2: if p is primary of view v and b has not yet accepted a pre-prepare-message
   for (v, s) and different r then
3:     send prepare(v, s, r, b)_b to all nodes
4: end if
```

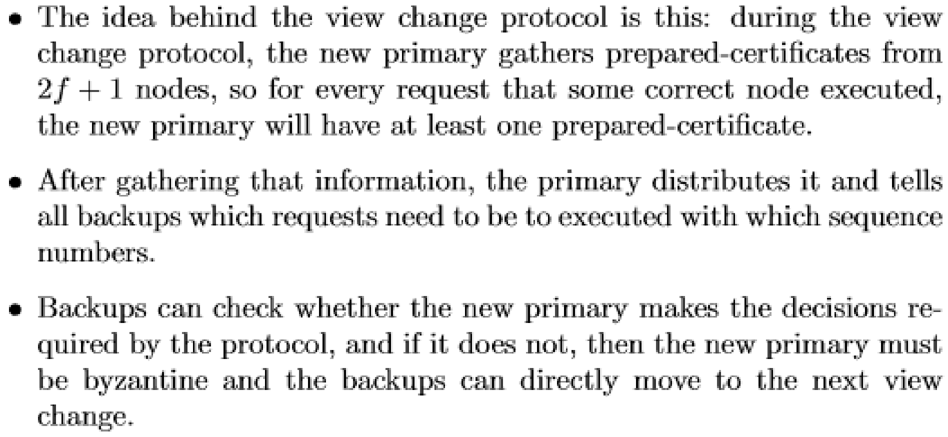


Algorithm 4.17 PBFT Agreement Protocol: Phase 3

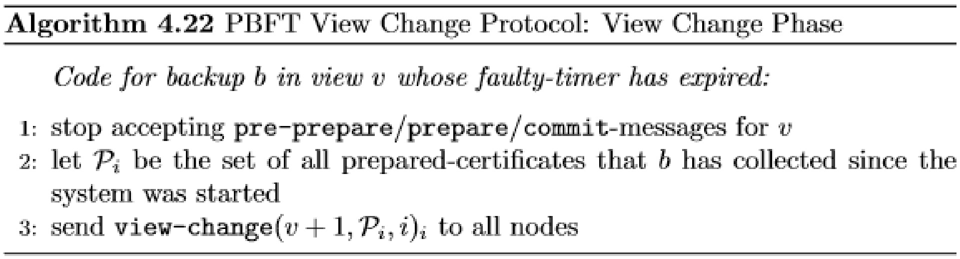
Code for node i that has pre-prepared r for (v, s):

```
wait until $2f$ prepare-messages matching $(v, s, r)$ have been accepted (including $i$'s own message, if it is a backup)
send $\texttt{commit}(v, s, i)_i$ to all nodes
wait until $2f + 1$ commit-messages (including $i$'s own) matching $(v, s)$ have been accepted
execute request $r$ once all requests with lower sequence numbers have been executed
send $\texttt{reply}(r)_i$ to client
```

- The client only needs one correct reply, so it waits for $f + 1$ reply messages.
- Once a single correct node executed the request, all correct nodes will eventually, with the same sequence number.
- If a client resends a request, nodes can look at the timestamp to figure out if they have already executed it.
- A correct backup does not send $prepare$ for the same $(view, sequencenumber)$ more than once. (Neither does a primary with a $pre\text{-}prepare$)



- The idea behind the view change protocol is this: during the view change protocol, the new primary gathers prepared-certificates from $2f + 1$ nodes, so for every request that some correct node executed, the new primary will have at least one prepared-certificate.
- After gathering that information, the primary distributes it and tells all backups which requests need to be to executed with which sequence numbers.
- Backups can check whether the new primary makes the decisions required by the protocol, and if it does not, then the new primary must be byzantine and the backups can directly move to the next view change.



Algorithm 4.22 PBFT View Change Protocol: View Change Phase

Code for backup b in view v whose faulty-timer has expired:

```
stop accepting pre-prepare/prepare/commit-messages for $v$
let $\mathcal{P}_i$ be the set of all prepared-certificates that $b$ has collected since the system was started
send $\texttt{view-change}(v + 1, \mathcal{P}_i, i)_i$ to all nodes
```

Algorithm 4.23 PBFT View Change Protocol: New View Phase - Primary

```
Code for primary $p$ of view $v+1$:
1: accept $2f+1$ view-change-messages (including possibly $p$'s own) in a set $\mathcal{V}$ (this is the new-view-certificate)
2: let $\mathcal{O}$ be a set of pre-prepare$(v+1, s, r, p)_p$ for all pairs $(s, r)$ where at least one prepared-certificate for $(s, r)$ exists in $\mathcal{V}$
3: let $s^{\mathcal{V}}_{max}$ be the highest sequence number for which $\mathcal{O}$ contains a pre-prepare-message
4: add to $\mathcal{O}$ a message pre-prepare$(v+1, s', \texttt{null}, p)_p$ for every sequence number $s' < s^{\mathcal{V}}_{max}$ for which $\mathcal{O}$ does not yet contain a pre-prepare-message

5: send new-view$(v+1, \mathcal{V}, \mathcal{O}, p)_p$ to all nodes
6: start processing requests for view $v+1$ according to Algorithm 4.12 starting from sequence number $s^{\mathcal{V}}_{max}+1$
```

Algorithm 4.24 PBFT View Change Protocol: New View Phase - Backup

```
Code for backup $b$ of view $v+1$ if $b$'s local view is $v' < v+1$:
accept new-view$(v+1, \mathcal{V}, \mathcal{O}, p)_p$
stop accepting pre-prepare-/prepare-/commit-messages for $v$// in case $b$ has not run Algorithm 4.22 for $v+1$ yet
set local view to $v+1$
if $p$ is primary of $v+1$ then
  if $\mathcal{O}$ was correctly constructed from $\mathcal{V}$ according to Algorithm 4.23 Lines 2 and 4 then
    respond to all pre-prepare-messages in $\mathcal{O}$ as in normal case operation, starting from Algorithm 4.15
    start accepting messages for view $v+1$
  else
    trigger view change to $v+2$ using Algorithm 4.22
  end if
end if
```

- A faulty new primary could delay the system indefinitely by never sending a `new-view`-message. To prevent this, as soon as a node sends its `view-change`-message for $v+1$, it starts its faulty-timer and stops it once it accepts a `new-view`-message for $v+1$. If the timer runs out before being stopped, the node triggers another view change.

- Since at most f consecutive primaries can be faulty, the system makes progress after at most $f+1$ view changes.

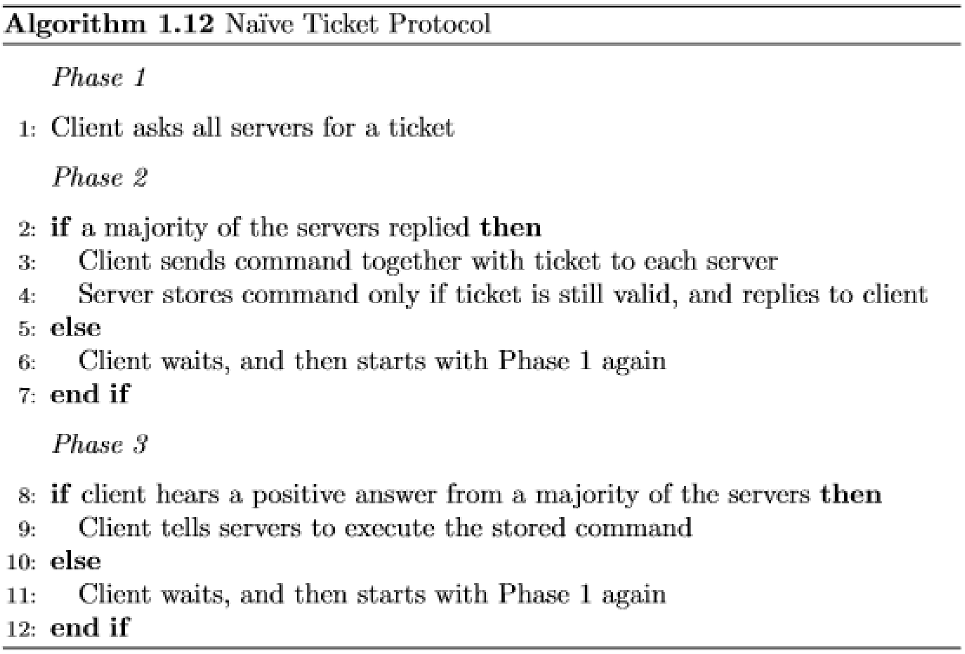
- We described a simplified version of PBFT; any practically relevant variant makes adjustments to what we presented. The references found in the chapter notes can be consulted for details that we did not include.

Paxos

Dienstag, 9. Januar 2018 13:51

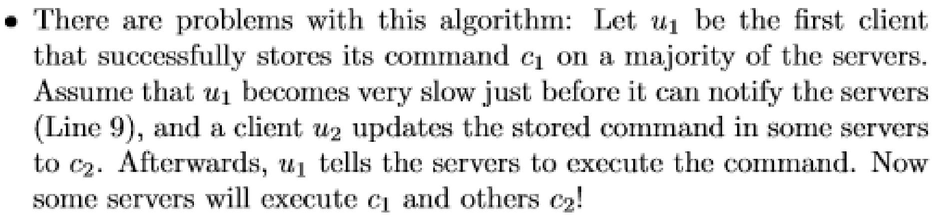
Ticket

Expires if the server issues a new one.

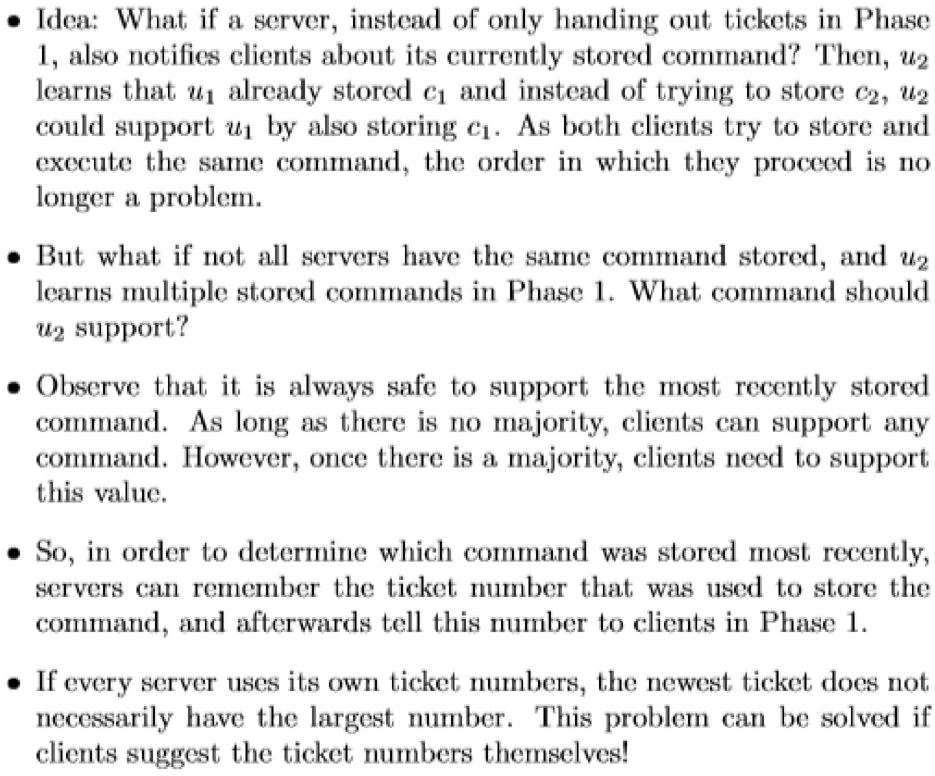


Algorithm 1.12 Naïve Ticket Protocol

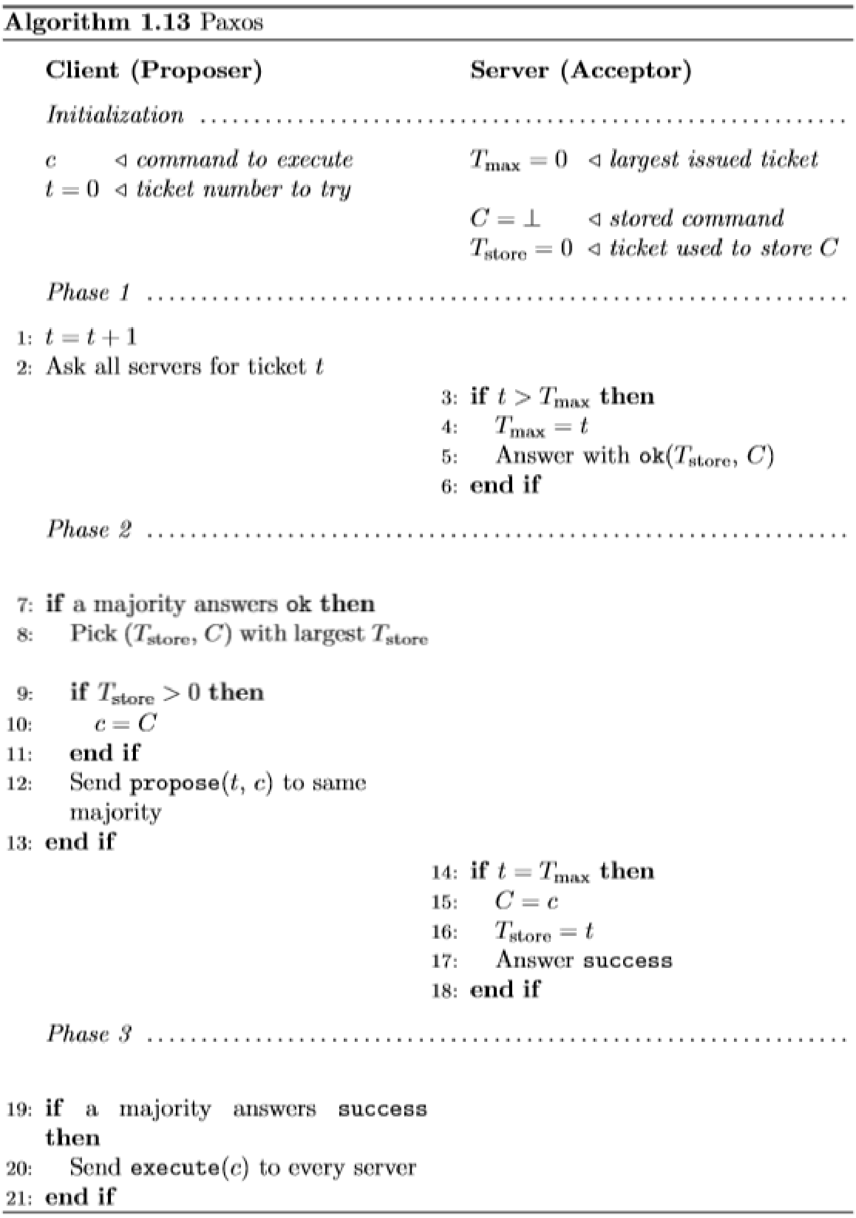
```
    Phase 1
 1: Client asks all servers for a ticket
    Phase 2
 2: if a majority of the servers replied then
 3:    Client sends command together with ticket to each server
 4:    Server stores command only if ticket is still valid, and replies to client
 5: else
 6:    Client waits, and then starts with Phase 1 again
 7: end if
    Phase 3
 8: if client hears a positive answer from a majority of the servers then
 9:    Client tells servers to execute the stored command
10: else
11:    Client waits, and then starts with Phase 1 again
12: end if
```



- There are problems with this algorithm: Let u_1 be the first client that successfully stores its command c_1 on a majority of the servers. Assume that u_1 becomes very slow just before it can notify the servers (Line 9), and a client u_2 updates the stored command in some servers to c_2. Afterwards, u_1 tells the servers to execute the command. Now some servers will execute c_1 and others c_2!



- Idea: What if a server, instead of only handing out tickets in Phase 1, also notifies clients about its currently stored command? Then, u_2 learns that u_1 already stored c_1 and instead of trying to store c_2, u_2 could support u_1 by also storing c_1. As both clients try to store and execute the same command, the order in which they proceed is no longer a problem.
- But what if not all servers have the same command stored, and u_2 learns multiple stored commands in Phase 1. What command should u_2 support?
- Observe that it is always safe to support the most recently stored command. As long as there is no majority, clients can support any command. However, once there is a majority, clients need to support this value.
- So, in order to determine which command was stored most recently, servers can remember the ticket number that was used to store the command, and afterwards tell this number to clients in Phase 1.
- If every server uses its own ticket numbers, the newest ticket does not necessarily have the largest number. This problem can be solved if clients suggest the ticket numbers themselves!



Algorithm 1.13 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
```

- Note that Paxos cannot make progress if half (or more) of the servers crash, as clients cannot achieve a majority anymore.

- The original description of Paxos uses three roles: Proposers, acceptors and learners. Learners have a trivial role: They do nothing, they just learn from other nodes which command was chosen.

- We assigned every node only one role. In some scenarios, it might be useful to allow a node to have multiple roles. For example in a peer-to-peer scenario nodes need to act as both client and server.

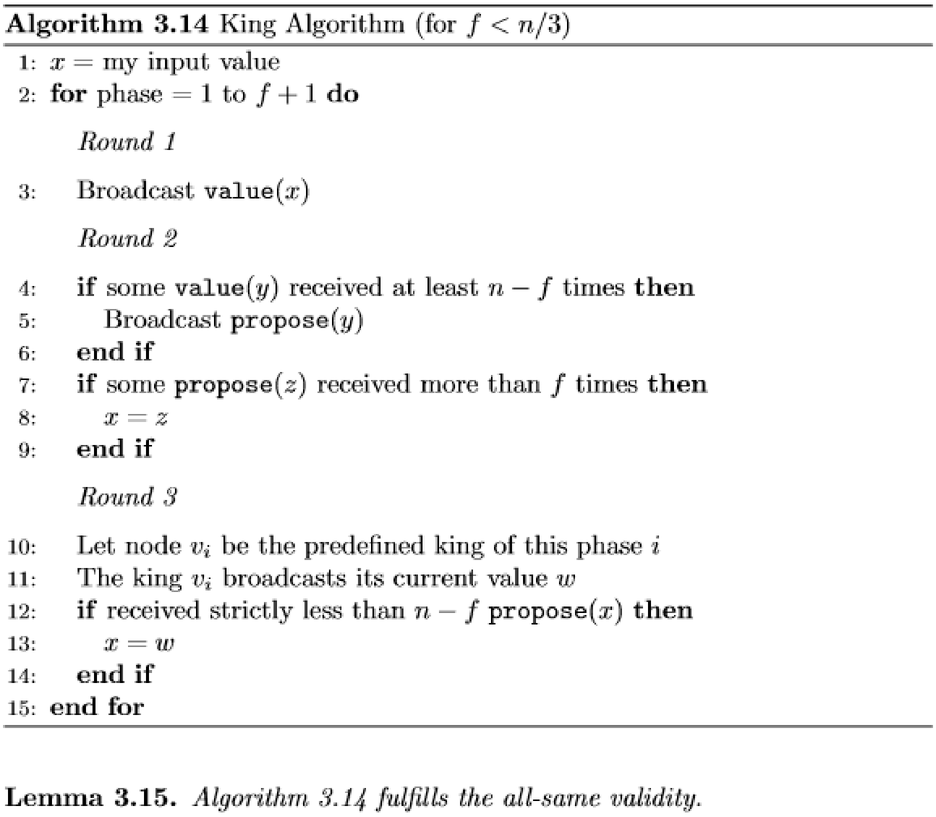
- Clients (Proposers) must be trusted to follow the protocol strictly. However, this is in many scenarios not a reasonable assumption. In such scenarios, the role of the proposer can be executed by a set of servers, and clients need to contact proposers, to propose values in their name.

- So far, we only discussed how a set of nodes can reach decision for a single command with the help of Paxos. We call such a single decision an *instance* of Paxos.

- For state replication as in Definition 1.8, we need to be able to execute multiple commands, we can extend each instance with an instance number, that is sent around with every message. Once the 1^{st} command is chosen, any client can decide to start a new instance and compete for the 2^{nd} command. If a server did not realize that the 1^{st} instance already came to a decision, the server can ask other servers about the decisions to catch up.

King Algorithm

Dienstag, 9. Januar 2018 15:26



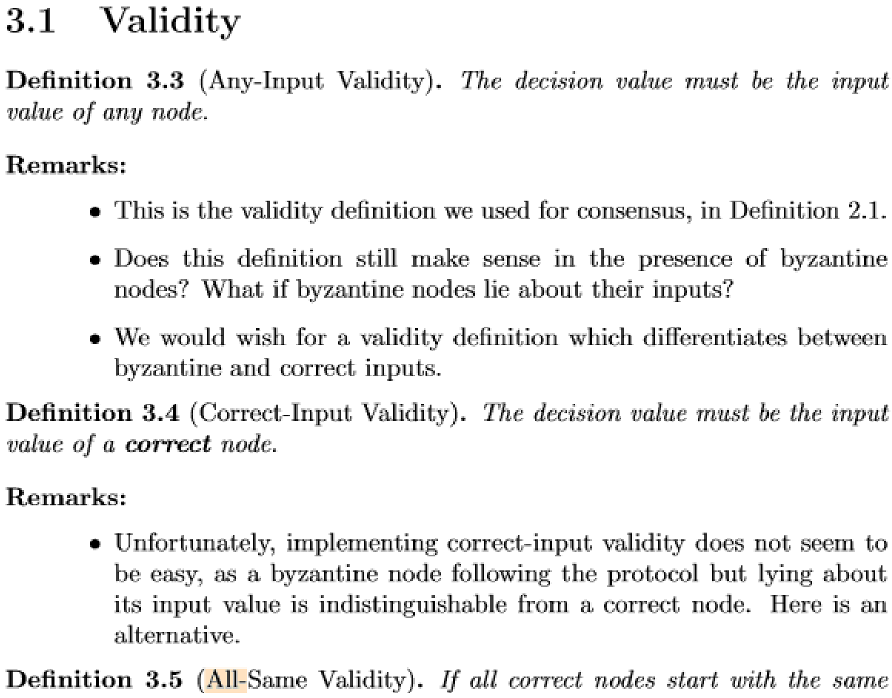
Algorithm 3.14 King Algorithm (for $f < n/3$)

```
1: $x$ = my input value
2: for phase = 1 to $f+1$ do
       Round 1
3:     Broadcast value($x$)
       Round 2
4:     if some value($y$) received at least $n-f$ times then
5:         Broadcast propose($y$)
6:     end if
7:     if some propose($z$) received more than $f$ times then
8:         $x = z$
9:     end if
       Round 3
10:    Let node $v_i$ be the predefined king of this phase $i$
11:    The king $v_i$ broadcasts its current value $w$
12:    if received strictly less than $n-f$ propose($x$) then
13:        $x = w$
14:    end if
15: end for
```

Lemma 3.15. *Algorithm 3.14 fulfills the all-same validity.*

$n - f$ because in line 5 there could be multiple proposals later if the byzantines send different values to different nodes. e.g. when half of the correct nodes value 0 and half value 1, then the byzantines could choose what value a node proposes, because that node would then receive n/2

We need f+1 phases so that at least one king is correct, otherwise the byzantine nodes could send every correct node a different value as king and they would never agree.



3.1 Validity

Definition 3.3 (Any-Input Validity). *The decision value must be the input value of any node.*

Remarks:

- This is the validity definition we used for consensus, in Definition 2.1.
- Does this definition still make sense in the presence of byzantine nodes? What if byzantine nodes lie about their inputs?
- We would wish for a validity definition which differentiates between byzantine and correct inputs.

Definition 3.4 (Correct-Input Validity). *The decision value must be the input value of a* ***correct*** *node.*

Remarks:

- Unfortunately, implementing correct-input validity does not seem to be easy, as a byzantine node following the protocol but lying about its input value is indistinguishable from a correct node. Here is an alternative.

Definition 3.5 (All-Same Validity). *If all correct nodes start with the same* its input value is indistinguishable from a correct node. Here is an alternative.

Definition 3.5 (All-Same Validity). *If all correct nodes start with the same input v, the decision value must be v.*

Remarks:

- If the decision values are binary, then correct-input validity is induced by all-same validity.
- If the input values are not binary, but for example from sensors that deliever values in $\mathbb{R}$, all-same validity is in most scenarios not really useful.

Definition 3.6 (Median Validity). *If the input values are orderable, e.g. $v \in \mathbb{R}$, byzantine outliers can be prevented by agreeing on a value close to the median of the correct input values – how close depends on the number of byzantine nodes f.*

Hashing

Mittwoch, 10. Januar 2018 08:05

Consistent Hashing

1. Choose a set of hash functions
2. Hash the filename
3. Hash the current nodes name
4. Store a copy of the file in the node where (1) and (2) differ the least for any hash function used

Number of stored values expected is $\frac{hashMaps \cdot Values}{nodes}$

Split Ordered Lists

Not relevant for this exam but This paper explains it. I stumbled about this question which I already had upvoted before 2017 apparently.

Locks

Mittwoch, 10. Januar 2018 09:18

Array Lock / Anderson Lock

Every thread has an array element that is set to *false* unless they have the permission to acquire the lock.

Think of the array as a ring, because we calculate indices modulo its size.

There is a shared AtomicInteger pointing to the tail. A new thread increases this integer and starts spinning on the (previous) tail flag. once it becomes true, the thread enters the critical section.

When it leaves the critical section, it sets the flag in the tail to true.

If no thread is spinning on it, the next thread will read true as soon as it enters, otherwise the spinning starts again.

The ring needs to provide a flag for every of the *n* processes. If there are *L* locks, that takes Memory of $O(Ln)$.

Each thread only spins on one memory location so there is not much invalidation traffic.

CLH Queue Lock

Every thread creates a Qnode which has a successor and a boolean field *wantLock*.

When a thread queues up, it uses an atomic GAS to set itself as the new tail and then starts spinning on its predecessors *wantLock*. Once it is false, the thread enters the critical section. … Now it wants to leave again. It sets its own *wantLock* to false. Its Qnode might now be the tail or spun on, so we leave it existent. But our predecessor does not need their Qnode anymore, so we take that Qnode and reuse it next time we want to lock. That way, we only need *n* memory allocated. (Bad on NUMA though).

That means that if we assume a thread to only hold one lock at a time and we have *L* locks, we only need to store *n* Qnodes (for every thread one) and additional *L* Qnodes as tails of empty queues. ⇒ $Memory \in O(L+n)$

MCS Queue Lock

Just like CLH, but every thread owns its own flag, so it's better on NUMA nodes.

When a thread queues up, it sets itself as the new tail and spins on its own *hasLock*. Once it is true, the thread enters the critical section.

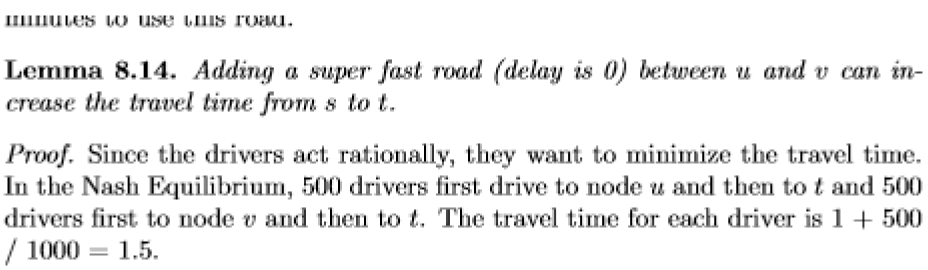
To leave, if it has a successor, it sets the *hasLock* of the successor to true. Our successor does not depend on us, so we can reuse our node to join the same or a different queue lock again.

If the *next* field is null, it uses compareAndSet to set the tail to null if the tail is currently this thread. if that worked, yey. If it didn't, some other thread has already set itself to tail but not yet set itself to next, so we have to spin on our next flag.

Because every thread can reuse its Qnode and we assume that every thread only holds one lock, we have Memory usage of $O(n+L)$ (because we still need the tail pointers)

Braess paradoxon

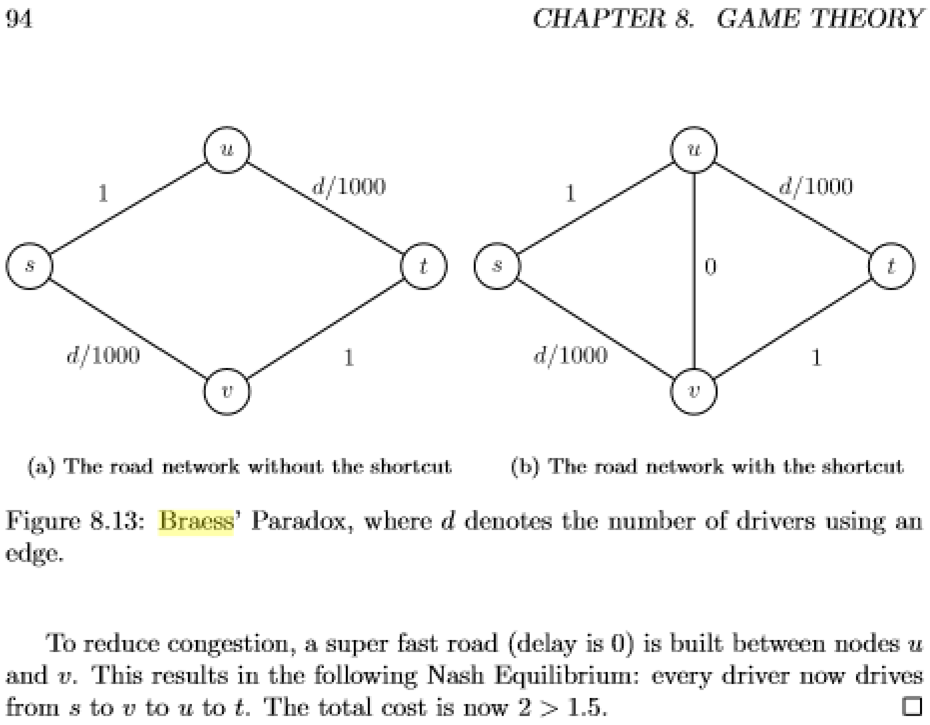
Mittwoch, 10. Januar 2018 13:46



minutes to use this road.

Lemma 8.14. *Adding a super fast road (delay is 0) between u and v can increase the travel time from s to t.*

Proof. Since the drivers act rationally, they want to minimize the travel time. In the Nash Equilibrium, 500 drivers first drive to node u and then to t and 500 drivers first to node v and then to t. The travel time for each driver is 1 + 500 / 1000 = 1.5.



(a) The road network without the shortcut (b) The road network with the shortcut

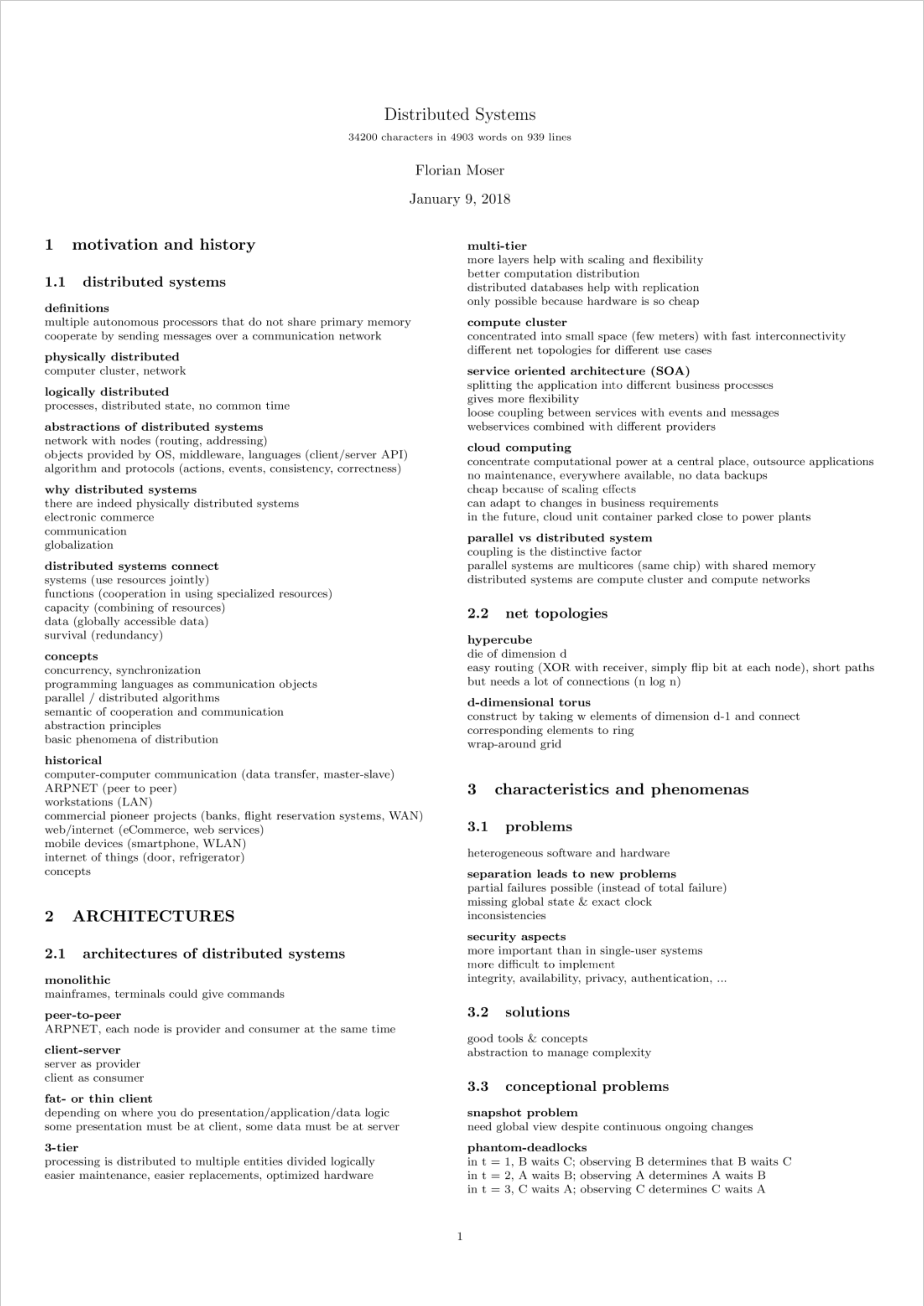
Figure 8.13: Braess' Paradox, where d denotes the number of drivers using an edge.

To reduce congestion, a super fast road (delay is 0) is built between nodes u and v. This results in the following Nash Equilibrium: every driver now drives from s to v to u to t. The total cost is now $2 > 1.5$. ☐

FLOMO

Mittwoch, 10. Januar 2018 14:03

FloMo_DistributedSystemsPart1



Distributed Systems

34200 characters in 4903 words on 939 lines

Florian Moser

January 9, 2018

1 motivation and history

1.1 distributed systems

definitions
multiple autonomous processors that do not share primary memory
cooperate by sending messages over a communication network

physically distributed
computer cluster, network

logically distributed
processes, distributed state, no common time

abstractions of distributed systems
network with nodes (routing, addressing)
objects provided by OS, middleware, languages (client/server API)
algorithm and protocols (actions, events, consistency, correctness)

why distributed systems
there are indeed physically distributed systems
electronic commerce
communication
globalization

distributed systems connect
systems (use resources jointly)
functions (cooperation in using specialized resources)
capacity (combining of resources)
data (globally accessible data)
survival (redundancy)

concepts
concurrency, synchronization
programming languages as communication objects
parallel / distributed algorithms
semantic of cooperation and communication
abstraction principles
basic phenomena of distribution

historical
computer-computer communication (data transfer, master-slave)
ARPNET (peer to peer)
workstations (LAN)
commercial pioneer projects (banks, flight reservation systems, WAN)
web/internet (eCommerce, web services)
mobile devices (smartphone, WLAN)
internet of things (door, refrigerator)
concepts

2 ARCHITECTURES

2.1 architectures of distributed systems

monolithic
mainframes, terminals could give commands

peer-to-peer
ARPNET, each node is provider and consumer at the same time

client-server
server as provider
client as consumer

fat- or thin client
depending on where you do presentation/application/data logic
some presentation must be at client, some data must be at server

3-tier
processing is distributed to multiple entities divided logically
easier maintenance, easier replacements, optimized hardware

multi-tier
more layers help with scaling and flexibility
better computation distribution
distributed databases help with replication
only possible because hardware is so cheap

compute cluster
concentrated into small space (few meters) with fast interconnectivity
different net topologies for different use cases

service oriented architecture (SOA)
splitting the application into different business processes
gives more flexibility
loose coupling between services with events and messages
webservices combined with different providers

cloud computing
concentrate computational power at a central place, outsource applications
no maintenance, everywhere available, no data backups
cheap because of scaling effects
can adapt to changes in business requirements
in the future, cloud unit container parked close to power plants

parallel vs distributed system
coupling is the distinctive factor
parallel systems are multicores (same chip) with shared memory
distributed systems are compute cluster and compute networks

2.2 net topologies

hypercube
die of dimension d
easy routing (XOR with receiver, simply flip bit at each node), short paths
but needs a lot of connections (n log n)

d-dimensional torus
construct by taking w elements of dimension d-1 and connect
corresponding elements to ring
wrap-around grid

3 characteristics and phenomenas

3.1 problems

heterogeneous software and hardware

separation leads to new problems
partial failures possible (instead of total failure)
missing global state & exact clock
inconsistencies

security aspects
more important than in single-user systems
more difficult to implement
integrity, availability, privacy, authentication, ...

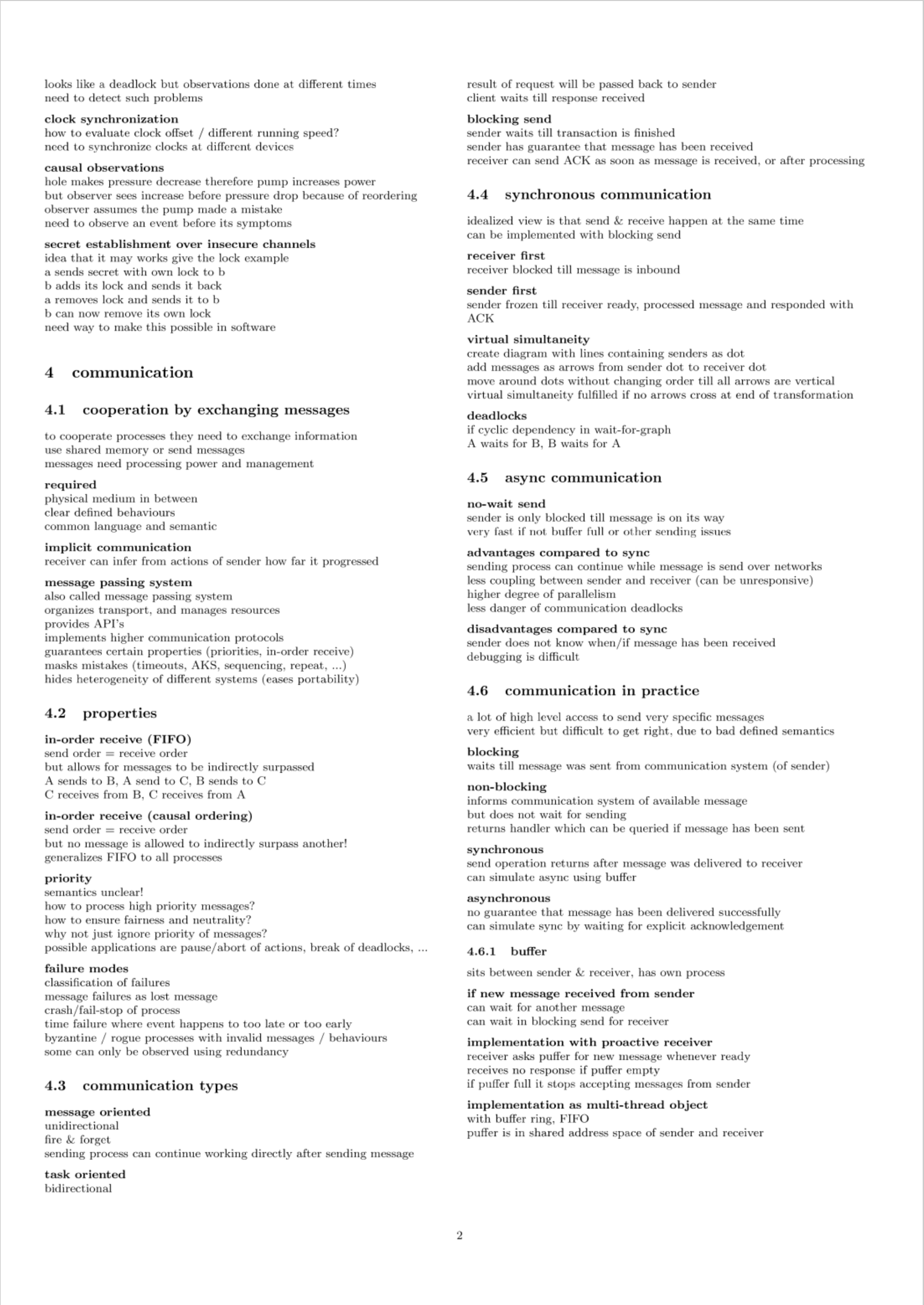
3.2 solutions

good tools & concepts
abstraction to manage complexity

3.3 conceptional problems

snapshot problem
need global view despite continuous ongoing changes

phantom-deadlocks
in t = 1, B waits C; observing B determines that B waits C
in t = 2, A waits B; observing A determines A waits B
in t = 3, C waits A; observing C determines C waits A



looks like a deadlock but observations done at different times
need to detect such problems

clock synchronization
how to evaluate clock offset / different running speed?
need to synchronize clocks at different devices

causal observations
hole makes pressure decrease therefore pump increases power
but observer sees increase before pressure drop because of reordering
observer assumes the pump made a mistake
need to observe an event before its symptoms

secret establishment over insecure channels
idea that it may works give the lock example
a sends secret with own lock to b
b adds its lock and sends it back
a removes lock and sends it to b
b can now remove its own lock
need way to make this possible in software

4 communication

4.1 cooperation by exchanging messages

to cooperate processes they need to exchange information
use shared memory or send messages
messages need processing power and management

required
physical medium in between
clear defined behaviours
common language and semantic

implicit communication
receiver can infer from actions of sender how far it progressed

message passing system
also called message passing system
organizes transport, and manages resources
provides API's
implements higher communication protocols
guarantees certain properties (priorities, in-order receive)
masks mistakes (timeouts, AKS, sequencing, repeat, ...)
hides heterogeneity of different systems (eases portability)

4.2 properties

in-order receive (FIFO)
send order = receive order
but allows for messages to be indirectly surpassed
A sends to B, A send to C, B sends to C
C receives from B, C receives from A

in-order receive (causal ordering)
send order = receive order
but no message is allowed to indirectly surpass another!
generalizes FIFO to all processes

priority
semantics unclear!
how to process high priority messages?
how to ensure fairness and neutrality?
why not just ignore priority of messages?
possible applications are pause/abort of actions, break of deadlocks, ...

failure modes
classification of failures
message failures as lost message
crash/fail-stop of process
time failure where event happens to too late or too early
byzantine / rogue processes with invalid messages / behaviours
some can only be observed using redundancy

4.3 communication types

message oriented
unidirectional
fire & forget
sending process can continue working directly after sending message

task oriented
bidirectional
result of request will be passed back to sender
client waits till response received

blocking send
sender waits till transaction is finished
sender has guarantee that message has been received
receiver can send ACK as soon as message is received, or after processing

4.4 synchronous communication

idealized view is that send & receive happen at the same time
can be implemented with blocking send

receiver first
receiver blocked till message is inbound

sender first
sender frozen till receiver ready, processed message and responded with ACK

virtual simultaneity
create diagram with lines containing senders as dot
add messages as arrows from sender dot to receiver dot
move around dots without changing order till all arrows are vertical
virtual simultaneity fulfilled if no arrows cross at end of transformation

deadlocks
if cyclic dependency in wait-for-graph
A waits for B, B waits for A

4.5 async communication

no-wait send
sender is only blocked till message is on its way
very fast if not buffer full or other sending issues

advantages compared to sync
sending process can continue while message is send over networks
less coupling between sender and receiver (can be unresponsive)
higher degree of parallelism
less danger of communication deadlocks

disadvantages compared to sync
sender does not know when/if message has been received
debugging is difficult

4.6 communication in practice

a lot of high level access to send very specific messages
very efficient but difficult to get right, due to bad defined semantics

blocking
waits till message was sent from communication system (of sender)

non-blocking
informs communication system of available message
but does not wait for sending
returns handler which can be queried if message has been sent

synchronous
send operation returns after message was delivered to receiver
can simulate async using buffer

asynchronous
no guarantee that message has been delivered successfully
can simulate sync by waiting for explicit acknowledgement

4.6.1 buffer

sits between sender & receiver, has own process

if new message received from sender
can wait for another message
can wait in blocking send for receiver

implementation with proactive receiver
receiver asks puffer for new message whenever ready
receives no response if puffer empty
if puffer full it stops accepting messages from sender

implementation as multi-thread object
with buffer ring, FIFO
puffer is in shared address space of sender and receiver

4.7 communication mechanisms

table
asynchronous (x1), synchronous (x2)
message (y1), task (y2)

most commonly used
asynchronous messages
synchronous tasks

RPC (x2, y2)
executes task on other machine, waits for confirmation
RPC (remove procedure call)

asynchronous RPC (x1, y2)
also called Remove Service Invocation
parallelisation of sever/client possible
to implement use await, callbacks, future-variables
C# Task, only waits if not finished computation

4.7.1 no-wait send (x1, y1)

implementations
with puffer; as seen above

pro
server/client are properly separated
simple implementation

contra
sender does not know if message has been received
needs to use puffers, which causes overhead (copying, space management)
needs flow control mechanisms

4.7.2 rendezvous (x2, y2)

three implementations
sender repeatedly contacts receiver till no more NACK received
sender sends message which is put in puffer at receiver
receiver sends ACK to sender as soon as he is ready

pro
small buffers only

contra
busy waiting
complex protocol

4.8 RPC

like a procedure call
clear semantics for executor
simple to program in high-level API's (like any other method call)
abstract complexity due to distributed factors as good as possible

example call
client calls procedure, stubs marshal, transport sends request
server receives request, stubs unpacking arguments, local procedure call
server produces result, stubs marshal, transport sends reply
client receives reply, stubs unpack result, result is returned

stubs
take care of packing/unpacking (converting representations)
set timeouts, raise exceptions, pass messages
simulate "local" procedure call
can be generated

capability of data structures
how to convert representations?
numbers (big endian / little endian)
characters (UTF8 / ASCII)
types like strings (length / '/0')
arrays (row / column wise)

marshalling
creating of message from parameters
flattening complex objects
use representations the other party understands

conversion
converting of objects in common notations, for example as XML
or "receiver makes it right" (send whatever, receiver has to correct)

transparency
RPC should behave as local procedure calls
not always possible (server/network failure, difference in live cycles)

performance transparency
RPC's slower than real local procedure call
communication size can be quite big
sudden delays possible

performance analysis
transport cheap
conversion (as headers, checksums) is expensive
copying is expensive
context-switch is relevant when using small messages

place transparency
target must be named explicitly
no global variables
no pointers/references

callback RPC
temporary role reversal
client receives status updates from server

context handles
structure which contains context information
enables server to remember client
is passed to client in reply, is included in the next request

broadcast/multicast
request is sent to other servers at the same time
broadcast sends to all, multicast only to some)
RPC is finished after first response (or client can wait for more results)

security
authentication when creating connection ("binding")
authentication of each single request
end-to-end encryption of messages
make it impossible to modify (digital signature, checksums, MAC)

"secure RPC" as example
session key k encrypts messages
request contains encrypted timestamp
first request contains time window
server accepts request if timestamp bigger than last, if inside time window
server reply contains the last timestamp for client-side authentication
encrypted timestamp ensure attacker can't generate message
small time window ensures attacker can't bruteforce the key

4.8.1 failure transparency

message can be lost (or too slow; can't be differentiated)
multiple failure causes, but mostly all-or-nothing behaviour
partial system fault (client or server) typical
different view of transaction state between server & client

missing request message
resend request after timeout
but how to choose timeout, how many retries, maybe server just too slow
possible repeating requests due to resend two requests

missing reply message
same treatment as missing request, client can't know difference
server can cache replies, resend if same request received again
but how to clean up cache (time & reply ACK's)

server crash
client can't differentiate crash before, after, in procedure
maybe inconsistent server state after reboot

client crash / not longer interested
server waits indefinitely for ACK of client
blocks resources due to orphans at server
use "is-alive" ping while running procedures, discard old processes
let client explicitly contact server for cleanup

4.8.2 failure semantics

maybe-semantic
no repetition of request
easy and efficient
useful for lookup services

at-least-once semantics
automatically repeat requests
stateless protocol on server side (no duplicates can be discovered)
nice for idempotent operations (reading a file)
maybe uses more resources than explicitly necessary

at-most-once semantic
can discover duplicates, then just resends persisted replies
nice for non-idempotent stuff
more expensive than at-least-once

exactly-once
not really possible
because if crashes occur no computations take place

4.9 more concepts

ports
communication end point which abstracts structure of receiver
one process can have multiple ports

channels
for example using ports
can also name them and send; read from them
broadcast with subscribers
very flexible because can change the connection structure any time

software bus
anonymous
can react to events
can send events

event channels
anonymous
can register for events
dispatches events
participants need to be always listening (maybe use buffers)

zeitüberwachter nachrichtenempfang
receiver sets max time he wants to wait, else other code is executed
also useful for blocking send

5 client-server

5.1 general

server provides infos
client consumes infos and provides front end for user

5.2 server

iterative server
will process one request at a time
take new request from puffer if finished with old
easy to realize, good for trivial stuff

concurrent server
concurrent processing of multiple requests

concurrent server with dynamic handlers
master creates slave "handler" for each request
may has fixed number of slaves ready for usage "process preallocation"
slave communicates directly with receiver
ceiling amount of slaves at the same time

stateless servers
every request must be fully described
HTTP theoretically stateless

state servers
can identify repeated requests, therefore idempotent
in HTTP server needs to identify customers

5.3 client

possibility for async RPC to communicate with server

5.4 tasks

non-pure
like writing a file

pure ("zustandsinvariant")
simple lookups

idempotent tasks
repeated tasks lead to same result (but can be non-pure)

5.5 web stuff

identify customers
URL rewriting, dynamic webpages
cookie can be the context-handle
identify with IP (but not uniquely)

SOA vs ROA
service vs resource oriented architecture (SOAP vs REST)

5.5.1 lookup service

connects client & server
server makes itself known in LUS (lookup service)
client asks LUS and import the provided service configuration

pro
register multiple provides for same task for scalability
validate authorization
can use polling to see if server is still responsive
can manage multiple versions

contra
lookup needs time
LUS is single point of failure
clients need to know LUS!

5.6 middleware

RPC libraries
client-sever paradigm
easy interface, code generation
security such as authorization, authentication, encryption

client-sever distribution platforms
lookup service, global namespace, global filesystem
supported multi threading

object-based distribution platforms
cooperation between distributed objects
object-oriented interface
object request broker (ORB) functions as middleware

5.6.1 CORBA

ORB to redirect method calls
IDL interface description language with stub generation
CORBA update failed in 2000, different interests and better competition

possible methods calls
synchronous (waits for response)
delayed synchronous (can get object later)
one way (fire & forget)

5.7 web services example SOAP

example for client-server model
internet is very homogeneous
web services define platform independent interface

keywords
HTTP (Hyper Text Transport Protocol) as transport layer
UDDI (Universal Description, Discovery and Integration) as lookup service
SOAP (Simple Object Access Protocol) specifies protocol
WSDL (Web Services Description Language) as service description

UUDI
currently not available cause money

SOAP envelope
each SOAP request is sent in an envelope
body containing the data serialized as XML
header which may specifies additional options

SOAP engine
server stubs are generated from a webservice implementation (buttom up)
client stubs from WSDL description (top down)

5.7.1 WSDL xml nodes

definitions
targetNamespace contains current element
xmlns;NS to add more namespaces

types
import other schemas, add own elements, add complexTypes

messages
can name messages, specifying the needed parameters

portType
describes a method
has operation sub nodes which describe input, messages and faults

binding
what protocol to use HTTP, SMTP, UDP
multiple bindings possible

service
where to access services
maps a binding to a concrete address (URL)

5.8 REST

ROA architecture
uses URI (Unique Resource Identifier)
created for the web, as best way to use it

REpresentational State Transfer
not resource, but representations are transmitted
get access to state of resource, can alter & send them back

usage model
hypermedia as engine of application state
client knows only base uri
server broadcast other uris per form or hyperlinks

5.8.1 principles

client-server
consists of components who can connect to clients, to server or both
User Agend which creates requests
Intermediary which redirects request potentially modifying them
Origin Server which has control of resources

statelessness
request contains all info for processing; context held client-side
crash/orphans less critical, easier scaling and monitoring, caching

caching
meta-data determines how long response is valid
clients/servers consult cache for answers without further processing

uniform interface
addressing done with URI
requests are standardized (GET, POST, ...)
standard representations (XML, JSON, ...)
resources can provide multiple formats, client chooses applicable

layered system
clients don't know about server
intermediaries can be added at any point

code on demand
server can externalize logic to the client

5.8.2 properties

scalability
statelessness allows efficient servers / load balancing
caching reduces communications

adaptability
uniform interfaces decouple server & client
layering allows manipulation later
code on demand allows to update active clients

observability
requests which contain all infos are easily traceable

reliability
thorough uniform interfaces & layering allows for redundancy

5.8.3 state persisting

resource state
static templates & resources from server

client state
active rendered state & its history
bookmarks preserve full URI
back button of browser allows to go back to the prior state

statelessness means
client & server state are strictly decoupled (hence sessions)

bad practices
url rewriting; encode client-specific information in requests
cookies; server has state of client possibly changing request interpretation
back button; server/client state disjoint, previous URI may stops working

6 Broadcast / Multicast

6.1 group communication

idealized
memory based communication where all receive immediately
message based communication where all receive at same time

pull
client requests infos from server
event driven

push
server sends infos to client
demand driven
client subscribes to channel, server publishes news

6.1.1 broadcast

target
send message to all members

real
network often not multicast, can simulate by sending a lot of single messages
non-deterministic time shift, no sending guarantees
multicast protocol needs to approximate

lost messages
due to network overload, receiver not listening
receivers are not in the same state anymore
need redundancy and complicated protocols to solve this

best effort broadcast
typically simple send without ACK
used to distribute non-critical information
used to implement higher protocols
very efficient if successful
no guarantees if and how many messages are delivered

reliable broadcast (wait for ACK)
waits for ACk for every single message
resends if none received
bad scaling because of polluting ACKs, need to distinguish duplicates

reliable broadcast (with NACK)
broadcasts contain identifier/sequence set by sender
receiver broadcasts missing messages with NACK, sender resends
sender can send empty messages to ensure receiver missed no messages
does not help if server / network crashes

reliable broadcast (flooding)
send message to all nodes except the originator
remember the sequence number of the message to avoid flooding twice
need only one connection to a not crashed server to receive the message

broadcast message ordering
can order messages differently
stricter semantics, principle is as cheap as possible, as perfect as needed
difficult to implement, less parallelization, less performance

FIFO
all broadcast messages from same sender are received in same order
but causality is not guaranteed

causal order
causality exists if there is a connection in space-time diagram from A to B
implies all messages are received according to the rules of causality

atomic
if two process receive the same two messages, they are in the same order
does not imply FIFO & causal order

order atomic with central sequencing
unicast from sender to sequencer
broadcast from sequencer to other members
sequencer waits for ACK before sending next message

order atomic with token
single token created which contains sequence number
member with token can send message
token is passed around in predefined order
messages delivered according to sequencing number
new token generated if owner timeouts
use explicit token request instead of passing if a lot of members

causal + atomic
comparable with memory based communication

also called virtual synchronous communication
events happen at the same logical time (which may not equals real time)
logical time only takes causality of messages into account
same as synchronous inside the system

6.1.2 multicast

target
send message to subgroup of members

why
simplify addressing
hiding of group assignment
logical unicast, groups have replaced individuals

hidden channels
messages which leave groups and return through another node
if those count as casually depended must be defined

dynamic groups
members can join/leave group at any time
what happens if this occurs while multicast operation in progress?
entry/exit should be atomic
senders should see the real members of the group at the time of sending

6.1.3 tuple rooms

target
decouple sender and receiver

what
virtual, global storage
data can be added, changed, removed from all members

linda
language for tuple rooms
out(t) (adds), in(t) (reads & removes), read(t) (reads)
tuple room implemented as associative storage
get tuple by condition; (”hi”, ?p) is tuple with ”hi” as first attribute
asynchronous operations (readp and inp(t) do not block, return bool)
synchronous operations (read and in(t) wait for correct tuple to appear)

able to model server-client
client places requests and waits for responses
server processes requests and places responses
client; out(”req”, guid, params); in(”resp”, guid, ?result);
server; in(”req”, ?guid, ?params); out(”resp”, guid, result);

some tuple rooms support additionally
persistence (tuple will not perish after termination)
transaction (important if multiple servers access tuple room)

problems
central tuple room is weakest link
replicated / disjunct distributed tuple rooms
difficult for structured programming and verification

JavaSpaces
tuple room for java
can persist objects and behaviour
part of Jini (middleware for java)
can transport code to receiver, use common objects
ordering of operations between different processes undefined

6.2 logical time

time is useful
state of system at specific point in time
causality between events (if x was before y, y cannot have caused x)
fair mutual exclusion (longest waiting is served)
other applications as timestamps

real time
asymmetric, transitivity, irreflexivity, linearity, infinite, continuous (always point in between), metric, every point is eventually reached

causal relation ($x<y$) exactly when
x,y from same process and x before y
x is a send, and y its corresponding receive
there is a z for $x<z$ and $z<y$
solve this with timestamps, called C(x)
if $e < e'$ then $C(e) < C(e')$ (time must imply causality)

logical clocks by lamport
at each event the clock of each process is increased
at send, send own clock inside request
at receive, take max(own, foreign clock) then increase it
to get injective ordering include process id when you need to decide

vector clocks
generalization of logical clock
each process has its own counter (sizeof(vector) = count(processes))

7 MUTEX

7.1 mutex

conflict with unique resource

solution requirements
safety (nothing bad will ever happen, exclusive access guaranteed)
liveness (eventually something good will happen, progress)
fairness (all have to make progress, all profit)

manager
manager coordinates access, has queue of processes which are waiting
process sends ”request”, waits ”grant”, notifies afterwards ”release”
simple, few messages
manager is single point of failure

7.1.1 global queue

replicate queue at each process
use FIFO queues, messages contain timestamp (real or Lamport)
requests and releases are sent to all, requests are confirmed with ACK

Lamport
3(n-1) messages
each member has own queue
can use mutex if first in queue & received ACK from all others
request mutex by broadcasting ”request” with timestamp, add to own queue
release mutex by broadcast ”release”, remove from queue
on receive of request, save into own queue and confirm with ACK
on receive of release, remove it from own queue

Ricart / Agrawala
2(n-1) messages
can use mutex when received ACK from all other members
request mutex by broadcasting ”request” with timestamp
on ”request”, send reply if (!self_requested ||sender_time <my_request_time)
else wait till released mutex

8 Security

8.1 security

requirements
authorization (only specific entities have access)
privacy (attackers can't read message)
authentication (sender is verified)
integrity (message is unmodified)
availability (no DoS possible)

need to fulfil advanced requirements
non-repudiation (cannot deny the sending/reception of message)
prosecution (needs logging, need access to otherwise private keys)
compliance (conform to law, terms)

problems with distribution
harder to guarantee security in distributed systems
no central security authority
systems often open which allows to easier spot possible attack points
standardized protocols are attackable as one can craft own packets
spatial distance makes it hard to locate attacker
heavy usage makes an attack more valuable
physical separation often not possible
tools such as wireless make it easier to launch an attack
heterogeneity allows more attack points
hard to enforce common security policy

passive attacks
observe communication
”who when with whom”
read messages

active attacks
modify messages (modify, remove, create, resend, delay)
impersonate (behave as another process, use foreign passwords)

malicious usage of services
deny usage of services with DoS

authenticity
of service, confirm that connected to real service
of message, verify sender & verify message integrity
of saved data, verify integrity

security
want to provide encryption, authorization, authentication
encrypt message, changes become visible
peer-authentication, ask question only associate can answer
password, but not tied to identity (sniffing, secrecy not enforcable)
one-way functions, but no mathematical proof such functions exist

one-way functions
pick public non-inversible hash function f
client chooses n_0, hashes n times, sends n_n to server
each new connection sends one n before, server able to authenticate client
each password used only once (no reply attacks)
server has no secrets as only already invalid passwords persisted

8.2 cryptosystems

encrypt with K1, decrypt with K2
asymmetric if K1 != K2
decryption is infeasible without the key
procedure should be public because difficult to keep secret, feedback useful

8.2.1 tricks

biased random number generators
1 / 0 may have different probabilities
therefore only choose pairs of 01 (=0) or 10 (=1)
transform 01001101011110 → 01010110

8.2.2 symmetric keys

advantages
1000 times faster than asymmetric

disadvantages
key must be secret
each communication partner needs different key
need to manage keys, high complexity
need to secretly exchange keys

examples
DES, AES

one-time pad
perfect encryption
crypto = M XOR pad
M = crypto XOR pad
if pad applied twice it is simply cancelled out
pad must never be used twice, or repeated, must be real random numbers
not practical because need large amount of authenticated encryption bits

8.2.3 asymmetric

public key server
must be authentic, communication must be secured

public key service
distributes certificated public key and its private key to member
transfers session keys securely and authenticated to the members

properties
every member has (p,s) public key p, secret key s
m can't be derived from {m}_p
s can't be derived from p or {m}_p with known m, p
m = {{m}_p}_s
maybe additionally m = {{m}_s}_p

advantages
exchange keys easy (p public, s not exchanged, 2n keys for n members)
authenticates owner (if able to decrypt {m}_p authentication successful)
digital signature (if able to generate {m}_s authentication successful)

8.2.4 authentication

symmetric way
A and B share key k
A → B n
B → A m' = {n}_k
A verifies that {m'}_k = n

asymmetric way
A → B n
B → A m_1 = {"command", n}_sB
A decrypts m_1 with public key of B and executes "command"
safe against replays (because of nonce), but not MitM
introduce public key server that A needs not to save B public key
need to secure public key server against tampering, impersonation

asymmetric way (both ways, introduce session key K)
n are nonces, m are sent messages, K is session key
use asymmetric to send nonces (na, nb)
nonces confirm key is established with correct associate
A → B m_1 = {na}_pB
B → A m_2 = {na, nb, K}_pA
A → B m_3 = {nb}_K

8.2.5 key agreement

with one time pads
A → B m_1 = {k}_a
B → A m_2 = {m_1}_b
A can now XOR with a, and learns b
A → B m_3 = {k}_b
but advisory can learn k too if all messages known

with diffie hellman
choose public c and p
A → B m_1 = 5^a mod p
B → A m_2 = 5^b mod p
key = m_1^b = m_2^a
not safe against MitM

8.2.6 attacks

replays
simply resend messages without knowing exact content
can uses nonces which are only valid once
can use increasing sequence numbers
can use encrypted send time and max timeout at receiver

MitM
attacker redirects traffic between A and B to himself

key faking
attacker additionally sits between key server & A
places as MitM between A and B
now can fake the public key of B to one X knows the private key

8.3 interlock protocol

securely communicate with attacker in between
B → A sends challenge only A can answer
A → B sends encrypted answer, but only half of bits
A → B sends rest of the answer
B checks that first message is received in very short time
X can perform MitM by establishing key with B impersonating A
but X needs whole A message to do so
if X forwards first part immediately, X is not able to perform MitM
if X buffers till both messages received then B knows about intruder

8.4 authentication with certificates

certificates of A is singed by a trusted authority
A → B secret encrypted with public key of B
B → A sends back decrypted secret, confirming it has the private key

8.5 zero knowledge proof

A proofs knowledge to B without giving away the solution
verifier and prover interact together
but verifier can only prove to himself that prover knows answer

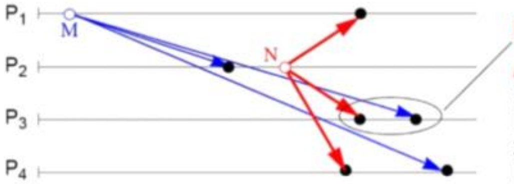
example graph isomorphy
prover says he knows isomorph graphs G1 = G2
prover construct H by renaming random knots of G1 or G2
verifier then requests mapping to G1 or G2
prover can do this easily as he knows H ~G1 and G1 ~G2
process is repeated

8.6 up for discussion

global queue algorithms → some examples please

Wie „gut“ ist atomarer Broadcast?

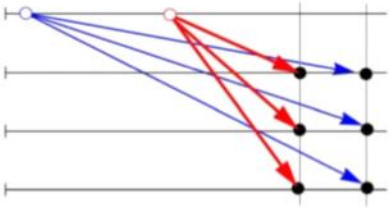
1) Ist atomar auch kausal?



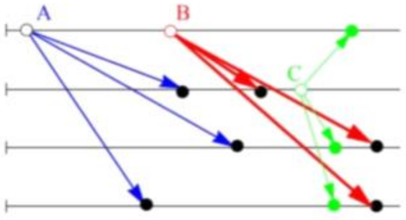
Nicht kausal!

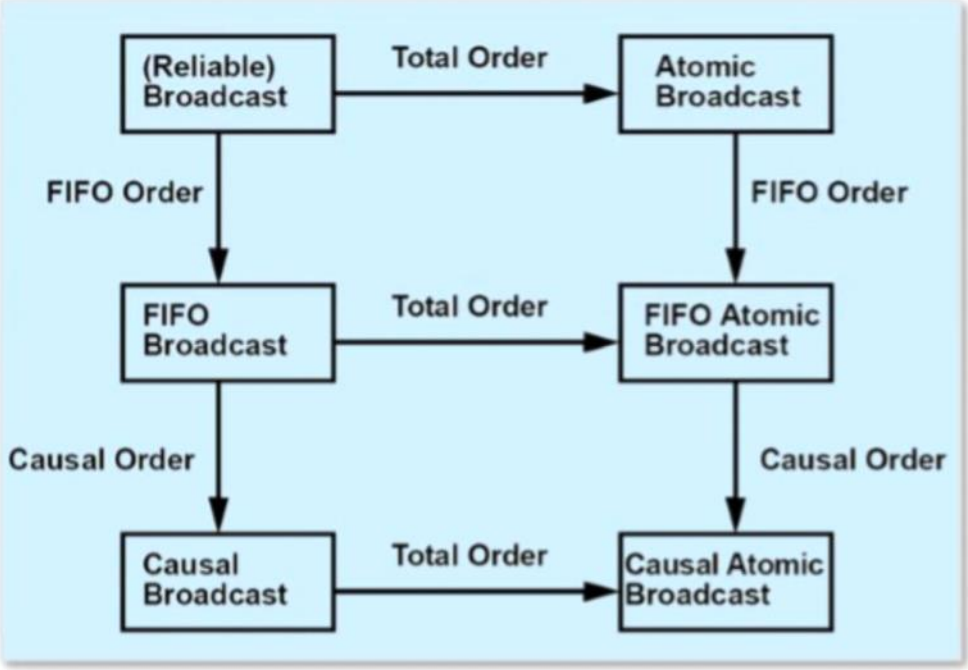
Atomar: P3 und P4 empfangen beide M, N – und zwar in gleicher Reihenfolge

2) Ist atomar wenigstens FIFO?



3) Ist atomar + FIFO evtl. kausal?





FLOMO extracted

Mittwoch, 10. Januar 2018 14:12

snapshot problem
need a global view despite continuous changes

phantom-deadlocks
The observer might see a deadlock because B waits for C, C for A and A for B. But that could have changed in the meantime (every wait is in a different timestep).

clock synchronisation
Assume drift is linear, but they can also have an offset if there is no drift

FIFO
send order = receive order
but allows messages to indirectly surpass other message via a different channel.

Causal Ordering
send order = receive order
indirectly surpassing is not allowed - anything causally dependent on the sending of A will not be received bevor A.

Priority
How to prioritize and how to ensure fairness and neutrality and what fairness means are unclear.

Failure Modes
Crash/fail-stop
time failure (too early or too late)
byzantine/rogue behaviour
problem during sending / receiving

Communication types

message oriented
unidirectional
fire&forget
sending process can continue working directly after sending message

task oriented
bidirectional
result of request will be passed back to sender. Client waits until the response has been received

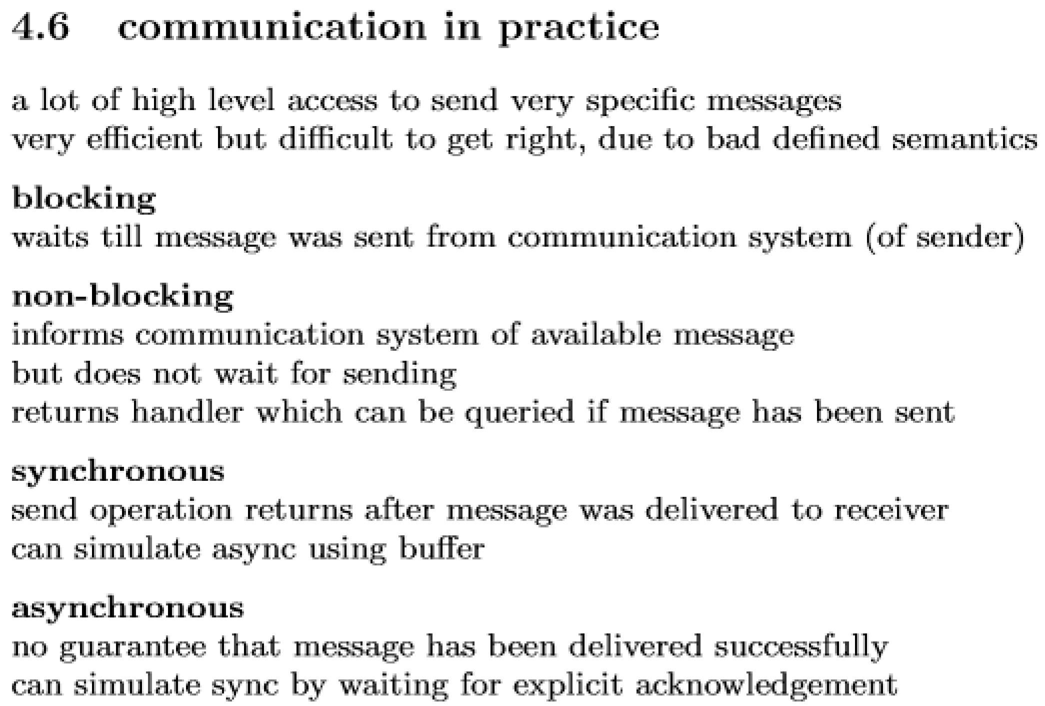
blocking send
The sender waits until it has a guarantee that the message has been received. The receiver might send the ACK before actually processing the message.

synchronous communication

- blocking send and receive. The sender freezes until the receiver was ready, processed the message and responded with ACK
- "virtual simultaneity" : rubber-band movement possible so that simultaneous events are simultaneous
- Deadlocks if cyclic wait-for-graph (both processes are receiving or sending)

async communication

difficult debugging but is faster and less coupling. higher degree of parallelism, less chance for deadlocks based on communication



4.6 communication in practice

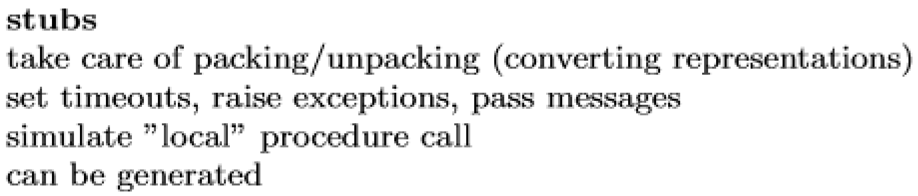
a lot of high level access to send very specific messages
very efficient but difficult to get right, due to bad defined semantics

blocking
waits till message was sent from communication system (of sender)

non-blocking
informs communication system of available message
but does not wait for sending
returns handler which can be queried if message has been sent

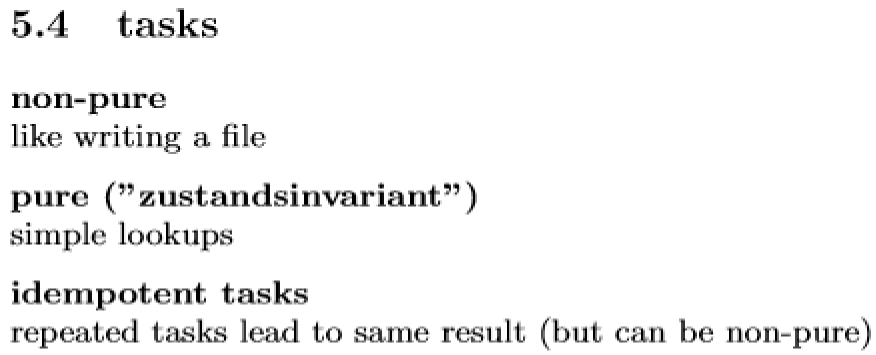
synchronous
send operation returns after message was delivered to receiver
can simulate async using buffer

asynchronous
no guarantee that message has been delivered successfully
can simulate sync by waiting for explicit acknowledgement



stubs
take care of packing/unpacking (converting representations)
set timeouts, raise exceptions, pass messages
simulate "local" procedure call
can be generated

maybe-semantic no repetition of request easy and efficient useful for lookup services
at-least-once semantics automatically repeat requests stateless protocol on server side (no duplicates can be discovered) nice for idempotent operations (reading a file) maybe uses more resources than explicitly necessary
at-most-once semantic can discover duplicates, then just resends persisted replies nice for non-idempotent stuff more expensive than at-least-once
exactly-once not really possible because if crashes occur no computations take place



5.4 tasks

non-pure
like writing a file

pure ("zustandsinvariant")
simple lookups

idempotent tasks
repeated tasks lead to same result (but can be non-pure)

REST

5.8.1 principles

client-server
consists of components who can connect to clients, to server or both
User Agend which creates requests
Intermediary which redirects request potentially modifying them
Origin Server which has control of resources

statelessness
request contains all info for processing; context held client-side
crash/orphans less critical, easier scaling and monitoring, caching

caching
meta-data determines how long response is valid
clients/servers consult cache for answers without further processing

uniform interface
addressing done with URI
requests are standardized (GET, POST, ...)
standard representations (XML, JSON, ...)
resources can provide multiple formats, client chooses applicable

layered system
clients don't know about server
intermediaries can be added at any point

code on demand
server can externalize logic to the client

5.8.2 properties

scalability
statelessness allows efficient servers / load balancing
caching reduces communications

adaptability
uniform interfaces decouple server & client
layering allows manipulation later
code on demand allows to update active clients

observability
requests which contain all infos are easily traceable

reliability
thorough uniform interfaces & layering allows for redundancy

8 Security

8.1 security

requirements
authorization (only specific entities have access)
privacy (attackers can't read message)
authentication (sender is verified)
integrity (message is unmodified)
availability (no DoS possible)

need to fulfil advanced requirements
non-repudiation (cannot deny the sending/reception of message)
prosecution (needs logging, need access to otherwise private keys)
compliance (conform to law, terms)

authenticity
of service, confirm that connected to real service
of message, verify sender & verify message integrity
of saved data, verify integrity

security
want to provide encryption, authorization, authentication
encrypt message, changes become visible
peer-authentication, ask question only associate can answer
password, but not tied to identity (sniffing, secrecy not enforcable)
one-way functions, but no mathematical proof such functions exist

8.2 cryptosystems

encrypt with K1, decrypt with K2
asymmetric if K1 != K2
decryption is infeasible without the key
procedure should be public because difficult to keep secret, feedback useful

8.2.1 tricks

biased random number generators
1 / 0 may have different probabilities
therefore only choose pairs of 01 (=0) or 10 (=1)
transform 01001101011110 → 01010110

8.2.2 symmetric keys

advantages
1000 times faster than asymmetric

disadvantages
key must be secret
each communication partner needs different key
need to manage keys, high complexity
need to secretly exchange keys

examples
DES, AES

one-time pad
perfect encryption
crypto = M XOR pad
M = crypto XOR pad
if pad applied twice it is simply cancelled out
pad must never be used twice, or repeated, must be real random numbers
not practical because need large amount of authenticated encryption bits

8.3 interlock protocol

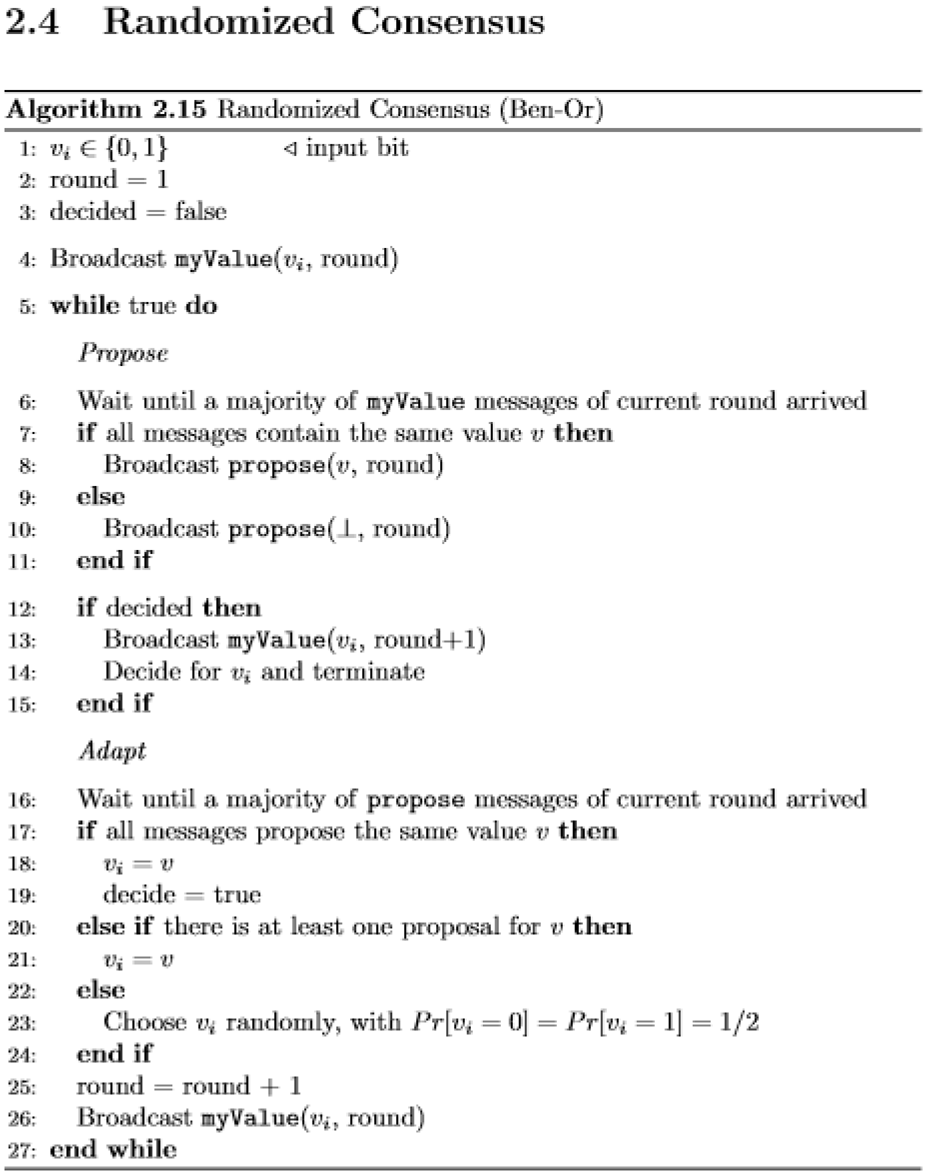
securely communicate with attacker in between
B → A sends challenge only A can answer
A → B sends encrypted answer, but only half of bits
A → B sends rest of the answer
B checks that first message is received in very short time
X can perform MitM by establishing key with B impersonating A
but X needs whole A message to do so
if X forwards first part immediately, X is not able to perform MitM
if X buffers till both messages received then B knows about intruder

example graph isomorphy

prover says he knows isomorph graphs G1 = G2
prover construct H by renaming random knots of G1 or G2
verifier then requests mapping to G1 or G2
prover can do this easily as he knows H ˜G1 and G1 ˜G2
process is repeated

algorithm dump

Mittwoch, 7. Februar 2018 18:19



2.4 Randomized Consensus

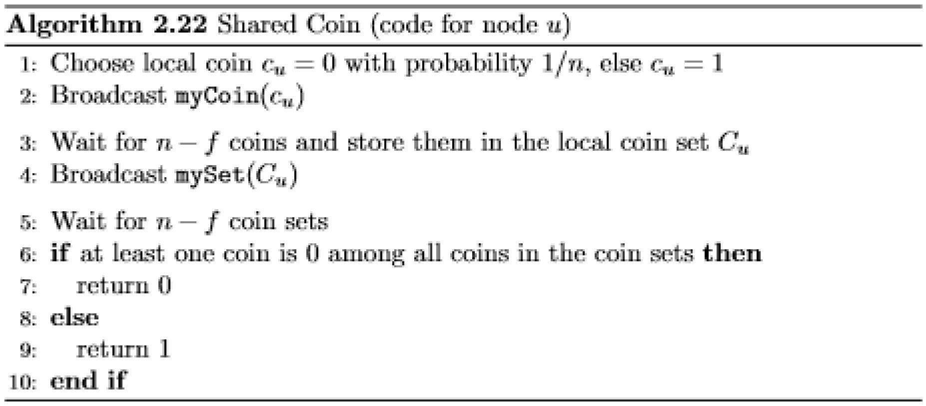
Algorithm 2.15 Randomized Consensus (Ben-Or)

```
$v_i \in \{0,1\}$          ◁ 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
        $v_i = v$
        decide = 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
```

broadcast a random value. If a majority answers with one value, propose this value => at most one value was proposed.
Then wait for a majority of propose messages. If all want the same, then we take that value and terminate after broadcasting it again for this and the next round. If some proposed *null*, then store the v but not don't terminate yet because others need our knowledge. Broadcast that value and restart.
If nobody of that majority proposed any value, choose a different random value, broadcast it and restart.

Some nodes propose, others don't because they see some disagreement within the (not complete, only the majority) set they receive.
We wait for a majority only, because that is enough. But then it is possible that other nodes still disagree (didn't get the value from a majority of broadcasts and thus propose *null*). So the other nodes will stay random until they get at least one value proposal - this must happen if we wait for a majority of proposals and there actually is a majority of value proposals. If there is a majority of value proposals, we're basically done: Everybody receives at least one and in the next round broadcasts and proposes that value. Otherwise, the setting changes randomly until a large number of nodes get the same value by chance so that the majority will propose the value.

2.5 Shared Coin

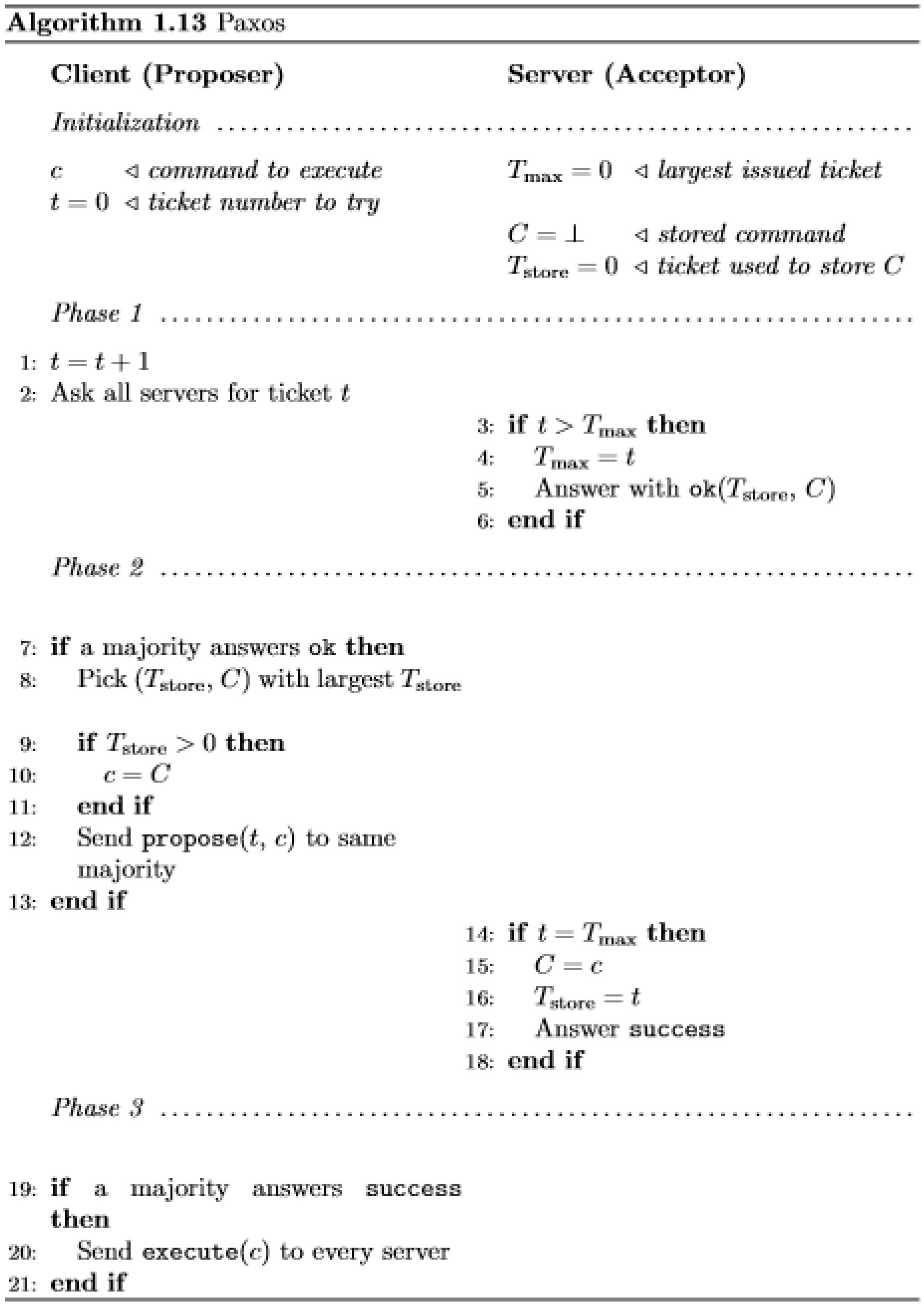


Algorithm 2.22 Shared Coin (code for node u)

```
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
```



Algorithm 1.13 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
```

Paxos does not guarantee termination. E.g. if no client ever gets a majority