Technische Universität Berlin

Distributed Algorithms 2016/17
Handout on System Models – Partial Synchrony

Odej Kao, Andreas Kliem | Complex and Distributed IT Systems

Preface

The findings presented here are mainly a summary of the following paper:

- [1]: Dwork, C., Lynch, N., & Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, *35*(2), 288-323.

Further related literature:

- [2]: Dolev, D., Dwork, C., & Stockmeyer, L. (1987). On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, *34*(1), 77-97.
- [3]: Chandra, T. D., & Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, *43*(2), 225-267.
- [4]: Larrea, M., Arévalo, S., & Fernndez, A. (1999, September). Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In*International Symposium on Distributed Computing* (pp. 34-49). Springer Berlin Heidelberg.
- [5]: George F. Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair - Distributed Systems; 5th Edition, Pearson Education (7. July 2011), ISBN13: 978-0273760597
    - General introduction to fundamental concepts

Introduction

Synchrony in Distributed System

–

Synchronous vs. Asynchronous System Model or is there something in between?

Introduction

System Models help us to:

- Understand if a design will work
- Understand and verify important properties of the system (e.g. using math models, simulation)
- Clarify understanding by abstracting unnecessary details away (e.g. hardware)

Besides the

- Physical Model (e.g. how do the nodes look like, how are they interconnected) and the
- Architectural Model (e.g. how do they communicate with each other – request reply protocol using a Client/Server approach)
- Fundamental Models exist (e.g. Failure Model, Security Model, Interaction Model)
  - Give a formal description of the properties which are shared by all architectural models
  - Interaction Models consider the structure and sequencing of the communication between the entities of the Distributed System
  - Common Interaction Models: Synchronous System Model and Asynchronous System Model

Introduction

What do we know about a Distributed Algorithm?:

- The steps to be taken by each of the processes of which the system is composed (i.e. the operations necessary in order to complete Alice and Bob‘s booking request)
- The messages that need to be transmitted between the processes to transfer information and coordinate activity

We do we not know?:

- The rate at which each process proceeds
- The timing of the transmission of the messages
- The state is difficult to describe -> each process and message transmission can fail independently

➢ Interaction Models define different constraints with respect to these properties and help us to understand the behavior of a Distributed Algorithm

Introduction

A Synchronous Distributed System defines following bounds:

- Known lower and upper bounds for each execution step of a process
- Each transmitted message is received within a known bounded time (often only upper bound is specified)
- Each local clock has a known drift rate

In a Synchronous Distributed System it is possible to use timeouts to detect failures

- It is possible to suggest upper and lower bounds, but very difficult to arrive at realistic values
- Synchronous Distributed Systems often used for modeling and simulation purposes
- Although difficult to realize in real world, synchronous Distributed Systems can be built (e.g. real time applications)

Introduction

Asynchronous Distributed Systems make no assumptions regarding bounds on:

- Process execution speed
- Message transmission delays
- Clock drift rates

Most Distributed Systems are asynchronous because processors and communication channels are shared:

- Being aware of the asynchronous nature of a Distributed System and the resulting implications, helps us to design our algorithms appropriately

Introduction – Atom(ic) Model

The Atom Model as the default model for the lecture is a specialization of the asynchronous model

- Attention: It is partly synchronized, but not partially synchronous!! -> will be explained later
  - According to the possible bounds, we can define the synchronicity of a model in terms of
  - Processors are synchronous or asynchronous
  - Communication is synchronous or asynchronous

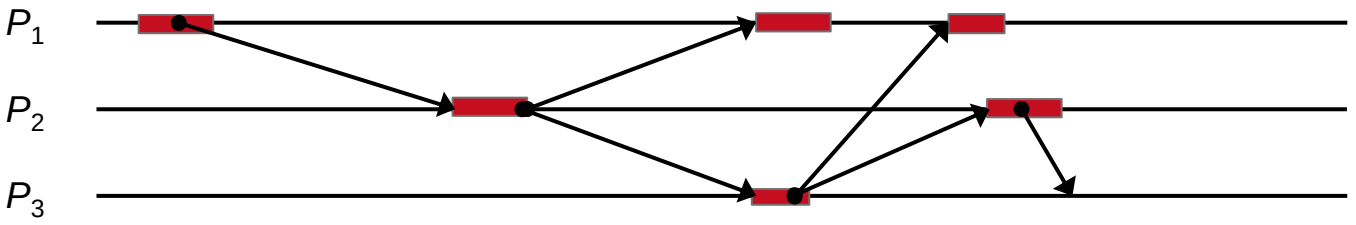
In case of the Atom Model, actions take no time

- As a result, a processor cannot execute faster or slower than any other processor
  - Processors are synchronous
  - Communication remains asynchronous
- Since Communication is asynchronous, the model still behaves like a asynchronous one. So what is the advantage?

Source: Mattern, F. (1987). Algorithms for distributed termination detection. *Distributed computing*, *2*(3), 161-175.

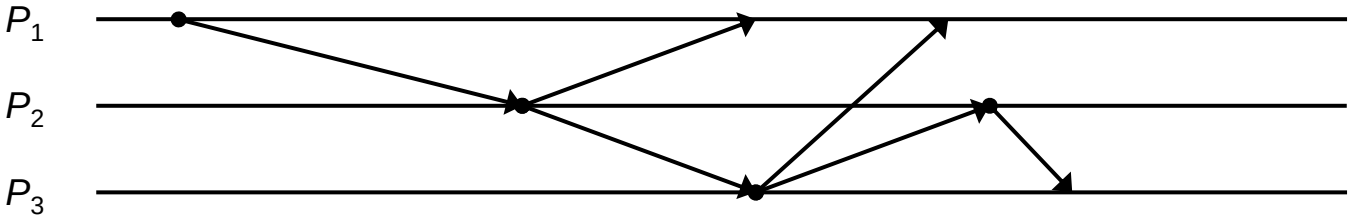
Introduction – Atom(ic) Model

The Atom Model turns this …



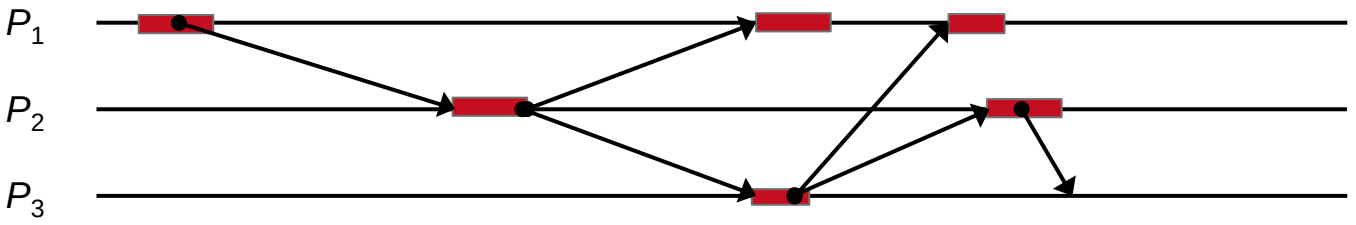
Into this …

- Actions are timeless
- We only need to count messages when we want to detect termination of a distributed algorithm



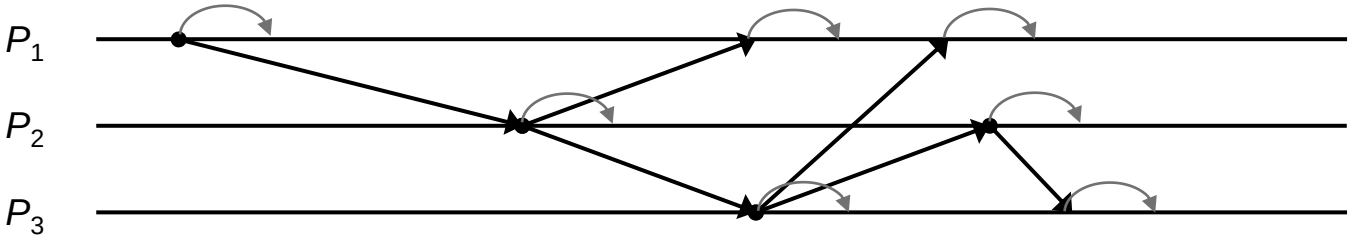
Introduction – Atom(ic) Model

But can we simply consider the processing time as inconsequential for termination ..



No, but we can do this ..

- Each process sends a “virtual” message to itself that takes as much time as it would be active



Introduction

In general …

In order to understand the implications of each particular Interaction Model, we can have a look at a common problem in Distributed Systems and investigate the impact the different constraints defined by the models have on the behavior and the results of an algorithm/protocol used to solve the problem.

- Since the atomic Model “only” affects termination and generally behaves like the asynchronous one, we will not further consider it here

The Problem of Reaching Agreement

The Problem of Reaching Agreement

We know protocols that achieve agreement for specific use cases:

- Processes involved in transferring money from one account to another must consistently agree to perform corresponding debit and credit operations
- In mutual exclusion, the processes agree on a process that can enter a critical section
- Consider an election, the processes agree on which is the elected process

The Problem of Reaching Agreement

Consensus Protocols attempt to solve the Agreement Problem in a general fashion

Consider following version of the problem:

- A collection of N processors $p_1, \ldots, p_N$, which communicate by sending messages
- Initially each processor p_i has a value v_i drawn from the domain V (e.g. V ={true, false})
- All correct processors must agree on the same value (i.e. each processor decides)
- The protocol should operate, if some processors are faulty
  - We say a consensus protocol is t-resilient, if it can tolerate t faulty processors
  - We can give the minimum number of processors required, such that a protocol can tolerate t faulty processors given a particular class of faults
  - E.g. the popular Byzantine Generals problem: Consensus is possible if $N \geq 3t + 1$ given an synchronous system model

The Problem of Reaching Agreement

Consensus Protocols attempt to solve the Agreement Problem in a general fashion

More formal definition of the problem:

- Following conditions should hold for every execution of a Consensus Algorithm:
  - Termination: Eventually each correct process sets its decision variable.
  - Agreement: If p_i and p_j are correct and have entered the decided state, then $v_i = v_j$ (i,j=1,2,…, N) (all correct processes have the same decision value)
  - Integrity (validity): If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

The Problem of Reaching Agreement

Smallest number of processors N for which a t-resilient Consensus Protocol exists

Failure Type	Synchronous	Asynchronous	Partially synchronous communication and synchronous processors
Fail-Stop	t	∞	2t+1
Omission	t	∞	2t+1
Authenticated Byzantine	t	∞	2t+1
Byzantine	3t+1	∞	3t+1

- Seems as if we are in serious trouble given the Asynchronous Interaction Model

The Problem of Reaching Agreement

Failure types

Failure Type	
Fail-Stop	processor p_i executes correctly, but can stop at any time. It cannot restart.
Omission	processor p_i follows its protocol correctly, but a send operation does not place m in buffer of recipient of a receive operation does not take all delivered messages from the buffer
Authenticated Byzantine	Arbitrary behavior, but messages can be signed, signature is unforgeable
Byzantine	Arbitrary behavior, worst possible failure semantic

The Problem of Reaching Agreement

Consensus in Asynchronous Systems?

Fischer et al. proved that no algorithm can guarantee to reach Consensus in asynchronous systems, even with just one process Fail-stop failure

- Sometimes referred to as FLP impossibility (Fischer, Lynch, Paterson)
- In asynchronous systems, we cannot distinguish between a crash and a slow process

- But how do our real world system work?
  - E.g. transaction based systems should be able to reach consensus (e.g. banking, flight booking, databases), otherwise we would be in serious trouble

The Problem of Reaching Agreement

Consensus in Asynchronous Systems?

Techniques for working around the impossibility result in asynchronous systems exist:

- For instance masking failures:
  - Mask any process failure that occurs
  - E.g. transactions use persistent storage to log the progress of a set of related operations
  - After a crash, the process can be restarted and recover to a consistent state using its persistent log
  - It will behave like a correct process, that sometimes takes a long time to perform an action

The Problem of Reaching Agreement

Consensus in Asynchronous Systems?

Techniques for working around the impossibility result in asynchronous systems exist:

- For instance using Failure Detectors:
  - Failure detectors can use further knowledge than just message passing behavior
  - If the detector is accurate, then processes can agree to deem processes that have not responded within a certain timeout (perfect by design failure detector)
    - Even if the process was just slow -> all other agree that they will ignore further messages
    - The asynchronous system is turned into a synchronous one
  - It has also been shown, that given certain conditions, Consensus can be solved even with unreliable failure detectors
  - Further reading on Failure Detectors:
    - For an introduction see [5] and „Verteilte Systeme“ Lecture Slides
    - Failure detectors are discussed in more detail in: [3][4]

The Problem of Reaching Agreement

Consensus in Asynchronous Systems?

Clearly, our real world systems do fit the asynchronous model better. However, are the assumptions of that model to weak?

- Is it practically feasible to assume that our network will more likely fail in delivering messages than succeed?
- Can we define something in-between the synchronous and asynchronous approach, that helps us to solve the Agreement Problem even if some processors are faulty?

Partial Synchrony

Partial Synchrony

Reminder: Synchronous Systems assume that

- Communication is synchronous: an upper bound for message delivery exists and is known
- Processors are synchronous: an upper bound on the rate at which one processors clock can run faster than another's exists and is known
- The existence of both bounds is necessary to achieve any resiliency, even under the weakest type of failures -> FLP impossibility: if communication is asynchronous or processors are asynchronous no consensus protocol resilient to even one fail/stop fault exists

Partial Synchrony

Given the separation of the synchrony property in terms of communication and processors, three types exist:

- Both, communication and processors are partial synchronous
- Communication is synchronous and processors are partial synchronous
- Processors are synchronous and communication is partial synchronous (i.e. partially synchronous communication)
    - This type will be considered here, for all other types refer to the literature (TODO… DWORK)

Partial Synchrony

Two cases of partially synchronous communication can be considered:

1. The upper bound on message delivery time exists, but is not know
2. We know the upper bound, but it holds only for a certain amount of time after a stabilization interval

Partial Synchrony

Two cases of partially synchronous communication can be considered:

1. The upper bound on message delivery time exists, but is not know
    - Impossibility results (FLP) do not apply -> communication is actually synchronous, we just do not know the upper bound
    - Processors in known Consensus Protocols need to know upper bound in order to know how long to wait in each round of message exchange
    - Pick upper bound arbitrary? -> not acceptable, since if bound too small, all processors could be considered as faulty
    - Further, we cannot assume that a probability distribution exists that allows us to estimate the bound
    - -> we need to design a protocol that operates without having the upper bound built in

Partial Synchrony

Two cases of partially synchronous communication can be considered:

1. The upper bound on message delivery time exists, but is not know
2. We know the upper bound, but it holds only for a certain amount of time after a stabilization interval
    - The message systems is sometimes unreliable (messages delivered late or not at all)
    - Problem: this unreliable system is at least as bad as an asynchronous one -> impossibility results
    - We must add a constraint: for each execution of the algorithm, the is a Global Stabilization Time (GST) – GST is unknown to the processors
    - After GST, the upper bound on message delivery holds for an amount L of time
        - It would be unrealistic that the system remains stable for ever

Partial Synchrony

Partially synchronous communication can be envisioned as a game between a protocol designer and an adversary:

- If the upper bound is known, the adversary names an integer and the protocol designer must provide an algorithm that is correct if the upper bound always holds
- If the upper bound is unknown, the protocol designer provides the algorithm, then the adversary names an upper bound and the algorithm must be correct if the bound always holds
- If the upper bound holds eventually, the adversary picks the upper bound, the protocol designer provides an algorithm knowing the bound, and the adversary picks a time T when the bound starts holding

➢ The following basic algorithm can be adapted to work correct given case 2 and 3 (partial synchronous communication)

- First, a Basic Round Model of Computation is defined
- The model is used to execute the algorithm/protocol and understand the behavior

Basic Round Model of Computation

Basic Round Model of Computation

Divide processing into synchronous rounds of message exchange

Each round consists of:

- Send subround
  - Each processor sends messages to any subset of processors
  - Send(m,p_i): put message into buffer corresponding to p_i
- Receive subround
  - Some subset of the messages sent to the processor during corresponding Send subround is delivered
  - Receive(p_i): removes some set S (possibly empty) of messages from p_i's buffer and delivers the messages to p_i
- Computation subround
  - Each processor executes a state transition based on the set of messages just received
  - The initial state of the processor p_i is determined by its initial value v_i in V
  - At some point in computation, a processor can irreversibly decide on a value in V

Basic Round Model of Computation

Not all messages that are sent need to arrive -> some can be lost

It is assumed that a round GST (*Global Stabilization Time*) exists

- All messages sent from correct processors to correct processors at round GST or afterwards are delivered in the round at which they were sent

- Loss of messages before GST does not necessarily make the sender or receiver faulty!

- All processors have a common numbering for the rounds, but do not know when round GST occurs

Basic Round Model of Computation

The following algorithm solves the Consensus Problem given the Basic Round Model

- We assume Fail-Stop and Omission Faults
- A t-resilient Consensus Protocol (our algorithm) exists for $N \geq 2t + 1$ processors
  - Again: t-resilient means that the algorithm is resilient to t faulty processors
- To achieve integrity condition of the Consensus Problem , a Variable called PROPER is introduced
  - PROPER contains a set of values that a processor knows to be proper
    - PROPER is always piggybacked on each message a processor sends
    - If all processors start with the same v, then v is the only proper value
    - If there are at least two different initial values, all values in V are proper
- Network is completely connected
  - Any processor can send to any (including itself)

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Remarks on PROPER:

- Each processor‘s PROPER set initally contains its own initial value v
- Remember, PROPER is piggybacked on each message
- If a processor receives a PROPER set that contains a value v not in its owe one, v is added

Remarks on rounds:

- Rounds are organized into alternating *trying* and *lock-release* phases
- The trying phase contains three rounds, the lock-release-phase contains one round
- A trying phase is always followed by a lock-release phase, both constitute a corresponding pair belonging to a particular processor
- Each pair is assigned an integer h and belongs to the processor h mod N
    - E.g. in case of 3 processors 1->1, 2->2, 3->3, 4->1, …

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Remarks on locks:

- At various times during the protocol/algorithm, a processor may lock a particular value v
- Each lock is associated with a phase number (the phase in which it has been created)
- For instance, if processor 2 creates a lock (true, 4), it thinks that processor 1 might decide on true at phase 4
- Processor 2 might later release the lock, if it learns that its assumption was wrong (i.e. processor 1 has not decided on true)
- Values in the PROPER set can be acceptable or not (i.e. considered as a possible agreement)
  - A value v is acceptable for a processor p, if p does not have a lock on any value except possibly v (i.e. if no value is locked or only v is locked)
  - We will see that multiple locks are possible in our model before GST -> due to lost messages

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

The following examples will now demonstrate the Consensus Protocol/Algorithm (we will use protocol from now on)

We assume:

- 3 processors -> N=3
- As a result the protocol is resilient to one faulty processor (t = 1)
  - However, for the first run we will assume all processors are correct
- Our value domain V is: V = {true, false}
- Four message types exist:
  - Announce acceptable values, e.g.: m = {acc={true, false}, 1, PROPER={true, false}}
    - A processor sends its list of acceptable values in phase 1 (remember, each message contains PROPER)
  - Lock a value, e.g.: m = {true, 1, PROPER={true, false}}
    - Processor 1 (because of phase ID) broadcasts a lock request for true
  - Acknowledge lock request in certain phase, e.g.: m = {ack, 1, PROPER={true, false}}
  - Release lock, e.g.: m = {locks={(true, 1)}, PROPER={true, false}}
    - A processor broadcasts its locks

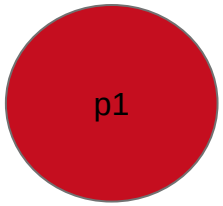
Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

First run with 3 of 3 processors being correct

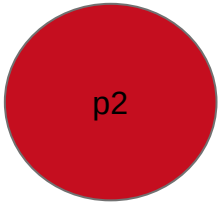
Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: initial state

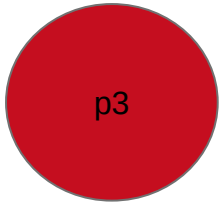
Round: --



v1 = true
PROPER = {true}
Acceptable = {true}



v2 = true
PROPER = {true}
Acceptable = {true}



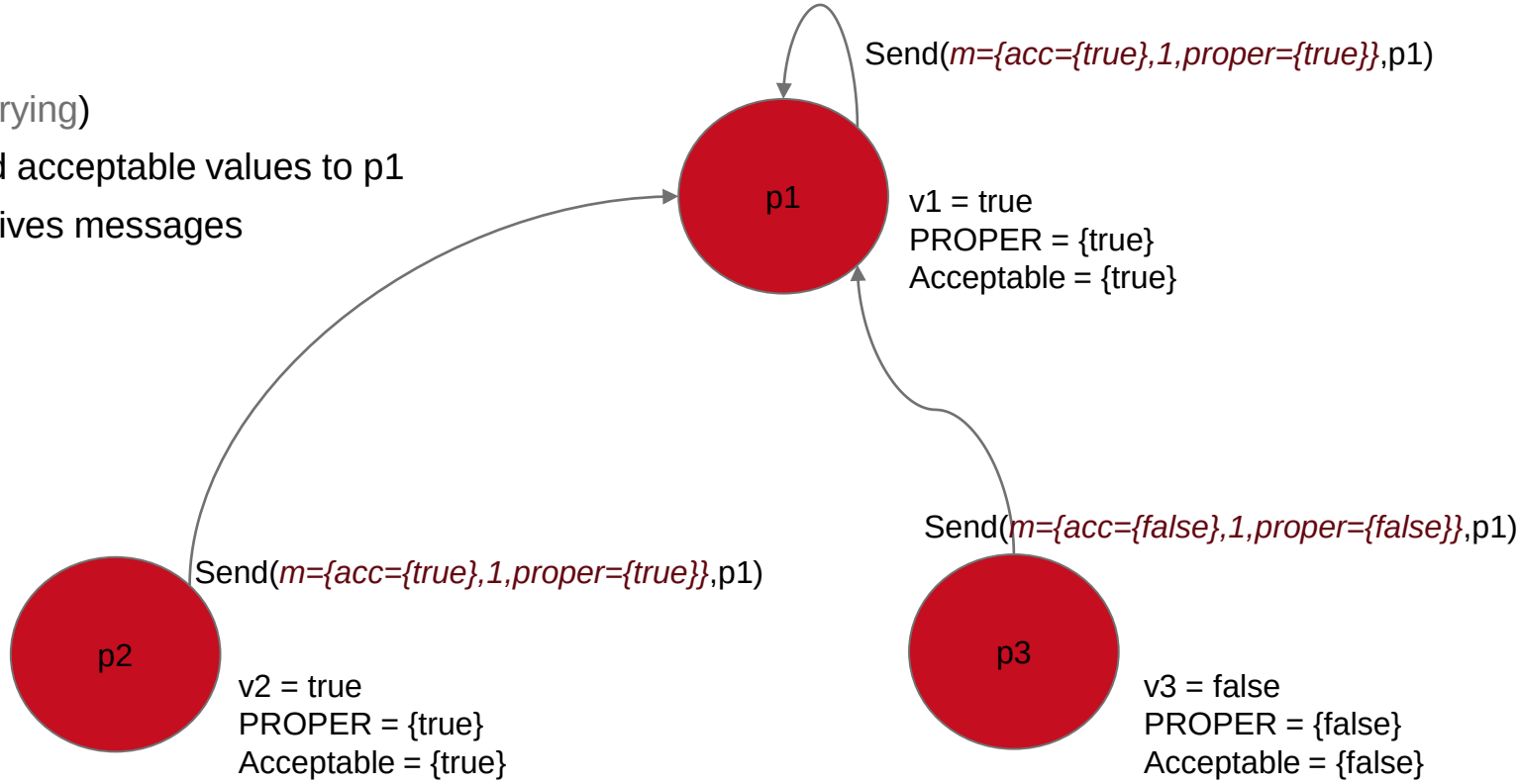
v3 = false
PROPER = {false}
Acceptable = {true}

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 1

Round: 1 (trying)

- All send acceptable values to p1
- p1 receives messages

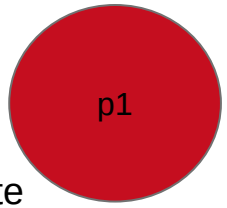


Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

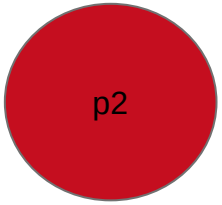
Phase: 1

Round: 1 (trying)

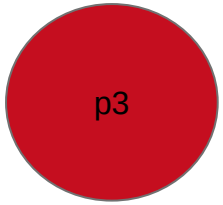
- p1 attempts to choose value to propose
- two of 3 (N) find true acceptable and proper
- Remember: 1 round contains send, recv, compute



v1 = true
PROPER = {true, false}
Acceptable = {true, false}



v2 = true
PROPER = {true}
Acceptable = {true}



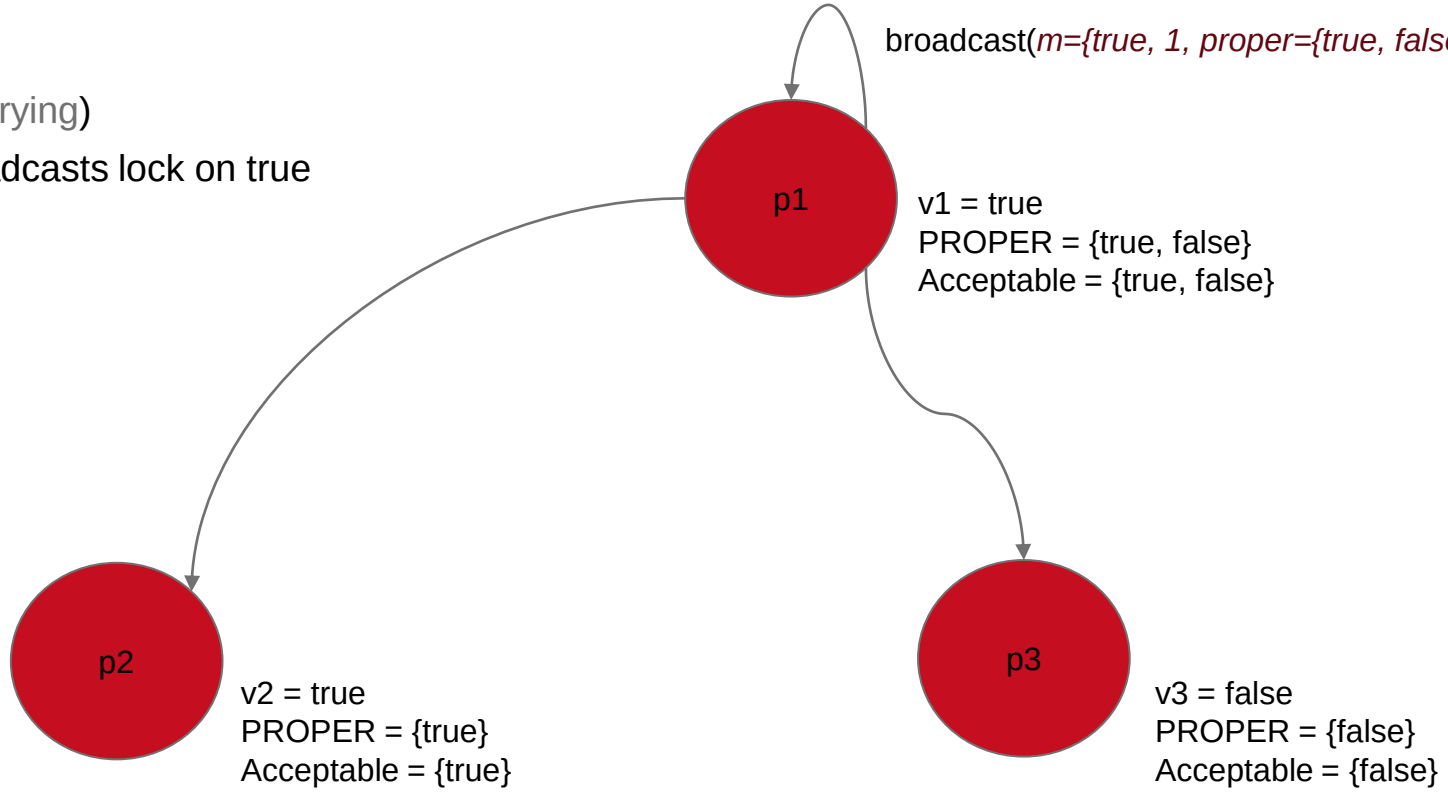
v3 = false
PROPER = {false}
Acceptable = {false}

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 1

Round: 2 (trying)

- p1 broadcasts lock on true

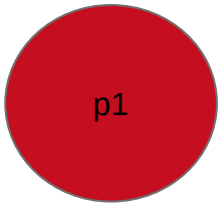


Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

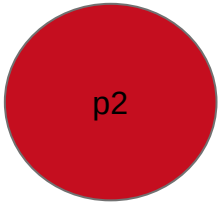
Phase: 1

Round: 2 (trying)

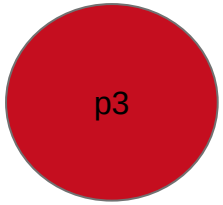
- All lock true (associated with phase 1)
- Any possibly existing earlier lock on true is released, true is immediately locked again



v1 = true
PROPER = {true, false}
Acceptable = {true}
Locks = {(true,1)}



v2 = true
PROPER = {true, false}
Acceptable = {true}
Locks = {(true,1)}



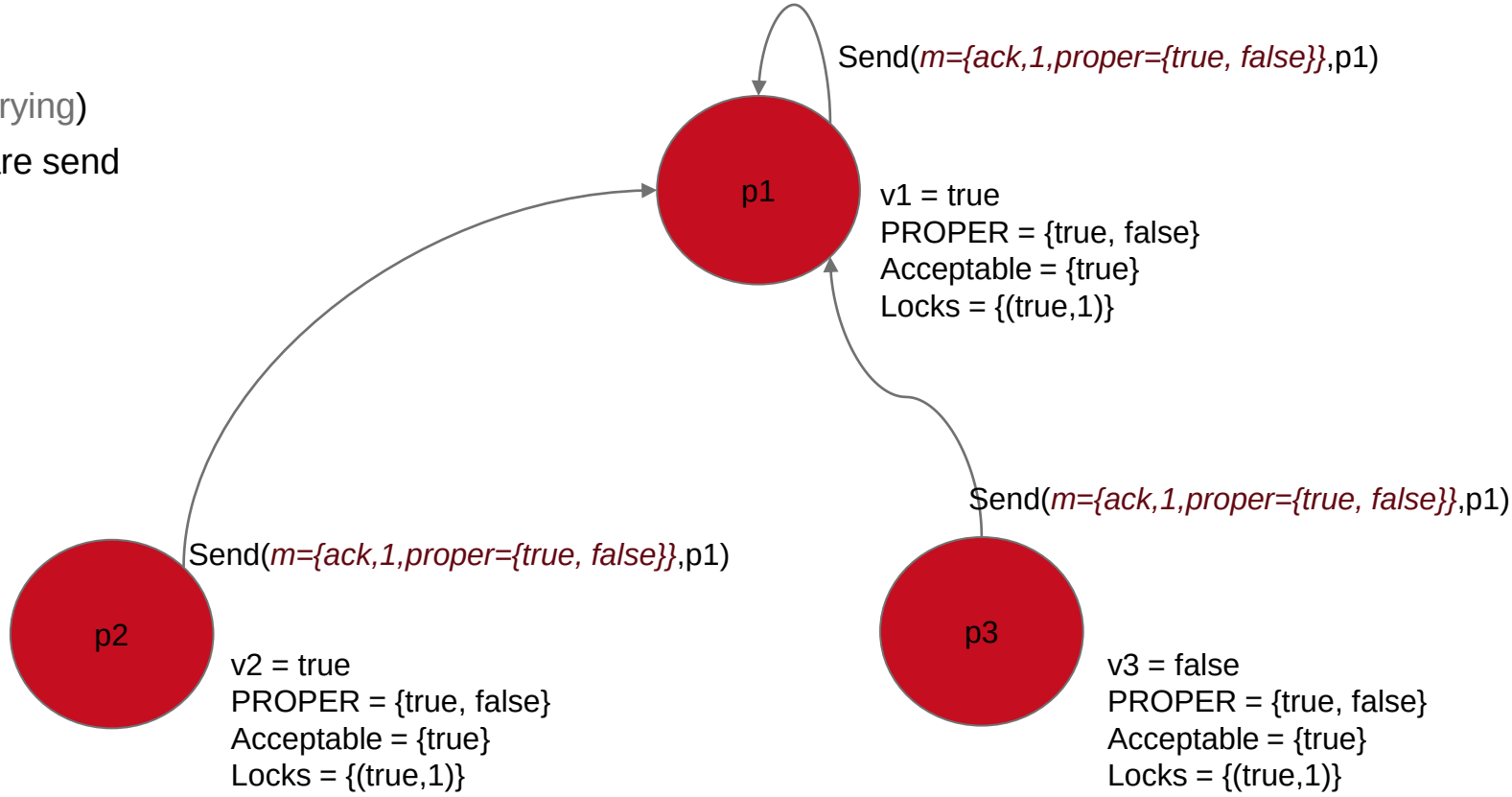
v3 = false
PROPER = {true, false}
Acceptable = {true}
Locks = {(true,1)}

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 1

Round: 3 (trying)

- ACKs are send

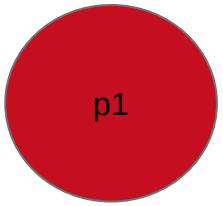


Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

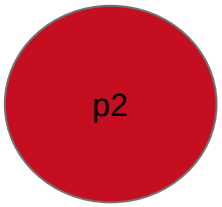
Phase: 1

Round: 3 (trying)

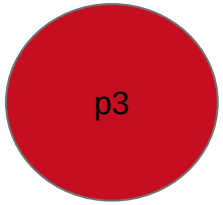
- p1 received more than t + 1 ACKs
- p1 decides true
- p1 keeps participating



v1 = true
PROPER = {true, false}
Acceptable = {true}
Locks = {(true,1)}
Decided -> true



v2 = true
PROPER = {true, false}
Acceptable = {true}
Locks = {(true,1)}



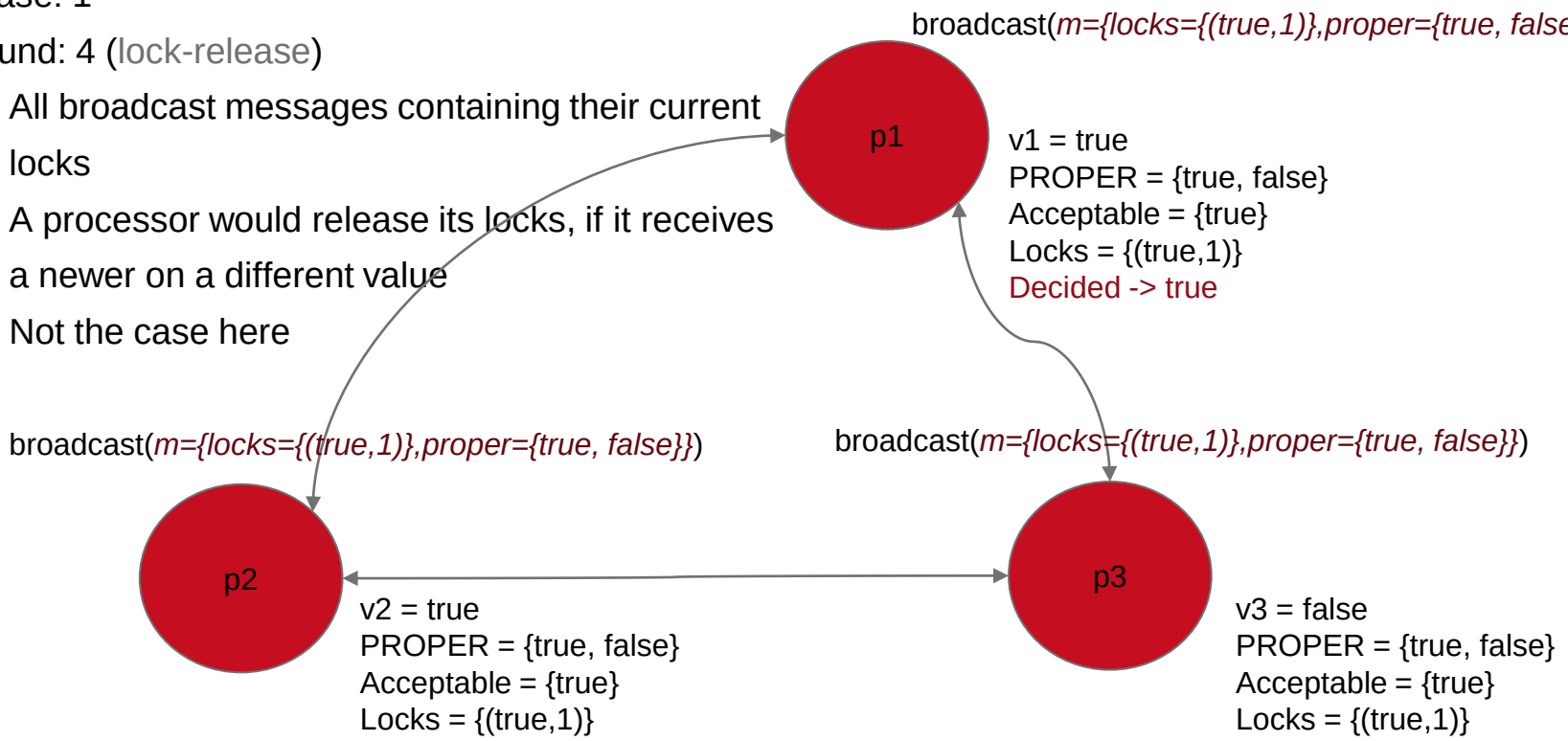
v3 = false
PROPER = {true, false}
Acceptable = {true}
Locks = {(true,1)}

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 1

Round: 4 (lock-release)

- All broadcast messages containing their current locks
- A processor would release its locks, if it receives a newer on a different value
- Not the case here



Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 2

Round: 5-7 (trying)

- All report acceptable to p2 -> all report true
- p2 sees more than N-t proposals for true and thus decides to send a lock request for true in phase 2
- All update their locks and acknowledge
- p2 receives more than t+1 ACKs and thus decides true

Round: 8 (lock-release)

- All broadcast their locks, since only true is locked, no locks are released

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 3

Round: 9-11 (trying)

- All report acceptable to p3 -> all report true
- p3 sees more than N-t proposals for true and thus decides to send a lock request for true in phase 3
- All update their locks and acknowledge
- p3 receives more than t+1 ACKs and thus decides true

Round: 12 (lock-release)

- All broadcast their locks, since only true is locked, no locks are released

All processors made a decision!

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Second run with 2 of 3 processors being correct

- p2 exhibits omission faults -> before GST, we do not know whether p2 is faulty or messaging is just unreliable

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 1

Round: 1-3 (trying)

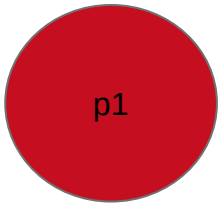
- All except p2 report acceptable to p1 -> p1 report true, p3 reports false
- p1 cannot make a decision on a value to propose -> there is no value that N-t = 2 processors have found to be acceptable
- No lock request will be broadcast and thus no ACKs are received, p1 cannot decide

Round: 4 (lock-release)

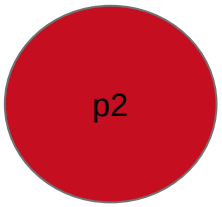
- No locks exist, so no lock-release broadcasts

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

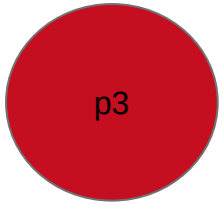
State after round 4 (of phase 1):



v1 = true
PROPER = {true, false}
Acceptable = {false}



v2 = true
PROPER = {true}
Acceptable = {true}



v3 = false
PROPER = {true, false}
Acceptable = {false}

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 2 -> we skip this, since p2 currently suffers from omission faults -> no changes

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 3

Round: 9-11 (trying)

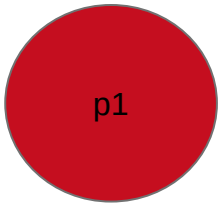
- All except p2 report acceptable to p3 -> p1 report true and false, p3 reports false
- p3 can make a decision on a value to propose -> 2 processors have found false to be acceptable
- p3 broadcasts lock-request, p1 and p3 reply, which is equal to t+1
    - p3 decides on false

Round: 12 (lock-release)

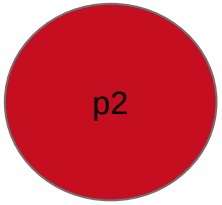
- No other locks than (false, 3) exist -> no locks are released

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

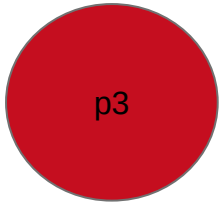
State after round 12 (of phase 3):



v1 = true
PROPER = {true, false}
Acceptable = {false}
Locks = {(false,3)}



v2 = true
PROPER = {true}
Acceptable = {true}



v3 = false
PROPER = {false}
Acceptable = {false}
Locks = {(false,3)}
Decided -> false

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Phase: 4

Round: 13-15 (trying)

- All except p2 report acceptable to p1 -> p1 reports false, p3 reports false
- p1 can make a decision on a value to propose -> 2 processors have found false to be acceptable
- p1 broadcasts lock-request, p1 and p3 reply, which is equal to t+1
  - Locks are updated
  - p1 decides on false

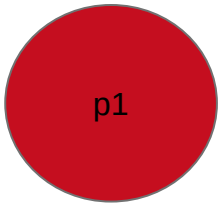
Round: 16 (lock-release)

- No other locks than (false, 4) exist -> no locks are released

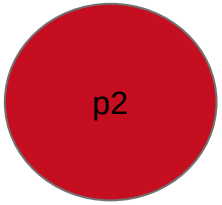
Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

State after round 16 (of phase 4):

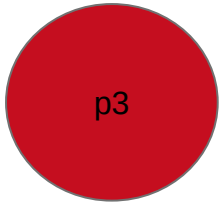
- Note that p2 still isn‘t considered as faulty, because our model allows messages to be late or lost before GST
- However, eventually GST will hold:
  - p2 either behaves correct and decides for false in its phase as well
  - or is recognized as faulty
  - ->termination of the protocol



v1 = true
PROPER = {true, false}
Acceptable = {false}
Locks = {(false,4)}
Decided -> false



v2 = true
PROPER = {true}
Acceptable = {true}



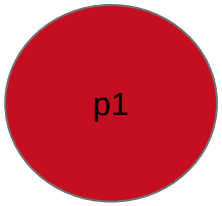
v3 = false
PROPER = {false}
Acceptable = {false}
Locks = {(false,4)}
Decided -> false

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

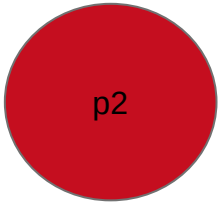
Homework: what happens if not all messages of p2 are lost before GST -> protocol still working?

For example:

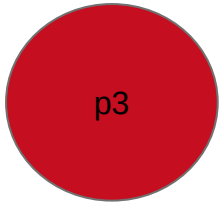
- P2 reports in phase one, but its ACK is lost
- …



v1 = true
?



v2 = true
?



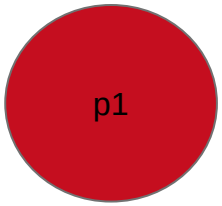
v3 = false
?

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Homework: what happens if not all messages of p2 are lost before GST -> protocol still working?

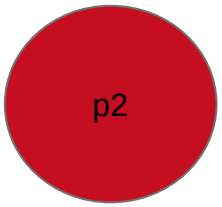
For example:

- P2 reports in phase one, but its ACK is lost
- …

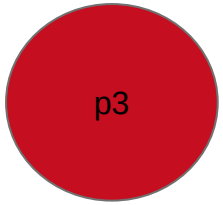


v1 = true
?

[1] gives shows in detail that the algorithm achieves termination, consistency and integrity (also known as strong unanimity)



v2 = true
?



v3 = false
?

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Summary of proof

Consistency:

- It is impossible that two distinct values are locked during the same phase, otherwise the corresponding processor must send conflicting lock-request (note that we do not consider Byzantine faults)
- If a processor decides on v at a phase k
  - At least t+1 processors have locked v
  - each of the locking processors will have a lock on v with at least k as the associated phase number for the rest of the protocol
  - Otherwise, some of them would have to find another value acceptable, which is impossible (note that N-t must find a value acceptable in order for a processor to send a lock-request for it, but this is impossible since already t+1 have another value locked!)
  - As a result, after the first decision was made, no other value can become acceptable

Integrity:

- If all have v as the initial value, only v is in all PROPER sets -> only v can ever become acceptable

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Summary of proof

Termination:

- If round GST has appeared at a trying phase or before, the set of locked values of all correct processors is either empty or contains one locked value -> as a result of the lock-release rule (only the lock with the highest phase number survives)
- Immediately after this lock-release phase, the next processor pi will reach a decision in its trying phase k
  - Due to communication in the previous lock-release phase, all PROPER sets contain all intial values
  - Only one of them can be locked (if at all)
  - since we assume that N -1 >= t +1 correct processors exist and we have passed GST, a proper and acceptable value will be found to decide on
  - All other processors will decide in the following rounds as well

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Finally, we need to show that we can simulate the Basic Round Model in the partial synchronous one

(i.e. given partial synchronous communication and synchronous processors)

- As a result, we can show that for each protocol in the Basic Model, a protocol in the partial synchronous model can be derived -> if we can solve Consensus in the Basic Round Model, we can solve it in the partial synchronous one

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

First, the case where the upper bound holds eventually is considered

- Assume that we derive a protocol A‘ from A, where A works for the Basic Round Model
- We need to show that we can simulate A in the partial synchronous model
  - A‘ is an simulation of A in the partial synchronous model

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

First, the case where the upper bound holds eventually is considered

- One single round of A is mapped to *R* = *N* + *upper bound* steps in A‘
  - Remember, that one single round in the Basic Model consists of a send-subround, a receive-subround, and a computation-subround
  - Thus, A‘ uses N steps to send messages to N processors (the others + itself)
  - Then, A‘ uses upper bound steps to receive messages (if we assume that the upper bound holds after GST, all replies are received)
  - Finally, the state transition (computation-subround) is simulated
  - Each processor always attaches a round identifier (identifying R), so that late messages arriving during another round are ignored -> independent communication during each round

  - The only difference to the Basic Round Model is that GST must not hold forever (and that we do not assume that an atomic broadcast exists)
    - Since we simluated A and mapped the rounds to N + upper bound, we can now specify the L (i.e. the time the uppder bound ust hold after GST in order for the protocol to terminate) -> see [1] for details

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

First, the case where the upper bound holds eventually is considered

- One single round of A is mapped to *R* = *N* + *upper bound* steps in A‘
  - Since we simulated A, it can be shown, that for any run e‘ of A‘, a corresponding run e of A with following properties exists:
    - All processors that are correct in e‘, are also correct in e
    - The fault types are the same in e‘ as in e
    - Every state transition of a correct processor in e is simulated in e‘ (by the corresponding correct processor)
    - ➢ Since Consensus is reached in e, it is reached in e‘ as well

Consensus with Fail-stop and Omission Faults (N >= 2t + 1)

Second, the case where the upper bound holds but is unknown is considered

- Again, assume that we derive a protocol A‘ from A, where A works for the Basic Round Model
- Again, we need to show hat A‘ can simulate A
- This time, we try to converge to the upper bound step by step
- One single round of A is mapped to $R_r = N + r$ in A‘, where r increase by 1 for each R_r
  - Again, we simulate send operations in N, receive in r and the state transition as the last step of R_r
  - If we assume that the upper bound holds for any run e‘ of A‘, we can define a corresponding run e of A (the simulation)
  - The number of steps taken for receive is sufficient for any round where r >= upper bound
  - As a result, e is an allowable run of A with upper bound as the GST round

Thanks for listening/reading

More details as well as algorithms for byzantine faults can be found in [1]