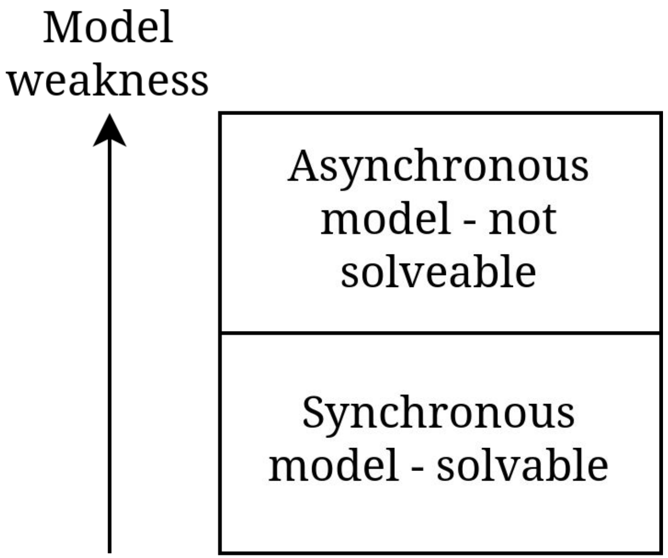
Distributed clock protocols for partially synchronous models

Consensus problem in different models

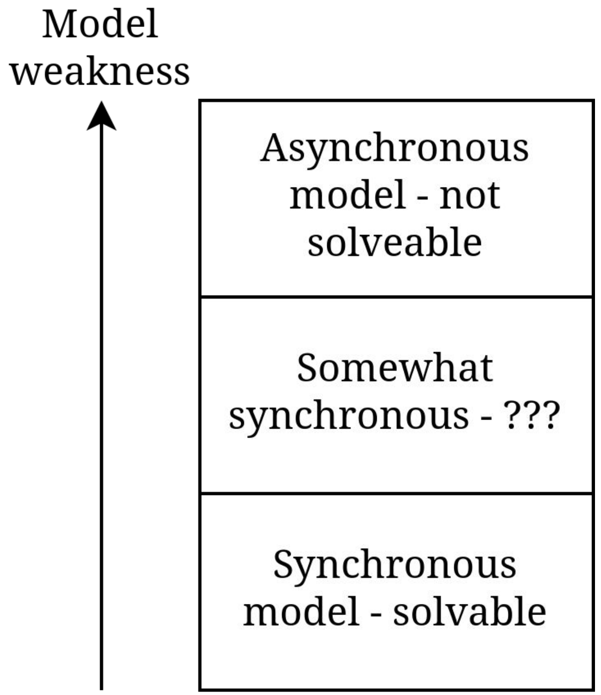
- FLP result states that we cannot solve consensus in a model with asynchronous message passing communication and crash failures even when there is only one possible failure.
- However, we can solve consensus in synchronous message passing model
    - O(f) rounds and n > 2f for authenticated Byzantine failures
    - O(f) rounds and n > 3f for Byzantine failures


Model weakness
Asynchronous model - not solveable
Synchronous model - solvable

Problem: solving consensus “in-between models”

Therefore, we would like to look at some model that is stronger than the asynchronous one, but at the same time weaker than the synchronous one.

- Can we solve consensus in the resulting model?
- What are the lower bounds for the resiliency of the resulting protocols?


Model weakness
Asynchronous model - not solveable
Somewhat synchronous - ???
Synchronous model - solvable

Partially synchronous models

- Those models are called **partially synchronous models**
- They are usually characterized by two parameters: an upper bound Δ on time to communicate a message and a lower bound Φ on processor speed.

Upper bound Δ on time to communicate a message

Here and for subsequent definitions, it is useful to imagine a real-time clock that works outside of the system and measures time in discrete integer-numbered steps.

We define the **upper bound Δ to communicate a message** in a partially synchronous model as the maximum amount of time until a message appears in the recipients' buffer after it was sent.

Suppose all processors run at the same speed.

- If $\Delta = 1$, is known to all processors and does not change throughout the execution, then this is the synchronous model
- In case $\Delta \neq 1$, is known to all processors and does not change throughout the execution, this model is equivalent to the synchronous one in terms of computational power

Lower bound Φ for processor speed

Another important characteristic of partially synchronous models is **the lower bound Φ for the processor speed**. This is a positive integer that denotes how slow the processor might run when compared to the real time.

Quick example: if Φ = 3, it means that no processor runs more than 3x slower than real time (it has to perform at least one operation per 3 real-time ticks).

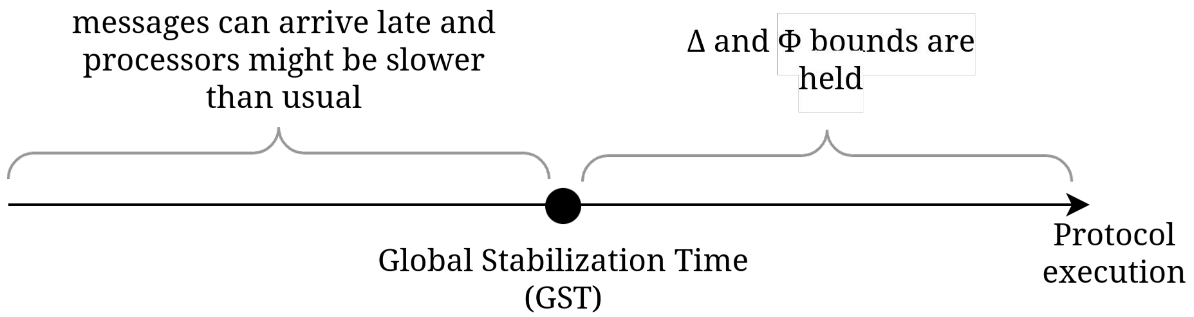
Note that Φ only sets the lowest processor speed possible. Processors might work faster than that. The upper bound for the processor speed is one operation per one tick of real-time.

Two partially synchronous models

There are two partially synchronous models we are going to discuss:

- Model with unknown communication time upper bound Δ and processors' speed lower bound Φ (Δ and Φ hold throughout the execution of the algorithm, but are unknown to the processors)

- Model where Δ and Φ are known, but they hold only after some specific point in time called Global Stabilization Time (GST). Processors do not know when GST will happen and when it has happened.


messages can arrive late and processors might be slower than usual
Δ and Φ bounds are held
Global Stabilization Time (GST)
Protocol execution

Problem

- Algorithms for solving consensus exist in partially synchronous models where communication is partially synchronous (Δ is either unknown or holds after GST) , but the processors run at the same speed ($\Phi = 1$).
- What if **both communication and the processor speeds** are partially synchronous?
- Can we still solve consensus in such a model?

More details about the models

- Message passing communication model
- The communication network is fully connected
- An atomic step of a processor p_i is either:
    - receive an unlimited number of messages from its buffer
    - send a message to a single processor p_j .
- We model each processor as a state machine.

Failure and failure tolerance

- We will consider protocols that solve consensus for Byzantine failures.
- We also assume that $N \geq 3f + 1$.

Solution: approximately common notion of time

- The algorithms that solve consensus in a model where $\Phi = 1$ have a common notion of time
- Our approach is to introduce an **approximately** common notion in the model where both Δ and Φ are unknown or hold after GST
- We will call the protocol that “synchronizes” times between processors **a distributed clock protocol**
- This protocol does not have explicit knowledge about Δ and Φ, so it can be used in both models that we have just described

Desired (informal) properties of the distr. clock protocol

We would like to obtain a protocol that has the following properties:

- Difference between local times of correct processors is bounded
- Difference between the local time of correct processors and real-time is bounded
- Each correct processor gets to perform all the operations it needs to solve consensus

High-level idea: clocks

Each processor is going to have its own software clock.

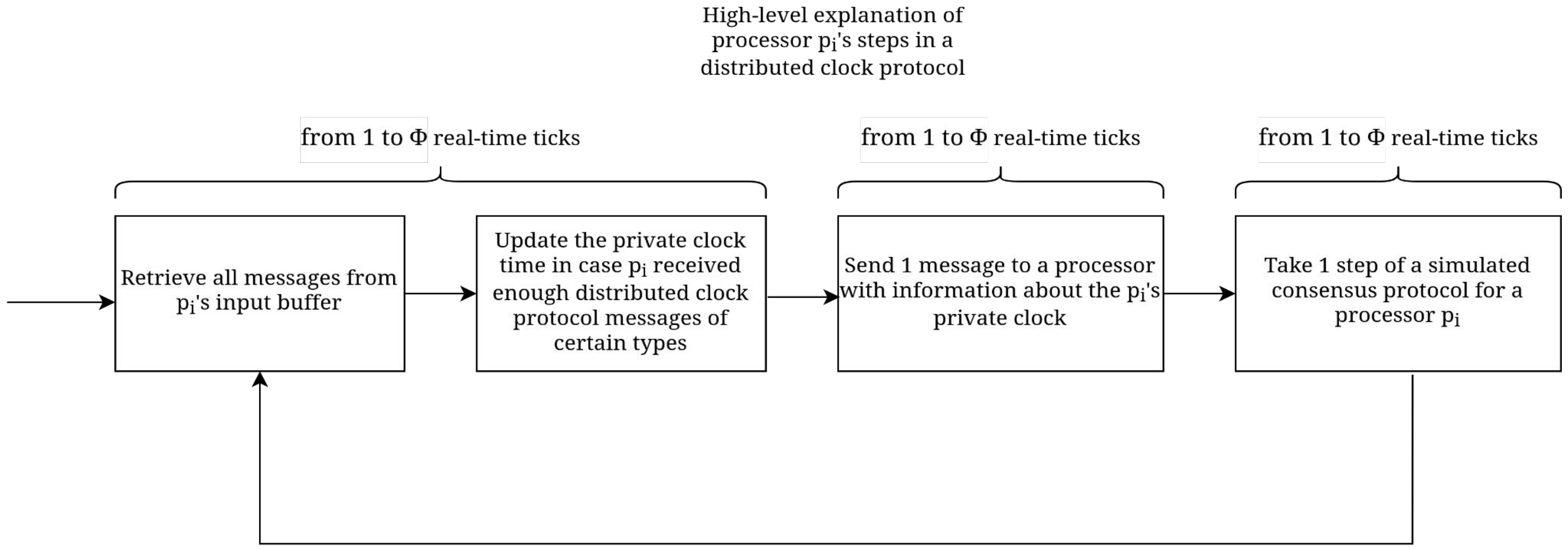
The protocols that solve consensus for $\Phi = 1$ are simulated by letting each processor use its private clock to determine the round in which it is in.

The processors will also exchange information about their local time (i.e., their private clock time), which will help to synchronize the time between all processors in the system.

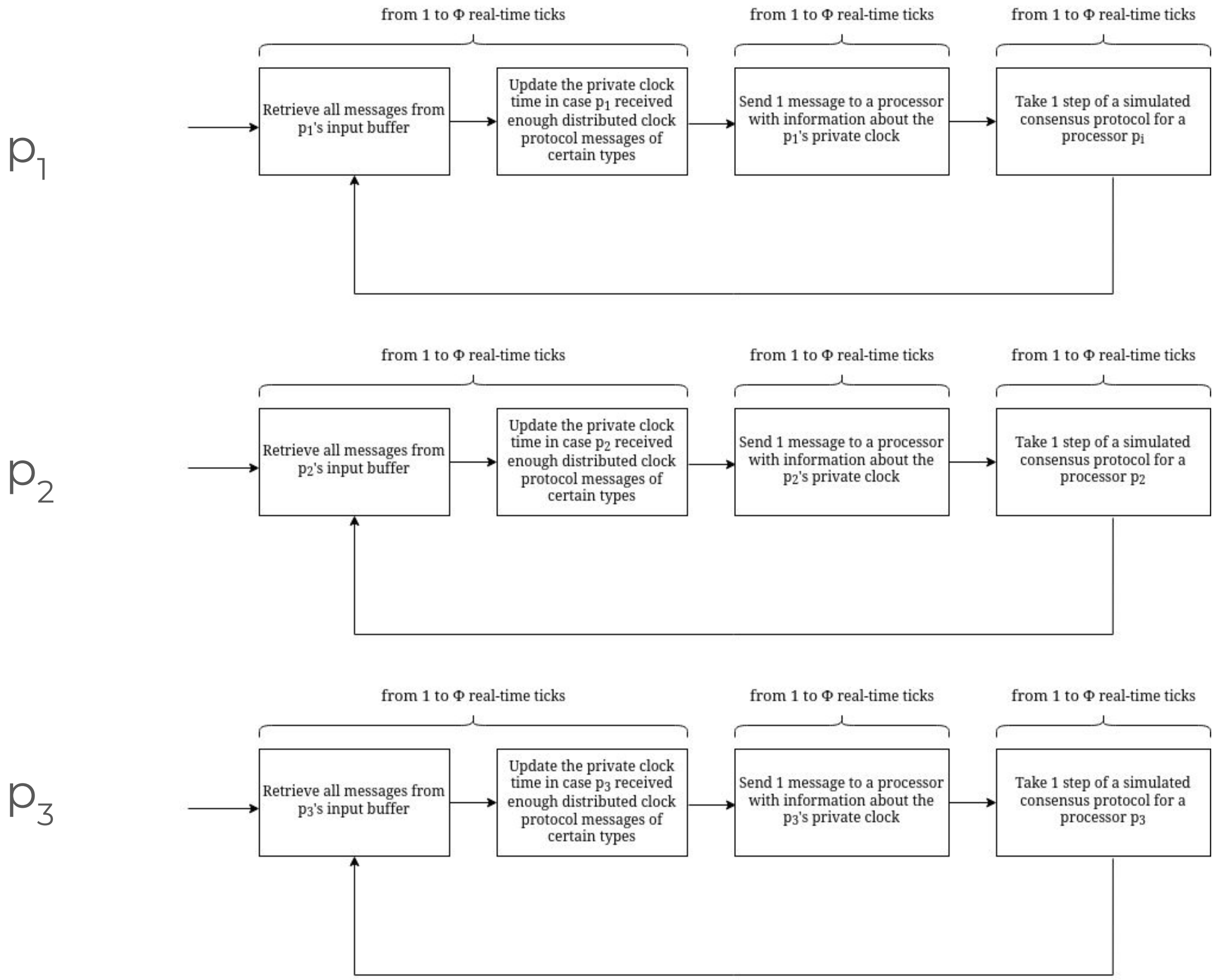
In order to simulate the consensus protocol, the processors will interleave the distributed clock protocol with the steps of the consensus protocol.

We assume that each round of the consensus protocol takes several “ticks” of private clock, which are integers that denote **the local time of the processor**.

Distributed clock protocol round


High-level explanation of processor p_i's steps in a distributed clock protocol
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
Retrieve all messages from p_i's input buffer
Update the private clock time in case p_i received enough distributed clock protocol messages of certain types
Send 1 message to a processor with information about the p_i's private clock
Take 1 step of a simulated consensus protocol for a processor p_i

Note: Clock update step is considered to be performed together with retrieving messages; **it does not take a separate "operation" to perform the clock update.**


p1
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
Retrieve all messages from p1's input buffer
Update the private clock time in case p1 received enough distributed clock protocol messages of certain types
Send 1 message to a processor with information about the p1's private clock
Take 1 step of a simulated consensus protocol for a processor pi
p2
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
Retrieve all messages from p2's input buffer
Update the private clock time in case p2 received enough distributed clock protocol messages of certain types
Send 1 message to a processor with information about the p2's private clock
Take 1 step of a simulated consensus protocol for a processor p2
p3
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
Retrieve all messages from p3's input buffer
Update the private clock time in case p3 received enough distributed clock protocol messages of certain types
Send 1 message to a processor with information about the p3's private clock
Take 1 step of a simulated consensus protocol for a processor p3

Sending information about time: ticks and claims

We also introduce two types of messages: *i-tick* and *i-claim*.

i-tick is a message consisting of non-negative integer i.
i_+- tick is a j-tick for any j ≥ i.
We say that a processor has broadcasted an i-tick if it has sent an i_+- tick to all processors.

i-claim is a message “I have broadcasted an i-tick”.
i_+- claim is a j-claim for any j ≥ i.
We say that a processor has broadcasted an i-claim if it has sent an i_+- claim to all processors.

Private clocks

For each processor p_i , their private clock time $c_i(s)$ is maximum j such that, by time s, processor p_i has received either

- messages from at least 2f + 1 different processors (at least f + 1 are from correct processors), where each message is a j_+-tick

OR

- messages from at least f + 1 different processors (at least one message is from a correct processor), where each message is either a $(j+1)_+$-tick or a $(j+1)_+$-claim.

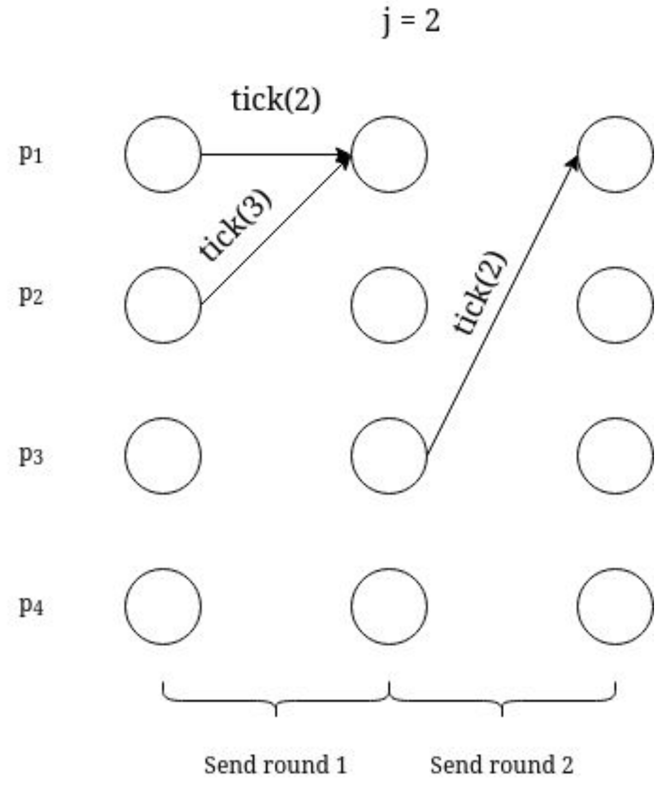
Note 1: $c_i(0) = 0$ for all *i*.

Note 2: messages from p_i to p_i (sender and receiver are identical) **count** for updating the private clock.

Updating private clocks

$c_i(s)$ = maximum j such that, by time s, p_i has received

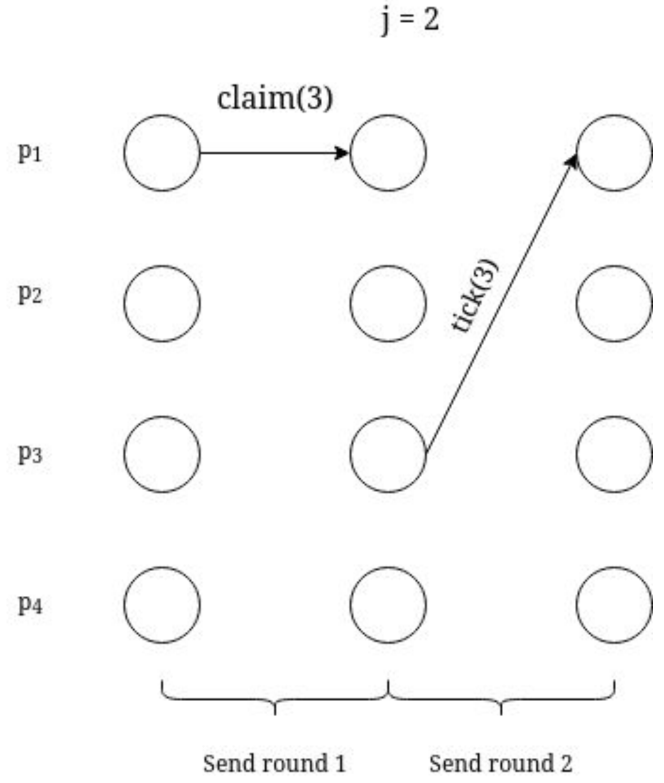
- messages from 2f + 1 different processors, where each message is a j_+-tick



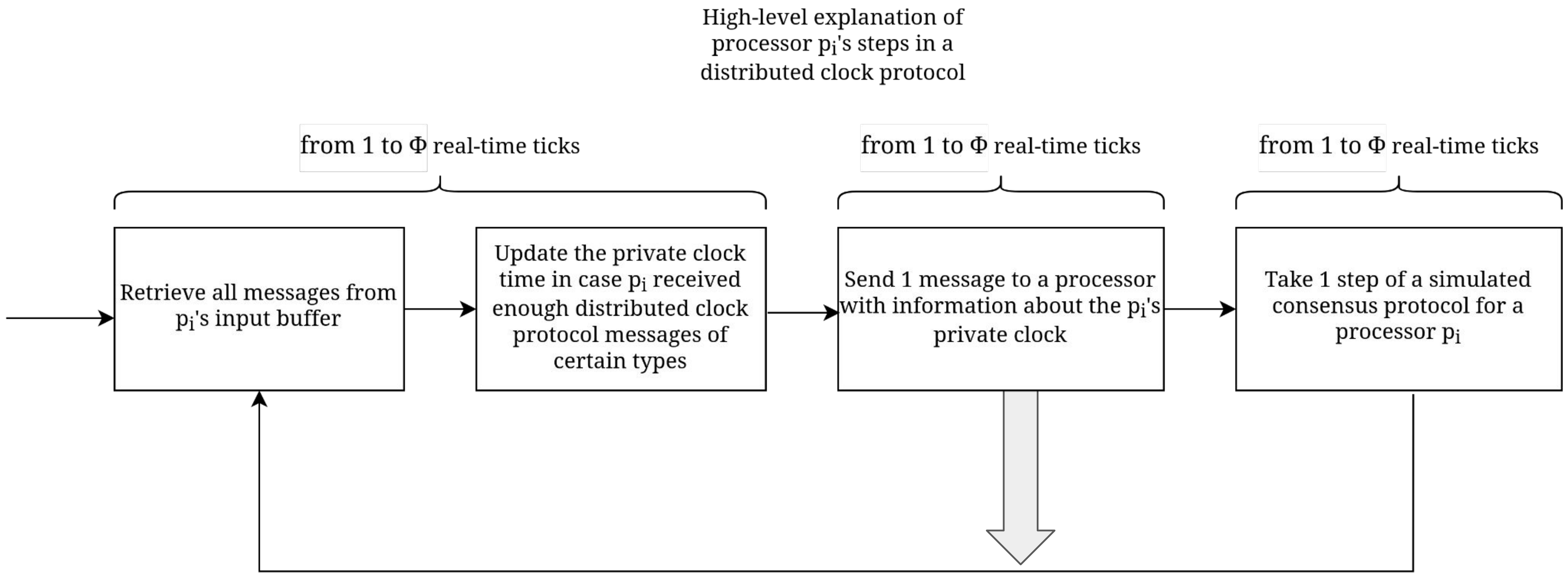
Updating private clocks

$c_i(s)$ = maximum j such that, by time s, p_i has received

- messages from f + 1 processors, where each message is either a $(j+1)_+$-tick or a $(j+1)_+$-claim.



Sending schedule


High-level explanation of processor p_i's steps in a distributed clock protocol
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
from 1 to Φ real-time ticks
Retrieve all messages from p_i's input buffer
Update the private clock time in case p_i received enough distributed clock protocol messages of certain types
Send 1 message to a processor with information about the p_i's private clock
Take 1 step of a simulated consensus protocol for a processor p_i

What exactly is being sent?

Sending schedule

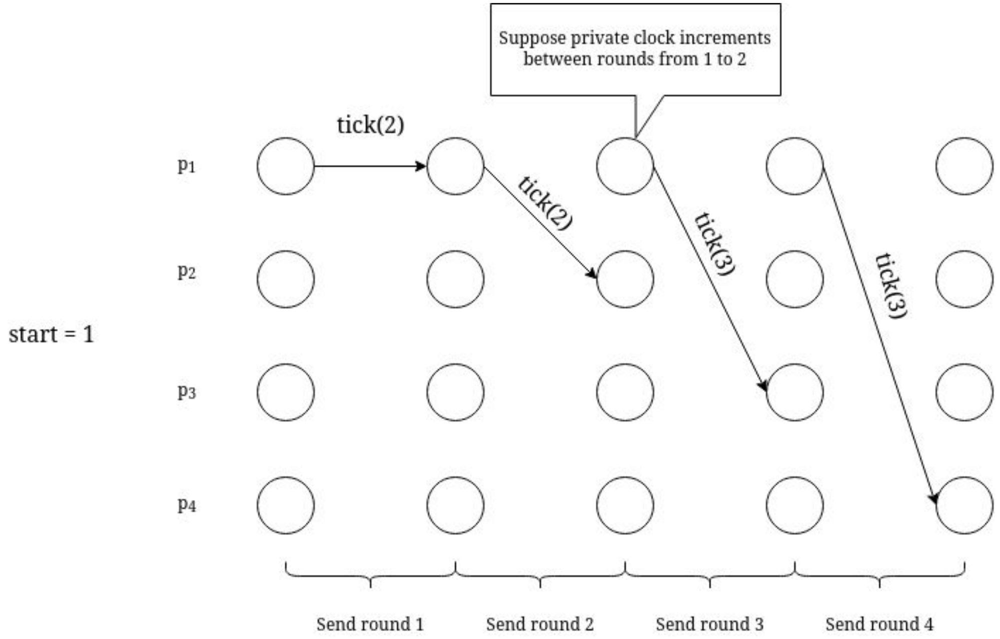
In the send step of a round of the protocol, the processor sends **one** message. The schedule for processor p_i to send messages is the following:

1. Before the schedule begins: p_i saves the current value of c_i to a local variable *start*
2. Send *(c_i + 1)* tick to all processors, one by one (note: c_i might increase during distributed clock protocol rounds)
3. Send (*start + 1)* claim to all processors, one by one (note: *start* **does not change** during distributed clock protocol rounds)
4. If c_i = start: go to line 3
5. If c_i > start: go to line 1

Start initialization & sending ticks

- Before the schedule begins: save the current c_i value to a local variable *start*
- Send *(c_i + 1)* tick to all processors, one by one (note: c_i might increase between distr. clock protocol rounds)

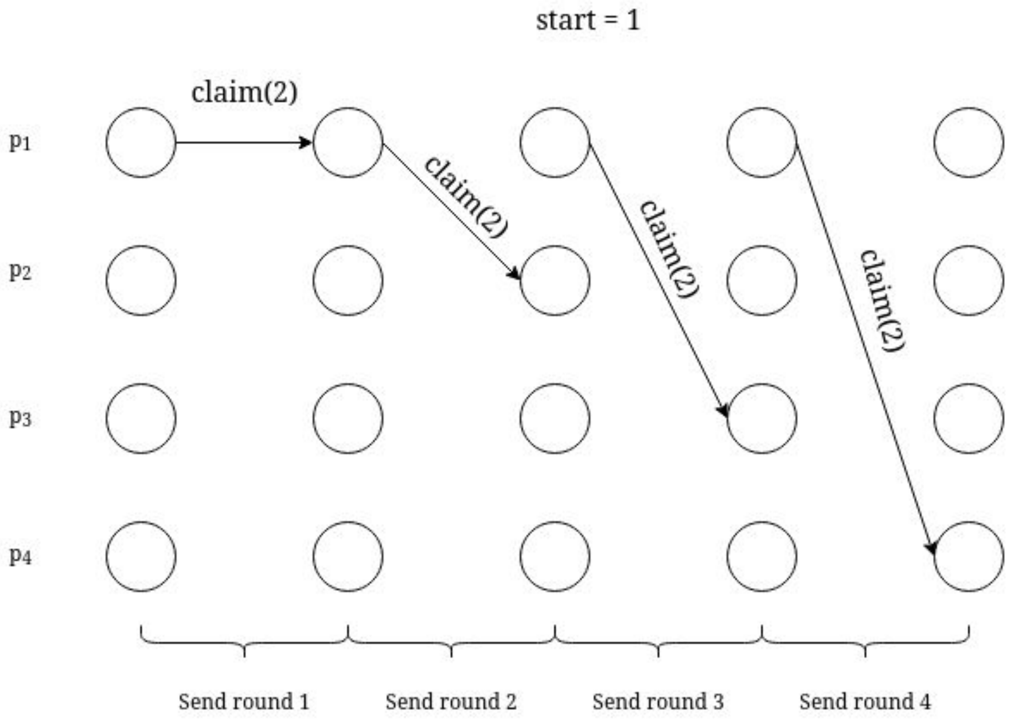
1. Before the schedule begins: p_i saves the current value of c_i to a local variable *start*
2. Send *(c_i + 1)* tick to all processors, one by one (note: c_i might increase during distributed clock protocol rounds)
3. Send (*start + 1)* claim to all processors, one by one (note: *start* **does not change** during distributed clock protocol rounds)
4. If c_i = start: go to line 3
5. If c_i > start: go to line 1



Start initialization & sending ticks

- Send *(start+1)* claims to all processors, one by one (note: *start* **does not change** between distr. clock protocol rounds)

```
1. Before the schedule begins: $p_i$ saves the current value of $c_i$ to a local variable *start*
2. Send $(c_i + 1)$ tick to all processors, one by one (note: $c_i$ might increase during distributed clock protocol rounds)
3. Send *(start + 1)* claim to all processors, one by one (note: *start* **does not change** during distributed clock protocol rounds)
4. If $c_i$ = start: go to line 3
5. If $c_i$ > start: go to line 1
```



Master clock

For convenience, we define a master clock, the state of which at time **s** depends on the global behaviour of the system. Its goal is to represent the desired time of all correct processors in the system.

The master clock time C(s), is defined at any real time s by:

C(s) = maximum j such that at least f + 1 **correct** processors have broadcast a j-tick by time s.

Since all private clocks start with 0, we say that C(0) = 0.

C(s) is a non-decreasing function of s.

Important lemma 1

Lemma 1. *For all $s \geq 0$ and for all i such that p_i is correct, $c_i(s) \leq C(s)$.*

Proof: By induction.

- Basis: $s = 0$. By assumption, for all i, $c_i(0) = C(0) = 0$.
- Inductive step: Fix some s and some correct p_i, and assume that the statement of the lemma is true for all $s' < s$. Let $j = c_i(s)$. By the definition of private clocks, there are two possibilities:
    - p_i has received j_+-*claims* from $2f + 1$ different processors. Since at least $f + 1$ of these j_+-*claims* are from correct processors, $C(s) \geq j$ by the definition of how master clock is updated.

Important lemma 1 (cont.)

Lemma 1. *For all $s \geq 0$ and for all i such that p_i is correct, $c_i(s) \leq C(s)$.*

Proof: By induction.

- p_i has received messages from $f + 1$ different processors, each of which is either *$(j+1)_+$-tick* or *$(j+1)_+$-claim*. Consider the earliest real time, *s''*, when some correct processor, say p_k, sends a *$(j+1)_+$-tick*. Note that $s'' < s$, so $c_k(s'') \leq C(s'')$ by the inductive hypothesis. By definition of the protocol, $c_k(s'') \geq j$ (in order to send a $(j+1)_+$-tick, c_k's private clock has to be at least j). Therefore, $j \leq c_k(s) \leq C(s'') \leq C(s)$.

1. Before the schedule begins: p_i saves the current value of c_i to a local variable *start*
2. Send *$(c_i + 1)$* tick to all processors, one by one (note: c_i might increase during distributed clock protocol rounds)
3. Send (*start + 1)* claim to all processors, one by one (note: *start* **does not change** during distributed clock protocol rounds)
4. If c_i = start: go to line 3
5. If c_i > start: go to line 1

Important lemma 2 (without proof) for GST model

Define D(elivery) = $\Delta + 3\Phi$. (Δ for message delivery, 3Φ to execute Receive operation).

Let s_0 be the minimum time such that **$C(s_0) \geq C(GST) + 2$**.

- For all $s \geq s_0 + D$ and for all correct processors p_i
  $C(s) - D - 1 \leq c_i(s) \leq C(s)$
  - **Distance between private clocks and the master clock is bounded**
- For all $s \geq s_0$,
  $C(s + 24N\Phi + 3D) \geq C(s) + 1$
  - **The master clock does not stop moving forward**

Important lemma 3 (without proof) for Δ and Φ unknown.

Define D(elivery) = $\Delta + 3\Phi$. (Δ for message delivery, 3Φ to execute Receive operation).

Let s_0 be the minimum time such that **$C(s_0) \geq 2$.**

- For all $s \geq s_0 + D$ and for all correct processors p_i
  $C(s) - D - 1 \leq c_i(s) \leq C(s)$
    - **Distance between private clocks and the master clock is bounded**
- For all $s \geq s_0$,
  $C(s + 24N\Phi + 3D) \geq C(s) + 1$
    - **The master clock does not stop moving forward**

Upper bound results (GST)

Fix algorithm A that solves consensus in a model when processors run at the same speed. Without loss of generality, we assume the consensus algorithm A has a certain structure. Each correct processor performs at most *N* “send message” operations and *1* “receive messages” operation per round of consensus algorithm A.

We also assume that all processors In the consensus protocol simulation execute “Send” operations first, and use the remaining round operations to execute “Receive” operations. Processors label the messages they send with round numbers.

Fix Δ and Φ for the GST model. We are going to prove that there is an algorithm which solves consensus in a model with partially synchronous Δ and Φ.

Since processors can only infer current time from their private clocks, they will use it to determine the round of algorithm *A* currently being simulated.

Determining round number

Fix $T = 3\Phi N + 2D + 2$, where $D = 3\Phi + \Delta$.

If $(r - 1)T \leq c_i(s) < rT$, then processor p_i determines at real time s that the current round is r.

Once again, it is important to note that the processor only knows the time of the private clock, and the $(r - 1)T \leq c_i(s) < rT$ inequality only deals with the time of the private clock.

Protocol correctness after GST

After GST, all correct processors should have time to perform all operations required to reach consensus.

Also, all correct processors should have time to receive all messages from other processors. Since they also need to send at most N messages per round, $3\Phi N + D$ is the amount of real time that is required for that.

Protocol correctness after GST

Assume that s is the first real time at which processor p_i's private clock reaches or exceeds $(r - 1)T$.

If $c_i(s + 3\Phi N + D) < rT$, then it means that there will always be enough time to perform all the required operations to reach consensus.

In simpler terms, the processor will perform all of its round *r* responsibilities before it thinks that it is already round *r+1*.

As it turns out, this is the case.

For any correct processor p_k in GST model:

$$c_k(s + 3\Phi N + D) \le rT$$

“Skipping rounds” before GST

Before GST, p_i might be slower than it should be (which is possible since Δ and Φ do not hold and it might take longer to send all of its messages for round R). In this case, p_i might get so far behind other processors that it will never catch up.

To deal with this, in case p_i infers that its next consensus protocol operation is for a round that is already over, it simulates all state transitions between current round and its last operation, but never sends a message. This is analogous to “losing a message before GST” in protocol with $\Phi = 1$. All messages that the processor has received for “skipped” rounds are taken into account when doing state transitions.

Final note about when round skipping stops

After some time after GST, this will never happen again since the previous result ensures every correct processor will have the time to perform all of its operations in the correct round.

Lemma 2. Let s_0 be the minimum time such that **$C(s_0) \geq C(GST) + 2$** ....