# Analyzing Paxos with Fault-Tolerant Multiparty Session Types

Bachelor thesis by Nicolas Daniel Torres

Date of submission: January 11, 2022

1. Review: Prof. Dr. Kirstin Peters
2. Review: M.Sc. Anna Schmitt
Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department

<institute>

<working group>

# Contents

# 1 Introduction

In distributed systems components on different computers coordinate and communicate via message passing to achieve a common goal. Sometimes, to achieve this goal, the individual components need to reach consensus, i.e., agree on the value of some data using a consensus algorithm. For example in state machine replication or when deciding which database transactions should be committed in what order. For such a distributed system to behave correctly the consensus algorithm needs to be correct. Thus, analyzing consensus algorithms is important.

To achieve consensus, consensus algorithms must satisfy the following properties: termination, validity, and agreement [2]. Proving these properties can be complicated. Model checking tools lead to big state-spaces so static analysis is preferable. For static analysis Multiparty Session Types are particularly interesting because session typing can ensure protocol conformance and the absence of communication errors and deadlocks [6].

Due to the presence of faulty processes and unreliable communication consensus algorithms are designed to be fault-tolerant. Modelling fault-tolerance is not possible using Multiparty Session Types, thus a fault-tolerant extension is necessary. Peters, Nestmann, and Wagner developed such an extension called Fault-Tolerant Multiparty Session Types.

In this work we will use Fault-Tolerant Multiparty Session Types to analyze the consensus algorithm Paxos, as described in [4].

# 2 Technical Preliminaries

First, we define the sorts, some additional notation, and use them to define the global type. Afterwards we define some sets and functions to create the processes.

# 3 Model

First, we specify some sorts with which we can then define the global type. Afterwards, we define the processes for the proposer and the acceptor. Finally, we will study an example run of the model.

## 3.1 Sorts

Sorts are basic data types. We assume the following sorts.

First, we have $\mathrm{Bool}$ which we define as a set.

$$\mathrm{Bool} = \{\mathrm{true}, \mathrm{false}\}$$

Second, we assume a set of values $\mathrm{Value}$.

Then, we have some sorts which we define using a grammar. Each of these definitions contains a type variable, which is a variable ranging over types. In this case the type variable in each definition is called $a$.

$$\mathrm{Maybe}\ \ a = \mathrm{Just}\ \ a \mid \mathrm{Nothing}$$

A value of type $\mathrm{Maybe}\ \ a$ can have the form $\mathrm{Just}\ \ a$ or $\mathrm{Nothing}$. Some examples include $\mathrm{Just}\ \ 4$ of type $\mathrm{Maybe}\ \ \mathbb{N}$, $\mathrm{Just}\ \ \mathrm{false}$ of type $\mathrm{Maybe}\ \ \mathrm{Bool}$, and $\mathrm{Nothing}$. $\mathrm{Nothing}$ itself does not dictate an exact type because its definition does not include the type variable $a$. The type is underspecified and is specified manually or through the context in which $\mathrm{Nothing}$ is used. It can be of type $\mathrm{Maybe}\ \ \mathbb{N}$, $\mathrm{Maybe}\ \ \mathrm{Bool}$, or any other type $b$ in $\mathrm{Maybe}\ \ b$. We use $\mathrm{Maybe}\ \ a$ where optional values are needed.

$$\mathrm{Proposal}\ \ a = \mathrm{Proposal}\ \ \mathbb{N}\ a$$

$\mathrm{Proposal}\ \ a$ only has one possible form, which is $\mathrm{Proposal}\ \ \mathbb{N}\ a$. A proposal contains its proposal number of type $\mathbb{N}$ and its value of type $a$. Again, $a$ is a variable ranging over types. An example for a value of type $\mathrm{Proposal}\ \ \mathrm{Bool}$ could be $\mathrm{Proposal}\ \ 1\ \ \mathrm{true}$ and an example for a value of type $\mathrm{Proposal}\ \ \mathrm{Maybe}\ \ \mathbb{N}$ could be $\mathrm{Proposal}\ \ 1\ \ \mathrm{Just}\ \ 1$. Note that $\mathrm{Proposal}\ \ 1\ \ \mathrm{Just}\ \ 1$ is of type $\mathrm{Proposal}\ \ a$ where $a = \mathrm{Maybe}\ \ b$ and $b = \mathbb{N}$. This sort models the proposals issued by the proposers in phase $2a$.

$$\mathrm{Promise}\ \ a = \mathrm{Promise}\ \ \mathrm{Maybe}\ \ \mathrm{Proposal}\ \ a \mid \mathrm{Nack}\ \ \mathbb{N}$$

$\mathrm{Promise}\ a$ has two possible forms. $\mathrm{Promise}\ \mathrm{Maybe}\ \mathrm{Proposal}\ a$ and $\mathrm{Nack}\ \mathbb{N}$. $\mathrm{Promise}\ \mathrm{Maybe}\ \mathrm{Proposal}\ a$ is the same as $\mathrm{Promise}\ c$ where $c = \mathrm{Maybe}\ b$ and $b = \mathrm{Proposal}\ a$. Possible values include $\mathrm{Nack}\ 1$ and $\mathrm{Promise}\ \mathrm{Just}\ \mathrm{Proposal}\ 1\ {-1}$ of type $\mathrm{Promise}\ \mathbb{Z}$. The actual type of $\mathrm{Nack}\ 1$, much like that of $\mathrm{Nothing}$, is underspecified. Again, we have to specify the exact type manually or through context.

In phase $1b$ the acceptors respond to the proposers prepare request with a value of type $\mathrm{Promise}\ \mathrm{Value}$. The prepare request contains a number $n$. The acceptors may respond to the prepare request with a promise to not accept any proposal numbered less than $n$ or with a rejection. In the first case the acceptor's response optionally includes the last proposal it accepted, if available, and is of the form $\mathrm{Promise}\ \mathrm{Maybe}\ \mathrm{Proposal}\ a$. In the second case it includes the highest $n$ that acceptor promised and is of the form $\mathrm{Nack}\ \mathbb{N}$.

## 3.2 Global Type

Since each proposer initiates its own session the global type can be defined for one proposer $p$ and a quorum of acceptors $A_Q$.

The last phase of Paxos contains no inter-process communication, so it is not modeled in the global type.

$$\begin{aligned} \mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}} = (\mu X) \left(\bigodot_{a\in A_Q} p \to_u a : l1a \left\langle \mathbb{N} \right\rangle\right) . \left(\bigodot_{a\in A_Q} a \to_u p : l1b \left\langle \mathrm{Promise}\ \mathrm{Value} \right\rangle\right) . \\ \left(p \to_w A_Q : \mathit{Accept}. \left(\bigodot_{a\in A_Q} p \to_u a : l2a \left\langle \mathrm{Proposal}\ \mathrm{Value} \right\rangle\right) .0 \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0\right) \end{aligned}$$

We can distinguish the individual phases of the Paxos algorithm by the labels $l1a$, $l1b$, and $l2a$.

In the first two steps, $1a$ and $1b$, the proposer sends its proposal number to each acceptor in $A_Q$ and listens for their responses. In step $2a$ the proposer decides whether to send an *Accept* or *Restart* message to restart the algorithm. This decision is broadcast to all acceptors in $A_Q$. Should the proposer crash the algorithm ends for this particular proposer and quorum of acceptors.

## 3.3 Functions

We define some functions which we use in the next section to define the processes.

$$\mathrm{proposalNumber} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$\mathrm{proposalNumber}_p\left(n\right)$ returns a proposal number for proposer $p$ when given a natural number $n$. It is used to pick a number for the prepare request in phase $1a$, which is also used in phase $2a$ in the actual proposal. We have two requirements for this function.

Let $\mathbb{P}$ be the set of proposers.

$$\forall p, q \in \mathbb{P}. \forall n, m \in \mathbb{N} : p \neq q \to \mathrm{proposalNumber}_p\left(n\right) \neq \mathrm{proposalNumber}_q\left(m\right)$$

Different proposers pick their proposal numbers from disjoint sets of numbers. This way different proposers never issue a proposal with the same proposal number.

$$\forall p \in \mathbb{P}. \forall n, m \in \mathbb{N} : n > m \rightarrow \mathrm{proposalNumber}_p\left(n\right) > \mathrm{proposalNumber}_p\left(m\right)$$

We require $\mathrm{proposalNumber}_p\left(n\right)$ to be strictly increasing for each proposer $p$ so every proposer uses a higher proposal number than any it has already used.

$$\mathrm{promiseValue} : \texttt{list of}\ \ \mathrm{Promise}\ \ a \rightarrow a$$

$\mathrm{promiseValue}\left(ps\right)$ returns a fresh value if none of the promises in $ps$ contain a value. Otherwise, the best value is returned. Usually, that means the value with the highest associated proposal number. A promise contains a value $v$ if it is of the form $\mathrm{Promise}\ \ \mathrm{Just}\ \ v$. With this function we can model the picking of a value for a proposal in phase $2a$.

$$\begin{aligned}
&\mathrm{anyNack} : \texttt{list of}\ \ \mathrm{Promise}\ \ a \rightarrow \mathrm{Bool}\\
&\mathrm{anyNack}\left([]\right) = \mathrm{false}\\
&\mathrm{anyNack}\left((\mathrm{Nack}\ \ \_\#\_)\right) = \mathrm{true}\\
&\mathrm{anyNack}\left((\_\#xs)\right) = \mathrm{anyNack}\left(xs\right)
\end{aligned}$$

$\mathrm{anyNack}\left(ps\right)$ returns $\mathrm{true}$ if the list contains at least one promise of the form $\mathrm{Nack}\ \ n$. Otherwise, it returns $\mathrm{false}$.

$$\begin{aligned}
&\mathrm{promiseCount} : \texttt{list of}\ \ \mathrm{Promise}\ \ a \rightarrow \mathbb{N}\\
&\mathrm{promiseCount}\left([]\right) = 0\\
&\mathrm{promiseCount}\left((\mathrm{Promise}\ \ \_\#xs)\right) = 1 + \mathrm{promiseCount}\left(xs\right)\\
&\mathrm{promiseCount}\left((\_\#xs)\right) = \mathrm{promiseCount}\left(xs\right)
\end{aligned}$$

$\mathrm{promiseCount}\left(ps\right)$ takes a list of promises $ps$ and calculates the number of promises in that list of that have the form $\mathrm{Promise}\ \ m$.

$\mathrm{anyNack}\left(ps\right)$ and $\mathrm{promiseCount}\left(ps\right)$ are used in the proposer to decide which branch to take in phase $2a$.

$$\begin{aligned}
&\mathrm{gt} : a \rightarrow \mathrm{Maybe}\ \ a \rightarrow \mathrm{Bool}\\
&\mathrm{gt}\left(\_, \mathrm{Nothing}\right) = \mathrm{true}\\
&\mathrm{gt}\left(a, \mathrm{Just}\ \ b\right) = a > b
\end{aligned}$$

$$\begin{aligned}
&\mathrm{ge} : a \rightarrow \mathrm{Maybe}\ \ a \rightarrow \mathrm{Bool}\\
&\mathrm{ge}\left(\_, \mathrm{Nothing}\right) = \mathrm{true}\\
&\mathrm{ge}\left(a, \mathrm{Just}\ \ b\right) = a \geq b
\end{aligned}$$

$$\begin{aligned}&\mathrm{nFromProposal} : \mathrm{Proposal}\ \ a \rightarrow \mathbb{N}\\&\mathrm{nFromProposal}\,(\mathrm{Proposal}\ \ n\ \_) = n\end{aligned}$$

$\mathrm{nFromProposal}\,(p)$ retrieves the proposal number $n$ inside proposal $p$, which has the form $\mathrm{Proposal}\ \ n\ pr$.

$\mathrm{nFromProposal}\,(p)$, $\mathrm{gt}\,(a, ma)$, and $\mathrm{ge}\,(a, ma)$ are used to extract and compare proposal numbers in phase $2b$ of the acceptor.

$$\mathrm{genA_Q} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \texttt{list of } \mathbb{N}$$

$\mathrm{genA_Q}\,(p, c_A, c_P)$ returns a randomly selected set $A_Q$ with $A_Q \subseteq A = \{1, \ldots, c_A\}$ and $|A_Q| > \frac{|A|}{2}$. $A_Q$ consists of any majority of acceptors. In Paxos a majority of acceptors forms a quorum, i.e. an accepting set with which a value can be chosen [3]. We use this function when initiating the proposers to give them a quorum of acceptors with which they communicate.

## 3.4 Processes

### 3.4.1 System Initialization

$$\begin{aligned}&\mathrm{Sys}\,(c_A, c_P) = \overline{o}\,[2]\,(t)\,.\,\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(c_A + 1, \mathrm{genA_Q}\,(c_A + c_P, c_A, c_P)\,, c_A + c_P, c_A + c_P, [])\\&\mid o\,[1]\,(t)\,.\Pi_{c_A<k<c_A+c_P}\ \ \mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(c_A + 1, \mathrm{genA_Q}\,(k, c_A, c_P)\,, k, k, [])\\&\mid \Pi_{1\leq j\leq c_A}\ \ \mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\,(j, c_A + 1, c_A, c_P, n_a, pr_a)\end{aligned}$$

$$\begin{aligned}&\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(p, A_Q, n, m, \overrightarrow{V}\right) = \overline{b_n}\,[i]\,(s)\,.\,\mathrm{P}^{\mathrm{P}}\\&\mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\,(a, p, c_A, c_P, n, pr) = \Pi_{c_A<k\leq c_A+c_P}\ b_k\,[a]\,(s)\,.\,\mathrm{P}^{\mathrm{A}}_1\end{aligned}$$

$\mathrm{Sys}\,(c_A, c_P)$, $\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(p, A_Q, n, m, \overrightarrow{V}\right)$, and $\mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\,(a, p, c_A, c_P, n, pr)$ describe the system initialization. $c_A$ and $c_P$ are the number of acceptors and proposers respectively.

An outer session is created through shared-point $o$. This outer session is not strictly necessary but was left in to allow for easier extension of the model. The acceptors are initialized using indices from 1 to $c_A$ and the proposers are initialized using indices from $c_A + 1$ to $c_A + c_P$.

$\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(p, A_Q, n, m, \overrightarrow{V}\right)$ is initialized with the proposer's role in its own session $p$, which is always $c_A + 1$, a quorum of acceptors $A_Q$, an index $n$, and a vector $\overrightarrow{V}$. Each proposer has the same role $p = c_A + 1$ but uses a different shared-point $b_n$ according to its index $n$. $m$ is initialized to the same value as $n$ but is never updated. $\overrightarrow{V}$ is used in the proposer to collect and evaluate the responses from the acceptors. It is always initialized with an empty list []. Shared-point $b_n$ is used to initiate a session. Afterwards, the process behaves like $\mathrm{P}^{\mathrm{P}}$.

$\mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\left(a, p, c_A, c_P, n, pr\right)$ is initialized with the acceptor's index $a$, the proposer index $p$, which is always $c_A + 1$, $c_A$, $c_P$, initial knowledge for the highest promised proposal number $n$, if available, and initial knowledge for the most recently accepted proposal $pr$, if available. $n$ is of type $\mathrm{Maybe}\ \mathbb{N}$ and $pr$ is of type $\mathrm{Maybe}\ (\mathrm{Proposal}\ \mathrm{Value})$ thus both can be $\mathrm{Nothing}$. Each of the proposers' session requests are accepted in a separate subprocess. These subprocesses run parallel to each other but still access the same values for $n$ and $pr$. We observe that each subprocess in an acceptor accesses a different channel $s$, since it is generated by the proposer and passed through when the proposers' session request is accepted. Afterwards, each subprocess behaves like $\mathrm{P}^{\mathrm{A}}_{1}$.

### 3.4.2 Proposer

To define the proposer and the acceptor we introduce a function $\mathrm{update}\,(n, m)$ which replaces the value inside $n$ with the value of $m$. We use this function to update the local variables of the processes.

$$\begin{aligned}
\mathrm{P}^{\mathrm{P}} = {} & (\mu X)\, \mathrm{update}\,(n, n+1)\,. \\
& \left(\bigodot_{a \in A_Q}\ s\,[p, a]!_u l1a\, \langle \mathrm{proposalNumber}_m\,(n) \rangle\right). \\
& \left(\bigodot_{a \in A_Q}\ s\,[a, p]?_u l1b\, \langle \bot \rangle\, (v_a)\right). \\
& \text{if}\ \ \mathrm{anyNack}\left(\overrightarrow{V}\right)\ \text{ or }\ \mathrm{promiseCount}\left(\overrightarrow{V}\right) < \left\lceil \frac{p}{2} \right\rceil \\
& \quad \text{then}\ \ s\,[p, A_Q]!_w \mathit{Restart}.X \\
& \quad \text{else} \\
& \quad\ s\,[p, A_Q]!_w \mathit{Accept}. \\
& \quad \left(\bigodot_{a \in A_Q}\ s\,[p, a]!_u l2a \left\langle \mathrm{Proposal}\ \ \mathrm{proposalNumber}_m\,(n)\ \ \mathrm{promiseValue}\left(\overrightarrow{V}\right)\right\rangle\right).0
\end{aligned}$$

At the start of the recursion $n$ is incremented to make sure every run of the recursion uses a different $n$ and thus a different proposal number. The proposal number is sent to every acceptor in $A_Q$ and their replies are gathered in $\overrightarrow{V}$ through $v_a$. Because $p = c_A + 1$, the minimum number of acceptors needed to form a majority is $\left\lceil \frac{p}{2} \right\rceil = \left\lceil \frac{c_A+1}{2} \right\rceil$. If any $\mathrm{Nack}\ x$ was received or the number of $\mathrm{Promise}\ y$ received is less than that needed for the smallest majority the proposer restarts the algorithm. Otherwise, the proposer sends its proposal to the acceptors and terminates.

### 3.4.3 Acceptor

```
$\mathrm{P}^{\mathrm{A}}_1 = (\mu X)\, s\,[p,a]?_u l1a\,\langle\bot\rangle\,(n')\,.$
 if $n' = \bot$
   then $s\,[a,p]!_u l1b\,\langle\bot\rangle\,.\,\mathrm{P}^{\mathrm{A}}_2$
   else
     if $\mathrm{gt}\,(n', n)$
     then $\mathrm{update}\,(n, n')\,.s\,[a,p]!_u l1b\,\langle\mathrm{Promise}\ \ pr\rangle\,.\,\mathrm{P}^{\mathrm{A}}_2$
     else $s\,[a,p]!_u l1b\,\langle\mathrm{Nack}\ \ n\rangle\,.\,\mathrm{P}^{\mathrm{A}}_2$

$\mathrm{P}^{\mathrm{A}}_2 = s\,[p,a]?_w Accept.s\,[p,a]?_u l2a\,\langle\bot\rangle\,(pr')\,.$
   if $pr' = \bot$
     then $0$
     else
       if $\mathrm{ge}\,(\mathrm{nFromProposal}\,(pr')\,, n)$
         then $\mathrm{update}\,(pr, pr')\,.\,\mathrm{update}\,(n, \mathrm{Just}\ \ \mathrm{nFromProposal}\,(pr'))\,.0$
         else $0$
 $\oplus\, Restart.X$
 $\oplus\, Abort.0$
```

For each proposer an acceptor has a corresponding subprocess, which behaves like $\mathrm{P}^{\mathrm{A}}_1$. These subprocesses access the same values for $n$ and $pr$. This means that updating these values with $\mathrm{update}\,(n, m)$ updates them for all subprocesses of an acceptor.

Each subprocess can communicate with one proposer. Thus, if that proposer does not or can not communicate with a particular subprocess of an acceptor then there is no need for that subprocess. It is possible that an acceptor participates in a proposer's session but is not contained in the proposer's quorum of acceptors $A_Q$, in which case the proposer does not communicate with that acceptor. It is also possible for a proposer to crash or otherwise terminate, in which case the proposer can not communicate with that acceptor.

Each subprocess starts out by potentially receiving a proposal number $n'$ from the corresponding proposer. If the acceptor does receive a proposal number $n'$ it responds with either $\mathrm{Promise}\ \ pr$ or $\mathrm{Nack}\ \ n$, depending on the values of $n'$ and $n$. If the acceptor does not receive a proposal number then it sends $\bot$ to the proposer. Sending $\bot$ to the proposer is only necessary to maintain the global type. In either case the subprocess moves on to receive the proposers' decision in phase $2a$.

Since the proposers' decision broadcast is weakly reliable, there are two cases in which the acceptor receives no decision. The proposer might have terminated or this particular acceptor is not in the proposers' quorum of acceptors $A_Q$. In either case this particular subprocess of the acceptor is no longer needed, because each subprocess of the acceptor exclusively communicates with one proposer. Thus, the subprocess terminates in the default branch $Abort$.

In the $Restart$ branch this particular subprocess of the acceptor restarts the algorithm to match the corresponding proposer.

In the $Accept$ branch the acceptor potentially receives a proposal $pr'$ from the corresponding proposer. The acceptor updates $n$ and $pr$ if the proposal number in $pr'$ is greater or equal to $n$. Then the subprocess terminates. If the acceptor does not receive a proposal or the proposal number of $pr'$ is less than $n$ the subprocess terminates without updating $n$ or $pr$.

## 3.5 Failure Patterns

Chandra and Toueg introduce a class of failure detectors $\Diamond\mathscr{S}$, which is called *eventually strong* in [1]. Failure detectors in $\Diamond\mathscr{S}$ satisfy the following properties: (1) eventually every process that crashes is permanently suspected by every correct process and (2) eventually some correct process is never suspected by any correct process.

In all three phases modeled in the global type it is possible to suspect senders. In phases $1a$ and $2a$, with labels $l1a$ and $l2a$ respectively, the acceptors may suspect some proposers. The proposers may suspect some acceptors in phase $1b$ with label $l1b$. Accordingly, $\texttt{FP}_{\texttt{uskip}}$ is implemented with a failure detector in $\Diamond\mathscr{S}$ for phases $1a$, $1b$, and $2a$.

Similarly, message loss is possible in all phases modeled in the global type. Thus, $\texttt{FP}_{\texttt{ml}}$ is also implemented with a failure detector in $\Diamond\mathscr{S}$ with one exception. $\texttt{FP}_{\texttt{ml}}\ (s, p, a, l)$ returns $\mathrm{true}$ if $p$ is a proposer, $a$ is an acceptor that is not contained in the proposers' quorum of acceptors, and $l = l1b$. Since any proposer only communicates with the acceptors in its quorum, we can discard any messages from acceptors outside it.

For the weakly reliable broadcast in phase $2a$, the failure pattern $\texttt{FP}_{\texttt{wskip}}$ returns $\mathrm{true}$ for sub-processes of acceptors if, and only if, the corresponding proposer crashed, otherwise terminated, or the corresponding proposer's quorum does not include that particular acceptor.

For Paxos to work a majority of acceptors needs to be alive. That means that the number of failed acceptors $f$ needs to satisfy $n > 2f$ where $n$ is the total number of acceptors, except in one case where there are 2 acceptors. Then, at most one acceptor may crash [3]. For acceptors $\texttt{FP}_{\texttt{crash}}$ returns $\mathrm{true}$ if, and only if, at least one more acceptor may crash, i.e. $n > 2(f + 1)$ is satisfied. Let $\mathbb{F}$ be the set of processes permanently suspected by a failure detector in $\Diamond\mathscr{S}$. For proposers $\texttt{FP}_{\texttt{crash}}$ returns $\mathrm{true}$ if $A_Q \setminus \mathbb{F}$ is not a quorum, i.e. if the set of acceptors in $A_Q$ that are not permanently suspected is not enough to form a majority of acceptors.

In Paxos there is no need to reject outdated messages so $\texttt{FP}_{\texttt{uget}}$ is implemented with a constant $\mathrm{true}$.

## 3.6 Example

In this section we will study an example run of the model with $3$ acceptors and $2$ proposers. First, we will take a look at the example scenario. Then we will examine the scenario using reduction rules starting at system initialization.

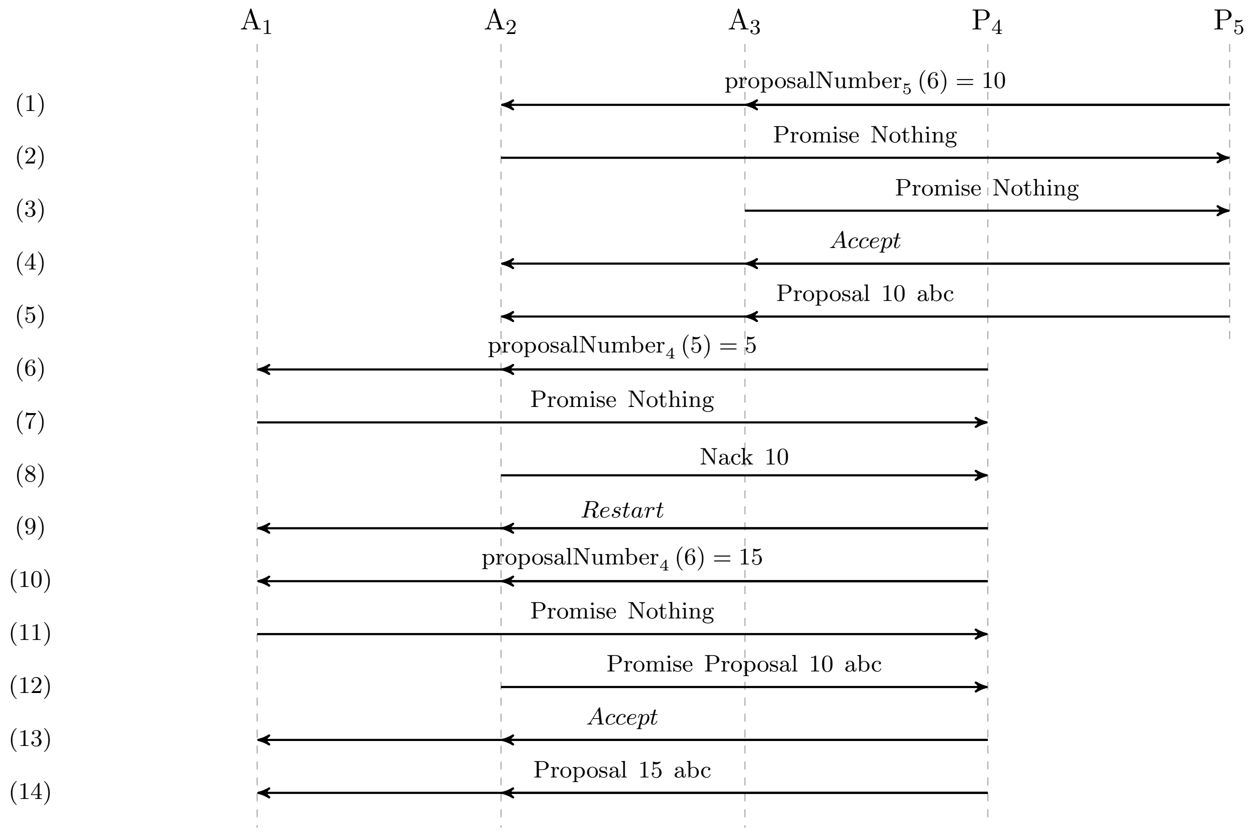

Figure 3.1: Example scenario with $3$ acceptors and $2$ proposers.

### 3.6.1 Scenario

Figure 3.1 provides an overview where $\mathrm{A}_1$, $\mathrm{A}_2$, and $\mathrm{A}_3$ are the acceptors and $\mathrm{P}_4$ and $\mathrm{P}_5$ are the proposers. In steps $(1)$ to $(5)$, $\mathrm{P}_5$ completes the Paxos algorithm with $\mathrm{A}_2$ and $\mathrm{A}_3$ and terminates.

At this point $\mathrm{A}_2$ has promised not to accept any proposal numbered less than $10$ and has accepted the value $\mathrm{abc}$. So, when $\mathrm{P}_4$ tries to use $5$ as its proposal number $(6)$, it receives $\mathrm{Nack}\ 10$ from $\mathrm{A}_2$ $(8)$ and has to restart the algorithm $(9)$.

$\mathrm{P}_4$ then runs through the Paxos algorithm with $\mathrm{A}_1$ and $\mathrm{A}_2$ starting with a new prepare request $(10)$ with a higher proposal number. In step $(12)$ $\mathrm{P}_4$ learns that value $\mathrm{abc}$ with proposal number $10$ has already been accepted by $\mathrm{A}_2$. Later, in step $(14)$, $\mathrm{P}_4$ issues a proposal with the value of the highest-numbered proposal that it receives as a response to its prepare request. In this case there is only one such proposal, which is $\mathrm{Proposal}\ 10\ \mathrm{abc}$.

In the end all $3$ acceptors have accepted the value $\mathrm{abc}$. $\mathrm{A}_1$ and $\mathrm{A}_2$ have accepted $\mathrm{Proposal}\ 15\ \mathrm{abc}$ and $\mathrm{A}_3$ has accepted $\mathrm{Proposal}\ 10\ \mathrm{abc}$.

### 3.6.2 Formulae

We set $c_A = 3$, $c_P = 2$, and $V = \{\mathrm{abc}, \mathrm{def}, \ldots, \mathrm{vwx}, \mathrm{yz}\}$.

#### System Initialization

After inserting $c_A$ and $c_P$ and applying $(\mathrm{Init})$ once for shared-point $a$ we have:

$$\begin{aligned}
&\mathrm{Sys}\left(c_A, c_P\right) = \mathrm{Sys}\left(3, 2\right) = \\
&o\left[1\right]\left(t\right).\Pi_{3<k<5}\ \mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(4, \mathrm{genA_Q}\left(k, 3, 2\right), k, k, []\right) \\
&\mid \overline{o}\left[2\right]\left(t\right).\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(4, \mathrm{genA_Q}\left(5, 3, 2\right), 5, 5, []\right) \\
&\mid \Pi_{1\le a\le 3}\ \mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\left(a, 4, 3, 2, n_a, pr_a\right) \\
&\\
&\longmapsto^{*} \left(\nu t\right)\left(\overline{b_4}\left[4\right]\left(s\right).\mathrm{P}^{\mathrm{P}} \ = \mathrm{P}_4\right. \\
&\mid \overline{b_5}\left[4\right]\left(r\right).\mathrm{P}^{\mathrm{P}} \ = \mathrm{P}_5 \\
&\mid \ \left(b_4\left[1\right]\left(s\right).\mathrm{P}^{\mathrm{A}}_1 \mid b_5\left[1\right]\left(r\right).\mathrm{P}^{\mathrm{A}}_1\right) \ = \mathrm{A}_1 \\
&\mid \ \left(b_4\left[2\right]\left(s\right).\mathrm{P}^{\mathrm{A}}_1 \mid b_5\left[2\right]\left(r\right).\mathrm{P}^{\mathrm{A}}_1\right) \ = \mathrm{A}_2 \\
&\mid \ \left(b_4\left[3\right]\left(s\right).\mathrm{P}^{\mathrm{A}}_1 \mid b_5\left[3\right]\left(r\right).\mathrm{P}^{\mathrm{A}}_1\right) \ = \mathrm{A}_3 \\
&\mid \left.\Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : []\right)
\end{aligned}$$

Note that the outer session created via shared-point $o$ isn't strictly necessary in the model. We apply $(\mathrm{Init})$ once for shared-point $b_4$ and once again for shared-point $b_5$ to obtain:

$$
\begin{array}{l}
\longmapsto^{*}\ (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
(\mu X)\,\mathrm{update}\,(n,5)\,.\left(\bigodot_{a\in\{1,2\}}\ s\,[4,a]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(n)\rangle\right)\ldots\ = \mathrm{P}_{4}\\
\mid\ (\mu X)\,\mathrm{update}\,(n,6)\,.\left(\bigodot_{a\in\{2,3\}}\ r\,[4,a]!_{u}l1a\,\langle \mathrm{proposalNumber}_{5}\,(n)\rangle\right)\ldots\ = \mathrm{P}_{5}\\
\mid\ \big((\mu X)\,s\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,r\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\big)\ = \mathrm{A}_{1}\\
\mid\ \big((\mu X)\,s\,[4,2]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,r\,[4,2]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\big)\ = \mathrm{A}_{2}\\
\mid\ \big((\mu X)\,s\,[4,3]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,r\,[4,3]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\big)\ = \mathrm{A}_{3}\\
\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : [\,]\mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : [\,]\mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : [\,]\big)
\end{array}
$$

Note that each process is shortened to only show the next few steps instead of the entire process.

### The Happy Path

After applying the updates in $\mathrm{P}_4$ and $\mathrm{P}_5$ the first inter-process communication can take place. In this case $\mathrm{P}_5$ communicates with $\mathrm{A}_2$ and $\mathrm{A}_3$. We apply $(\mathrm{USend})$ and $(\mathrm{UGet})$ twice to send $\mathrm{proposalNumber}_5\,(6) = 10$ to $\mathrm{A}_2$ and $\mathrm{A}_3$.

$$
\begin{array}{l}
\longmapsto^{*}\ (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
(\mu X)\,s\,[4,1]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(5)\rangle\,.s\,[4,2]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(5)\rangle\ldots\ = \mathrm{P}_{4}\\
\mid\ (\mu X)\,r\,[2,4]?_{u}l1b\,\langle\bot\rangle\,(v_2)\,.r\,[3,4]?_{u}l1b\,\langle\bot\rangle\,(v_3)\ldots\ = \mathrm{P}_{5}\\
\mid\ \big((\mu X)\,s\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,r\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\big)\ = \mathrm{A}_{1}\\
\mid\ \big((\mu X)\,s\,[4,2]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,\mathrm{if}\ \ 10=\bot\,\mathrm{then}\ \ s\,[2,4]!_{u}l1b\,\langle\bot\rangle\,.\,\mathrm{P}_{2}^{\mathrm{A}}\ \mathrm{else}\ \ldots\big)\ = \mathrm{A}_{2}\\
\mid\ \big((\mu X)\,s\,[4,3]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,\mathrm{if}\ \ 10=\bot\,\mathrm{then}\ \ s\,[3,4]!_{u}l1b\,\langle\bot\rangle\,.\,\mathrm{P}_{2}^{\mathrm{A}}\ \mathrm{else}\ \ldots\big)\ = \mathrm{A}_{3}\\
\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : [\,]\mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : [\,]\mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : [\,]\big)
\end{array}
$$

Since $10 \neq \bot$ both $\mathrm{A}_2$ and $\mathrm{A}_3$ move into their respective $\mathrm{else}$ branches.

$$
\begin{array}{l}
=\ (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
(\mu X)\,s\,[4,1]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(5)\rangle\,.s\,[4,2]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(5)\rangle\ldots\ = \mathrm{P}_{4}\\
\mid\ (\mu X)\,r\,[2,4]?_{u}l1b\,\langle\bot\rangle\,(v_2)\,.r\,[3,4]?_{u}l1b\,\langle\bot\rangle\,(v_3)\ldots\ = \mathrm{P}_{5}\\
\mid\ \big((\mu X)\,s\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,r\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\big)\ = \mathrm{A}_{1}\\
\mid\ \big((\mu X)\,s\,[4,2]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,\mathrm{if}\ \ \mathrm{gt}\,(10,\mathrm{Nothing})\ \mathrm{then}\ \ldots\mathrm{else}\ \ldots\big)\ = \mathrm{A}_{2}\\
\mid\ \big((\mu X)\,s\,[4,3]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,\mathrm{if}\ \ \mathrm{gt}\,(10,\mathrm{Nothing})\ \mathrm{then}\ \ldots\mathrm{else}\ \ldots\big)\ = \mathrm{A}_{3}\\
\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : [\,]\mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : [\,]\mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : [\,]\big)
\end{array}
$$

Because $\mathrm{gt}\,(10,\mathrm{Nothing})$ returns $\mathrm{true}$, $\mathrm{A}_2$ and $\mathrm{A}_3$ move into their respective $\mathrm{then}$ branches. After executing $\mathrm{update}\,(n,10)$, $\mathrm{A}_2$ and $\mathrm{A}_3$ are ready to send their responses to $\mathrm{P}_5$.

$$
\begin{aligned}
&= (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
&(\mu X)\,s\,[4,1]!_u l1a\,\langle \mathrm{proposalNumber}_4\,(5)\rangle\,.s\,[4,2]!_u l1a\,\langle \mathrm{proposalNumber}_4\,(5)\rangle \ldots\ = \mathrm{P}_4\\
&\mid\ (\mu X)\,r\,[2,4]?_u l1b\,\langle\bot\rangle\,(v_2)\,.r\,[3,4]?_u l1b\,\langle\bot\rangle\,(v_3)\ldots\ = \mathrm{P}_5\\
&\mid\ \big((\mu X)\,s\,[4,1]?_u l1a\,\langle\bot\rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,r\,[4,1]?_u l1a\,\langle\bot\rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ldots\big)\ = \mathrm{A}_1\\
&\mid\ \big((\mu X)\,s\,[4,2]?_u l1a\,\langle\bot\rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,r\,[2,4]!_u l1b\,\langle \mathrm{Promise\ \ Nothing}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\big)\ = \mathrm{A}_2\\
&\mid\ \big((\mu X)\,s\,[4,3]?_u l1a\,\langle\bot\rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ldots\ \mid\ (\mu X)\,r\,[3,4]!_u l1b\,\langle \mathrm{Promise\ \ Nothing}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\big)\ = \mathrm{A}_3\\
&\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : []\mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : []\mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : []\big)
\end{aligned}
$$

We apply (USend) and (UGet) twice to do just that. Note that we also apply (USkip) to $\mathrm{A}_1$, evaluate its branches, and apply (USend) to $\mathrm{A}_1$ and then (ML) to $\mathrm{P}_5$ to discard the dummy message. All three acceptors move into $\mathrm{P}_2^{\mathrm{A}}$.

$$
\begin{aligned}
&\longmapsto^*\ (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
&(\mu X)\,s\,[4,1]!_u l1a\,\langle \mathrm{proposalNumber}_4\,(5)\rangle\,.s\,[4,2]!_u l1a\,\langle \mathrm{proposalNumber}_4\,(5)\rangle \ldots\ = \mathrm{P}_4\\
&\mid\ (\mu X)\,r\,[4,\{2,3\}]!_w \mathit{Accept}.r\,[4,2]!_u l2a\,\langle \mathrm{Proposal\ \ 10\ \ abc}\rangle \ldots\ = \mathrm{P}_5\\
&\mid\ ((\mu X)\ldots\ \mid\ (\mu X)\,r\,[4,1]?_w \mathit{Accept}\cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0)\ = \mathrm{A}_1\\
&\mid\ ((\mu X)\ldots\ \mid\ (\mu X)\,r\,[4,2]?_w \mathit{Accept}\cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0)\ = \mathrm{A}_2\\
&\mid\ ((\mu X)\ldots\ \mid\ (\mu X)\,r\,[4,3]?_w \mathit{Accept}\cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0)\ = \mathrm{A}_3\\
&\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : []\mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : []\mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : []\big)
\end{aligned}
$$

$\mathrm{P}_5$ broadcasts its decision *Accept* to $\mathrm{A}_2$ and $\mathrm{A}_3$. By applying (WSel) once, (WBran) twice we obtain:

$$
\begin{aligned}
&\longmapsto^*\ (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
&(\mu X)\,s\,[4,1]!_u l1a\,\langle \mathrm{proposalNumber}_4\,(5)\rangle\,.s\,[4,2]!_u l1a\,\langle \mathrm{proposalNumber}_4\,(5)\rangle \ldots\ = \mathrm{P}_4\\
&\mid\ (\mu X)\,r\,[4,2]!_u l2a\,\langle \mathrm{Proposal\ \ 10\ \ abc}\rangle\,.r\,[4,3]!_u l2a\,\langle \mathrm{Proposal\ \ 10\ \ abc}\rangle \ldots\ = \mathrm{P}_5\\
&\mid\ ((\mu X)\ldots\ \mid\ (\mu X)\,r\,[4,1]?_w \mathit{Accept}\cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0)\ = \mathrm{A}_1\\
&\mid\ \big((\mu X)\ldots\ \mid\ (\mu X)\,r\,[4,2]?_u l2a\,\langle\bot\rangle\,\big(pr'\big)\,.\,\mathrm{if}\ \ldots\big)\ = \mathrm{A}_2\\
&\mid\ \big((\mu X)\ldots\ \mid\ (\mu X)\,r\,[4,3]?_u l2a\,\langle\bot\rangle\,\big(pr'\big)\,.\,\mathrm{if}\ \ldots\big)\ = \mathrm{A}_3\\
&\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : []\mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : []\mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : []\big)
\end{aligned}
$$

Now $\mathrm{P}_5$ can send its proposal to $\mathrm{A}_2$ and $\mathrm{A}_3$ and terminate. To do so we apply (USend) and (UGet) twice. $\mathrm{A}_2$ and $\mathrm{A}_3$ accept the proposal and the respective subprocesses terminate. Note that we apply (WSkip) in $\mathrm{A}_1$ and terminate that subprocess as well.

$$
\begin{array}{l}
\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,\big( \\
(\mu X)\, s\,[4,1]!_u l1a\,\langle \mathrm{proposalNumber}_4\,(5)\rangle\, .s\,[4,2]!_u l1a\,\langle \mathrm{proposalNumber}_4\,(5)\rangle \ldots = \mathrm{P}_4 \\
\mid\ (\mu X)\, s\,[4,1]?_u l1a\,\langle \bot \rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ldots = \mathrm{A}_1 \\
\mid\ (\mu X)\, s\,[4,2]?_u l1a\,\langle \bot \rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ldots = \mathrm{A}_2 \\
\mid\ (\mu X)\, s\,[4,3]?_u l1a\,\langle \bot \rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ldots = \mathrm{A}_3 \\
\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : []\big)
\end{array}
$$

At this point the local variables of $\mathrm{A}_2$ and $\mathrm{A}_3$ are $n = 10$ and $pr = \mathrm{Proposal}\ \ 10\ \ \mathrm{abc}$. $\mathrm{A}_1$ has not updated its local variables $n = \mathrm{Nothing}$ and $pr = \mathrm{Nothing}$.

### Restarting the Algorithm

Next, $\mathrm{P}_4$ sends prepare requests with a proposal number less than 10, which is rejected by $\mathrm{A}_2$. $\mathrm{P}_4$ then decides to restart the algorithm. We apply $(\mathrm{USend})$ and $(\mathrm{UGet})$ twice. We also apply $(\mathrm{USkip})$ once in $\mathrm{A}_3$.

$$
\begin{array}{l}
\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,\big( \\
(\mu X)\, s\,[1,4]?_u l1b\,\langle \bot \rangle\,(v_1)\,.s\,[2,4]?_u l1b\,\langle \bot \rangle\,(v_2) \ldots = \mathrm{P}_4 \\
\mid\ (\mu X)\,\mathrm{if}\ \ 5 = \bot\,\mathrm{then}\ \ \mathrm{P}_2^{\mathrm{A}}\ \mathrm{else}\ \ldots = \mathrm{A}_1 \\
\mid\ (\mu X)\,\mathrm{if}\ \ 5 = \bot\,\mathrm{then}\ \ \mathrm{P}_2^{\mathrm{A}}\ \mathrm{else}\ \ldots = \mathrm{A}_2 \\
\mid\ (\mu X)\,\mathrm{if}\ \ \bot = \bot\,\mathrm{then}\ \ \mathrm{P}_2^{\mathrm{A}}\ \mathrm{else}\ \ldots = \mathrm{A}_3 \\
\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : []\big)
\end{array}
$$

$\mathrm{A}_3$ moves directly to $\mathrm{P}_2^{\mathrm{A}}$ whereas $\mathrm{A}_1$ and $\mathrm{A}_2$ send their responses to $\mathrm{P}_4$ before moving to $\mathrm{P}_2^{\mathrm{A}}$. $\mathrm{A}_1$ also updates its local variable $n = 5$.

$$
\begin{array}{l}
= (\nu t)\,(\nu s)\,(\nu r)\,\big( \\
(\mu X)\, s\,[1,4]?_u l1b\,\langle \bot \rangle\,(v_1)\,.s\,[2,4]?_u l1b\,\langle \bot \rangle\,(v_2) \ldots = \mathrm{P}_4 \\
\mid\ (\mu X)\, s\,[1,4]!_u l1b\,\langle \mathrm{Promise}\ \ \mathrm{Nothing} \rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\ = \mathrm{A}_1 \\
\mid\ (\mu X)\, s\,[2,4]!_u l1b\,\langle \mathrm{Nack}\ \ 10 \rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\ = \mathrm{A}_2 \\
\mid\ (\mu X)\, r\,[4,3]?_w \mathit{Accept} \cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0 = \mathrm{A}_3 \\
\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l} : []\big)
\end{array}
$$

Applying $(\mathrm{USend})$ and $(\mathrm{UGet})$ twice and evaluating the branching in $\mathrm{P}_4$ yields:

$$\begin{array}{l}
\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
(\mu X)\, s\,[4,\{1,2\}]!_w \mathit{Restart}.X \;=\mathrm{P}_4\\
\mid\ (\mu X)\, s\,[4,1]?_w \mathit{Accept} \cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0 = \mathrm{A}_1\\
\mid\ (\mu X)\, s\,[4,1]?_w \mathit{Accept} \cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0 = \mathrm{A}_2\\
\mid\ (\mu X)\, r\,[4,3]?_w \mathit{Accept} \cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0 = \mathrm{A}_3\\
\mid \Pi_{1\le k,l\le 4,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ t_{k\to l} : []\big)
\end{array}$$

$\mathrm{P}_4$ sends its decision to restart the algorithm to $\mathrm{A}_1$ and $\mathrm{A}_2$ by applying $(\mathrm{WSel})$ once and $(\mathrm{WBran})$ twice. $\mathrm{A}_3$ terminates after applying $(\mathrm{WSkip})$.

$$\begin{array}{l}
\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
(\mu X)\, s\,[4,1]!_u l1a\,\langle 15\rangle\,.s\,[4,2]!_u l1a\,\langle 15\rangle \ldots \;=\mathrm{P}_4\\
\mid\ (\mu X)\, s\,[4,1]?_u l1a\,\langle\bot\rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ \ldots\ =\mathrm{A}_1\\
\mid\ (\mu X)\, s\,[4,2]?_u l1a\,\langle\bot\rangle\,\big(n'\big)\,.\,\mathrm{if}\ \ \ldots\ =\mathrm{A}_2\\
\mid \Pi_{1\le k,l\le 4,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ t_{k\to l} : []\big)
\end{array}$$

### The Happy Path, Again

This time $\mathrm{P}_4$ uses a high enough proposal number so that $\mathrm{A}_1$ and $\mathrm{A}_2$ both promise not to accept any proposal numbered less than that. By applying $(\mathrm{USend})$ and $(\mathrm{UGet})$ and evaluating the branches in the remaining acceptors we arrive at:

$$\begin{array}{l}
\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
(\mu X)\, s\,[1,4]?_u l1b\,\langle\bot\rangle\,(v_1)\,.s\,[2,4]?_u l1b\,\langle\bot\rangle\,(v_2)\,.\,\mathrm{if}\ \ \ldots\ =\mathrm{P}_4\\
\mid\ (\mu X)\, s\,[1,4]!_u l1b\,\langle\mathrm{Promise\ \ Nothing}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\ =\mathrm{A}_1\\
\mid\ (\mu X)\, s\,[2,4]!_u l1b\,\langle\mathrm{Promise\ \ Proposal\ \ 10}\ abc\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\ =\mathrm{A}_2\\
\mid \Pi_{1\le k,l\le 4,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ t_{k\to l} : []\big)
\end{array}$$

Note that, at this point, $\mathrm{A}_1$ and $\mathrm{A}_2$ have updated their respective $n$ to 15.

Because $\mathrm{A}_2$ has already accepted a proposal, it responds to $\mathrm{P}_4$'s prepare request with that proposal. Twice more we apply $(\mathrm{USend})$ and $(\mathrm{UGet})$ and evaluate the branch in $\mathrm{P}_4$ to obtain:

$$\begin{array}{l}
\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
(\mu X)\, s\,[4,\{1,2\}]!_w \mathit{Accept}.\ldots\ =\mathrm{P}_4\\
\mid\ (\mu X)\, s\,[4,1]?_w \mathit{Accept} \cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0 = \mathrm{A}_1\\
\mid\ (\mu X)\, s\,[4,1]?_w \mathit{Accept} \cdots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0 = \mathrm{A}_2\\
\mid \Pi_{1\le k,l\le 4,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ t_{k\to l} : []\big)
\end{array}$$

$\mathrm{P}_4$ has received enough promises to send its own proposal. The value for that proposal is $\mathrm{abc}$ because that is the value of the highest-numbered proposal $\mathrm{P}_4$ received as a response to its prepare request. First, we apply $(\mathrm{WSel})$ and $(\mathrm{WBran})$.

$$\begin{aligned}
&\longmapsto^* (\nu t)\,(\nu s)\,(\nu r)\,\big(\\
&(\mu X)\, s\,[4,1]!_u l2a\,\langle \mathrm{Proposal}\ \ 15\ \ \mathrm{abc}\rangle\,.s\,[4,2]!_u l2a\,\langle \mathrm{Proposal}\ \ 15\ \ \mathrm{abc}\rangle\,.0 = \mathrm{P}_4\\
&\mid\ (\mu X)\, s\,[4,1]?_u l2a\,\langle\perp\rangle\,\big(pr'\big)\,.\,\mathrm{if}\ \ \ldots\ = \mathrm{A}_1\\
&\mid\ (\mu X)\, s\,[4,2]?_u l2a\,\langle\perp\rangle\,\big(pr'\big)\,.\,\mathrm{if}\ \ \ldots\ = \mathrm{A}_2\\
&\mid \Pi_{1\le k,l\le 4,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ t_{k\to l} : []\big)
\end{aligned}$$

Then we apply $(\mathrm{USend})$ and $(\mathrm{UGet})$ to send the proposal from $\mathrm{P}_4$ to the acceptors. $\mathrm{P}_4$ terminates and the acceptors accept the received proposal and then terminate as well.

$$\longmapsto^*\ (\nu t)\,(\nu s)\,(\nu r)\,\big(\Pi_{1\le k,l\le 4,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 4,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ t_{k\to l} : []\big)$$

Afterwards $\mathrm{A}_1$ and $\mathrm{A}_2$ have $n = 15$ and $pr = \mathrm{Proposal}\ \ 15\ \ \mathrm{abc}$ and $\mathrm{A}_3$ has $n = 10$ and $pr = \mathrm{Proposal}\ \ 10\ \ \mathrm{abc}$. All acceptors have accepted the value $\mathrm{abc}$.

# 4 Analysis

We take the model from the previous chapter, type-check it, and discuss what the type check means for agreement, validity, and termination of the Paxos algorithm. To execute the type check we project the global type to local types and use the typing rules given in [5] to prove that our model is well-typed.

## 4.1 Local Types

Because no communication takes place in the outer session, the outer session's type is $\mathrm{G} = 0$. Every projection of G to a local type is $\mathrm{G} \restriction_k = 0$ for every $k$.

For $1 \leq a \leq c_A$ and $c_A + 1 \leq p \leq c_A + c_P$ we define the projections of the global type $\mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}}$.

$$\begin{gathered}\mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}} \restriction_p = (\mu x) \left( \bigodot_{a \in A_Q} \ [a]!_u l1a \, \langle \mathbb{N} \rangle \right) . \left( \bigodot_{a \in A_Q} \ [a]?_u l1b \, \langle \mathrm{Promise\ \ Value} \rangle \right) . \\ \left( [A_Q]!_w \mathit{Accept}. \left( \bigodot_{a \in A_Q} \ [a]!_u l2a \, \langle \mathrm{Proposal\ \ Value} \rangle \right) .0 \oplus \mathit{Restart}.x \oplus \mathit{Abort}.0 \right)\end{gathered}$$

$\mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}} \restriction_p$ defines the local type for proposers. First, the proposer sends a proposal number to all acceptors in its quorum in phase $1a$. It receives their responses in phase $1b$ and then branches in phase $2a$. We can see that the proposer communicates with all acceptors in its quorum in every phase.

$$\begin{gathered}\mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}} \restriction_a = (\mu x) \, [p]?_u l1a \, \langle \mathbb{N} \rangle \, . \, [p]!_u l1b \, \langle \mathrm{Promise\ \ Value} \rangle \, . \\ ([p]?_w \mathit{Accept}. \, [p]?_u l2a \, \langle \mathrm{Proposal\ \ Value} \rangle \, .0 \oplus \mathit{Restart}.x \oplus \mathit{Abort}.0)\end{gathered}$$

$\mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}} \restriction_a$ defines the local type for acceptors, assuming a proposer $p$. Since Paxos defines two roles that communicate with each other, their local types complement each other. An acceptor first receives a proposal number, then it responds with a $\mathrm{Promise\ \ Value}$. Finally, it receives the proposer's branching choice.

## 4.2 Type Check

$$\Gamma = o : \mathrm{G} \cdot b_{c_A+1} : \mathrm{G}_{\mathrm{p},\mathrm{A}_{\mathrm{Q}}} \cdot b_{c_A+2} : \mathrm{G}_{\mathrm{p},\mathrm{A}_{\mathrm{Q}}} \cdot \ldots \cdot b_{c_A+c_P} : \mathrm{G}_{\mathrm{p},\mathrm{A}_{\mathrm{Q}}} \cdot c_A : \mathbb{N} \cdot c_P : \mathbb{N}$$

$\Gamma$ contains the type for our shared-points $o$ and $b_n$ where $c_A + 1 \leq n \leq c_A + c_P$.

We start the type-check with the global environment $\Gamma$ and the entry-point of the model $\mathrm{Sys}\,(c_A, c_P)$. Then, we apply the typing rules in [5] in a proof tree and show that the model can be derived from the axioms $(\mathrm{Var})$ and $(\mathrm{End})$.

### 4.2.1 System Initialization

$$\dfrac{\dfrac{(S_1)}{\Gamma \vdash \overline{o}\,[2]\,(t)\ldots \rhd \emptyset} \qquad \dfrac{\dfrac{(S_2)}{\Gamma \vdash o\,[1]\,(t)\ldots \rhd \emptyset} \qquad \dfrac{(S_3)}{\Gamma \vdash \Pi_{1\leq a\leq c_A}\ \mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\,(\ldots) \rhd \emptyset}}{\Gamma \vdash o\,[1]\,(t)\ldots \mid \Pi_{1\leq a\leq c_A}\ \mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\,(\ldots) \rhd \emptyset}\,(\mathrm{Par})}{\Gamma \vdash \overline{o}\,[2]\,(t)\,.\,\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(\ldots)\ \mid o\,[1]\,(t)\ldots \mid \Pi_{1\leq a\leq c_A}\ \mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\,(\ldots) \rhd \emptyset}\,(\mathrm{Par})$$

We apply $(\mathrm{Par})$ twice and split off into three sub-proofs $(S_1)$, $(S_2)$, and $(S_3)$.

$$(S_1) = \dfrac{\dfrac{(P)}{\Gamma \vdash \overline{b_{c_A+c_P}}\,[c_A+1]\,(s)\,.\,\mathrm{P}^{\mathrm{P}} \rhd t\,[2] : \mathrm{G} \upharpoonright_2}}{\Gamma \vdash \overline{o}\,[2]\,(t)\,.\,\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(c_A+1, \mathrm{genA}_{\mathrm{Q}}\,(c_A+c_P, c_A, c_P)\,, c_A+c_P, c_A+c_P, []) \rhd \emptyset}\,(\mathrm{Rec})$$

In $(S_1)$ we apply $(\mathrm{Rec})$ once and defer the rest of the proof-tree. Note that, since $\mathrm{G} \upharpoonright_2 = 0$, $t\,[2] : \mathrm{G} \upharpoonright_2 = \emptyset$. This is relevant later when continuing $(P)$.

$$(S_2) = \dfrac{\dfrac{\dfrac{(P)}{\Gamma \vdash \overline{b_{c_A+1}}\,[c_A+1]\,(s)\,.\,\mathrm{P}^{\mathrm{P}} \rhd \emptyset} \qquad \ldots \qquad \dfrac{(P)}{\Gamma \vdash \overline{b_{c_A+c_P-1}}\,[c_A+1]\,(s)\,.\,\mathrm{P}^{\mathrm{P}} \rhd \emptyset}}{\Gamma \vdash \Pi_{c_A<k<c_A+c_P}\ \mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(c_A+1, \mathrm{genA}_{\mathrm{Q}}\,(k, c_A, c_P)\,, k, k, []) \rhd t\,[1] : G \upharpoonright_1}\,(\mathrm{Par})^{c_P-1}}{\Gamma \vdash o\,[1]\,(t)\,.\Pi_{c_A<k<c_A+c_P}\ \mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(\ldots) \rhd \emptyset}\,(\mathrm{Acc})$$

Applying $(\mathrm{Acc})$ in $(S_2)$ requires that $o : \mathrm{G} \in \Gamma$. $(\mathrm{Par})$ is applied $c_P - 1$ times to separate all the proposer processes. Each individual proposer can be type-checked with the same proof-tree $(P)$. Because $\mathrm{G} \upharpoonright_1 = 0$, $t\,[1] : \mathrm{G} \upharpoonright_1 = \emptyset$. The session environment $\Delta$ in $(P)$ is empty for every proposer.

$$(S_3) = \cfrac{\cfrac{\cfrac{\cfrac{(A_1)}{\Gamma \vdash \mathrm{P}^{\mathrm{A}}_{1} \triangleright s\,[a] : \mathrm{G}_{\mathrm{p},\mathrm{A}_{\mathrm{Q}}} \upharpoonright_{a}}}{\Gamma \vdash b_k\,[a]\,(s)\,.\,\mathrm{P}^{\mathrm{A}}_{1} \triangleright \emptyset}\,(\mathrm{Acc}) \qquad \dots}{\Gamma \vdash \Pi_{c_A<k\leq c_A+c_P}\ b_k\,[c_A]\,(s)\,.\,\mathrm{P}^{\mathrm{A}}_{1} \triangleright \emptyset}\,(\mathrm{Par})^{c_P} \qquad \dots}{\Gamma \vdash \Pi_{1\leq j\leq c_A}\ \left(\Pi_{c_A<k\leq c_A+c_P}\ b_k\,[j]\,(s)\,.\,\mathrm{P}^{\mathrm{A}}_{1}\right) \triangleright \emptyset}\,(\mathrm{Par})^{c_A}$$

$(\mathrm{Par})$ is applied $c_A$ times to separate the individual acceptors and then $c_P$ times for each acceptor to separate the individual subprocesses. Since every subprocess of every acceptor behaves like $\mathrm{P}^{\mathrm{A}}_{1}$ and has the same local type, the same proof-tree $(A_1)$ can be applied. Applying $(\mathrm{Acc})$ to every subprocess of every acceptor requires $\forall k \in \mathbb{N} : (c_A + 1 \leq k \wedge k \leq c_A + c_P) \rightarrow b_k : \mathrm{G}_{\mathrm{p},\mathrm{A}_{\mathrm{Q}}} \in \Gamma$.

Note that only one acceptor and one of its subprocesses is shown in $(S_3)$. The rest has been left out to improve readability.

### 4.2.2 Proposer

Let $p = c_A + 1, A_Q = \mathrm{genA_Q}\,(k, c_A, c_P)\,, n = k, m = k, \overrightarrow{V} = []$ where $c_A < k \leq c_A + c_P$. This gives us the values for the arguments of $\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}$. We observe that $\Gamma \Vdash p : \mathbb{N}$, $\Gamma \Vdash k : \mathbb{N}$, $\Gamma \Vdash n : \mathbb{N}$, $\Gamma \Vdash m : \mathbb{N}$, and $\Gamma \Vdash A_Q : \texttt{list of}\ \mathbb{N}$. $p$, $k$, $n$, and $m$ are natural numbers and $A_Q$ is a list of natural numbers under global environment $\Gamma$.

To abbreviate the proposer's local type in the following proof-trees we define the following sub-formulae.

$$\mathrm{T}^{\mathrm{P}}_{\mathrm{acc}} = \left(\bigodot_{a\in A_Q}\ [a]!_{u}l2a\,\langle\mathrm{Proposal\ \ Value}\rangle\right).0$$

$$\mathrm{T}^{\mathrm{P}}_{\mathrm{branch}} = \left([A_Q]!_{w}\mathit{Accept}.\,\mathrm{T}^{\mathrm{P}}_{\mathrm{acc}} \oplus \mathit{Restart}.x \oplus \mathit{Abort}.0\right)$$

Note that $\mathrm{G}_{\mathrm{p},\mathrm{A}_{\mathrm{Q}}} \upharpoonright_{p}= (\mu x)\left(\bigodot_{a\in A_Q}\ [a]!_{u}l1a\,\langle\mathbb{N}\rangle\right).\left(\bigodot_{a\in A_Q}\ [a]?_{u}l1b\,\langle\mathrm{Promise\ \ Value}\rangle\right).\mathrm{T}^{\mathrm{P}}_{\mathrm{branch}}$.

In order to shorten the proposer's process we define some variables.

$$e = \mathrm{anyNack}\left(\overrightarrow{V}\right)\ \text{ or }\ \mathrm{promiseCount}\left(\overrightarrow{V}\right) < \left\lceil\frac{p}{2}\right\rceil$$

$$pn = \mathrm{proposalNumber}_{m}\,(n)$$

$$\mathit{prop} = \mathrm{Proposal\ \ proposalNumber}_{m}\,(n)\ \ \mathrm{promiseValue}\left(\overrightarrow{V}\right)$$

The actual values of $e$, $pn$, and $\mathit{prop}$ are not relevant for the type check. We observe that $\Gamma \Vdash e : \mathrm{Bool}$, $\Gamma \Vdash pn : \mathbb{N}$, and $\Gamma \Vdash \mathit{prop} : \mathrm{Proposal\ \ Value}$.

To further abbreviate the terms in the proof-trees we define two global environments $\Gamma'$ and $\Gamma''$.

$$\Gamma' = \Gamma \cdot X : x$$

$\Gamma'$ contains $\Gamma$ and a type for the recursion variable $X$.

$$\Gamma'' = \Gamma' \cdot v_a : \text{Promise Value}, \forall a \in A_Q$$

$\Gamma''$ contains $\Gamma'$ and types for the entries of $\overrightarrow{V}$. These are added to the global environment when applying $(\text{UGet})$ in phase $1b$.

$$(P) = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(P_t)}{\Gamma'' \vdash s\,[p, A_Q]!_w Restart.X \,\triangleright\, s\,[p] : \mathrm{T}^{\mathrm{P}}_{\text{branch}}} \qquad \dfrac{(P_f)}{\Gamma'' \vdash s\,[p, A_Q]!_w Accept.\ldots \,\triangleright\, s\,[p] : \mathrm{T}^{\mathrm{P}}_{\text{branch}}}}{\Gamma'' \vdash \text{if } \ e \,\text{then } \ s\,[p, A_Q]!_w Restart.X \text{ else } \ s\,[p, A_Q]!_w Accept.\ldots \,\triangleright\, s\,[p] : \mathrm{T}^{\mathrm{P}}_{\text{branch}}}\,(\text{If})}{\Gamma' \vdash \left(\bigodot_{a \in A_Q} \ s\,[a, p]?_u l1b\,\langle \bot \rangle\,(v_a)\right)\ldots \,\triangleright\, s\,[p] : \left(\bigodot_{a \in A_Q} \ [a]?_u l1b\,\langle \text{Promise Value} \rangle\right)}\,(\text{UGet})^{|A_Q|}}{\Gamma' \vdash \left(\bigodot_{a \in A_Q} \ s\,[p, a]!_u l1a\,\langle pn \rangle\right)\ldots \,\triangleright\, s\,[p] : \left(\bigodot_{a \in A_Q} \ [a]!_u l1a\,\langle \mathbb{N} \rangle\right)\ldots}\,(\text{USend})^{|A_Q|}}{\Gamma' \vdash \text{update}\,(n, n+1)\ldots \,\triangleright\, s\,[p] : \left(\bigodot_{a \in A_Q} \ [a]!_u l1a\,\langle \mathbb{N} \rangle\right)\ldots}\,(??)}{\Gamma \vdash (\mu X)\,\text{update}\,(n, n+1)\ldots \,\triangleright\, s\,[p] : (\mu x)\left(\bigodot_{a \in A_Q} \ [a]!_u l1a\,\langle \mathbb{N} \rangle\right)\ldots}\,(\text{Rec})}{\Gamma \vdash \overline{b_n}\,[p]\,(s)\,.\,\mathrm{P}^{\mathrm{P}} \,\triangleright\, \emptyset}\,(\text{Req})$$

$(P)$ is the continuation of $(S_1)$ and $(S_2)$. In both proof-trees the session environment $\Delta$ was empty. Here, we apply $(\text{Req})$ and add $s\,[p] : \mathrm{G}_{\mathrm{p},\mathrm{A_Q}} \upharpoonright_p$ to the session environment. Applying $(\text{Rec})$ changes the global environment from $\Gamma$ to $\Gamma'$. $(??)$ only changes the process and lets us continue. First $(\text{USend})$ and then $(\text{UGet})$ is applied for every acceptor in $A_Q$. $(\text{UGet})$ expands the session environment to $\Gamma''$. $(\text{If})$ splits the proof-tree into $(P_t)$ and $(P_f)$.

$$(P_t) = \dfrac{\dfrac{}{\Gamma'' \vdash X \,\triangleright\, s\,[p] : x}\,(\text{Var})}{\Gamma'' \vdash s\,[p, A_Q]!_w Restart.X \,\triangleright\, s\,[p] : \left([A_Q]!_w Accept.\, \mathrm{T}^{\mathrm{P}}_{\text{acc}} \oplus Restart.x \oplus Abort.0\right)}\,(\text{WSel})$$

We apply $(\text{WSel})$ and then $(\text{Var})$ to finish $(P_t)$.

$$(P_f) = \dfrac{\dfrac{\dfrac{}{\Gamma'' \vdash 0 \,\triangleright\, s\,[p] : 0}\,(\text{End})}{\Gamma'' \vdash \left(\bigodot_{a \in A_Q} \ s\,[p, a]!_u l2a\,\langle prop \rangle\right).0 \,\triangleright\, s\,[p] : \left(\bigodot_{a \in A_Q} \ [a]!_u l2a\,\langle \text{Proposal Value} \rangle\right).0}\,(\text{USend})^{|A_Q|}}{\Gamma'' \vdash s\,[p, A_Q]!_w Accept.\ldots. \,\triangleright\, s\,[p] : \left([A_Q]!_w Accept.\, \mathrm{T}^{\mathrm{P}}_{\text{acc}} \oplus Restart.x \oplus Abort.0\right)}\,(\text{WSel})$$

After applying $(\text{USend})$ we can apply $(\text{USend})$ once for every acceptor in $A_Q$. Finally, we can finish $(P_f)$ — and with it $(P)$ — by applying $(\text{End})$.

### 4.2.3 Acceptor

First, we define the arguments of $\mathrm{P}^{\mathrm{A}}_{\mathrm{init}}$ and $\mathrm{P}^{\mathrm{A}}_{1}$. Let $a = j$ and $p = c_A + 1$ where $1 \leq j \leq c_A$. With session environment $\Gamma$ we have $\Gamma \Vdash a : \mathbb{N}$ and $\Gamma \Vdash p : \mathbb{N}$.

To improve readability of the proof-trees we break down the acceptor's process and local type.

$$\begin{aligned}
&\mathrm{P}^{\mathrm{A}}_{\mathrm{acc}} = s\,[p,a]?_u l2a\,\langle\bot\rangle\,\left(pr'\right).\,\mathrm{if}\ \ pr' = \bot\\
&\ \ \mathrm{then}\ \ 0\\
&\ \ \mathrm{else}\ \ \mathrm{if}\ \ \mathrm{ge}\left(\mathrm{nFromProposal}\left(pr'\right),n\right)\\
&\ \ \ \ \mathrm{then}\ \ \mathrm{update}\left(pr,pr'\right).\,\mathrm{update}\left(n,\mathrm{Just}\ \ \mathrm{nFromProposal}\left(pr'\right)\right).0\\
&\ \ \ \ \mathrm{else}\ \ 0
\end{aligned}$$

We can see that $\mathrm{P}^{\mathrm{A}}_{2}$ contains $\mathrm{P}^{\mathrm{A}}_{\mathrm{acc}}$ as $\mathrm{P}^{\mathrm{A}}_{2} = s\,[p,a]?_w Accept.\,\mathrm{P}^{\mathrm{A}}_{\mathrm{acc}} \oplus Restart.X \oplus Abort.0$.

$$\mathrm{P}^{\mathrm{A}}_{\mathrm{t}} = \mathrm{update}\left(n,n'\right).s\,[a,p]!_u l1b\,\langle\mathrm{Promise}\ \ pr\rangle\,.\,\mathrm{P}^{\mathrm{A}}_{2}$$

$$\mathrm{P}^{\mathrm{A}}_{\mathrm{f}} = s\,[a,p]!_u l1b\,\langle\mathrm{Nack}\ \ n\rangle\,.\,\mathrm{P}^{\mathrm{A}}_{2}$$

$$\mathrm{P}^{\mathrm{A}}_{\mathrm{gt}} = \mathrm{if}\ \ \mathrm{gt}\left(n',n\right)\,\mathrm{then}\ \ \mathrm{P}^{\mathrm{A}}_{\mathrm{t}}\,\mathrm{else}\ \ \mathrm{P}^{\mathrm{A}}_{\mathrm{f}}$$

With $\mathrm{P}^{\mathrm{A}}_{\mathrm{gt}}$, $\mathrm{P}^{\mathrm{A}}_{1}$ can be written as $\mathrm{P}^{\mathrm{A}}_{1} = (\mu X)\,s\,[p,a]?_u l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ \ n' = \bot\,\mathrm{then}\ \ s\,[a,p]!_u l1b\,\langle\bot\rangle\,.\,\mathrm{P}^{\mathrm{A}}_{2}\,\mathrm{else}\ \ \mathrm{P}^{\mathrm{A}}_{\mathrm{gt}}$.

$$\mathrm{T}^{\mathrm{A}}_{\mathrm{acc}} = [p]?_u l2a\,\langle\mathrm{Proposal}\ \ \mathrm{Value}\rangle\,.0$$

$$\mathrm{T}^{\mathrm{A}}_{\mathrm{branch}} = \left([p]?_w Accept.\,\mathrm{T}^{\mathrm{A}}_{\mathrm{acc}} \oplus Restart.x \oplus Abort.0\right)$$

$$\mathrm{T}^{\mathrm{A}}_{\mathrm{1b}} = [p]!_u l1b\,\langle\mathrm{Promise}\ \ \mathrm{Value}\rangle\,.\,\mathrm{T}^{\mathrm{A}}_{\mathrm{branch}}$$

The acceptor's local type $\mathrm{G}_{\mathrm{p},\mathrm{A}_{\mathrm{Q}}} \upharpoonright_a$ can be written as $\mathrm{G}_{\mathrm{p},\mathrm{A}_{\mathrm{Q}}} \upharpoonright_a = (\mu x)\,[p]?_u l1a\,\langle\mathbb{N}\rangle\,.\,\mathrm{T}^{\mathrm{A}}_{\mathrm{1b}}$.

Finally, we define the global environments $\Gamma'$, $\Gamma''$, and $\Gamma'''$.

$$\begin{aligned}
&\Gamma' = \Gamma \cdot X : x\\
&\Gamma'' = \Gamma' \cdot n' : \mathbb{N}\\
&\Gamma''' = \Gamma'' \cdot pr' : \mathrm{Proposal}\ \ \mathrm{Value}
\end{aligned}$$

$\Gamma'$ contains $\Gamma$ and assigns the type $x$ to $X$. $\Gamma''$ additionally maps $n'$ to type $\mathbb{N}$. $\Gamma'''$ adds type $\mathrm{Proposal}\ \ \mathrm{Value}$ for $pr'$.

$$(A_1) = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A_2)}{\Gamma'' \vdash \mathrm{P}^{\mathrm{A}}_{2} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{branch}}}}{\Gamma'' \vdash s\,[a,p]!_u l1b\,\langle\bot\rangle\,.\,\mathrm{P}^{\mathrm{A}}_{2} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{1b}}}\;(\text{USend}) \qquad \dfrac{(A_{gt})}{\Gamma'' \vdash \mathrm{P}^{\mathrm{A}}_{\mathrm{gt}} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{1b}}}}{\Gamma'' \vdash \text{if } \ n' = \bot \text{ then } \ s\,[a,p]!_u l1b\,\langle\bot\rangle\,.\,\mathrm{P}^{\mathrm{A}}_{2} \text{ else } \text{ if } \ \mathrm{gt}\,(n',n)\ldots \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{1b}}}\;(\text{If})}{\Gamma' \vdash s\,[p,a]?_u l1a\,\langle\bot\rangle\,(n')\,.\,\text{if } \ n' = \bot \ldots \triangleright s\,[a] : [p]?_u l1a\,\langle\mathbb{N}\rangle\,.\,\mathrm{T}^{\mathrm{A}}_{\mathrm{1b}}}\;(\text{UGet})}{\Gamma \vdash (\mu X)\,s\,[p,a]?_u l1a\,\langle\bot\rangle\,(n')\ldots. \triangleright s\,[a] : (\mu x)\,[p]?_u l1a\,\langle\mathbb{N}\rangle\ldots.}\;(\text{Rec})$$

After applying $(\mathrm{Acc})$ in $(S_3)$ the session environment contains the acceptor's local type $\mathrm{G}_{\mathrm{p,A_Q}} \upharpoonright_a$. We apply $(\mathrm{Rec})$ and $(\mathrm{UGet})$ and the split the proof-tree with $(\mathrm{If})$. By applying $(\mathrm{Rec})$ and $(\mathrm{UGet})$ the global environment expands from $\Gamma$ to $\Gamma'$ to $\Gamma''$. On the left branch we apply $(\mathrm{USend})$ and defer to $(A_2)$. The right branch is deferred to $(A_{gt})$.

Since the process of the right branch contains an if-then-else and unreliable-send statements before continuing to $\mathrm{P}^{\mathrm{A}}_{2}$, we will examine this branch first. Much like the left branch, the proof-tree of the right branch can later be deferred to $(A_2)$.

$$(A_{gt}) = \dfrac{\dfrac{(A_t)}{\Gamma'' \vdash \mathrm{P}^{\mathrm{A}}_{\mathrm{t}} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{1b}}} \qquad \dfrac{(A_f)}{\Gamma'' \vdash \mathrm{P}^{\mathrm{A}}_{\mathrm{f}} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{1b}}}}{\Gamma'' \vdash \text{if } \ \mathrm{gt}\,(n',n) \text{ then } \ \mathrm{P}^{\mathrm{A}}_{\mathrm{t}} \text{ else } \ \mathrm{P}^{\mathrm{A}}_{\mathrm{f}} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{1b}}}\;(\text{If})$$

First, we split the proof-tree with $(\mathrm{If})$. We defer the resulting branches to separate proof-trees $(A_t)$ and $(A_f)$.

$$(A_t) = \dfrac{\dfrac{(A_2)}{\Gamma'' \vdash \mathrm{P}^{\mathrm{A}}_{2} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{branch}}}}{\Gamma'' \vdash s\,[a,p]!_u l1b\,\langle\mathrm{Nack} \ \ n\rangle\,.\,\mathrm{P}^{\mathrm{A}}_{2} \triangleright s\,[a] : [p]!_u l1b\,\langle\mathrm{Promise} \ \ \mathrm{Value}\rangle\,.\,\mathrm{T}^{\mathrm{A}}_{\mathrm{branch}}}\;(\text{USend})$$

$$(A_f) = \dfrac{\dfrac{(A_2)}{\Gamma'' \vdash \mathrm{P}^{\mathrm{A}}_{2} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{branch}}}}{\Gamma'' \vdash s\,[a,p]!_u l1b\,\langle\mathrm{Nack} \ \ n\rangle\,.\,\mathrm{P}^{\mathrm{A}}_{2} \triangleright s\,[a] : [p]!_u l1b\,\langle\mathrm{Promise} \ \ \mathrm{Value}\rangle\,.\,\mathrm{T}^{\mathrm{A}}_{\mathrm{branch}}}\;(\text{USend})$$

In both, $(A_t)$ and $(A_f)$, we apply $(\mathrm{USend})$. Now we can defer to $(A_2)$, which is the proof-tree for $\mathrm{P}^{\mathrm{A}}_{2}$.

$$(A_2) = \dfrac{\dfrac{(A_{Accept})}{\Gamma'' \vdash \mathrm{P}^{\mathrm{A}}_{\mathrm{acc}} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{acc}}} \qquad \dfrac{}{\Gamma'' \vdash X \triangleright s\,[a] : x}\;(\text{Var}) \qquad \dfrac{}{\Gamma'' \vdash 0 \triangleright s\,[a] : 0}\;(\text{End})}{\Gamma'' \vdash \mathrm{P}^{\mathrm{A}}_{2} \triangleright s\,[a] : \mathrm{T}^{\mathrm{A}}_{\mathrm{branch}}}\;(\text{WBran})$$

By applying $(\mathrm{WBran})$ we separate the three branches. From left to right we get an *Accept*-, a *Restart*-, and an *Abort*-branch. We defer the *Accept*-branch to $(A_{Accept})$. The *Restart*-branch can be finished by applying $(\mathrm{Var})$ and the *Abort*-branch by applying $(\mathrm{End})$.

$$(A_{Accept}) = \cfrac{\cfrac{\cfrac{}{\Gamma''' \vdash 0 \rhd s\,[a] : 0}\,(\mathrm{End}) \qquad \cfrac{\cfrac{(A_{update})}{\Gamma''' \vdash \mathrm{update}\,(pr, pr')\,.\,.\,.\,. \rhd s\,[a] : 0} \qquad \cfrac{}{\Gamma''' \vdash 0 \rhd s\,[a] : 0}\,(\mathrm{End})}{\Gamma''' \vdash \mathrm{if}\ \ \mathrm{ge}\,(\mathrm{nFromProposal}\,(pr')\,, n)\,\mathrm{then}\ \ .\,.\,.\,\mathrm{else}\ \ 0 \rhd s\,[a] : 0}\,(\mathrm{If})}{\Gamma''' \vdash \mathrm{if}\ \ pr' = \bot\,\mathrm{then}\ \ 0\,\mathrm{else}\ \ .\,.\,. \rhd s\,[a] : 0}\,(\mathrm{If})}{\Gamma'' \vdash s\,[p, a]?_u l2a\,\langle\bot\rangle\,(pr')\,.\,.\,.\,. \rhd s\,[a] : [p]?_u l2a\,\langle \mathrm{Proposal}\ \ \mathrm{Value}\rangle\,.0}\,(\mathrm{UGet})$$

We apply $(\mathrm{UGet})$ and expand the global session to $\Gamma'''$. The proof-tree is split twice by applying $(\mathrm{If})$ twice. The right-most and left-most proof-trees are finished by applying $(\mathrm{End})$. We defer the proof in the middle to keep $(A_{Accept})$ readable.

$$(A_{update}) = \cfrac{\cfrac{\cfrac{}{\Gamma''' \vdash 0 \rhd s\,[a] : 0}\,(\mathrm{End})}{\Gamma''' \vdash \mathrm{update}\,(n, \mathrm{Just}\ \ \mathrm{nFromProposal}\,(pr'))\,.0 \rhd s\,[a] : 0}\,(??)}{\Gamma''' \vdash \mathrm{update}\,(pr, pr')\,.\,.\,.\,. \rhd s\,[a] : 0}\,(??)$$

Finally, we apply $(??)$ twice and $(\mathrm{End})$ once. This concludes the type check and proves that the model is well-typed.

## 4.3 Termination, Agreement, Validity

### 4.3.1 Termination

The global type and well-typedness ensure the absence of deadlocks. This means that the processes either loop forever or terminate. Acceptors terminate if all of their sub-processes terminate. Each sub-process of an acceptor corresponds to one proposer. A sub-process can only terminate via the weakly reliable broadcast in $\mathrm{P}_2^{\mathrm{A}}$, which depends on the corresponding proposer. If that proposer crashes or its quorum does not include the acceptor, the sub-process terminates because $\mathtt{FP}_{\mathtt{wskip}}$ returns $\mathrm{true}$ and the default branch is *Abort*, which terminates immediately. The termination of a sub-process with a correct proposer requires the termination of that proposer. Thus, we need to prove that correct proposers terminate to prove termination for our model.

If the set of acceptors in a proposer's quorum $A_Q$ that are correct is not enough to form a majority of acceptors, that proposer repeatedly restarts the algorithm. In this case the proposer will be unable to issue a valid proposal. Because $\mathtt{FP}_{\mathtt{crash}}$ returns $\mathrm{true}$ if $A_Q \setminus \mathbb{F}$, where $\mathbb{F}$ is the set of processes permanently suspected by a failure detector in $\Diamond\mathscr{S}$, is not a quorum, the proposer eventually crashes. Proposers either complete the Paxos algorithm after phase $2a$ or crash.

In [4] Lamport describes a scenario in which two proposers loop endlessly, never having their proposals accepted: Proposer $p$ completes phase 1 for a proposal number $n_1$. Another proposer $q$ then completes phase 1 for a proposal number $n_2 > n_1$. Proposer $p$'s phase 2 accept requests for a proposal numbered $n_1$ are ignored because the acceptors have all promised not to accept any new proposal numbered less than $n_2$. So, proposer $p$ then begins and completes phase 1 for a new proposal number $n_3 > n_2$, causing the second phase 2 accept requests of proposer $q$ to be ignored. And so on.

From [4] we know that this problem is solved by electing a single distinguished proposer to be the leader. The leader eventually picks a proposal number high enough for its proposal to be accepted. The model assumes some sort of leader selection. A new leader is elected when the previous leader terminates.

### 4.3.2 Agreement

Any proposer $p$ requires that the set of correct acceptors in its quorum of acceptors is itself a quorum, i.e. an accepting set with which a value can be chosen [3]. Should message loss occur in labels $l1a$ or $l1b$, $p$ restarts the algorithm. This broadcast is weakly reliable and thus only fails when $p$ crashes or terminates because $\mathtt{FP}_{\mathtt{wskip}}$ disallows suspicion of correct live proposers in acceptors. Given the definition of $\mathrm{promiseValue}$, $p$ will only propose a fresh value if none of the acceptors have accepted a proposal yet. At least one acceptor in every other proposer's quorum is contained in $p$'s quorum. Thus, if a majority of acceptors accept $p$'s proposal it is sent to every other proposer when they reach phase $1b$. These proposers then propagate the accepted value by proposing it again but with a higher proposal number. This way all correct acceptors accept the same value.

### 4.3.3 Validity

To prove validity for our model we examine the communication structure and the origins of the accepted values. Because the model is well-typed we know the communication structure is as specified in global type $\mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}}$. From [5] we know that validity then holds globally if it holds for each local process.

Labels $l1b$ and $l2a$ are used to send values that can be accepted.

Label $l1b$ is used to send messages of sort $\mathrm{Promise}\ \ \mathrm{Value}$. These messages are sent from the acceptors to a proposer and may contain the acceptors' accepted proposal. Should an acceptor previously have accepted a proposal, that proposal then contains the accepted value. The accepted proposal $pr$ is either the acceptor's initial accepted proposal $pr_a$ or a proposal that was previously proposed by a proposer. $pr$ is sent over $l1b$ without alteration. The proposer receiving these messages stores them in $\overrightarrow{V}$ without changing their values.

Proposers send a message of sort $\mathrm{Proposal}\ \ \mathrm{Value}$ to their quorum of acceptors over label $l2a$. To do so, proposers pick the best value from a proposal in $\overrightarrow{V}$, if any is available, with $\mathrm{promiseValue}$. An entry in $\overrightarrow{V}$ contains a proposal $prop$ if it is of the form $\mathrm{Promise}\ \ \mathrm{Just}\ \ prop$. These proposals are either some acceptor's initial accepted proposal or a proposal proposed by a proposer. Not all entries in $\overrightarrow{V}$ contain a proposal but if at least one does, $\mathrm{promiseValue}$ returns the value of one of them. If no entry in $\overrightarrow{V}$ contains a proposal a fresh value is chosen and returned. In both cases the return value of $\mathrm{promiseValue}$ is not altered before being sent over label $l2a$, which constitutes proposing that value. Acceptors that receive and accept this proposal store it without alteration.

Since label $l1a$ is not used to transmit values that can be accepted, we conclude that validity holds for each local process and thus globally.

# Bibliography

[1] Tushar Deepak Chandra and Sam Toueg. “Unreliable Failure Detectors for Reliable Distributed Systems”. In: *J. ACM* 43.2 (Mar. 1996), pp. 225–267. ISSN: 0004-5411. DOI: `10.1145/226643.226647`. URL: `https://doi.org/10.1145/226643.226647`.

[2] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (3rd Edition)*. Addison-Wesley, 2001, p. 452.

[3] Leslie Lamport. “Lower Bounds for Asynchronous Consensus”. In: *Distrib. Comput.* 19.2 (Oct. 2006), pp. 104–125. ISSN: 0178-2770. DOI: `10.1007/s00446-006-0155-x`. URL: `https://doi.org/10.1007/s00446-006-0155-x`.

[4] Leslie Lamport. “Paxos Made Simple”. In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (Dec. 2001), pp. 51–58. URL: `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

[5] K. Peters, U. Nestmann, and C. Wagner. “Fault-Tolerant Multiparty Session Types”. Provided by K. Peters. 2021.

[6] A. Scalas and N. Yoshida. “Multiparty session types, beyond duality”. In: *Journal of Logical and Algebraic Methods in Programming* 97 (2018), pp. 55–84.