Analyzing Paxos with Fault-Tolerant Multiparty Session Types

Bachelor thesis by Nicolas Daniel Torres
Date of submission: February 22, 2022

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Theory of Parallel Systems

Contents

1 Introduction

In this work we use Fault-Tolerant Multiparty Session Types (FTMPST) to model and analyze the Paxos consensus algorithm.

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. For example, in state machine replication or when deciding which database transactions should be committed in what order.

Consensus algorithms solve this consensus problem. These algorithms must satisfy validity, agreement, and termination to achieve consensus among multiple agents [4].

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 [12].

Since agents and communication between agents may fail, consensus algorithms are designed to be fault-tolerant, but modelling fault-tolerance is not possible with Multiparty Session Types. Peters, Nestmann, and Wagner extended Multiparty Session Types with fault-tolerance to create FTMPST.

Our goal is to provide an analysis of a consensus algorithm using FTMPST. Specifically, we use FTMPST to construct a model of Paxos as described by Lamport in [8]. Then, we analyze our model and prove that it satisfies validity, agreement, and termination.

First, we cover the technical preliminaries which consist of introductions to the Paxos algorithm, FTMPST, and additions to the syntax and semantics of FTMPST.

Second, we introduce the model, including a global type, processes that execute the Paxos algorithm, and our system requirements. We also examine an example run of the model.

Then, we analyze our model by type checking it. This ensures the model is well-typed. We utilize well-typedness of our model to prove that it satisfies validity, agreement, and termination.

Finally, we summarize our work, reason about the choices we made, and present possible topics for future research.

2 Technical Preliminaries

2.1 Paxos

Lamport describes the Paxos algorithm in [8]. First, he outlines the problem and then the algorithm to solve that problem.

2.1.1 The Problem

Assume a collection of processes that can propose values. A consensus algorithm ensures that a single one among the proposed values is chosen. If no value is proposed, then no value should be chosen. If a value has been chosen, then processes should be able to learn the chosen value. The safety requirements for consensus are:

- **Validity**: Only a value that has been proposed may be chosen,
- **Agreement**: Only a single value is chosen, and
- **Termination**: If a value has been proposed and sufficient processes remain non-faulty, then eventually some value is chosen (compare to [10]).

Assume that agents can communicate with one another by sending messages. We use the customary asynchronous, non-Byzantine model, in which:

- Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.
- Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.

We assume two classes of agents: proposers and acceptors.

A *quorum* is a set of acceptors that can choose a value proposed by a proposer [7]. For Paxos a quorum is a majority of acceptors [7].

2.1.2 The Algorithm

Phase 1. (a) A proposer selects a proposal number n and sends a *prepare* request with number n to a majority of acceptors.

(b) If an acceptor receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Phase 2. (a) If the proposer receives a response to its *prepare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than n.

A proposer can make multiple proposals, so long as it follows the algorithm for each one. It can abandon a proposal in the middle of the protocol at any time. It is probably a good idea to abandon a proposal if some proposer has begun trying to issue a higher-numbered one. Therefore, if an acceptor ignores a *prepare* or *accept* request because it has already received a *prepare* request with a higher number, then it should probably inform the proposer, who should then abandon its proposal. This is a performance optimization that does not affect correctness.

To guarantee progress, a distinguished proposer must be selected as the only one to try issuing proposals. If the distinguished proposer can communicate successfully with a majority of acceptors, and if it uses a proposal with number greater than any already used, then it will succeed in issuing a proposal that is accepted. By abandoning a proposal and trying again if it learns about some request with a higher proposal number, the distinguished proposer will eventually choose a high enough proposal number.

If enough of the system is working properly, liveness can therefore be achieved by electing a single distinguished proposer.

2.1.3 The Implementation

The Paxos algorithm [9] assumes a network of processes. In its consensus algorithm, each process plays the role of proposer, acceptor. The algorithm chooses a leader, which plays the roles of the distinguished proposer. The Paxos consensus algorithm is precisely the one described above, where requests and responses are sent as ordinary messages. Stable storage, preserved during failures, is used to maintain the information that the acceptor must remember. An acceptor records its intended response in stable storage before actually sending the response.

Different proposers choose their numbers from disjoint sets of numbers, so two different proposers never issue a proposal with the same number. Each proposer remembers (in stable storage) the highest-numbered proposal it has tried to issue, and begins phase 1 with a higher proposal number than any it has already used.

2.2 Fault-Tolerant Multiparty Session Types

Our model of the Paxos algorithm uses a fault-tolerant extension of Multiparty Session Types introduced by Peters, Nestmann, and Wagner in [11]. The following explanation of Fault-Tolerant Multiparty Session Types is from that same paper.

We consider three sources of failure in an unreliable communication: (1) the sender may crash before it releases the message, (2) the receiver may crash before it can consume the message, or (3) the communication medium may lose the message. Since types consider only static and predictable information, we do not distinguish between different kinds of failure or model their source in types. Instead, we only allow types, i.e., the specifications of systems, to distinguish between potentially faulty and reliable interactions.

We consider three levels of failures in interactions:

Strongly Reliable (r): Neither the sender nor the receiver can crash as long as they are involved in this interaction. The message cannot be lost by the communication medium. This form corresponds to reliable communication as it was described in [1] in the context of distributed algorithms. This is the standard, failure-free case.

Weakly Reliable (w): Both the sender and the receiver might crash at every possible point during this interaction. But the communication medium cannot lose the message.

Unreliable (u): Both the sender and the receiver might crash at every possible point during this interaction and the communication medium might lose the message. There are no guarantees that this interaction—or any part of it—takes place.

2.2.1 Syntax

We assume that the sets $\mathcal{N}$ of names $a, s, x \ldots$; $\mathcal{R}$ of roles $\mathsf{n}, \mathsf{r}, \ldots$; $\mathcal{L}$ of labels $l, l_{\mathrm{d}}, \ldots$; $\mathcal{V}_{\mathrm{T}}$ of type variables t; and $\mathcal{V}_{\mathrm{P}}$ of process variables X are pairwise distinct. To simplify the reduction semantics of our session calculus, we use natural numbers as roles (compare to [6]). Sorts S range over $\mathbb{B}, \mathbb{N}, \ldots$. The set $\mathcal{E}$ of expressions $e, v, b, \ldots$ is constructed from the standard Boolean operations, natural numbers, names, and (in)equalities.

Global types specify the desired communication structure of systems from a global point of view. In local types this global view is projected to the specification of a single role/participant. We use standard MPST ([5, 6]) extended by operators for unreliable communication and weakly reliable as shown in Fig. 2.1.

The processes $\overline{a}[\mathsf{n}](s).P$ and $a[\mathsf{r}](s).P$ initialize a new session s with n roles via the shared channel a and then proceed as P. We identify sessions with their unique session channel.

The type $\mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\langle \mathrm{S} \rangle.G$ specifies a strongly reliable communication from role r_1 to role r_2 to transmit a value of the sort S and then continues with G. A system with this type will be guaranteed to perform a corresponding action. In a session s this communication is implemented by the sender $s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{r}}\langle e \rangle.P_1$ (specified as $[\mathsf{r}_2]!_{\mathrm{r}}\langle \mathrm{S} \rangle.T_1$) and the receiver $s[\mathsf{r}_2, \mathsf{r}_1]?_{\mathrm{r}}(x).P_2$ (specified as $[\mathsf{r}_1]?_{\mathrm{r}}\langle \mathrm{S} \rangle.T_2$). As result of the communication, the receiver instantiates x in its continuation P_2 with the received value.

The type $\mathsf{r}_1 \to_{\mathrm{u}} \mathsf{r}_2{:}l\langle \mathrm{S} \rangle.G$ specifies an unreliable communication from r_1 to r_2 transmitting (if successful) a label l and a value of type S and then continues (regardless of the success of this communication) with G. The unreliable counterparts of senders and receivers are $s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{u}}l\langle e \rangle.P_1$ (specified as $[\mathsf{r}_2]!_{\mathrm{u}}l\langle \mathrm{S} \rangle.T_1$) and

Global Types	Local Types	Processes
		$P ::= \overline{a}[\mathsf{n}](s).P$
		$\mid\ a[\mathsf{r}](s).P$
$G ::= \mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\langle \mathrm{S}\rangle.G$	$T ::= [\mathsf{r}_2]!_{\mathrm{r}}\langle \mathrm{S}\rangle.T$	$\mid\ s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}\langle e\rangle.P$
	$\mid\ [\mathsf{r}_1]?_{\mathrm{r}}\langle \mathrm{S}\rangle.T$	$\mid\ s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{r}}(x).P$
$\mid\ \mathsf{r}_1 \to_{\mathrm{u}} \mathsf{r}_2{:}l\langle \mathrm{S}\rangle.G$	$\mid\ [\mathsf{r}_2]!_{\mathrm{u}}l\langle \mathrm{S}\rangle.T$	$\mid\ s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{u}}l\langle e\rangle.P$
	$\mid\ [\mathsf{r}_1]?_{\mathrm{u}}l\langle \mathrm{S}\rangle.T$	$\mid\ s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{u}}l\langle v\rangle(x).P$
$\mid\ \mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\{l_i.G_i\}_{i\in \mathrm{I}}$	$\mid\ [\mathsf{r}_2]!_{\mathrm{r}}\{l_i.T_i\}_{i\in \mathrm{I}}$	$\mid\ s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}l.P$
	$\mid\ [\mathsf{r}_1]?_{\mathrm{r}}\{l_i.T_i\}_{i\in \mathrm{I}}$	$\mid\ s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{r}}\{l_i.P_i\}_{i\in \mathrm{I}}$
$\mid\ \mathsf{r} \to_{\mathrm{w}} \mathsf{R}{:}\{l_i.G_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$	$\mid\ [\mathsf{R}]!_{\mathrm{w}}\{l_i.T_i\}_{i\in \mathrm{I}}$	$\mid\ s[\mathsf{r},\mathsf{R}]!_{\mathrm{w}}l.P$
	$\mid\ [\mathsf{r}]?_{\mathrm{w}}\{l_i.T_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$	$\mid\ s[\mathsf{r}_j,\mathsf{r}]?_{\mathrm{w}}\{l_i.P_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$
$\mid\ G_1 \mid\mid G_2$		$\mid\ P_1 \mid P_2$
$\mid\ (\mu t)G \mid t \mid \mathtt{end}$	$\mid\ (\mu t)T \mid t \mid \mathtt{end}$	$\mid\ (\mu X)P \mid X \mid \mathbf{0}$
		$\mid\ \mathtt{if}\ b\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$
		$\mid\ (\nu x)P \mid \bot$
$\mid\ \mathsf{r}_1 \to \mathsf{r}_2{:}\langle s'[\mathsf{r}]{:}T\rangle.G$	$\mid\ [\mathsf{r}_2]!\langle s'[\mathsf{r}]{:}T\rangle.T'$	$\mid\ s[\mathsf{r}_1,\mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P$
	$\mid\ [\mathsf{r}_1]?\langle s'[\mathsf{r}]{:}T\rangle.T'$	$\mid\ s[\mathsf{r}_2,\mathsf{r}_1]?(\!(s'[\mathsf{r}])\!).P$
		$\mid\ s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}$

Message Types	Messages
$\mathrm{MT} ::= \langle \mathrm{S}\rangle^{\mathrm{r}} \mid l\langle \mathrm{S}\rangle^{\mathrm{u}} \mid l^{\mathrm{r}} \mid l^{\mathrm{w}} \mid s[\mathsf{r}]$	$\mathrm{M} ::= \langle v\rangle^{\mathrm{r}} \mid l\langle v\rangle^{\mathrm{u}} \mid l^{\mathrm{r}} \mid l^{\mathrm{w}} \mid s[\mathsf{r}]$

Figure 2.1: Syntax of Fault-Tolerant MPST

$s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{u}}l\langle v\rangle(x).P_2$ (specified as $[\mathsf{r}_1]?_{\mathrm{u}}l\langle \mathrm{S}\rangle.T_2$). The receiver $s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{u}}l\langle v\rangle(x).P_2$ declares a default value v that is used instead of a received value to instantiate x after a failure.

The strongly reliable branching $\mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\{l_i.G_i\}_{i\in \mathrm{I}}$ allows r_1 to pick one of the branches offered by r_2. We identify the branches with their respective label. Selection of a branch is implemented by $s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}l.P$ (specified as $[\mathsf{r}_2]!_{\mathrm{r}}\{l_i.T_i\}_{i\in \mathrm{I}}$). Upon receiving branch l_j from r_1 the process $s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{r}}\{l_i.P_i\}_{i\in \mathrm{I}}$ (specified as $[\mathsf{r}_1]?_{\mathrm{r}}\{l_i.T_i\}_{i\in \mathrm{I}}$) continues with P_j.

The weakly reliable counterpart of branching is $\mathsf{r} \to_{\mathrm{w}} \mathsf{R}{:}\{l_i.G_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$, where $\mathsf{R} \subseteq \mathcal{R}$ and l_{d} with $\mathrm{d} \in \mathrm{I}$ is the default branch. We use a broadcast from r to all roles in R to ensure that the sender can influence several participants with its decision consistently as it is the case for strongly reliable branching. The type system will ensure that this branching construct is weakly reliable, i.e., the involved participants might crash, but no message is lost. Because of that, all processes that are not crashed will move to the same branch. We often abbreviate branching w.r.t. to a small set of branches by omitting the set brackets and instead separating the branches by $\oplus$, where the last branch is always the default branch. In contrast to the strongly reliable cases, the weakly reliable selection $s[\mathsf{r},\mathsf{R}]!_{\mathrm{w}}l.P$ (specified as $[\mathsf{R}]!_{\mathrm{w}}\{l_i.T_i\}_{i\in \mathrm{I}}$) allows to broadcast its decision to R and $s[\mathsf{r}_j,\mathsf{r}]?_{\mathrm{w}}\{l_i.P_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$ (specified as $[\mathsf{r}]?_{\mathrm{w}}\{l_i.T_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$) defines a default label l_{d}.

The $\bot$ denotes a process that crashed. Similar to [6], we use message queues to implement asynchrony in sessions. Therefore, session initialization introduces a directed and initially empty message queue $s_{\mathsf{r}_1\to\mathsf{r}_2}{:}[\,]$ for each pair of roles $\mathsf{r}_1 \neq \mathsf{r}_2$ of the session s. We have five kinds of messages and corresponding message types in Fig. 2.1—one for each kind of interaction.

The remaining operators for independence $G \mid\mid G'$; parallel composition $P \mid P'$; recursion $(\mu t)G$, $(\mu X)P$; inaction $\mathtt{end}$, $\mathbf{0}$; conditionals $\mathtt{if}\ b\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$; session delegation $\mathsf{r}_1 \to \mathsf{r}_2{:}\langle s'[\mathsf{r}]{:}T\rangle.G$, $s[\mathsf{r}_1,\mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P$,

$s[\mathsf{r}_2, \mathsf{r}_1]?((s'[\mathsf{r}])).P$; and restriction $(\nu x)P$ are all standard.

Our type system verifies processes, i.e., implementations, against a specification that is a global type. Since processes implement local views, local types are used as a mediator between the global specification and the respective local end points. To ensure that the local types correspond to the global type, they are derived by *projection*. Instead of the projection function described in [6] we use a more relaxed variant of projection as introduced in [13].

Projection maps global types onto the respective local type for a given role p. The projections of the new global types are obtained straightforwardly from the projection of their respective strongly reliable counterparts:

$$(\mathsf{r}_1 \to_\diamond \mathsf{r}_2{:}\mathfrak{S}.G)\upharpoonright_\mathsf{p} \triangleq \begin{cases} [\mathsf{r}_2]!_\diamond\mathfrak{S}.G\upharpoonright_\mathsf{p} & \text{if } \mathsf{p} = \mathsf{r}_1 \\ [\mathsf{r}_1]?_\diamond\mathfrak{S}.G\upharpoonright_\mathsf{p} & \text{if } \mathsf{p} = \mathsf{r}_2 \\ G\upharpoonright_\mathsf{p} & \text{otherwise} \end{cases}$$

where either $\diamond = \mathrm{r}$, $\mathfrak{S} = \langle \mathrm{S} \rangle$ or $\diamond = \mathrm{u}$, $\mathfrak{S} = l\langle \mathrm{S} \rangle$ and

$$\left(\mathsf{r}_1 \to_\diamond \mathfrak{R}{:}\{l_i.G_i\}_{i\in \mathrm{I}\mathfrak{D}}\right)\upharpoonright_\mathsf{p} \triangleq \begin{cases} [\mathfrak{R}]!_\diamond\{l_i.G_i\upharpoonright_\mathsf{p}\}_{i\in\mathrm{I}} & \text{if } \mathsf{p} = \mathsf{r}_1 \\ [\mathsf{r}_1]?_\diamond\{l_i.G_i\upharpoonright_\mathsf{p}\}_{i\in \mathrm{I}\mathfrak{D}} & \text{if } \mathfrak{B} \\ \bigsqcup_{i\in\mathrm{I}} (G_i\upharpoonright_\mathsf{p}) & \text{otherwise} \end{cases}$$

where either $\diamond = \mathrm{r}$, $\mathfrak{R} = \mathsf{r}_2$, $\mathfrak{B}$ is $\mathsf{p} = \mathsf{r}_2$, $\mathfrak{D}$ is empty or $\diamond = \mathrm{w}$, $\mathfrak{R} = \mathsf{R}$, $\mathfrak{B}$ is $\mathsf{p} \in \mathsf{R}$, $\mathfrak{D}$ is $, l_\mathrm{d}$. In the last case of strongly reliable or weakly reliable branching—when projecting onto a role that does not participate in this branching—we map to $\bigsqcup_{i\in\mathrm{I}} (G_i\upharpoonright_\mathsf{p}) = (G_1\upharpoonright_\mathsf{p}) \sqcup \ldots \sqcup (G_n\upharpoonright_\mathsf{p})$. The operation $\sqcup$ is (similar to [13]) inductively defined as:

$$\begin{aligned} T \sqcup T &= T \\ ([\mathsf{r}]?_\mathrm{r}\mathrm{I}_1) \sqcup ([\mathsf{r}]?_\mathrm{r}\mathrm{I}_2) &= [\mathsf{r}]?_\mathrm{r}(\mathrm{I}_1 \sqcup \mathrm{I}_2) \\ ([\mathsf{r}]?_\mathrm{w}\mathrm{I}_1) \sqcup ([\mathsf{r}]?_\mathrm{w}\mathrm{I}_2) &= [\mathsf{r}]?_\mathrm{w}(\mathrm{I}_1 \sqcup \mathrm{I}_2) \quad \text{if } \mathrm{I}_1 \text{ and } \mathrm{I}_2 \text{ have the same default branch} \\ \mathrm{I} \sqcup \emptyset &= \mathrm{I} \\ \mathrm{I} \sqcup (\{l.T\} \cup \mathrm{J}) &= \begin{cases} \{l.\,(T' \sqcup T)\} \cup ((\mathrm{I} \setminus \{l.T'\}) \sqcup \mathrm{J}) & \text{if } l.T' \in \mathrm{I} \\ \{l.T\} \cup (\mathrm{I} \sqcup \mathrm{J}) & \text{if } l \notin \mathrm{I} \end{cases} \end{aligned}$$

where $T, T' \in \mathcal{T}$, $l \notin \mathrm{I}$ is short hand for $\nexists T'.\ l.T' \in \mathrm{I}$, and is undefined in all other cases. The mergeability relation $\sqcup$ states that two types are identical up to their branching types, where only branches with distinct labels are allowed to be different. This ensures that if the sender r_1 in $\mathsf{r}_1 \to_\mathrm{r} \mathsf{r}_2{:}\{l_i.G_i\}_{i\in\mathrm{I}}$ decides to branch then only processes that are informed about this decision can adapt their behavior accordingly; else projection is **not** defined.

The remaining global types are projected as follows:

$$(G_1 \mid\mid G_2)\upharpoonright_\mathsf{p} \triangleq \begin{cases} G_1\upharpoonright_\mathsf{p} & \text{if } \mathsf{p} \notin \mathrm{R}(G_2) \\ G_2\upharpoonright_\mathsf{p} & \text{if } \mathsf{p} \notin \mathrm{R}(G_1) \end{cases} \qquad ((\mu t)G)\upharpoonright_\mathsf{p} \triangleq \begin{cases} (\mu t)G\upharpoonright_\mathsf{p} & \text{if } \mathsf{p} \in \mathrm{R}(G) \\ \mathtt{end} & \text{otherwise} \end{cases}$$

$$t\upharpoonright_\mathsf{p} \triangleq t \qquad \mathtt{end}\upharpoonright_\mathsf{p} \triangleq \mathtt{end}$$

Projecting a recursive global type results in a recursive local type if p occurs in the body of the recursion or else in successful termination. Type variables and successful termination are mapped onto themselves. We

denote a global type G as *projectable* if for all $\mathsf{r} \in \mathrm{R}(G)$ the projection $G\upharpoonright_{\mathsf{r}}$ is defined. We restrict our attention to projectable global types.

In types $(\mu t)G$ and $(\mu t)T$ the type variable t is *bound*. In processes $(\mu X)P$ the process variable X is bound. Similarly, all names in round brackets are bound in the remainder of the respective process, e.g. s is bound in P by $\overline{a}[\mathsf{n}](s).P$ and x is bound in P by $s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{r}}(x).P$. A variable or name is *free* if it is not bound. Let $\mathrm{FN}(P)$ return the free names of P.

We use '.' (as e.g. in $\overline{a}[\mathsf{r}](s).P$) to denote sequential composition. In all operators the *prefix* before '.' guards the *continuation* after the '.'. Let $\prod_{1 \leq i \leq n} P_i$ abbreviate the parallel composition $P_1 \mid \ldots \mid P_n$.

We write $\mathrm{nsr}(G)$, $\mathrm{nsr}(T)$, and $\mathrm{nsr}(P)$, if none of the prefixes in G, T, and P is strongly reliable or for delegation and if P does not contain message queues.

The combination of a session channel and a role uniquely identifies a participant of a session, called an *actor*. A process has an actor $s[\mathsf{r}]$ if it has an action prefix on s, where r is the first role mentioned in the prefix. Let $\mathrm{A}(P)$ be the set of actors of P.

2.2.2 Semantics

The application of a substitution $\{y/x\}$ on a term A, denoted as $A\{y/x\}$, is defined as the result of replacing all free occurrences of x in A by y, possibly applying alpha-conversion to avoid capture or name clashes. For all names $n \in \mathcal{N} \setminus \{x\}$ the substitution behaves as the identity mapping. We use substitution on types as well as processes and naturally extend substitution to the substitution of variables by terms (to unfold recursions) and names by expressions (to instantiate a bound name with a received value).

Labels allow us to distinguish between different branches. Our MPST variant will ensure that all occurrences of the same label are associated with the same sort. We assume a predicate $\doteq$ that compares two labels and is valid if the parts of the labels that do not refer to runtime information correspond. We require that $\doteq$ is unambiguous on labels used in types [11].

We use structural congruence to abstract from syntactically different processes with the same meaning, where $\equiv$ is the least congruence that satisfies alpha conversion and the rules:

$$P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$$
$$(\mu X)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$$
$$(\nu x)\,(P_1 \mid P_2) \equiv P_1 \mid (\nu x)P_2 \quad \text{if } x \notin \mathrm{FN}(P_1)$$

Reduction

The reduction semantics of the session calculus is defined in Fig. 2.2, where we follow [6]: we assume that session initialization is synchronous and communication within a session is asynchronous implemented using message queues.

Rule (Init) initializes a session with n roles. Session initialization introduces a fresh session channel and unguards the participants of the session. Finally, the message queues of this session are initialized with the empty list under the restriction of the session channel.

Rule (RSend) implements an asynchronous strongly reliable message transmission. As a result the value $\mathrm{eval}(y)$ is wrapped in a message and added to the end of the corresponding message queue and the continuation of

(Init)	$\overline{a}[\mathsf{n}](s).P_{\mathsf{n}} \mid \prod_{1\leq \mathsf{i}\leq \mathsf{n}-1} a[\mathsf{i}](s).P_{\mathsf{i}} \longmapsto (\nu s)\left(\prod_{1\leq \mathsf{i}\leq \mathsf{n}} P_{\mathsf{i}} \mid \prod_{1\leq \mathsf{i},\mathsf{j}\leq \mathsf{n},\mathsf{i}\neq\mathsf{j}} s_{\mathsf{i}\to\mathsf{j}}{:}[\,]\right)$	if $a \neq s$
(RSend)	$s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}\langle y\rangle.P \mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}\#\langle v\rangle^{\mathrm{r}}$	if $\mathrm{eval}(y) = v$
(RGet)	$s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{r}}(x).P \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\langle v\rangle^{\mathrm{r}}\#\mathrm{M} \longmapsto P\{^{v}/_{x}\} \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M}$	
(USend)	$s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{u}}l\langle y\rangle.P \mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}\#l\langle v\rangle^{\mathrm{u}}$	if $\mathrm{eval}(y) = v$
(UGet)	$s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{u}}l\langle dv\rangle(x).P \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}l'\langle v\rangle^{\mathrm{u}}\#\mathrm{M} \longmapsto P\{^{v}/_{x}\} \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M}$	if $l \doteq l'$, $\mathtt{FP}_{\mathtt{uget}}(s,\mathsf{r}_1,\mathsf{r}_2,l')$
(USkip)	$s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{u}}l\langle dv\rangle(x).P \longmapsto P\{^{dv}/_{x}\}$	if $\mathtt{FP}_{\mathtt{uskip}}(s,\mathsf{r}_1,\mathsf{r}_2,l)$
(ML)	$s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l\langle v\rangle^{\mathrm{u}}\#\mathrm{M} \longmapsto s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}$	if $\mathtt{FP}_{\mathtt{ml}}(s,\mathsf{r}_1,\mathsf{r}_2,l)$
(RSel)	$s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}l.P \mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}\#l^{\mathrm{r}}$	
(RBran)	$s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{r}}\{l_i.P_i\}_{i\in\mathrm{I}} \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}l^{\mathrm{r}}\#\mathrm{M} \longmapsto P_j \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M}$	if $l \doteq l_j$, $j \in \mathrm{I}$
(WSel)	$s[\mathsf{r},\mathsf{R}]!_{\mathrm{w}}l.P \mid \prod_{\mathsf{r}_i\in\mathsf{R}} s_{\mathsf{r}\to\mathsf{r}_i}{:}\mathrm{M}_i \longmapsto P \mid \prod_{\mathsf{r}_i\in\mathsf{R}} s_{\mathsf{r}\to\mathsf{r}_i}{:}\mathrm{M}_i\#l^{\mathrm{w}}$	
(WBran)	$s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{w}}\{l_i.P_i\}_{i\in\mathrm{I},l_{\mathrm{d}}} \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}l^{\mathrm{w}}\#\mathrm{M} \longmapsto P_j \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M}$	if $l \doteq l_j$, $j \in \mathrm{I}$
(WSkip)	$s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{w}}\{l_i.P_i\}_{i\in\mathrm{I},l_{\mathrm{d}}} \longmapsto P_{\mathrm{d}}$	if $\mathtt{FP}_{\mathtt{wskip}}(s,\mathsf{r}_1,\mathsf{r}_2)$
(Crash)	$P \longmapsto \bot$	if $\mathtt{FP}_{\mathtt{crash}}(P)$
(If-T)	$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \longmapsto P$	if e is true
(If-F)	$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \longmapsto P'$	if e is false
(Deleg)	$s[\mathsf{r}_1,\mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P \mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}\#s'[\mathsf{r}]$	
(SRecv)	$s[\mathsf{r}_1,\mathsf{r}_2]?(\!(s'[\mathsf{r}])\!).P \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}s''[\mathsf{r}']\#\mathrm{M} \longmapsto P\{^{s''}/_{s'}\}\{^{\mathsf{r}'}/_{\mathsf{r}}\} \mid \mathrm{M}$	
(Par)	$P_1 \mid P_2 \longmapsto P_1' \mid P_2$	if $P_1 \longmapsto P_1'$
(Res)	$(\nu x)P \longmapsto (\nu x)P'$	if $P \longmapsto P'$
(Rec)	$(\mu X)P \longmapsto P\{^{(\mu X)P}/_{X}\}$	
(Struc)	$P_1 \longmapsto P_1'$	if $P_1 \equiv P_2$, $P_2 \longmapsto P_2'$, $P_2' \equiv P_1'$

Figure 2.2: Reduction Rules ($\longmapsto$) of Fault-Tolerant Processes.

the sender is unguarded. Rule (USend) is the counterpart of (RSend) for unreliable senders. (RGet) consumes a message that is marked as strongly reliable with the index r from the head of the respective message queue and replaces in the unguarded continuation of the receiver the bound variable x by the received value y.

There are two rules for the reception of a message in an unreliable communication that are guided by failure patterns. *Failure patterns* are predicates that we deliberately choose not to define here (see below). They allow us to provide information about the underlying communication medium and the reliability of processes. Rule (UGet) is similar to Rule (RGet), but specifies a failure pattern $\mathtt{FP}_{\mathtt{uget}}$ to decide whether this step is allowed. The Rule (USkip) allows to skip the reception of a message in an unreliable communication using a failure pattern $\mathtt{FP}_{\mathtt{uskip}}$ and instead substitutes the bound variable x in the continuation with the default value dv. The failure pattern $\mathtt{FP}_{\mathtt{uskip}}$ tells us whether a reception can be skipped.

Rule (RSel) puts the label l selected by r_1 at the end of the message queue towards r_2. Its weakly reliable counterpart (WSel) is similar, but puts the label at the end of all relevant message queues. With (RBran) a label is consumed from the top of a message queue and the receiver moves to the indicated branch. There are again two weakly reliable counterparts of (RBran). Rule (WBran) is similar to (RBran), whereas (WSkip) allows r_1 to skip the message and to move to its default branch if the failure pattern $\mathtt{FP}_{\mathtt{wskip}}$ holds.

The Rules (Crash) for *crash failures* and (ML) for *message loss*, describe failures of a system. With Rule (Crash) P can crash if $\mathtt{FP}_{\mathtt{crash}}$. (ML) allows to drop an unreliable message if the failure pattern $\mathtt{FP}_{\mathtt{ml}}$ is valid.

The remaining rules for conditionals, session delegation, parallel composition, restriction, recursion, and structural congruence in Fig. 2.2 are standard.

We augmented our reduction semantics in Fig. 2.2 by five different failure patterns that we deliberately do not specify, although we usually assume that the failure patterns $\mathtt{FP}_{\mathtt{uget}}$, $\mathtt{FP}_{\mathtt{uskip}}$, and $\mathtt{FP}_{\mathtt{wskip}}$ use only local information, whereas $\mathtt{FP}_{\mathtt{ml}}$ and $\mathtt{FP}_{\mathtt{crash}}$ may use global information of the system in the current run.

Typing

The type of a process P is checked in a *typed judgment*, i.e., triples $\Gamma \vdash P \triangleright \Delta$, where

$$\begin{aligned} \Gamma &::= \emptyset \mid \Gamma \cdot x{:}\mathrm{S} \mid \Gamma \cdot a{:}G \mid \Gamma \cdot l{:}\mathrm{S} \\ \Delta &::= \emptyset \mid \Delta \cdot s[\mathsf{r}]{:}T \mid \Delta \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}^* \end{aligned}$$

Assignments in Γ relate variables to their sort, shared channels to the type of the session they introduce, and connect labels with a sort. Session environments collect the local types of actors and the list of message types of queues, i.e., MT^* denotes a list of message types.

We write $x \sharp \Gamma$ and $x \sharp \Delta$ if the name x does not occur in Γ and Δ, respectively. We use $\cdot$ to add an assignment provided that the new assignment is not in conflict with the type environment, i.e., $\Gamma \cdot A$ implies that the respective name/variable/label in A is not contained in Γ and $\Delta \cdot A$ implies that the respective actor/queue in A is not contained in Δ. These conditions on $\cdot$ for global and session environments are referred to as *linearity*. We restrict in the following our attention to linear environments. We abstract in session environments from assignments towards terminated local types, i.e., $\Delta \cdot s\,[r] : 0 = \Delta$.

We write $\mathrm{nsr}(\Delta)$ if $\mathrm{nsr}(T)$ for all local types T in Δ and if Δ does not contain message queues. With $\Gamma \Vdash y{:}\mathrm{S}$ we check that y is an expression of the sort S if all names x in y are replaced by arbitrary values of sort S_x for $x{:}\mathrm{S}_x \in \Gamma$.

$$(\mathsf{Req})\ \frac{a{:}G \in \Gamma \quad |\mathrm{R}(G)| = \mathsf{n}}{\Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{n}]{:}G\upharpoonright_{\mathsf{n}}} \qquad (\mathsf{Acc})\ \frac{a{:}G \in \Gamma \quad 0 < \mathsf{r} < |\mathrm{R}(G)|}{\Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}]{:}G\upharpoonright_{\mathsf{r}}}$$

$$(\mathsf{RSend})\ \frac{\Gamma \Vdash y{:}\mathrm{S} \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{r}}\langle y\rangle.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{r}}\langle \mathrm{S}\rangle.T}$$

$$(\mathsf{RGet})\ \frac{x\sharp(\Gamma, \Delta, s) \quad \Gamma \cdot x{:}\mathrm{S} \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{r}}(x).P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{r}}\langle \mathrm{S}\rangle.T}$$

$$(\mathsf{USend})\ \frac{\Gamma \Vdash y{:}\mathrm{S} \quad l \doteq l' \quad l'{:}\mathrm{S} \in \Gamma \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{u}}l\langle y\rangle.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{u}}l'\langle \mathrm{S}\rangle.T}$$

$$(\mathsf{UGet})\ \frac{x\sharp(\Gamma, \Delta, s) \quad \Gamma \Vdash v{:}\mathrm{S} \quad l \doteq l' \quad l'{:}\mathrm{S} \in \Gamma \quad \Gamma \cdot x{:}\mathrm{S} \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{u}}l\langle v\rangle(x).P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{u}}l'\langle \mathrm{S}\rangle.T}$$

$$(\mathsf{RSel})\ \frac{j \in \mathrm{I} \quad l \doteq l_j \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T_j}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{r}}l.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{r}}\{l_i.T_i\}_{i\in\mathrm{I}}} \qquad (\mathsf{Var})\ \frac{}{\Gamma \cdot X{:}t \vdash X \triangleright s[\mathsf{r}]{:}t}$$

$$(\mathsf{RBran})\ \frac{\forall j \in \mathrm{I}_2.\ \exists i \in \mathrm{I}_1.\ l_i \doteq l_j \wedge \Gamma \vdash P_i \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T_j}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{r}}\{l_i.P_i\}_{i\in\mathrm{I}_1} \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{r}}\{l_i.T_i\}_{i\in\mathrm{I}_2}}$$

$$(\mathsf{WSel})\ \frac{j \in \mathrm{I} \quad l \doteq l_j \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}]{:}T_j}{\Gamma \vdash s[\mathsf{r}, \mathsf{R}]!_{\mathrm{w}}l.P \triangleright \Delta \cdot s[\mathsf{r}]{:}[\mathsf{R}]!_{\mathrm{w}}\{l_i.T_i\}_{i\in\mathrm{I}}} \qquad (\mathsf{Crash})\ \frac{\mathrm{nsr}(\Delta)}{\Gamma \vdash \bot \triangleright \Delta}$$

$$(\mathsf{WBran})\ \frac{l_{\mathrm{d}} \doteq l'_{\mathrm{d}} \quad \forall j \in \mathrm{I}_2.\ \exists i \in \mathrm{I}_1.\ l_i \doteq l_j \wedge \Gamma \vdash P_i \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T_j}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{w}}\{l_i.P_i\}_{i\in\mathrm{I}_1, l_{\mathrm{d}}} \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{w}}\{l_i.T_i\}_{i\in\mathrm{I}_2, l'_{\mathrm{d}}}}$$

$$(\mathsf{Deleg})\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!\langle s'[\mathsf{r}]{:}T'\rangle.T \cdot s'[\mathsf{r}]{:}T'}$$

$$(\mathsf{SRecv})\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T \cdot s'[\mathsf{r}]{:}T'}{\Gamma \vdash s[\mathsf{r}_1, \mathsf{r}_2]?(\!(s'[\mathsf{r}])\!).P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?\langle s'[\mathsf{r}]{:}T'\rangle.T} \qquad (\mathsf{End})\ \frac{}{\Gamma \vdash \mathbf{0} \triangleright \emptyset}$$

$$(\mathsf{If})\ \frac{\Gamma \Vdash e{:}\mathbb{B} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \triangleright \Delta} \qquad (\mathsf{Par})\ \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta'}{\Gamma \vdash P \mid P' \triangleright \Delta \cdot \Delta'}$$

$$(\mathsf{Res1})\ \frac{x\sharp(\Gamma, \Delta) \quad \Gamma \cdot x{:}\mathrm{S} \vdash P \triangleright \Delta}{\Gamma \vdash (\nu x)P \triangleright \Delta} \qquad (\mathsf{Rec})\ \frac{\Gamma \cdot X{:}t \vdash P \triangleright \Delta \cdot s[\mathsf{r}]{:}T}{\Gamma \vdash (\mu X)P \triangleright \Delta \cdot s[\mathsf{r}]{:}(\mu t)T}$$

Figure 2.3: Typing Rules for Fault-Tolerant Systems.

A process P is *well-typed* w.r.t. Γ and Δ if $\Gamma \vdash P \triangleright \Delta$ can be derived from the rules in the Fig. 2.3. We concentrate on the interaction cases, where we observe that all new cases are quite similar to their strongly reliable counterparts.

Rule (RSend) checks strongly reliable senders, i.e., requires a matching strongly reliable sending in the local type of the actor and compares the actor with this type. With $\Gamma \Vdash y{:}\mathrm{S}$ we check that y is an expression of the sort S if all names x in y are replaced by arbitrary values of sort S_x for $x{:}\mathrm{S}_x \in \Gamma$. Then the continuation of the process is checked against the continuation of the type. The unreliable case is very similar, but additionally checks that the label is assigned to the sort of the expression in Γ. Rule (RGet) type strongly reliable receivers, where again the prefix is checked against a corresponding type prefix and the assumption $x{:}\mathrm{S}$ is added to the type check of the continuation. Again the unreliable case is very similar, but apart from the label also checks the sort of the default value.

Rule (RSel) checks the strongly reliable selection prefix, that the selected label matches one of the specified labels, and that the process continuation is well-typed w.r.t. the type continuation following the selected label. The only difference in the weakly reliable case is the set of roles for the receivers. For strongly reliable branching we have to check the prefix and that for each branch in the type there is a matching branch in the process that is well-typed w.r.t. the respective branch in the type. For the weakly reliable case we have to additionally check that the default labels of the process and the type coincide.

Rule (Crash) for crashed processes checks that $\mathrm{nsr}(\Delta)$.

Failure Patterns

We have to fix conditions on failure patterns to ensure that subject reduction and progress hold in general.

Condition 1 (Failure Pattern)**.** *1. If* $\mathtt{FP}_{\mathtt{crash}}(P)$*, then* $\mathrm{nsr}(P)$*.*

2. *The failure pattern* $\mathtt{FP}_{\mathtt{uget}}(s, \mathsf{r}_1, \mathsf{r}_2, l)$ *is always valid.*
3. *The pattern* $\mathtt{FP}_{\mathtt{ml}}(s, \mathsf{r}_1, \mathsf{r}_2, l)$ *is valid iff* $\mathtt{FP}_{\mathtt{uskip}}(s, \mathsf{r}_2, \mathsf{r}_1, l)$ *is valid.*
4. *If* $\mathtt{FP}_{\mathtt{crash}}(P)$ *and* $s[\mathsf{r}] \in \mathrm{A}(P)$ *then eventually* $\mathtt{FP}_{\mathtt{uskip}}(s, \mathsf{r}_2, \mathsf{r}, l)$ *and also* $\mathtt{FP}_{\mathtt{wskip}}(s, \mathsf{r}_2, \mathsf{r}, l)$ *for all* r_2, l*.*
5. *If* $\mathtt{FP}_{\mathtt{crash}}(P)$ *and* $s[\mathsf{r}] \in \mathrm{A}(P)$ *then eventually* $\mathtt{FP}_{\mathtt{ml}}(s, \mathsf{r}_1, \mathsf{r}, l)$ *for all* r_1, l*.*
6. *If* $\mathtt{FP}_{\mathtt{wskip}}(s, \mathsf{r}_1, \mathsf{r}_2)$ *then* $s[\mathsf{r}_2]$ *is crashed, i.e., the system does no longer contain an actor* $s[\mathsf{r}_2]$ *and the message queue* $s_{\mathsf{r}_2 \to \mathsf{r}_1}$ *is empty.*

The crash of a process should not block strongly reliable actions, i.e., only processes with $\mathrm{nsr}(P)$ can crash (Condition 1.1). Condition 1.2 requires that no process can refuse to consume a message on its queue. This condition prevents deadlocks that may arise from refusing a message m that is never dropped from the message queue. Condition 1.3 requires that if a message can be dropped from a message queue then the corresponding receiver has to be able to skip this message and vice versa. Similarly, processes that wait for messages from a crashed process have to be able to skip (Condition 1.4) and all messages of a queue towards a crashed receiver can be dropped (Condition 1.5). Finally, weakly reliable branching requests should not be lost. To ensure that the receiver of such a branching request can proceed if the sender is crashed but is not allowed to skip the reception of the branching request before the sender crashed, we require that $\mathtt{FP}_{\mathtt{wskip}}(s, \mathsf{r}_1, \mathsf{r}_2)$ is false as long as $s[\mathsf{r}_2]$ is alive or messages on the respective queue are still in transit (Condition 1.6).

Coherence intuitively describes that a session environment captures all local endpoints of a collection of global types. Since we capture all relevant global types in the global environment, we define coherence on pairs of global and session environments.

Theorem 1 (Subject Reduction)**.** *If* $\Gamma \vdash P \triangleright \Delta$*,* Γ, Δ *are coherent, and* $P \longmapsto P'$*, then there are some* Δ' *such that* $\Gamma \vdash P' \triangleright \Delta'$*.*

Progress states that no part of a well-typed and coherent system can block other parts, that eventually all matching communication partners of strongly reliable and weakly reliable communications (that are not crashed) are unguarded, and that there are no communication mismatches. Subject reduction and progress together then imply *session fidelity*, i.e., that processes behave as specified in their global types.

To ensure that the interleaving of sessions and session delegation cannot introduce deadlocks, we assume an interaction type system as introduced in [2, 6]. For this type system it does not matter whether the considered actions are strongly reliable, weakly reliable, or unreliable. More precisely, we can adapt the interaction type system of [2] in a straightforward way to the above session calculus, where unreliable communication and weakly reliable branching is treated in exactly the same way as strongly reliable communication/branching. We say that P *is free of cyclic dependencies between sessions* if this interaction type system does not detect any cyclic dependencies.

In the literature there are different formulations of progress. We are interested in a rather strict definition of progress that ensures that well-typed systems cannot block. Therefore, we need an additional assumption on session requests and acceptances. Coherence ensures the existence of communication partners within sessions only. If we want to avoid blocking, we need to be sure, that no participant of a session is missing during its initialization. Note that without action prefixes all participants either terminated or crashed.

Theorem 2 (Progress/Session Fidelity)**.** *Let* $\Gamma \vdash P \triangleright \Delta$*,* Γ, Δ *be coherent, and let* P *be free of cyclic dependencies between sessions. Assume that in the derivation of* $\Gamma \vdash P \triangleright \Delta$*, whenever* $\overline{a}[\mathsf{n}](s).Q$ *or* $a[\mathsf{r}](s).Q$ *with* $a{:}G$*, then there are* $\overline{a}[\mathsf{n}](s).Q_n$ *or* $a[\mathsf{r}_i](s).Q_i$ *for all* $1 \leq \mathsf{r}_i < \mathsf{n}$*.*

1. *Then either* P *does not contain any action prefixes or* $P \longmapsto P'$*.*
2. *If* P *does not contain recursion, then there exists* P' *such that* $P \longmapsto^* P'$ *and* P' *does not contain any action prefixes.*

2.3 Additional Notation

2.3.1 Prefixes and Branches

In [11] the authors introduced notation for the construction of global types and processes.

Let $\left(\bigodot_{1 \leq i \leq n} \pi_i\right).G$ abbreviate the sequence $\pi_1. \ldots .\pi_n.G$ to simplify the presentation, where $G \in \mathcal{G}$ is a global type and $\pi_1, \ldots, \pi_n$ are sequences of prefixes. More precisely, each π_i is of the form $\pi_{i,1}. \ldots .\pi_{i,m}$ and each $\pi_{i,j}$ is a type prefix of the form $\mathsf{r}_1 \to_{\mathrm{u}} \mathsf{r}_2{:}l\langle \mathrm{S} \rangle$ or $\mathsf{r} \to_{\mathrm{w}} \mathsf{R}{:}l_1.T_1 \oplus \ldots \oplus l_n.T_n \oplus l_{\mathrm{d}}$, where the latter case represents a weakly reliable branching prefix with the branches $l_1, \ldots, l_n, l_{\mathrm{d}}$, the default branch l_{d}, and where the next global type provides the missing specification for the default case.

Let $\left(\bigodot_{1\leq i\leq n}\pi_i\right).P$ abbreviate the sequence $\pi_1.\ldots.\pi_n.P$, where $P \in \mathcal{P}$ is a process and $\pi_1, \ldots, \pi_n$ are sequences of prefixes.

2.3.2 Function Calls as Prefixes

We extend FTMPST by adding function calls as prefixes. These functions allow us to add side effects to our model.

Let $f : \tilde{X} \to \bot$ be a function with domain $\tilde{X} = X_1 \times \ldots \times X_n$ and $P \in \mathcal{P}$ a process. Then $f\,(x_1, \ldots, x_n)\,.P$ is also a process in $\mathcal{P}$.

We introduce reduction rule (Func) as $f\,(x_1, \ldots, x_n)\,.P \longmapsto P$. After the call to f, the process behaves like P.

Additionally, we define typing rule (Func). Let $F_{\tilde{X},\bot}$ be the set of functions $g : \tilde{X} \to \bot$.

$$(\mathrm{Func})\ \frac{f \in F_{\tilde{X},\bot} \qquad \Gamma \Vdash (x_1, \ldots, x_n) : \tilde{X} \qquad \Gamma \vdash P \rhd \Delta}{\Gamma \vdash f\,(x_1, \ldots, x_n)\,.P \rhd \Delta}$$

In (Func) we require that f is a function from $\tilde{X}$ to $\bot$ and expression $(x_1, \ldots, x_n)$ is of the sort $\tilde{X}$. The latter implies that for $1 \leq i \leq n$ each sub-expression x_i is of the sort X_i if all names w in x_i are replaced by arbitrary values of sort S_w for $w : S_w \in \Gamma$. The sorts of the arguments given correspond to the sorts of the arguments expected in f. Applying (Func) does not affect the global environment Γ or the session environment Δ.

The addition of function calls as prefixes has no significant impact on any properties or theorems in [11].

2.3.3 Type Parameters

We introduce a notation to define sorts using a grammar and type parameters.

Let a and b be any specified sorts. We define a sort $\mathrm{S\ a\ b}$.

$$\mathrm{S\ a\ b} = A\ \mathrm{a} \mid B\ \mathrm{b} \mid C\ \mathrm{a\ b} \mid D \mid E\ \mathbb{N}$$

The values in $\mathrm{S\ a\ b}$ are either of the form $A\ \mathrm{a}$, $B\ \mathrm{b}$, $C\ \mathrm{a\ b}$, D, or $E\ \mathbb{N}$. We say S of a and b is either an A of a, a B of b, a C of a and b, a D, or an E of $\mathbb{N}$. A, B, C, D, and E are constructors. A, B, and E each take one value, C takes two, and D takes none. Note that E does not take a value of a or b but still constructs a value in $\mathrm{S\ a\ b}$.

3 Model

First, we specify some sorts that are used to 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 Bool which we define as a set.

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

Second, we assume a set of proposable values Value. This set contains at least two elements of any kind. For example, $\mathrm{Value} = \{\mathrm{abc}, \mathrm{def}, \ldots, \mathrm{vwx}, \mathrm{yz}\}$.

Then, we have some sorts which we define using a grammar. Each of these definitions contains a type parameter, which is a variable ranging over types. In this case the type parameter 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 parameter 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 $\mathrm{Maybe}\ b$ for any other type b. We use $\mathrm{Maybe}\ a$ where optional values are needed.

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

$\mathrm{Proposal}\ a$ only has one possible form, which is $\mathrm{Prop}\ \mathbb{N}\ a$. 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 $\mathrm{Prop}\ 1\ \mathrm{true}$ and an example for a value of type $\mathrm{Proposal}\ (\mathrm{Maybe}\ \mathbb{N})$ could be $\mathrm{Prop}\ 1\ (\mathrm{Just}\ 1)$. This sort models the proposals issued by the proposers in phase $2a$.

$$\mathrm{Promise}\ a = \mathrm{Promise}\ (\mathrm{Maybe}\ (\mathrm{Proposal}\ a)) \mid \mathrm{Nack}$$

$\mathrm{Promise}\ a$ has two possible forms. $\mathrm{Promise}\ (\mathrm{Maybe}\ (\mathrm{Proposal}\ a))$ and Nack. Possible values include Nack and $\mathrm{Promise}\ (\mathrm{Just}\ (\mathrm{Prop}\ 1\ \mathrm{false}))$ of type $\mathrm{Promise}\ \mathrm{Bool}$. The actual type of Nack, 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 is of the form Nack.

3.2 Global Type

Since each proposer initiates its own session the global type can be defined for one proposer k and a corresponding quorum Q_k. Let $p = |\,\mathrm{Q}_k\,| + 1$ and $\mathrm{R} = \{1, \ldots, |\,\mathrm{Q}_k\,|\}$.

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

We observe that the proposer of a session always uses the highest role p in its session and communicates with all roles from 1 to $p - 1$.

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

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 Q_p 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 Q_p. Should the proposer crash the algorithm ends for this particular proposer and the subprocesses of the acceptors in its quorum.

3.3 Functions

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

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

$\mathrm{propNumber}_p\,(\mathrm{n})$ 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.

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

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

$$\forall p \in \mathbb{P}.\forall n, m \in \mathbb{N} : n > m \rightarrow \text{propNumber}_p\,(\text{n}) > \text{propNumber}_p\,(m)$$

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

$$\text{promValue} : \texttt{list of}\ \text{Promise}\ a \rightarrow a$$

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

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

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

$$\begin{aligned}
&\text{promCount} : \texttt{list of}\ \text{Promise}\ a \rightarrow \mathbb{N}\\
&\text{promCount}\,([]) = 0\\
&\text{promCount}\,((\text{Promise}\ _\#xs)) = 1 + \text{promCount}\,(xs)\\
&\text{promCount}\,((\text{Nack}\,\#xs)) = \text{promCount}\,(xs)
\end{aligned}$$

$\text{promCount}\,(ps)$ takes a list of promises ps and counts the number of promises that have the form $\text{Promise}\ m$. $\text{anyNack}\,(ps)$ and $\text{promCount}\,(ps)$ are used in the proposer to decide which branch to take in phase $2a$.

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

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

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

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

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

$$\mathrm{gen\,Q} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \texttt{list of}\ \mathbb{N}$$

$\mathrm{gen\,Q}\,(p, c_A, c_P)$ returns a randomly selected set Q with $\mathrm{Q} \subseteq A = \{1, \ldots, c_A\}$ and $|\,\mathrm{Q}\,| > \frac{|A|}{2}$ as a sorted list. Q consists of any majority of acceptors. In Paxos a majority of acceptors forms a quorum, i.e., an accepting set that can choose a value [7]. We use this function to define a quorum corresponding to a proposer. The proposer then communicates with the acceptors in its quorum.

$$\begin{aligned}&\mathrm{posRec} : \mathbb{N} \times \mathbb{N} \times \texttt{list of}\ \mathbb{N} \to \mathbb{N}\\&\mathrm{posRec}\,(n, c, (n\#\_)) = c + 1\\&\mathrm{posRec}\,(n, c, (x\#xs)) = \mathrm{posRec}\,(n, c + 1, xs)\\&\mathrm{posRec}\,(\_, \_, []) = 0\end{aligned}$$

$$\begin{aligned}&\mathrm{pos} : \mathbb{N} \times \texttt{list of}\ \mathbb{N} \to \mathbb{N}\\&\mathrm{pos}\,(n, xs) = \mathrm{posRec}\,(n, 0, xs)\end{aligned}$$

$\mathrm{pos}\,(n, xs)$ calculates the position of the first occurrence of n in xs. It uses $\mathrm{posRec}\,(n, c, xs)$, which recursively checks each element of the list. $\mathrm{posRec}\,(n, c, xs)$ and thus $\mathrm{pos}\,(n, xs)$ return 0 in cases where n is not an element of xs. We observe that $1 \leq \mathrm{pos}\,(n, xs) \leq |xs|$ for every list xs and $n \in xs$.

3.4 Processes

In this section we define the processes to model Paxos. We start with the system initialization, which contains the session initialization and starts the proposers and acceptors with the correct arguments. Next, we define the proposer and acceptor processes, which model the Paxos algorithm itself.

3.4.1 System Initialization

Let c_A and c_P be the number of acceptors and proposers respectively. We require that $c_A > 0$ and $c_P > 0$. Let $\mathrm{Q}_k = \mathrm{gen}\,\mathrm{Q}\,(k, c_A, c_P)$ be the quorum and n_k a register for proposer with process index k where $c_A + 1 \leq k \leq c_A + c_P$. Finally, let n_j and pr_j be registers for an acceptor with process index j where $1 \leq j \leq c_A$.

Acceptors are initialized using process indices from 1 to c_A and proposers are initialized using process indices from $c_A + 1$ to $c_A + c_P$.

Each proposer consists of one process, starts their own session and communicates with its corresponding quorum. The quorum contains the process indices of selected acceptors.

Each Acceptor consists of multiple subprocesses. One subprocess of every acceptor in a proposer's quorum participates in that proposer's session. Thus, a session has one proposer and multiple acceptor subprocesses.

The role of each participant in a session is given by the position of its process index in a sorted list. The process with the smallest index has role 1, the second-smallest index has role 2, and so on. Per definition proposers have a higher process index than any acceptor and obtain the highest role in the session. That role is always one more than the number of acceptors in its quorum.

For example, let $c_A = 3$ and $c_P = 2$, A_1, A_2, and A_3 are acceptors and P_4 and P_5 are proposers. Let P_5's quorum be $\mathrm{Q}_5 = [2, 3]$. A_2, A_3, and P_5 participate in P_5's session. A_2's role is 1, because its process index 2 is the first element in Q_5. A_3's role is 2, because its process index 3 is the second element in Q_5. P_5's role is $|\,\mathrm{Q}_5\,| + 1 = 2 + 1 = 3$.

$$
\begin{aligned}
\mathrm{Sys}\,(c_A, c_P) =& \overline{o}\,[2]\,(z)\,.\,\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(c_A + c_P, |\,\mathrm{Q}_{c_A+c_P}\,| + 1, \left\{1, \ldots, |\,\mathrm{Q}_{c_A+c_P}\,|\right\}, c_A, \mathrm{n}_{c_A+c_P}, []\right) \\
& |\; o\,[1]\,(z)\,.\Pi_{c_A+1 \leq k < c_A+c_P}\;\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(k, |\,\mathrm{Q}_k\,| + 1, \{1, \ldots, |\,\mathrm{Q}_k\,|\}, c_A, \mathrm{n}_k, []\right) \\
& |\; \Pi_{1 \leq j \leq c_A}\;\mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\left(j, c_A, c_P, \mathrm{n}_j, \mathrm{pr}_j\right)
\end{aligned}
$$

$$
\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(k, p, \mathrm{R}, c_A, \mathrm{n}_k, \overrightarrow{V}\right) = \overline{b_k}\,[p]\,(s)\,.\,\mathrm{P}^{\mathrm{P}}\left(s, k, p, \mathrm{R}, c_A, \mathrm{n}_k, \overrightarrow{V}\right)
$$

$$
\begin{aligned}
\mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\left(j, c_A, c_P, \mathrm{n}_j, \mathrm{pr}_j\right) =& \Pi_{c_A+1 \leq k \leq c_A+c_P}\;(\\
& \texttt{if}\; j \notin \mathrm{Q}_k \\
& \texttt{then}\; 0 \\
& \texttt{else}\; b_k\,[\mathrm{pos}\,(j, \mathrm{Q}_k)]\,(s)\,.\,\mathrm{P}^{\mathrm{A}}_1\left(s, \mathrm{pos}\,(j, \mathrm{Q}_k), |\,\mathrm{Q}_k\,| + 1, \mathrm{n}_j, \mathrm{pr}_j\right)\;)
\end{aligned}
$$

$\mathrm{Sys}\,(c_A, c_P)$, $\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(k, p, \mathrm{R}, c_A, \mathrm{n}_k, \overrightarrow{V}\right)$, and $\mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\left(j, c_A, c_P, \mathrm{n}_j, \mathrm{pr}_j\right)$ describe the system initialization.

An outer session is created through shared channel o and the acceptors and proposers are initialized.

$\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\left(k, p, \mathrm{R}, c_A, \mathrm{n}_k, \overrightarrow{V}\right)$ is initialized with the proposer's process index k, the proposer's role p in its session, a set of roles R,the number of acceptors c_A, register n_k, and a vector $\overrightarrow{V}$. k uniquely identifies a proposer when calling $\mathrm{propNumber}$. The proposer's role p is initialized with $|\,\mathrm{Q}_k\,| + 1$. The set of roles R contains the roles the acceptors in Q_k will use in the proposer's session. Register n_k is implemented as stable storage, which is preserved during failures. It is used as a counter to make sure every run of the Paxos algorithm uses a higher

proposal number. $\overrightarrow{V}$ is used in the proposer to collect and evaluate the responses from the acceptors. It is always initialized with an empty list []. Shared channel b_k is used to initiate a session. Afterwards, the process behaves like $\mathrm{P}^{\mathrm{P}}\left(s, k, p, \mathrm{R}, c_A, \mathrm{n}_k, \overrightarrow{V}\right)$. We assume a mechanism for electing a distinguished proposer, which acts as the leader [8]. The leader is the only proposer that can try issuing proposers. A new leader is elected via the same mechanism when the previous leader terminates or crashes.

$\mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\left(j, c_A, c_P, \mathrm{n}_j, \mathrm{pr}_j\right)$ is initialized with the acceptor's process index j, the number of acceptors c_A, the number of proposers c_P, initial knowledge for the highest promised proposal number in register n_j, if available, and initial knowledge for the most recently accepted proposal in register pr_j, if available. n_j holds a value of type $\mathrm{Maybe}\ \mathbb{N}$ and pr_j holds a value of type $\mathrm{Maybe}\ (\mathrm{Proposal}\ \mathrm{Value})$ thus both can contain $\mathrm{Nothing}$. We assume both registers are implemented via stable storage, which is preserved during failures. The proposers' session requests are accepted in separate subprocesses. These subprocesses run parallel to each other but still access the same registers n_j and pr_j. We observe that each subprocess in an acceptor accesses a different channel s, since it is generated by the proposer when its session request is accepted. Note that the role of an acceptor's subprocess is the position of the acceptor's process index in the corresponding proposer's quorum. Afterwards, each subprocess behaves like $\mathrm{P}^{\mathrm{A}}_1\left(s, a, p, \mathrm{n}_j, \mathrm{pr}_j\right)$.

3.4.2 Proposer

To define the proposer and the acceptor we introduce a function $\mathrm{update}\,(r, v)$ which replaces the value inside register r with v. We use this function to update the registers of the processes.

$$
\begin{aligned}
\mathrm{P}^{\mathrm{P}}\left(s, k, p, \mathrm{R}, c_A, \mathrm{n}_k, \overrightarrow{V}\right) = (\mu X)\ & \mathrm{update}\,(\mathrm{n}_k, \mathrm{n}_k + 1)\,. \\
& \left(\bigodot_{a \in \mathrm{R}} s\,[p, a]!_u l1a\,\langle \mathrm{propNumber}_k\,(\mathrm{n}_k)\rangle\right). \\
& \left(\bigodot_{a \in \mathrm{R}} s\,[p, a]?_u l1b\,\langle \bot\rangle\,(v_a)\right). \\
& \texttt{if}\ \mathrm{anyNack}\left(\overrightarrow{V}\right)\ \texttt{or}\ \left(\mathrm{promCount}\left(\overrightarrow{V}\right) < \left\lceil \frac{c_A + 1}{2} \right\rceil\right) \\
& \texttt{then}\ s\,[p, \mathrm{R}]!_w \mathit{Restart}.X \\
& \texttt{else} \\
& \quad s\,[p, \mathrm{R}]!_w \mathit{Accept}. \\
& \quad \left(\bigodot_{a \in \mathrm{R}} s\,[p, a]!_u l2a\left\langle \mathrm{Prop}\ \left(\mathrm{propNumber}_k\,(\mathrm{n}_k)\right)\ \left(\mathrm{promValue}\left(\overrightarrow{V}\right)\right)\right\rangle\right).0
\end{aligned}
$$

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

3.4.3 Acceptor

$$
\begin{aligned}
\mathrm{P}_1^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) = {} & (\mu X)\, s\,[a,p]?_u l1a\,\langle\bot\rangle\,\left(n'\right). \\
& \texttt{if } n' = \bot \\
& \texttt{then } s\,[a,p]!_u l1b\,\langle\bot\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) \\
& \texttt{else } \big(\\
& \quad \texttt{if } \mathrm{gt}\left(n',\mathrm{n}_j\right) \\
& \quad \texttt{then } \mathrm{update}\left(\mathrm{n}_j,\mathrm{Just}\ n'\right).s\,[a,p]!_u l1b\,\langle\mathrm{Promise}\ \mathrm{pr}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) \\
& \quad \texttt{else } s\,[a,p]!_u l1b\,\langle\mathrm{Nack}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)\big)
\end{aligned}
$$

$$
\mathrm{P}_2^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) = s\,[a,p]?_w\{\mathit{Accept}.\,\mathrm{P}_3^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0\}
$$

$$
\begin{aligned}
\mathrm{P}_3^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) = {} & s\,[a,p]?_u l2a\,\langle\bot\rangle\,\left(pr'\right). \\
& \texttt{if } pr' = \bot \texttt{ then } 0 \\
& \texttt{else } \big(\\
& \quad \texttt{if } \mathrm{ge}\left(\mathrm{nFromProp}\left(pr'\right),\mathrm{n}_j\right) \\
& \quad \texttt{then } \mathrm{update}\left(\mathrm{pr}_j,\mathrm{Just}\ pr'\right).\mathrm{update}\left(\mathrm{n}_j,\mathrm{Just}\ \left(\mathrm{nFromProp}\left(pr'\right)\right)\right).0 \\
& \quad \texttt{else } 0\big)
\end{aligned}
$$

An acceptor a has a corresponding subprocess for each proposer p with quorum Q_p where $a \in \mathrm{Q}_p$. These subprocesses behave like $\mathrm{P}_1^{\mathrm{A}}\left(s,a,p,\mathrm{n},\mathrm{pr}\right)$ and access the same registers n_j and pr_j. Updating these registers with $\mathrm{update}\left(r,v\right)$ updates them for all subprocesses of an acceptor.

Each subprocess can communicate with one proposer. Thus, if that proposer can not communicate with a particular subprocess of an acceptor there is no need for that subprocess.

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

Since the proposer's decision broadcast is weakly reliable the acceptor receives no decision only if the proposer crashed. In that 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 $\mathit{Restart}$ 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_j and pr_j if the proposal number in pr' is greater or equal to the value in n_j. Then the subprocess terminates. If the acceptor does not receive a proposal or the proposal number of pr' is less than the value in n_j the subprocess terminates without updating n_j or pr_j.

3.5 Failure Patterns

A failure detector is a subsystem that detects process failures and crashes. Chandra and Toueg introduce a class of failure detectors $\Diamond\mathscr{S}$ called *eventually strong* in [3]. 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 phase $1a$ with label $l1a$ the acceptors may suspect some proposers and in phase $1b$ with label $l1b$ the proposers may suspect some acceptors. Accordingly, $\texttt{FP}_{\texttt{uskip}}$ and $\texttt{FP}_{\texttt{ml}}$ are implemented with a failure detector in $\Diamond\mathscr{S}$ for phases $1a$ and $1b$. For phase $2a$ with label $l2a$ both $\texttt{FP}_{\texttt{uskip}}$ and $\texttt{FP}_{\texttt{ml}}$ return false to ensure the proposal sent by the proposer to the members of its quorum is not lost. Thus, the unreliable communication is treated like the weakly reliable broadcast before it.

For the weakly reliable broadcast in phase $2a$, the failure pattern $\texttt{FP}_{\texttt{wskip}}$ eventually returns true for subprocesses of acceptors if the corresponding proposer crashed.

We require that every correct proposer is able to communicate with a quorum for Paxos to work. Therefore, $\texttt{FP}_{\texttt{crash}}$ eventually returns true for a subprocess of an acceptor a only if the corresponding proposer with quorum Q either crashed or still has access to a quorum without that particular acceptor, i.e., $\mathrm{Q} \setminus \{a\}$ is still a quorum.

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

These failure patterns satisfy the Conditions 1.1–1.6.

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.

3.6.1 Scenario

Figure 3.1 provides an overview where A_1, A_2, and A_3 are the acceptors and P_4 and P_5 are the proposers. P_5 is elected to be the leader. In steps (1) to (5), P_5 completes the Paxos algorithm with A_2 and A_3 and terminates.

At this point A_2 has promised not to accept any proposal numbered less than 10 and has accepted the value abc. So, when P_4 tries to use 5 as its proposal number (6), it receives Nack from A_2 (8) and has to restart the algorithm (9).

P_4 then runs through the Paxos algorithm with A_1 and A_2 starting with a new prepare request (10) with a higher proposal number. In step (12) P_4 learns that value abc with proposal number 10 has already been accepted by A_2. Later, in step (14), 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 $\mathrm{Prop}\ 10\ \mathrm{abc}$.

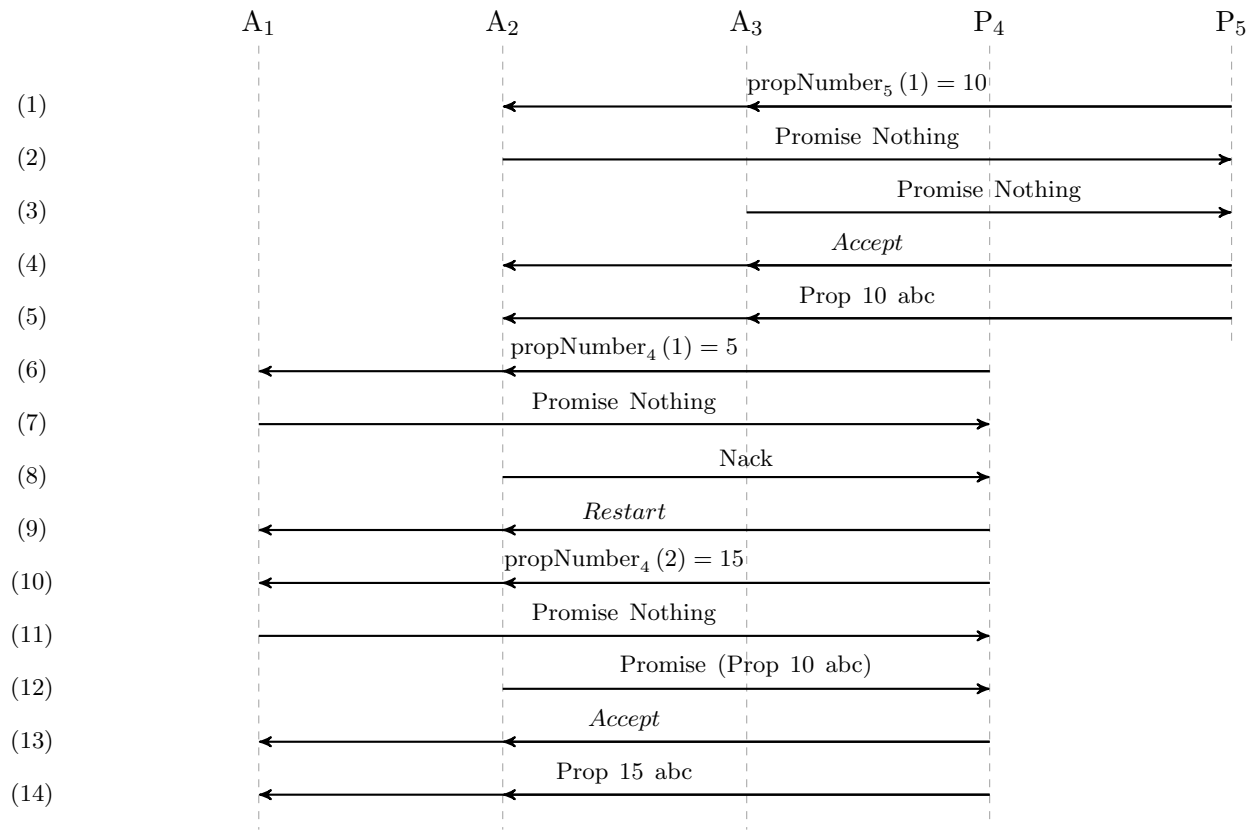

Figure 3.1: Example scenario with 3 acceptors and 2 proposers.

In the end all 3 acceptors have accepted the value abc. A_1 and A_2 have accepted $\mathrm{Prop}\ 15\ \mathrm{abc}$ and A_3 has accepted $\mathrm{Prop}\ 10\ \mathrm{abc}$.

3.6.2 Formulae

We set $c_A = 3$, $c_P = 2$, $\mathrm{Value} = \{\mathrm{abc}, \mathrm{def}, \ldots, \mathrm{vwx}, \mathrm{yz}\}$. $\mathrm{Q}_4 = \mathrm{gen\,Q}\,(4,3,2) = [1,2]$ will be the quorum for P_4 and $\mathrm{Q}_5 = \mathrm{gen\,Q}\,(5,3,2) = [2,3]$ will be the quorum for P_5.

Let n_1, n_2, n_3, n_4, n_5, pr_1, pr_2, and pr_3 be registers. n_1, n_2, and n_3 hold a value of type $\mathrm{Maybe}\ \mathbb{N}$. n_4 and n_5 hold a value of type $\mathbb{N}$. pr_1, pr_2, and pr_3 hold a value of type $\mathrm{Maybe}\ (\mathrm{Proposal}\ \mathrm{Value})$. n_j and pr_j correspond to acceptor $\mathrm{A_j}$ for $j \in \{1,2,3\}$. n_k corresponds to proposer $\mathrm{P_k}$ for $k \in \{4,5\}$. We set the following initial values.

A_1	A_2	A_3	P_4	P_5
$\mathrm{n}_1 = \mathrm{Nothing}$	$\mathrm{n}_2 = \mathrm{Nothing}$	$\mathrm{n}_3 = \mathrm{Nothing}$	$\mathrm{n}_4 = 0$	$\mathrm{n}_5 = 0$
$\mathrm{pr}_1 = \mathrm{Nothing}$	$\mathrm{pr}_2 = \mathrm{Nothing}$	$\mathrm{pr}_3 = \mathrm{Nothing}$		

System Initialization

$$\begin{aligned}\mathrm{Sys}\,(3,2) = &\overline{o}\,[2]\,(z)\,.\,\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(5,3,\{1,2\},3,\mathrm{n}_5,[]) \\ &\mid o\,[1]\,(z)\,.\Pi_{3<k<5}\ \mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(k,3,\{1,2\},3,\mathrm{n}_k,[]) \\ &\mid \Pi_{1\le j\le 3}\ \mathrm{P}^{\mathrm{A}}_{\mathrm{init}}\,(j,3,2,\mathrm{n}_j,\mathrm{pr}_j)\end{aligned}$$

After inserting c_A and c_P and applying (Init) once for shared channel o we have:

$$\begin{aligned}&\longmapsto^{*} (\nu z)\,(\\ &\quad \overline{b_5}\,[3]\,(r)\,.\,\mathrm{P}^{\mathrm{P}}\,(r,5,3,\{1,2\},3,\mathrm{n}_5,[])\ = \mathrm{P}_5 \\ &\quad \mid \overline{b_4}\,[3]\,(s)\,.\,\mathrm{P}^{\mathrm{P}}\,(s,4,3,\{1,2\},3,\mathrm{n}_4,[])\ = \mathrm{P}_4 \\ &\quad \mid (\texttt{if}\ 1 \notin \{1,2\}\ \texttt{then}\ 0\ \texttt{else}\ b_4\,[1]\,(s)\ldots \mid \texttt{if}\ 1 \notin \{2,3\}\ \texttt{then}\ 0\texttt{else}\ \ldots)\ = \mathrm{A}_1 \\ &\quad \mid (\texttt{if}\ 2 \notin \{1,2\}\ \texttt{then}\ 0\ \texttt{else}\ b_4\,[2]\,(s)\ldots \mid \texttt{if}\ 2 \notin \{2,3\}\ \texttt{then}\ 0\ \texttt{else}\ b_5\,[1]\,(r)\ldots)\ = \mathrm{A}_2 \\ &\quad \mid (\texttt{if}\ 3 \notin \{1,2\}\ \texttt{then}\ 0\ \texttt{else}\ \ldots \mid \texttt{if}\ 3 \notin \{2,3\}\ \texttt{then}\ 0\ \texttt{else}\ b_5\,[2]\,(r)\ldots)\ = \mathrm{A}_3 \\ &\quad \mid \Pi_{1\le k,l\le 2,k\ne l}\ z_{k\to l} : [])\end{aligned}$$

We apply $(\mathrm{If\text{-}T})$ to the left subprocess of A_3 and the right subprocess of A_1 and terminate them. All other acceptor subprocesses advance by applying $(\mathrm{If\text{-}F})$. Note that each process is shortened to only show the next few steps instead of the entire process.

$$
\begin{aligned}
\longmapsto^{*} & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& \overline{b_5}\,[3]\,(r)\,.\,\mathrm{P}^{\mathrm{P}}\,(r, 5, 3, \{1,2\}, 3, \mathrm{n}_5, []) \;= \mathrm{P}_5 \\
& \mid \overline{b_4}\,[3]\,(s)\,.\,\mathrm{P}^{\mathrm{P}}\,(s, 4, 3, \{1,2\}, 3, \mathrm{n}_4, []) \;= \mathrm{P}_4 \\
& \mid b_4\,[1]\,(s)\,.\,\mathrm{P}_1^{\mathrm{A}}\,(s, 1, 3, \mathrm{n}_1, \mathrm{pr}_1) \;= \mathrm{A}_1 \\
& \mid \left(b_4\,[2]\,(s)\,.\,\mathrm{P}_1^{\mathrm{A}}\,(s, 2, 3, \mathrm{n}_2, \mathrm{pr}_2) \;\mid b_5\,[1]\,(r)\,.\,\mathrm{P}_1^{\mathrm{A}}\,(r, 1, 3, \mathrm{n}_2, \mathrm{pr}_2)\right) \;= \mathrm{A}_2 \\
& \mid b_5\,[2]\,(r)\,.\,\mathrm{P}_1^{\mathrm{A}}\,(r, 2, 3, \mathrm{n}_3, \mathrm{pr}_3) \;= \mathrm{A}_3 \\
& \mid \Pi_{1 \le k,l \le 2, k \neq l}\; z_{k \to l} : []\big)
\end{aligned}
$$

We apply (Init) for each shared channel b_4 and b_5 and unfold the calls to $\mathrm{P}^{\mathrm{P}}\left(s, k, p, \mathrm{R}, c_A, \mathrm{n}, \overrightarrow{V}\right)$ and $\mathrm{P}_1^{\mathrm{A}}\,(s, a, p, \mathrm{n}, \mathrm{pr})$ to obtain:

$$
\begin{aligned}
\longmapsto^{*} & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& (\mu X)\,\mathrm{update}\,(\mathrm{n}_5, 0+1)\,.\,\left(\bigodot_{a \in \{1,2\}} r\,[3, a]!_u l1a\,\langle \mathrm{propNumber}_5\,(\mathrm{n}_5)\rangle\right)\ldots \;= \mathrm{P}_5 \\
& \mid\; (\mu X)\,\mathrm{update}\,(\mathrm{n}_4, 0+1)\,.\,\left(\bigodot_{a \in \{1,2\}} s\,[3, a]!_u l1a\,\langle \mathrm{propNumber}_4\,(\mathrm{n}_4)\rangle\right)\ldots \;= \mathrm{P}_4 \\
& \mid\; (\mu X)\,s\,[1,3]?_u l1a\,\langle\bot\rangle\,(n')\,\texttt{.if}\;\ldots \;= \mathrm{A}_1 \\
& \mid\; \left((\mu X)\,s\,[2,3]?_u l1a\,\langle\bot\rangle\,(n')\,\texttt{.if}\;\ldots \;\mid\; (\mu X)\,r\,[1,3]?_u l1a\,\langle\bot\rangle\,(n')\,\texttt{.if}\;\ldots\right) \;= \mathrm{A}_2 \\
& \mid\; (\mu X)\,r\,[2,3]?_u l1a\,\langle\bot\rangle\,(n')\,\texttt{.if}\;\ldots \;= \mathrm{A}_3 \\
& \mid \Pi_{1 \le k,l \le 3, k \neq l}\; s_{k \to l} : [] \mid \Pi_{1 \le k,l \le 3, k \neq l}\; r_{k \to l} : [] \mid \Pi_{1 \le k,l \le 2, k \neq l}\; z_{k \to l} : []\big)
\end{aligned}
$$

The Happy Path

We apply (Rec) to every process and (Func) to both proposers. Now, the first inter-process communication can take place. In this case P_5 communicates with A_2 and A_3. We apply (USend) and (UGet) twice to send $\mathrm{propNumber}_5\,(1) = 10$ to A_2 and A_3.

$$
\begin{aligned}
\longmapsto^{*} & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& r\,[3,1]?_u l1b\,\langle\bot\rangle\,(v_1)\,.r\,[3,2]?_u l1b\,\langle\bot\rangle\,(v_2)\ldots \;= \mathrm{P}_5 \\
& \mid s\,[3,1]!_u l1a\,\langle \mathrm{propNumber}_4\,(1)\rangle\,.s\,[3,2]!_u l1a\,\langle \mathrm{propNumber}_4\,(1)\rangle\ldots \;= \mathrm{P}_4 \\
& \mid s\,[1,3]?_u l1a\,\langle\bot\rangle\,(n')\,\texttt{.if}\;\ldots \;= \mathrm{A}_1 \\
& \mid \left(s\,[2,3]?_u l1a\,\langle\bot\rangle\,(n')\,\texttt{.if}\;\ldots \mid \texttt{if}\;10 = \bot\;\texttt{then}\;\ldots\;\texttt{else}\;(\texttt{if}\;\mathrm{gt}\,(10, \mathrm{n}_2)\ldots)\right) \;= \mathrm{A}_2 \\
& \mid \texttt{if}\;10 = \bot\;\texttt{then}\;\ldots\;\texttt{else}\;\left(\texttt{if}\;\mathrm{gt}\,(n', \mathrm{n}_3)\;\texttt{then}\;\mathrm{update}\,(\mathrm{n}_3, n')\ldots\;\texttt{else}\;\ldots\right) \;= \mathrm{A}_3 \\
& \mid \Pi_{1 \le k,l \le 3, k \neq l}\; s_{k \to l} : [] \mid \Pi_{1 \le k,l \le 3, k \neq l}\; r_{k \to l} : [] \mid \Pi_{1 \le k,l \le 2, k \neq l}\; z_{k \to l} : []\big)
\end{aligned}
$$

Since $10 \neq \bot$ both A_2 and A_3 move into their respective `else` branches by applying (If-F).

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& r\,[3,1]?_u l1b\,\langle\bot\rangle\,(v_1)\,.r\,[3,2]?_u l1b\,\langle\bot\rangle\,(v_2)\ldots = \mathrm{P}_5 \\
& \mid s\,[3,1]!_u l1a\,\langle \mathrm{propNumber}_4\,(1)\rangle\,.s\,[3,2]!_u l1a\,\langle \mathrm{propNumber}_4\,(1)\rangle\ldots = \mathrm{P}_4 \\
& \mid s\,[1,3]?_u l1a\,\langle\bot\rangle\,(n')\,.\texttt{if}\ \ldots = \mathrm{A}_1 \\
& \mid \big(s\,[2,3]?_u l1a\,\langle\bot\rangle\,(n')\,.\texttt{if}\ \ldots \mid \texttt{if}\ \mathrm{gt}\,(10,\mathrm{Nothing})\ \texttt{then}\ \ldots\ \texttt{else}\ \ldots\big) = \mathrm{A}_2 \\
& \mid \texttt{if}\ \mathrm{gt}\,(10,\mathrm{Nothing})\ \texttt{then}\ \mathrm{update}\,\big(\mathrm{n}_3,\mathrm{Just}\ n'\big)\,.r\,[2,3]!_u l1b\,\langle\mathrm{Promise}\ \mathrm{pr}_3\rangle\ldots\ \texttt{else}\ \ldots = \mathrm{A}_3 \\
& \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 3,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\big)
\end{aligned}
$$

Because $\mathrm{gt}\,(10,\mathrm{Nothing})$ returns true, A_2 and A_3 move into their respective `then` branches by applying (If-T). After executing $\mathrm{update}\,(\mathrm{n}_2,\mathrm{Just}\ 10)$ and $\mathrm{update}\,(\mathrm{n}_3,\mathrm{Just}\ 10)$ with (Func), A_2 and A_3 are ready to send their responses to P_5.

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& r\,[3,1]?_u l1b\,\langle\bot\rangle\,(v_1)\,.r\,[3,2]?_u l1b\,\langle\bot\rangle\,(v_2)\,.\texttt{if}\ \mathrm{anyNack}\left(\overrightarrow{V}\right)\ \texttt{or}\ \mathrm{promCount}\left(\overrightarrow{V}\right) < 2\ldots = \mathrm{P}_5 \\
& \mid s\,[3,1]!_u l1a\,\langle \mathrm{propNumber}_4\,(1)\rangle\,.s\,[3,2]!_u l1a\,\langle \mathrm{propNumber}_4\,(1)\rangle\ldots = \mathrm{P}_4 \\
& \mid s\,[1,3]?_u l1a\,\langle\bot\rangle\,(n')\,.\texttt{if}\ \ldots = \mathrm{A}_1 \\
& \mid \big(s\,[2,3]?_u l1a\,\langle\bot\rangle\,(n')\,.\texttt{if}\ \ldots \mid r\,[1,3]!_u l1b\,\langle\mathrm{Promise}\ \mathrm{Nothing}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\,(r,1,3,\mathrm{n}_2,\mathrm{pr}_2)\big) = \mathrm{A}_2 \\
& \mid r\,[2,3]!_u l1b\,\langle\mathrm{Promise}\ \mathrm{Nothing}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\,(r,2,3,\mathrm{n}_3,\mathrm{pr}_3) = \mathrm{A}_3 \\
& \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 3,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\big)
\end{aligned}
$$

We apply (USend) and (UGet) twice to do just that. A_3 and one subprocess of A_2 move into $\mathrm{P}_2^{\mathrm{A}}\,(s,a,p,\mathrm{n},\mathrm{pr})$. P_5 has $\overrightarrow{V} = [\mathrm{Promise}\ \mathrm{Nothing},\mathrm{Promise}\ \mathrm{Nothing}]$. $\mathrm{anyNack}\left(\overrightarrow{V}\right)$ returns false and $\mathrm{promCount}\left(\overrightarrow{V}\right)$ returns 2. Thus, $\mathrm{anyNack}\left(\overrightarrow{V}\right)\ \texttt{or}\ \mathrm{promCount}\left(\overrightarrow{V}\right) < 2$ evaluates to false. By applying (If-F) P_5 advances to its `else` branch.

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& r\,[3,\{1,2\}]!_w \mathit{Accept}.r\,[3,1]!_u l2a\,\langle\mathrm{Prop}\ 10\ \mathrm{abc}\rangle\ldots = \mathrm{P}_5 \\
& \mid s\,[3,1]!_u l1a\,\langle \mathrm{propNumber}_4\,(1)\rangle\,.s\,[3,2]!_u l1a\,\langle \mathrm{propNumber}_4\,(1)\rangle\ldots = \mathrm{P}_4 \\
& \mid s\,[1,3]?_u l1a\,\langle\bot\rangle\,(n')\,.\texttt{if}\ \ldots = \mathrm{A}_1 \\
& \mid \big(s\,[2,3]?_u l1a\,\langle\bot\rangle\,(n')\,.\texttt{if}\ \ldots \mid r\,[1,3]?_w\{\mathit{Accept}\ldots \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0\}\big) = \mathrm{A}_2 \\
& \mid r\,[2,3]?_w\{\mathit{Accept}.\,\mathrm{P}_3^{\mathrm{A}}\,(r,2,3,\mathrm{n}_3,\mathrm{pr}_3) \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0\} = \mathrm{A}_3 \\
& \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 3,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\big)
\end{aligned}
$$

P_5 broadcasts its decision *Accept* to A_2 and A_3. By applying (WSel) once, (WBran) twice we obtain:

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& r\,[3,1]!_u l2a\,\langle \text{Prop } 10 \text{ abc}\rangle\,.r\,[3,2]!_u l2a\,\langle \text{Prop } 10 \text{ abc}\rangle\,.0 = \mathrm{P}_5\\
& \mid s\,[3,1]!_u l1a\,\langle \text{propNumber}_4\,(1)\rangle\,.s\,[3,2]!_u l1a\,\langle \text{propNumber}_4\,(1)\rangle \ldots = \mathrm{P}_4\\
& \mid s\,[1,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\texttt{if } \ldots = \mathrm{A}_1\\
& \mid \left(s\,[2,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\texttt{if } \ldots \mid r\,[1,3]?_u l2a\,\langle \bot\rangle\,(pr')\,.\texttt{if } \ldots\right) = \mathrm{A}_2\\
& \mid r\,[2,3]?_u l2a\,\langle \bot\rangle\,(pr')\,.\texttt{if } pr' = \bot \texttt{ then } 0 \texttt{ else } (\texttt{if } \ldots) = \mathrm{A}_3\\
& \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 3,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\,)
\end{aligned}
$$

Now P_5 can send its proposal to A_2 and A_3 and terminate. To do so we apply (USend) and (UGet) twice.

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& \mid s\,[3,1]!_u l1a\,\langle \text{propNumber}_4\,(1)\rangle\,.s\,[3,2]!_u l1a\,\langle \text{propNumber}_4\,(1)\rangle \ldots = \mathrm{P}_4\\
& \mid s\,[1,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\texttt{if } \ldots = \mathrm{A}_1\\
& \mid \left(s\,[2,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\texttt{if } \ldots \mid \texttt{if } (\text{Prop } 10 \text{ abc}) = \bot \texttt{ then } 0 \texttt{ else } (\texttt{if } \ldots)\right) = \mathrm{A}_2\\
& \mid \texttt{if } (\text{Prop } 10 \text{ abc}) = \bot \texttt{ then } 0 \texttt{ else } \left(\texttt{if } \text{ge}\,(10, \mathrm{n}_3) \texttt{ then } \text{update}\left(\mathrm{pr}_3, \text{Just } pr'\right) \ldots \texttt{ else } 0\right) = \mathrm{A}_3\\
& \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 3,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\,)
\end{aligned}
$$

Because $(\text{Prop } 10 \text{ abc}) = \bot$ evaluates to false we can apply (If-F) to A_2 and A_3. Then, we can apply (If-T) to A_2 and A_3 because $\text{ge}\,(10, \text{Just } 10)$ returns true.

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& \mid s\,[3,1]!_u l1a\,\langle \text{propNumber}_4\,(1)\rangle\,.s\,[3,2]!_u l1a\,\langle \text{propNumber}_4\,(1)\rangle \ldots = \mathrm{P}_4\\
& \mid s\,[1,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\texttt{if } \ldots = \mathrm{A}_1\\
& \mid \left(s\,[2,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\texttt{if } \ldots \mid \text{update}\,(\mathrm{pr}_2, \text{Just } (\text{Prop } 10 \text{ abc}))\,.\,\text{update}\,(\mathrm{n}_2, \text{Just } 10)\,.0\right) = \mathrm{A}_2\\
& \mid \text{update}\,(\mathrm{pr}_3, \text{Just } (\text{Prop } 10 \text{ abc}))\,.\,\text{update}\,(\mathrm{n}_3, \text{Just } 10)\,.0 = \mathrm{A}_3\\
& \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 3,k\neq l}\ r_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\,)
\end{aligned}
$$

A_2 and A_3 terminate after applying (Func) twice and updating their registers. They have accepted the proposal. P_4 is the new leader.

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& \mid s\,[3,1]!_u l1a\,\langle 5\rangle\,.s\,[3,2]!_u l1a\,\langle 5\rangle \ldots = \mathrm{P}_4\\
& \mid s\,[1,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\texttt{if } n' = \bot \texttt{ then } \ldots \texttt{ else } \left(\texttt{if } \text{gt}\,(n', \mathrm{n}_1) \ldots\right) = \mathrm{A}_1\\
& \mid s\,[2,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\texttt{if } n' = \bot \texttt{ then } \ldots \texttt{ else } \left(\texttt{if } \text{gt}\,(n', \mathrm{n}_2) \ldots\right) = \mathrm{A}_2\\
& \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\,)
\end{aligned}
$$

At this point the registers hold the following values.

A_1	A_2	A_3	P_4	P_5
$n_1 = \text{Nothing}$	$n_2 = \text{Just } 10$	$n_3 = \text{Just } 10$	$n_4 = 1$	$n_5 = 1$
$pr_1 = \text{Nothing}$	$pr_2 = \text{Just (Prop 10 abc)}$	$pr_3 = \text{Just (Prop 10 abc)}$		

Restarting the Algorithm

Next, P_4 sends prepare requests with a proposal number less than 10, which A_2 rejects. P_4 then decides to restart the algorithm. We apply (USend) and (UGet) twice.

$$
\begin{aligned}
&\longmapsto^* (\nu z)\,(\nu s)\,(\nu r)\,(\\
&\quad \mid s\,[3,1]?_u l1b\,\langle\bot\rangle\,(v_1)\,.s\,[3,2]?_u l1b\,\langle\bot\rangle\,(v_2)\,.\ldots \;=\; P_4\\
&\quad \mid \texttt{if } 5 = \bot \texttt{ then } \ldots \texttt{ else } (\texttt{if } \mathrm{gt}\,(5, n_1) \texttt{ then } \mathrm{update}\,(n_1, \text{Just } 5)\ldots \texttt{ else } \ldots) \;=\; A_1\\
&\quad \mid \texttt{if } 5 = \bot \texttt{ then } \ldots \texttt{ else } (\texttt{if } \mathrm{gt}\,(5, n_2) \texttt{ then } \ldots \texttt{ else } s\,[2,3]!_u l1b\,\langle\text{Nack}\rangle\,.\,P_2^A\,(s, 2, 3, n_2, pr_2)) \;=\; A_2\\
&\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\; s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\; z_{k\to l} : [])
\end{aligned}
$$

Because $5 = \bot$ evaluates to false we apply (If-F) to A_1 and A_2. We apply (If-F) to A_2 again because $\mathrm{gt}\,(5, \text{Just } 10)$ returns false. $\mathrm{gt}\,(5, \text{Nothing})$ returns true, so we advance A_1 to its `then` branch by applying (If-T).

$$
\begin{aligned}
&\longmapsto^* (\nu z)\,(\nu s)\,(\nu r)\,(\\
&\quad \mid s\,[3,1]?_u l1b\,\langle\bot\rangle\,(v_1)\,.s\,[3,2]?_u l1b\,\langle\bot\rangle\,(v_2)\,.\ldots \;=\; P_4\\
&\quad \mid \mathrm{update}\,(n_1, \text{Just } 5)\,.s\,[1,3]!_u l1b\,\langle\text{Promise Nothing}\rangle\,.\,P_2^A\,(s, 1, 3, n_1, pr_1) \;=\; A_1\\
&\quad \mid s\,[2,3]!_u l1b\,\langle\text{Nack}\rangle\,.\,P_2^A\,(s, 2, 3, n_2, pr_2) \;=\; A_2\\
&\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\; s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\; z_{k\to l} : [])
\end{aligned}
$$

We apply (Func) to A_1 to update its register n_1. Applying (USend) and (UGet) twice then yields:

$$
\begin{aligned}
&\longmapsto^* (\nu z)\,(\nu s)\,(\nu r)\,(\\
&\quad \mid \texttt{if } \mathrm{anyNack}\left(\overrightarrow{V}\right) \texttt{ or } \mathrm{promCount}\left(\overrightarrow{V}\right) < 2 \texttt{ then } s\,[3,\{1,2\}]!_w Restart.X \texttt{ else } \ldots \;=\; P_4\\
&\quad \mid s\,[1,3]?_w\{Accept.\,P_3^A\,(s, 1, 3, n_1, pr_1) \oplus Restart.X \oplus Abort.0\} \;=\; A_1\\
&\quad \mid s\,[2,3]?_w\{Accept.\,P_3^A\,(s, 2, 3, n_2, pr_2) \oplus Restart.X \oplus Abort.0\} \;=\; A_2\\
&\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\; s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\; z_{k\to l} : [])
\end{aligned}
$$

P_4 has $\overrightarrow{V} = [\text{Promise Nothing}, \text{Nack}]$. $\mathrm{anyNack}\left(\overrightarrow{V}\right)$ returns true and $\mathrm{promCount}\left(\overrightarrow{V}\right)$ returns 1. Thus, $\mathrm{anyNack}\left(\overrightarrow{V}\right)$ `or` $\mathrm{promCount}\left(\overrightarrow{V}\right) < 2$ evaluates to true. We apply (If-T) to P_4.

$$
\begin{aligned}
&\longmapsto^{*} (\nu z)\,(\nu s)\,(\nu r)\,(\\
&\quad s\,[3,\{1,2\}]!_w \mathit{Restart}.\,(\mu X)\,\mathrm{update}\,(\mathrm{n}_4, \mathrm{n}_4+1)\,.s\,[3,1]!_u l1a\,\langle \mathrm{propNumber}_4\,(\mathrm{n}_4)\rangle \ldots &&= \mathrm{P}_4\\
&\quad \mid s\,[1,3]?_w\{\mathit{Accept}\ldots \oplus \mathit{Restart}.\,(\mu X)\,s\,[1,3]?_u l1a\,\langle \bot\rangle\,(n')\ldots \oplus \mathit{Abort}.0\} &&= \mathrm{A}_1\\
&\quad \mid s\,[2,3]?_w\{\mathit{Accept}\ldots \oplus \mathit{Restart}.\,(\mu X)\,s\,[2,3]?_u l1a\,\langle \bot\rangle\,(n')\ldots \oplus \mathit{Abort}.0\} &&= \mathrm{A}_2\\
&\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : [])
\end{aligned}
$$

P_4 sends its decision to restart the algorithm to A_1 and A_2 by applying (WSel) once and (WBran) twice.

$$
\begin{aligned}
&\longmapsto^{*} (\nu z)\,(\nu s)\,(\nu r)\,(\\
&\quad (\mu X)\,\mathrm{update}\,(\mathrm{n}_4, 2)\,.s\,[3,1]!_u l1a\,\langle 15\rangle\,.s\,[3,2]!_u l1a\,\langle 15\rangle\,.s\,[3,1]?_u l1b\,\langle \bot\rangle\,(v_1)\ldots &&= \mathrm{P}_4\\
&\quad \mid (\mu X)\,s\,[1,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\mathtt{if}\ n' = \bot\ \mathtt{then}\ \ldots\ \mathtt{else}\ (\mathtt{if}\ \mathrm{gt}\,(n', \mathrm{n}_1)\ldots) &&= \mathrm{A}_1\\
&\quad \mid (\mu X)\,s\,[2,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\mathtt{if}\ n' = \bot\ \mathtt{then}\ \ldots\ \mathtt{else}\ (\mathtt{if}\ \mathrm{gt}\,(n', \mathrm{n}_2)\ldots) &&= \mathrm{A}_2\\
&\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : [])
\end{aligned}
$$

At this point the registers hold the following values.

A_1	A_2	A_3	P_4	P_5
$\mathrm{n}_1 = \mathrm{Just}\ 5$	$\mathrm{n}_2 = \mathrm{Just}\ 10$	$\mathrm{n}_3 = \mathrm{Just}\ 10$	$\mathrm{n}_4 = 1$	$\mathrm{n}_5 = 1$
$\mathrm{pr}_1 = \mathrm{Nothing}$	$\mathrm{pr}_2 = \mathrm{Just}\ (\mathrm{Prop}\ 10\ \mathrm{abc})$	$\mathrm{pr}_3 = \mathrm{Just}\ (\mathrm{Prop}\ 10\ \mathrm{abc})$		

The Happy Path, Again

First, we apply (Rec) to each process and then (Func) to P_4.

$$
\begin{aligned}
&\longmapsto^{*} (\nu z)\,(\nu s)\,(\nu r)\,(\\
&\quad s\,[3,1]!_u l1a\,\langle 15\rangle\,.s\,[3,2]!_u l1a\,\langle 15\rangle\,.s\,[3,1]?_u l1b\,\langle \bot\rangle\,(v_1)\ldots &&= \mathrm{P}_4\\
&\quad \mid s\,[1,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\mathtt{if}\ n' = \bot\ \mathtt{then}\ \ldots\ \mathtt{else}\ (\mathtt{if}\ \mathrm{gt}\,(n', \mathrm{n}_1)\ldots) &&= \mathrm{A}_1\\
&\quad \mid s\,[2,3]?_u l1a\,\langle \bot\rangle\,(n')\,.\mathtt{if}\ n' = \bot\ \mathtt{then}\ \ldots\ \mathtt{else}\ (\mathtt{if}\ \mathrm{gt}\,(n', \mathrm{n}_2)\ldots) &&= \mathrm{A}_2\\
&\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : [])
\end{aligned}
$$

This time P_4 uses a high enough proposal number so that A_1 and A_2 both promise not to accept any proposal numbered less than that. $15 = \bot$ evaluates to false and $\mathrm{gt}\,(15, \mathrm{Just}\ 10)$ and $\mathrm{gt}\,(15, \mathrm{Just}\ 5)$ return true. By applying (USend) twice in the remaining proposer and (UGet), (If-F), and (If-T) in the remaining acceptors we arrive at:

$$
\begin{array}{l}
\longmapsto^{*} (\nu z)\,(\nu s)\,(\nu r)\,(\\
\quad s\,[3,1]?_u l1b\,\langle \bot \rangle\,(v_1)\,.s\,[3,2]?_u l1b\,\langle \bot \rangle\,(v_2)\,.\texttt{if}\ \ldots = \mathrm{P}_4\\
\quad \mid \text{update}\,(\mathrm{n}_1, \text{Just}\ 15)\,.s\,[1,3]!_u l1b\,\langle \text{Promise Nothing} \rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\,(s,1,3,\mathrm{n}_1,\mathrm{pr}_1) = \mathrm{A}_1\\
\quad \mid \text{update}\,(\mathrm{n}_2, \text{Just}\ 15)\,.s\,[2,3]!_u l1b\,\langle \text{Promise (Prop 10 } abc) \rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\,(s,2,3,\mathrm{n}_2,\mathrm{pr}_2) = \mathrm{A}_2\\
\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\,)
\end{array}
$$

A_1 and A_2 update their respective registers n to Just 15 by applying (Func). Because A_2 has already accepted a proposal, it responds to P_4's prepare request with that proposal. Twice more we apply (USend) and (UGet) to obtain:

$$
\begin{array}{l}
\longmapsto^{*} (\nu z)\,(\nu s)\,(\nu r)\,(\\
\quad \texttt{if}\ \text{anyNack}\left(\overrightarrow{V}\right)\ \texttt{or}\ \text{promCount}\left(\overrightarrow{V}\right) < 2\ \texttt{then}\ \ldots\ \texttt{else}\ s\,[3,\{1,2\}]!_w \mathit{Accept}\ldots = \mathrm{P}_4\\
\quad \mid s\,[1,3]?_w\{\mathit{Accept}.\,\mathrm{P}_3^{\mathrm{A}}\,(s,1,3\,\mathrm{n}_1,\mathrm{pr}_1) \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0\} = \mathrm{A}_1\\
\quad \mid s\,[2,3]?_w\{\mathit{Accept}.\,\mathrm{P}_3^{\mathrm{A}}\,(s,2,3,\mathrm{n}_2,\mathrm{pr}_2) \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0\} = \mathrm{A}_2\\
\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\,)
\end{array}
$$

P_4 has $\overrightarrow{V} = [\text{Promise Nothing}, \text{Promise (Just (Prop 10 abc))}]$. $\text{anyNack}\left(\overrightarrow{V}\right)\ \texttt{or}\ \text{promCount}\left(\overrightarrow{V}\right) < 2$ evaluates to false and we can apply (If-F).

$$
\begin{array}{l}
\longmapsto^{*} (\nu z)\,(\nu s)\,(\nu r)\,(\\
\quad s\,[3,\{1,2\}]!_w \mathit{Accept}.s\,[3,1]!_u l2a\,\langle \text{Prop 15 abc} \rangle\,.s\,[3,2]!_u l2a\,\langle \text{Prop 15 abc} \rangle\,.0 = \mathrm{P}_4\\
\quad \mid s\,[1,3]?_w\{\mathit{Accept}.\,\mathrm{P}_3^{\mathrm{A}}\,(s,1,3\,\mathrm{n}_1,\mathrm{pr}_1) \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0\} = \mathrm{A}_1\\
\quad \mid s\,[2,3]?_w\{\mathit{Accept}.\,\mathrm{P}_3^{\mathrm{A}}\,(s,2,3,\mathrm{n}_2,\mathrm{pr}_2) \oplus \mathit{Restart}.X \oplus \mathit{Abort}.0\} = \mathrm{A}_2\\
\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\,)
\end{array}
$$

P_4 has received enough promises to send its own proposal. The value for that proposal is abc because that is the value of the highest-numbered proposal P_4 received as a response to its prepare request. First, we apply (WSel) and (WBran).

$$
\begin{array}{l}
\longmapsto^{*} (\nu z)\,(\nu s)\,(\nu r)\,(\\
\quad s\,[3,1]!_u l2a\,\langle \text{Prop 15 abc} \rangle\,.s\,[3,2]!_u l2a\,\langle \text{Prop 15 abc} \rangle\,.0 = \mathrm{P}_4\\
\quad \mid s\,[1,3]?_u l2a\,\langle \bot \rangle\,(pr')\,.\texttt{if}\ pr' = \bot\ \texttt{then}\ 0\ \texttt{else}\ \left(\texttt{if}\ \text{ge}\left(\text{nFromProp}\left(pr'\right), \mathrm{n}_1\right)\ldots\right) = \mathrm{A}_1\\
\quad \mid s\,[2,3]?_u l2a\,\langle \bot \rangle\,(pr')\,.\texttt{if}\ pr' = \bot\ \texttt{then}\ 0\ \texttt{else}\ \left(\texttt{if}\ \text{ge}\left(\text{nFromProp}\left(pr'\right), \mathrm{n}_2\right)\ldots\right) = \mathrm{A}_2\\
\quad \mid \Pi_{1\le k,l\le 3,k\neq l}\ s_{k\to l} : [] \mid \Pi_{1\le k,l\le 2,k\neq l}\ z_{k\to l} : []\,)
\end{array}
$$

Then we apply (USend) and (UGet) to send the proposal from P_4 to the acceptors. P_4 terminates.

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& \texttt{if}\ (\text{Prop}\ \ 15\ \ \text{abc}) = \bot\ \texttt{then}\ 0\ \texttt{else}\ \left(\texttt{if}\ \ \text{ge}\,(15, \text{n}_1)\ \ \texttt{then}\ \ \text{update}\left(\text{pr}_1, \text{Just}\ \ pr'\right) \ldots\ \texttt{else}\ 0\right)\ = \text{A}_1\\
& \mid \texttt{if}\ \ (\text{Prop}\ \ 15\ \ \text{abc}) = \bot\ \texttt{then}\ 0\ \texttt{else}\ \left(\texttt{if}\ \ \text{ge}\,(15, \text{n}_2)\ \ \texttt{then}\ \ \text{update}\left(\text{pr}_2, \text{Just}\ \ pr'\right) \ldots\ \texttt{else}\ 0\right)\ = \text{A}_2\\
& \mid \Pi_{1 \leq k,l \leq 3, k \neq l}\ s_{k \to l} : [] \mid \Pi_{1 \leq k,l \leq 2, k \neq l}\ z_{k \to l} : []\big)
\end{aligned}
$$

$(\text{Prop}\ \ 15\ \ \text{abc}) = \bot$ evaluates to false and $\text{ge}\,(15, \text{Just}\ \ 15)$ returns true. So, we apply (If-F) and then (If-T) to the acceptors.

$$
\begin{aligned}
\longmapsto^* & (\nu z)\,(\nu s)\,(\nu r)\,(\\
& \text{update}\left(\text{pr}_1, \text{Just}\ \ pr'\right).\text{update}\left(\text{n}_1, \text{Just}\ \ \left(\text{nFromProp}\left(pr'\right)\right)\right).0 = \text{A}_1\\
& \mid\ \text{update}\left(\text{pr}_2, \text{Just}\ \ pr'\right).\text{update}\left(\text{n}_2, \text{Just}\ \ \left(\text{nFromProp}\left(pr'\right)\right)\right).0 = \text{A}_2\\
& \mid \Pi_{1 \leq k,l \leq 3, k \neq l}\ s_{k \to l} : [] \mid \Pi_{1 \leq k,l \leq 2, k \neq l}\ z_{k \to l} : []\big)
\end{aligned}
$$

Finally, we apply (Func) twice to each acceptor. With that, A_1 and A_2 have accepted P_4's proposal.

$$
\longmapsto^*\ (\nu z)\,(\nu s)\,(\nu r)\,0
$$

The final values the registers contain are as follows.

A_1	A_2	A_3	P_4	P_5
$\text{n}_1 = \text{Just}\ \ 15$	$\text{n}_2 = \text{Just}\ \ 15$	$\text{n}_3 = \text{Just}\ \ 10$	$\text{n}_4 = 2$	$\text{n}_5 = 1$
$\text{pr}_1 = \text{Just}\ \ (\text{Prop}\ \ 15\ \ \text{abc})$	$\text{pr}_2 = \text{Just}\ \ (\text{Prop}\ \ 15\ \ \text{abc})$	$\text{pr}_3 = \text{Just}\ \ (\text{Prop}\ \ 10\ \ \text{abc})$		

All acceptors have accepted the value abc.

4 Analysis

We take the model from the previous chapter, type check it, and use the type check to prove agreement, validity, and termination for our model of the Paxos algorithm. To execute the type check we project the global type to local types and use the typing rules given in [11].

4.1 Local Types

Because no communication occurs in the outer session, the outer session's global type is $\mathrm{G} = 0$. The projection of G to a local type is $\mathrm{G} \upharpoonright_k = 0$ for every k.

We assume a proposer k where $c_A + 1 \leq k \leq c_A + c_P$ and a corresponding quorum $\mathrm{Q}_k \subseteq \{1, \ldots, c_A\}$. Let $p = |\,\mathrm{Q}_k\,| + 1$, $\mathrm{R} = \{1, \ldots, |\,\mathrm{Q}_k\,|\}$, $a \in \mathrm{R}$, and $q \in \mathbb{N} \setminus (\mathrm{R} \cup \{p\})$. We project the global type $\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}}$ to the local types $\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_p$, $\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_a$, and $\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_q$.

$$\begin{aligned}\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_p = (\mu t)\, \big(\textstyle\bigodot_{a \in \mathrm{R}} [a]!_u l1a\, \langle \mathbb{N} \rangle\big)\, .\, \big(\textstyle\bigodot_{a \in \mathrm{R}} [a]?_u l1b\, \langle \mathrm{Promise\ \ Value} \rangle\big)\, .\\ [\mathrm{R}]!_w \left\{ \mathit{Accept}.\, \big(\textstyle\bigodot_{a \in \mathrm{R}} [a]!_u l2a\, \langle \mathrm{Proposal\ \ Value} \rangle\big)\, .0, \mathit{Restart}.t, \mathit{Abort}.0 \right\}\end{aligned}$$

$\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_p$ 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.

$$\begin{aligned}\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_a = (\mu t)\, [p]?_u l1a\, \langle \mathbb{N} \rangle\, .\, [p]!_u l1b\, \langle \mathrm{Promise\ \ Value} \rangle\, .\\ [p]?_w \left\{ \mathit{Accept}.\, [p]?_u l2a\, \langle \mathrm{Proposal\ \ Value} \rangle\, .0, \mathit{Restart}.t, \mathit{Abort}.0 \right\}_{\mathit{Abort}}\end{aligned}$$

$$\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_q = 0$$

$\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_a$ defines the local type for acceptors contained in the proposer's quorum Q_p. Since Paxos defines two types of agents 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.

$\mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_q$ is empty because q is not a role that participates in the global type.

4.2 Type Check

Let $c_A, c_P \in \mathbb{N}$ and $\mathrm{Q}_k = \operatorname{gen}\mathrm{Q}\,(k, c_A, c_P)$ for $c_A + 1 \leq k \leq c_A + c_P$.

$$\Gamma = \Big(o : \mathrm{G} \cdot b_{c_A+1} : \mathrm{G}_{\mathrm{c_A}+1,\mathrm{Q}_{\mathrm{c_A}+1}} \cdot \ldots \cdot b_{c_A+c_P} : \mathrm{G}_{\mathrm{c_A}+\mathrm{c_P},\mathrm{Q}_{\mathrm{c_A}+\mathrm{c_P}}}\Big) \\ \cdot (1 : \mathbb{N} \cdot \ldots \cdot c_A : \mathbb{N} \cdot c_A + 1 : \mathbb{N} \cdot \ldots \cdot c_A + c_P : \mathbb{N})$$

Γ contains the type for our shared channels o, b_k where $c_A + 1 \leq k \leq c_A + c_P$, and all numbers between 1 and $c_A + c_P$.

Note that $0 < c_P < c_A + c_P$ because $c_A > 0 \wedge c_P > 0$. Thus, $c_P : \mathbb{N} \in \Gamma$.

We start the type check with the global environment Γ and the entry point of the model $\mathrm{Sys}\,(c_A, c_P)$. Then, we apply the typing rules in [11] in a proof tree and show that the model can be derived from the axioms (Var) and (End).

4.2.1 System Initialization

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

We apply (Par) twice and split off into the proof trees (S_1), (S_2), and (S_3).

$$(S_1) = \dfrac{\dfrac{(P)}{\Gamma \vdash \overline{b_{c_A+c_P}}\,[p]\,(s)\,.\,\mathrm{P}^{\mathrm{P}}\,(s, k, p, \mathrm{R}, c_A, \mathrm{n}_{c_A+c_P}, []) \rhd z\,[2] : \mathrm{G} \upharpoonright_2}\,(\mathrm{Req})}{\Gamma \vdash \overline{o}\,[2]\,(z)\,.\,\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(c_A + c_P, |\,\mathrm{Q}_{c_A+c_P}\,| + 1, \{1, \ldots, |\,\mathrm{Q}_{c_A+c_P}\,|\}, c_A, \mathrm{n}_{c_A+c_P}, []) \rhd \emptyset}\,(\mathrm{Req})$$

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

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

Applying (Acc) in (S_2) requires that $o : \mathrm{G} \in \Gamma$. (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 $\mathrm{G} \upharpoonright_1 = 0$, $z\,[1] : \mathrm{G} \upharpoonright_1 = \emptyset$. The session environment Δ in (P) is empty for every proposer.

$$(S_3) = \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 0 \triangleright \emptyset}\,(\mathrm{End}) \quad \dfrac{(S_4)}{\Gamma \vdash b_k\,[\mathrm{pos}\,(j, \mathrm{Q}_k)]\,(s)\,.\,\mathrm{P}_1^{\mathrm{A}}\,(\ldots) \triangleright \emptyset}}{\Gamma \vdash \mathtt{if}\ j \notin \mathrm{Q}_k\ \mathtt{then}\ 0\ \mathtt{else}\ b_k\,[\mathrm{pos}\,(j, \mathrm{Q}_k)]\,(s)\,.\,\mathrm{P}_1^{\mathrm{A}}\,(\ldots) \triangleright \emptyset}\,(\mathrm{If}) \quad \ldots}{\Gamma \vdash \Pi_{c_A+1 \leq k \leq c_A+c_P}\ (\mathtt{if}\ j \notin \mathrm{Q}_k\ \mathtt{then}\ 0\ \mathtt{else}\ b_k\,[\mathrm{pos}\,(j, \mathrm{Q}_k)]\,(s) \ldots .) \triangleright \emptyset}\,(\mathrm{Par})^{c_P} \quad \ldots}{\Gamma \vdash \Pi_{1 \leq j \leq c_A}\ \mathrm{P}_{\mathrm{init}}^{\mathrm{A}}\left(j, c_A, c_P, \mathrm{n}_j, \mathrm{pr}_j\right) \triangleright \emptyset}\,(\mathrm{Par})^{c_A}$$

(Par) is applied c_A times to separate the individual acceptors and then c_P times for each acceptor to separate the individual subprocesses. We apply (If) and split the proof tree. Then, we finish the left proof tree by applying (End). The right proof tree is in (S_4). Let $a = \mathrm{pos}\,(j, \mathrm{Q}_k)$. Note that $a \in \{1, \ldots, |\,\mathrm{Q}_k\,|\}$ because $j \in \mathrm{Q}_k$.

$$(S_4) = \dfrac{(A_1)}{\Gamma \vdash b_k\,[a]\,(s)\,.\,\mathrm{P}_1^{\mathrm{A}}\left(s, a, |\,\mathrm{Q}_k\,| + 1, \mathrm{n}_j, \mathrm{pr}_j\right) \triangleright \emptyset}\,(\mathrm{Acc})$$

Applying (Acc) to every subprocess of every acceptor requires that every shared channel b_k where $c_A + 1 \leq k \leq c_A + c_P$ is assigned the type $\mathrm{G}_{\mathrm{k,Q_k}}$ in Γ.

$$\forall k \in \mathbb{N} : (c_A + 1 \leq k \wedge k \leq c_A + c_P) \rightarrow b_k : \mathrm{G}_{\mathrm{k,Q_k}} \in \Gamma$$

Note that only one acceptor and one of its subprocesses is shown in (S_3) and (S_4). The other subprocesses and acceptors only differ in their values for the iteration variables k and j. The rest has been left out to improve readability.

4.2.2 Proposer

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

$$\mathrm{T}_{\mathrm{acc}}^{\mathrm{P}} = \left(\bigodot_{a \in \mathrm{R}} [a]!_u l2a\,\langle \mathrm{Proposal\ \ Value} \rangle\right).0$$
$$\mathrm{T}_{\mathrm{branch}}^{\mathrm{P}} = [\mathrm{R}]!_w \left\{ \mathit{Accept}.\,\mathrm{T}_{\mathrm{acc}}^{\mathrm{P}} \oplus \mathit{Restart}.t \oplus \mathit{Abort}.0 \right\}$$

Note that $\mathrm{G}_{\mathrm{k,Q_k}} \upharpoonright_p = (\mu t)\left(\bigodot_{a \in \mathrm{R}} [a]!_u l1a\,\langle \mathbb{N} \rangle\right).\left(\bigodot_{a \in \mathrm{R}} [a]?_u l1b\,\langle \mathrm{Promise\ \ Value} \rangle\right).\mathrm{T}_{\mathrm{branch}}^{\mathrm{P}}$.

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

$$
\begin{aligned}
e &= \mathrm{anyNack}\left(\overrightarrow{V}\right) \texttt{ or } \left(\mathrm{promCount}\left(\overrightarrow{V}\right) < \left\lceil \frac{c_A + 1}{2} \right\rceil\right) \\
pn &= \mathrm{propNumber}_k\left(\mathrm{n}_k\right) \\
prop &= \mathrm{Prop}\ \left(\mathrm{propNumber}_k\left(\mathrm{n}_k\right)\right)\ \left(\mathrm{promValue}\left(\overrightarrow{V}\right)\right)
\end{aligned}
$$

We observe that $\Gamma \Vdash e : \mathrm{Bool}$, $\Gamma \Vdash pn : \mathbb{N}$, and $\Gamma \Vdash prop : \mathrm{Proposal\ \ Value}$.

To further abbreviate the terms in the proof trees we define two global environments Γ' and Γ''.

$$
\Gamma' = \Gamma \cdot X : t
$$

Γ' contains Γ and a type for the recursion variable X.

$$
\Gamma'' = \Gamma' \cdot v_1 : \mathrm{Promise\ \ Value} \cdot \ldots \cdot v_{|\,\mathrm{Q}_k\,|} : \mathrm{Promise\ \ Value}
$$

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

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

(P) is the continuation of (S_1) and (S_2). In both proof trees the session environment Δ was empty. Here, we apply (Req) and add $s\,[p] : \mathrm{G}_{\mathrm{k},\mathrm{Q}_\mathrm{k}} \upharpoonright_p$ to the session environment. Applying (Rec) changes the global environment from Γ to Γ'. (Func) advances the process. First (USend) and then (UGet) is applied for every acceptor role in R. (UGet) expands the session environment to Γ''. (If) splits the proof tree into (P_t) and (P_f).

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

We apply (WSel) and then (Var) to finish (P_t).

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

After applying (USend) we can apply (USend) once for every acceptor in Q. Finally, we can finish (P_f) — and with it (P) — by applying (End).

4.2.3 Acceptor

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

$$\begin{aligned}
\mathrm{P}^{\mathrm{A}}_{\mathrm{t}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) &= \mathrm{update}\left(\mathrm{n}_k,n'\right).s\,[a,p]!_u l1b\,\langle \text{Promise pr}_k \rangle.\,\mathrm{P}^{\mathrm{A}}_2\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)\\
\mathrm{P}^{\mathrm{A}}_{\mathrm{f}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) &= s\,[a,p]!_u l1b\,\langle \text{Nack} \rangle.\,\mathrm{P}^{\mathrm{A}}_2\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)\\
\mathrm{P}^{\mathrm{A}}_{\mathrm{gt}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) &= \texttt{if } \mathrm{gt}\left(n',\mathrm{n}\right) \texttt{ then } \mathrm{P}^{\mathrm{A}}_{\mathrm{t}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) \texttt{ else } \mathrm{P}^{\mathrm{A}}_{\mathrm{f}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)
\end{aligned}$$

With $\mathrm{P}^{\mathrm{A}}_{\mathrm{gt}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)$, $\mathrm{P}^{\mathrm{A}}_1\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)$ can be rewritten.

$$\begin{aligned}
\mathrm{P}^{\mathrm{A}}_1\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) = (\mu X)\,s\,[p,a]?_u l1a\,\langle \bot \rangle\,\left(n'\right).&\\
&\texttt{if } n' = \bot\\
&\texttt{then } s\,[a,p]!_u l1b\,\langle \bot \rangle.\,\mathrm{P}^{\mathrm{A}}_2\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)\\
&\texttt{else } \mathrm{P}^{\mathrm{A}}_{\mathrm{gt}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)
\end{aligned}$$

$$\begin{aligned}
\mathrm{T}^{\mathrm{A}}_{\mathrm{acc}} &= [p]?_u l2a\,\langle \text{Proposal Value} \rangle.0\\
\mathrm{T}^{\mathrm{A}}_{\mathrm{branch}} &= [p]?_w \left\{Accept.\,\mathrm{T}^{\mathrm{A}}_{\mathrm{acc}}, Restart.t, Abort.0\right\}_{Abort}\\
\mathrm{T}^{\mathrm{A}}_{\mathrm{1b}} &= [p]!_u l1b\,\langle \text{Promise Value} \rangle.\,\mathrm{T}^{\mathrm{A}}_{\mathrm{branch}}
\end{aligned}$$

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

Finally, we define the global environments Γ', Γ'', and Γ'''.

$$\begin{aligned}
\Gamma' &= \Gamma \cdot X : t\\
\Gamma'' &= \Gamma' \cdot n' : \mathbb{N}\\
\Gamma''' &= \Gamma'' \cdot pr' : \text{Proposal Value}
\end{aligned}$$

Γ' contains Γ and assigns the type t to X. Γ'' additionally maps n' to type $\mathbb{N}$. Γ''' adds type Proposal Value for pr'.

$$(A_1) = \dfrac{\dfrac{\dfrac{\dfrac{(A_2)}{\Gamma'' \vdash \mathrm{P}_2^{\mathrm{A}}(\ldots) \rhd s\,[a] : \mathrm{T}_{\mathrm{branch}}^{\mathrm{A}}}}{\Gamma'' \vdash s\,[a,p]!_u l1b\,\langle\bot\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}(\ldots) \rhd s\,[a] : \mathrm{T}_{1\mathrm{b}}^{\mathrm{A}}}\,(\mathrm{USend}) \quad \dfrac{(A_{gt})}{\Gamma'' \vdash \mathrm{P}_{\mathrm{gt}}^{\mathrm{A}}(\ldots) \rhd s\,[a] : \mathrm{T}_{1\mathrm{b}}^{\mathrm{A}}}}{\Gamma'' \vdash \texttt{if}\ n' = \bot\ \texttt{then}\ s\,[a,p]!_u l1b\,\langle\bot\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}(\ldots)\ \texttt{else if}\ \mathrm{gt}\,(n', \mathrm{n}_j) \ldots \rhd s\,[a] : \mathrm{T}_{1\mathrm{b}}^{\mathrm{A}}}\,(\mathrm{If})}{\Gamma' \vdash s\,[p,a]?_u l1a\,\langle\bot\rangle\,(n')\,.\texttt{if}\ n' = \bot \ldots \rhd s\,[a] : [p]?_u l1a\,\langle\mathbb{N}\rangle\,.\,\mathrm{T}_{1\mathrm{b}}^{\mathrm{A}}}\,(\mathrm{UGet})}{\Gamma \vdash (\mu X)\,s\,[p,a]?_u l1a\,\langle\bot\rangle\,(n') \ldots . \rhd s\,[a] : (\mu t)\,[p]?_u l1a\,\langle\mathbb{N}\rangle \ldots .}\,(\mathrm{Rec})$$

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

We examine (A_{gt}) first, because it contains the proof tree of the left branch (A_2).

$$(A_{gt}) = \dfrac{(A_t) \qquad (A_f)}{\Gamma'' \vdash \texttt{if}\ \mathrm{gt}\,(n', \mathrm{n}_j)\ \texttt{then}\ \mathrm{P}_{\mathrm{t}}^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)\ \texttt{else}\ \mathrm{P}_{\mathrm{f}}^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) \rhd s\,[a] : \mathrm{T}_{1\mathrm{b}}^{\mathrm{A}}}\,(\mathrm{If})$$

First, we split the proof tree with (If). We continue the resulting branches in separate proof trees (A_t) and (A_f).

$$(A_t) = \dfrac{(A_2)}{\Gamma'' \vdash s\,[a,p]!_u l1b\,\langle\mathrm{Nack}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}(\ldots) \rhd s\,[a] : [p]!_u l1b\,\langle\mathrm{Promise\ \ Value}\rangle\,.\,\mathrm{T}_{\mathrm{branch}}^{\mathrm{A}}}\,(\mathrm{USend})$$

$$(A_f) = \dfrac{(A_2)}{\Gamma'' \vdash s\,[a,p]!_u l1b\,\langle\mathrm{Nack}\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}(\ldots) \rhd s\,[a] : [p]!_u l1b\,\langle\mathrm{Promise\ \ Value}\rangle\,.\,\mathrm{T}_{\mathrm{branch}}^{\mathrm{A}}}\,(\mathrm{USend})$$

In both, (A_t) and (A_f), we apply (USend). We continue with (A_2), which is the proof tree for $\mathrm{P}_2^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right)$.

$$(A_2) = \dfrac{(A_{Accept}) \qquad \dfrac{}{\Gamma'' \vdash X \rhd s\,[a] : t}\,(\mathrm{Var}) \qquad \dfrac{}{\Gamma'' \vdash 0 \rhd s\,[a] : 0}\,(\mathrm{End})}{\Gamma'' \vdash s\,[a,p]?_w\left\{Accept.\,\mathrm{P}_3^{\mathrm{A}}\left(s,a,p,\mathrm{n}_j,\mathrm{pr}_j\right) \oplus Restart.X \oplus Abort.0\right\} \rhd s\,[a] : \mathrm{T}_{\mathrm{branch}}^{\mathrm{A}}}\,(\mathrm{WBran})$$

By applying (WBran) we separate the three branches. From left to right we get an *Accept*, a *Restart*, and an *Abort* branch. The proof tree for the *Accept* branch is in (A_{Accept}). The *Restart* branch can be finished by applying (Var) and the *Abort* branch by applying (End).

$$(A_{Accept}) = \dfrac{\dfrac{\dfrac{}{\Gamma''' \vdash 0 \rhd s\,[a] : 0}\,(\mathrm{End}) \quad \dfrac{\dfrac{(A_{update})}{\Gamma''' \vdash \mathrm{update}\left(\mathrm{pr}_j, pr'\right) \ldots . \rhd s\,[a] : 0} \quad \dfrac{}{\Gamma''' \vdash 0 \rhd s\,[a] : 0}\,(\mathrm{End})}{\Gamma''' \vdash \texttt{if}\ \mathrm{ge}\left(\mathrm{nFromProp}\left(pr'\right), \mathrm{n}_j\right)\ \texttt{then}\ \ldots\ \texttt{else}\ 0 \rhd s\,[a] : 0}\,(\mathrm{If})}{\Gamma''' \vdash \texttt{if}\ pr' = \bot\ \texttt{then}\ 0\ \texttt{else}\ \ldots \rhd s\,[a] : 0}\,(\mathrm{If})}{\Gamma'' \vdash s\,[p, a]?_u l2a\,\langle \bot \rangle\,(pr') \ldots . \rhd s\,[a] : [p]?_u l2a\,\langle \mathrm{Proposal\ \ Value} \rangle\,.0}\,(\mathrm{UGet})$$

We apply (UGet) and expand the global session to Γ'''. The proof tree is split twice by applying (If) twice. The proof trees on the left and on the right are finished by applying (End). To keep this proof tree readable we continue the proof of the remaining branch in (A_{update}).

$$(A_{update}) = \dfrac{\dfrac{\dfrac{}{\Gamma''' \vdash 0 \rhd s\,[a] : 0}\,(\mathrm{End})}{\Gamma''' \vdash \mathrm{update}\left(\mathrm{n}_j, \mathrm{Just\ \ nFromProp}\left(pr'\right)\right).0 \rhd s\,[a] : 0}\,(\mathrm{Func})}{\Gamma''' \vdash \mathrm{update}\left(\mathrm{pr}_j, pr'\right) \ldots . \rhd s\,[a] : 0}\,(\mathrm{Func})$$

Finally, we apply (Func) twice and (End) once. This concludes the type check and proves that the model is well-typed.

4.3 Validity, Agreement, and Termination

Finally, we can prove **Validity**: only a value that has been proposed may be chosen [8], **Agreement**: only a single value is chosen [8], and **Termination**: if a value has been proposed and sufficient processes remain non-faulty, then eventually some value is chosen (compare to [10]).

4.3.1 Validity

To prove that only a value that has been proposed may be chosen we examine the communication structure and the origins of the accepted values in our model. Because the model is well-typed we know the communication structure is as specified in global type $\mathrm{G}_{\mathrm{k,Q_k}}$. Session fidelity ensures there are no communication mismatches [11]. From [11] 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 $\mathrm{Promise\ \ Value}$. 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. pr_j contains a proposal that was proposed by a proposer either in the most recent or a previous run of the algorithm and the data it holds 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 $\mathrm{Proposal\ \ Value}$ 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 $\mathrm{promValue}$. An entry in $\overrightarrow{V}$ contains a proposal $prop$ if it is of the form $\mathrm{Promise\ \ (Just\ \ } prop\mathrm{)}$. 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, $\mathrm{promValue}$ 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 $\mathrm{promValue}$ 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.

4.3.2 Agreement

We examine the flow of accepted values through the network to prove that only a single value is chosen. $\mathtt{FP}_{\mathtt{crash}}$ guarantees that the set of correct acceptors in the quorum of any correct proposer k is itself a quorum, i.e., an accepting set that can choose a value [7]. Should message loss occur in labels $l1a$ or $l1b$, k broadcasts its decision to restart the algorithm. This broadcast is weakly reliable and thus only fails if k crashes because $\mathtt{FP}_{\mathtt{wskip}}$ disallows suspicion of correct proposers by acceptors. Given the definition of $\mathrm{promValue}$, k will only propose a fresh value if it has not received any accepted proposals from an acceptor.

In Paxos a quorum is a majority of acceptors. For any two majorities of acceptors there exists at least one that is contained in both majorities. So, at least one acceptor in every other proposer's quorum is contained in k's quorum. Thus, if a majority of acceptors accept k'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 Termination

We want to prove that if a value has been proposed and sufficient processes remain non-faulty, then eventually some value is chosen.

Well-typedness ensures that the processes follow the communication structure in $\mathrm{G}_{\mathrm{k,Q_k}}$ and session fidelity ensures the absence of deadlocks [11]. Thus, processes either loop forever or terminate. We only need to prove that the implementation eventually exits the loop given our system requirements. Assuming a failure detector in $\Diamond\mathscr{S}$, eventually any correct proposer is not suspected by any correct acceptor and does not suspect any correct acceptor. $\mathtt{FP}_{\mathtt{crash}}$ guarantees that all correct proposers can communicate with a set of correct acceptors that forms a quorum.

If all proposers crashed then no value can be proposed and no value will be chosen. Termination as defined above is satisfied in this case.

If at least one proposer is correct, there exists a distinguished proposer because our model assumes a mechanism for electing such a proposer. The distinguished proposer executes the algorithm, communicating with its quorum. Eventually, it will choose a high enough proposal number and receive enough promises to send a proposal. The decision to send a proposal as well as the proposal itself are received by enough correct acceptors to form a quorum. These messages can not be lost due to the weakly reliable branching and our implementation of $\mathtt{FP}_{\mathtt{ml}}$ for label $l2a$. The proposal is accepted by a majority of acceptors and thus, a value is chosen. The distinguished proposer and the corresponding subprocesses of the acceptors in its quorum terminate. The process repeats with the next distinguished proposer, if any. Every subprocess of an acceptor corresponding to a crashed proposer can skip the weakly reliable broadcast and terminate.

4.3.4 Result

The preceding proofs show that our model satisfies validity, agreement, and termination. Regarding validity and termination, the type check assures the absence of communication mismatches. To prove validity, we only need to examine the data flow of the accepted values to finish the proof. For termination, we show that the implementation eventually exits the loop, given our system requirements, i.e., the failure detectors and implementations of the failure patterns. Agreement follows from the data flow of accepted values, overlapping quorums, and the way a proposer chooses a proposal value.

5 Conclusion

In this work we have provided a practical example of a static analysis of Paxos using FTMPST.

5.1 Summary

First, we introduced Paxos and FTMPST. For the specification of Paxos we chose [8] because in it Lamport describes the most basic form of the Paxos algorithm. Paxos is designed to solve the consensus problem in a network where processes or their communications may experience failures. To model the fault-tolerance of Paxos we used FTMPST, which are Multiparty Session Types extended with fault-tolerance.

We extended FTMPST with function calls as prefixes to model side effects, e.g. updating a register and introduced notation to abbreviate sequences of prefixes and weakly reliable branches. Additionally, we specified a notation for defining sorts with type parameters.

The sorts Bool, $\mathrm{Maybe\ a}$, $\mathrm{Proposal\ a}$, and $\mathrm{Promise\ a}$ are introduced. These represent data types used in the global type and processes.

Next, we specified the global type for a session with one proposer and a quorum. This mirrors the phases $1a$, $1b$ and $2a$ described in [8]. Phase $2b$ is not modelled in the global type because it contains no inter-process communication. The global type can be projected to local types, which are needed for the type check.

To construct the processes some functions are introduced. Some model various aspects of the algorithm, e.g. picking proposal numbers from disjoint sets of numbers, picking a proposal value from a list of promises, evaluating lists of promises, and generating sets of acceptors that constitute a quorum. Others allowed us to simplify working with the specified structure of the sorts in the processes, i.e., accessing internal values and comparing values of different sorts. One more function was introduced to correctly assign roles to processes within a session.

With these functions we defined processes for system initialization, proposers, and acceptors.

Each process is assigned a process index. An acceptor subprocess only accepts a proposer's session request if its process index is in that proposer's quorum. However, the process index is not the role the acceptor subprocess has in that session. So, we mapped the process index of each acceptor subprocess to its position within the corresponding proposer's quorum to obtain its role. This way proposers only communicate with acceptors inside their quorum, and we avoid communication mismatches.

Including quorums in our model raised the complexity of the system initialization specifically. Otherwise, if all proposers communicated with all acceptors the process index would equal that process' role in a session.

The processes for proposers and acceptors followed directly from the description of the Paxos algorithm and the global type.

The system requirements were modelled in FTMPST using failure patterns. We used the failure patterns to prove termination for our model.

In order to demonstrate how the model works we examined an example run with three acceptors and two proposers. We described a scenario where one proposer completes the algorithm and another tries to run the algorithm and fails but retries and succeeds. Then, we applied the appropriate reduction rules to the formulae.

Finally, After defining and studying an example run of our model, we analyze it with the goal of proving validity, agreement, and termination. To prove these properties we ensured our model is well-typed by type checking it. To execute the type check we projected the global type to local types first. Using well-typedness of our model we proved validity, agreement, and termination. This satisfies our goal of analyzing Paxos with FTMPST.

5.2 Future Work

There are two notable differences between the global type and Paxos. First, in phase $1b$ of Paxos only acceptors that received a prepare request send a message to the proposer, whereas in the global type all acceptors in the session send a message to the proposer. Second, the global type splits phase $2a$ into two messages. A weakly reliable branching to broadcast the proposers decision about whether to restart the algorithm or send an accept request and the accept request itself. In Paxos the proposer does not inform the acceptors of its decision to restart or send an accept request. Future research is needed to assess how close the model is to the Paxos algorithm and how it could be improved.

Another question for future work would be how our model compares to a different model of the same algorithm. We chose to define the global type for a single proposer and its quorum. An alternative solution might define the global type for all proposers and acceptors and have them participate in the same session.

Bibliography

[1] M.K. Aguilera, W. Chen, and S. Toueg. “Heartbeat: A timeout-free failure detector for quiescent reliable communication”. In: *Distributed Algorithms*. Ed. by M. Mavronicolas and P. Tsigas. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 126–140. ISBN: 978-3-540-69600-1.

[2] L. Bettini et al. “Global Progress in Dynamically Interleaved Multiparty Sessions”. In: *CONCUR 2008 - Concurrency Theory*. Ed. by F. van Breugel and M. Chechik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 418–433. ISBN: 978-3-540-85361-9.

[3] T.D. Chandra and S. 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.

[4] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (3rd Edition)*. Addison-Wesley, 2001, p. 452.

[5] K. Honda, N. Yoshida, and M. Carbone. “Multiparty Asynchronous Session Types”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 273–284. ISBN: 9781595936899. DOI: 10.1145/1328438.1328472. URL: https://doi.org/10.1145/1328438.1328472.

[6] K. Honda, N. Yoshida, and M. Carbone. “Multiparty Asynchronous Session Types”. In: *J. ACM* 63.1 (Mar. 2016). ISSN: 0004-5411. DOI: 10.1145/2827695. URL: https://doi.org/10.1145/2827695.

[7] L. 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.

[8] L. 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/.

[9] L. Lamport. “The Part-Time Parliament”. In: *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].* (May 1998). ACM SIGOPS Hall of Fame Award in 2012. URL: https://www.microsoft.com/en-us/research/publication/part-time-parliament/.

[10] Leslie Lamport. *Generalized Consensus and Paxos*. Tech. rep. MSR-TR-2005-33. Mar. 2005, p. 60. URL: https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/.

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

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

[13] N. Yoshida et al. “Parameterised Multiparty Session Types”. In: *Foundations of Software Science and Computational Structures*. Ed. by L. Ong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 128–145. ISBN: 978-3-642-12032-9.