

Analyzing Paxos With Fault-Tolerant Multiparty Session Types

Bachelor thesis by Nicolas Daniel Torres
Date of submission: January 31, 2022

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



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department

<institute>

Theory of Parallel Systems

Contents

1	Introduction	3
2	Technical Preliminaries	4
2.1	Paxos	4
2.1.1	The Problem	4
2.1.2	The Algorithm	5
2.1.3	The Implementation	5
2.2	Fault-Tolerant Multiparty Session Types	6
2.3	Additional Notation	14
2.3.1	Prefixes And Branches	14
2.3.2	Function Calls As Prefixes	14
2.3.3	Type Parameters	15
3	Model	16
3.1	Sorts	16
3.2	Global Type	17
3.3	Functions	17
3.4	Processes	19
3.4.1	System Initialization	19
3.4.2	Proposer	21
3.4.3	Acceptor	21
3.5	Failure Patterns	22
3.6	Example	23
3.6.1	Scenario	23
3.6.2	Formulae	24
4	Analysis	29
4.1	Local Types	29
4.2	Type Check	30
4.2.1	System Initialization	30
4.2.2	Proposer	31
4.2.3	Acceptor	33
4.3	Validity, Agreement, Termination	35
4.3.1	Validity	35
4.3.2	Agreement	36
4.3.3	Termination	36
5	Conclusion	37

1 Introduction

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

To achieve consensus, consensus algorithms must satisfy the following properties: termination, validity, and agreement [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].

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

In this work we will use Fault-Tolerant Multiparty Session Types to analyze the consensus algorithm Paxos, as described in [8].

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.

We assume that the sets \mathcal{N} of names $a, s, x \dots$; \mathcal{R} of roles n, r, \dots ; \mathcal{L} of labels l, l_d, \dots ; \mathcal{V}_T of type variables t ; and \mathcal{V}_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 $\mathbb{B}, \mathbb{N}, \dots$. The set \mathcal{E} of expressions e, v, b, \dots 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 $\bar{a}[n](s).P$ and $a[r](s).P$ 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 $r_1 \rightarrow_r r_2 : \langle S \rangle . G$ 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 $s[r_1, r_2]!_r \langle e \rangle . P_1$ (specified as $[r_2]!_r \langle S \rangle . T_1$) and the receiver $s[r_2, r_1]?_r(x) . P_2$ (specified as $[r_1]?_r \langle S \rangle . T_2$). As result of the communication, the receiver instantiates x in its continuation P_2 with the received value.

The type $r_1 \rightarrow_u r_2 : l \langle S \rangle . G$ 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 $s[r_1, r_2]!_u l \langle e \rangle . P_1$ (specified as $[r_2]!_u l \langle S \rangle . T_1$) and $s[r_2, r_1]?_u l \langle v \rangle (x) . P_2$ (specified as $[r_1]?_u l \langle S \rangle . T_2$). The receiver $s[r_2, r_1]?_u l \langle v \rangle (x) . P_2$ declares a default value v that is used instead of a received value to instantiate x after a failure.

The strongly reliable branching $r_1 \rightarrow_r r_2 : \{l_i . G_i\}_{i \in I}$ 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 $s[r_1, r_2]!_r l . P$ (specified

Global Types	Local Types	Processes
$G ::= r_1 \rightarrow_r r_2 : \langle S \rangle . G$ $ r_1 \rightarrow_u r_2 : l \langle S \rangle . G$ $ r_1 \rightarrow_r r_2 : \{l_i . G_i\}_{i \in I}$ $ r \rightarrow_w R : \{l_i . G_i\}_{i \in I, l_d}$ $ G_1 \parallel G_2$ $ (\mu t)G \mid t \mid \text{end}$ $ r_1 \rightarrow r_2 : \langle s'[r] : T \rangle . G$	$T ::= [r_2]!_r \langle S \rangle . T$ $ [r_1]?_r \langle S \rangle . T$ $ [r_2]!_u l \langle S \rangle . T$ $ [r_1]?_u l \langle S \rangle . T$ $ [r_2]!_r \{l_i . T_i\}_{i \in I}$ $ [r_1]?_r \{l_i . T_i\}_{i \in I}$ $ [R]!_w \{l_i . T_i\}_{i \in I}$ $ [r]?_w \{l_i . T_i\}_{i \in I, l_d}$ $ (\mu t)T \mid t \mid \text{end}$ $ [r_2]! \langle s'[r] : T \rangle . T'$ $ [r_1]? \langle s'[r] : T \rangle . T'$	$P ::= \bar{a}[n](s).P$ $ a[r](s).P$ $ s[r_1, r_2]!_r \langle e \rangle . P$ $ s[r_2, r_1]?_r (x).P$ $ s[r_1, r_2]!_u l \langle e \rangle . P$ $ s[r_2, r_1]?_u l \langle v \rangle (x).P$ $ s[r_1, r_2]!_r l . P$ $ s[r_2, r_1]?_r \{l_i . P_i\}_{i \in I}$ $ s[r, R]!_w l . P$ $ s[r_j, r]?_w \{l_i . P_i\}_{i \in I, l_d}$ $ P_1 \mid P_2$ $ (\mu X)P \mid X \mid \mathbf{0}$ $ \text{if } b \text{ then } P_1 \text{ else } P_2$ $ (\nu x)P \mid \perp$ $ s[r_1, r_2]! \langle s'[r] \rangle . P$ $ s[r_2, r_1]? \langle s'[r] \rangle . P$ $ s_{r_1 \rightarrow r_2} : M$
Message Types		Messages
$MT ::= \langle S \rangle^r \mid l \langle S \rangle^u \mid l^r \mid l^w \mid s[r]$		$M ::= \langle v \rangle^r \mid l \langle v \rangle^u \mid l^r$ $ l^w \mid s[r]$

Figure 2.1: Syntax of Fault-Tolerant MPST

as $[r_2]!_r \{l_i . T_i\}_{i \in I}$. Upon receiving branch l_j from r_1 the process $s[r_2, r_1]?_r \{l_i . P_i\}_{i \in I}$ (specified as $[r_1]?_r \{l_i . T_i\}_{i \in I}$) continues with P_j .

The weakly reliable counterpart of branching is $r \rightarrow_w R : \{l_i . G_i\}_{i \in I, l_d}$, where $R \subseteq \mathcal{R}$ and l_d with $d \in I$ 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 $s[r, R]!_w l . P$ (specified as $[R]!_w \{l_i . T_i\}_{i \in I}$) allows to broadcast its decision to R and $s[r_j, r]?_w \{l_i . P_i\}_{i \in I, l_d}$ (specified as $[r]?_w \{l_i . T_i\}_{i \in I, l_d}$) defines a default label l_d .

The \perp 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 $s_{r_1 \rightarrow r_2} : []$ for each pair of roles $r_1 \neq r_2$ 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 $G \parallel G'$; parallel composition $P \mid P'$; recursion $(\mu t)G$, $(\mu X)P$; inaction end , $\mathbf{0}$; conditionals $\text{if } b \text{ then } P_1 \text{ else } P_2$; session delegation $r_1 \rightarrow r_2 : \langle s'[r] : T \rangle . G$, $s[r_1, r_2]! \langle s'[r] \rangle . P$, $s[r_2, r_1]? \langle s'[r] \rangle . P$; 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:

$$(r_1 \rightarrow_{\diamond} r_2 : \mathfrak{S}.G) \upharpoonright_p \triangleq \begin{cases} [r_2]!_{\diamond} \mathfrak{S}.G \upharpoonright_p & \text{if } p = r_1 \\ [r_1]?_{\diamond} \mathfrak{S}.G \upharpoonright_p & \text{if } p = r_2 \\ G \upharpoonright_p & \text{otherwise} \end{cases}$$

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

$$(r_1 \rightarrow_{\diamond} \mathfrak{R} : \{l_i.G_i\}_{i \in I \mathcal{D}}) \upharpoonright_p \triangleq \begin{cases} [\mathfrak{R}]!_{\diamond} \{l_i.G_i \upharpoonright_p\}_{i \in I} & \text{if } p = r_1 \\ [r_1]?_{\diamond} \{l_i.G_i \upharpoonright_p\}_{i \in I \mathcal{D}} & \text{if } \mathfrak{B} \\ \bigsqcup_{i \in I} (G_i \upharpoonright_p) & \text{otherwise} \end{cases}$$

where either $\diamond = r$, $\mathfrak{R} = r_2$, \mathfrak{B} is $p = r_2$, \mathcal{D} is empty or $\diamond = w$, $\mathfrak{R} = R$, \mathfrak{B} is $p \in R$, \mathcal{D} is l_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 $\bigsqcup_{i \in I} (G_i \upharpoonright_p) = (G_1 \upharpoonright_p) \sqcup \dots \sqcup (G_n \upharpoonright_p)$. The operation \sqcup is (similar to [13]) inductively defined as:

$$\begin{aligned} T \sqcup T &= T \\ ([r]?_r I_1) \sqcup ([r]?_r I_2) &= [r]?_r (I_1 \sqcup I_2) \\ ([r]?_w I_1) \sqcup ([r]?_w I_2) &= [r]?_w (I_1 \sqcup I_2) \quad \text{if } I_1 \text{ and } I_2 \text{ have the same default branch} \\ I \sqcup \emptyset &= I \\ I \sqcup (\{l.T\} \cup J) &= \begin{cases} \{l.(T' \sqcup T)\} \cup ((I \setminus \{l.T'\}) \sqcup J) & \text{if } l.T' \in I \\ \{l.T\} \cup (I \sqcup J) & \text{if } l \notin I \end{cases} \end{aligned}$$

where $T, T' \in \mathcal{T}$, $l \notin I$ is short hand for $\nexists T'$. $l.T' \in I$, 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 $r_1 \rightarrow_r r_2 : \{l_i.G_i\}_{i \in I}$ 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:

$$\begin{aligned} (G_1 \parallel G_2) \upharpoonright_p &\triangleq \begin{cases} G_1 \upharpoonright_p & \text{if } p \notin R(G_2) \\ G_2 \upharpoonright_p & \text{if } p \notin R(G_1) \end{cases} \quad ((\mu t)G) \upharpoonright_p \triangleq \begin{cases} (\mu t)G \upharpoonright_p & \text{if } p \in R(G) \\ \text{end} & \text{otherwise} \end{cases} \\ t \upharpoonright_p &\triangleq t \quad \text{end} \upharpoonright_p \triangleq \text{end} \end{aligned}$$

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 $r \in R(G)$ the projection $G \upharpoonright_r$ 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 $\bar{a}[n](s).P$ and x is bound in P by $s[r_1, r_2]?_r(x).P$. A variable or name is *free* if it is not bound. Let $\text{FN}(P)$ return the free names of P .

We use $'.'$ (as e.g. in $\bar{a}[r](s).P$) 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 $P_1 \mid \dots \mid P_n$.

We write $\text{nsr}(G)$, $\text{nsr}(T)$, and $\text{nsr}(P)$, 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 $s[r]$ if it has an action prefix on s , where r is the first role mentioned in the prefix. Let $A(P)$ be the set of actors of P .

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:

$$\begin{aligned} P \mid \mathbf{0} &\equiv P & P_1 \mid P_2 &\equiv P_2 \mid P_1 & P_1 \mid (P_2 \mid P_3) &\equiv (P_1 \mid P_2) \mid P_3 \\ (\mu X)\mathbf{0} &\equiv \mathbf{0} & (\nu x)\mathbf{0} &\equiv \mathbf{0} & (\nu x)(\nu y)P &\equiv (\nu y)(\nu x)P \\ (\nu x)(P_1 \mid P_2) &\equiv P_1 \mid (\nu x)P_2 & \text{if } x &\notin \text{FN}(P_1) \end{aligned}$$

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 $\text{eval}(y)$ is wrapped in a message and added to the end of the corresponding message queue and the continuation of 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 FP_{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 FP_{uskip} and instead substitutes the bound variable x in the continuation with the default value dv . The failure pattern FP_{uskip} tells us whether a reception can be skipped.

$$\begin{array}{ll}
\text{(Init)} & \bar{a}[n](s).P_n \mid \prod_{1 \leq i \leq n-1} a[i](s).P_i \mapsto (\nu s) \left(\prod_{1 \leq i \leq n} P_i \mid \prod_{1 \leq i, j \leq n, i \neq j} s_{i \rightarrow j} : [] \right) \\
& \text{if } a \neq s \\
\text{(RSend)} & s[r_1, r_2]!_r \langle y \rangle . P \mid s_{r_1 \rightarrow r_2} : M \mapsto P \mid s_{r_1 \rightarrow r_2} : M \# \langle v \rangle^r \quad \text{if } \text{eval}(y) = v \\
\text{(RGet)} & s[r_1, r_2]?_r \langle x \rangle . P \mid s_{r_2 \rightarrow r_1} : \langle v \rangle^r \# M \mapsto P\{v/x\} \mid s_{r_2 \rightarrow r_1} : M \\
\text{(USend)} & s[r_1, r_2]!_u \langle y \rangle . P \mid s_{r_1 \rightarrow r_2} : M \mapsto P \mid s_{r_1 \rightarrow r_2} : M \# \langle v \rangle^u \quad \text{if } \text{eval}(y) = v \\
\text{(UGet)} & s[r_1, r_2]?_u \langle dv \rangle \langle x \rangle . P \mid s_{r_2 \rightarrow r_1} : l' \langle v \rangle^u \# M \mapsto P\{v/x\} \mid s_{r_2 \rightarrow r_1} : M \\
& \text{if } l \doteq l', \text{FP}_{\text{uget}}(s, r_1, r_2, l') \\
\text{(USkip)} & s[r_1, r_2]?_u \langle dv \rangle \langle x \rangle . P \mapsto P\{dv/x\} \quad \text{if } \text{FP}_{\text{uskip}}(s, r_1, r_2, l) \\
\text{(ML)} & s_{r_1 \rightarrow r_2} : l \langle v \rangle^u \# M \mapsto s_{r_1 \rightarrow r_2} : M \quad \text{if } \text{FP}_{\text{ml}}(s, r_1, r_2, l) \\
\text{(RSel)} & s[r_1, r_2]!_r l . P \mid s_{r_1 \rightarrow r_2} : M \mapsto P \mid s_{r_1 \rightarrow r_2} : M \# l^r \\
\text{(RBran)} & s[r_1, r_2]?_r \{l_i . P_i\}_{i \in I} \mid s_{r_2 \rightarrow r_1} : l^r \# M \mapsto P_j \mid s_{r_2 \rightarrow r_1} : M \quad \text{if } l \doteq l_j, j \in I \\
\text{(WSel)} & s[r, R]!_w l . P \mid \prod_{r_i \in R} s_{r \rightarrow r_i} : M_i \mapsto P \mid \prod_{r_i \in R} s_{r \rightarrow r_i} : M_i \# l^w \\
\text{(WBran)} & s[r_1, r_2]?_w \{l_i . P_i\}_{i \in I, l_d} \mid s_{r_2 \rightarrow r_1} : l^w \# M \mapsto P_j \mid s_{r_2 \rightarrow r_1} : M \quad \text{if } l \doteq l_j, j \in I \\
\text{(WSkip)} & s[r_1, r_2]?_w \{l_i . P_i\}_{i \in I, l_d} \mapsto P_d \quad \text{if } \text{FP}_{\text{wskip}}(s, r_1, r_2) \\
\text{(Crash)} & P \mapsto \perp \quad \text{if } \text{FP}_{\text{crash}}(P) \\
\text{(If-T)} & \text{if } e \text{ then } P \text{ else } P' \mapsto P \quad \text{if } e \text{ is true} \\
\text{(If-F)} & \text{if } e \text{ then } P \text{ else } P' \mapsto P' \quad \text{if } e \text{ is false} \\
\text{(Deleg)} & s[r_1, r_2]! \langle \langle s'[r] \rangle \rangle . P \mid s_{r_1 \rightarrow r_2} : M \mapsto P \mid s_{r_1 \rightarrow r_2} : M \# s'[r] \\
\text{(SRecv)} & s[r_1, r_2]? \langle \langle s'[r] \rangle \rangle . P \mid s_{r_2 \rightarrow r_1} : s''[r'] \# M \mapsto P\{s''/s'\}\{r'/r\} \mid M \\
\text{(Par)} & P_1 \mid P_2 \mapsto P'_1 \mid P_2 \quad \text{if } P_1 \mapsto P'_1 \\
\text{(Res)} & (\nu x)P \mapsto (\nu x)P' \quad \text{if } P \mapsto P' \\
\text{(Rec)} & (\mu X)P \mapsto P\{(\mu X)P/X\} \\
\text{(Struc)} & P_1 \mapsto P'_1 \quad \text{if } P_1 \equiv P_2, P_2 \mapsto P'_2, P'_2 \equiv P'_1
\end{array}$$

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

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 FP_{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 FP_{crash} . (ML) allows to drop an unreliable message if the failure pattern FP_{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 FP_{uget} , FP_{uskip} , and FP_{wskip} use only local information, whereas FP_{ml} and FP_{crash} may use global information of the system in the current run.

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

$$\begin{aligned} \Gamma &::= \emptyset \mid \Gamma \cdot x:S \mid \Gamma \cdot a:G \mid \Gamma \cdot l:S \\ \Delta &::= \emptyset \mid \Delta \cdot s[r]:T \mid \Delta \cdot s_{r_1 \rightarrow r_2}:\text{MT}^* \end{aligned}$$

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 $x \# \Gamma$ and $x \# \Delta$ 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 write $\text{nsr}(\Delta)$ if $\text{nsr}(T)$ for all local types T in Δ and if Δ does not contain message queues. With $\Gamma \Vdash y:S$ 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 $x:S_x \in \Gamma$.

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 $\Gamma \Vdash y:S$ 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 $x:S_x \in \Gamma$. 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 $x:S$ 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

$$\begin{array}{c}
\text{(Req)} \frac{a:G \in \Gamma \quad |\mathbf{R}(G)| = n \quad \Gamma \vdash P \triangleright \Delta \cdot s[n]:G|_n}{\Gamma \vdash \bar{a}[n](s).P \triangleright \Delta} \quad \text{(Acc)} \frac{a:G \in \Gamma \quad 0 < r < |\mathbf{R}(G)| \quad \Gamma \vdash P \triangleright \Delta \cdot s[r]:G|_r}{\Gamma \vdash a[r](s).P \triangleright \Delta} \\
\\
\text{(RSend)} \frac{\Gamma \Vdash y:S \quad \Gamma \vdash P \triangleright \Delta \cdot s[r_1]:T}{\Gamma \vdash s[r_1, r_2]!_r \langle y \rangle . P \triangleright \Delta \cdot s[r_1]:[r_2]!_r \langle S \rangle . T} \\
\text{(RGet)} \frac{x^\sharp(\Gamma, \Delta, s) \quad \Gamma \cdot x:S \vdash P \triangleright \Delta \cdot s[r_1]:T}{\Gamma \vdash s[r_1, r_2]?_r \langle x \rangle . P \triangleright \Delta \cdot s[r_1]:[r_2]?_r \langle S \rangle . T} \\
\text{(USend)} \frac{\Gamma \Vdash y:S \quad l \doteq l' \quad l':S \in \Gamma \quad \Gamma \vdash P \triangleright \Delta \cdot s[r_1]:T}{\Gamma \vdash s[r_1, r_2]!_u \langle y \rangle . P \triangleright \Delta \cdot s[r_1]:[r_2]!_u \langle l' \rangle \langle S \rangle . T} \\
\text{(UGet)} \frac{x^\sharp(\Gamma, \Delta, s) \quad \Gamma \Vdash v:S \quad l \doteq l' \quad l':S \in \Gamma \quad \Gamma \cdot x:S \vdash P \triangleright \Delta \cdot s[r_1]:T}{\Gamma \vdash s[r_1, r_2]?_u \langle v \rangle \langle x \rangle . P \triangleright \Delta \cdot s[r_1]:[r_2]?_u \langle l' \rangle \langle S \rangle . T} \\
\\
\text{(RSel)} \frac{j \in I \quad l \doteq l_j \quad \Gamma \vdash P \triangleright \Delta \cdot s[r_1]:T_j}{\Gamma \vdash s[r_1, r_2]!_r l . P \triangleright \Delta \cdot s[r_1]:[r_2]!_r \{l_i.T_i\}_{i \in I}} \quad \text{(Var)} \frac{}{\Gamma \cdot X:t \vdash X \triangleright s[r]:t} \\
\text{(RBran)} \frac{\forall j \in I_2. \exists i \in I_1. l_i \doteq l_j \wedge \Gamma \vdash P_i \triangleright \Delta \cdot s[r_1]:T_j}{\Gamma \vdash s[r_1, r_2]?_r \{l_i.P_i\}_{i \in I_1} \triangleright \Delta \cdot s[r_1]:[r_2]?_r \{l_i.T_i\}_{i \in I_2}} \\
\text{(WSel)} \frac{j \in I \quad l \doteq l_j \quad \Gamma \vdash P \triangleright \Delta \cdot s[r]:T_j}{\Gamma \vdash s[r, R]!_w l . P \triangleright \Delta \cdot s[r]:[R]!_w \{l_i.T_i\}_{i \in I}} \quad \text{(Crash)} \frac{\text{nsr}(\Delta)}{\Gamma \vdash \perp \triangleright \Delta} \\
\text{(WBran)} \frac{l_d \doteq l'_d \quad \forall j \in I_2. \exists i \in I_1. l_i \doteq l_j \wedge \Gamma \vdash P_i \triangleright \Delta \cdot s[r_1]:T_j}{\Gamma \vdash s[r_1, r_2]?_w \{l_i.P_i\}_{i \in I_1, l_d} \triangleright \Delta \cdot s[r_1]:[r_2]?_w \{l_i.T_i\}_{i \in I_2, l'_d}} \\
\\
\text{(Deleg)} \frac{\Gamma \vdash P \triangleright \Delta \cdot s[r_1]:T}{\Gamma \vdash s[r_1, r_2]! \langle s'[r] \rangle . P \triangleright \Delta \cdot s[r_1]:[r_2]! \langle s'[r]:T' \rangle . T \cdot s'[r]:T'} \\
\text{(SRecv)} \frac{\Gamma \vdash P \triangleright \Delta \cdot s[r_1]:T \cdot s'[r]:T'}{\Gamma \vdash s[r_1, r_2]? \langle s'[r] \rangle . P \triangleright \Delta \cdot s[r_1]:[r_2]? \langle s'[r]:T' \rangle . T} \quad \text{(End)} \frac{}{\Gamma \vdash \mathbf{0} \triangleright \emptyset} \\
\\
\text{(If)} \frac{\Gamma \Vdash e:\mathbb{B} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta}{\Gamma \vdash \text{if } e \text{ then } P \text{ else } P' \triangleright \Delta} \quad \text{(Par)} \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta'}{\Gamma \vdash P \mid P' \triangleright \Delta \cdot \Delta'} \\
\\
\text{(Res1)} \frac{x^\sharp(\Gamma, \Delta) \quad \Gamma \cdot x:S \vdash P \triangleright \Delta}{\Gamma \vdash (\nu x)P \triangleright \Delta} \quad \text{(Rec)} \frac{\Gamma \cdot X:t \vdash P \triangleright \Delta \cdot s[r]:T}{\Gamma \vdash (\mu X)P \triangleright \Delta \cdot s[r]:(\mu t)T}
\end{array}$$

Figure 2.3: Typing Rules for Fault-Tolerant Systems.

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 $\text{nsr}(\Delta)$.

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

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

2. The failure pattern $\text{FP}_{\text{uget}}(s, r_1, r_2, l)$ is always valid.

3. The pattern $\text{FP}_{\text{ml}}(s, r_1, r_2, l)$ is valid iff $\text{FP}_{\text{uskip}}(s, r_2, r_1, l)$ is valid.

4. If $\text{FP}_{\text{crash}}(P)$ and $s[r] \in A(P)$ then eventually $\text{FP}_{\text{uskip}}(s, r_2, r, l)$ and also $\text{FP}_{\text{wskip}}(s, r_2, r, l)$ for all r_2, l .

5. If $\text{FP}_{\text{crash}}(P)$ and $s[r] \in A(P)$ then eventually $\text{FP}_{\text{ml}}(s, r_1, r, l)$ for all r_1, l .

6. If $\text{FP}_{\text{wskip}}(s, r_1, r_2)$ then $s[r_2]$ is crashed, i.e., the system does no longer contain an actor $s[r_2]$ and the message queue $s_{r_2 \rightarrow r_1}$ is empty.

The crash of a process should not block strongly reliable actions, i.e., only processes with $\text{nsr}(P)$ 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 $\text{FP}_{\text{wskip}}(s, r_1, r_2)$ is false as long as $s[r_2]$ 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 $P \mapsto P'$, 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 $\bar{a}[n](s).Q$ or $a[r](s).Q$ with $a:G$, then there are $\bar{a}[n](s).Q_n$ or $a[r_i](s).Q_i$ for all $1 \leq r_i < n$.*

1. *Then either P does not contain any action prefixes or $P \mapsto P'$.*
2. *If P does not contain recursion, then there exists P' such that $P \mapsto^* P'$ 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 $(\odot_{1 \leq i \leq n} \pi_i).G$ abbreviate the sequence $\pi_1 \dots \pi_n.G$ to simplify the presentation, where $G \in \mathcal{G}$ is a global type and π_1, \dots, π_n are sequences of prefixes. More precisely, each π_i is of the form $\pi_{i,1} \dots \pi_{i,m}$ and each $\pi_{i,j}$ is a type prefix of the form $r_1 \rightarrow_u r_2:l\langle S \rangle$ or $r \rightarrow_w R:l_1.T_1 \oplus \dots \oplus l_n.T_n \oplus l_d$, where the latter case represents a weakly reliable branching prefix with the branches l_1, \dots, l_n, l_d , the default branch l_d , and where the next global type provides the missing specification for the default case.

Let $(\odot_{1 \leq i \leq n} \pi_i).P$ abbreviate the sequence $\pi_1 \dots \pi_n.P$, where $P \in \mathcal{P}$ is a process and π_1, \dots, π_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 $f : \tilde{X} \rightarrow \perp$ be a function with domain $\tilde{X} = X_1 \times \dots \times X_n$ and $P \in \mathcal{P}$ a process. Then $f(x_1, \dots, x_n).P$ is also a process in \mathcal{P} .

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

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

$$\text{(Func)} \frac{f \in F_{\tilde{X}, \perp} \quad \Gamma \Vdash (x_1, \dots, x_n) : \tilde{X} \quad \Gamma \vdash P \triangleright \Delta}{\Gamma \vdash f(x_1, \dots, x_n).P \triangleright \Delta}$$

In (Func) we require that f is a function from \tilde{X} to \perp and expression (x_1, \dots, 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 $S \ a \ b$.

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

The values in $S \ a \ b$ are either of the form $A \ a$, $B \ b$, $C \ a \ b$, 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 $S \ a \ b$.

3 Model

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

3.1 Sorts

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

First, we have `Bool` which we define as a set.

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

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

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 .

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

A value of type `Maybe a` can have the form `Just a` or `Nothing`. Some examples include `Just 4` of type `Maybe \mathbb{N}` , `Just false` of type `Maybe Bool`, and `Nothing`. `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 `Nothing` is used. It can be of type `Maybe \mathbb{N}` , `Maybe Bool`, or `Maybe b` for any other type b . We use `Maybe a` where optional values are needed.

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

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

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

Promise a has two possible forms. Promise (Maybe (Proposal a)) and Nack. Possible values include Nack and Promise (Just (Proposal 1 false)) of type Promise Bool. The actual type of Nack, much like that of 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 Promise 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 Promise (Maybe (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 p and a quorum Q .

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

$$G_{p,AQ} = (\mu t) \left(\bigodot_{a \in Q} p \rightarrow_u a : l1a \langle \mathbb{N} \rangle \right) . \left(\bigodot_{a \in Q} a \rightarrow_u p : l1b \langle \text{Promise Value} \rangle \right) . \\ p \rightarrow_w Q : \left\{ \text{Accept} . \left(\bigodot_{a \in Q} p \rightarrow_u a : l2a \langle \text{Proposal Value} \rangle \right) . 0 \oplus \text{Restart} . t \oplus \text{Abort} . 0 \right\}$$

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 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 . 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.

$$\text{proposalNumber} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$\text{proposalNumber}_p(n)$ 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.

$$\forall p, q \in \mathbb{P}. \forall n, m \in \mathbb{N} : p \neq q \rightarrow \text{proposalNumber}_p(n) \neq \text{proposalNumber}_q(m)$$

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

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

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

$$\text{promiseValue} : \text{list of Promise } a \rightarrow a$$

$\text{promiseValue}(ps)$ 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 $\text{Promise Just } v$. With this function we can model the picking of a value for a proposal in phase 2a.

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

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

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

$\text{promiseCount}(ps)$ takes a list of promises ps and counts the number of promises that have the form $\text{Promise } m$.

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

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

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

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

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

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

$$\text{genQ} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \text{list of } \mathbb{N}$$

$\text{genQ}(p, c_A, c_P)$ returns a randomly selected set Q with $Q \subseteq A = \{1, \dots, c_A\}$ and $|Q| > \frac{|A|}{2}$ 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 with which a value can be chosen [7]. We use this function when initiating the proposers to give them a quorum with which they communicate.

$$\begin{aligned} \text{positionHelper} &: \mathbb{N} \times \mathbb{N} \times \text{list of } \mathbb{N} \rightarrow \mathbb{N} \\ \text{positionHelper}(n, c, (n\#_)) &= c + 1 \\ \text{positionHelper}(n, c, (x\#xs)) &= \text{positionHelper}(n, c + 1, xs) \end{aligned}$$

$$\begin{aligned} \text{position} &: \mathbb{N} \times \text{list of } \mathbb{N} \rightarrow \mathbb{N} \\ \text{position}(n, xs) &= \text{positionHelper}(n, 0, xs) \end{aligned}$$

$\text{position}(n, xs)$ calculates the position of the first occurrence of n in xs . It uses $\text{positionHelper}(n, c, xs)$, which recursively checks each element of the list. $\text{positionHelper}(n, c, xs)$ and thus $\text{position}(n, xs)$ are purposely left undefined for cases where n is not an element of xs . We observe that $1 \leq \text{position}(n, xs) \leq |xs|$ for every list xs and $n \in xs$.

3.4 Processes

3.4.1 System Initialization

Let c_A and c_P be the number of acceptors and proposers respectively. Let $Q_k = \text{genQ}(k, c_A, c_P)$ 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 $Q_5 = [2, 3]$. 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 $|Q_5| + 1 = 2 + 1 = 3$.

$$\begin{aligned} \text{Sys}(c_A, c_P) = & \bar{o}[2](z) \cdot P_{\text{init}}^P(c_A + c_P, |Q_{c_A+c_P}| + 1, \{1, \dots, |Q_{c_A+c_P}|\}, c_A, n_{c_A+c_P}, []) \\ & | o[1](z) \cdot \Pi_{c_A+1 \leq k < c_A+c_P} P_{\text{init}}^P(k, |Q_k| + 1, \{1, \dots, |Q_k|\}, c_A, n_k, []) \\ & | \Pi_{1 \leq j \leq c_A} P_{\text{init}}^A(j, c_A, c_P, n_j, \text{pr}_j) \end{aligned}$$

$$\begin{aligned} P_{\text{init}}^P(k, p, R, c_A, n, \vec{V}) = & \bar{b}_k[p](s) \cdot P^P(s, k, p, R, c_A, n, \vec{V}) \\ P_{\text{init}}^A(j, c_A, c_P, n, \text{pr}) = & \Pi_{c_A+1 \leq k \leq c_A+c_P} \\ & (\text{if } j \notin Q_k \text{ then } 0 \text{ else } b_k[\text{position}(j, Q_k)](s) \cdot P_1^A(s, \text{position}(j, Q_k), |Q_k| + 1, n, \text{pr})) \end{aligned}$$

$\text{Sys}(c_A, c_P)$, $P_{\text{init}}^P(k, p, R, c_A, n, \vec{V})$, and $P_{\text{init}}^A(j, c_A, c_P, n, \text{pr})$ describe the system initialization.

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

$P_{\text{init}}^P(k, p, R, c_A, n, \vec{V})$ 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 $n = n_k$, and a vector \vec{V} . k uniquely identifies a proposer when calling `proposalNumber`. The proposer's role p is initialized with $|Q_k| + 1$. The set of roles R contains the roles the acceptors in Q_k will use in the proposer's session. Register n 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. \vec{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 $P^P(s, k, p, Q, c_A, n, \vec{V})$. 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 proposals. A new leader is elected via the same mechanism when the previous leader terminates or crashes.

$P_{\text{init}}^A(j, c_A, c_P, n, \text{pr})$ 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 $n = n_j$, if available, and initial knowledge for the most recently accepted proposal in register $\text{pr} = \text{pr}_j$, if available. n holds a value of type `Maybe \mathbb{N}` and pr holds a value of type `Maybe (Proposal Value)` thus both can contain `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 and pr . 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 $P_1^A(s, a, p, n, pr)$.

3.4.2 Proposer

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

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

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 \vec{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 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

$$\begin{aligned}
P_1^A(s, a, p, n, pr) &= (\mu X) s[p, a]?_u l1a \langle \perp \rangle (n') . \\
&(\text{if } n' = \perp \text{ then } s[a, p]!_u l1b \langle \perp \rangle . P_2^A(s, a, p, n, pr) \\
&\quad \text{else (if gt}(n', n) \\
&\quad \quad \text{then update}(n, n') . s[a, p]!_u l1b \langle \text{Promise } pr \rangle . P_2^A(s, a, p, n, pr) \\
&\quad \quad \text{else } s[a, p]!_u l1b \langle \text{Nack} \rangle . P_2^A(s, a, p, n, pr)) \\
P_2^A(s, a, p, n, pr) &= s[p, a]?_w \{ \text{Accept}.s[p, a]?_u l2a \langle \perp \rangle (pr') . \\
&(\text{if } pr' = \perp \text{ then } 0 \\
&\quad \text{else (if ge}(nFromProposal(pr'), n) \\
&\quad \quad \text{then update}(pr, \text{Just } pr') . \text{update}(n, \text{Just}(nFromProposal(pr')))) . 0 \\
&\quad \text{else } 0) \} \\
&\oplus \text{Restart}.X \\
&\oplus \text{Abort}.0 \}
\end{aligned}$$

An acceptor a has a corresponding subprocess for each proposer p with quorum Q_p where $a \in Q_p$. These subprocesses behave like $P_1^A(s, a, p, n, pr)$ and access the same registers n and pr . Updating these registers with update (r, v) 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 Promise pr or Nack, depending on the values of n' and n . If the acceptor does not receive a proposal number then it sends \perp to the proposer. Sending \perp 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 *Restart* branch this particular subprocess of the acceptor restarts the algorithm to match the corresponding proposer.

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

3.5 Failure Patterns

Chandra and Toueg introduce a class of failure detectors $\diamond\mathcal{S}$ called *eventually strong* in [3]. Failure detectors in $\diamond\mathcal{S}$ 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, FP_{uskip} and FP_{ml} are implemented with a failure detector in $\diamond\mathcal{S}$ for phases 1a and 1b. For phase 2a with label *l2a* both FP_{uskip} and FP_{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 FP_{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, FP_{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., $Q \setminus \{a\}$ is still a quorum.

In Paxos there is no need to reject outdated messages so FP_{uget} is implemented with a constant true.

These failure patterns satisfy the Conditions 1.1–1.6.

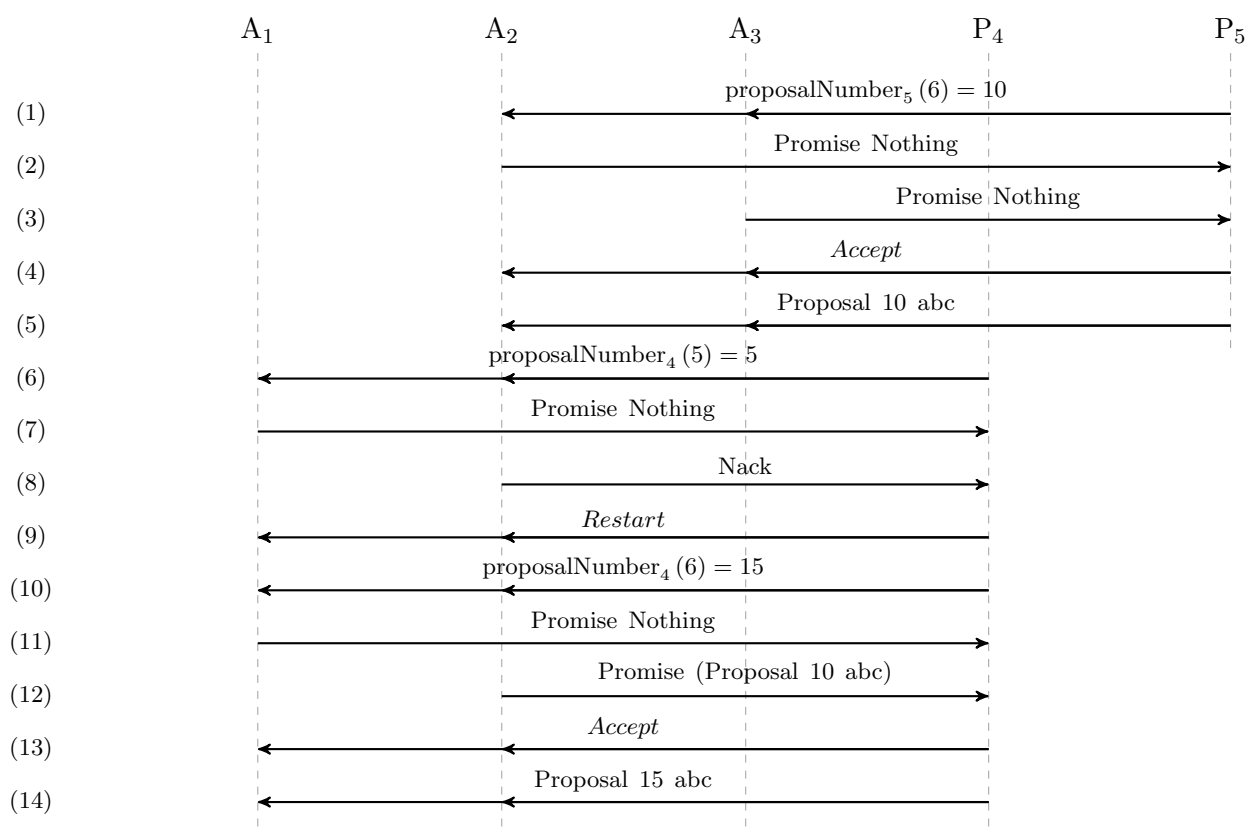


Figure 3.1: Example scenario with 3 acceptors and 2 proposers.

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 A₁, A₂, and A₃ are the acceptors and P₄ and P₅ are the proposers. P₅ is elected to be the leader. In steps (1) to (5), P₅ completes the Paxos algorithm with A₂ and A₃ and terminates.

At this point A₂ has promised not to accept any proposal numbered less than 10 and has accepted the value abc. So, when P₄ tries to use 5 as its proposal number (6), it receives `Nack` from A₂ (8) and has to restart the algorithm (9).

P₄ then runs through the Paxos algorithm with A₁ and A₂ starting with a new prepare request (10) with a higher proposal number. In step (12) P₄ learns that value abc with proposal number 10 has already been accepted by A₂. Later, in step (14), P₄ 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 Proposal 10 abc.

In the end all 3 acceptors have accepted the value abc. A_1 and A_2 have accepted Proposal 15 abc and A_3 has accepted Proposal 10 abc.

3.6.2 Formulae

We set $c_A = 3$, $c_P = 2$, $\text{Value} = \{\text{abc}, \text{def}, \dots, \text{vwxyz}\}$. $Q_4 = \text{gen Q}(4, 3, 2) = \{1, 2\}$ will be the quorum for P_4 and $Q_5 = \text{gen Q}(5, 3, 2) = \{2, 3\}$ will be the quorum for P_5 .

System Initialization

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

$$\begin{aligned}
 \text{Sys}(c_A, c_P) &= \text{Sys}(3, 2) = \\
 &o[1](z) . \Pi_{3 < k < 5} P_{\text{init}}^P(4) \text{gen Q}(k, 3, 2)kk[] \\
 &\quad | \bar{o}[2](z) . P_{\text{init}}^P(4) \text{gen Q}(5, 3, 2)55[] \\
 &\quad | \Pi_{1 \leq a \leq 3} P_{\text{init}}^A(a) 432n_a pr_a \\
 &\xrightarrow{*} (\nu z) (\bar{b}_4[4](s) . P^P() = P_4 \\
 &\quad | \bar{b}_5[4](r) . P^P() = P_5 \\
 &\quad | (b_4[1](s) . P_1^A \mid b_5[1](r) . P_1^A) = A_1 \\
 &\quad | (b_4[2](s) . P_1^A \mid b_5[2](r) . P_1^A) = A_2 \\
 &\quad | (b_4[3](s) . P_1^A \mid b_5[3](r) . P_1^A) = A_3 \\
 &\quad | \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
 \end{aligned}$$

Note that the outer session created via shared channel o is not strictly necessary in the model. We apply (Init) once for shared channel b_4 and once again for shared channel b_5 to obtain:

$$\begin{aligned}
 &\xrightarrow{*} (\nu z) (\nu s) (\nu r) (\\
 &(\mu X) \text{update}(n, 5) . \left(\bigodot_{a \in \{1, 2\}} s[4, a]!_u l1a \langle \text{proposalNumber}_4(n) \rangle \right) \dots = P_4 \\
 &\quad | (\mu X) \text{update}(n, 6) . \left(\bigodot_{a \in \{2, 3\}} r[4, a]!_u l1a \langle \text{proposalNumber}_5(n) \rangle \right) \dots = P_5 \\
 &\quad | ((\mu X) s[4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots \mid (\mu X) r[4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots) = A_1 \\
 &\quad | ((\mu X) s[4, 2]?_u l1a \langle \perp \rangle (n') . \text{if} \dots \mid (\mu X) r[4, 2]?_u l1a \langle \perp \rangle (n') . \text{if} \dots) = A_2 \\
 &\quad | ((\mu X) s[4, 3]?_u l1a \langle \perp \rangle (n') . \text{if} \dots \mid (\mu X) r[4, 3]?_u l1a \langle \perp \rangle (n') . \text{if} \dots) = A_3 \\
 &\quad | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
 \end{aligned}$$

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

The Happy Path

After applying the updates in P_4 and P_5 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 $\text{proposalNumber}_5(6) = 10$ to A_2 and A_3 .

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4(5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4(5) \rangle \dots = P_4 \\
& | (\mu X) r [2, 4]?_u l1b \langle \perp \rangle (v_2) . r [3, 4]?_u l1b \langle \perp \rangle (v_3) \dots = P_5 \\
& | ((\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) r [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots) = A_1 \\
& | ((\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) \text{if } 10 = \perp \text{ then } s [2, 4]!_u l1b \langle \perp \rangle . P_2^A \text{ else } \dots) = A_2 \\
& | ((\mu X) s [4, 3]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) \text{if } 10 = \perp \text{ then } s [3, 4]!_u l1b \langle \perp \rangle . P_2^A \text{ else } \dots) = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

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

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4(5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4(5) \rangle \dots = P_4 \\
& | (\mu X) r [2, 4]?_u l1b \langle \perp \rangle (v_2) . r [3, 4]?_u l1b \langle \perp \rangle (v_3) \dots = P_5 \\
& | ((\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) r [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots) = A_1 \\
& | ((\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) \text{if gt}(10, \text{Nothing}) \text{ then } \dots \text{ else } \dots) = A_2 \\
& | ((\mu X) s [4, 3]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) \text{if gt}(10, \text{Nothing}) \text{ then } \dots \text{ else } \dots) = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

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

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4(5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4(5) \rangle \dots = P_4 \\
& | (\mu X) r [2, 4]?_u l1b \langle \perp \rangle (v_2) . r [3, 4]?_u l1b \langle \perp \rangle (v_3) \dots = P_5 \\
& | ((\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) r [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots) = A_1 \\
& | ((\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) r [2, 4]!_u l1b \langle \text{Promise Nothing} \rangle . P_2^A) = A_2 \\
& | ((\mu X) s [4, 3]?_u l1a \langle \perp \rangle (n') . \text{if} \dots | (\mu X) r [3, 4]!_u l1b \langle \text{Promise Nothing} \rangle . P_2^A) = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

We apply (USend) and (UGet) twice to do just that. Note that we also apply (USkip), (If-T), and then (USend) to A_1 and then (ML) to P_5 to discard the dummy message \perp . All three acceptors move into P_2^A .

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle \dots = P_4 \\
& | (\mu X) r [4, \{2, 3\}]!_w \text{Accept}. r [4, 2]!_u l2a \langle \text{Proposal } 10 \text{ abc} \rangle \dots = P_5 \\
& | ((\mu X) \dots | (\mu X) r [4, 1]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \}) = A_1 \\
& | ((\mu X) \dots | (\mu X) r [4, 2]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \}) = A_2 \\
& | ((\mu X) \dots | (\mu X) r [4, 3]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \}) = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

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

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle \dots = P_4 \\
& | (\mu X) r [4, 2]!_u l2a \langle \text{Proposal } 10 \text{ abc} \rangle . r [4, 3]!_u l2a \langle \text{Proposal } 10 \text{ abc} \rangle \dots = P_5 \\
& | ((\mu X) \dots | (\mu X) r [4, 1]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \}) = A_1 \\
& | ((\mu X) \dots | (\mu X) r [4, 2]?_u l2a \langle \perp \rangle (pr') . \text{if} \dots = A_2 \\
& | ((\mu X) \dots | (\mu X) r [4, 3]?_u l2a \langle \perp \rangle (pr') . \text{if} \dots = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

Now P_5 can send its proposal to A_2 and A_3 and terminate. P_4 will be the new leader. To do so we apply (USend) and (UGet) twice. A_2 and A_3 accept the proposal and the respective subprocesses terminate. Note that we apply (WSkip) in A_1 and terminate that subprocess as well.

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle \dots = P_4 \\
& | (\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if} \dots = A_1 \\
& | (\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if} \dots = A_2 \\
& | (\mu X) s [4, 3]?_u l1a \langle \perp \rangle (n') . \text{if} \dots = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

At this point the registers of A_2 and A_3 hold $n = 10$ and $pr = \text{Proposal } 10 \text{ abc}$. A_1 has not updated its registers $n = \text{Nothing}$ and $pr = \text{Nothing}$.

Restarting The Algorithm

Next, P_4 sends prepare requests with a proposal number less than 10, which is rejected by A_2 . P_4 then decides to restart the algorithm. We apply (USend) and (UGet) twice. We also apply (USkip) once in A_3 .

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [1, 4]?_u l1b \langle \perp \rangle (v_1) . s [2, 4]?_u l1b \langle \perp \rangle (v_2) \dots = P_4 \\
& | (\mu X) \text{if } 5 = \perp \text{ then } P_2^A \text{ else } \dots = A_1 \\
& | (\mu X) \text{if } 5 = \perp \text{ then } P_2^A \text{ else } \dots = A_2 \\
& | (\mu X) \text{if } \perp = \perp \text{ then } P_2^A \text{ else } \dots = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

We can apply (If-T) to A_3 and (If-F) to A_1 and A_2 . A_3 moves directly to P_2^A whereas A_1 and A_2 send their responses to P_4 before moving to P_2^A . A_1 also updates its register $n = 5$ via (Func).

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [1, 4]?_u l1b \langle \perp \rangle (v_1) . s [2, 4]?_u l1b \langle \perp \rangle (v_2) \dots = P_4 \\
& | (\mu X) s [1, 4]!_u l1b \langle \text{Promise Nothing} \rangle . P_2^A = A_1 \\
& | (\mu X) s [2, 4]!_u l1b \langle \text{Nack} \rangle . P_2^A = A_2 \\
& | (\mu X) r [4, 3]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \} = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

Applying (USend) and (UGet) twice and (If-T) in P_4 yields:

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [4, \{1, 2\}]!_w \text{Restart}.X = P_4 \\
& | (\mu X) s [4, 1]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \} = A_1 \\
& | (\mu X) s [4, 1]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \} = A_2 \\
& | (\mu X) r [4, 3]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \} = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

P_4 sends its decision to restart the algorithm to A_1 and A_2 by applying (WSel) once and (WBran) twice. A_3 terminates after applying (WSkip).

$$\begin{aligned}
& \mapsto^* (\nu z) (\nu s) (\nu r) (\\
& (\mu X) s [4, 1]!_u l1a \langle 15 \rangle . s [4, 2]!_u l1a \langle 15 \rangle \dots = P_4 \\
& | (\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots = A_1 \\
& | (\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if } \dots = A_2 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
\end{aligned}$$

The Happy Path, Again

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. By applying (USend) in the remaining proposer and (UGet), (If-F), (If-T), (Func) in the remaining acceptors we arrive at:

$$\begin{aligned} & \mapsto^* (\nu z) (\nu s) (\nu r) (\\ & (\mu X) s [1, 4]?_u l1b \langle \perp \rangle (v_1) . s [2, 4]?_u l1b \langle \perp \rangle (v_2) . \text{if} \dots = P_4 \\ & | (\mu X) s [1, 4]!_u l1b \langle \text{Promise Nothing} \rangle . P_2^A = A_1 \\ & | (\mu X) s [2, 4]!_u l1b \langle \text{Promise (Proposal 10 abc)} \rangle . P_2^A = A_2 \\ & | \prod_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \prod_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \prod_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : []) \end{aligned}$$

Note that, at this point, A_1 and A_2 have updated their respective registers n to 15.

Because A_2 has already accepted a proposal, it responds to P_4 's prepare request with that proposal. Twice more we apply (USend), (UGet), and (If-F) in P_4 to obtain:

$$\begin{aligned} & \mapsto^* (\nu z) (\nu s) (\nu r) (\\ & (\mu X) s [4, \{1, 2\}]!_w \text{Accept} \dots = P_4 \\ & | (\mu X) s [4, 1]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \} = A_1 \\ & | (\mu X) s [4, 1]?_w \{ \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \} = A_2 \\ & | \prod_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \prod_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \prod_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : []) \end{aligned}$$

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 (WBrn).

$$\begin{aligned} & \mapsto^* (\nu z) (\nu s) (\nu r) (\\ & (\mu X) s [4, 1]!_u l2a \langle \text{Proposal 15 abc} \rangle . s [4, 2]!_u l2a \langle \text{Proposal 15 abc} \rangle . 0 = P_4 \\ & | (\mu X) s [4, 1]?_u l2a \langle \perp \rangle (pr') . \text{if} \dots = A_1 \\ & | (\mu X) s [4, 2]?_u l2a \langle \perp \rangle (pr') . \text{if} \dots = A_2 \\ & | \prod_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \prod_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \prod_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : []) \end{aligned}$$

Then we apply (USend) and (UGet) to send the proposal from P_4 to the acceptors. P_4 terminates. We apply (If-F), (If-T), and (Func) twice to the acceptors. With that, A_1 and A_2 have accepted P_4 's proposal.

$$\mapsto^* (\nu z) (\nu s) (\nu r) (\prod_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \prod_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \prod_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])$$

Afterwards A_1 and A_2 have $n = 15$ and $pr = \text{Proposal 15 abc}$ and A_3 has $n = 10$ and $pr = \text{Proposal 10 abc}$. All acceptors have accepted the value abc .

4 Analysis

We take the model from the previous chapter, type check it, and discuss how we 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 type is $G = 0$. The projection of G to a local type is $G \downarrow_k = 0$ for every k .

For $1 \leq a \leq c_A$ and $c_A + 1 \leq p \leq c_A + c_P$ we project the global type G_{p,A_Q} to local types $G_{p,A_Q} \downarrow_p$ and $G_{p,A_Q} \downarrow_a$.

$$G_{p,A_Q} \downarrow_p = (\mu t) \left(\odot_{a \in Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) . \left(\odot_{a \in Q} [a]?_u l1b \langle \text{Promise Value} \rangle \right) . \\ [Q]!_w \left\{ \text{Accept} . \left(\odot_{a \in Q} [a]!_u l2a \langle \text{Proposal Value} \rangle \right) . 0, \text{Restart} . t, \text{Abort} . 0 \right\}$$

$G_{p,A_Q} \downarrow_p$ 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.

$$G_{p,A_Q} \downarrow_a = (\mu t) [p]?_u l1a \langle \mathbb{N} \rangle . [p]!_u l1b \langle \text{Promise Value} \rangle . \\ [p]?_w \left\{ \text{Accept} . [p]?_u l2a \langle \text{Proposal Value} \rangle . 0, \text{Restart} . t, \text{Abort} . 0 \right\}_{\text{Abort}}$$

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

4.2 Type Check

$$\Gamma = (o : G \cdot b_{c_A+1} : G_{p,A_Q} \cdot \dots \cdot b_{c_A+c_P} : G_{p,A_Q}) \cdot (1 : \mathbb{N} \cdot \dots \cdot c_A : \mathbb{N} \cdot c_A + 1 : \mathbb{N} \cdot \dots \cdot c_A + c_P : \mathbb{N} \cdot c_P : \mathbb{N})$$

Γ contains the type for our shared channels o and b_p where $c_A + 1 \leq p \leq c_A + c_P$.

We start the type check with the global environment Γ and the entry point of the model $\text{Sys}(c_A, c_P)$. 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

$$\frac{\begin{array}{c} (S_1) \\ \Gamma \vdash \bar{o}[2](z) \dots \triangleright \emptyset \end{array} \quad \frac{\begin{array}{c} (S_2) \\ \Gamma \vdash o[1](z) \dots \triangleright \emptyset \end{array} \quad \frac{\begin{array}{c} (S_3) \\ \Gamma \vdash \Pi_{1 \leq a \leq c_A} P_{\text{init}}^A(\dots) \triangleright \emptyset \end{array}}{\Gamma \vdash o[1](z) \dots \mid \Pi_{1 \leq a \leq c_A} P_{\text{init}}^A(\dots) \triangleright \emptyset} (\text{Par})}{\Gamma \vdash \bar{o}[2](z) \cdot P_{\text{init}}^P(\dots) \mid o[1](z) \dots \mid \Pi_{1 \leq a \leq c_A} P_{\text{init}}^A(\dots) \triangleright \emptyset} (\text{Par})$$

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

$$(S_1) = \frac{\begin{array}{c} (P) \\ \Gamma \vdash \overline{b_{c_A+c_P}}[c_A+1](s) \cdot P^P() \triangleright z[2] : G \vdash_2 \end{array}}{\Gamma \vdash \bar{o}[2](z) \cdot P_{\text{init}}^P(c_A+1) \text{ gen } Q(c_A+c_P, c_A, c_P) c_A+c_P c_A+c_P \square \triangleright \emptyset} (\text{Rec})$$

In (S_1) we apply (Rec) once. The rest of the proof tree is in (P) . Note that, since $G \vdash_2 = 0$, $z[2] : G \vdash_2 = \emptyset$. This is relevant later when continuing (P) .

$$(S_2) = \frac{\frac{\begin{array}{c} (P) \\ \Gamma \vdash \overline{b_{c_A+1}}[c_A+1](s) \cdot P^P() \triangleright \emptyset \end{array} \quad \dots \quad \begin{array}{c} (P) \\ \Gamma \vdash \overline{b_{c_A+c_P-1}}[c_A+1](s) \cdot P^P() \triangleright \emptyset \end{array}}{\Gamma \vdash \Pi_{c_A < k < c_A+c_P} P_{\text{init}}^P(c_A+1) \text{ gen } Q(k, c_A, c_P) k k \square \triangleright z[1] : G \vdash_1} (\text{Par})^{c_P-1}}{\Gamma \vdash o[1](z) \cdot \Pi_{c_A < k < c_A+c_P} P_{\text{init}}^P(\dots) \triangleright \emptyset} (\text{Acc})$$

Applying (Acc) in (S_2) requires that $o : G \in \Gamma$. (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 $G \vdash_1 = 0$, $z[1] : G \vdash_1 = \emptyset$. The session environment Δ in (P) is empty for every proposer.

$$(A_1) \quad \frac{\Gamma \vdash P_1^A \triangleright s[a] : G_{p,A_Q} \upharpoonright_a}{\Gamma \vdash b_k[a](s) \cdot P_1^A \triangleright \emptyset} (\text{Acc}) \quad \dots \quad (\text{Par})^{c_P}$$

$$(S_3) = \frac{\Gamma \vdash \Pi_{c_A < k \leq c_A + c_P} b_k[c_A](s) \cdot P_1^A \triangleright \emptyset}{\Gamma \vdash \Pi_{1 \leq j \leq c_A} (\Pi_{c_A < k \leq c_A + c_P} b_k[j](s) \cdot P_1^A) \triangleright \emptyset} \dots (\text{Par})^{c_A}$$

(Par) is applied c_A times to separate the individual acceptors and then c_P times for each acceptor to separate the individual subprocesses. Since every subprocess of every acceptor behaves like P_1^A and has the same local type, the same proof tree (A_1) can be applied. Applying (Acc) to every subprocess of every acceptor requires $\forall k \in \mathbb{N} : (c_A + 1 \leq k \wedge k \leq c_A + c_P) \rightarrow b_k : G_{p,A_Q} \in \Gamma$.

Note that only one acceptor and one of its subprocesses is shown in (S_3) . 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

Let $p = c_A + 1$, $Q = \text{gen } Q(k, c_A, c_P)$, $n = k$, $m = k$, $\vec{V} = []$ where $c_A < k \leq c_A + c_P$. This gives us the values for the arguments of P_{init}^P . We observe that $\Gamma \Vdash p : \mathbb{N}$, $\Gamma \Vdash k : \mathbb{N}$, $\Gamma \Vdash n : \mathbb{N}$, $\Gamma \Vdash m : \mathbb{N}$, and $\Gamma \Vdash Q : \text{list of } \mathbb{N}$. n contains a natural number. p , k , and m are natural numbers and Q is a list of natural numbers under global environment Γ .

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

$$T_{\text{acc}}^P = \left(\odot_{a \in Q} [a]!_u l2a \langle \text{Proposal Value} \rangle \right) . 0$$

$$T_{\text{branch}}^P = [Q]!_w \{ \text{Accept} . T_{\text{acc}}^P \oplus \text{Restart} . t \oplus \text{Abort} . 0 \}$$

Note that $G_{p,A_Q} \upharpoonright_p = (\mu t) \left(\odot_{a \in Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \cdot \left(\odot_{a \in Q} [a]?_u l1b \langle \text{Promise Value} \rangle \right) \cdot T_{\text{branch}}^P$.

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

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

$$pn = \text{proposalNumber}_m(n)$$

$$prop = \text{Proposal} \left(\text{proposalNumber}_m(n) \right) \left(\text{promiseValue}(\vec{V}) \right)$$

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

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 .

$$\Gamma'' = \Gamma' \cdot v_1 : \text{Promise Value} \cdot \dots \cdot v_{|Q|} : \text{Promise Value}$$

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

$$\begin{array}{c}
(P_t) \qquad \qquad \qquad (P_f) \\
\frac{\Gamma'' \vdash s[p, Q]!_w \text{Restart}.X \triangleright s[p] : T_{\text{branch}}^P \quad \Gamma'' \vdash s[p, Q]!_w \text{Accept} \dots \triangleright s[p] : T_{\text{branch}}^P}{\Gamma'' \vdash \text{if } e \text{ then } s[p, Q]!_w \text{Restart}.X \text{ else } s[p, Q]!_w \text{Accept} \dots \triangleright s[p] : T_{\text{branch}}^P} \text{ (If)} \\
\frac{\Gamma' \vdash \left(\bigodot_{a \in Q} s[a, p]?_u l1b \langle \perp \rangle (v_a) \right) \dots \triangleright s[p] : \left(\bigodot_{a \in Q} [a]?_u l1b \langle \text{Promise Value} \rangle \right)}{\Gamma' \vdash \left(\bigodot_{a \in Q} s[a, p]?_u l1b \langle \perp \rangle (v_a) \right) \dots \triangleright s[p] : \left(\bigodot_{a \in Q} [a]?_u l1b \langle \text{Promise Value} \rangle \right)} \text{ (UGet)}^{|Q|} \\
\frac{\Gamma' \vdash \left(\bigodot_{a \in Q} s[p, a]!_u l1a \langle pn \rangle \right) \dots \triangleright s[p] : \left(\bigodot_{a \in Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots}{\Gamma' \vdash \left(\bigodot_{a \in Q} s[p, a]!_u l1a \langle pn \rangle \right) \dots \triangleright s[p] : \left(\bigodot_{a \in Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots} \text{ (Func)} \\
\frac{\Gamma' \vdash \text{update}(n, n+1) \dots \triangleright s[p] : \left(\bigodot_{a \in Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots}{\Gamma' \vdash \text{update}(n, n+1) \dots \triangleright s[p] : \left(\bigodot_{a \in Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots} \text{ (Rec)} \\
(P) = \frac{\Gamma \vdash (\mu X) \text{update}(n, n+1) \dots \triangleright s[p] : (\mu t) \left(\bigodot_{a \in Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots}{\Gamma \vdash \overline{b_{MMM}}[p](s).P^P() \triangleright \emptyset} \text{ (Req)}
\end{array}$$

(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 $s[p] : G_{p, A_Q} \vdash_p$ 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 in Q . (UGet) expands the session environment to Γ'' . (If) splits the proof tree into (P_t) and (P_f).

$$(P_t) = \frac{\frac{\Gamma'' \vdash X \triangleright s[p] : t}{\Gamma'' \vdash X \triangleright s[p] : t} \text{ (Var)}}{\Gamma'' \vdash s[p, Q]!_w \text{Restart}.X \triangleright s[p] : [Q]!_w \{ \text{Accept}.T_{\text{acc}}^P \oplus \text{Restart}.t \oplus \text{Abort}.0 \}} \text{ (WSel)}$$

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

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

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

First, we define the arguments of P_{init}^A and P_1^A . Let $a = j$ and $p = c_A + 1$ where $1 \leq j \leq c_A$. With session environment Γ we have $\Gamma \Vdash a : \mathbb{N}$ and $\Gamma \Vdash p : \mathbb{N}$.

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

$$\begin{aligned} P_{\text{acc}}^A &= s[p, a]?_u l2a \langle \perp \rangle (pr') . \text{if } pr' = \perp \\ &\quad \text{then } 0 \\ &\quad \text{else if } \text{ge}(\text{nFromProposal}(pr'), n) \\ &\quad \quad \text{then } \text{update}(pr, pr') . \text{update}(n, \text{Just } \text{nFromProposal}(pr')) . 0 \\ &\quad \text{else } 0 \end{aligned}$$

We can see that P_2^A contains P_{acc}^A as $P_2^A = s[p, a]?_w \{ \text{Accept}. P_{\text{acc}}^A \oplus \text{Restart}. X \oplus \text{Abort}. 0 \}$.

$$P_t^A = \text{update}(n, n') . s[a, p]!_u l1b \langle \text{Promise } pr \rangle . P_2^A$$

$$P_f^A = s[a, p]!_u l1b \langle \text{Nack} \rangle . P_2^A$$

$$P_{\text{gt}}^A = \text{if } \text{gt}(n', n) \text{ then } P_t^A \text{ else } P_f^A$$

With P_{gt}^A , P_1^A can be written as $P_1^A = (\mu X) s[p, a]?_u l1a \langle \perp \rangle (n') . \text{if } n' = \perp \text{ then } s[a, p]!_u l1b \langle \perp \rangle . P_2^A \text{ else } P_{\text{gt}}^A$.

$$T_{\text{acc}}^A = [p]?_u l2a \langle \text{Proposal Value} \rangle . 0$$

$$T_{\text{branch}}^A = [p]?_w \{ \text{Accept}. T_{\text{acc}}^A, \text{Restart}. t, \text{Abort}. 0 \}_{\text{Abort}}$$

$$T_{\text{lb}}^A = [p]!_u l1b \langle \text{Promise Value} \rangle . T_{\text{branch}}^A$$

The acceptor's local type $G_{p, A_Q} \vdash_a$ can be written as $G_{p, A_Q} \vdash_a = (\mu t) [p]?_u l1a \langle \mathbb{N} \rangle . T_{\text{lb}}^A$.

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' .

$$\begin{aligned}
& (A_2) \\
& \frac{\Gamma'' \vdash P_2^A \triangleright s[a] : T_{\text{branch}}^A}{\Gamma'' \vdash s[a, p]!_u l1b \langle \perp \rangle . P_2^A \triangleright s[a] : T_{1b}^A} \text{(USend)} \quad (A_{gt}) \\
& \frac{\Gamma'' \vdash P_{gt}^A \triangleright s[a] : T_{1b}^A}{\Gamma'' \vdash \text{if } n' = \perp \text{ then } s[a, p]!_u l1b \langle \perp \rangle . P_2^A \text{ else if } gt(n', n) \dots \triangleright s[a] : T_{1b}^A} \text{(If)} \\
& \frac{\Gamma'' \vdash \text{if } n' = \perp \text{ then } s[a, p]!_u l1b \langle \perp \rangle . P_2^A \text{ else if } gt(n', n) \dots \triangleright s[a] : T_{1b}^A}{\Gamma' \vdash s[p, a]?_u l1a \langle \perp \rangle (n') . \text{if } n' = \perp \dots \triangleright s[a] : [p]?_u l1a \langle \mathbb{N} \rangle . T_{1b}^A} \text{(UGet)} \\
(A_1) &= \frac{\Gamma' \vdash s[p, a]?_u l1a \langle \perp \rangle (n') . \text{if } n' = \perp \dots \triangleright s[a] : [p]?_u l1a \langle \mathbb{N} \rangle . T_{1b}^A}{\Gamma \vdash (\mu X) s[p, a]?_u l1a \langle \perp \rangle (n') \dots \triangleright s[a] : (\mu t) [p]?_u l1a \langle \mathbb{N} \rangle \dots} \text{(Rec)}
\end{aligned}$$

After applying (Acc) in (S_3) the session environment contains the acceptor's local type $G_{p, A_Q} \upharpoonright_a$. We apply (Rec) and (UGet) and then 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) .

$$\begin{aligned}
& (A_t) \quad (A_f) \\
(A_{gt}) &= \frac{\Gamma'' \vdash P_t^A \triangleright s[a] : T_{1b}^A \quad \Gamma'' \vdash P_f^A \triangleright s[a] : T_{1b}^A}{\Gamma'' \vdash \text{if } gt(n', n) \text{ then } P_t^A \text{ else } P_f^A \triangleright s[a] : T_{1b}^A} \text{(If)}
\end{aligned}$$

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

$$\begin{aligned}
& (A_2) \\
(A_t) &= \frac{\Gamma'' \vdash P_2^A \triangleright s[a] : T_{\text{branch}}^A}{\Gamma'' \vdash s[a, p]!_u l1b \langle \text{Nack} \rangle . P_2^A \triangleright s[a] : [p]!_u l1b \langle \text{Promise Value} \rangle . T_{\text{branch}}^A} \text{(USend)} \\
& (A_2) \\
(A_f) &= \frac{\Gamma'' \vdash P_2^A \triangleright s[a] : T_{\text{branch}}^A}{\Gamma'' \vdash s[a, p]!_u l1b \langle \text{Nack} \rangle . P_2^A \triangleright s[a] : [p]!_u l1b \langle \text{Promise Value} \rangle . T_{\text{branch}}^A} \text{(USend)}
\end{aligned}$$

In both, (A_t) and (A_f) , we apply (USend). Now we can examine (A_2) , which is the proof tree for P_2^A .

$$\begin{aligned}
& (A_{\text{Accept}}) \\
(A_2) &= \frac{\Gamma'' \vdash P_{\text{acc}}^A \triangleright s[a] : T_{\text{acc}}^A \quad \frac{\Gamma'' \vdash X \triangleright s[a] : t}{\Gamma'' \vdash P_2^A \triangleright s[a] : T_{\text{branch}}^A} \text{(Var)} \quad \frac{\Gamma'' \vdash 0 \triangleright s[a] : 0}{\Gamma'' \vdash P_2^A \triangleright s[a] : T_{\text{branch}}^A} \text{(End)}}{\Gamma'' \vdash P_2^A \triangleright s[a] : T_{\text{branch}}^A} \text{(WBran)}
\end{aligned}$$

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).

$$\begin{array}{c}
(A_{update}) \\
\frac{\frac{\Gamma''' \vdash 0 \triangleright s[a] : 0}{\Gamma''' \vdash 0 \triangleright s[a] : 0} (End) \quad \frac{\frac{\Gamma''' \vdash \text{update}(\text{pr}, \text{pr}') \dots \triangleright s[a] : 0 \quad \Gamma''' \vdash 0 \triangleright s[a] : 0}{\Gamma''' \vdash \text{if ge}(\text{nFromProposal}(\text{pr}'), \text{n}) \text{ then } \dots \text{ else } 0 \triangleright s[a] : 0} (If)}{\Gamma''' \vdash \text{if } \text{pr}' = \perp \text{ then } 0 \text{ else } \dots \triangleright s[a] : 0} (If) \\
(A_{Accept}) = \frac{\Gamma''' \vdash \text{if } \text{pr}' = \perp \text{ then } 0 \text{ else } \dots \triangleright s[a] : 0}{\Gamma'' \vdash s[p, a]?_ul2a \langle \perp \rangle (\text{pr}') \dots \triangleright s[a] : [p]?_ul2a \langle \text{Proposal Value} \rangle .0} (UGet)
\end{array}$$

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}) .

$$\begin{array}{c}
\frac{\Gamma''' \vdash 0 \triangleright s[a] : 0}{\Gamma''' \vdash 0 \triangleright s[a] : 0} (End) \\
(A_{update}) = \frac{\frac{\Gamma''' \vdash \text{update}(\text{n}, \text{Just } \text{nFromProposal}(\text{pr}')) .0 \triangleright s[a] : 0}{\Gamma''' \vdash \text{update}(\text{n}, \text{Just } \text{nFromProposal}(\text{pr}')) .0 \triangleright s[a] : 0} (Func)}{\Gamma''' \vdash \text{update}(\text{pr}, \text{pr}') \dots \triangleright s[a] : 0} (Func)
\end{array}$$

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, 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 [7]).

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 $G_{P,AQ}$. 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 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. The accepted proposal pr is either the acceptor's already accepted proposal pr_a or a proposal that was previously proposed by a proposer. pr is sent over $l1b$ without alteration. The proposer receiving these messages stores them in \vec{V} without changing their values.

Proposers send a message of sort Proposal Value to their quorum of acceptors over label $l2a$. To do so, proposers pick the best value from a proposal in \vec{V} , if any is available, with promiseValue . An entry in \vec{V} contains a proposal prop if it is of the form Promise Just prop . These proposals are either some acceptor's initial accepted proposal or a proposal proposed by a proposer. Not all entries in \vec{V} contain a proposal but if

at least one does, `promiseValue` returns the value of one of them. If no entry in \vec{V} contains a proposal a fresh value is chosen and returned. In both cases the return value of `promiseValue` is not altered before being sent over label *l2a*, which constitutes proposing that value. Acceptors that receive and accept this proposal store it without alteration.

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

4.3.2 Agreement

We examine the flow of accepted values through the network to prove that only a single value is chosen. FP_{crash} guarantees that the set of correct acceptors in the quorum of any correct proposer p is itself a quorum i.e., an accepting set that can choose a value [7]. Should message loss occur in labels *l1a* or *l1b*, p restarts the algorithm. This broadcast is weakly reliable and thus only fails if p crashes because FP_{skip} disallows suspicion of correct proposers by acceptors. Given the definition of `promiseValue`, p 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 p 's quorum. Thus, if a majority of acceptors accept p 's proposal it is sent to every other proposer when they reach phase *1b*. These proposers then propagate the accepted value by proposing it again but with a higher proposal number. This way all correct acceptors accept the same value.

4.3.3 Termination

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

The global type and well-typedness ensure the absence of deadlocks. Thus, processes either loop forever or terminate. Assuming a failure detector in $\diamond\mathcal{S}$, eventually any correct proposer is not suspected by any correct acceptor and does not suspect any correct acceptor. FP_{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 FP_{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.



5 Conclusion

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.