Analyzing Paxos with Fault-Tolerant Multiparty Session Types

Bachelor thesis by Nicolas Daniel Torres
Date of submission: January 17, 2022

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department
<institute>
<working group>

Contents

1 Introduction

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

To achieve consensus, consensus algorithms must satisfy the following properties: termination, validity, and agreement [7]. 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 [17].

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 [12].

2 Technical Preliminaries

In this chapter we will introduce the Paxos consensus algorithm, Multiparty Session Types (MPST), Fault-Tolerant Multiparty Session Types (FTMPST), and some notation.

2.1 Paxos

Lamport describes the Paxos algorithm in [12]. First, he outlines the problem and then the algorithm to solve that problem. The following text is from [12] and includes most of chapter 2. The title of each section was left unchanged.

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:

- Only a value that has been proposed may be chosen,
- Only a single value is chosen, and
- A process never learns that a value has been chosen unless it actually has been.

We won't try to specify precise liveness requirements. However, the goal is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value.

We let the three roles in the consensus algorithm be performed by three classes of agents: proposers, acceptors, and learners. In an implementation, a single process may act as more than one agent, but the mapping from agents to processes does not concern us here.

Assume that agents can communicate with one another by sending mes- sages. 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.

2.1.2 Choosing a Value

The easiest way to choose a value is to have a single acceptor agent. A proposer sends a proposal to the acceptor, who chooses the first proposed value that it receives. Although simple, this solution is unsatisfactory because the failure of the acceptor makes any further progress impossible.

So, let's try another way of choosing a value. Instead of a single acceptor, let's use multiple acceptor agents. A proposer sends a proposed value to a set of acceptors. An acceptor may *accept* the proposed value. The value is chosen when a large enough set of acceptors have accepted it. How large is large enough? To ensure that only a single value is chosen, we can let a large enough set consist of any majority of the agents. Because any two majorities have at least one acceptor in common, this works if an acceptor can accept at most one value. (There is an obvious generalization of a majority that has been observed in numerous papers, apparently starting with [13].)

In the absence of failure or message loss, we want a value to be chosen even if only one value is proposed by a single proposer. This suggests the requirement:

P1 *An acceptor must accept the first proposal that it receives.*

But this requirement raises a problem. Several values could be proposed by different proposers at about the same time, leading to a situation in which every acceptor has accepted a value, but no single value is accepted by a majority of them. Even with just two proposed values, if each is accepted by about half the acceptors, failure of a single acceptor could make it impossible to learn which of the values was chosen.

P1 and the requirement that a value is chosen only when it is accepted by a majority of acceptors imply that an acceptor must be allowed to accept more than one proposal. We keep track of the different proposals that an acceptor may accept by assigning a (natural) number to each proposal, so a proposal consists of a proposal number and a value. To prevent confusion, we require that different proposals have different numbers. How this is achieved depends on the implementation, so for now we just assume it. A value is chosen when a single proposal with that value has been accepted by a majority of the acceptors. In that case, we say that the proposal (as well as its value) has been chosen.

We can allow multiple proposals to be chosen, but we must guarantee that all chosen proposals have the same value. By induction on the proposal number, it suffices to guarantee:

P2 *If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.*

Since numbers are totally ordered, condition P2 guarantees the crucial safety property that only a single value is chosen.

To be chosen, a proposal must be accepted by at least one acceptor. So, we can satisfy P2 by satisfying

$P2^a$ *If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v.*

We still maintain P1 to ensure that some proposal is chosen. Because communication is asynchronous, a proposal could be chosen with some particular acceptor c never having received any proposal. Suppose a new proposer "wakes up" and issues a higher-numbered proposal with a different value. P1 requires c to accept this proposal, violating $P2^a$. Maintaining both P1 and $P2^a$ requires strengthening $P2^a$ to:

$\mathrm{P2}^{\mathrm{b}}$ *If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.*

Since a proposal must be issued by a proposer before it can be accepted by an acceptor, $\mathrm{P2}^{\mathrm{b}}$ implies $\mathrm{P2}^{\mathrm{a}}$, which in turn implies $\mathrm{P2}$.

To discover how to satisfy $\mathrm{P2}^{\mathrm{b}}$, let's consider how we would prove that it holds. We would assume that some proposal with number m and value v is chosen and show that any proposal issued with number $n > m$ also has value v. We would make the proof easier by using induction on n, so we can prove that proposal number n has value v under the additional assumption that every proposal issued with a number in $m \dots (n-1)$ has value v, where $i \dots j$ denotes the set of numbers from i through j. For the proposal numbered m to be chosen, there must be some set C consisting of a majority of acceptors such that every acceptor in C accepted it. Combining this with the induction assumption, the hypothesis that m is chosen implies:

Every acceptor in C has accepted a proposal with number in $m \dots (n-1)$, and every proposal with number in $m \dots (n-1)$ accepted by any acceptor has value v.

Since any set S consisting of a majority of acceptors contains at least one member of C, we can conclude that a proposal numbered n has value v by ensuring that the following invariant is maintained:

$\mathrm{P2}^{\mathrm{c}}$ *For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S.*

We can therefore satisfy $\mathrm{P2}^{\mathrm{b}}$ by maintaining the invariance of $\mathrm{P2}^{\mathrm{c}}$.

To maintain the invariance of $\mathrm{P2}^{\mathrm{c}}$, a proposer that wants to issue a proposal numbered n must learn the highest-numbered proposal with number less than n, if any, that has been or will be accepted by each acceptor in some majority of acceptors. Learning about proposals already accepted is easy enough; predicting future acceptances is hard. Instead of trying to predict the future, the proposer controls it by extracting a promise that there won't be any such acceptances. In other words, the proposer requests that the acceptors not accept any more proposals numbered less than n. This leads to the following algorithm for issuing proposals.

1. A proposer chooses a new proposal number n and sends a request to each member of some set of acceptors, asking it to respond with:

   a) A promise never again to accept a proposal numbered less than n, and

   b) The proposal with the highest number less than n that it has accepted, if any.

   I will call such a request a *prepare* request with number n.

2. If the proposer receives the requested responses from a majority of the acceptors, then it can issue a proposal with number n and value v, where v is the value of the highest-numbered proposal among the responses, or is any value selected by the proposer if the responders reported no proposals.

A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted. (This need not be the same set of acceptors that responded to the initial requests.) Let's call this an *accept* request.

This describes a proposer's algorithm. What about an acceptor? It can receive two kinds of requests from proposers: *prepare* requests and *accept* requests. An acceptor can ignore any request without compromising safety. So, we need to say only when it is allowed to respond to a request. It can always respond to a *prepare* request. It can respond to an *accept* request, accepting the proposal, iff it has not promised not to. In other words:

P1^a *An acceptor can accept a proposal numbered n iff it has not responded to a* prepare *request having a number greater than n.*

Observe that P1^a subsumes P1.

We now have a complete algorithm for choosing a value that satisfies the required safety properties — assuming unique proposal numbers. The final algorithm is obtained by making one small optimization.

Suppose an acceptor receives a prepare request numbered n, but it has already responded to a prepare request numbered greater than n, thereby promising not to accept any new proposal numbered n. There is then no reason for the acceptor to respond to the new prepare request, since it will not accept the proposal numbered n that the proposer wants to issue. So we have the acceptor ignore such a prepare request. We also have it ignore a prepare request for a proposal it has already accepted.

With this optimization, an acceptor needs to remember only the highest-numbered proposal that it has ever accepted and the number of the highest-numbered prepare request to which it has responded. Because P2^c must be kept invariant regardless of failures, an acceptor must remember this information even if it fails and then restarts. Note that the proposer can always abandon a proposal and forget all about it — as long as it never tries to issue another proposal with the same number.

Putting the actions of the proposer and acceptor together, we see that the algorithm operates in the following two phases.

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. (Correctness is maintained, even though requests and/or responses for the proposal may arrive at their destinations long after the proposal was abandoned.) 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.

2.1.3 Progress

It's easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen. Proposer p completes phase 1 for a proposal number n_1. Another proposer q then completes phase 1 for a proposal number $n_2 > n_1$. Proposer p's phase 2 *accept* requests for a proposal numbered n_1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n_2. So, proposer p then begins and completes phase 1 for a new proposal number $n_3 > n_2$, causing the second phase 2 *accept* requests of proposer q to be ignored. And so on.

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 (proposer, acceptors, and communication network) is working properly, liveness can therefore be achieved by electing a single distinguished proposer. The famous result of Fischer, Lynch, and Patterson [8] implies that a reliable algorithm for electing a proposer must use either randomness or real time—for example, by using timeouts. However, safety is ensured regardless of the success or failure of the election.

2.1.4 The Implementation

The Paxos algorithm [14] assumes a network of processes. In its consensus algorithm, each process plays the role of proposer, acceptor, and learner. The algorithm chooses a leader, which plays the roles of the distinguished proposer and the distinguished learner. The Paxos consensus algorithm is precisely the one described above, where requests and responses are sent as ordinary messages. (Response messages are tagged with the corresponding proposal number to prevent confusion.) 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.

All that remains is to describe the mechanism for guaranteeing that no two proposals are ever issued with the same number. 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 Quorum

Lamport provided a definition of quorums in [11]. The definition relies on other concepts, which were also defined in [11]. The following includes the relevant assumptions and definitions from that paper.

Assumption (Agent)**.** The set of acceptors is finite, and there are at least two proposers and two learners.

Assumption (Value)**.** There are at least two proposable values.

We let n be the number of acceptors, and we define an *agent* to be a proposer, a learner, or an acceptor.

We begin by defining what an event is. We assume that the events performed by a single agent are totally ordered. (Events performed concurrently by a single agent can be ordered arbitrarily.) So an event e specifies an agent e_{agent} and a positive integer e_{num}, indicating that e is the e_{num}^{th} event performed by e_{agent}. An event can be performed either spontaneously or upon receipt of a message. For a message-receiving event e, we let e_{rcvd} be a triple $\langle m, a, i \rangle$, indicating that the event was triggered by the receipt of a message m sent by the i^{th} event of agent a. For simplicity, we assume that each event e sends exactly one message e_{msg}, which can be received by any agent (including itself). The sending of a possibly empty set $\mathcal{M}$ of messages can be modeled by letting e_{msg} equal $\mathcal{M}$ and having an event that receives $\mathcal{M}$ ignore any of its elements not meant for the receiver. Since we are concerned with when learning occurs and not with termination, we don't care if an agent ever stops sending messages.

For any set S of events, we define the precedence relation $\preceq_S$ on S to be the transitive closure of the relation $\rightarrow$ such that $d \rightarrow e$ iff either (i) $d_{agent} = e_{agent}$ and $d_{num} \leq e_{num}$ or (ii) e is a message-receiving event such that $e_{rcvd} = \langle d_{msg}, d_{agent}, d_{num} \rangle$. A scenario is then defined as follows.

Definition (Scenario)**.** A *scenario* S is a set of events such that

- For any agent a, the set of events in S performed by a consists of k_a events numbered from 1 through k_a, for some natural number k_a.
- For every message-receiving event e in S, there exists an event d in S different from e such that $e_{rcvd} = \langle d_{msg}, d_{agent}, d_{num} \rangle$.
- $\preceq_S$ is a partial order on S.

A *prefix* of a scenario T consists of a set of events in T that precede all other events in T. The precise definition is:

Definition (Prefix)**.** A subset S of a scenario T is a *prefix* of T, written $S \sqsubseteq T$, iff for any events d in T and e in S, if $d \preceq_T e$ then d is in S.

It is easy to see that any prefix of a scenario is also a scenario.

An *algorithm* is defined to be any non-empty set of scenarios. Let $\text{Agents}(S)$ be the set of all agents that perform events in the set S of events, and $A \setminus B$ is the subset of the set A consisting of all elements not in the set B.

Definition (Asynchronous Algorithm)**.** An *asynchronous algorithm* Alg is a set of scenarios such that:

A 1. Every prefix of a scenario in Alg is in Alg.

A 2. If T and U are scenarios of Alg and S is a prefix of both T and U such that $\mathrm{Agents}\,(T \setminus S)$ and $\mathrm{Agents}\,(U \setminus S)$ are disjoint sets, then $T \cup U$ is a scenario of Alg.

We now assume that there are proposing and learning events. A proposing event e is one in which proposer e_{agent} proposes a value $e_{proposed}$. A learning event e is one in which learner e_{agent} learns a value $e_{learned}$. Nontriviality and consistency are defined by:

Definition (Nontriviality)**.** An algorithm Alg is *nontrivial* iff, for every scenario S in Alg and any learning event e in S, there is a proposing event d in S with $d_{proposed} = e_{learned}$.

Definition (Consistency)**.** An algorithm Alg is *consistent* iff, for every scenario S in Alg, if d and e are learning events in S, then $d_{learned} = e_{learned}$.

We define a *consensus algorithm* to be an algorithm that is nontrivial and consistent.

A quorum is a set of acceptors that is large enough to choose a value, regardless of what steps of the algorithm have been performed so far. Here, we define an *accepting set* to be one large enough that it can choose a value starting *ab initio*. A quorum is therefore an accepting set.

Definition (Accepting Set)**.** A set Q of acceptors is *accepting* for a proposer p in algorithm Alg iff for every value v and learner l, there is a scenario S of Alg with $\mathrm{Agents}\,(S) \subseteq \{p, l\} \cup Q$ such that S has an event in which p proposes v and an event in which l learns v.

Definition (Quorum)**.** A set Q of acceptors is a *quorum* for an algorithm Alg iff it is an accepting set in Alg for every proposer and, for every proposer p, learner l, and scenario S of Alg, there exists a scenario T of Alg such that (i) S is a prefix of T, (ii) $\mathrm{Agents}\,(T \setminus S) \subseteq \{p, l\} \cup Q$, and (iii) T contains a learning event of l.

2.3 Multiparty Session Types

Multiparty Session Types (MPST) are used to statically ensure correctly coordinated behavior in systems without global control ([10, 6]). One important such property is progress, i.e., the absence of deadlock. Like with every other static typing approach, the main advantage is their efficiency, i.e., they avoid the problem of state space explosion.

MPST are designed to abstractly capture the structure of communication protocols. They describe global behaviors as *sessions*, i.e., units of conversations [10, 2, 3]. The participants of such sessions are called *roles*. *Global types* specify protocols from a global point of view. These types are used to reason about processes formulated in a *session calculus*. Most of the existing session calculi are variants of the π-calculus [15].

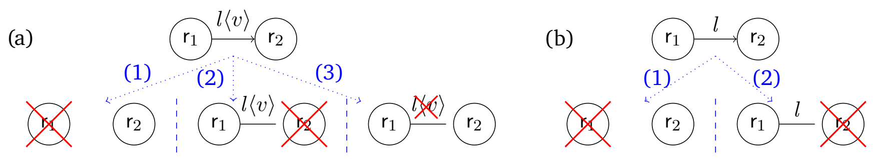

Figure 2.1: Unreliable Communication (a) and Weakly Reliable Branching (b).

2.4 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 [16]. The following explanation of Fault-Tolerant Multiparty Session Types is from that same paper. We included sections 2, 3, 4, and 5. The title of each section was left unchanged.

2.4.1 Fault-Tolerance in Distributed Algorithms

We consider three sources of failure in an unreliable communication (Fig. 2.1(a)): (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. The design of a distributed algorithm may allow it to handle some kinds of failures better than others. Failures are unpredictable events that occur at runtime. 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.

A fault-tolerant algorithm has to solve its task despite such failures. Remember that MPST analyse the communication structure. Accordingly, we need a mechanism to tolerate faults in the communication structure. We want our type system to ensure that a faulty interaction neither blocks the overall protocol nor influences the communication structure of the system after this fault. We consider an unreliable communication as fault-tolerant if a failure does not influence the guarantees that our type system provides for the overall communication structure except for this particular communication. Moreover, if a potentially unreliable communication is executed successfully, then our type system ensures the same guarantees as for reliable communication such as e.g. the absence of communication mismatches.

To ensure that a failure does not block the algorithm, both the receiver and the sender need to be allowed to proceed without their unreliable communication partner. Therefore, the receiver of an unreliable communication is required to specify a default value that, in the case of failure, is used instead of the value the process was supposed to receive. The type system ensures the existence of such default values and checks their sort. Moreover, we augment unreliable communication with labels that help us to avoid communication mismatches. This is enough to ensure that the communication structure of a distributed algorithm is fault-tolerant.

Branching in the context of failures is more difficult, because a branch marks a decision point in a specification, i.e., the participants of the session are supposed to behave differently w.r.t. this decision. In an unreliable setting it is difficult to ensure that all participants are informed consistently about such a decision and adapt their behaviour accordingly.

Consider a reliable branching that is decided by a process r_1 and transmitted to r_2. If we try to execute such a branching despite failures, we observe that there are again three ways in that this branching can go wrong (Fig. 2.1(b)): (1) The sender may crash before it releases its decision. This will block r_2, because it is missing the information about the branch it should move to. (2) The receiver might crash. (3) The message of r_1 about the decided branch is lost. Then again r_2 is blocked.

Case (2) can be dealt with similar to unreliable communication, i.e., by marking the branching as potentially faulty and by ensuring that a crash of r_2 will not block another process. For Case (1) we declare one of the offered branches as default to that r_2 moves if r_1 has crashed. Then r_2 will not necessarily move to the branch that r_1 had in mind before it crashed, but to a valid/specified branch and, since r_1 is crashed, no two processes move to different branches. The main problem is in Case (3). Let r_1 move to a non-default branch and transmit its decision to r_2, this message gets lost, and r_2 moves to the default branch. Now both processes did move to branches that are described by their types; but they are in different branches. Accordingly, this case violates the specification in the type, and we want to reject it. More precisely, 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. In this case, it is difficult for the type system to ensure interesting properties in branching.

2.4.2 Fault-Tolerant Types and Processes

We assume that the sets $\mathcal{N}$ of names $a, s, x \ldots$; $\mathcal{R}$ of roles $\mathsf{n}, \mathsf{r}, \ldots$; $\mathcal{L}$ of labels $l, l_{\mathrm{d}}, \ldots$; $\mathcal{V}_{\mathrm{T}}$ of type variables t; and $\mathcal{V}_{\mathrm{P}}$ of process variables X are pairwise distinct. For clarity, we often distinguish names into *values*, i.e., the payload of messages, *shared channels*, or *session channels* according to their usage; there is, however, no need to formally distinguish between different kinds of names. To simplify the reduction semantics of our session calculus, we use natural numbers as roles (compare to [10]). Sorts S range over $\mathbb{B}, \mathbb{N}, \ldots$. The set $\mathcal{E}$ of expressions $e, v, b, \ldots$ is constructed from the standard Boolean operations, natural numbers, names, and (in)equalities.

Global types specify the desired communication structure of systems from a global point of view. In local types this global view is projected to the specification of a single role/participant. We use standard MPST ([9, 10]) extended by operators for unreliable communication and weakly reliable branching that are highlighted in blue colour in Fig. 2.2.

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

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

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

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

Figure 2.2: Syntax of Fault-Tolerant MPST

The type $\mathsf{r}_1 \to_{\mathrm{u}} \mathsf{r}_2{:}l\langle \mathrm{S}\rangle.G$ specifies an unreliable communication from r_1 to r_2 transmitting (if successful) a label l and a value of type S and then continues (regardless of the success of this communication) with G. The unreliable counterparts of senders and receivers are $s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{u}}l\langle e\rangle.P_1$ (specified as $[\mathsf{r}_2]!_{\mathrm{u}}l\langle \mathrm{S}\rangle.T_1$) and $s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{u}}l\langle v\rangle(x).P_2$ (specified as $[\mathsf{r}_1]?_{\mathrm{u}}l\langle \mathrm{S}\rangle.T_2$). The receiver $s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{u}}l\langle v\rangle(x).P_2$ declares a default value v that is used instead of a received value to instantiate x after a failure. Moreover, a label is communicated that helps us to ensure that a faulty unreliable communication has no influence on later actions.

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

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

The $\perp$ denotes a process that crashed. Similar to [10], we use message queues to implement asynchrony in sessions. Therefore, session initialisation introduces a directed and initially empty message queue $s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}[\,]$ for each pair of roles $\mathsf{r}_1 \neq \mathsf{r}_2$ of the session s. The separate message queues ensure that messages with different sources or destinations are not ordered, but each message queue is FIFO. Since strongly reliable, weakly reliable, and unreliable forms of interaction might be implemented differently (e.g. by TCP or UDP), it make sense to further split the message queues into three message queues for each pair $\mathsf{r}_1 \neq \mathsf{r}_2$ such that different kinds of messages do not need to be ordered. To simplify the presentation of examples in this paper and not to blow up the number of message queues, we stick to a single message queue for each pair $\mathsf{r}_1 \neq \mathsf{r}_2$, but the correctness of our type system does not depend on this decision. We have five kinds of messages and corresponding message types in Fig. 2.2—one for each kind of interaction.

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

Consider the specification $G_{\text{dice,r}}$ of a simple dice game in a bar

$$(\mu t)3 \to_{\text{r}} 1{:}\langle \mathbb{N}\rangle.3 \to_{\text{r}} 2{:}\langle \mathbb{N}\rangle.3 \to_{\text{r}} 1{:}\{\mathit{roll}.3 \to_{\text{r}} 2{:}\mathit{roll}.t,\ \mathit{exit}.3 \to_{\text{r}} 2{:}\mathit{exit}.\texttt{end}\} \tag{1}$$

where the dealer Role 3 continues to roll a dice and tell its value to player 1 and then to roll another time for player 2 until the dealer decides to exit the game.

We can combine strongly reliable communication/branching and unreliable communication, e.g. by ordering a drink before each round in $G_{\text{dice,r}}$.

$$\begin{aligned}&(\mu t)3 \to_{\text{u}} 4{:}\mathit{drink}\langle \mathbb{N}\rangle.3 \to_{\text{r}} 1{:}\langle \mathbb{N}\rangle.3 \to_{\text{r}} 2{:}\langle \mathbb{N}\rangle.\\&3 \to_{\text{r}} 1{:}\{\mathit{roll}.3 \to_{\text{r}} 2{:}\mathit{roll}.t,\quad \mathit{exit}.3 \to_{\text{r}} 2{:}\mathit{exit}.\texttt{end}\}\end{aligned}$$

where role 4 represents the bar tender and the noise of the bar may swallow these orders. Moreover, we can remove the branching and specify a variant of the dice game in that 3 keeps on rolling the dice forever, but, e.g. due to a bar fight, one of our three players might get knocked out at some point or the noise of this fight might swallow the announcements of role 3:

$$G_{\text{dice,u}} = (\mu t)3 \to_{\text{u}} 1{:}\mathit{roll}\langle \mathbb{N}\rangle.3 \to_{\text{u}} 2{:}\mathit{roll}\langle \mathbb{N}\rangle.t \tag{2}$$

To restore the branching despite the bar fight that causes failures, we need the weakly reliable branching mechanism.

$$\begin{aligned}G_{\text{dice}} = (\mu t)3 \to_{\text{w}} \{1, 2\}:\ &\mathit{play}.3 \to_{\text{u}} 1{:}\mathit{roll}\langle \mathbb{N}\rangle.3 \to_{\text{u}} 2{:}\mathit{roll}\langle \mathbb{N}\rangle.t,\\ \oplus\ &\mathit{end}.3 \to_{\text{u}} 1{:}\mathit{win}\langle \mathbb{B}\rangle.3 \to_{\text{u}} 2{:}\mathit{win}\langle \mathbb{B}\rangle.\texttt{end}\end{aligned} \tag{3}$$

If 3 is knocked out by the fight, i.e., crashes, the game cannot continue. Then 1 and 2 move to the default branch end, have to skip the respective unreliable communications, and terminate. But the game can continue as long as 3 and at least one of the players $1, 2$ participate.

An implementation of G_{dice} is $P_{\text{dice}} = P_3 \mid P_1 \mid P_2$, where for $\mathsf{i} \in \{1, 2\}$:

$$\begin{aligned}P_3 = \overline{a}[3](s).(\mu X)&\texttt{if}\ x_1 \leq 21 \wedge x_2 \leq 21\\ &\quad\texttt{then}\ s[3, \{1, 2\}]!_{\text{w}}\mathit{play}.s[3, 1]!_{\text{u}}\mathit{roll}\langle \mathsf{roll}(x_1)\rangle.s[3, 2]!_{\text{u}}\mathit{roll}\langle \mathsf{roll}(x_2)\rangle.X\\ &\quad\texttt{else}\ s[3, \{1, 2\}]!_{\text{w}}\mathit{end}.s[3, 1]!_{\text{u}}\mathit{win}\langle x_1 \leq 21\rangle.s[3, 2]!_{\text{u}}\mathit{win}\langle x_2 \leq 21\rangle.\mathbf{0}\\ P_{\mathsf{i}} = a[\mathsf{i}](s).(\mu X)&s[\mathsf{i}, 3]?_{\text{w}}\mathit{play}.s[\mathsf{i}, 3]?_{\text{u}}\mathit{roll}\langle x\rangle(x).X \oplus \mathit{end}.s[\mathsf{i}, 3]?_{\text{u}}\mathit{win}\langle \texttt{f}\rangle(w).\mathbf{0}\end{aligned}$$

Role 3 stores the sums of former dice rolls for the two players in its local variables x_1 and x_2, and $\mathsf{roll}(x_i)$ rolls a dice and adds its value to the respective x_i. Role 3 keeps rolling dice until the sum x_i for one of the players exceeds 21. If both sums x_1 and x_2 exceed 21 in the same round, then 3 wins, i.e., both players receive `f`; else, the player that stayed below 21 wins and receives `t`. The players 1 and 2 use their respective last known sum that is stored in x as default value for the unreliable communication in the branch $play$ and `f` as default value in the branch end. The last branch, i.e., end, is the default branch.

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 [10] we use a more relaxed variant of projection as introduced in [20].

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

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

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

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

where either $\diamond = \mathrm{r}$, $\mathfrak{R} = \mathsf{r}_2$, $\mathfrak{B}$ is $\mathsf{p} = \mathsf{r}_2$, $\mathfrak{D}$ is empty or $\diamond = \mathrm{w}$, $\mathfrak{R} = \mathsf{R}$, $\mathfrak{B}$ is $\mathsf{p} \in \mathsf{R}$, $\mathfrak{D}$ is $, l_{\mathrm{d}}$. In the last case of strongly reliable or weakly reliable branching—when projecting onto a role that does not participate in this branching—we map to $\bigsqcup_{i\in\mathrm{I}}(G_i\restriction_{\mathsf{p}}) = (G_1\restriction_{\mathsf{p}}) \sqcup \ldots \sqcup (G_n\restriction_{\mathsf{p}})$. The operation $\sqcup$ is (similar to [20]) inductively defined as:

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

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

The remaining global types are projected as follows:

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

$$t\restriction_{\mathsf{p}} \triangleq t \qquad \mathsf{end}\restriction_{\mathsf{p}} \triangleq \mathsf{end}$$

The projection of $G_1 \parallel G_2$ on p is **not** defined if p occurs on both sides of this parallel composition; it is $G_i{\upharpoonright_{\mathsf{p}}}$ if p occurs in exactly one side $i \in \{1, 2\}$; or it is $(G_1 \parallel G_2){\upharpoonright_{\mathsf{p}}} = G_1{\upharpoonright_{\mathsf{p}}} = G_2{\upharpoonright_{\mathsf{p}}} = \mathtt{end}$ if p does not occur at all. Projecting a recursive global type results in a recursive local type if p occurs in the body of the recursion or else in successful termination. Type variables and successful termination are mapped onto themselves. We denote a global type G as *projectable* if for all $\mathsf{r} \in \mathrm{R}(G)$ the projection $G{\upharpoonright_{\mathsf{r}}}$ is defined. We restrict our attention to projectable global types.

Projecting the global type $G_{\text{dice,r}}$ in (1) results in the local types

$$\begin{aligned}
T_{3:\text{dice,r}} &= (\mu t)[1]!_{\mathrm{r}}\langle\mathbb{N}\rangle.[2]!_{\mathrm{r}}\langle\mathbb{N}\rangle.[1]!_{\mathrm{r}}\{\mathit{roll}.[2]!_{\mathrm{r}}\mathit{roll}.t, \quad \mathit{exit}.[2]!_{\mathrm{r}}\mathit{exit}.\mathtt{end}\}\\
T_{\mathrm{i}:\text{dice,r}} &= (\mu t)[3]?_{\mathrm{r}}\langle\mathbb{N}\rangle.[3]?_{\mathrm{r}}\{\mathit{roll}.t, \quad \mathit{exit}.\mathtt{end}\}
\end{aligned}$$

where the types of the two players $T_{1:\text{dice,r}} = T_{2:\text{dice,r}} = T_{\mathrm{i}:\text{dice,r}}$ are identical. The projection of the outer branching in $G_{\text{dice,r}}$ on 2 results in $[3]?_{\mathrm{r}}\mathit{roll}.t$ for the first branch and $[3]?_{\mathrm{r}}\mathit{exit}.\mathtt{end}$ for the second branch. These two $[3]?_{\mathrm{r}}$ types are unified by $\sqcup$ into a single $[3]?_{\mathrm{r}}$ type with two branches.

Projection maps G_{dice} in (3) to:

$$\begin{aligned}
T_{3:\text{dice}} &= (\mu t)[\{1, 2\}]!_{\mathrm{w}} \ \mathit{play}.[1]!_{\mathrm{u}}\mathit{roll}\langle\mathbb{N}\rangle.[2]!_{\mathrm{u}}\mathit{roll}\langle\mathbb{N}\rangle.t\\
&\qquad\qquad\qquad\quad \oplus \mathit{end}.[1]!_{\mathrm{u}}\mathit{win}\langle\mathbb{B}\rangle.[2]!_{\mathrm{u}}\mathit{win}\langle\mathbb{B}\rangle.\mathtt{end}\\
T_{\mathrm{i}:\text{dice}} &= (\mu t)[3]?_{\mathrm{w}}(\mathit{play}.[3]?_{\mathrm{u}}\mathit{roll}\langle\mathbb{N}\rangle.t \oplus \mathit{end}.[3]?_{\mathrm{u}}\mathit{win}\langle\mathbb{B}\rangle.\mathtt{end})
\end{aligned}$$

where $\mathsf{i} \in \{1, 2\}$ and both $T_{\mathrm{i}:\text{dice}}$ are obtained by the second case of projection. The type system will ensure that either 3 transmits the request to branch to both players 1, 2 simultaneously and, since these messages cannot be lost, all players that are not crashed move to same branch or 3 crashes and all remaining players move to the default branch.

Assume instead that 3 can only inform one of the players $1, 2$ at once. The type

$$\begin{aligned}
(\mu t)3 \rightarrow_{\mathrm{w}} \{1\}: \ &\mathit{play}.3 \rightarrow_{\mathrm{u}} 1{:}\mathit{roll}\langle\mathbb{N}\rangle.3 \rightarrow_{\mathrm{u}} 2{:}\mathit{roll}\langle\mathbb{N}\rangle.t\\
\oplus \ &\mathit{end}.3 \rightarrow_{\mathrm{u}} 1{:}\mathit{win}\langle\mathbb{B}\rangle.3 \rightarrow_{\mathrm{u}} 2{:}\mathit{win}\langle\mathbb{B}\rangle.\mathtt{end}
\end{aligned}$$

is not projectable, because $\sqcup$ does not allow to unify the projections $[3]?_{\mathrm{u}}\mathit{roll}\langle\mathbb{N}\rangle.t$ and $[3]?_{\mathrm{u}}\mathit{win}\langle\mathbb{B}\rangle.\mathtt{end}$ of the two branches of 2. Replacing the two unreliable communications with 2 by strongly reliable communications implies that neither 3 nor 2 fail. The type

$$\begin{aligned}
&(\mu t)3 \rightarrow_{\mathrm{w}} \{1\}:\\
&\qquad \mathit{play}.3 \rightarrow_{\mathrm{u}} 1{:}\mathit{roll}\langle\mathbb{N}\rangle.3 \rightarrow_{\mathrm{w}} \{2\}:(\mathit{play}.3 \rightarrow_{\mathrm{u}} 2{:}\mathit{roll}\langle\mathbb{N}\rangle.t \oplus \mathit{end}.\mathtt{end})\\
&\qquad \oplus \mathit{end}.3 \rightarrow_{\mathrm{u}} 1{:}\mathit{win}\langle\mathbb{B}\rangle.3 \rightarrow_{\mathrm{w}} \{2\}:(\mathit{play}.\mathtt{end} \oplus \mathit{end}.3 \rightarrow_{\mathrm{u}} 2{:}\mathit{win}\langle\mathbb{B}\rangle.\mathtt{end})
\end{aligned}$$

where 3 informs its two players subsequently about the chosen branch is projectable. But it introduces the two additional branches $\mathit{end}.\mathtt{end}$ and $\mathit{play}.\mathtt{end}$, i.e., 3 is allowed to choose the branches for the players $1, 2$ separately and differently, whereas in (1) as well as in (3) the players $1, 2$ are always in the same branch. Because of that, we allow for broadcast in weakly reliable branching such that 3 can inform both players consistently without introducing additional and not-intended branches.

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

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

We write $\mathrm{nsr}(G)$, $\mathrm{nsr}(T)$, and $\mathrm{nsr}(P)$, if none of the prefixes in G, T, and P is strongly reliable or for delegation and if P does not contain message queues. Let $\mathrm{R}(G)$ return all roles that occur in G. A global type is *well-formed* if (1) it neither contains free nor unguarded type variables, (2) $\mathrm{R}(G) = \{1, \ldots, |\mathrm{R}(G)|\}$, (3) for all its subterms of the form $\mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\langle \mathrm{S}\rangle.G$ and $\mathsf{r}_1 \to_{\mathrm{u}} \mathsf{r}_2{:}l\langle \mathrm{S}\rangle.G$, we have $\mathsf{r}_1 \neq \mathsf{r}_2$, (4) for all its subterms of the form $\mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\{l_i.G_i\}_{i\in\mathrm{I}}$ and $\mathsf{r} \to_{\mathrm{w}} \mathsf{R}{:}\{l_i.G_i\}_{i\in\mathrm{I},l_{\mathrm{d}}}$, we have $\mathsf{r}_1 \neq \mathsf{r}_2$, $\mathsf{r} \notin \mathsf{R}$, $\mathrm{d} \in \mathrm{I}$, and the labels l_i are pairwise distinct, and (5) for all its subterms of the form $G_1 \mid\mid G_2$, we have $\mathrm{R}(G_1) \cap \mathrm{R}(G_2) = \emptyset$. We restrict our attention to well-formed global types for that projection is defined on all its roles.

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

2.4.3 A Semantics with Failure Patterns

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).

We use labels for two purposes: they allow us to distinguish between different branches, as usual in MPST-frameworks, and we assume that they may carry additional runtime information such as timestamps. Of course, the presented type system remains valid if we use labels without additional information. In contrast to standard MPST (as e.g. in [10]) and to support unreliable communication, 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. If runtime information are irrelevant, $\doteq$ can be instantiated with equality. We require that $\doteq$ is unambiguous on labels used in types, i.e., given two labels of processes l_P, l'_P and two labels of types l_T, l'_T then $l_P \doteq l'_P \wedge l_P \doteq l_T \Rightarrow l'_P \doteq l_T$ and $l_P \doteq l_T \wedge l_T \not\doteq l'_T \Rightarrow l_P \not\doteq l'_T$.

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:

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

The reduction semantics of the session calculus is defined in Fig. 2.3 and Fig. 2.4, where we follow [10]: we assume that session initialisation is synchronous and communication within a session is asynchronous implemented using message queues.

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

$$\begin{array}{lll}
\text{(Init)} & \overline{a}[\mathsf{n}](s).P_{\mathsf{n}} \mid \prod_{1 \leq \mathsf{i} \leq \mathsf{n}-1} a[\mathsf{i}](s).P_{\mathsf{i}} \longmapsto (\nu s)\left(\prod_{1 \leq \mathsf{i} \leq \mathsf{n}} P_{\mathsf{i}} \mid \prod_{1 \leq \mathsf{i},\mathsf{j} \leq \mathsf{n}, \mathsf{i} \neq \mathsf{j}} s_{\mathsf{i} \to \mathsf{j}}{:}[\,]\right) & \\
 & & \text{if } a \neq s \\
\text{(RSend)} & s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{r}}\langle y \rangle.P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\#\langle v \rangle^{\mathrm{r}} & \text{if } \operatorname{eval}(y) = v \\
\text{(RGet)} & s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{r}}(x).P \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\langle v \rangle^{\mathrm{r}}\#\mathrm{M} \longmapsto P\{{}^{v}\!/\!_{x}\} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{M} & \\
\text{(USend)} & s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{u}} l\langle y \rangle.P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\#l\langle v \rangle^{\mathrm{u}} & \text{if } \operatorname{eval}(y) = v \\
\text{(UGet)} & s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{u}} l\langle dv \rangle(x).P \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}l'\langle v \rangle^{\mathrm{u}}\#\mathrm{M} \longmapsto P\{{}^{v}\!/\!_{x}\} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{M} & \\
 & & \text{if } l \doteq l',\ \mathtt{FP}_{\mathtt{uget}}(s, \mathsf{r}_1, \mathsf{r}_2, l') \\
\text{(USkip)} & s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{u}} l\langle dv \rangle(x).P \longmapsto P\{{}^{dv}\!/\!_{x}\} & \text{if } \mathtt{FP}_{\mathtt{uskip}}(s, \mathsf{r}_1, \mathsf{r}_2, l) \\
\text{(ML)} & s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l\langle v \rangle^{\mathrm{u}}\#\mathrm{M} \longmapsto s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} & \text{if } \mathtt{FP}_{\mathtt{ml}}(s, \mathsf{r}_1, \mathsf{r}_2, l) \\
\text{(RSel)} & s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{r}} l.P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\#l^{\mathrm{r}} & \\
\text{(RBran)} & s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{r}}\{l_i.P_i\}_{i \in \mathrm{I}} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}l^{\mathrm{r}}\#\mathrm{M} \longmapsto P_j \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{M} & \text{if } l \doteq l_j,\ j \in \mathrm{I} \\
\text{(WSel)} & s[\mathsf{r}, \mathsf{R}]!_{\mathrm{w}} l.P \mid \prod_{\mathsf{r}_i \in \mathsf{R}} s_{\mathsf{r} \to \mathsf{r}_i}{:}\mathrm{M}_i \longmapsto P \mid \prod_{\mathsf{r}_i \in \mathsf{R}} s_{\mathsf{r} \to \mathsf{r}_i}{:}\mathrm{M}_i\#l^{\mathrm{w}} & \\
\text{(WBran)} & s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{w}}\{l_i.P_i\}_{i \in \mathrm{I}, l_{\mathrm{d}}} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}l^{\mathrm{w}}\#\mathrm{M} \longmapsto P_j \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{M} & \text{if } l \doteq l_j,\ j \in \mathrm{I} \\
\text{(WSkip)} & s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{w}}\{l_i.P_i\}_{i \in \mathrm{I}, l_{\mathrm{d}}} \longmapsto P_{\mathrm{d}} & \text{if } \mathtt{FP}_{\mathtt{wskip}}(s, \mathsf{r}_1, \mathsf{r}_2) \\
\text{(Crash)} & P \longmapsto \perp & \text{if } \mathtt{FP}_{\mathtt{crash}}(P)
\end{array}$$

Figure 2.3: Reduction Rules ($\longmapsto$) of Fault-Tolerant Processes (Part 1).

Rule (RSend) implements an asynchronous strongly reliable message transmission. As a result the value $\operatorname{eval}(y)$ 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 $\mathtt{FP}_{\mathtt{uget}}$ to decide whether this step is allowed. This failure pattern could, e.g., be used to reject messages that are too old. The Rule (USkip) allows to skip the reception of a message in an unreliable communication using a failure pattern $\mathtt{FP}_{\mathtt{uskip}}$ and instead substitutes the bound variable x in the continuation with the default value dv. The failure pattern $\mathtt{FP}_{\mathtt{uskip}}$ tells us whether a reception can be skipped (e.g. via failure detector).

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

The Rules (Crash) for *crash failures* and (ML) for *message loss*, describe failures of a system. With Rule (Crash) P can crash if $\mathtt{FP}_{\mathtt{crash}}$, where $\mathtt{FP}_{\mathtt{crash}}$ can e.g. model immortal processes or global bounds on the number of crashes. (ML) allows to drop an unreliable message if the failure pattern $\mathtt{FP}_{\mathtt{ml}}$ is valid. $\mathtt{FP}_{\mathtt{ml}}$ allows, e.g., to implement safe channels that never lose messages or a global bound on the number of lost messages.

The remaining rules for conditionals, session delegation, parallel composition, restriction, recursion, and structural congruence in Fig. 2.4 are standard.

We augmented our reduction semantics in Fig. 2.3 by five different failure patterns that we deliberately do not specify, although we usually assume that the failure patterns $\mathtt{FP}_{\mathtt{uget}}$, $\mathtt{FP}_{\mathtt{uskip}}$, and $\mathtt{FP}_{\mathtt{wskip}}$ use only local

$$
\begin{array}{lll}
\text{(If-T)} & \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P' \longmapsto P & \text{if } e \text{ is true} \\
\text{(If-F)} & \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P' \longmapsto P' & \text{if } e \text{ is false} \\
\text{(Deleg)} & s[\mathsf{r}_1, \mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}] \rangle\!\rangle.P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\#s'[\mathsf{r}] & \\
\text{(SRecv)} & s[\mathsf{r}_1, \mathsf{r}_2]?(\!(s'[\mathsf{r}])\!).P \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}s''[\mathsf{r}']\#\mathrm{M} \longmapsto P\{^{s''}/_{s'}\}\{^{\mathsf{r}'}/_{\mathsf{r}}\} \mid \mathrm{M} & \\
\text{(Par)} & P_1 \mid P_2 \longmapsto P_1' \mid P_2 & \text{if } P_1 \longmapsto P_1' \\
\text{(Res)} & (\nu x)P \longmapsto (\nu x)P' & \text{if } P \longmapsto P' \\
\text{(Rec)} & (\mu X)P \longmapsto P\{^{(\mu X)P}/_{X}\} & \\
\text{(Struc)} & P_1 \longmapsto P_1' & \text{if } P_1 \equiv P_2,\ P_2 \longmapsto P_2',\ P_2' \equiv P_1'
\end{array}
$$

Figure 2.4: Reduction Rules ($\longmapsto$) of Fault-Tolerant Processes (Part 2).

information, whereas $\mathtt{FP_{ml}}$ and $\mathtt{FP_{crash}}$ may use global information of the system in the current run. We provide these predicates to allow for the implementation of system requirements or abstractions like failure detectors that are typical for distributed algorithms. Directly including them in the semantics has the advantage that all traces satisfy the corresponding requirements, i.e., all traces are valid w.r.t. the assumed system requirements. An example for the instantiation of these patterns is given implicitly via the Conditions 1.1–1.6 in Section 2.4.4 and explicitly in Section **??**. If we instantiate the patterns $\mathtt{FP_{uget}}$ with true and the patterns $\mathtt{FP_{uskip}}$, $\mathtt{FP_{wskip}}$, $\mathtt{FP_{crash}}$, $\mathtt{FP_{ml}}$ with false, then we obtain a system without failures. In contrast, the instantiation of all five patterns with true results in a system where failures can happen completely non-deterministically at any time.

One of the properties that our type system has to ensure even in the case of failures is the absence of communication mismatches, i.e., the type of a transmitted value has to be the type that the receiver expects. The global type $1 \to_{\mathrm{u}} 2{:}l_1\langle\mathbb{N}\rangle.1 \to_{\mathrm{u}} 2{:}l_2\langle\mathbb{B}\rangle.\mathtt{end}$ specifies two subsequent unreliable communications in that values of different sorts are transmitted. If the first message with its natural number is lost but the second message containing a Boolean value is transmitted, 2 could wrongly receive a Boolean value although it still waits for a natural number. To avoid this mismatch, we add a label to unreliable communication and ensure (by the typing rules) that the same label is never associated with different types. Similarly, labels are used in [4] to avoid communication errors. Accordingly, we require $l_1 \neq l_2$ such that the reduction rules in Fig. 2.3 will not allow to consume the Boolean message before 2 has reduced its first prefix.

We do that, because we think of labels not only as identifiers for branching but also as some kind of meta data of messages as they can be often found in communication media or as they are assumed by many distributed algorithms. Our unreliable communication mechanism exploits such meta data to guarantee strong properties about the communication structure including the described absence of communication mismatches.

Note that we keep the failure patterns abstract and do not model how to check them in producing runs. Indeed system requirements such as bounds on the number of processes that can crash usually cannot be checked, but result from observations, i.e., system designers ensure that a violation of this bound is very unlikely and algorithm designers are willing to ignore these unlikely events. In particular, $\mathtt{FP_{ml}}$ and $\mathtt{FP_{crash}}$ are thus often implemented as oracles for verification, whereas e.g. $\mathtt{FP_{uskip}}$ and $\mathtt{FP_{wskip}}$ are often implemented by system specific time-outs. Note that we are talking about implementing these failure patterns and not formalising them. Failure patterns are abstractions of real world system requirements or software. We implement them by conditions providing the necessary guarantees that we need in general (i.e., for subject reduction and progress) or for the verification of concrete algorithms. In practise, we expect that the systems on that the verified algorithms are running satisfy the respective conditions. Accordingly, the session channels, roles, labels, and processes mentioned in Fig. 2.3 are not parameters of the failure patterns, but just a vehicle to more formally specify the conditions on failure patterns in Section 2.4.4.

Similarly, strongly reliable and weakly reliable interactions in potentially faulty systems are abstractions.

They are usually implemented by handshakes and redundancy; replicated servers against crash failures and retransmission of late messages against message loss. Algorithm designers have to be aware of the additional costs of these interactions.

Consider the implementation of $G_{\text{dice,u}}$ in (2), i.e., an infinite variant of the dice game, where the players 1 and 2 use their respective last known sum x_i of former dice rolls as default values:

$$\begin{aligned} P_{\text{dice,u}} &= P_{3,\text{u}} \mid P_{\text{i,u}} \mid P_{2,\text{u}} \\ P_{3,\text{u}} &= \overline{a}[3](s).(\mu X)s[3,1]!_{\text{u}}roll\langle \mathsf{roll}(x_1)\rangle.s[3,2]!_{\text{u}}roll\langle \mathsf{roll}(x_2)\rangle.X \\ P_{\text{i,u}} &= a[\text{i}](s).(\mu X)s[\text{i},3]?_{\text{u}}roll\langle x_\text{i}\rangle(x_\text{i}).X \end{aligned}$$

An unreliable communication in a global type specifies a communication that, due to system failures, may or may not happen. Moreover, regardless of the successful completion of this unreliable communication, the future behaviour of a well-typed system will follow its specification in the global type. Since the players 1 and 2 repeat the same kind of unreliable action, they may lose track of the current round. If they successfully receive a new sum of dice rolls from 3 they cannot be sure on how often 3 actually did roll the dice. Because of lost messages, they may have missed some former announcements of 3 and, because of their ability to skip the reception of messages, they may have proceeded to the next round before 3 rolled a dice. Because the information about the current round is irrelevant for the communication structure in this case, there is no need to enforce round information.

2.4.4 Typing Fault-Tolerant Processes

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

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

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

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

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

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. 2.5 and 2.6. We concentrate on the interaction cases, where we observe that all new cases are quite similar to their strongly reliable counterparts.

Rule (RSend) checks strongly reliable senders, i.e., requires a matching strongly reliable sending in the local type of the actor and compares the actor with this type. With $\Gamma \Vdash y{:}\mathrm{S}$ we check that y is an expression of the sort S if all names x in y are replaced by arbitrary values of sort S_x for $x{:}\mathrm{S}_x \in \Gamma$. Then the continuation of the

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.5: Typing Rules for Fault-Tolerant Systems.

$$(\mathsf{Res2})\ \frac{\begin{array}{c}\{s[\mathsf{r}]{:}G\!\restriction_{\mathsf{r}} \mid \mathsf{r} \in \mathrm{R}(G)\} \cdot \{s_{\mathsf{r}\to\mathsf{r}'}{:}[\,] \mid \mathsf{r},\mathsf{r}' \in \mathrm{R}(G') \wedge \mathsf{r} \neq \mathsf{r}'\} \overset{s}{\Rightarrow} \Delta' \\ s\sharp(\Gamma,\Delta) \quad a{:}G \in \Gamma \quad \Gamma \vdash P \triangleright \Delta \cdot \Delta'\end{array}}{\Gamma \vdash (\nu s)P \triangleright \Delta}$$

$$(\mathsf{MQComR})\ \frac{\Gamma \Vdash v{:}\mathrm{S} \quad \Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\langle v\rangle^{\mathrm{r}}\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\langle \mathrm{S}\rangle^{\mathrm{r}}\#\mathrm{MT}}$$

$$(\mathsf{MQComU})\ \frac{\Gamma \Vdash v{:}\mathrm{S} \quad l \doteq l' \quad l'{:}\mathrm{S} \in \Gamma \quad \Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l\langle v\rangle^{\mathrm{u}}\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l'\langle \mathrm{S}\rangle^{\mathrm{u}}\#\mathrm{MT}}$$

$$(\mathsf{MQBranR})\ \frac{l \doteq l' \quad \Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l^{\mathrm{r}}\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l'^{\,\mathrm{r}}\#\mathrm{MT}}$$

$$(\mathsf{MQBranW})\ \frac{l \doteq l' \quad \Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l^{\mathrm{w}}\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l'^{\,\mathrm{w}}\#\mathrm{MT}}$$

$$(\mathsf{MQDeleg})\ \frac{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}s'[\mathsf{r}]\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}s'[\mathsf{r}]\#\mathrm{MT}} \qquad (\mathsf{MQNil})\ \frac{}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}[\,] \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}[\,]}$$

Figure 2.6: Runtime Typing Rules for Fault-Tolerant Systems.

process is checked against the continuation of the type. The unreliable case is very similar, but additionally checks that the label is assigned to the sort of the expression in Γ. Rule (RGet) type strongly reliable receivers, where again the prefix is checked against a corresponding type prefix and the assumption $x{:}\mathrm{S}$ is added to the type check of the continuation. Again the unreliable case is very similar, but apart from the label also checks the sort of the default value.

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

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

Figure 2.6 presents the runtime typing rules, i.e., the typing rules for processes that may result from steps of a system that implements a global type. Since it covers only operators that are not part of initial systems, a type checking tool might ignore them. We need these rules however for the proofs of progress and subject reduction. Under the assumption that initial systems cannot contain crashed processes, Rule (Crash) may be moved to the set of runtime typing rules.

We have to prove that our extended type system satisfies the standard properties of MPST, i.e., subject reduction and progress. *Subject reduction* tells us that derivatives of well-typed systems are again well-typed. This is fundamental, since it ensures that our formalism can be used to analyse processes by static type checking. We extend subject reduction such that it provides some information on how the session environment evolves alongside reductions of the system. Therefore we introduce a reduction relation $\overset{s}{\mapsto}$ on session environments, that emulates the reduction steps of processes. As an example consider the rule $\Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{r}}\langle \mathrm{S}\rangle.T \cdot s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]{:}T \cdot s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}\#\langle \mathrm{S}\rangle^{\mathrm{r}}$ that emulates the transfer of a value in (RSend). Let $\overset{s}{\Rightarrow}$ denote the reflexive and transitive closure of $\overset{s}{\mapsto}$.

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.

Definition 1 (Coherence)**.** *The type environments* Γ, Δ *are* coherent *if, for all session channels* s *in* Δ*, there exists a global type* G *in* Γ *such that the restriction of* Δ *on assignments with* s *is the set* Δ' *such that* $\{s[\mathsf{r}]{:}G\upharpoonright_{\mathsf{r}} \mid \mathsf{r} \in \mathrm{R}(G)\} \cdot \{s_{\mathsf{r}\to\mathsf{r}'}{:}[\,] \mid \mathsf{r}, \mathsf{r}' \in \mathrm{R}(G)\} \overset{s}{\mapsto}{}^{*} \Delta'$.

Because of the failure pattern in the reduction semantics in Fig. 2.3, subject reduction and progress do not hold in general. Instead we have to fix conditions on failure patterns that ensure these properties. Subjection reduction needs one condition on crashed processes and progress requires that the instantiation of the failure patterns is such that they do not block parts of the system. In fact, different instantiations of these failure patterns may allow for progress. We leave it to further work to determine what kind of conditions on failure patterns or requirements on their interactions are necessary to prove these properties. Here, we consider only one such set of conditions.

Condition 1 (Failure Pattern)**.**

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

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

The combination of these 6 conditions might appear quite restrictive on a first glance. For example the combination of the Conditions 1.4 and 1.6 ensures the correct behaviour of weakly reliable branching such that branching messages can be skipped if and only if the respective sender has crashed. An implementation of such a weakly reliable interaction in an asynchronous system that is subject to message losses and process crashes, might require something like a perfect failure detector or actually solving consensus[1]. Here it is important to remember that these conditions are minimal assumptions on the system requirements and that system requirements are just abstractions. Parts of them may be realised by actual software-code (which

[1]Note that the example we present in Section **??** is a consensus algorithm. So, if the Conditions 1 require a solution of consensus, an example on top of that solving consensus would be pointless.

then allows to check them), whereas other parts of the system requirements may not be realised at all but rather observed (which then does not allow to very them). Weakly reliable branching is a good example of this case. The easiest way to obtain a weakly reliable interaction, is by using a handshake communication and time-outs. If the sender time-outs while waiting for an acknowledgement, it resends the message. If the sender does not hear from its receiver for a long enough period of time, it assumes that the receiver has crashed and proceeds. With carefully chosen time-frames for the time-outs, this approach is a compromise between correctness and efficiency. In a theoretical sense, it is clearly not correct. There is no time-frame such that the sender can be really sure that the receiver has crashed. From a practical point of view, this is not so problematic, since in many systems failures are very unlikely. If failures that are so severe that they are not captured by the time-outs are extremely unlikely, then it is often much more efficient to just accept that the algorithm is not correct in these cases. Trying to obtain an algorithm that is always correct might be impossible or at least usually results into very inefficient algorithms. Moreover, verifying this requires to also verify the underlying communication infrastructure and the way in that failures may occur, which is impossible or at least impracticable. Because of that, it is an established method to verify the correctness of algorithms w.r.t. given system requirements (e.g. in [5, 12, 18]), even if these system requirements are not verified and often do not hold in all (but only nearly all) cases.

With Conditions 1, we can now analyse our fault-tolerant type system.

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

The proof is by induction on the derivation of $P \longmapsto P'$. In every case, we use the information about the structure of the processes to generate partial proof trees for the respective typing judgement. Additionally, we use Condition 1.1 to ensure that the type environment of a crashed process cannot contain the types of reliable communication prefixes.

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, 10]. 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 this sense fault-tolerance is more flexible than explicit failure handling as e.g. [19] has to exclude interleaved sessions.

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 initialisation. Note that without action prefixes all participants either terminated or crashed.

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

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

2.5 Additional Notation

2.5.1 Prefixes and Branches

In [16] the authors introduced notation for the construction of global types and processes.

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

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

2.5.2 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 $f : X \rightarrow \emptyset$ be a function with domain $X = X_1 \times \ldots \times X_n$ and $P \in \mathcal{P}$ a process. Then $f\left(x_1, \ldots, x_n\right).P$ is also a process in $\mathcal{P}$.

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

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

$$(\mathrm{Func})\ \frac{f \in F_{X,\emptyset} \qquad \Gamma \Vdash \left(x_1, \ldots, x_n\right) : \mathbb{X} \qquad \Gamma \vdash P \triangleright \Delta}{\Gamma \vdash f\left(x_1, \ldots, x_n\right).P \triangleright \Delta}$$

In (Func) we require that f is a function from X to $\emptyset$ and expression $(x_1, \ldots, x_n)$ is of the sort 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 proofs in [16].

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.

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

Second, we assume a set of values Value.

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

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

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

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

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

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

$\text{Promise}\ a$ has two possible forms. $\text{Promise}\ \text{Maybe}\ \text{Proposal}\ a$ and $\text{Nack}\ \mathbb{N}$. $\text{Promise}\ \text{Maybe}\ \text{Proposal}\ a$ is the same as $\text{Promise}\ c$ where $c = \text{Maybe}\ b$ and $b = \text{Proposal}\ a$. Possible values include $\text{Nack}\ 1$ and $\text{Promise}\ \text{Just}\ \text{Proposal}\ 1\ {-1}$ of type $\text{Promise}\ \mathbb{Z}$. The actual type of $\text{Nack}\ 1$, 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 $\text{Promise}\ \text{Value}$. 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 $\text{Promise}\ \text{Maybe}\ \text{Proposal}\ a$. In the second case it includes the highest n that acceptor promised and is of the form $\text{Nack}\ \mathbb{N}$.

3.2 Global Type

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

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

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

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

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

3.3 Functions

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

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

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

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

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

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

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

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.

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

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

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

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

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

$\text{promiseCount}\,(ps)$ takes a list of promises ps and calculates the number of promises in that list of 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 $2a$.

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

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

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

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

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

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

$\mathrm{genA_Q}\,(p, c_A, c_P)$ returns a randomly selected set A_Q with $A_Q \subseteq A = \{1, \ldots, c_A\}$ and $|A_Q| > \frac{|A|}{2}$. A_Q consists of any majority of acceptors. In Paxos a majority of acceptors forms a quorum, i.e. an accepting set with which a value can be chosen [11]. We use this function when initiating the proposers to give them a quorum of acceptors with which they communicate.

3.4 Processes

3.4.1 System Initialization

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

$$\begin{aligned}&\mathrm{P^P_{init}}\left(p, A_Q, n, m, \vec{V}\right) = \overline{b_n}\,[i]\,(s)\,.\,\mathrm{P^P}\\&\mathrm{P^A_{init}}\,(a, p, c_A, c_P, n, pr) = \Pi_{c_A < k \le c_A + c_P}\ b_k\,[a]\,(s)\,.\,\mathrm{P^A_1}\end{aligned}$$

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

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

$\mathrm{P^P_{init}}\left(p, A_Q, n, m, \vec{V}\right)$ is initialized with the proposer's role in its own session p, which is always $c_A + 1$, a quorum of acceptors A_Q, an index n, and a vector $\vec{V}$. Each proposer has the same role $p = c_A + 1$ but uses a different shared-point b_n according to its index n. m is initialized to the same value as n but is never updated. $\vec{V}$ is used in the proposer to collect and evaluate the responses from the acceptors. It is always initialized with an empty list $[]$. Shared-point b_n is used to initiate a session. Afterwards, the process behaves like $\mathrm{P^P}$. We assume a mechanism for electing a distinguished proposer, which acts as the leader [12]. The leader is the only proposer that can try issuing proposers. A new leader is elected via the same mechanism when the previous leader terminates or crashes.

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

3.4.2 Proposer

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

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

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

3.4.3 Acceptor

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

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

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

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

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

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

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

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

3.5 Failure Patterns

Chandra and Toueg introduce a class of failure detectors $\Diamond\mathscr{S}$, which is called *eventually strong* in [5]. Failure detectors in $\Diamond\mathscr{S}$ satisfy the following properties: (1) eventually every process that crashes is permanently suspected by every correct process and (2) eventually some correct process is never suspected by any correct process.

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

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

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

For Paxos to work a majority of acceptors needs to be alive. That means that the number of failed acceptors f needs to satisfy $n > 2f$ where n is the total number of acceptors, except in one case where there are 2 acceptors. Then, at most one acceptor may crash [11]. For acceptors $\texttt{FP}_{\texttt{crash}}$ returns true if, and only if, at least one more acceptor may crash, i.e. $n > 2(f + 1)$ is satisfied. Let $\mathbb{F}$ be the set of processes permanently suspected by a failure detector in $\Diamond\mathscr{S}$. For proposers $\texttt{FP}_{\texttt{crash}}$ returns true if $A_Q \setminus \mathbb{F}$ is not a quorum, i.e. if the set of acceptors in A_Q that are not permanently suspected is not enough to form a majority of acceptors.

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

3.6 Example

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

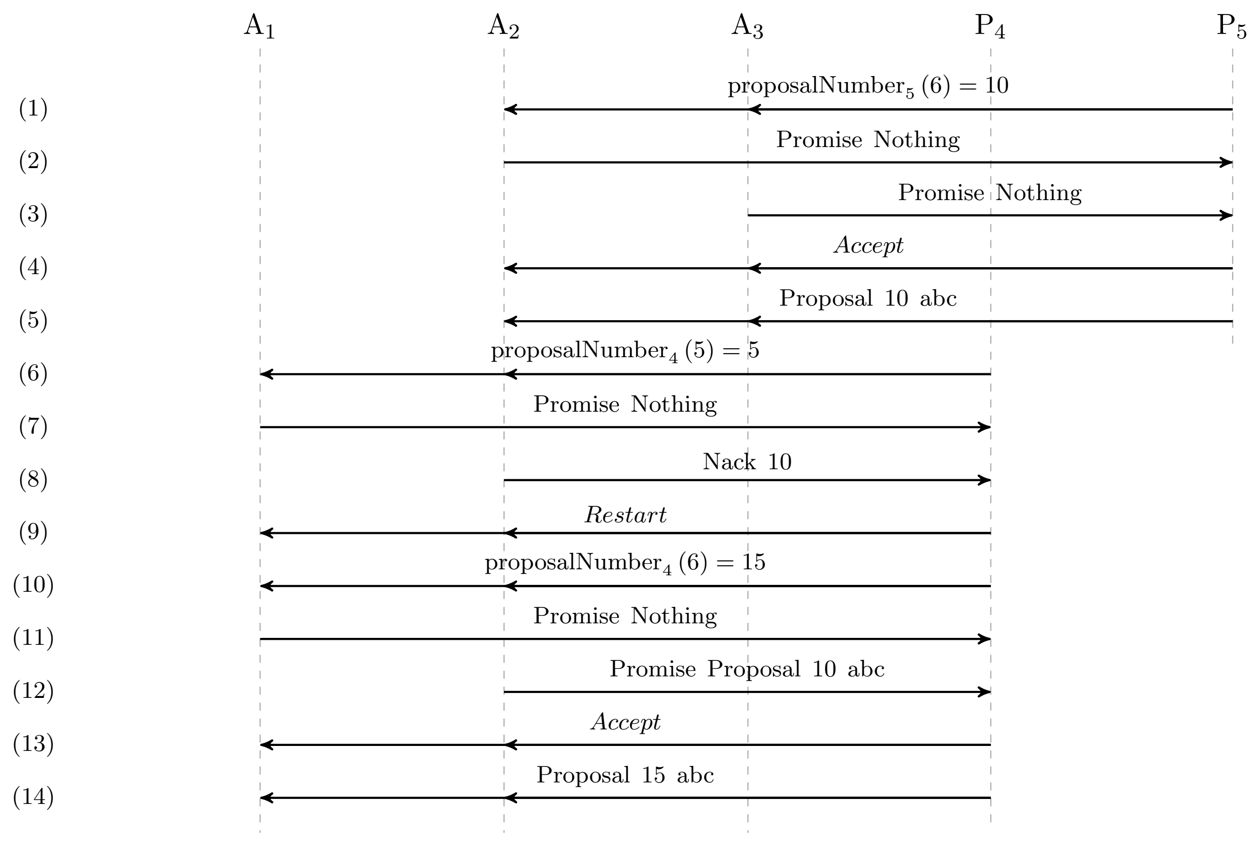

Figure 3.1: Example scenario with 3 acceptors and 2 proposers.

3.6.1 Scenario

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

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

P_4 then runs through the Paxos algorithm with A_1 and A_2 starting with a new prepare request (10) with a higher proposal number. In step (12) P_4 learns that value abc with proposal number 10 has already been accepted by A_2. Later, in step (14), P_4 issues a proposal with the value of the highest-numbered proposal that it receives as a response to its prepare request. In this case there is only one such proposal, which is $\mathrm{Proposal}\ 10\ \mathrm{abc}$.

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

3.6.2 Formulae

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

System Initialization

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

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

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

$$\begin{aligned}
&\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,(\\
&(\mu X)\,\mathrm{update}\,(n,5)\,.\left(\textstyle\bigodot_{a\in\{1,2\}}\ s\,[4,a]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(n)\rangle\right)\ldots\ {=\mathrm{P}_4}\\
&\mid\ (\mu X)\,\mathrm{update}\,(n,6)\,.\left(\textstyle\bigodot_{a\in\{2,3\}}\ r\,[4,a]!_{u}l1a\,\langle \mathrm{proposalNumber}_{5}\,(n)\rangle\right)\ldots\ {=\mathrm{P}_5}\\
&\mid\ \big((\mu X)\,s\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,r\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\big)\ {=\mathrm{A}_1}\\
&\mid\ \big((\mu X)\,s\,[4,2]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,r\,[4,2]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\big)\ {=\mathrm{A}_2}\\
&\mid\ \big((\mu X)\,s\,[4,3]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,r\,[4,3]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\big)\ {=\mathrm{A}_3}\\
&\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l}:[\,]\ \mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l}:[\,]\ \mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l}:[\,])
\end{aligned}$$

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 $\mathrm{proposalNumber}_5\,(6) = 10$ to A_2 and A_3.

$$\begin{aligned}
&\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,(\\
&(\mu X)\,s\,[4,1]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(5)\rangle\,.s\,[4,2]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(5)\rangle\ldots\ {=\mathrm{P}_4}\\
&\mid\ (\mu X)\,r\,[2,4]?_{u}l1b\,\langle\bot\rangle\,(v_2)\,.r\,[3,4]?_{u}l1b\,\langle\bot\rangle\,(v_3)\ldots\ {=\mathrm{P}_5}\\
&\mid\ \big((\mu X)\,s\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,r\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\big)\ {=\mathrm{A}_1}\\
&\mid\ \big((\mu X)\,s\,[4,2]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,\mathrm{if}\ 10=\bot\ \mathrm{then}\ s\,[2,4]!_{u}l1b\,\langle\bot\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\ \mathrm{else}\ldots\big)\ {=\mathrm{A}_2}\\
&\mid\ \big((\mu X)\,s\,[4,3]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,\mathrm{if}\ 10=\bot\ \mathrm{then}\ s\,[3,4]!_{u}l1b\,\langle\bot\rangle\,.\,\mathrm{P}_2^{\mathrm{A}}\ \mathrm{else}\ldots\big)\ {=\mathrm{A}_3}\\
&\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l}:[\,]\ \mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l}:[\,]\ \mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l}:[\,])
\end{aligned}$$

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

$$\begin{aligned}
&\longmapsto^{*} (\nu t)\,(\nu s)\,(\nu r)\,(\\
&(\mu X)\,s\,[4,1]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(5)\rangle\,.s\,[4,2]!_{u}l1a\,\langle \mathrm{proposalNumber}_{4}\,(5)\rangle\ldots\ {=\mathrm{P}_4}\\
&\mid\ (\mu X)\,r\,[2,4]?_{u}l1b\,\langle\bot\rangle\,(v_2)\,.r\,[3,4]?_{u}l1b\,\langle\bot\rangle\,(v_3)\ldots\ {=\mathrm{P}_5}\\
&\mid\ \big((\mu X)\,s\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,r\,[4,1]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\big)\ {=\mathrm{A}_1}\\
&\mid\ \big((\mu X)\,s\,[4,2]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,\mathrm{if}\ \mathrm{gt}\,(10,\mathrm{Nothing})\ \mathrm{then}\ldots\mathrm{else}\ldots\big)\ {=\mathrm{A}_2}\\
&\mid\ \big((\mu X)\,s\,[4,3]?_{u}l1a\,\langle\bot\rangle\,(n')\,.\,\mathrm{if}\ldots\ \mid\ (\mu X)\,\mathrm{if}\ \mathrm{gt}\,(10,\mathrm{Nothing})\ \mathrm{then}\ldots\mathrm{else}\ldots\big)\ {=\mathrm{A}_3}\\
&\mid \Pi_{1\le k,l\le 4,k\ne l}\ s_{k\to l}:[\,]\ \mid \Pi_{1\le k,l\le 4,k\ne l}\ r_{k\to l}:[\,]\ \mid \Pi_{1\le k,l\le 2,k\ne l}\ t_{k\to l}:[\,])
\end{aligned}$$

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

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

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. All three acceptors move into $\mathrm{P}_2^{\mathrm{A}}$.

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

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

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

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.

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

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

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

We can apply (If-T) to A_3 and (If-F) to A_1 and A_2. A_3 moves directly to $\mathrm{P}^{\mathrm{A}}_2$ whereas A_1 and A_2 send their responses to P_4 before moving to $\mathrm{P}^{\mathrm{A}}_2$. A_1 also updates its local variable $n = 5$ via (Func).

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

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

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

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).

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

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:

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

Note that, at this point, A_1 and A_2 have updated their respective 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:

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

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

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

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

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

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

4 Analysis

We take the model from the previous chapter, type-check it, and discuss what the type check means for agreement, validity, and termination of the Paxos algorithm. To execute the type check we project the global type to local types and use the typing rules given in [16] to prove that our model is well-typed.

4.1 Local Types

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

For $1 \le a \le c_A$ and $c_A + 1 \le p \le c_A + c_P$ we define the projections of the global type $\mathrm{G_{p,A_Q}}$.

$$\mathrm{G_{p,A_Q}} \upharpoonright_p = (\mu x) \left(\bigodot_{a \in A_Q} \; [a]!_u l1a \,\langle \mathbb{N} \rangle \right) . \left(\bigodot_{a \in A_Q} \; [a]?_u l1b \,\langle \text{Promise Value} \rangle \right) .$$
$$\left([A_Q]!_w \mathit{Accept}. \left(\bigodot_{a \in A_Q} \; [a]!_u l2a \,\langle \text{Proposal Value} \rangle \right) .0 \oplus \mathit{Restart}.x \oplus \mathit{Abort}.0 \right)$$

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

$$\mathrm{G_{p,A_Q}} \upharpoonright_a = (\mu x) \, [p]?_u l1a \,\langle \mathbb{N} \rangle \, . \, [p]!_u l1b \,\langle \text{Promise Value} \rangle \, .$$
$$([p]?_w \mathit{Accept}. \, [p]?_u l2a \,\langle \text{Proposal Value} \rangle \, .0 \oplus \mathit{Restart}.x \oplus \mathit{Abort}.0)$$

$\mathrm{G_{p,A_Q}} \upharpoonright_a$ 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

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

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

We start the type-check with the global environment Γ and the entry-point of the model $\mathrm{Sys}\,(c_A, c_P)$. Then, we apply the typing rules in [16] in a proof tree and show that the model can be derived from the axioms (Var) and (End).

4.2.1 System Initialization

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

We apply (Par) twice and split off into three sub-proofs (S_1), (S_2), and (S_3).

$$(S_1) = \dfrac{\dfrac{(P)}{\Gamma \vdash \overline{b_{c_A+c_P}}\,[c_A+1]\,(s)\,.\,\mathrm{P}^{\mathrm{P}} \rhd t\,[2] : \mathrm{G}\restriction_2}}{\Gamma \vdash \overline{o}\,[2]\,(t)\,.\,\mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(c_A+1, \mathrm{genA}_\mathrm{Q}\,(c_A+c_P, c_A, c_P)\,, c_A+c_P, c_A+c_P, []) \rhd \emptyset}\,(\mathrm{Rec})$$

In (S_1) we apply (Rec) once and defer the rest of the proof-tree. Note that, since $\mathrm{G}\restriction_2 = 0$, $t\,[2] : \mathrm{G}\restriction_2 = \emptyset$. This is relevant later when continuing (P).

$$(S_2) = \dfrac{\dfrac{\dfrac{(P)}{\Gamma \vdash \overline{b_{c_A+1}}\,[c_A+1]\,(s)\,.\,\mathrm{P}^{\mathrm{P}} \rhd \emptyset} \quad \ldots \quad \dfrac{(P)}{\Gamma \vdash \overline{b_{c_A+c_P-1}}\,[c_A+1]\,(s)\,.\,\mathrm{P}^{\mathrm{P}} \rhd \emptyset}}{\Gamma \vdash \Pi_{c_A<k<c_A+c_P}\ \mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(c_A+1, \mathrm{genA}_\mathrm{Q}\,(k, c_A, c_P)\,, k, k, []) \rhd t\,[1] : G\restriction_1}\,(\mathrm{Par})^{c_P-1}}{\Gamma \vdash o\,[1]\,(t)\,.\Pi_{c_A<k<c_A+c_P}\ \mathrm{P}^{\mathrm{P}}_{\mathrm{init}}\,(\ldots) \rhd \emptyset}\,(\mathrm{Acc})$$

Applying (Acc) in (S_2) requires that $o : \mathrm{G} \in \Gamma$. (Par) is applied $c_P - 1$ times to separate all the proposer processes. Each individual proposer can be type-checked with the same proof-tree (P). Because $\mathrm{G}\restriction_1 = 0$, $t\,[1] : \mathrm{G}\restriction_1 = \emptyset$. The session environment Δ in (P) is empty for every proposer.

$$(S_3) = \dfrac{\dfrac{\dfrac{\dfrac{(A_1)}{\Gamma \vdash \mathrm{P}_1^{\mathrm{A}} \rhd s\,[a] : \mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}} \upharpoonright_a}}{\Gamma \vdash b_k\,[a]\,(s)\,.\,\mathrm{P}_1^{\mathrm{A}} \rhd \emptyset}\,(\mathrm{Acc}) \quad \dots}{\Gamma \vdash \Pi_{c_A < k \leq c_A + c_P}\ b_k\,[c_A]\,(s)\,.\,\mathrm{P}_1^{\mathrm{A}} \rhd \emptyset}\,(\mathrm{Par})^{c_P} \quad \dots}{\Gamma \vdash \Pi_{1 \leq j \leq c_A}\ \left(\Pi_{c_A < k \leq c_A + c_P}\ b_k\,[j]\,(s)\,.\,\mathrm{P}_1^{\mathrm{A}}\right) \rhd \emptyset}\,(\mathrm{Par})^{c_A}$$

(Par) is applied c_A times to separate the individual acceptors and then c_P times for each acceptor to separate the individual subprocesses. Since every subprocess of every acceptor behaves like $\mathrm{P}_1^{\mathrm{A}}$ and has the same local type, the same proof-tree (A_1) can be applied. Applying (Acc) to every subprocess of every acceptor requires $\forall k \in \mathbb{N} : (c_A + 1 \leq k \wedge k \leq c_A + c_P) \rightarrow b_k : \mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}} \in \Gamma$.

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

4.2.2 Proposer

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

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

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

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

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

$$e = \mathrm{anyNack}\left(\overrightarrow{V}\right)\ \mathrm{or}\ \mathrm{promiseCount}\left(\overrightarrow{V}\right) < \left\lceil \frac{p}{2} \right\rceil$$
$$pn = \mathrm{proposalNumber}_m\,(n)$$
$$\mathit{prop} = \mathrm{Proposal\ \ proposalNumber}_m\,(n)\ \ \mathrm{promiseValue}\left(\overrightarrow{V}\right)$$

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

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

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

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

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

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

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

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

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

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

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

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

4.2.3 Acceptor

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$(A_{update}) = \dfrac{\dfrac{\dfrac{}{\Gamma''' \vdash 0 \triangleright s[a]:0}(\text{End})}{\Gamma''' \vdash \text{update}(n, \text{Just nFromProposal}(pr')).0 \triangleright s[a]:0}(\text{Func})}{\Gamma''' \vdash \text{update}(pr, pr') \ldots . \triangleright s[a]:0}(\text{Func})$$

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

4.3 Termination, Agreement, Validity

4.3.1 Termination

The global type and well-typedness ensure the absence of deadlocks. This means that the processes either loop forever or terminate. Acceptors terminate if all of their sub-processes terminate. Each sub-process of an acceptor corresponds to one proposer. A sub-process can only terminate via the weakly reliable broadcast in P_2^{A}, which depends on the corresponding proposer. If that proposer crashes or its quorum does not include the acceptor, the sub-process terminates because $\texttt{FP}_{\texttt{wskip}}$ returns true and the default branch is *Abort*, which terminates immediately. The termination of a sub-process with a correct proposer requires the termination of that proposer. Thus, we need to prove that correct proposers terminate to prove termination for our model.

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

In [12] Lamport describes a scenario in which two proposers loop endlessly, never having their proposals accepted: Proposer p completes phase 1 for a proposal number n_1. Another proposer q then completes phase 1 for a proposal number $n_2 > n_1$. Proposer p's phase 2 accept requests for a proposal numbered n_1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n_2. So, proposer p then begins and completes phase 1 for a new proposal number $n_3 > n_2$, causing the second phase 2 accept requests of proposer q to be ignored. And so on.

From [12] we know that this problem is solved by electing a single distinguished proposer to be the leader. The leader eventually picks a proposal number high enough for its proposal to be accepted. The model assumes some sort of leader selection. A new leader is elected when the previous leader terminates.

4.3.2 Agreement

Any proposer p requires that the set of correct acceptors in its quorum of acceptors is itself a quorum, i.e. an accepting set with which a value can be chosen [11]. Should message loss occur in labels $l1a$ or $l1b$, p restarts the algorithm. This broadcast is weakly reliable and thus only fails when p crashes or terminates because $\mathtt{FP}_{\mathtt{wskip}}$ disallows suspicion of correct live proposers in acceptors. Given the definition of $\mathrm{promiseValue}$, p will only propose a fresh value if none of the acceptors have accepted a proposal yet. At least one acceptor in every other proposer's quorum is contained in p's quorum. Thus, if a majority of acceptors accept p's proposal it is sent to every other proposer when they reach phase $1b$. These proposers then propagate the accepted value by proposing it again but with a higher proposal number. This way all correct acceptors accept the same value.

4.3.3 Validity

To prove validity for our model we examine the communication structure and the origins of the accepted values. Because the model is well-typed we know the communication structure is as specified in global type $\mathrm{G}_{\mathrm{p},\mathrm{A}_\mathrm{Q}}$. From [16] we know that validity then holds globally if it holds for each local process.

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

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

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

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

Bibliography

[1] 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] L. Bocchi et al. “A Theory of Design-by-Contract for Distributed Multiparty Interactions”. In: *CONCUR 2010 - Concurrency Theory*. Ed. by P. Gastin and F. Laroussinie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 162–176. ISBN: 978-3-642-15375-4.

[4] L. Caires and H.T. Vieira. “Conversation types”. In: *Theoretical Computer Science* 411.51 (2010). European Symposium on Programming 2009, pp. 4399–4440. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/j.tcs.2010.09.010`. URL: `https://www.sciencedirect.com/science/article/pii/S0304397510004895`.

[5] 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`.

[6] M. Coppo et al. “A Gentle Introduction to Multiparty Asynchronous Session Types”. In: *Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*. Ed. by M. Bernardo and Einar B. Johnsen. Cham: Springer International Publishing, 2015, pp. 146–178. ISBN: 978-3-319-18941-3. DOI: `10.1007/978-3-319-18941-3_4`. URL: `https://doi.org/10.1007/978-3-319-18941-3_4`.

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

[8] M.J. Fischer, N.A. Lynch, and M.S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. ISSN: 0004-5411. DOI: `10.1145/3149.214121`. URL: `https://doi.org/10.1145/3149.214121`.

[9] 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`.

[10] 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`.

[11] 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.

[12] 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/.

[13] L. Lamport. “The Implementation of Reliable Distributed Multiprocess Systems”. In: *Computer Networks* 2 (Aug. 1978), pp. 95–114. URL: https://www.microsoft.com/en-us/research/publication/implementation-reliable-distributed-multiprocess-systems/.

[14] 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/.

[15] R. Milner, J. Parrow, and D. Walker. “A calculus of mobile processes, I”. In: *Information and Computation* 100.1 (1992), pp. 1–40. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90008-4. URL: https://www.sciencedirect.com/science/article/pii/0890540192900084.

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

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

[18] A.S. Tanenbaum and M. van Steen. *Distributed Systems: principles and paradigms*. Pearson Prentice Hall, 2017.

[19] M. Viering et al. “A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems”. In: *Programming Languages and Systems*. Ed. by A. Ahmed. Cham: Springer International Publishing, 2018, pp. 799–826. ISBN: 978-3-319-89884-1.

[20] 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.