

# Analyzing Paxos with Fault-Tolerant Multiparty Session Types

Bachelor thesis by Nicolas Daniel Torres  
Date of submission: December 30, 2021

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



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Computer Science  
Department

<institute>

<working group>

---

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Technical Preliminaries</b>	<b>4</b>
<b>3</b>	<b>Model</b>	<b>5</b>
3.1	Sorts . . . . .	5
3.2	Global Type . . . . .	6
3.3	Functions . . . . .	6
3.4	Processes . . . . .	8
3.4.1	System Initialization . . . . .	8
3.4.2	Proposer . . . . .	8
3.4.3	Acceptor . . . . .	9
3.5	Failure Patterns . . . . .	10
3.6	Example . . . . .	10
3.6.1	Scenario . . . . .	12
3.6.2	Formulae . . . . .	12
<b>4</b>	<b>Analysis</b>	<b>16</b>
4.1	Local Types . . . . .	16
4.2	Type Check . . . . .	16
4.2.1	System Initialization . . . . .	16
4.2.2	Proposer . . . . .	17
4.2.3	Acceptor . . . . .	18
<b>5</b>	<b>Discussion</b>	<b>21</b>

---

# 1 Introduction

---

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

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

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

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



---

## 2 Technical Preliminaries

---

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

---

## 3 Model

---

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

---

### 3.1 Sorts

---

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

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

`Bool = {true, 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  $a$ .

`Maybe a = Just a | Nothing`

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

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

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

`Promise a = Promise Maybe Proposal a | Nack  $\mathbb{N}$`

`Promise a` has two possible forms. `Promise Maybe Proposal a` and `Nack  $\mathbb{N}$` . Note that `Promise Maybe Proposal a` is the same as `Promise c` where  $c = \text{Maybe } b$  and  $b = \text{Proposal } a$ . Possible values include `Nack 1` and `Promise Just Proposal 1 -1` of type `Promise  $\mathbb{Z}$` . The actual type of `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 `Promise Value`. The prepare request contains a number  $n$ . The acceptors may respond to the prepare request with a promise to

not accept any proposal numbered less than  $n$  or with a rejection. In the first case the acceptor's response optionally includes the last proposal it accepted, if available, and is of the form *Promise Maybe Proposal  $a$* . In the second case it includes the highest  $n$  that acceptor promised and is of the form *Nack  $N$* .

---

## 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.

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

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

In the first two steps,  $1a$  and  $1b$ , the proposer sends its proposal number to each acceptor in  $A_Q$  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.

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

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

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

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

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

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

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

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

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

```
anyNack : list of Promise a → Bool
anyNack ([]) = false
anyNack ((Nack _#_)) = true
anyNack ((_#xs)) = anyNack (xs)
```

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

```
promiseCount : list of Promise a → ℕ
promiseCount ([]) = 0
promiseCount ((Promise _#xs)) = 1 + promiseCount (xs)
promiseCount ((_#xs)) = promiseCount (xs)
```

$\text{promiseCount}(ps)$  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.

```
gt : a → Maybe a → Bool
gt (_, Nothing) = true
gt (a, Just b) = a > b
```

```
ge : a → Maybe a → Bool
ge (_, Nothing) = true
ge (a, Just b) = a ≥ b
```

```
nFromProposal : Proposal a → ℕ
nFromProposal (Proposal n _) = n
```

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

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

```
genAQ : ℕ × ℕ × ℕ → ℕ
```

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

## 3.4 Processes

### 3.4.1 System Initialization

$$\begin{aligned} \text{Sys}(ac, pc) &= \bar{o}[2](t) \cdot P_{\text{init}}^P(ac+1, \text{genA}_Q(ac+pc, ac, pc), ac+pc, ac+pc, []) \\ &| \bar{o}[1](t) \cdot \Pi_{ac < k < ac+pc} P_{\text{init}}^P(ac+1, \text{genA}_Q(k, ac, pc), k, k, []) \\ &| \Pi_{1 \leq j \leq ac} P_{\text{init}}^A(j, ac+1, ac, pc, n_a, pr_a) \end{aligned}$$

$$P_{\text{init}}^P(p, A_Q, n, m, \vec{V}) = \bar{b}_n[i](s) \cdot P^P$$

$$P_{\text{init}}^A(a, p, ac, pc, n, pr) = \Pi_{ac < k \leq ac+pc} b_k[a](s) \cdot P_1^A$$

$\text{Sys}(ac, pc)$ ,  $P_{\text{init}}^P(p, A_Q, n, m, \vec{V})$ , and  $P_{\text{init}}^A(a, p, ac, pc, n, pr)$  describe the system initialization.  $ac$  and  $pc$  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  $ac$  and the proposers are initialized using indices from  $ac+1$  to  $ac+pc$ .

$P_{\text{init}}^P(p, A_Q, n, m, \vec{V})$  is initialized with the proposer's role in its own session  $p$ , which is always  $ac+1$ , a quorum of acceptors  $A_Q$ , an index  $n$ , and a vector  $\vec{V}$ . Each proposer has the same role  $p = ac+1$  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  $P^P$ .

$P_{\text{init}}^A(a, p, ac, pc, n, pr)$  is initialized with the acceptor's index  $a$ , the proposer index  $p$ , which is always  $ac+1$ ,  $ac$ ,  $pc$ , 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 Maybe  $\mathbb{N}$  and  $pr$  is of type Maybe (Proposal Value) thus both can be 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  $P_1^A$ .

### 3.4.2 Proposer

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

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



$$s[p, A_Q]!_w Accept. \\ \left( \bigodot_{a \in A_Q} s[p, a]!_u l2a \left\langle \text{Proposal } \text{proposalNumber}_m(n) \text{ promiseValue } (\vec{V}) \right\rangle \right).0$$

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  $\vec{V}$  through  $v_a$ . Because  $p = ac + 1$ , the minimum number of acceptors needed to form a majority is  $\left\lceil \frac{p}{2} \right\rceil = \left\lceil \frac{ac+1}{2} \right\rceil$ . If any Nack  $x$  was received or the number of 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

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

For each proposer an acceptor has a corresponding subprocess, which behaves like  $P_1^A$ . These subprocesses access the same values for  $n$  and  $pr$ . This means that updating these values with  $\text{update}(n, m)$  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 proposers' session but is not contained in the proposers' 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 Promise  $pr$  or 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  $\diamond\mathcal{W}$ , which is called *eventually weak* in [1]. Failure detectors in  $\diamond\mathcal{W}$  satisfy the following properties: (1) eventually every process that crashes is permanently suspected by some 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,  $FP_{\text{uskip}}$  is implemented with a failure detector in  $\diamond\mathcal{W}$  for phases 1a, 1b, and 2a.

Similarly, message loss is possible in all phases modeled in the global type. Thus,  $FP_{\text{ml}}$  is also implemented with a failure detector in  $\diamond\mathcal{W}$  with one exception.  $FP_{\text{ml}}(s, p, a, l)$  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  $FP_{\text{wskip}}$  returns true if, and only if, the corresponding proposer crashed or otherwise terminated.

For Paxos to work a majority of acceptors needs to be alive. That means that the number of failed acceptors  $f$  needs to satisfy  $n > 2f$  where  $n$  is the total number of acceptors, except in one case where there are 2 acceptors. Then, at most one acceptor may crash [3].  $FP_{\text{crash}}$  returns true if, and only if, at least one more acceptor may crash, i.e.  $n > 2(f + 1)$  is satisfied.

In Paxos there is no need to reject outdated messages so  $FP_{\text{uget}}$  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.

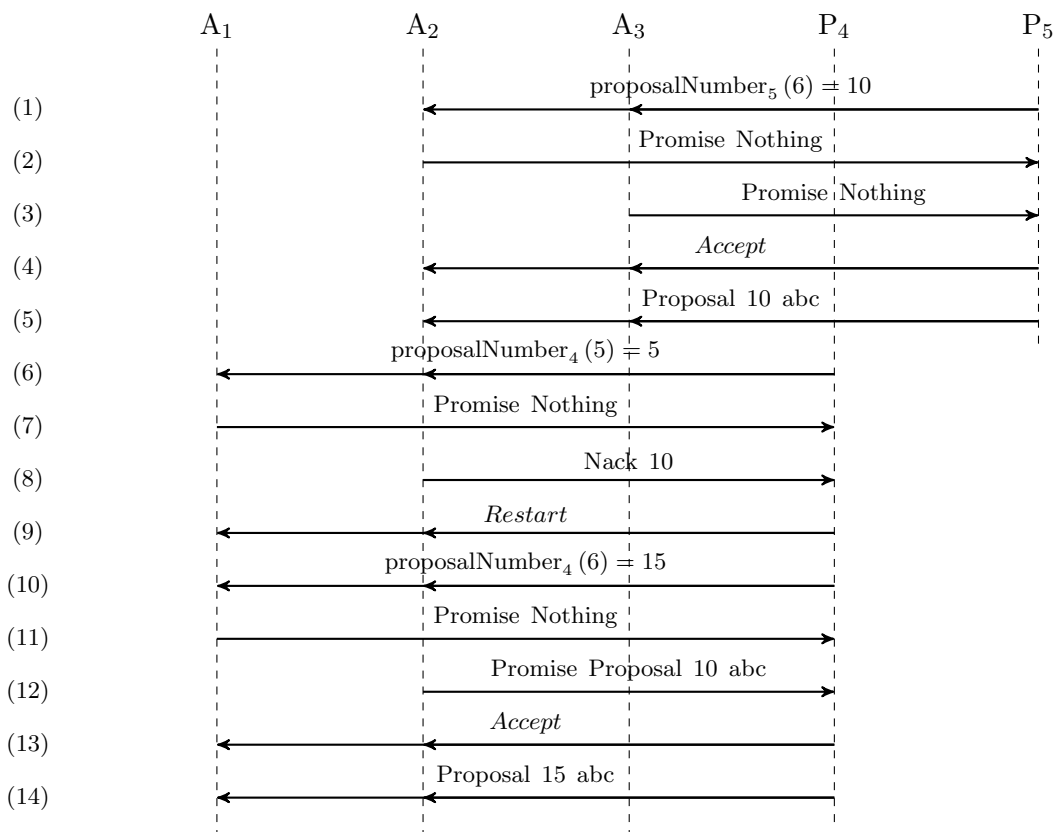


Figure 3.1: Example scenario with 3 acceptors and 2 proposers.

### 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. In steps (1) to (5),  $P_5$  completes the Paxos algorithm with  $A_2$  and  $A_3$  and terminates.

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

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

### 3.6.2 Formulae

We set  $ac = 3$ ,  $pc = 2$ , and  $V = \{abc, def, \dots, vwx, yz\}$ .

#### System Initialization

After inserting  $ac$  and  $pc$  and applying (Init) once for shared-point  $a$  we have:

$$\begin{aligned} & \text{Sys}(ac, pc) = \text{Sys}(3, 2) = o[1](t) . \Pi_{3 < k < 5} P_{\text{init}}^P(4, \text{genA}_Q(k, 3, 2), k, k, []) \\ & | \bar{o}[2](t) . P_{\text{init}}^P(4, \text{genA}_Q(5, 3, 2), 5, 5, []) \\ & | \Pi_{1 \leq a \leq 3} P_{\text{init}}^A(a, 4, 3, 2, n_a, pr_a) \\ & \mapsto^* (\nu t) (\bar{b}_4[4](s) . P^P = P_4 \\ & | \bar{b}_5[4](r) . P^P = P_5 \\ & | (b_4[1](s) . P_1^A | b_5[1](r) . P_1^A) = A_1 \\ & | (b_4[2](s) . P_1^A | b_5[2](r) . P_1^A) = A_2 \\ & | (b_4[3](s) . P_1^A | b_5[3](r) . P_1^A) = A_3 \\ & | \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : []) \end{aligned}$$

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:

$$\begin{aligned} & \mapsto^* (\nu t) (\nu s) (\nu r) ( \\ & (\mu X) \text{update}(n, 5) . \left( \bigodot_{a \in \{1, 2\}} s[4, a]!_u l1a \langle \text{proposalNumber}_4(n) \rangle \right) \dots = P_4 \\ & | (\mu X) \text{update}(n, 6) . \left( \bigodot_{a \in \{2, 3\}} r[4, a]!_u l1a \langle \text{proposalNumber}_5(n) \rangle \right) \dots = P_5 \\ & | ((\mu X) s[4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) r[4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots) = A_1 \\ & | ((\mu X) s[4, 2]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) r[4, 2]?_u l1a \langle \perp \rangle (n') . \text{if } \dots) = A_2 \\ & | ((\mu X) s[4, 3]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) r[4, 3]?_u l1a \langle \perp \rangle (n') . \text{if } \dots) = A_3 \\ & | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] | \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] | \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : []) \end{aligned}$$

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

## The Happy Path

After applying the updates in  $P_4$  and  $P_5$  the first inter-process communication can take place. In this case  $P_5$  communicates with  $A_2$  and  $A_3$ . We apply (USend) and (UGet) twice to send  $\text{proposalNumber}_5(6) = 10$  to  $A_2$  and  $A_3$ .

$$\begin{aligned} & \mapsto^* (\nu t) (\nu s) (\nu r) ( \\ & (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4(5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4(5) \rangle \dots = P_4 \\ & | (\mu X) r [2, 4]?_u l1b \langle \perp \rangle (v_2) . r [3, 4]?_u l1b \langle \perp \rangle (v_3) \dots = P_5 \\ & | ((\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) r [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots) = A_1 \\ & | ((\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) \text{if } 10 = \perp \text{ then } s [2, 4]!_u l1b \langle \perp \rangle . P_2^A \text{ else } \dots) = A_2 \\ & | ((\mu X) s [4, 3]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) \text{if } 10 = \perp \text{ then } s [3, 4]!_u l1b \langle \perp \rangle . P_2^A \text{ else } \dots) = A_3 \\ & | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : \square ) \end{aligned}$$

Since  $10 \neq \perp$  both  $A_2$  and  $A_3$  move into their respective else branches.

$$\begin{aligned} & = (\nu t) (\nu s) (\nu r) ( \\ & (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4(5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4(5) \rangle \dots = P_4 \\ & | (\mu X) r [2, 4]?_u l1b \langle \perp \rangle (v_2) . r [3, 4]?_u l1b \langle \perp \rangle (v_3) \dots = P_5 \\ & | ((\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) r [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots) = A_1 \\ & | ((\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) \text{if } \text{gt}(10, \text{Nothing}) \text{ then } \dots \text{else } \dots) = A_2 \\ & | ((\mu X) s [4, 3]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) \text{if } \text{gt}(10, \text{Nothing}) \text{ then } \dots \text{else } \dots) = A_3 \\ & | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : \square ) \end{aligned}$$

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

$$\begin{aligned} & = (\nu t) (\nu s) (\nu r) ( \\ & (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4(5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4(5) \rangle \dots = P_4 \\ & | (\mu X) r [2, 4]?_u l1b \langle \perp \rangle (v_2) . r [3, 4]?_u l1b \langle \perp \rangle (v_3) \dots = P_5 \\ & | ((\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) r [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots) = A_1 \\ & | ((\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) r [2, 4]!_u l1b \langle \text{Promise Nothing} \rangle . P_2^A) = A_2 \\ & | ((\mu X) s [4, 3]?_u l1a \langle \perp \rangle (n') . \text{if } \dots | (\mu X) r [3, 4]!_u l1b \langle \text{Promise Nothing} \rangle . P_2^A) = A_3 \\ & | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : \square ) \end{aligned}$$

We apply (USend) and (UGet) twice to do just that. Note that we also apply (USkip) to  $A_1$ , evaluate its branches, and apply (USend) to  $A_1$  and then (ML) to  $P_5$  to discard the dummy message. All three acceptors move into  $P_2^A$ .

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

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

$$\begin{aligned}
& \mapsto^* (\nu t) (\nu s) (\nu r) ( \\
& (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle \dots = P_4 \\
& | (\mu X) r [4, 2]!_u l2a \langle \text{Proposal } 10 \text{ abc} \rangle . r [4, 3]!_u l2a \langle \text{Proposal } 10 \text{ abc} \rangle \dots = P_5 \\
& | ((\mu X) \dots | (\mu X) r [4, 1]?_w \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0) = A_1 \\
& | ((\mu X) \dots | (\mu X) r [4, 2]?_u l2a \langle \perp \rangle (pr') . \text{if } \dots = A_2 \\
& | ((\mu X) \dots | (\mu X) r [4, 3]?_u l2a \langle \perp \rangle (pr') . \text{if } \dots = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : \square)
\end{aligned}$$

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

$$\begin{aligned}
& \mapsto^* (\nu t) (\nu s) (\nu r) ( \\
& (\mu X) s [4, 1]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle . s [4, 2]!_u l1a \langle \text{proposalNumber}_4 (5) \rangle \dots = P_4 \\
& | (\mu X) s [4, 1]?_u l1a \langle \perp \rangle (n') . \text{if } \dots = A_1 \\
& | (\mu X) s [4, 2]?_u l1a \langle \perp \rangle (n') . \text{if } \dots = A_2 \\
& | (\mu X) s [4, 3]?_u l1a \langle \perp \rangle (n') . \text{if } \dots = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : \square)
\end{aligned}$$

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  $A_3$ .

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

$A_3$  moves directly to  $P_2^A$  whereas  $A_1$  and  $A_2$  send their responses to  $P_4$  before moving to  $P_2^A$ .  $A_1$  also updates its local variable  $n = 5$ .

$$\begin{aligned}
& = (\nu t) (\nu s) (\nu r) ( \\
& (\mu X) s [1, 4]?_u l1b \langle \perp \rangle (v_1) . s [2, 4]?_u l1b \langle \perp \rangle (v_2) \dots = P_4 \\
& | (\mu X) s [1, 4]!_u l1b \langle \text{Promise Nothing} \rangle . P_2^A = A_1 \\
& | (\mu X) s [2, 4]!_u l1b \langle \text{Nack } 10 \rangle . P_2^A = A_2 \\
& | (\mu X) r [4, 3]?_w \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 = A_3 \\
& | \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : \square \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : \square)
\end{aligned}$$

Applying (USend) and (UGet) twice and evaluating the branching in  $P_4$  yields:

$$\begin{aligned}
& \mapsto^* (\nu t) (\nu s) (\nu r) ( \\
& (\mu X) s [4, \{1, 2\}]!_w \text{Restart}.X \\
& | (\mu X) s [4, 1]?_w \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0 \\
& | (\mu X) s [4, 1]?_w \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0
\end{aligned}$$

$(\mu X) r [4, 3] ?_w \text{Accept} \dots \oplus \text{Restart}.X \oplus \text{Abort}.0$   
 $\mid \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : []$

$P_4$  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).

$\mapsto^* (\nu t) (\nu s) (\nu r) ($   
 $(\mu X) s [4, 1] !_u l1a \langle 15 \rangle . s [4, 2] !_u l1a \langle 15 \rangle \dots = P_4$   
 $\mid (\mu X) s [4, 1] ?_u l1a \langle \perp \rangle (n') . \text{if } \dots = A_1$   
 $\mid (\mu X) s [4, 2] ?_u l1a \langle \perp \rangle (n') . \text{if } \dots = A_2$   
 $\mid \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])$

## 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) and (UGet) and evaluating the branches in the remaining acceptors we arrive at:

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

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) and (UGet) and evaluate the branch in  $P_4$  to obtain:

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

$P_4$  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).

$\mapsto^* (\nu t) (\nu s) (\nu r) ($   
 $(\mu X) s [4, 1] !_u l2a \langle \text{Proposal 15 } abc \rangle . s [4, 2] !_u l2a \langle \text{Proposal 15 } abc \rangle . 0 = P_4$   
 $\mid (\mu X) s [4, 1] ?_u l2a \langle \perp \rangle (pr') . \text{if } \dots = A_1$   
 $\mid (\mu X) s [4, 2] ?_u l2a \langle \perp \rangle (pr') . \text{if } \dots = A_2$   
 $\mid \Pi_{1 \leq k, l \leq 4, k \neq l} s_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 4, k \neq l} r_{k \rightarrow l} : [] \mid \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])$

Then we apply (USend) and (UGet) to send the proposal from  $P_4$  to the acceptors.  $P_4$  terminates and the acceptors accept the received proposal and then terminate as well.

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

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

## 4 Analysis

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

### 4.1 Local Types

$G \downarrow_k = 0$  for every  $k$ .

For  $1 \leq a \leq ac$  and  $ac + 1 \leq p \leq ac + pc$  we define the projections of the global type  $G_{p,A_Q}$ .

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

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

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

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

### 4.2 Type Check

#### 4.2.1 System Initialization

$$\Gamma = o : G \cdot b_{ac+1} : G_{p,A_Q} \cdot b_{ac+2} : G_{p,A_Q} \cdot \dots \cdot b_{ac+pc} : G_{p,A_Q}.$$

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



$(S_1)$

$$(S_1) = \frac{\frac{(P)}{\Gamma \vdash \overline{b_{ac+pc}}[ac+1](s) \cdot P^P \triangleright t[2] : G \vdash_2}}{\Gamma \vdash \bar{o}[2](t) \cdot P_{init}^P(ac+1, \text{genA}_Q(ac+pc, ac, pc), ac+pc, ac+pc, []) \triangleright \emptyset} \text{ (Rec)}$$

Since  $G \vdash_2 = 0$ ,  $t[2] : G \vdash_2 = \emptyset$ . This is relevant later when continuing proof  $(P)$ .

$(S_2)$

$$(S_2) = \frac{\frac{\frac{(P)}{\Gamma \vdash \overline{b_{ac+1}}[ac+1](s) \cdot P^P \triangleright \emptyset} \quad \dots \quad \frac{(P)}{\Gamma \vdash \overline{b_{ac+pc-1}}[ac+1](s) \cdot P^P \triangleright \emptyset}}{\Gamma \vdash \Pi_{ac < k < ac+pc} P_{init}^P(ac+1, \text{genA}_Q(k, ac, pc), k, k, []) \triangleright t[1] : G \vdash_1} \text{ (Par)}^{??}}{\Gamma \vdash o[1](t) \cdot \Pi_{ac < k < ac+pc} P_{init}^P(\dots) \triangleright \emptyset} \text{ (Acc)}$$

Because  $G \vdash_1 = 0$ ,  $t[1] : G \vdash_1 = \emptyset$ . So  $\Delta$  for each proposer is empty at this point.

$(S_3)$

$$(S_3) = \frac{\frac{\frac{(A)}{\Gamma \vdash P_1^A \triangleright s[a] : G_{p,A_Q} \vdash_j}}{\Gamma \vdash b_k[j](s) \cdot P_1^A \triangleright \emptyset} \text{ (Acc)} \quad \dots \quad \text{ (Par)}^*}{\Gamma \vdash \Pi_{ac < k \leq ac+pc} b_k[ac](s) \cdot P_1^A \triangleright \emptyset} \text{ (Par)}^* \quad \dots \quad \text{ (Par)}^*}{\Gamma \vdash \Pi_{1 \leq a \leq ac} (\Pi_{ac < k \leq ac+pc} b_k[a](s) \cdot P_1^A) \triangleright \emptyset} \text{ (Par)}^*$$

#### 4.2.2 Proposer

For simplicity, let  $p = ac + 1$ ,  $A_Q = \text{genA}_Q(y, ac, pc)$ ,  $n = y$ ,  $m = y$ ,  $\vec{V} = []$  where  $ac < y \leq ac + pc$ .

$$T_{acc}^P = \left( \bigodot_{a \in A_Q} [a]!_{ul} 2a \langle \text{Proposal Value} \rangle \right) . 0$$

$$T_{branch}^P = ([A_Q]!_w \text{Accept} . T_{acc}^P \oplus \text{Restart} . x \oplus \text{Abort} . 0)$$

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

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

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

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

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

(P)

$$\begin{array}{c}
 \frac{(P_t)}{\Gamma'' \vdash s[p, A_Q]!_w \text{Restart}.X \triangleright s[p] : T_{\text{branch}}^P} \quad \frac{(P_f)}{\Gamma'' \vdash s[p, A_Q]!_w \text{Accept} \dots \triangleright s[p] : T_{\text{branch}}^P} \\
 \hline
 \Gamma'' \vdash \text{if } e \text{ then } s[p, A_Q]!_w \text{Restart}.X \text{ else } s[p, A_Q]!_w \text{Accept} \dots \triangleright s[p] : T_{\text{branch}}^P \quad (\text{If}) \\
 \hline
 \Gamma' \vdash \left( \odot_{a \in A_Q} s[a, p]?_u l1b \langle \perp \rangle (v_a) \right) \dots \triangleright s[p] : \left( \odot_{a \in A_Q} [a]?_u l1b \langle \text{Promise Value} \rangle \right) \quad (\text{UGet})^{|A_Q|} \\
 \hline
 \Gamma' \vdash \left( \odot_{a \in A_Q} s[p, a]!_u l1a \langle pn \rangle \right) \dots \triangleright s[p] : \left( \odot_{a \in A_Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots \quad (\text{USend})^{|A_Q|} \\
 \hline
 \Gamma' \vdash \left( \odot_{a \in A_Q} s[p, a]!_u l1a \langle pn \rangle \right) \dots \triangleright s[p] : \left( \odot_{a \in A_Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots \quad ? \\
 \hline
 \Gamma' \vdash \text{update}(n, n+1) \dots \triangleright s[p] : \left( \odot_{a \in A_Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots \quad (\text{Rec}) \\
 \hline
 \Gamma \vdash (\mu X) \text{update}(n, n+1) \dots \triangleright s[p] : (\mu x) \left( \odot_{a \in A_Q} [a]!_u l1a \langle \mathbb{N} \rangle \right) \dots \quad (\text{Req}) \\
 (P) = \frac{}{\Gamma \vdash \bar{b}_n[p](s) \cdot P^P \triangleright \emptyset}
 \end{array}$$

(P<sub>t</sub>)

$$(P_t) = \frac{\frac{}{\Gamma \vdash X \triangleright s[p] : x} (\text{Var})}{\Gamma \vdash s[p, A_Q]!_w \text{Restart}.X \triangleright s[p] : ([A_Q]!_w \text{Accept} \cdot T_{\text{acc}}^P \oplus \text{Restart}.x \oplus \text{Abort}.0)} (\text{WSel})$$

(P<sub>f</sub>)

$$\begin{array}{c}
 \frac{}{\Gamma \vdash 0 \triangleright s[p] : 0} (\text{End}) \\
 \hline
 (P_f) = \frac{\Gamma \vdash \left( \odot_{a \in A_Q} s[p, a]!_u l2a \langle prop \rangle \right) \cdot 0 \triangleright s[p] : \left( \odot_{a \in A_Q} [a]!_u l2a \langle \text{Proposal Value} \rangle \right) \cdot 0}{\Gamma \vdash s[p, A_Q]!_w \text{Accept} \dots \triangleright s[p] : ([A_Q]!_w \text{Accept} \cdot T_{\text{acc}}^P \oplus \text{Restart}.x \oplus \text{Abort}.0)} (\text{WSel})
 \end{array}$$

### 4.2.3 Acceptor

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

$i = ac + 1$

$P_{\text{acc}}^A = s[p, a]?_u l2a \langle \perp \rangle (pr') \cdot \text{if } pr' = \perp$   
 then 0  
 else if  $\text{ge}(\text{nFromProposal}(pr'), n)$

then  $\text{update}(pr, pr') . \text{update}(n, \text{Just } n\text{FromProposal}(pr')) . 0$   
 else  $0$

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

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

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

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

$$T_{branch}^A = ([p]?_w \text{Accept} . T_{acc}^A \oplus \text{Restart}.x \oplus \text{Abort}.0)$$

$$T_{1b}^A = [p]!_u l1b \langle \text{Promise Value} \rangle . T_{branch}^A$$

(A)

$$(A) = \frac{\frac{\frac{\Gamma \cdot X : x \vdash P_2^A \triangleright s[a] : T_{branch}^A}{\Gamma \cdot X : x \vdash s[a, p]!_u l1b \langle \perp \rangle . P_2^A \triangleright s[a] : T_{1b}^A} \text{ (USend)}}{\Gamma \cdot X : x \vdash \text{if } n' = \perp \text{ then } P_2^A \text{ else if } gt(n', n) \dots \triangleright s[a] : T_{1b}^A} \text{ (If)}}{\frac{\frac{\Gamma \cdot X : x \vdash s[p, a]?_u l1a \langle \perp \rangle (n') . \text{if } n' = \perp \dots \triangleright s[a] : [p]?_u l1a \langle \mathbb{N} \rangle . T_{1b}^A}{\Gamma \vdash (\mu X) s[p, a]?_u l1a \langle \perp \rangle (n') \dots \triangleright s[a] : (\mu x) [p]?_u l1a \langle \mathbb{N} \rangle \dots} \text{ (UGet)}}{\Gamma \vdash (\mu X) s[p, a]?_u l1a \langle \perp \rangle (n') \dots \triangleright s[a] : (\mu x) [p]?_u l1a \langle \mathbb{N} \rangle \dots} \text{ (Rec)}$$

(A<sub>gt</sub>)

$$(A_{gt}) = \frac{\frac{\frac{\Gamma \cdot X : x \vdash P_t^A \triangleright s[a] : T_{1b}^A}{\Gamma \cdot X : x \vdash \text{if } gt(n', n) \text{ then } P_t^A \text{ else } P_f^A \triangleright s[a] : T_{1b}^A} \text{ (If)}}{\Gamma \cdot X : x \vdash \text{if } gt(n', n) \text{ then } P_t^A \text{ else } P_f^A \triangleright s[a] : T_{1b}^A} \text{ (If)}$$

(A<sub>t</sub>)

$$(A_t) = \frac{\frac{\frac{\Gamma \cdot X : x \vdash P_2^A \triangleright s[a] : T_{branch}^A}{\Gamma \cdot X : x \vdash s[a, p]!_u l1b \langle \text{Nack } n \rangle . P_2^A \triangleright s[a] : [p]!_u l1b \langle \text{Promise Value} \rangle . T_{branch}^A} \text{ (USend)}}{\Gamma \cdot X : x \vdash s[a, p]!_u l1b \langle \text{Nack } n \rangle . P_2^A \triangleright s[a] : [p]!_u l1b \langle \text{Promise Value} \rangle . T_{branch}^A} \text{ (USend)}$$

(A<sub>f</sub>)

$$(A_f) = \frac{\frac{\frac{\Gamma \cdot X : x \vdash P_2^A \triangleright s[a] : T_{branch}^A}{\Gamma \cdot X : x \vdash s[a, p]!_u l1b \langle \text{Nack } n \rangle . P_2^A \triangleright s[a] : [p]!_u l1b \langle \text{Promise Value} \rangle . T_{branch}^A} \text{ (USend)}}{\Gamma \cdot X : x \vdash s[a, p]!_u l1b \langle \text{Nack } n \rangle . P_2^A \triangleright s[a] : [p]!_u l1b \langle \text{Promise Value} \rangle . T_{branch}^A} \text{ (USend)}$$

$(A_c)$

$$(A_c) = \frac{\frac{(A_{Accept})}{\Gamma \cdot X : x \vdash P_{acc}^A \triangleright s[a] : T_{acc}^A} \quad \frac{\Gamma \cdot X : x \vdash X \triangleright s[a] : x}{\Gamma \cdot X : x \vdash P_2^A \triangleright s[a] : T_{branch}^A} \text{(Var)} \quad \frac{\Gamma \cdot X : x \vdash 0 \triangleright s[a] : 0}{\Gamma \cdot X : x \vdash 0 \triangleright s[a] : 0} \text{(End)}}{\Gamma \cdot X : x \vdash P_2^A \triangleright s[a] : T_{branch}^A} \text{(WBran)}$$

$(A_{Accept})$

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

$(A_{update})$

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



---

## 5 Discussion

---

DISCUSSION

---

## Bibliography

---

- [1] Tushar Deepak Chandra and Sam Toueg. “Unreliable Failure Detectors for Reliable Distributed Systems”. In: *J. ACM* 43.2 (Mar. 1996), pp. 225–267. ISSN: 0004-5411. DOI: 10.1145/226643.226647. URL: <https://doi.org/10.1145/226643.226647>.
- [2] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (3rd Edition)*. Addison-Wesley, 2001, p. 452.
- [3] Leslie Lamport. “Lower Bounds for Asynchronous Consensus”. In: *Distrib. Comput.* 19.2 (Oct. 2006), pp. 104–125. ISSN: 0178-2770. DOI: 10.1007/s00446-006-0155-x. URL: <https://doi.org/10.1007/s00446-006-0155-x>.
- [4] Leslie Lamport. “Paxos Made Simple”. In: *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001) (Dec. 2001), pp. 51–58. URL: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>.
- [5] K. Peters, U. Nestmann, and C. Wagner. “Fault-Tolerant Multiparty Session Types”. Provided by K. Peters. 2021.
- [6] A. Scalas and N. Yoshida. “Multiparty session types, beyond duality”. In: *Journal of Logical and Algebraic Methods in Programming* 97 (2018), pp. 55–84.