

# Analyzing Paxos with Fault-Tolerant Multiparty Session Types

Bachelor thesis by Nicolas Daniel Torres  
Date of submission: November 15, 2021

1. Review: Prof. Dr. Kirstin Peters  
Darmstadt



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Computer Science  
Department  
<institute>  
<working group>

---

# Contents

---

<b>1</b>	<b>Model and Analysis</b>	<b>3</b>
1.1	Sorts . . . . .	3
1.2	Global Type . . . . .	3
1.3	Functions . . . . .	4
1.4	Processes . . . . .	5
1.4.1	System Initialization . . . . .	5
1.4.2	Proposer . . . . .	6
1.4.3	Acceptor . . . . .	7
1.5	Failure Patterns . . . . .	8
1.6	Example . . . . .	9
1.6.1	Scenario . . . . .	10
1.6.2	Formulae . . . . .	10

---

# 1 Model and Analysis

---

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.

---

## 1.1 Sorts

---

We assume the following sorts.

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

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

Value = Set of values.

Promise  $a = \{\text{Promise (Maybe (Proposal } a)), \text{Nack } \mathbb{N}\}$

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

---

## 1.2 Global Type

---

Since each proposer initiates its own session the global type can be defined for one proposer. A quorum of acceptors  $A_Q$  is assumed.

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

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

---

---

$\oplus Restart.X$   
 $\oplus Abort.end$

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.

---

## 1.3 Functions

---

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

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

$proposalNumber(a, b)$  returns a proposal number when given two natural numbers.

$promiseValue : [Promise\ a] \rightarrow a$

$promiseValue(ps)$  returns a new 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  $Promise\ (Just\ v)$ .

$anyNack : [Promise\ a] \rightarrow Bool$

$anyNack([]) = false$

$anyNack((Nack\ \_ : \_)) = true$

$anyNack((\_ : xs)) = anyNack(xs)$

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

$promiseCount : [Promise\ a] \rightarrow \mathbb{N}$

$promiseCount([]) = 0$

$promiseCount((Promise\ \_ : xs)) = 1 + promiseCount(xs)$

$promiseCount((\_ : xs)) = promiseCount(xs)$

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

---

$gt : a \rightarrow \text{Maybe } a \rightarrow \text{Bool}$

$gt (\_, \text{Nothing}) = \text{true}$

$gt (a, \text{Just } b) = a > b$

$ge : a \rightarrow \text{Maybe } a \rightarrow \text{Bool}$

$ge (\_, \text{Nothing}) = \text{true}$

$ge (a, \text{Just } b) = a \geq b$

$n\text{FromProposal} : \text{Proposal } a \rightarrow \mathbb{N}$

$n\text{FromProposal} (\text{Proposal } n \_) = n$

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

$\text{genA}_Q : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$\text{genA}_Q (i, ac, pc)$  returns a randomly selected set  $A_Q$  with  $A_Q \subseteq A = \{1, \dots, ac\}$  and  $|A_Q| > \frac{|A|}{2}$ .

$\text{update } (n, m)$  replaces the value inside  $n$  with the value in  $m$ .

---

## 1.4 Processes

---

### 1.4.1 System Initialization

$\text{Sys } (ac, pc) = \bar{a} [2] (t) . P_{\text{init}}^p (ac + 1, \text{genA}_Q (ac + pc, ac, pc), ac + pc, ac + pc, [])$

$| a [1] (t) . \Pi_{ac < i < ac+pc} P_{\text{init}}^p (ac + 1, \text{genA}_Q (i, ac, pc), i, i, [])$

$| \Pi_{1 \leq j \leq ac} P_{\text{init}}^a (j, ac + 1, ac, pc, n_j, pr_j)$

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

$P_{\text{init}}^a (j, i, ac, pc, n, pr) = \Pi_{ac < k \leq ac+pc} b_k [j] (s) . P^a$

$\text{Sys } (ac, pc)$ ,  $P_{\text{init}}^p (i, A_Q, n, m, \vec{V})$ , and  $P_{\text{init}}^a (j, i, ac, pc, n, pr)$  describe the system initialization.  $ac$  and  $pc$  are the number of acceptors and proposers respectively.

An outer session is created through shared-point  $a$ . 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  $ac + pc$ .

$P_{\text{init}}^p(i, A_Q, n, m, \vec{V})$  is initialized with the proposer's role in its own session  $i$ , 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  $i = ac + 1$  but uses a different shared-point  $b_n$  according to its index  $n$ .  $\vec{V}$  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  $P^p$ .

$P_{\text{init}}^a(j, i, ac, pc, n, pr)$  is initialized with the acceptor's index  $j$ , the proposer index  $i$ , 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  $P^a$ .

### 1.4.2 Proposer

$P^p = (\mu X) \text{ update } (n, n + 1) .$   
 $\left( \odot_{j \in A_Q} s[i, j]!_u l1a \langle \text{proposalNumber } (n, m) \rangle \right) .$   
 $\left( \odot_{j \in A_Q} s[j, i]?_u l1b \langle \perp \rangle (v_j) \right) .$   
 if anyNack  $\left( \vec{V} \right)$  or promiseCount  $\left( \vec{V} \right) < \left\lceil \frac{i}{2} \right\rceil$   
 then  $s[i, A_Q]!_w \text{Restart}.X$   
 else  
 $s[i, A_Q]!_w \text{Accept} .$   
 $\odot_{j \in A_Q} s[i, j]!_u l2a \left\langle \text{Proposal proposalNumber } (n, m) \text{ promiseValue } \left( \vec{V} \right) \right\rangle .$   
 end

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  $v_j$ . Because  $i = ac + 1$ , the minimum number of acceptors needed to form a majority is  $\left\lceil \frac{i}{2} \right\rceil = \left\lceil \frac{ac+1}{2} \right\rceil$ . 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.

### 1.4.3 Acceptor

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

For each proposer an acceptor has a corresponding subprocess, which behaves like  $P^a$ . 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 no response to the proposer. 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  $pr$ .

---

## 1.5 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  $\diamond\mathcal{W}$ .

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 [2].  $FP_{\text{crash}}$  returns true if, and only if, at least one more acceptor may crash, i.e.  $n > 2(f + 1)$  is satisfied.



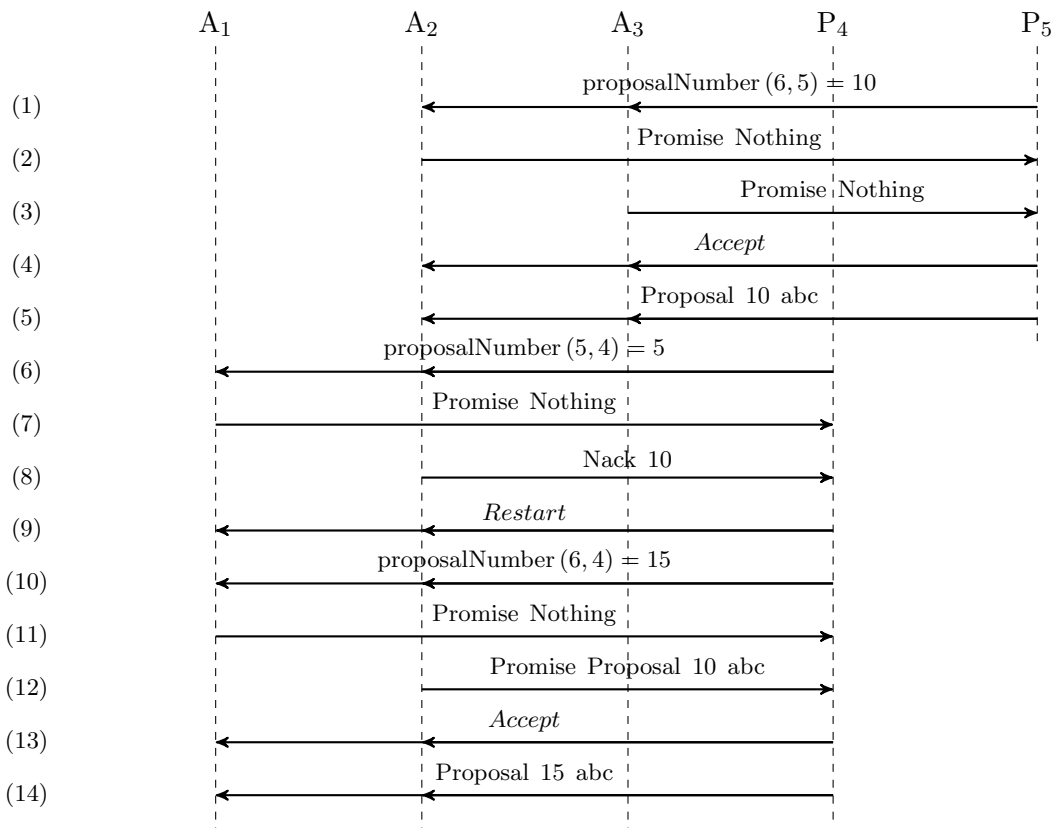


Figure 1.1: Example scenario with 3 acceptors and 2 proposers.

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

## 1.6 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.

### 1.6.1 Scenario

Figure 1.1 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 4 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*.

### 1.6.2 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 we have:

$$\begin{aligned}
 \text{Sys}(ac, pc) &= \text{Sys}(3, 2) = a[1](t) \cdot \Pi_{3 < i < 5} P_{\text{init}}^p(4, \text{genA}_Q(i, 3, 2), i, i, []) \\
 &| \bar{a}[2](t) \cdot P_{\text{init}}^p(4, \text{genA}_Q(5, 3, 2), 5, 5, []) \\
 &| \Pi_{1 \leq j \leq 3} P_{\text{init}}^a(j, 4, 3, 2, n_j, pr_j) \\
 &\longmapsto (\nu t) (\bar{b}_4[4](s) \cdot P^p = P_4 \\
 &| \bar{b}_5[4](r) \cdot P^p = P_5 \\
 &| (b_4[1](s) \cdot P^a | b_5[1](r) \cdot P^a) = A_1 \\
 &| (b_4[2](s) \cdot P^a | b_5[2](r) \cdot P^a) = A_2 \\
 &| (b_4[3](s) \cdot P^a | b_5[3](r) \cdot P^a) = A_3 \\
 &| \Pi_{1 \leq k, l \leq 2, k \neq l} t_{k \rightarrow l} : [])
 \end{aligned}$$

We apply (Init) thrice for shared-point  $b_4$  and thrice again for shared-point  $b_5$  to obtain:

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

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  $\text{proposalNumber } (6, 5) = 10$  to  $A_2$  and  $A_3$ .

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

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

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

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

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

We apply (USend) and (UGet) twice to do just that. Note that we also apply (USkip) to  $A_1$  and evaluate its branches. All three acceptors move into  $P_{\text{cont}}^a$ .

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

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

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

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.

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

At this point the local variables of  $A_2$  and  $A_3$  are  $n = 10$  and  $pr = \text{Proposal } 10$  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  $A_3$ .

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

$A_3$  moves directly to  $P_{\text{cont}}^a$  whereas  $A_1$  and  $A_2$  send their responses to  $P_4$  before moving to  $P_{\text{cont}}^a$ .  $A_1$  also updates its local variable  $n = 5$ .

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

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

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

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

$$\begin{aligned} & \mapsto (\nu t) (\nu s) (\nu r) ( \\ & (\mu X) s [4, 1]!_u l1a \langle 15 \rangle . s [4, 2]!_u l1a \langle 15 \rangle \dots = P_4 \end{aligned}$$

---


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

### 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:

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

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:

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

$P_4$  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).

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

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

---

## 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] 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>.