INTRODUCING TIME PASSAGE TO THE REVERSIBLE SEMANTICS FOR ERLANG

Yuna Sadamoto, Shoji Yuen

Nagoya University, Japan

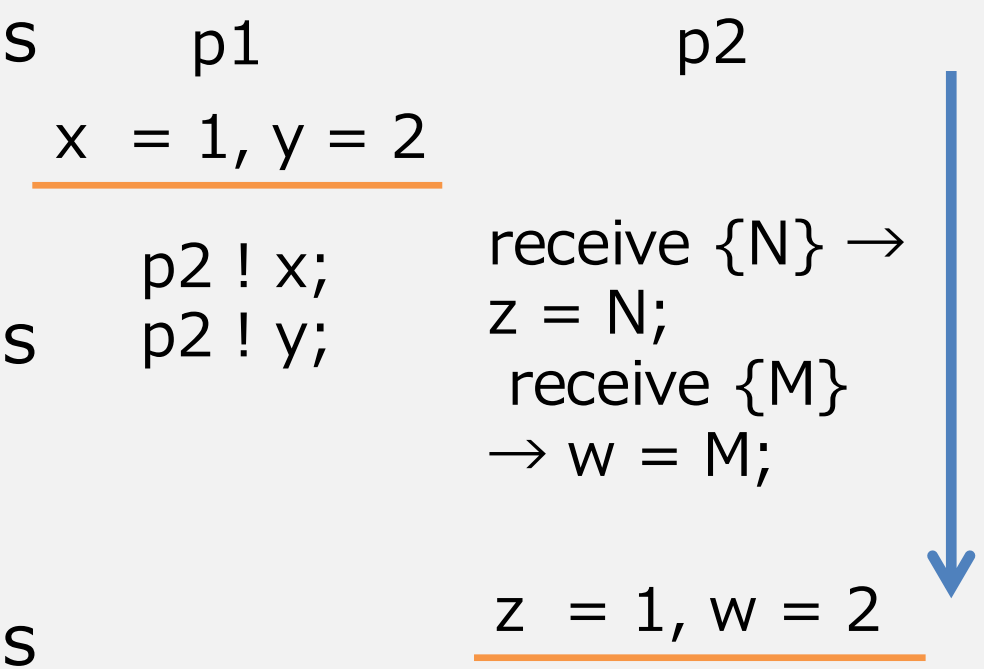
Claudio Antares Mezzina

University of Urbino, Italy

CONTENTS

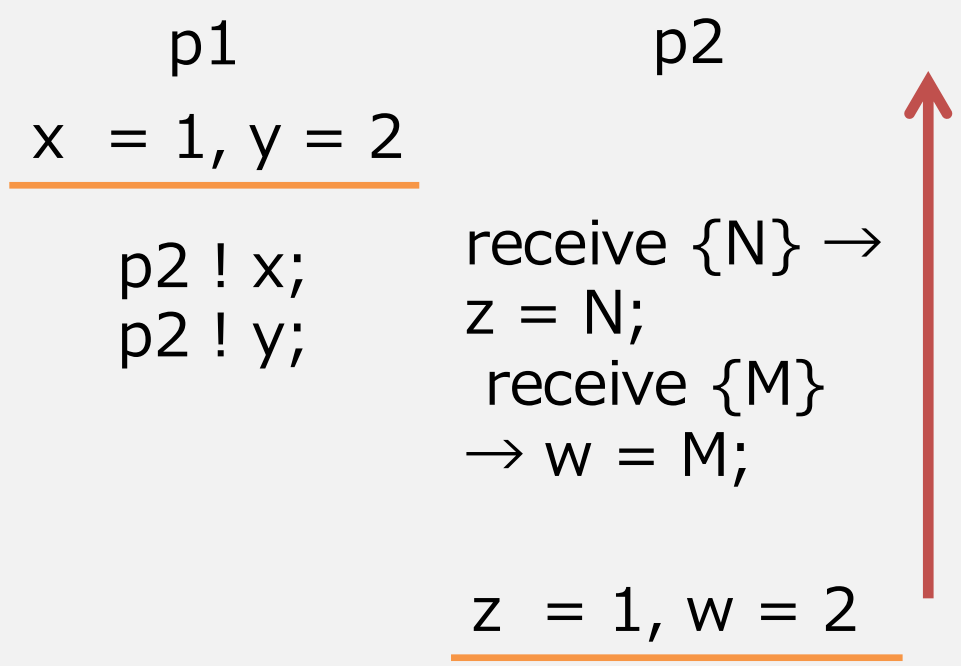
Forward Execution in Erlang

1. Initially, in p1, x is set to 1 and y is set to 2
2. p1 sends message 1 to p2
3. p2 receives message 1 and assigns 1 to z
4. p1 sends message 2 to p2
5. p2 receives message 2 and assigns 2 to w
6. Then, z is equal to 1 and w is equal to 2



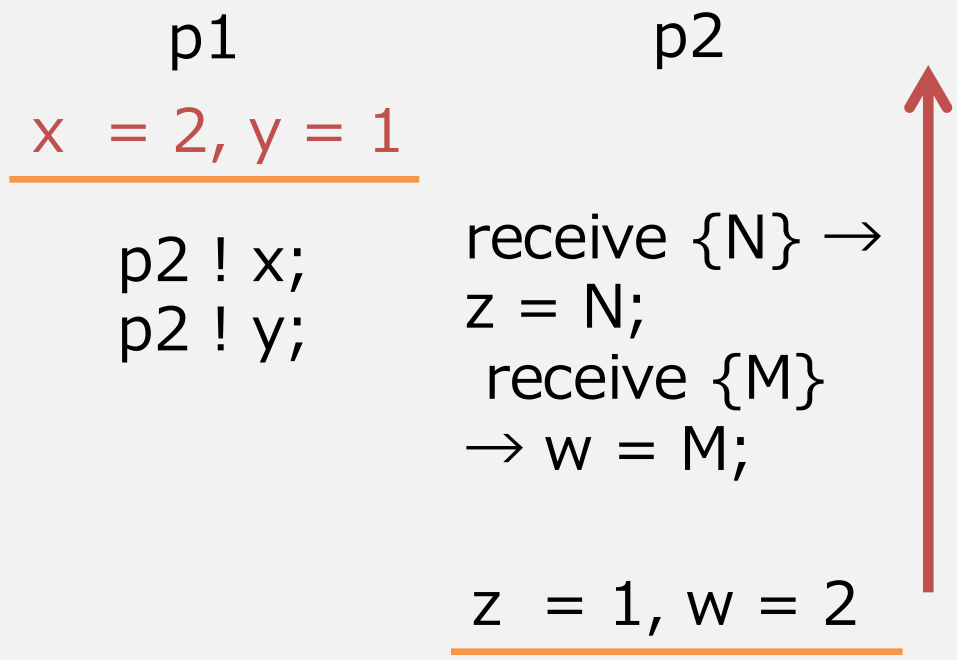
Backward Execution in Erlang

6. z is equal to 1 and w is equal to 2
5. p2 receives message 2 and assigns 2 to w
4. p1 sends message 2 to p2
3. p2 receives message 1 and assigns 1 to z
2. p1 sends message 1 to p2
1. Then, in p1, x is set to 1 and y is set to 2



Another Backward Execution in Erlang

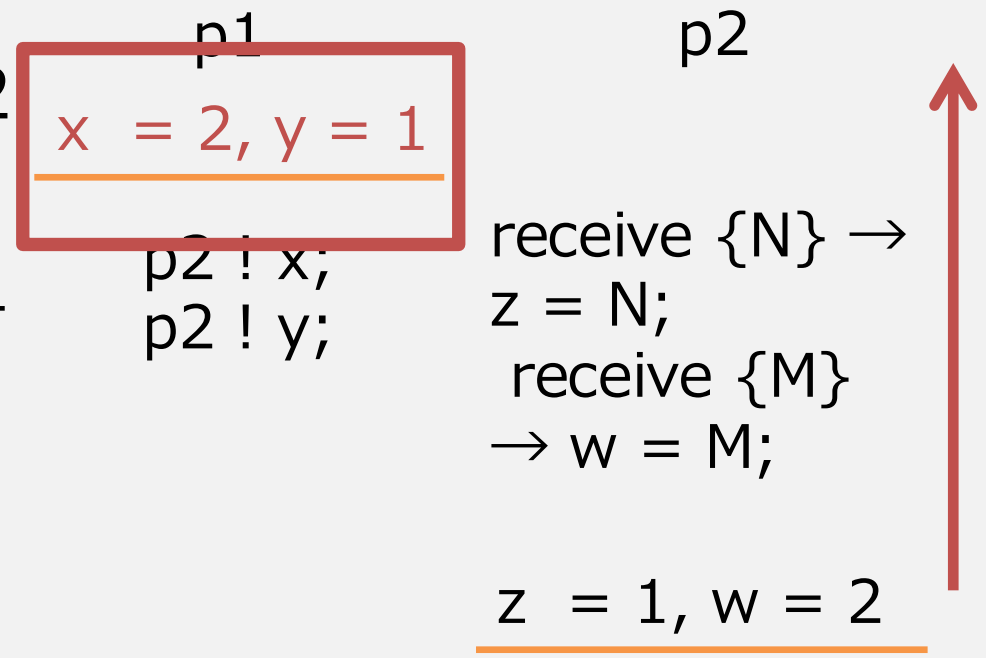
6. z is equal to 1 and w is equal to 2
5. p2 receives message 2 and assigns 2 to w
3. p2 receives message 1 and assigns 1 to z
4. p1 sends message 2 to p2
2. p1 sends message 1 to p2
1. Then, in p1, x is set to 2 and y is set to 1



each process ○ whole system ✕

Another Backward Execution in Erlang

6. z is equal to 1 and w is equal to 2
5. p2 receives message 2 and assigns 2 to w
3. p2 receives message 1 and assigns 1 to z
4. p1 sends message 2 to p2
2. p1 sends message 1 to p2
1. Then, in p1, x is set to 2 and y is set to 1



each process ○ whole system ✕

Erlang's causal consistent reversible semantics [Lanese+, 18][Lami+, 24]

-- forward reversible semantics

A system configuration is represented as a pair $\Gamma ; \Pi$,
Γ : global mailbox and Π : processes

$$[\textsc{Send}] \quad \frac{(\theta, e, S) \xrightarrow{\mathsf{send}(p_r, v)} (\theta', e', S') \quad \lambda \text{ is fresh}}{\Gamma; \langle p_s, h, \theta, e, S\rangle \mid \Pi \rightharpoonup \Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_s, \mathsf{send}(\theta, e, S, \{v, \lambda\}) : h, \theta', e', S'\rangle \mid \Pi}$$

- λ is a unique identifier carried by each message
- Add $(p_s, p_r, \{v, \lambda\})$ to Γ
- Update the history
- Update (θ', e', S') in Π

Erlang's causal consistent reversible semantics [Lanese+, 18][Lami+, 24]

-- forward reversible semantics

A system configuration is represented as a pair $\Gamma\,;\Pi$,
Γ : global mailbox and Π : processes

$$[\textsc{Send}]\quad \frac{(\theta, e, S) \xrightarrow{\mathsf{send}(p_r, v)} (\theta', e', S') \qquad \lambda \text{ is fresh}}{\Gamma; \langle p_s, h, \theta, e, S\rangle \mid \Pi \rightharpoonup \Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_s, \mathsf{send}(\theta, e, S, \{v, \lambda\}) : h, \theta', e', S'\rangle \mid \Pi}$$

- λ is a unique identifier carried by each message
- Add $(p_s, p_r, \{v, \lambda\})$ to Γ
- Update the history
- Update (θ', e', S') in Π

Erlang's causal consistent reversible semantics

[Lanese+, 18][Lami+, 24]

-- forward reversible semantics

A system configuration is represented as a pair $\Gamma ; \Pi$,
Γ : global mailbox and Π : processes

$$[\textsc{Send}] \quad \frac{(\theta, e, S) \xrightarrow{\mathsf{send}(p_r, v)} (\theta', e', S') \qquad \lambda \text{ is fresh}}{\Gamma; \langle p_s, h, \theta, e, S\rangle \mid \Pi \rightharpoonup \Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_s, \mathsf{send}(\theta, e, S, \{v, \lambda\}) : h, \theta', e', S'\rangle \mid \Pi}$$

- λ is a unique identifier carried by each message
- Add $(p_s, p_r, \{v, \lambda\})$ to Γ
- **Update the history**
- Update (θ', e', S') in Π

Erlang's causal consistent reversible semantics [Lanese+, 18][Lami+, 24]

-- forward reversible semantics

A system configuration is represented as a pair $\Gamma ; \Pi$,
Γ : global mailbox and Π : processes

$$[\textsc{Send}] \quad \frac{(\theta, e, S) \xrightarrow{\mathsf{send}(p_r, v)} (\theta', e', S') \qquad \lambda \text{ is fresh}}{\Gamma; \langle p_s, h, \theta, e, S\rangle \mid \Pi \rightharpoonup \Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_s, \mathsf{send}(\theta, e, S, \{v, \lambda\}) : h, \theta', e', S'\rangle \mid \Pi}$$

- λ is a unique identifier carried by each message
- Add $(p_s, p_r, \{v, \lambda\})$ to Γ
- Update the history
- Update (θ', e', S') in Π

Erlang's causal consistent reversible semantics [Lanese+, 18][Lami+, 24]

-- backward reversible semantics

A system configuration is represented as a pair $\Gamma ; \Pi$,
Γ : global mailbox and Π : processes

$$[\textsc{Send}] \quad \Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_s, \mathsf{send}(\theta, e, S, \{v, \lambda\}) : h, \theta', e', S' \rangle \mid \Pi \longleftarrow \Gamma; \langle p_s, h, \theta, e, S \rangle \mid \Pi$$

- θ, e and S are restored from the history
- The history is popped from h

Erlang's causal consistent reversible semantics [Lanese+, 18][Lami+, 24]

-- backward reversible semantics

A system configuration is represented as a pair $\Gamma ; \Pi$,
Γ : global mailbox and Π : processes

$$[\text{SEND}] \quad \Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_s, \text{send}(\theta, e, S, \{v, \lambda\}) : h, \theta', e', S' \rangle \mid \Pi \longleftarrow \Gamma; \langle p_s, h, \theta, e, S \rangle \mid \Pi$$

- θ, e and S are restored from the history
- The history is popped from h

CONTENTS

TIME PASSAGE IN ERLANG

Erlang Program Example 1

main : creates processes p1, p2 and p3

p1 : pauses for 2 milliseconds, then sends message 1 to p3

p2 : pauses for 1 millisecond, then sends message 2 to p3

p3 : outputs the first received message

```
1   -module(time).
2   -export([main/0, p1/1, p2/1, p3/0]).
3
4
5   main() ->
6       Pid = spawn(?MODULE, p3, []),
7       spawn(?MODULE, p1, [Pid]),
8       spawn(?MODULE, p2, [Pid]).
9
10  p1(Pid) ->
11      timer:sleep(2),
12      Pid ! {1}.
13
14  p2(Pid) ->
15      timer:sleep(1),
16      Pid ! {2}.
17
18  p3() ->
19      receive
20          {N} ->
21              io:format("Received: ~p~n", [N])
22      end.
```

TIME PASSAGE IN ERLANG

Erlang Program Example 1

main : creates processes p1, p2 and p3

p1 : pauses for 2 milliseconds, then sends message 1 to p3

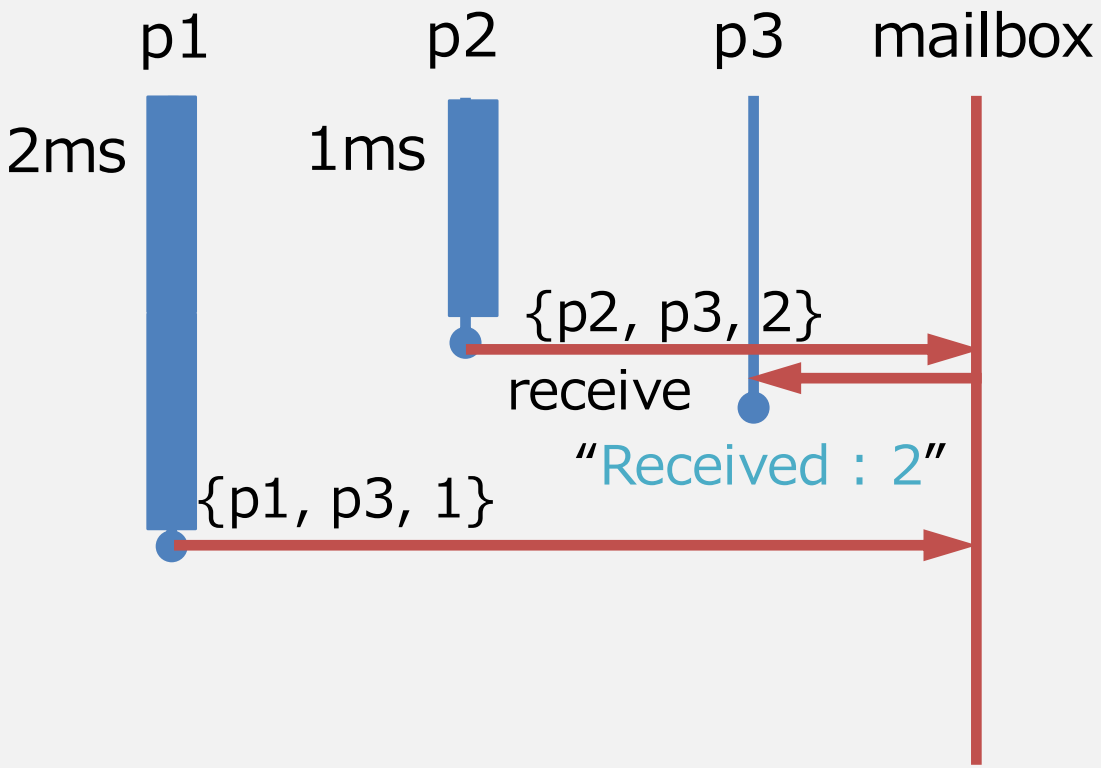
p2 : pauses for 1 millisecond, then sends message 2 to p3

p3 : outputs the first received message

```
1   -module(time).
2   -export([main/0, p1/1, p2/1, p3/0]).
3
4
5   main() ->
6       Pid = spawn(?MODULE, p3, []),
7       spawn(?MODULE, p1, [Pid]),
8       spawn(?MODULE, p2, [Pid]).
9
10  p1(Pid) ->
11      timer:sleep(2),
12      Pid ! {1}.
13
14  p2(Pid) ->
15      timer:sleep(1),
16      Pid ! {2}.
17
18  p3() ->
19      receive
20          {N} ->
21              io:format("Received: ~p~n", [N])
22      end.
```

TIME PASSAGE IN THE ERLANG SEMANTICS

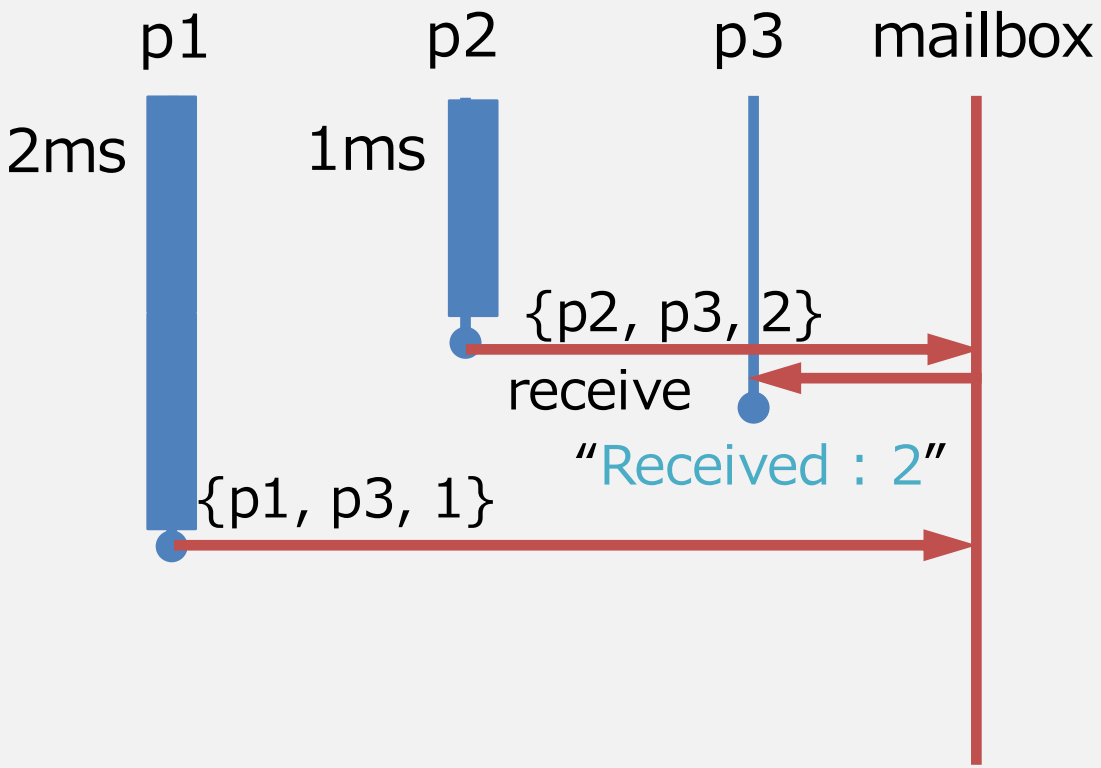
Erlang Program Example 1




```
1   -module(time).
2   -export([main/0, p1/1, p2/1, p3/0]).
3
4
5   main() ->
6       Pid = spawn(?MODULE, p3, []),
7       spawn(?MODULE, p1, [Pid]),
8       spawn(?MODULE, p2, [Pid]).
9
10  p1(Pid) ->
11      timer:sleep(2),
12      Pid ! {1}.
13
14  p2(Pid) ->
15      timer:sleep(1),
16      Pid ! {2}.
17
18  p3() ->
19      receive
20          {N} ->
21              io:format("Received: ~p~n", [N])
22      end.
```

TIME PASSAGE IN THE ERLANG SEMANTICS

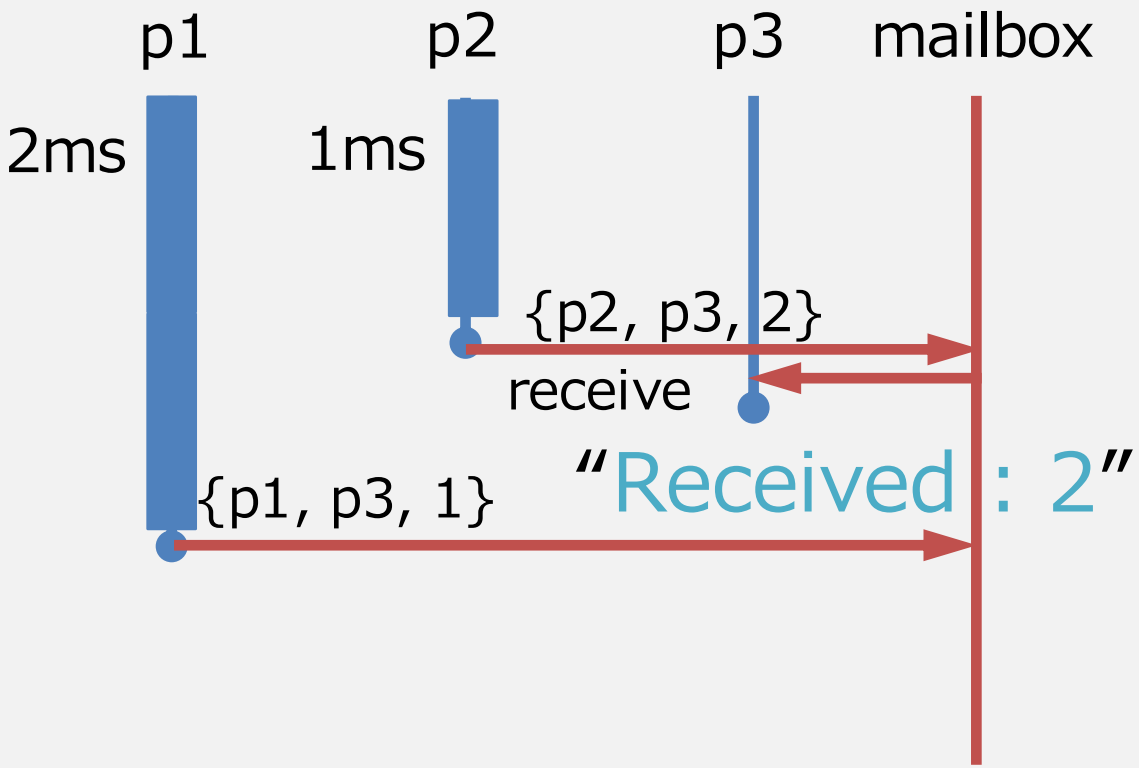
Erlang Program Example 1




```
1   -module(time).
2   -export([main/0, p1/1, p2/1, p3/0]).
3
4
5   main() ->
6       Pid = spawn(?MODULE, p3, []),
7       spawn(?MODULE, p1, [Pid]),
8       spawn(?MODULE, p2, [Pid]).
9
10  p1(Pid) ->
11      timer:sleep(2),
12      Pid ! {1}.
13
14  p2(Pid) ->
15      timer:sleep(1),
16      Pid ! {2}.
17
18  p3() ->
19      receive
20          {N} ->
21              io:format("Received: ~p~n", [N])
22      end.
```

TIME PASSAGE IN THE ERLANG SEMANTICS

Erlang Program Example 1




```
1   -module(time).
2   -export([main/0, p1/1, p2/1, p3/0]).
3
4
5   main() ->
6       Pid = spawn(?MODULE, p3, []),
7       spawn(?MODULE, p1, [Pid]),
8       spawn(?MODULE, p2, [Pid]).
9
10  p1(Pid) ->
11      timer:sleep(2),
12      Pid ! {1}.
13
14  p2(Pid) ->
15      timer:sleep(1),
16      Pid ! {2}.
17
18  p3() ->
19      receive
20          {N} ->
21              io:format("Received: ~p~n", [N])
22      end.
```

TIME PASSAGE IN THE ERLANG SEMANTICS

Erlang Program Example 1

- Running it in CauDEr,
  Received: 1 be produced instead

```
1   -module(time).
2   -export([main/0, p1/1, p2/1, p3/0]).
3
4
5   main() ->
6       Pid = spawn(?MODULE, p3, []),
7       spawn(?MODULE, p1, [Pid]),
8       spawn(?MODULE, p2, [Pid]).
9
10  p1(Pid) ->
16      Pid ! {2}.
17
18  p3() ->
21          io:format("Received: ~p~n", [N])
22      end.
```

Received: 1

Lack of time passage

TIME PASSAGE IN THE ERLANG SEMANTICS

Erlang Program Example 1

- Time passage in Erlang is represented as receive after n → ok end [Cesarini +, 10]
- receive $\overline{cl}$ after $v^T \rightarrow exprs$ end is not included in the reversible semantics in Erlang [Lanese+,18]

```
1  -module(time_new).
2  -export([main/0, p1/1, p2/1, p3/0]).
3
4
5  main() ->
6      Pid = spawn(?MODULE, p3, []),
7      spawn(?MODULE, p1, [Pid]),
8      spawn(?MODULE, p2, [Pid]).
9
10 p1(Pid) ->
11     receive after 2 -> ok end,
12     Pid ! {1}.
13
14 p2(Pid) ->
15     receive after 1 -> ok end,
16     Pid ! {2}.
17
18 p3() ->
19     receive
20         {N} ->
21             io:format("Received: ~p~n", [N])
22     end.
```

Lack of time passage

TIME PASSAGE IN THE ERLANG SEMANTICS

Erlang Program Example 1

- Time passage in Erlang is represented as receive after n → ok end [Cesarini +, 10]
- receive $\overline{cl}$ after $v^T \rightarrow exprs$ end is not included in the reversible semantics in Erlang [Lanese+,18]

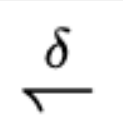
```
1  -module(time_new).
2  -export([main/0, p1/1, p2/1, p3/0]).
3
4
5  main() ->
6      Pid = spawn(?MODULE, p3, []),
7      spawn(?MODULE, p1, [Pid]),
8      spawn(?MODULE, p2, [Pid]).
9
10 p1(Pid) ->
11     receive after 2 -> ok end,
12     Pid ! {1}.
13
14 p2(Pid) ->
15     receive after 1 -> ok end,
16     Pid ! {2}.
17
18 p3() ->
19     receive
20         {N} ->
21             io:format("Received: ~p~n", [N])
22     end.
```

Lack of time passage

CONTENTS

- Background
- Time passage in the Erlang semantics
- Reversible Timed Semantics
- Integration to CauDEr
- Demonstration
- Conclusion

REVERSIBLE TIMED SEMANTICS

- We introduce discrete time passage into the reversible semantics of Erlang following transitions for time passage
-  and  represent discrete time passage
- We define transitions for time passage from the following three properties [Ulidowski+,04]
  - Maximal progress
  - Maximal time synchrony
  - Time determinacy

- Maximal progress

$$if\ \Gamma;\Pi \rightharpoonup then\ \Gamma;\Pi \overset{\delta}{\not\rightharpoonup}$$

$$if\ \Gamma;\Pi \leftharpoondown then\ \Gamma;\Pi \overset{\delta}{\not\leftharpoondown}$$

- When the system performs an action, no time passes
- In other words, time passage has the lowest priority
- In the backward discrete time passage, maximal progress holds

- Maximal time synchrony

each $X_1 \ldots X_m$ is a delayable process

$$\frac{X_1 \overset{\delta}{\rightharpoonup} X_1' \qquad X_2 \overset{\delta}{\rightharpoonup} X_2' \quad \ldots \quad X_m \overset{\delta}{\rightharpoonup} X_m' \qquad X_{m+1} \not\rightharpoonup X_{m+1}' \quad \ldots \quad X_n \not\rightharpoonup X_n'}{f(X_1, X_2, \ldots, X_n) \overset{\delta}{\rightharpoonup} f(X_1', X_2', \ldots, X_n')}$$

$$\frac{X_1 \overset{\delta}{\leftharpoonup} X_1' \qquad X_2 \overset{\delta}{\leftharpoonup} X_2' \quad \ldots \quad X_m \overset{\delta}{\leftharpoonup} X_m' \qquad X_{m+1} \not\leftharpoonup X_{m+1}' \quad \ldots \quad X_n \not\leftharpoonup X_n'}{f(X_1, X_2, \ldots, X_n) \overset{\delta}{\leftharpoonup} f(X_1', X_2', \ldots, X_n')}$$

- If all active processes advance time by the same discrete amount, then the system advances time
- In the backward discrete time passage, maximal time synchrony holds

- Time determinacy

$$if\ \Gamma;\Pi \xrightarrow{\delta} \Gamma';\Pi'\ and\ \Gamma;\Pi \xrightarrow{\delta} \Gamma'';\Pi''\ then\ \Gamma' = \Gamma''\ and\ \Pi' = \Pi''$$

$$if\ \Gamma;\Pi \xleftarrow{\delta} \Gamma';\Pi'\ and\ \Gamma;\Pi \xleftarrow{\delta} \Gamma'';\Pi''\ then\ \Gamma' = \Gamma''\ and\ \Pi' = \Pi''$$

- If the system advances time and transitions to both $\Gamma';\Pi'$ and $\Gamma";\Pi"$, then $\Gamma' = \Gamma"$ and $\Pi' = \Pi"$
- In the backward discrete time passage, time determinacy holds

The syntax for receiving messages with a timeout

receive $\overline{cl}$ after $v^T \rightarrow exprs$ end

- $v^T \rightarrow exprs$ is a timeout clause
- If no message is received for v^T milliseconds, the expression $exprs$ is executed
- v^T is defined as an integer-typed expression, or the atom infinity (= no timeout)

expression-level semantics of the receive expression with a timeout clause

$$(\theta, C[\text{receive } \overline{cl} \text{ after } v^T \rightarrow exprs \text{ end}], S) \xrightarrow{\text{rec}(\kappa, \overline{cl}, v^T \rightarrow exprs)} (\theta, \kappa, C[\_]: S)$$

- the label includes the timeout clause
- κ serves as a place holder for receive action [Lanese+, 18]

Process $< \mathrm{p}, \mathrm{h}, \theta, e, S, t >$

- t is a timer for receiving message
- The process initiated the receive action
- For untimed behavior, t is set to zero
- $\delta(d)$ represents the passage of d units of time in h

Forward Timed Semantics

$$[\textsc{Receive}]\quad \frac{\theta, e, S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl},v^T \to exprs)} \theta', e', S' \qquad \mathsf{matchrec}(\overline{cl}\theta, v) = (\theta_i, e_i)}{\begin{array}{l}\Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_r, h, \theta, e, S, t\rangle \mid \Pi \rightharpoonup \\ \quad \Gamma; \langle p_r, \mathsf{rec}(\theta, e, S, p_s, \{v, \lambda\}, t) : h, \theta'\theta_i, e'[\kappa \mapsto e_i], S', 0\rangle \mid \Pi\end{array}}$$

$$[\textsc{Timeout}]\quad \frac{\begin{array}{r}\delta(\theta, e, S) \quad \wedge\ \theta, e, S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl},v^T \to exprs)} \theta', e', S' \\ \wedge\ \mathsf{matchmes}(\overline{cl}\theta, \Gamma_p) = \varnothing\end{array}}{\Gamma; \langle p, h, \theta, e, S, v^T\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{tout}(\theta, e, S, v^T) : h, \theta', e'[\kappa \mapsto exprs], S', 0\rangle \mid \Pi}$$

$$[\textsc{Delay}]\quad \frac{\begin{array}{l}\forall\ i \leq n.\ \left[\begin{array}{l}\mathsf{Alive}(\theta_i, e_i, S_i) \Rightarrow (\mathsf{matchmes}(\overline{cl_i}\theta_i, \Gamma_{p_i}) = \varnothing \\ \quad \wedge\ (\theta_i, e_i, S_i \xrightarrow{\mathsf{rec}(\kappa_i,\overline{cl_i},v_i^T \to exprs_i)} \theta_i', e_i', S_i') \wedge\ v_i^T - t_i \geq d)\end{array}\right] \\ \quad \wedge\ d > 0\ \wedge\ \exists j \leq n. \mathsf{Alive}(\theta_j, e_j, S_j)\end{array}}{\begin{array}{l}\Gamma; \langle p_1, h_1, \theta_1, e_1, S_1, t_1\rangle \mid \ldots \mid \langle p_n, h_n, \theta_n, e_n, S_n, t_n\rangle \overset{\delta(d)}{\rightharpoonup} \\ \quad \Gamma; \langle p_1, \delta(d) : h_1, \theta_1', e_1'[\kappa_1 \mapsto \mathsf{receive}\ \overline{cl_1}\ \mathsf{after}\ v_1^T \to exprs_1\ \mathsf{end}], S_1', t_1 + d\rangle \\ \quad \mid \ldots \mid \langle p_n, \delta(d) : h_n, \theta_n', e_n'[\kappa_n \mapsto \mathsf{receive}\ \overline{cl_n}\ \mathsf{after}\ v_n^T \to exprs_n\ \mathsf{end}], S_n', t_n + d\rangle\end{array}}$$

Forward Timed Semantics : RECEIVE

$$[\textsc{Receive}] \quad \frac{\theta, e, S \xrightarrow{\mathsf{rec}(\kappa, \overline{cl}, v^T \to exprs)} \theta', e', S' \qquad \mathsf{matchrec}(\overline{cl}\theta, v) = (\theta_i, e_i)}{\Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_r, h, \theta, e, S, t\rangle \mid \Pi \longrightarrow \Gamma; \langle p_r, \mathsf{rec}(\theta, e, S, p_s, \{v, \lambda\}, t) : h, \theta'\theta_i, e'[\kappa \mapsto e_i], S', 0\rangle \mid \Pi}$$

- Some message is available in the mailbox, receive the message
- The process no longer needs to wait for time, then t is reset to 0

Forward Timed Semantics : RECEIVE

$$[\textsc{Receive}] \quad \frac{\theta, e, S \xrightarrow{\mathsf{rec}(\kappa, \overline{cl}, v^T \to exprs)} \theta', e', S' \qquad \mathsf{matchrec}(\overline{cl}\theta, v) = (\theta_i, e_i)}{\Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_r, h, \theta, e, S, t\rangle \mid \Pi \rightharpoonup \Gamma; \langle p_r, \mathsf{rec}(\theta, e, S, p_s, \{v, \lambda\}, t) : h, \theta'\theta_i, e'[\kappa \mapsto e_i], S', 0\rangle \mid \Pi}$$

- Some message is available in the mailbox, receive the message
- The process no longer needs to wait for time, then t is reset to 0

Forward Timed Semantics : RECEIVE

$$[\textsc{Receive}]\quad \frac{\theta, e, S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl},v^T \to exprs)} \theta', e', S' \qquad \mathsf{matchrec}(\overline{cl}\theta, v) = (\theta_i, e_i)}{\Gamma \cup \{(p_s, p_r, \{v, \lambda\})\}; \langle p_r, h, \theta, e, S, t\rangle \mid \Pi \rightharpoonup \Gamma; \langle p_r, \mathsf{rec}(\theta, e, S, p_s, \{v, \lambda\}, t) : h, \theta'\theta_i, e'[\kappa \mapsto e_i], S', 0\rangle \mid \Pi}$$

- Some message is available in the mailbox, receive the message
- The process no longer needs to wait for time, then t is reset to 0

Forward Timed Semantics : TIMEOUT

$$[\textsc{Timeout}] \quad \frac{\delta(\theta, e, S) \quad \wedge\, \theta, e, S \xrightarrow{\mathsf{rec}(\kappa, \overline{cl}, v^T \to exprs)} \theta', e', S' \quad \wedge\, \mathsf{matchmes}(\overline{cl}\theta, \Gamma_p) = \varnothing}{\Gamma; \langle p, h, \theta, e, S, v^T \rangle \,|\, \Pi \rightharpoonup \Gamma; \langle p, \mathsf{tout}(\theta, e, S, v^T) : h, \theta', e'[\kappa \mapsto exprs], S', 0 \rangle | \Pi}$$

- No message is available in the mailbox, and t reaches v^T, the TIMEOUT rule applies
- $\mathsf{tout}(\theta, e, S, v^T)$ is saved to the history
- The process no longer needs to wait for time, then t is reset to 0

Forward Timed Semantics : TIMEOUT

$$[\textsc{Timeout}] \quad \frac{\begin{array}{r}\delta(\theta,e,S) \quad \wedge\ \theta,e,S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl},v^T \to exprs)} \theta',e',S' \\ \wedge\ \mathsf{matchmes}(\overline{cl}\theta,\Gamma_p) = \varnothing\end{array}}{\Gamma;\langle p,h,\theta,e,S,v^T\rangle \,|\, \Pi \rightharpoonup \Gamma;\langle p,\mathsf{tout}(\theta,e,S,v^T):h,\theta',e'[\kappa \mapsto exprs],S',0\rangle|\Pi}$$

- No message is available in the mailbox, and t reaches v^T, the TIMEOUT rule applies
- $\mathsf{tout}(\theta,e,S,v^T)$ is saved to the history
- The process no longer needs to wait for time, then t is reset to 0

Forward Timed Semantics : TIMEOUT

$$[\textsc{Timeout}] \quad \frac{\begin{array}{c}\delta(\theta,e,S) \quad \wedge\ \theta,e,S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl},v^T \to exprs)} \theta',e',S' \\ \wedge\ \mathsf{matchmes}(\overline{cl}\theta,\Gamma_p) = \varnothing\end{array}}{\Gamma;\langle p,h,\theta,e,S,v^T\rangle \,|\, \Pi \rightharpoonup \Gamma;\langle p,\mathsf{tout}(\theta,e,S,v^T):h,\theta',e'[\kappa \mapsto exprs],S',0\rangle|\Pi}$$

- When a process evaluates a receive expression and no message is available in the mailbox, and t reaches v^T, the TIMEOUT rule applies
- $\mathsf{tout}(\theta,e,S,v^T)$ is saved to the history
- The process no longer needs to wait for time, then t is reset to 0

Forward Timed Semantics : TIMEOUT

$$[\textsc{Timeout}] \quad \frac{\delta(\theta, e, S) \quad \wedge\, \theta, e, S \xrightarrow{\mathsf{rec}(\kappa, \overline{cl}, v^T \to exprs)} \theta', e', S' \quad \wedge\, \mathsf{matchmes}(\overline{cl}\theta, \Gamma_p) = \varnothing}{\Gamma; \langle p, h, \theta, e, S, v^T \rangle \,|\, \Pi \rightharpoonup \Gamma; \langle p, \mathsf{tout}(\theta, e, S, v^T) : h, \theta', e'[\kappa \mapsto exprs], S', 0 \rangle | \Pi}$$

- When a process evaluates a receive expression and no message is available in the mailbox, and t reaches v^T, the TIMEOUT rule applies
- $\mathsf{tout}(\theta, e, S, v^T)$ is saved to the history
- The process no longer needs to wait for time, then t is reset to 0

Forward Timed Semantics : DELAY

When all active processes evaluate receive actions and receive no message, DELAY rule applies

$$[\textsc{Delay}]\quad \frac{\begin{array}{l}\forall\, i \le n.\ \Big[\ \mathsf{Alive}(\theta_i, e_i, S_i) \Rightarrow \big(\mathsf{matchmes}(\overline{cl_i}\theta_i, \Gamma_{p_i}) = \varnothing \\ \qquad \wedge\ (\theta_i, e_i, S_i \xrightarrow{\mathsf{rec}(\kappa_i, \overline{cl_i}, v_i^t \to exprs_i)} \theta_i', e_i', S_i') \wedge\ v_i^T - t_i \ge d\big)\ \Big] \\ \qquad \wedge\ d > 0\ \wedge\ \exists j \le n.\mathsf{Alive}(\theta_j, e_j, S_j)\end{array}}{\begin{array}{l}\Gamma; \langle p_1, h_1, \theta_1, e_1, S_1, t_1\rangle | \ldots | \langle p_n, h_n, \theta_n, e_n, S_n, t_n\rangle \xrightarrow{\delta(d)} \\ \quad \Gamma; \langle p_1, \delta(d) : h_1, \theta_1', e_1'[\kappa_1 \mapsto \mathsf{receive}\ \overline{cl_1}\ \mathsf{after}\ v_1^T \to exprs_1\ \mathsf{end}], S_1', t_1 + d\rangle \\ \quad | \ldots | \langle p_n, \delta(d) : h_n, \theta_n', e_n'[\kappa_n \mapsto \mathsf{receive}\ \overline{cl_n}\ \mathsf{after}\ v_n^T \to exprs_n\ \mathsf{end}], S_n', t_n + d\rangle\end{array}}$$

- d is saved to the history
- The timer of each process advances d; therefore, t_i becomes $t_i + \mathrm{d}$
- This rule also follows maximal time synchrony because all active processes advance time by the same discrete amount, then the system advances time

Forward Timed Semantics : DELAY

When all active processes evaluate receive actions and receive no message, DELAY rule applies

$$[\textsc{Delay}] \quad \frac{\forall\, i \le n.\left[\begin{array}{l}\mathsf{Alive}(\theta_i, e_i, S_i) \Rightarrow (\mathsf{matchmes}(\overline{cl_i}\theta_i, \Gamma_{p_i}) = \varnothing \\ \quad \wedge\ (\theta_i, e_i, S_i \xrightarrow{\mathsf{rec}(\kappa_i, \overline{cl_i}, v_i^T \to exprs_i)} \theta_i', e_i', S_i') \wedge\ v_i^T - t_i \ge d)\end{array}\right] \quad \wedge\, d > 0 \wedge \exists j \le n.\mathsf{Alive}(\theta_j, e_j, S_j)}{\begin{array}{l}\Gamma; \langle p_1, h_1, \theta_1, e_1, S_1, t_1\rangle | \ldots | \langle p_n, h_n, \theta_n, e_n, S_n, t_n\rangle \overset{\delta(d)}{\rightarrow} \\ \quad \Gamma; \langle p_1, \delta(d) : h_1, \theta_1', e_1'[\kappa_1 \mapsto \mathsf{receive}\ \overline{cl_1}\ \mathsf{after}\ v_1^T \to exprs_1\ \mathsf{end}], S_1', t_1 + d\rangle \\ \quad | \ldots | \langle p_n, \delta(d) : h_n, \theta_n', e_n'[\kappa_n \mapsto \mathsf{receive}\ \overline{cl_n}\ \mathsf{after}\ v_n^T \to exprs_n\ \mathsf{end}], S_n', t_n + d\rangle\end{array}}$$

- d is saved to the history
- The timer of each process advances d; therefore, t_i becomes $t_i + d$
- maximal time synchrony because all active processes advance time by the same discrete amount

Forward Timed Semantics : DELAY

When all active processes evaluate receive actions and receive no message, DELAY rule applies

$$[\textsc{Delay}]\quad \frac{\forall\, i \le n.\left[\begin{array}{l}\mathsf{Alive}(\theta_i, e_i, S_i) \Rightarrow (\mathsf{matchmes}(\overline{cl_i}\theta_i, \Gamma_{p_i}) = \varnothing \\ \quad \wedge\ (\theta_i, e_i, S_i \xrightarrow{\mathsf{rec}(\kappa_i, \overline{cl_i}, v_i^T \to exprs_i)} \theta_i', e_i', S_i') \wedge\ v_i^T - t_i \ge d)\end{array}\right]}{\wedge\ d > 0 \wedge \exists j \le n.\mathsf{Alive}(\theta_j, e_j, S_j)}$$

$$\Gamma; \langle p_1, h_1, \theta_1, e_1, S_1, t_1\rangle | \ldots | \langle p_n, h_n, \theta_n, e_n, S_n, t_n\rangle \xrightarrow{\delta(d)}$$
$$\Gamma; \langle p_1, \delta(d) : h_1, \theta_1', e_1'[\kappa_1 \mapsto \mathsf{receive}\ \overline{cl_1}\ \mathsf{after}\ v_1^T \to exprs_1\ \mathsf{end}], S_1', t_1 + d\rangle$$
$$| \ldots | \langle p_n, \delta(d) : h_n, \theta_n', e_n'[\kappa_n \mapsto \mathsf{receive}\ \overline{cl_n}\ \mathsf{after}\ v_n^T \to exprs_n\ \mathsf{end}], S_n', t_n + d\rangle$$

- d is saved to the history
- The timer of each process advances to $t_i + \mathrm{d}$
- maximal time synchrony because all active processes advance time by the same discrete amount

Forward Timed Semantics : DELAY

When all active processes evaluate receive actions and receive no message, DELAY rule applies

$$[\text{DELAY}]\quad \frac{\forall\, i \le n.\ \Big[\ \mathsf{Alive}(\theta_i, e_i, S_i) \Rightarrow (\mathsf{matchmes}(\overline{cl_i}\theta_i, \Gamma_{p_i}) = \varnothing \ \wedge\ (\theta_i, e_i, S_i \xrightarrow{\mathsf{rec}(\kappa_i, \overline{cl_i}, v_i^T \to exprs_i)} \theta_i', e_i', S_i') \wedge\ v_i^T - t_i \ge d)\ \Big] \quad \wedge\ d > 0\ \wedge\ \exists j \le n.\mathsf{Alive}(\theta_j, e_j, S_j)}{\Gamma; \langle p_1, h_1, \theta_1, e_1, S_1, t_1\rangle | \ldots | \langle p_n, h_n, \theta_n, e_n, S_n, t_n\rangle \overset{\delta(d)}{\rightharpoonup} \Gamma; \langle p_1, \delta(d) : h_1, \theta_1', e_1'[\kappa_1 \mapsto \mathsf{receive}\ \overline{cl_1}\ \mathsf{after}\ v_1^T \to exprs_1\ \mathsf{end}], S_1', t_1 + d\rangle | \ldots | \langle p_n, \delta(d) : h_n, \theta_n', e_n'[\kappa_n \mapsto \mathsf{receive}\ \overline{cl_n}\ \mathsf{after}\ v_n^T \to exprs_n\ \mathsf{end}], S_n', t_n + d\rangle}$$

- d is saved to the history
- The timer of each process advances to $t_i + \mathrm{d}$
- maximal time synchrony because all active processes advance time by the same discrete amount

Backward Timed Semantics

$$[\underline{\text{RECEIVE}}] \quad \Gamma; \langle p_r, \mathsf{rec}(\theta, e, S, p_s, \{v, \lambda\}, v_T) : h, \theta', e', S', 0\rangle \mid \Pi \leftharpoonup \Gamma \cup (p_s, p_r, \{v, \lambda\}); \langle p_r, h, \theta, e, S, v_T\rangle \mid \Pi$$

$$[\underline{\text{TIMEOUT}}] \quad \Gamma; \langle p, \mathsf{tout}(\theta, e, S, v_T) : h, \theta', e', S', 0\rangle \mid \Pi \leftharpoonup \Gamma; \langle p, h, \theta, e, S, v_T\rangle \mid \Pi$$

$$[\underline{\text{DELAY}}] \quad \Gamma; \langle p_1, \delta(d) : h_1, \theta_1, e_1, S'_1, v_1\rangle \mid \ldots \mid \langle p_n, \delta(d) : h_n, \theta_n, e_n, S'_n, v_n\rangle \overset{\delta(d)}{\leftharpoonup} \Gamma; \langle p_1, h_1, \theta_1, e_1, S_1, v_1 - d\rangle \mid \ldots \mid \langle p_n, h_n, \theta_n, e_n, S_n, v_n - d\rangle$$

Backward Timed Semantics

$$[\text{RECEIVE}] \quad \Gamma; \langle p_r, \text{rec}(\theta, e, S, p_s, \{v, \lambda\}, v_T) : h, \theta', e', S', 0\rangle \mid \Pi \leftarrow \Gamma \cup (p_s, p_r, \{v, \lambda\}); \langle p_r, h, \theta, e, S, v_T\rangle \mid \Pi$$

- When rec(θ, e, S, $p_s, \{v, \lambda\}, v^T$) appears at the top of h, the backward RECEIVE rule applies
- The process restores θ, e, S and t stored in the history
- Γ restores $(p_s, p_r, \{v, \lambda\})$ stored in the history
- The history is popped form h

Backward Timed Semantics

$$[\underline{\text{RECEIVE}}] \quad \Gamma; \langle p_r, \text{rec}(\theta, e, S, p_s, \{v, \lambda\}, v_T) : h, \theta', e', S', 0\rangle \mid \Pi \leftharpoondown \Gamma \cup (p_s, p_r, \{v, \lambda\}); \langle p_r, h, \theta, e, S, v_T\rangle \mid \Pi$$

- When rec(θ, e, S, $p_s, \{v, \lambda\}, v^T$) appears at the top of h, the backward RECEIVE rule applies
- The process restores θ, e, S and t stored in the history
- Γ restores $(p_s, p_r, \{v, \lambda\})$ stored in the history
- The history is popped form h

Backward Timed Semantics

$$[\underline{\text{RECEIVE}}] \quad \Gamma; \langle p_r, \text{rec}(\theta, e, S, p_s, \{v, \lambda\}, v_T) : h, \theta', e', S', 0\rangle \mid \Pi \leftharpoonup \Gamma \cup (p_s, p_r, \{v, \lambda\}); \langle p_r, h, \theta, e, S, v_T\rangle \mid \Pi$$

- When rec(θ, e, S, $p_s, \{v, \lambda\}, v^T$) appears at the top of h, the backward RECEIVE rule applies
- The process restores θ, e, S and t stored in the history
- Γ restores $(p_s, p_r, \{v, \lambda\})$ stored in the history
- The history is popped form h

Backward Timed Semantics

$$[\underline{\text{TIMEOUT}}] \quad \Gamma; \langle p, \text{tout}(\theta, e, S, v_T) : h, \theta', e', S', 0 \rangle \mid \Pi \leftarrow \Gamma; \langle p, h, \theta, e, S, v_T \rangle \mid \Pi$$

- When tout(θ, e, S, v^T) appears at the top of h, the backward TIMEOUT rule applies
- The process restores θ, e, S and t stored in h
- The history is popped form h

Backward Timed Semantics

$$[\underline{\text{TIMEOUT}}] \quad \Gamma; \langle p, \text{tout}(\theta, e, S, v_T) : h, \theta', e', S', 0 \rangle \mid \Pi \leftarrow \Gamma; \langle p, h, \theta, e, S, v_T \rangle \mid \Pi$$

- When tout(θ, e, S, v^T) appears at the top of h, the backward TIMEOUT rule applies
- The process restores θ, e, S and t stored in h
- The history is popped form h

Backward Timed Semantics

$$[\textsc{Timeout}] \quad \Gamma; \langle p, \mathsf{tout}(\theta, e, S, v_T) : h, \theta', e', S', 0\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, \theta, e, S, v_T\rangle \mid \Pi$$

- When tout(θ, e, S, v^T) appears at the top of h, the backward TIMEOUT rule applies
- The process restores θ, e, S and t stored in h
- The history is popped form h

Backward Timed Semantics

$$[\text{DELAY}] \quad \Gamma;\ \langle p_1, \delta(d) : h_1, \theta_1, e_1, S_1', v_1\rangle | \ldots | \langle p_n, \delta(d) : h_n, \theta_n, e_n, S_n', v_n\rangle$$
$$\overset{\delta(d)}{\leftarrow} \Gamma;\ \langle p_1, h_1, \theta_1, e_1, S_1, v_1 - d\rangle | \ldots | \langle p_n, h_n, \theta_n, e_n, S_n, v_n - d\rangle$$

- When $\delta(\mathrm{d})$ appears at the top of h for a process, all other processes also have $\delta(\mathrm{d})$ at the top of h and the backward DELAY rule applies
- The process restores $v_i - \mathrm{d}$ where d is stored in h
- The history form all h in Π is popped

Backward Timed Semantics

$$[\underline{\text{DELAY}}] \quad \Gamma;\ \langle p_1, \delta(d) : h_1, \theta_1, e_1, S'_1, v_1\rangle | \ldots | \langle p_n, \delta(d) : h_n, \theta_n, e_n, S'_n, v_n\rangle$$
$$\overset{\delta(d)}{\leftarrow} \Gamma;\ \langle p_1, h_1, \theta_1, e_1, S_1, v_1 - d\rangle | \ldots | \langle p_n, h_n, \theta_n, e_n, S_n, v_n - d\rangle$$

- When δ(d) appears at the top of h for a process, all other processes also have δ(d) at the top of h and the backward DELAY rule applies
- The process restores $v_i - \mathrm{d}$ where d is stored in h
- The history form all h in Π is popped

Backward Timed Semantics

$$[\underline{\text{DELAY}}] \quad \Gamma;\ \langle p_1, \delta(d) : h_1, \theta_1, e_1, S_1', v_1\rangle | \ldots | \langle p_n, \delta(d) : h_n, \theta_n, e_n, S_n', v_n\rangle$$
$$\overset{\delta(d)}{\leftarrow} \Gamma;\ \langle p_1, h_1, \theta_1, e_1, S_1, v_1 - d\rangle | \ldots | \langle p_n, h_n, \theta_n, e_n, S_n, v_n - d\rangle$$

- When δ(d) appears at the top of h for a process, all other processes also have δ(d) at the top of h and the backward DELAY rule applies
- The process restores v_i – d where d is stored in h
- The history form all h in Π is popped

REVERSIBLE TIMED SEMANTICS

- loop lemma holds for time passage
- Time additivity

$$if\ \Gamma;\Pi \overset{\delta(d_1)}{\rightharpoonup} \Gamma';\Pi'\ and\ \Gamma';\Pi' \overset{\delta(d_2)}{\rightharpoonup} \Gamma'';\Pi''\ then\ \Gamma;\Pi \overset{\delta(d_1+d_2)}{\rightharpoonup} \Gamma'';\Pi''$$

Due to loop lemma,

$$if\ \Gamma';\Pi' \overset{\delta(d_1)}{\leftharpoonup} \Gamma;\Pi\ and\ \Gamma'';\Pi'' \overset{\delta(d_2)}{\leftharpoonup} \Gamma';\Pi'\ then\ \Gamma'';\Pi'' \overset{\delta(d_1+d_2)}{\leftharpoonup} \Gamma;\Pi$$

CONTENTS

- Background
- Time passage in the Erlang semantics
- Reversible Timed Semantics
- Integration to CauDEr
- Demonstration
- Conclusion

$$\Gamma;\Pi \xrightarrow{\delta(d)} \Gamma;\Pi'$$

Π does no action until some process timeout
→ choose d for next timeout

- d is the shortest waiting time of all processes
- If all processes become inactive or receiving with $v^T = infinity$, no further time passage occurs

INTEGRATION TO CAUDER

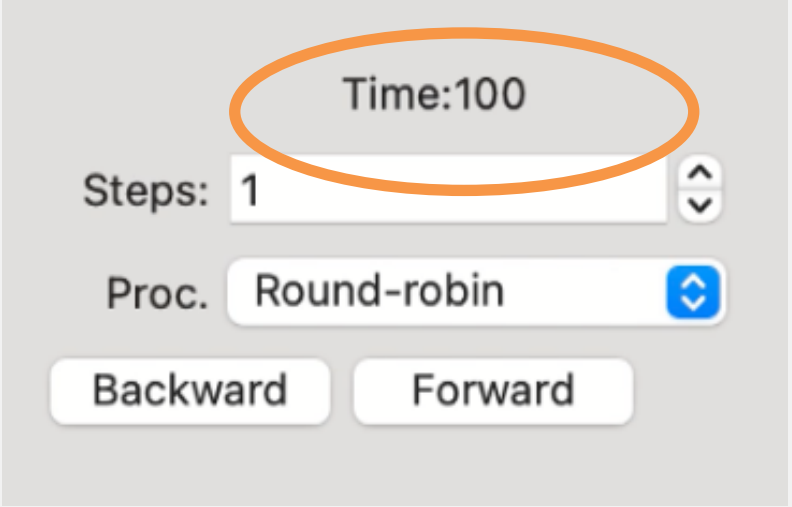
① Process status icon

New ! • Added 🕐 which means the process is waiting for a time passage

New ! • Added ❗ which means the process is ready to time out

② Global clock

New ! • Added a GUI to display elapsed time (not in the proceedings)



INTEGRATION TO CAUDER

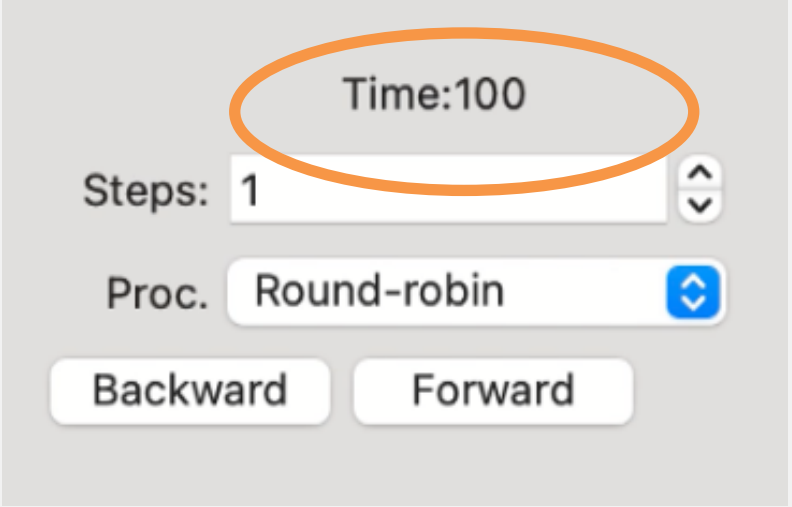
① Process status icon

New ! • Added 🕐 which means the process is waiting for a time passage

New ! • Added ❗ which means the process is ready to time out

② Global clock

New ! • Added a GUI to display elapsed time (not in the proceedings)



INTEGRATION TO CAUDER

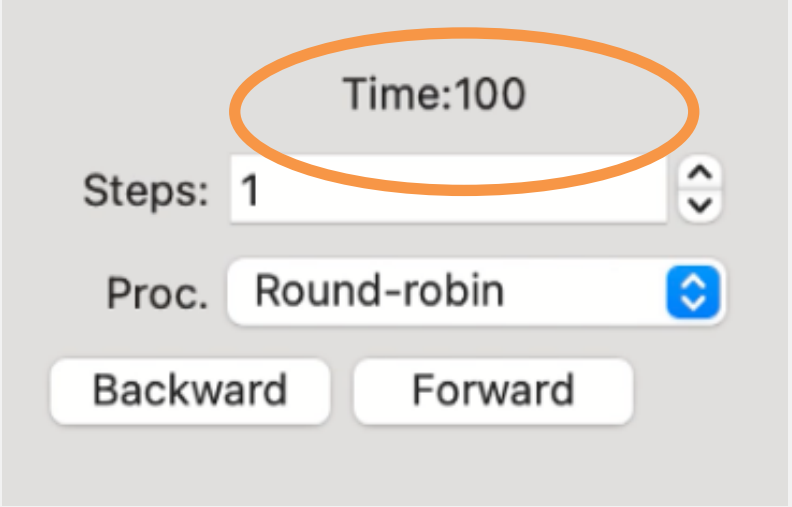
① Process status icon

New ! • Added 🕐 which means the process is waiting for a time passage

New ! • Added ❗ which means the process is ready to time out

② Global clock

New ! • Added a GUI to display elapsed time (not in the proceedings)

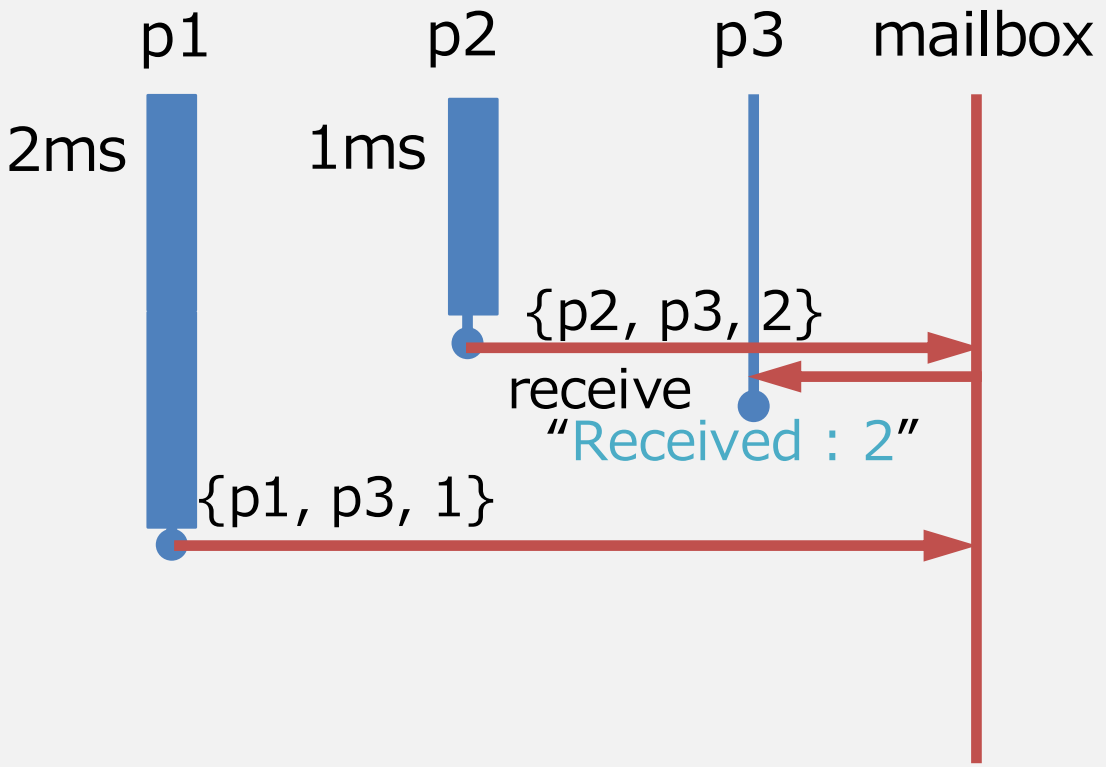


CONTENTS

- Background
- Time passage in the Erlang semantics
- Reversible Timed Semantics
- Integration to CauDEr
- Demonstration
- Conclusion

TIME PASSAGE IN THE ERLANG SEMANTICS

Erlang Program Example I

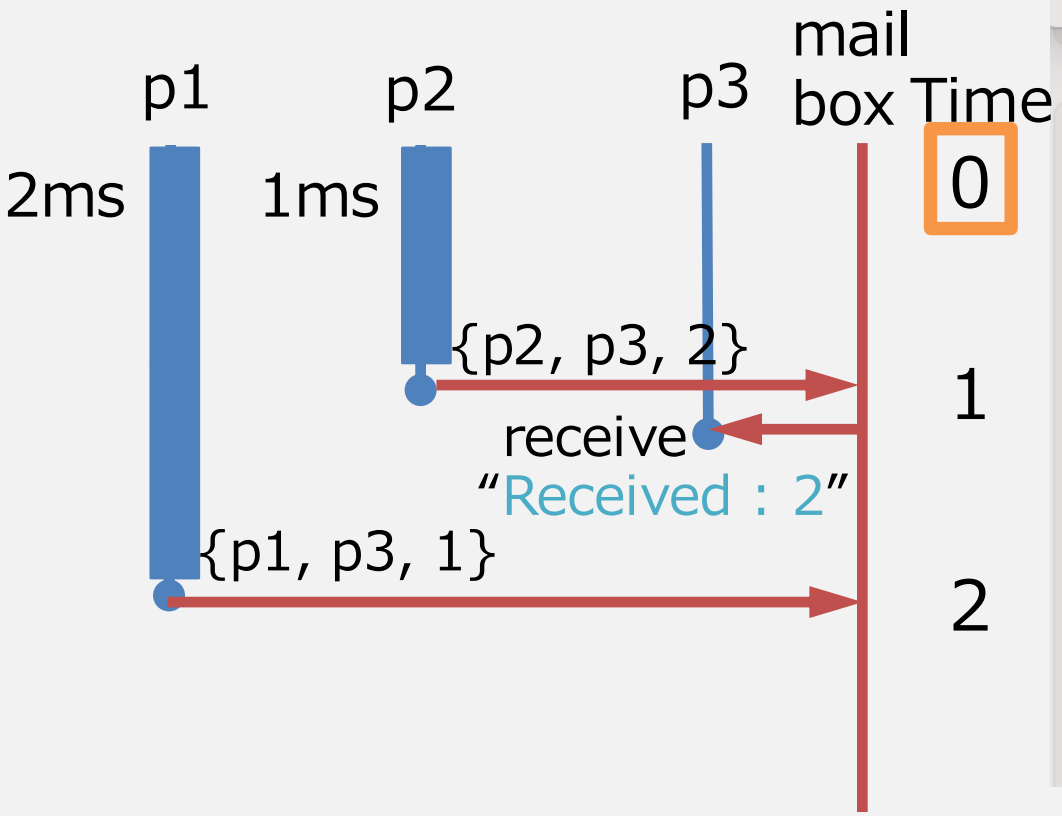


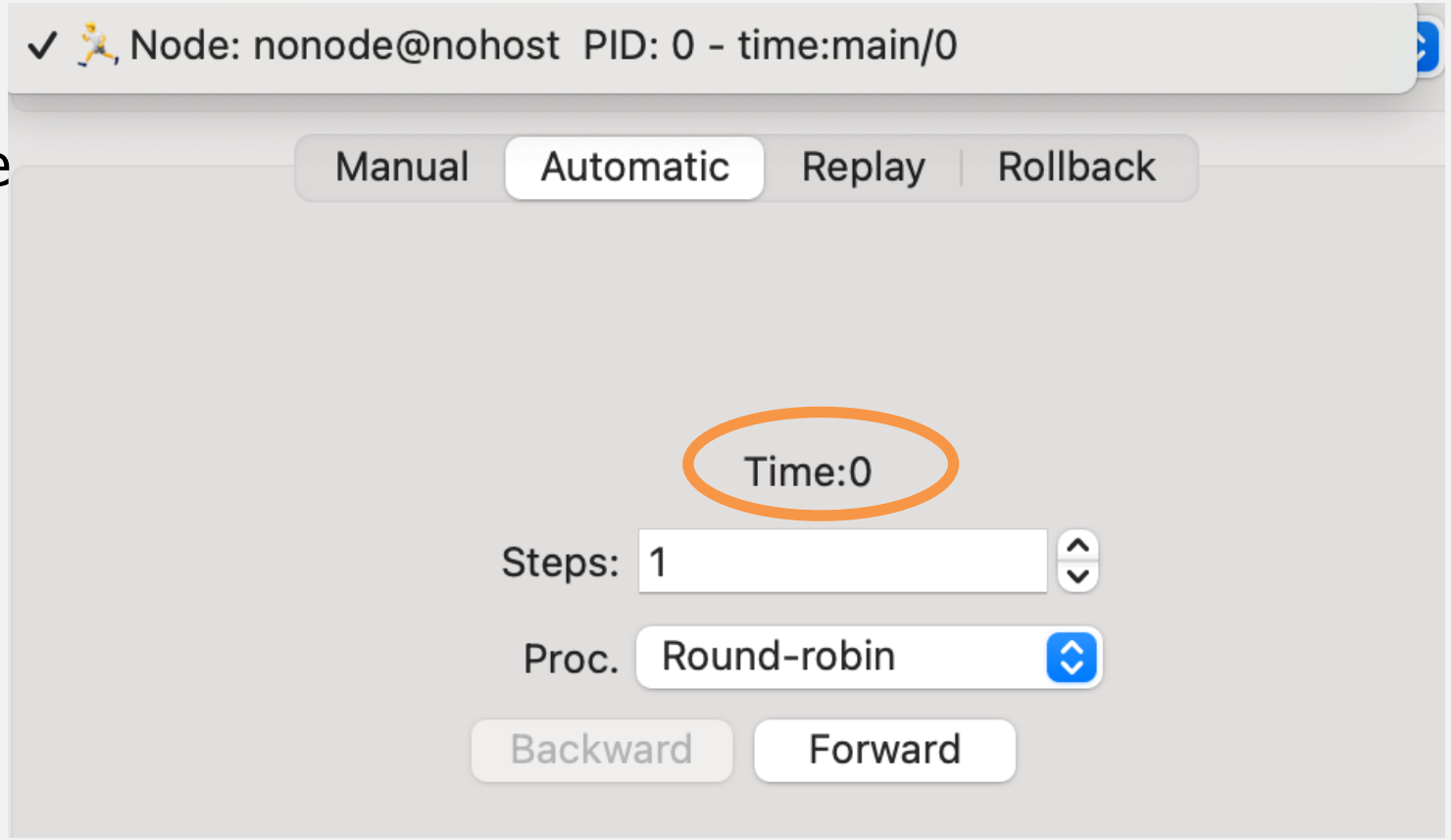

```
1   -module(time_new).
2   -export([main/0, p1/1, p2/1, p3/0]).
3
4
5   main() ->
6       Pid = spawn(?MODULE, p3, []),
7       spawn(?MODULE, p1, [Pid]),
8       spawn(?MODULE, p2, [Pid]).
9
10  p1(Pid) ->
11      receive after 2 -> ok end,
12      Pid ! {1}.
13
14  p2(Pid) ->
15      receive after 1 -> ok end,
16      Pid ! {2}.
17
18  p3() ->
19      receive
20          {N} ->
21              io:format("Received: ~p~n", [N])
22      end.
```

DEMONSTRATION

Forward Execution

- Initial state



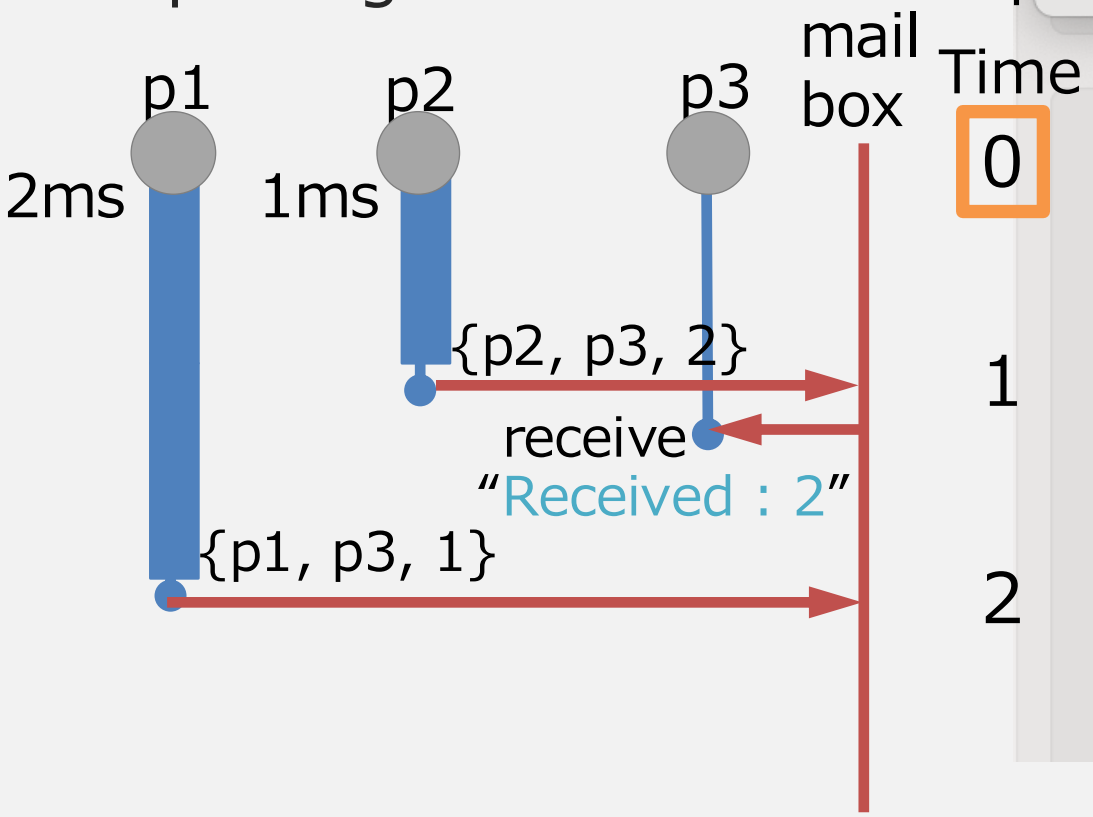


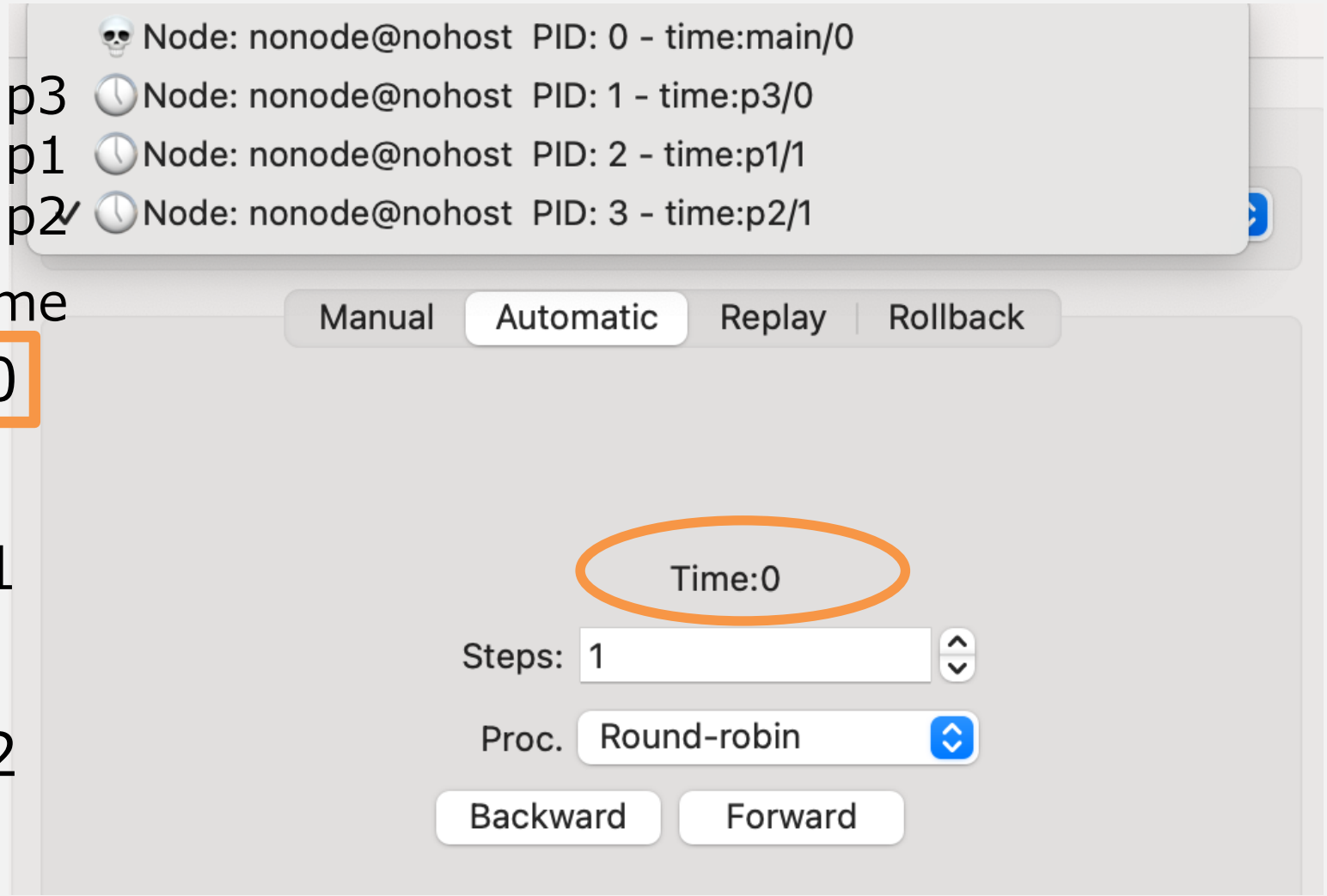
Start of Forward execution

DEMONSTRATION

Forward Execution

- All processes wait for time passage

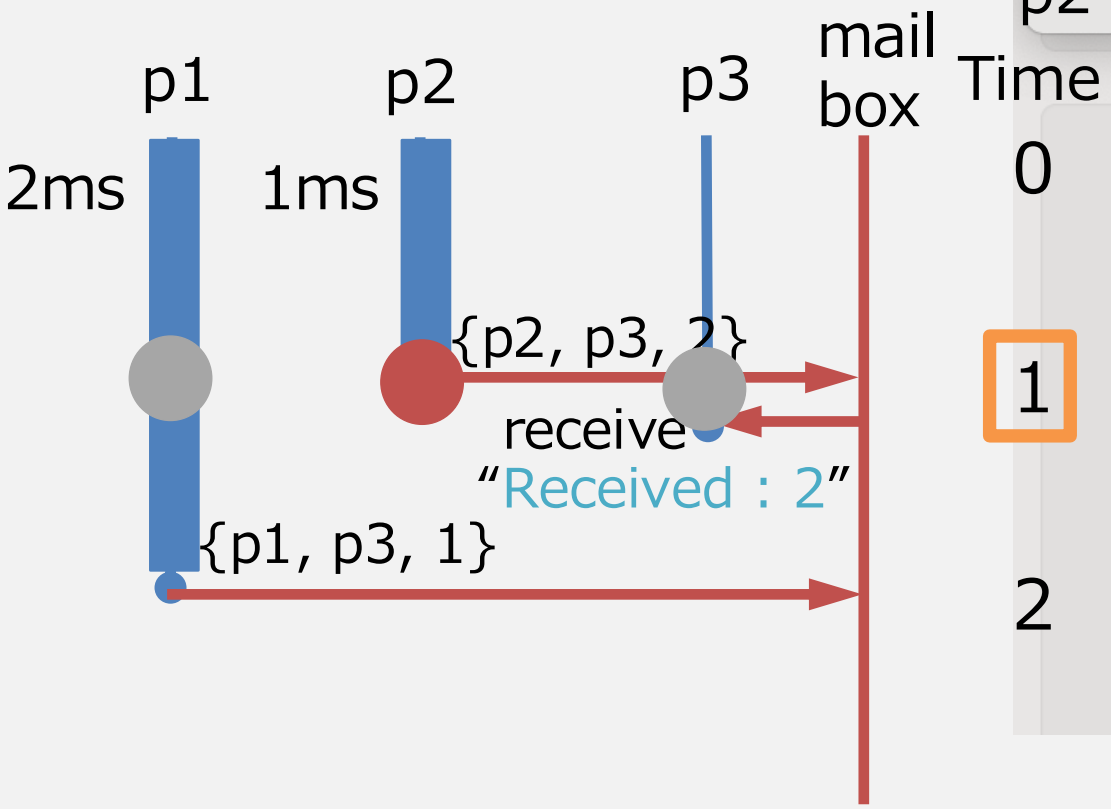


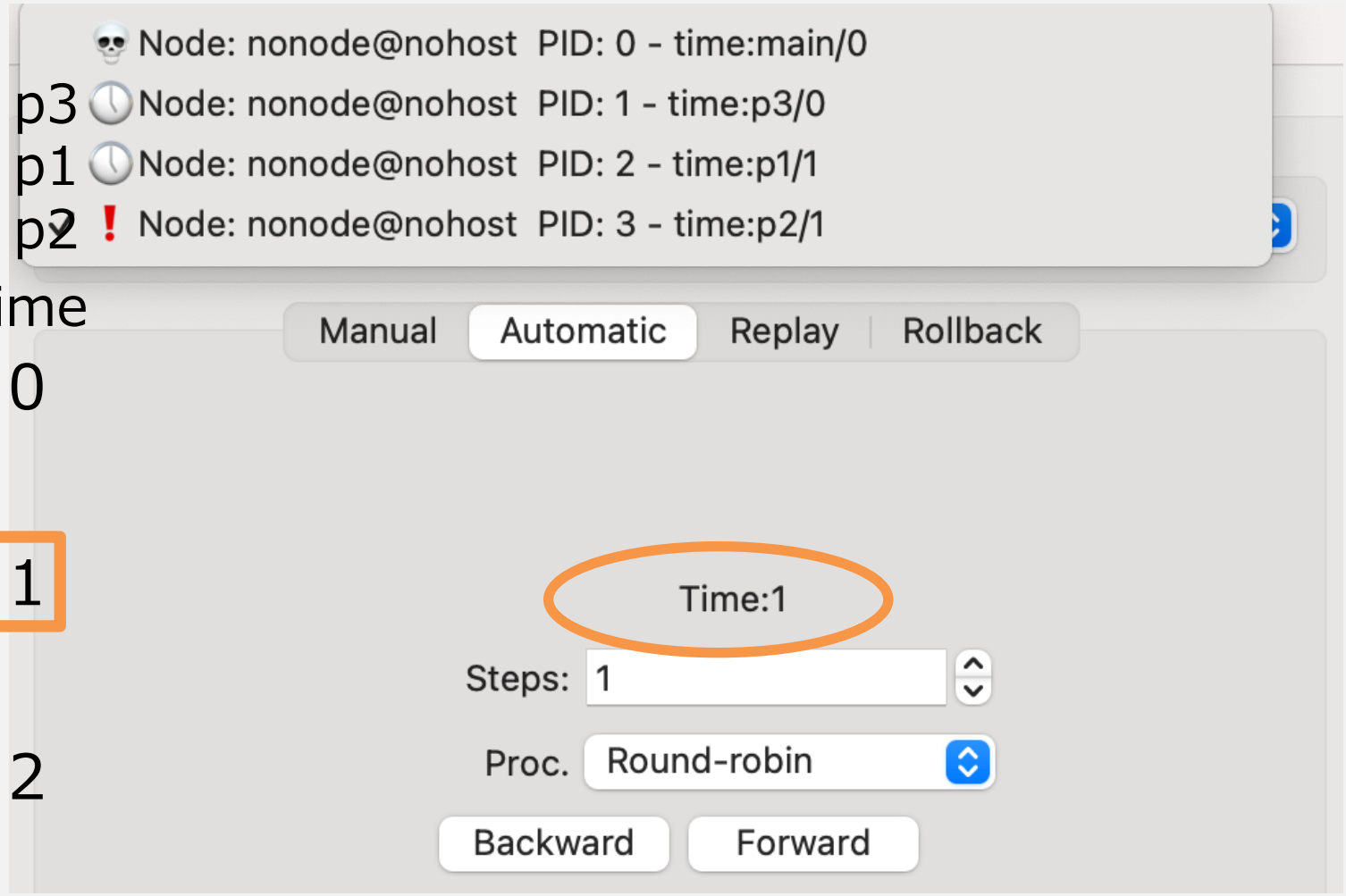


DEMONSTRATION

Forward Execution

- GUI changes to 'Time: 1’

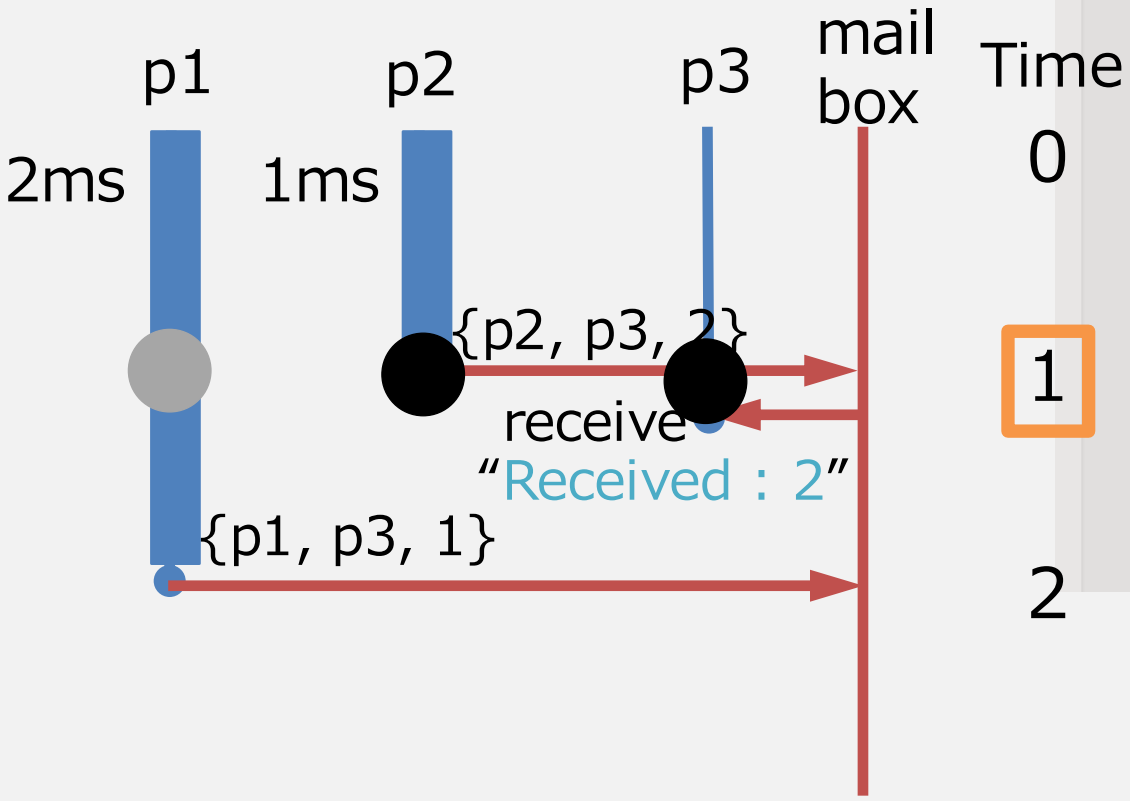


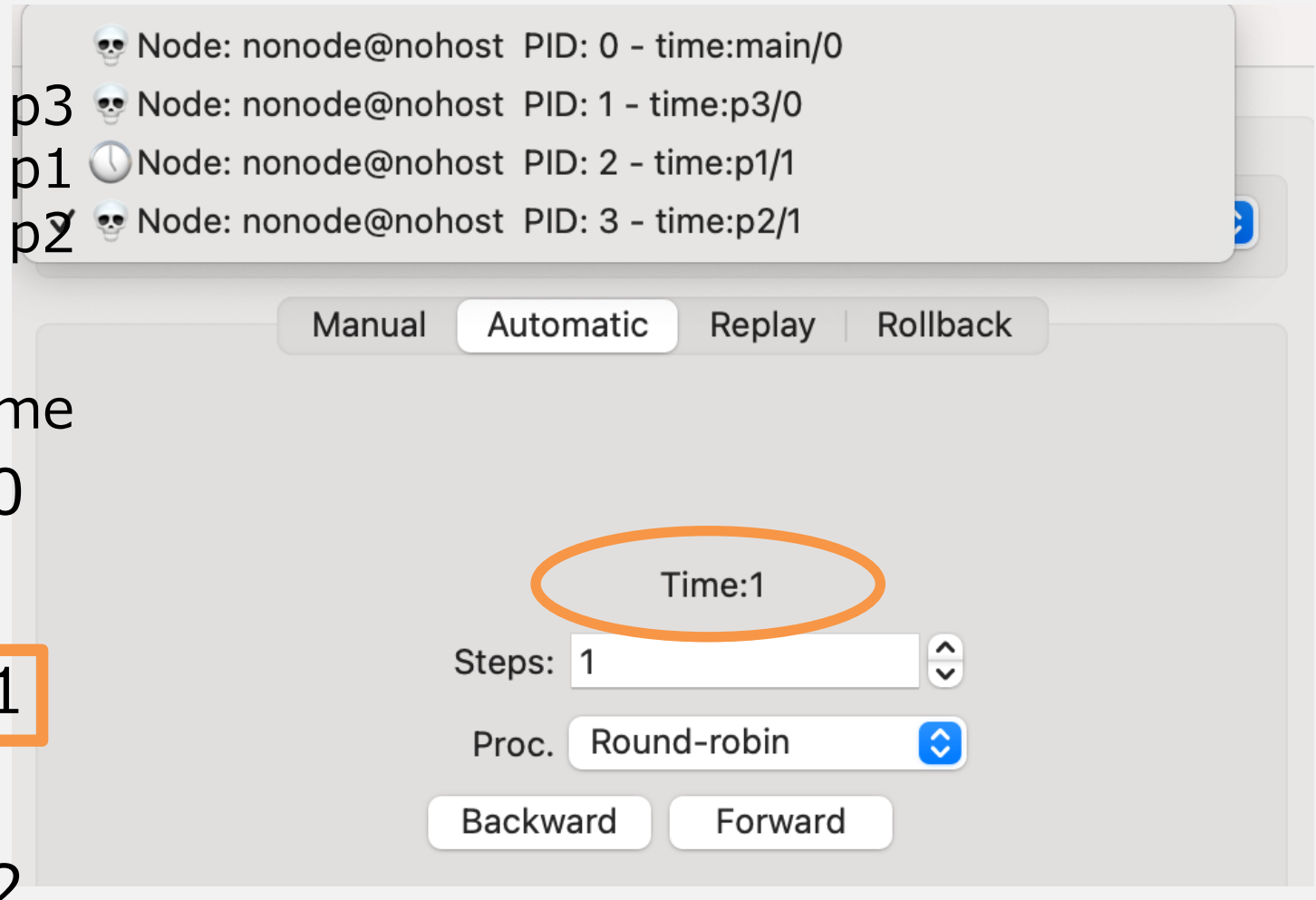


DEMONSTRATION

Forward Execution

- p2 and p3 are inactive
- p1 waits for time passage

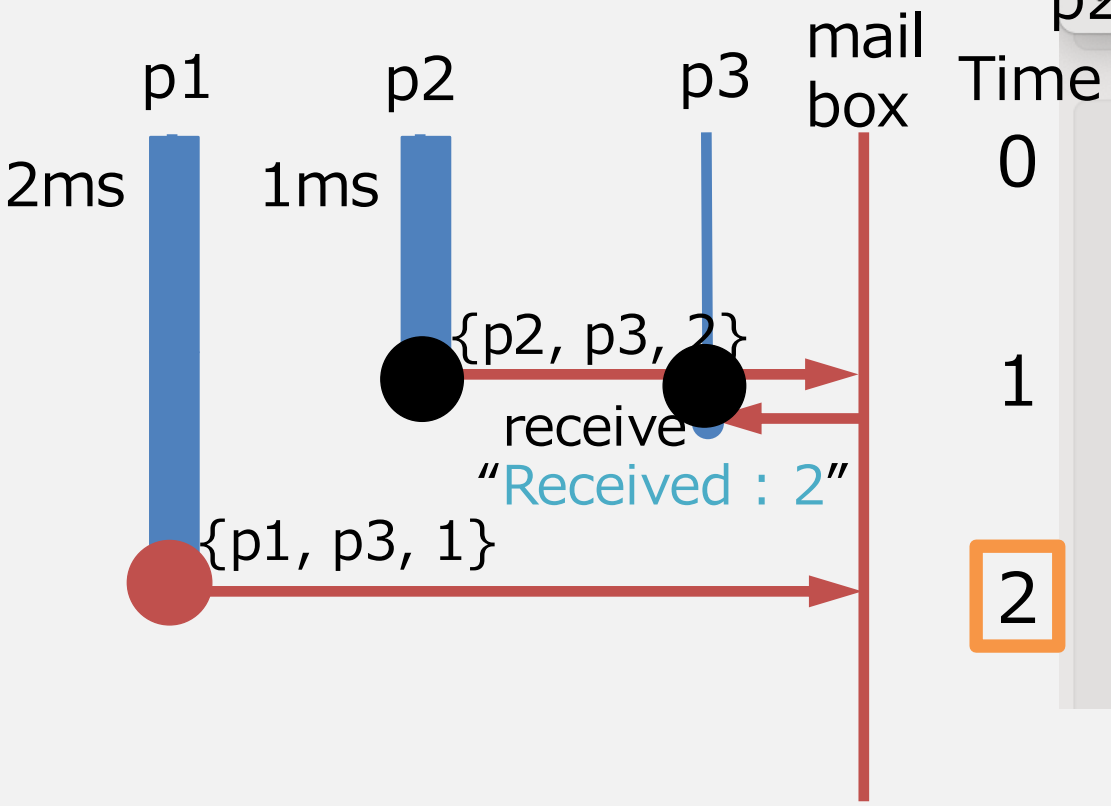


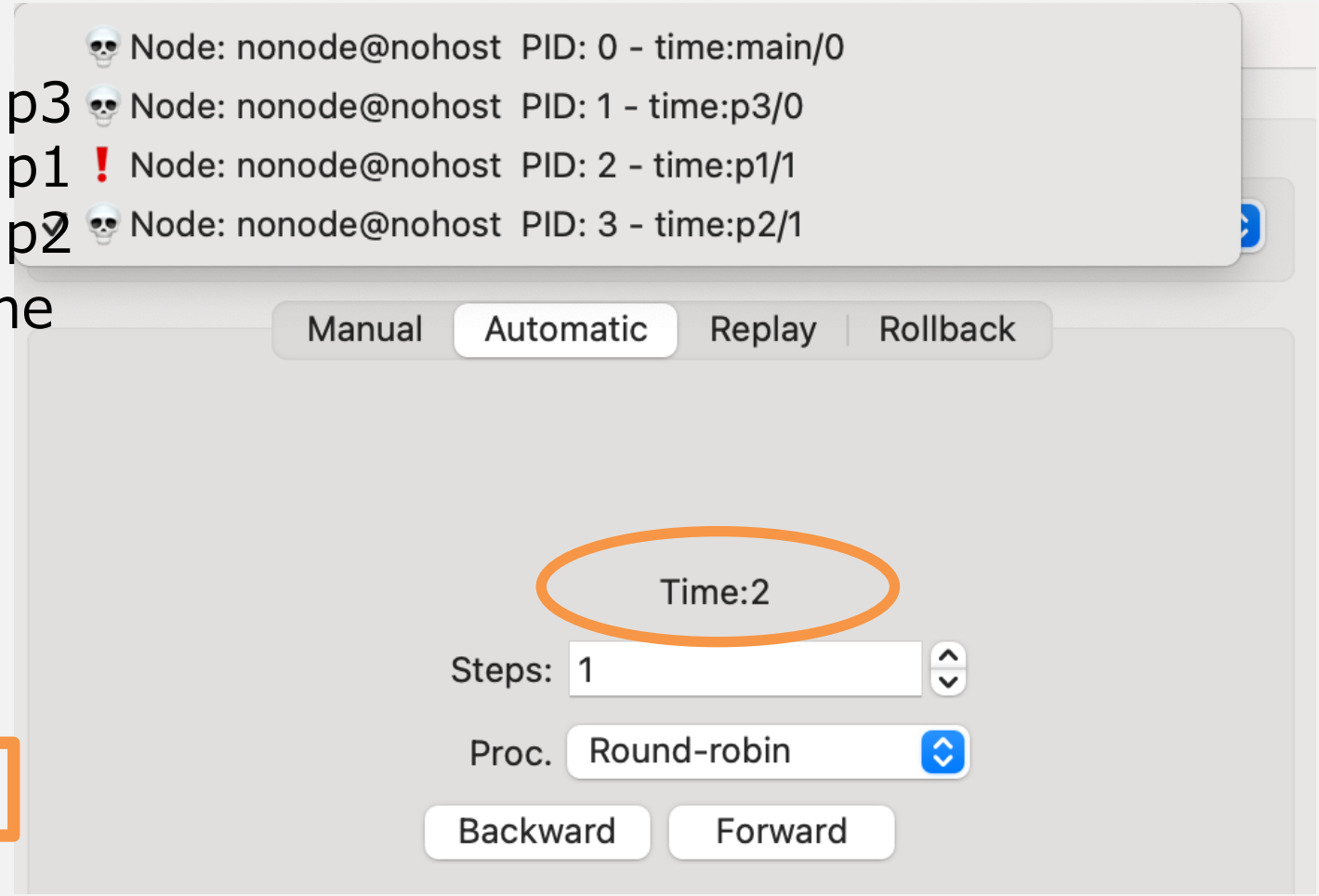


DEMONSTRATION

Forward Execution

- GUI changes to 'Time: 2’

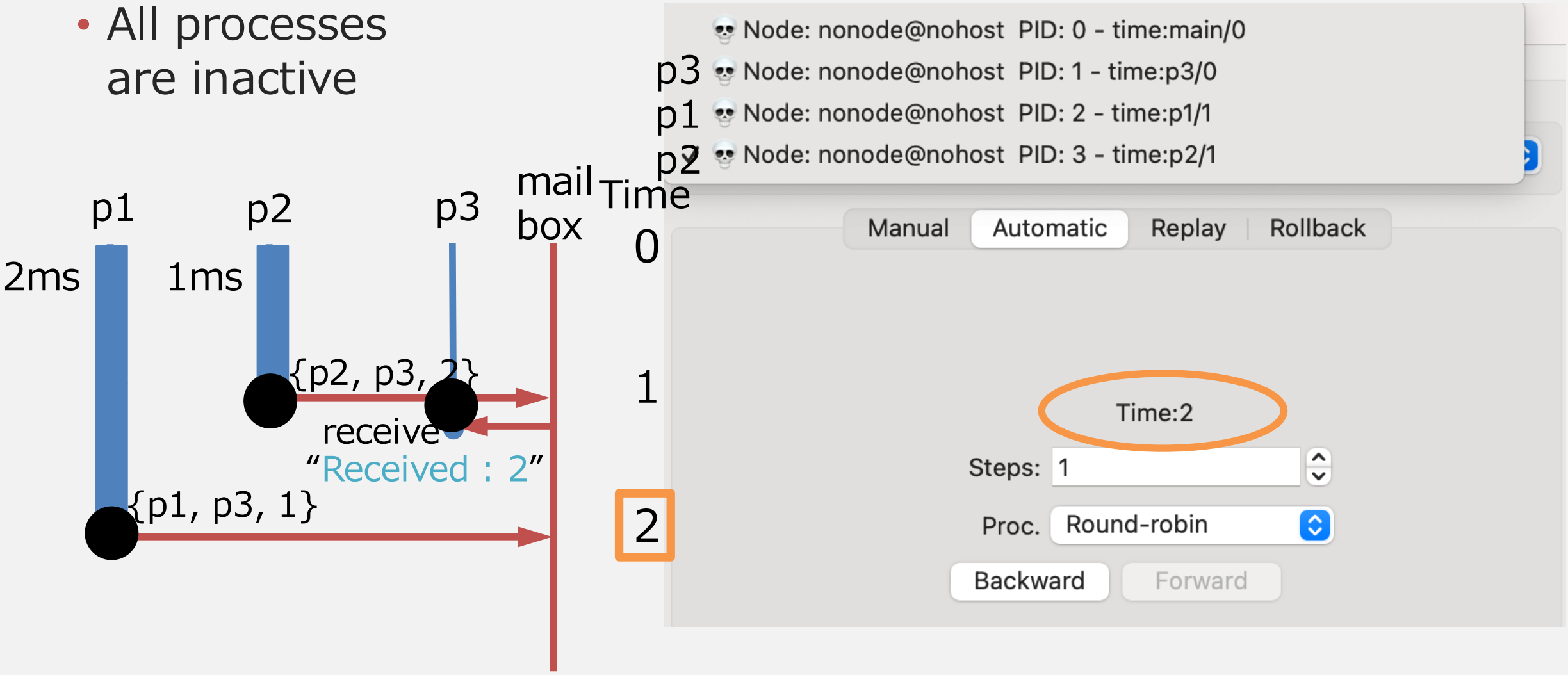




DEMONSTRATION

Forward Execution

- All processes are inactive

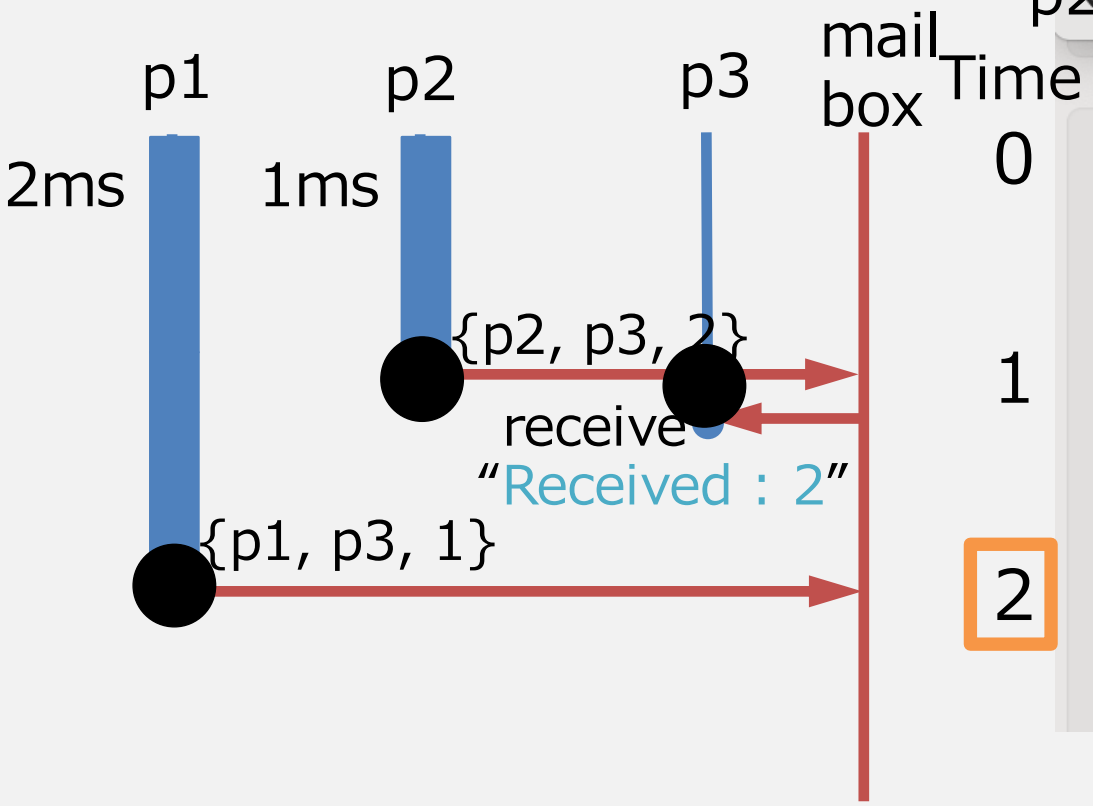


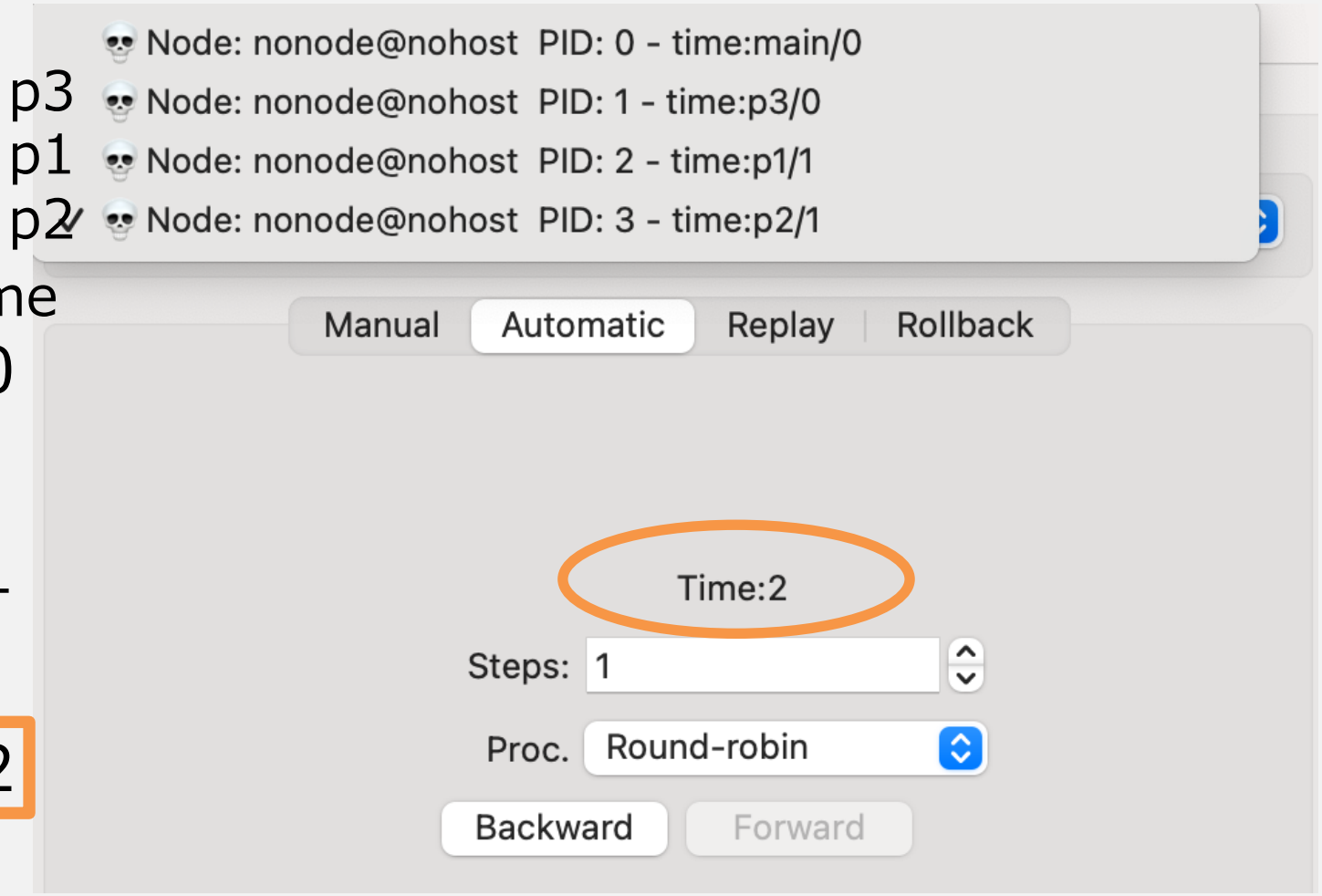
End of Forward execution

DEMONSTRATION

Backward Execution

- All processes are inactive



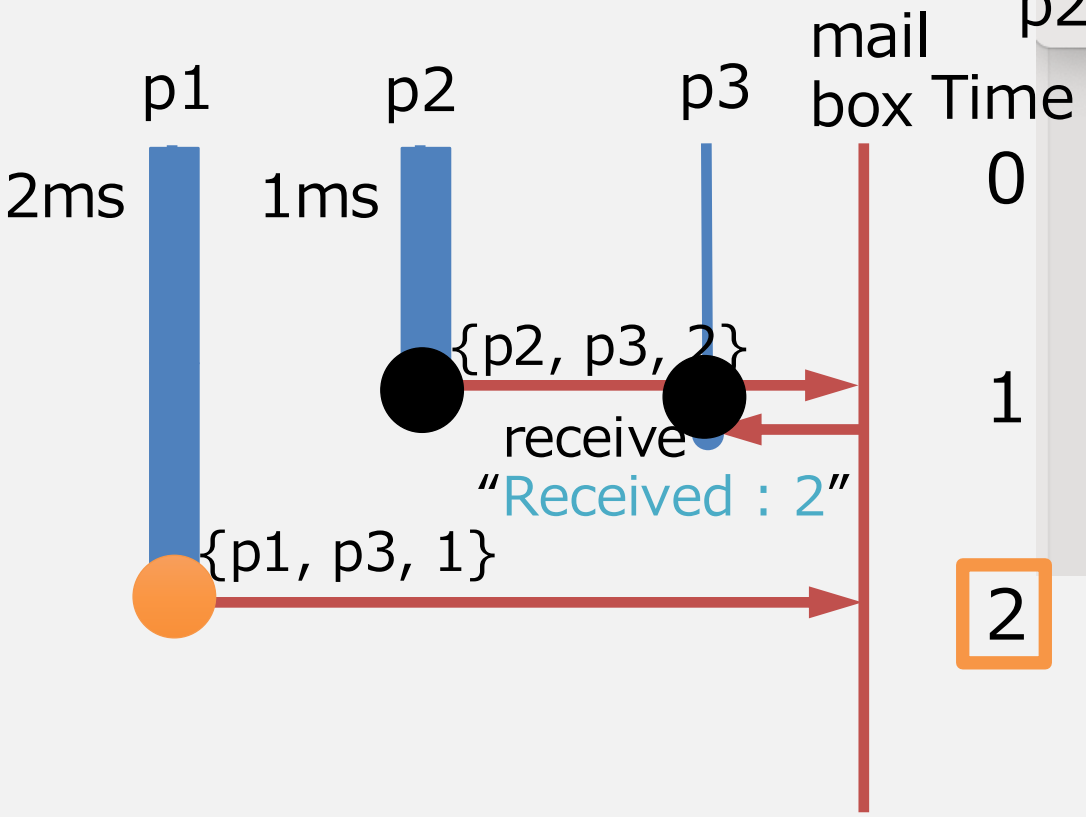


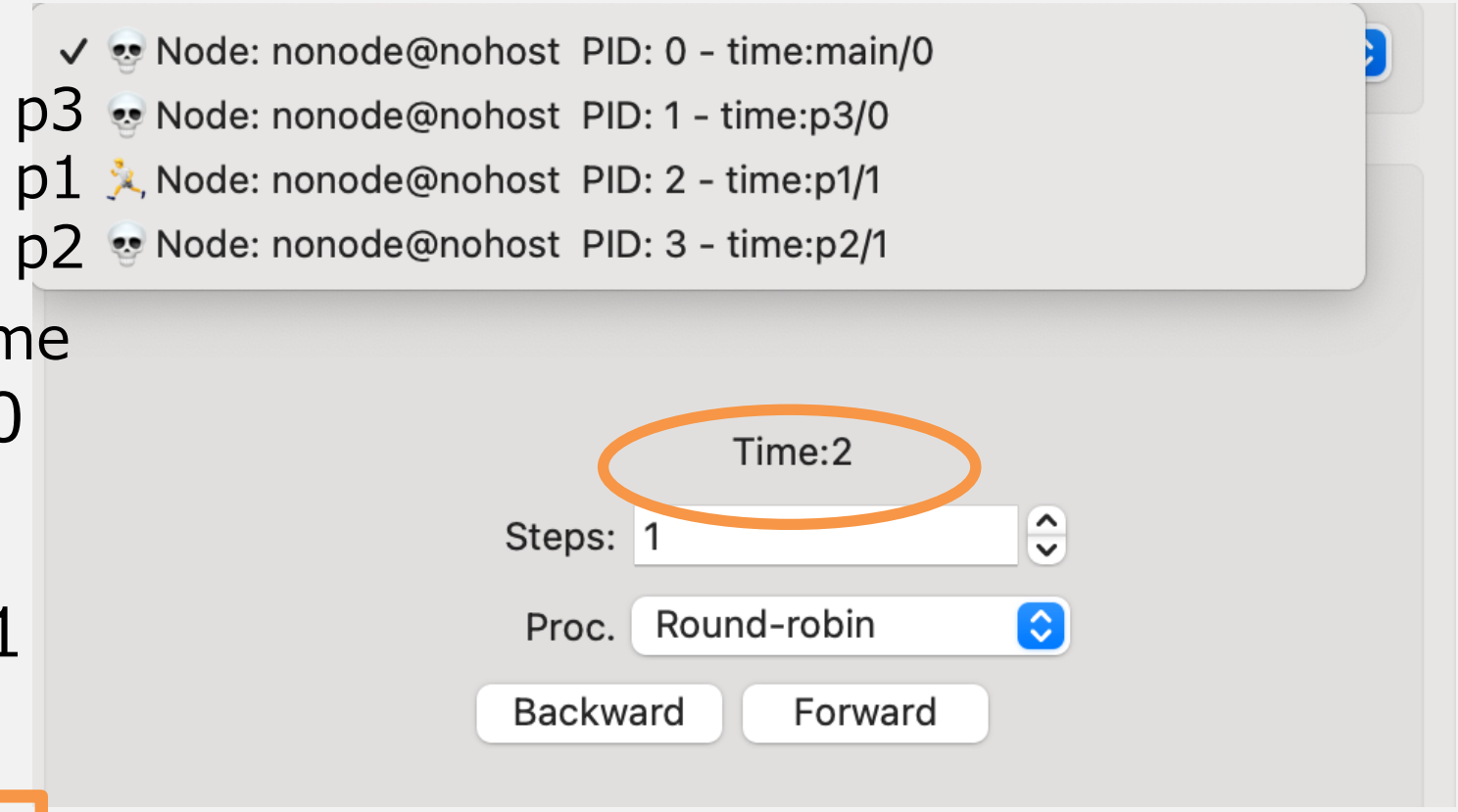
Start of Backward execution

DEMONSTRATION

Backward Execution

- p1 wakes up

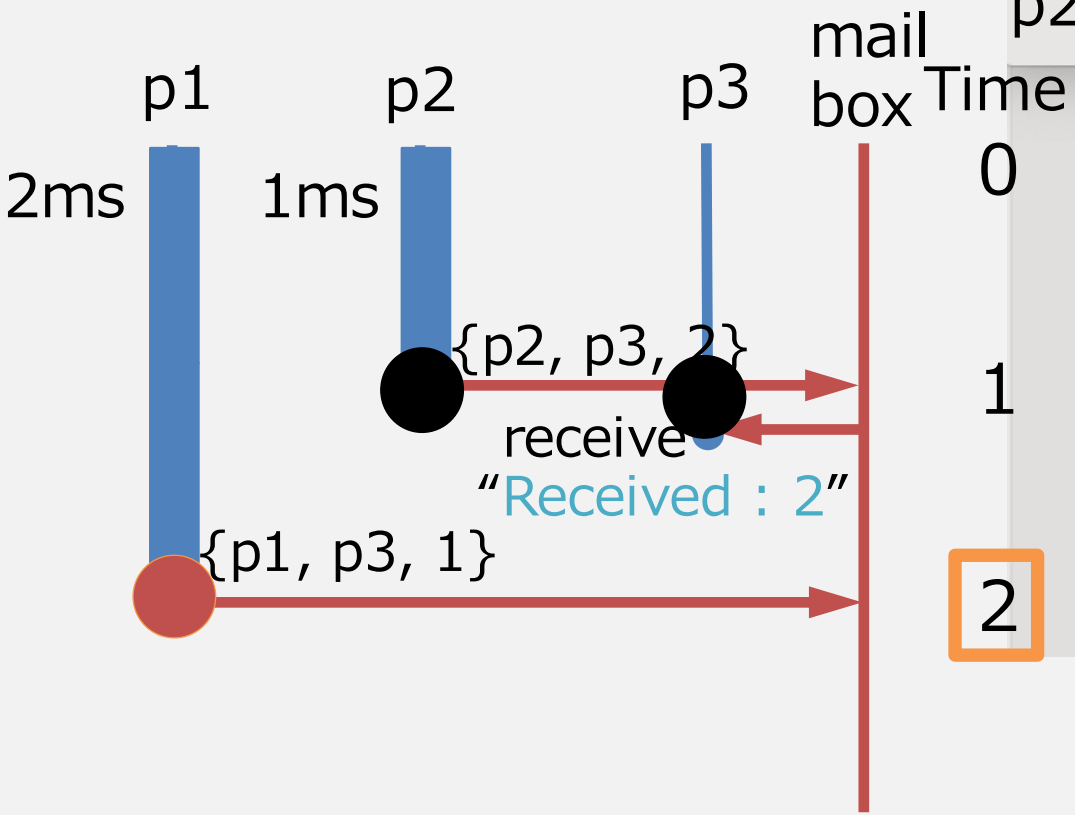


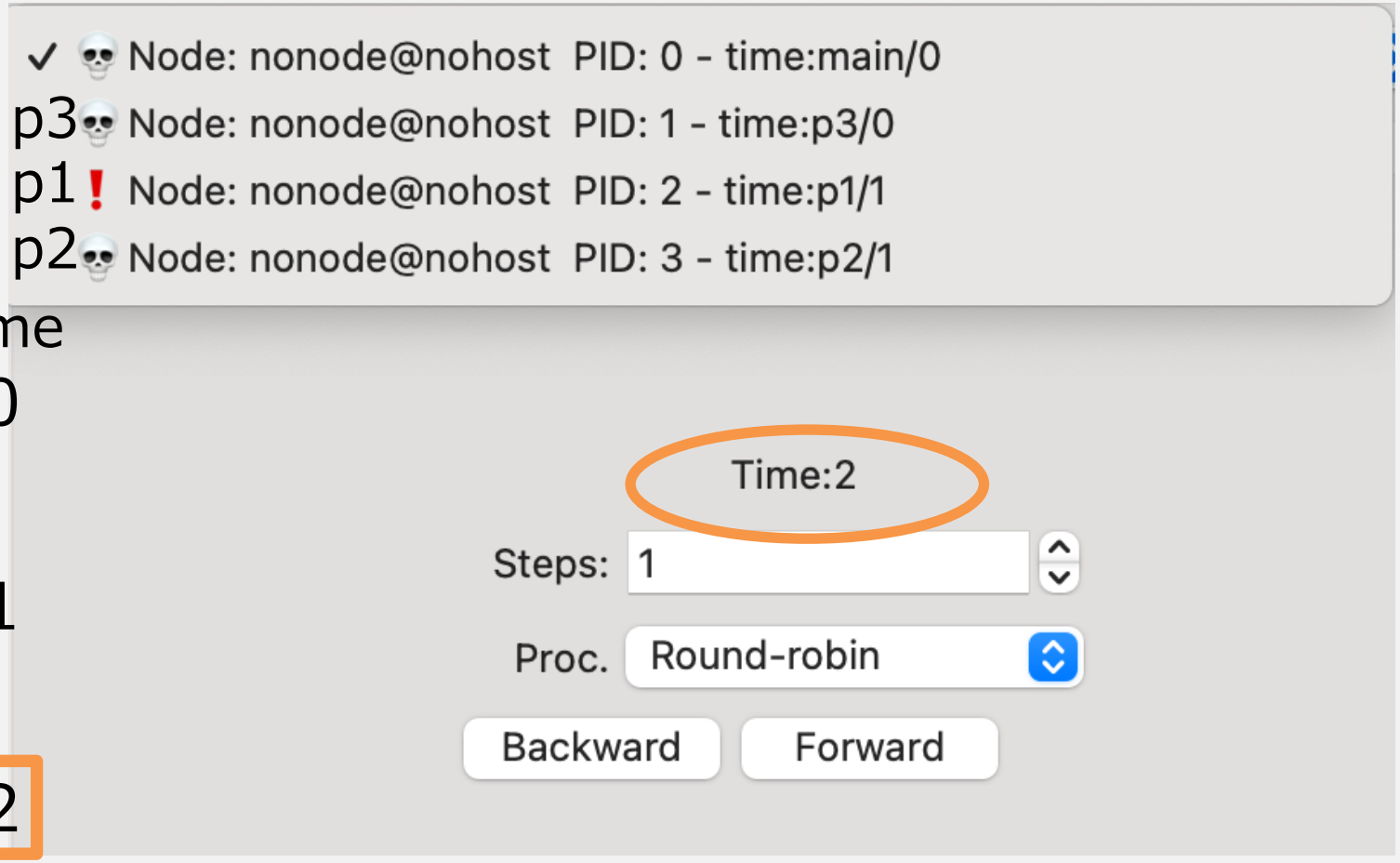


DEMONSTRATION

Backward Execution

- The top of history in p1 is timeout

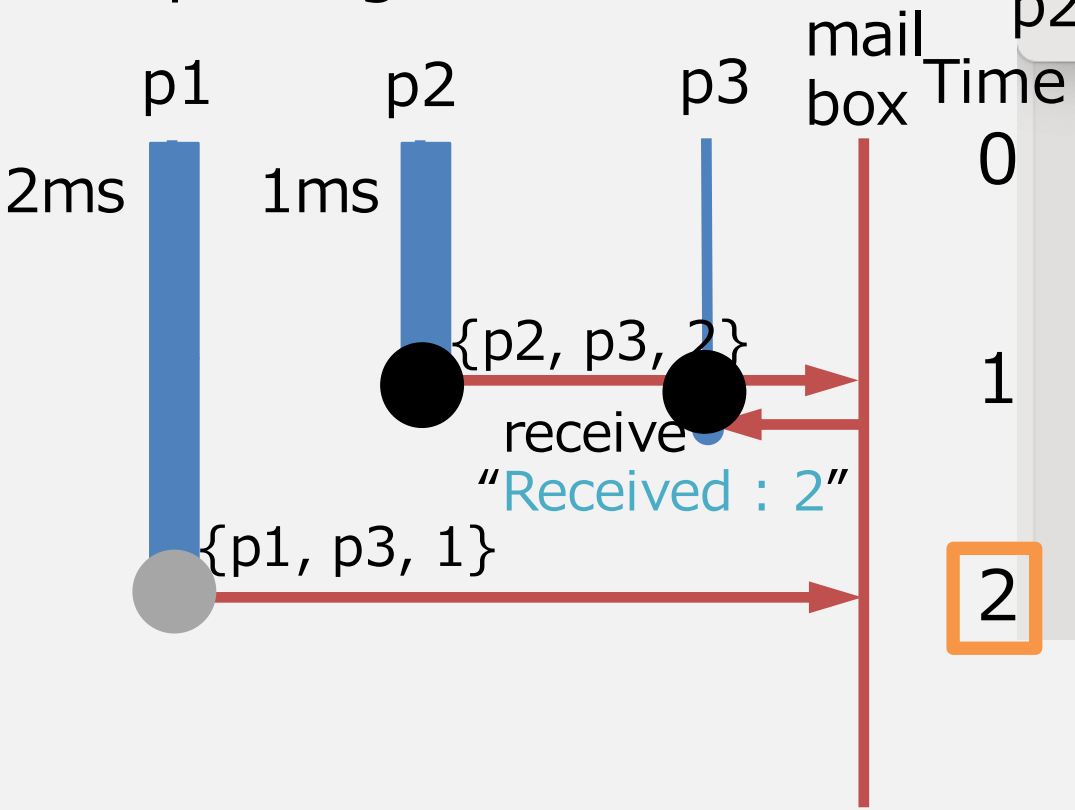


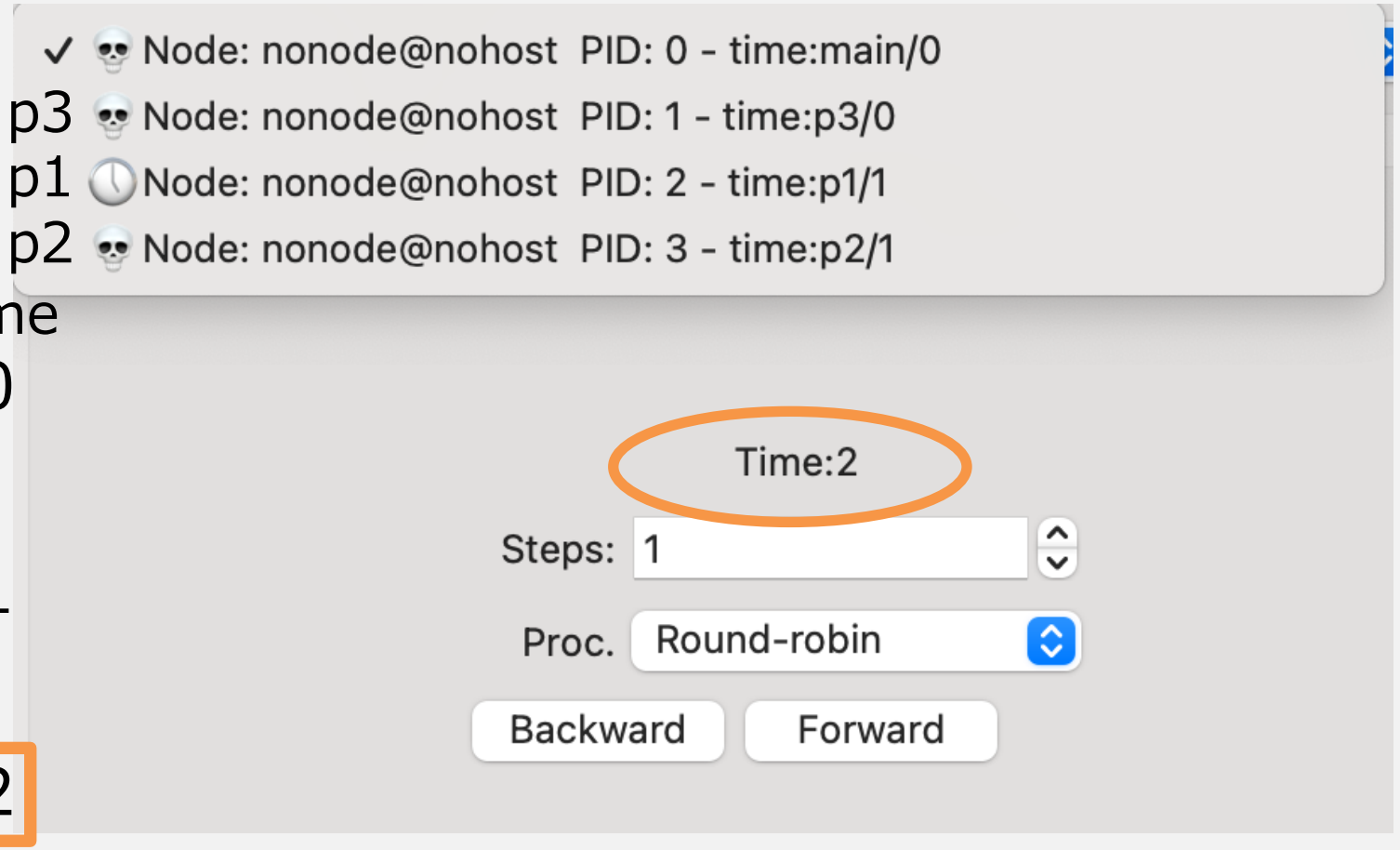


DEMONSTRATION

Backward Execution

- All processes wait for time passage

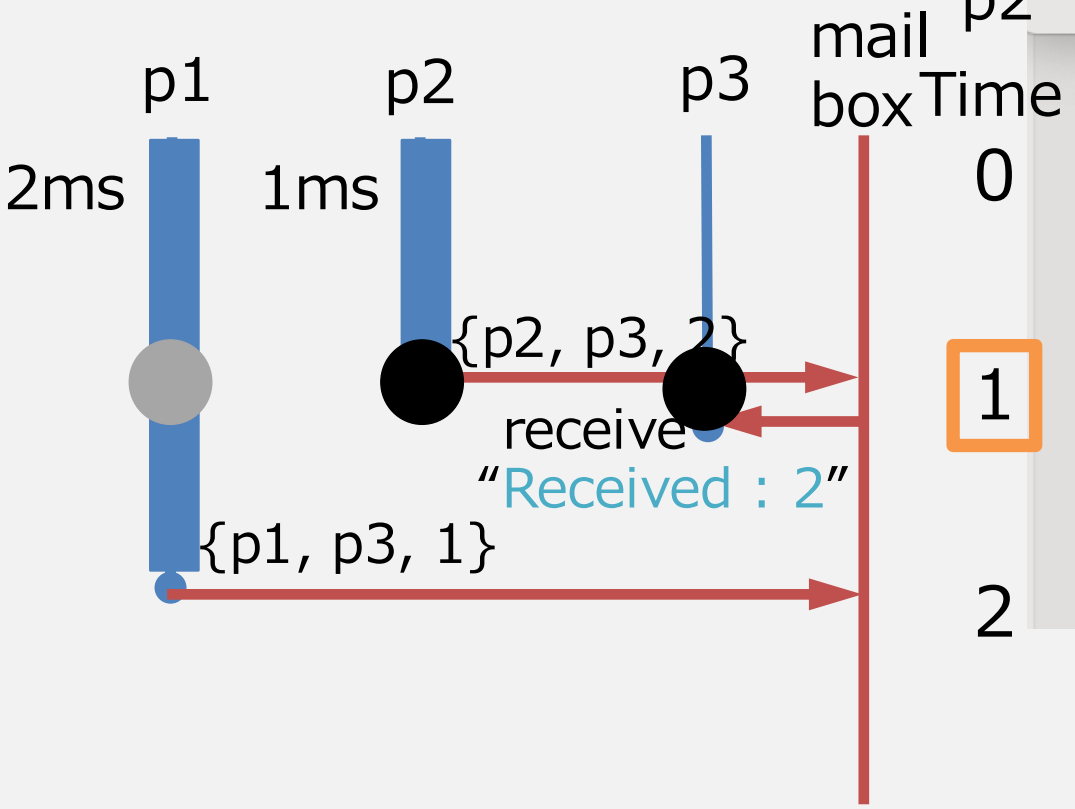


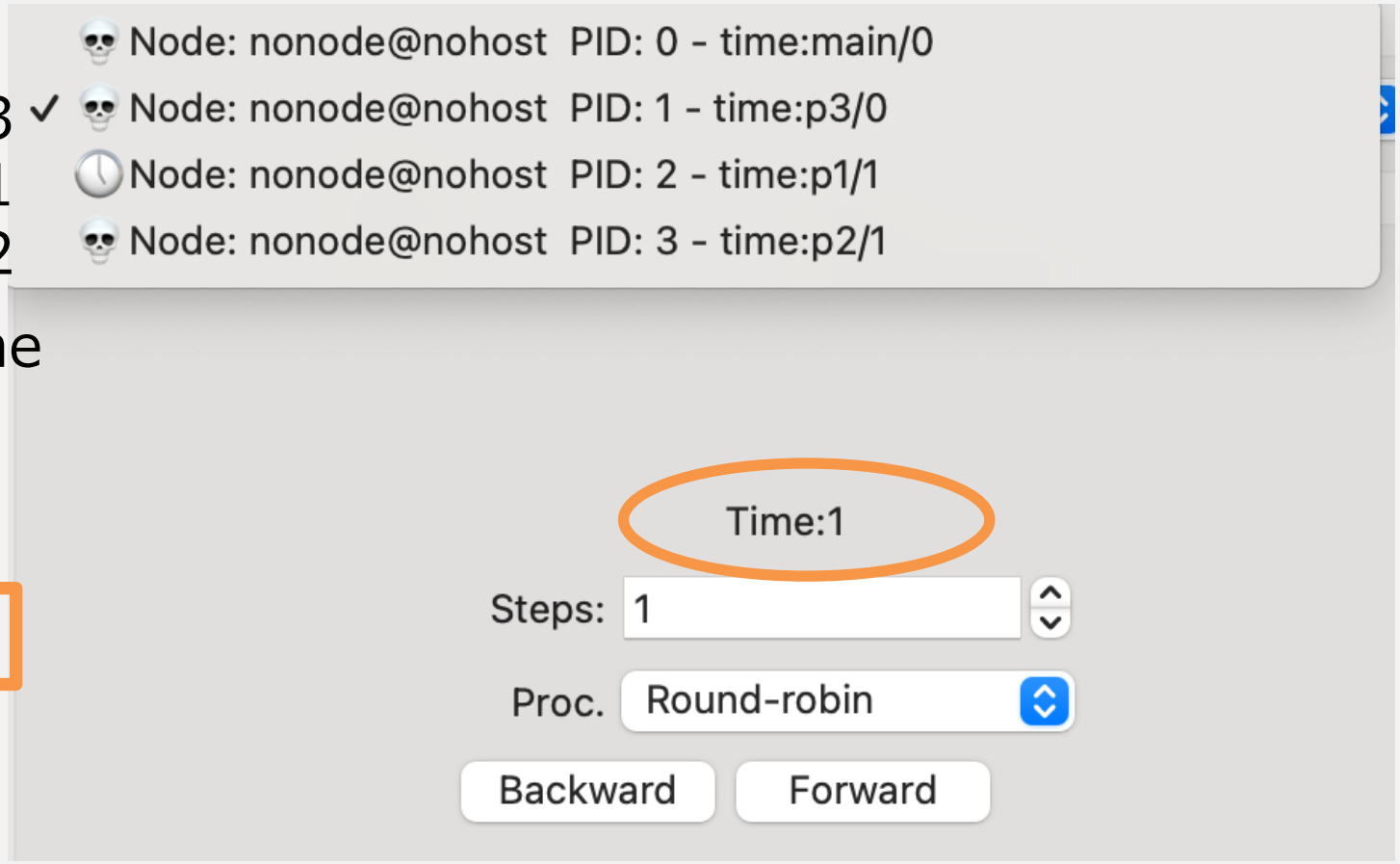


DEMONSTRATION

Backward Execution

- GUI changes to 'Time: 1’

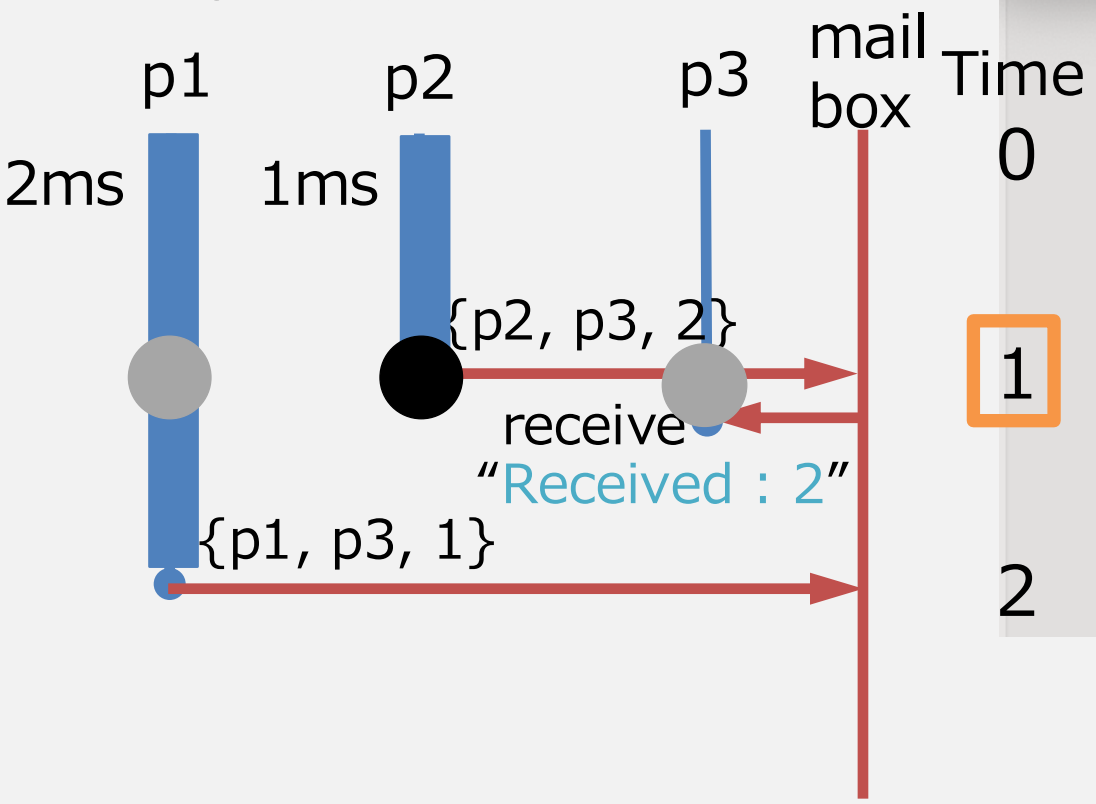


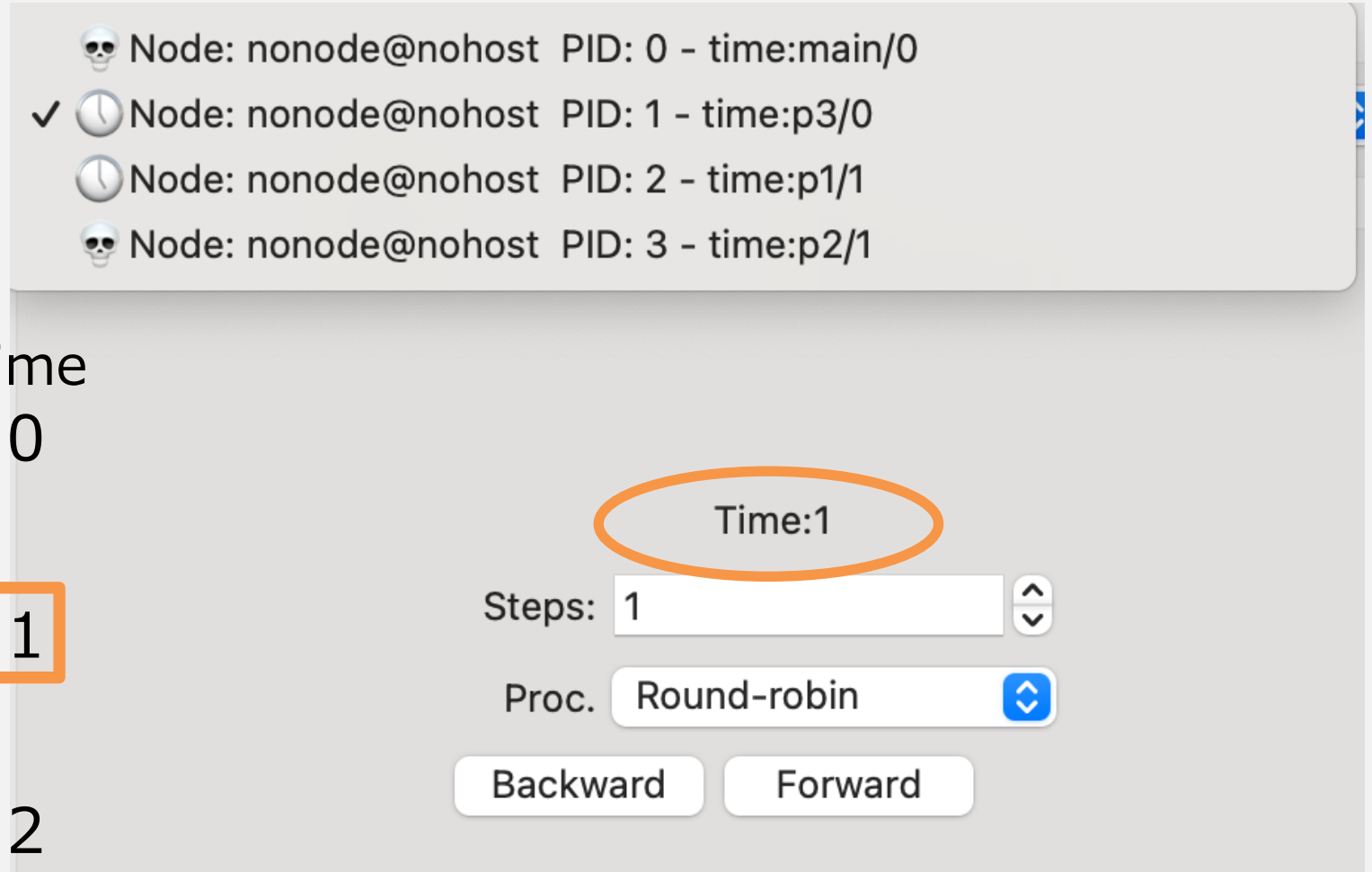


DEMONSTRATION

Backward Execution

- p3 waits for time passage after reversing to receive

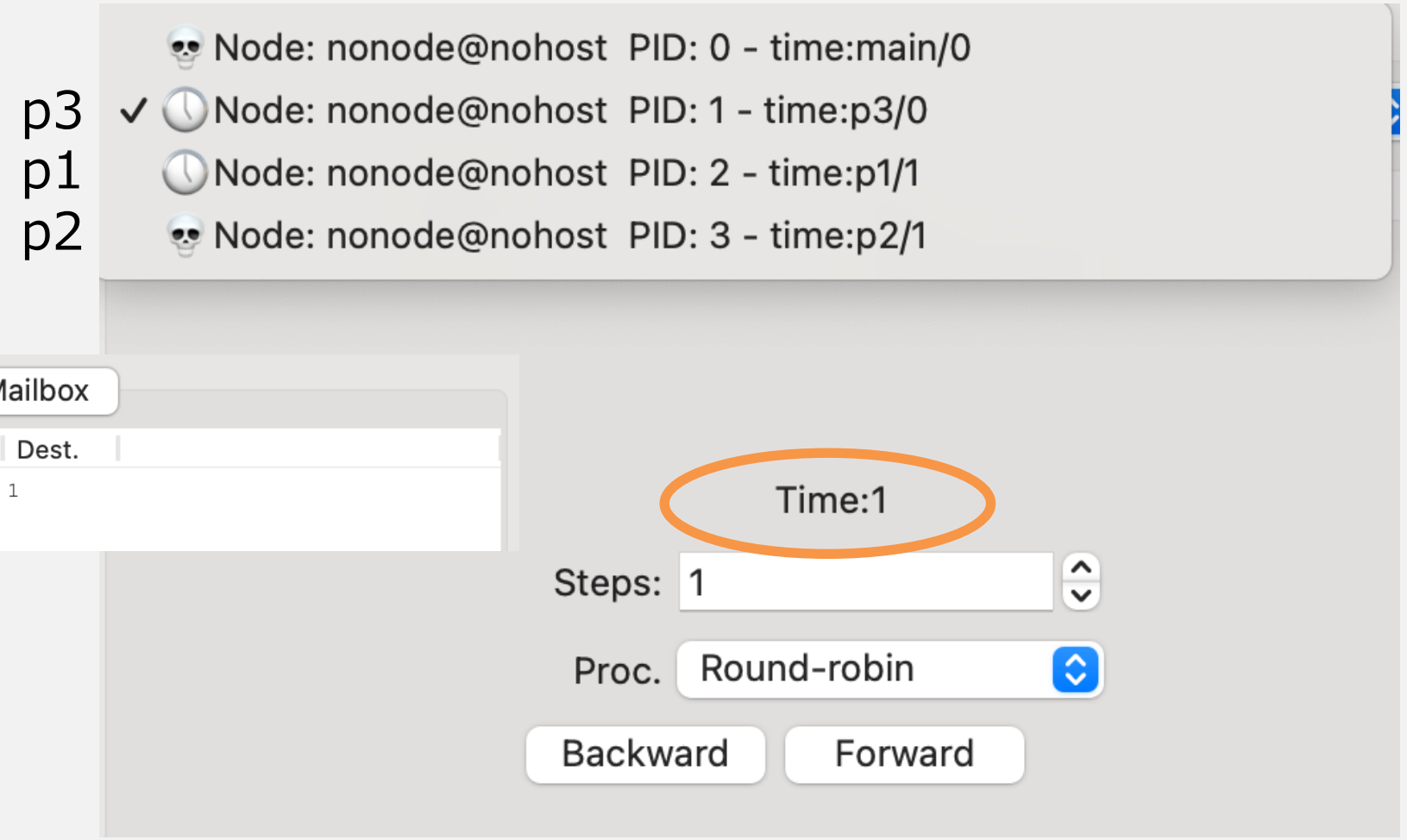




DEMONSTRATION

Backward Execution

- p3 waits for time passage after reversing to receive

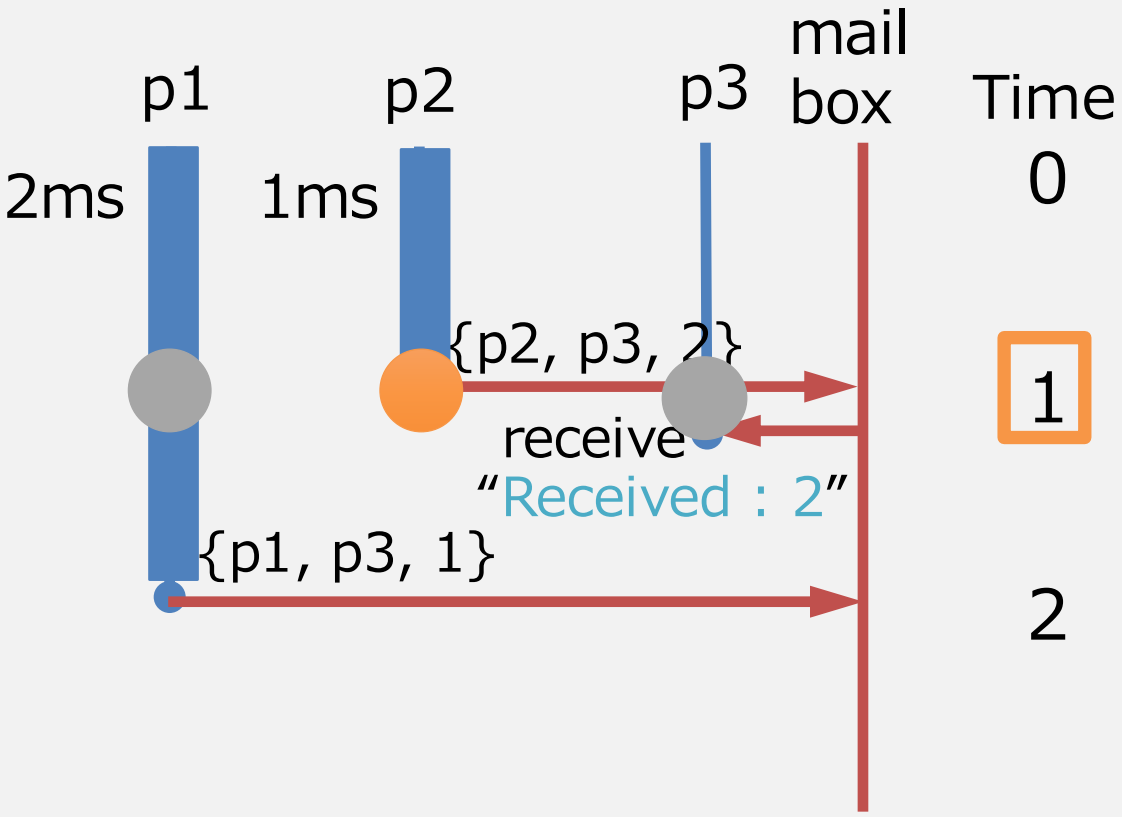


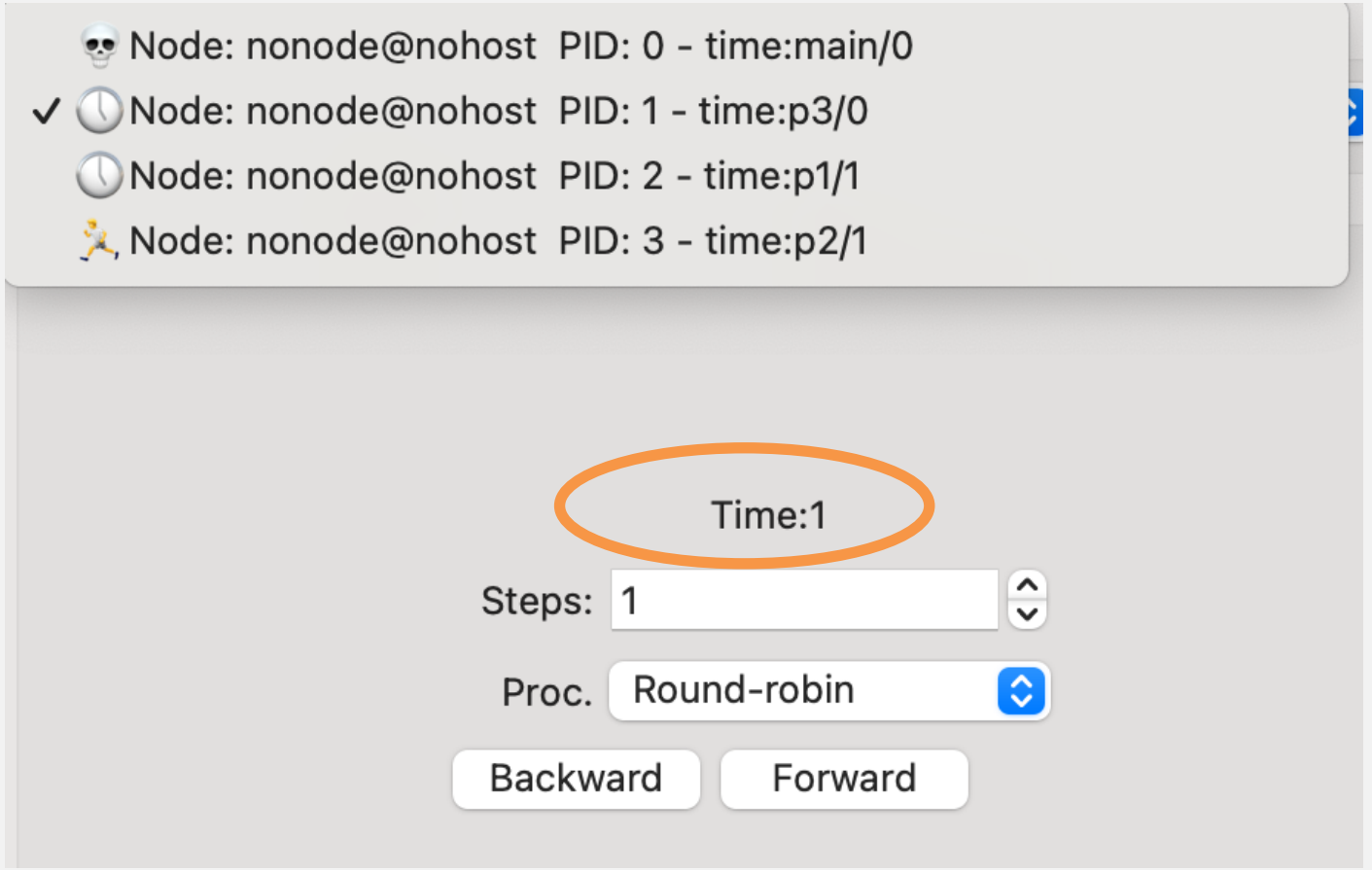
- Mailbox has message{2}

DEMONSTRATION

Backward Execution

- p2 wakes up

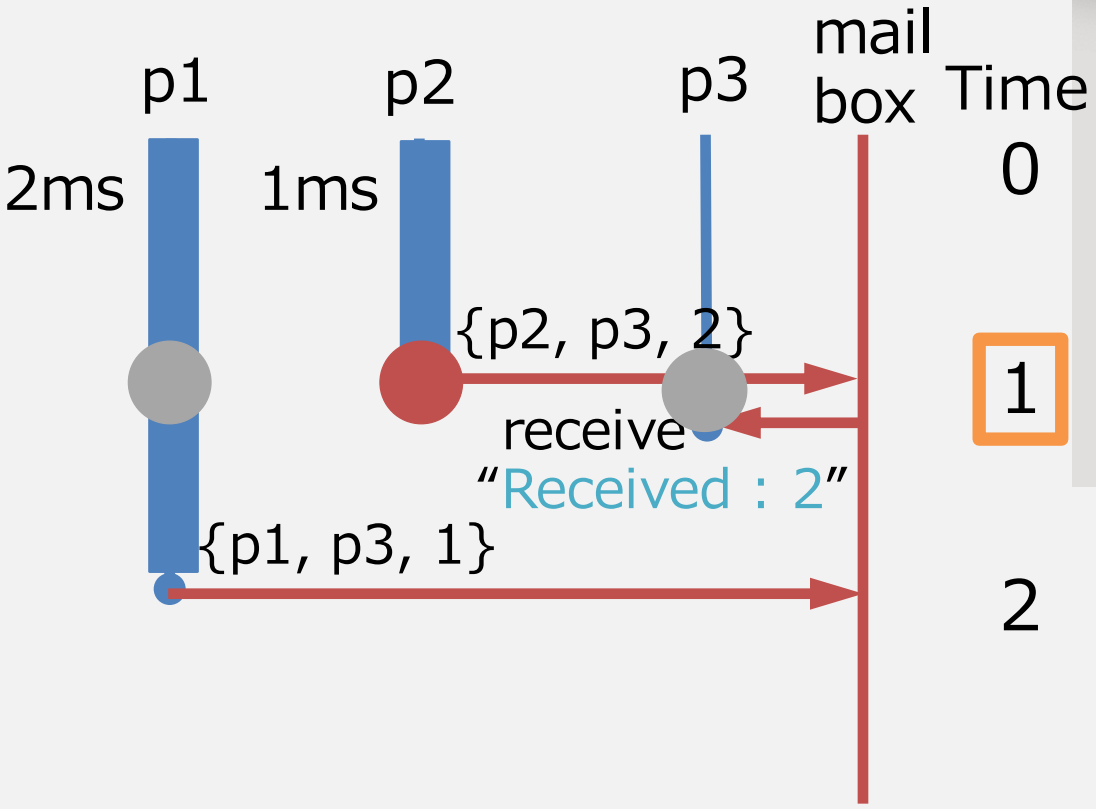


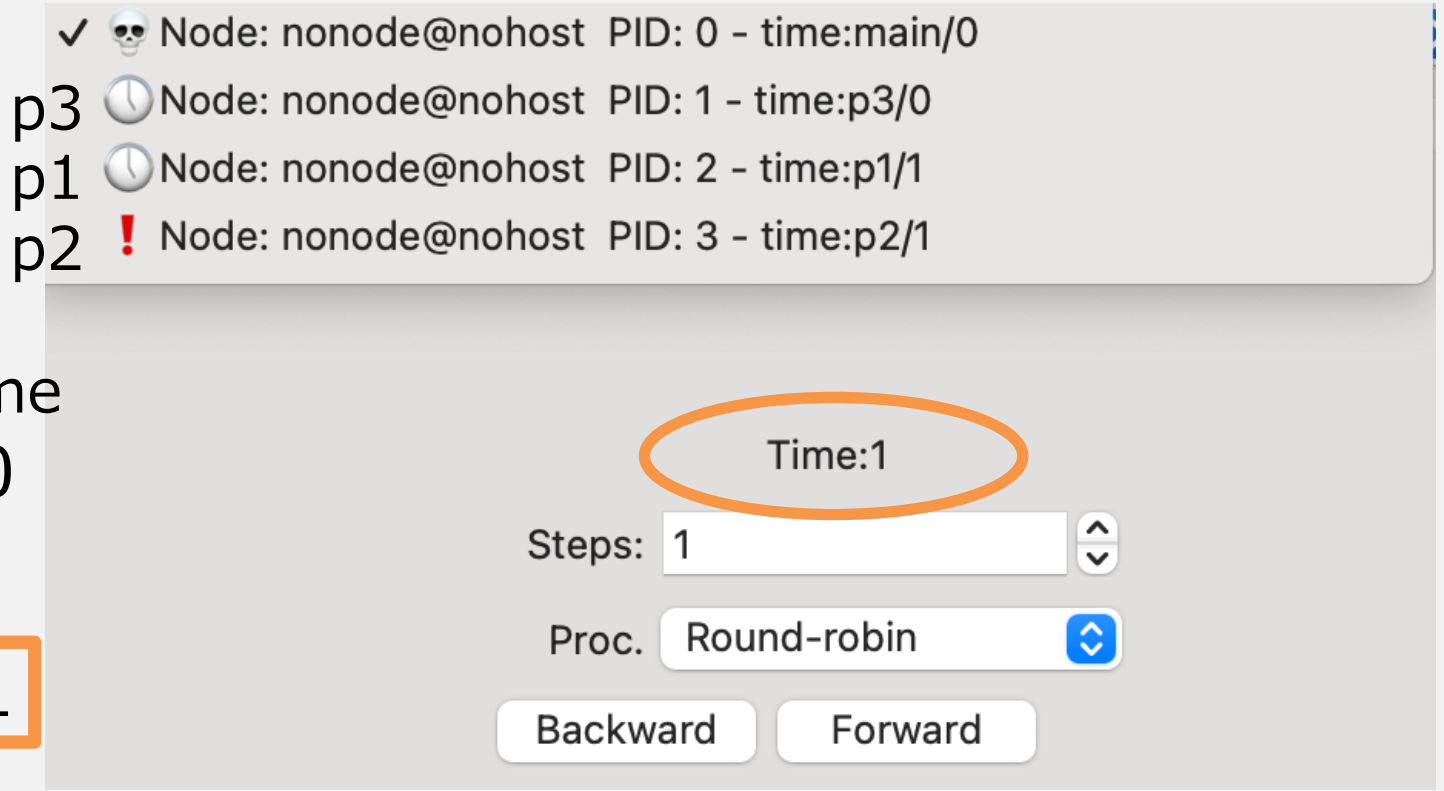


DEMONSTRATION

Backward Execution

- The top of history in p2 is timeout

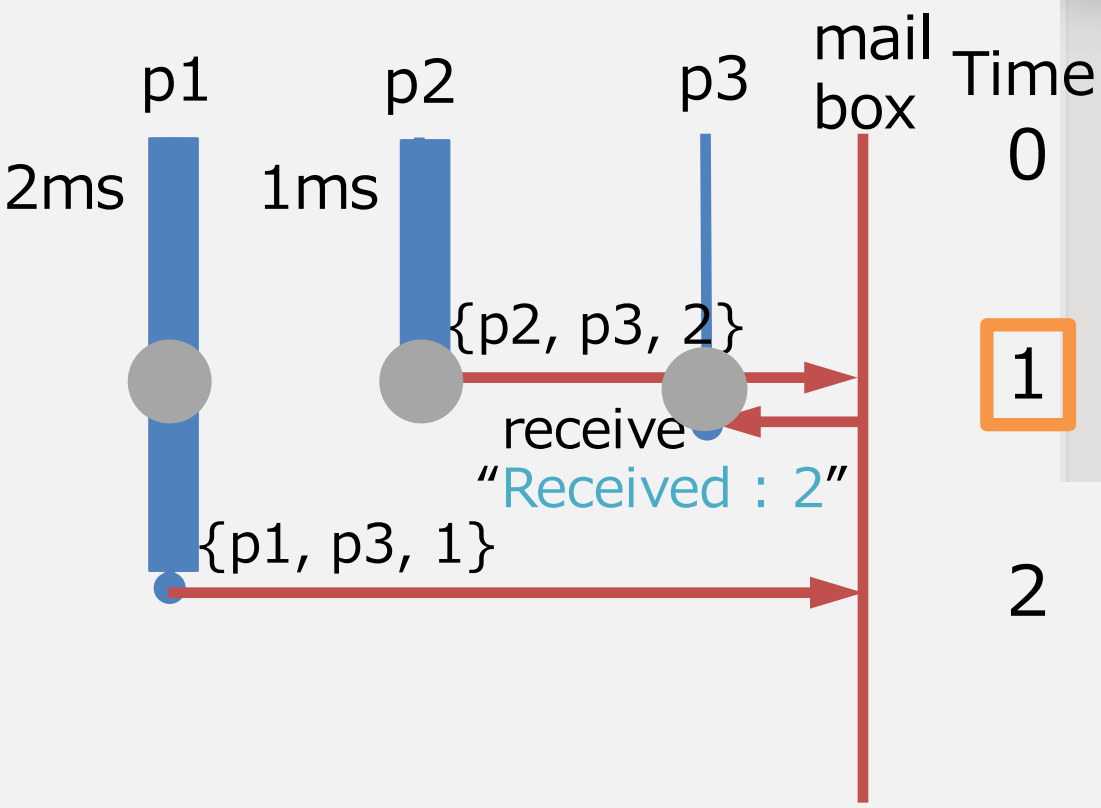


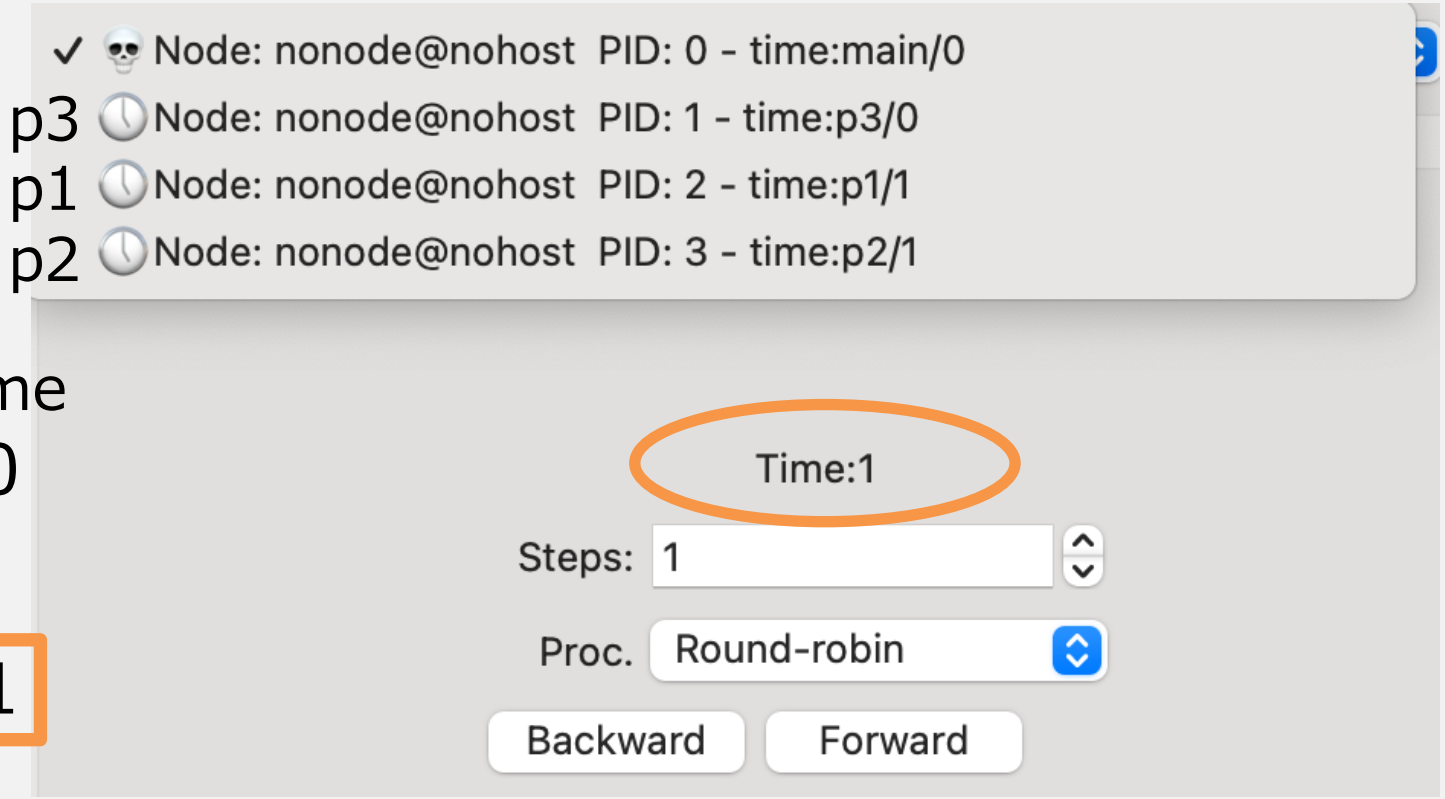


DEMONSTRATION

Backward Execution

- All processes wait for time passage

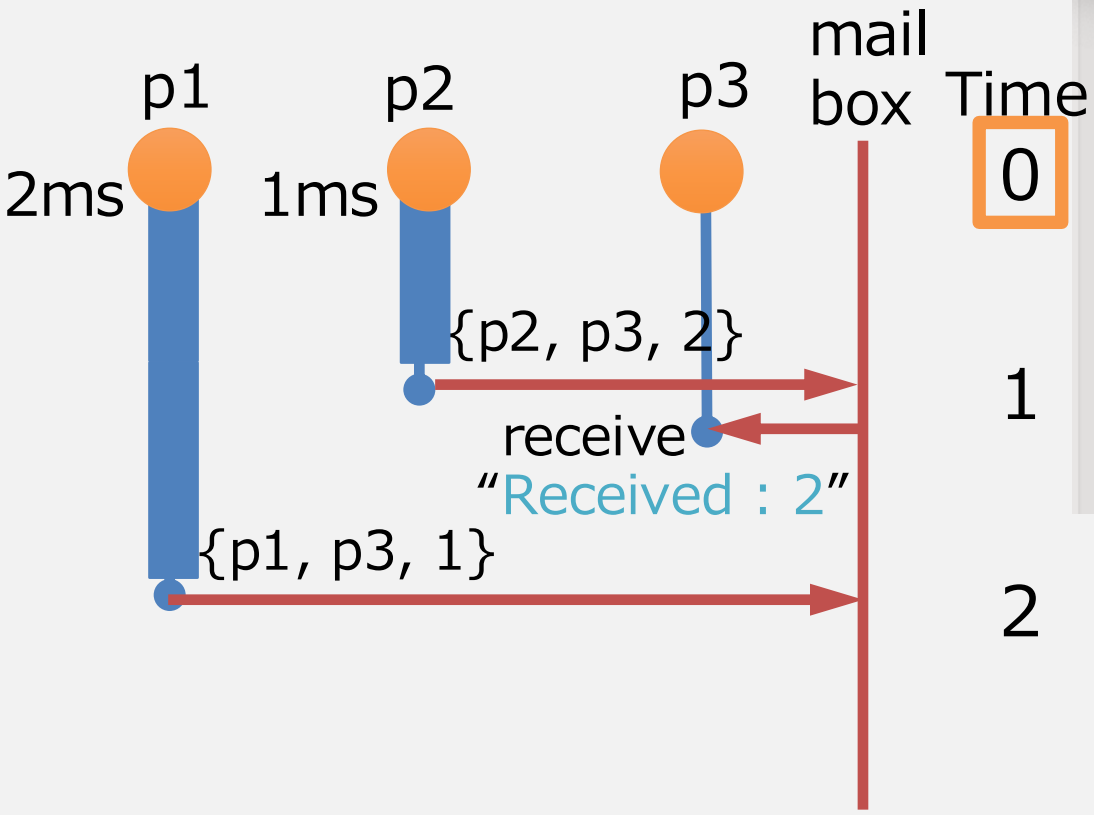


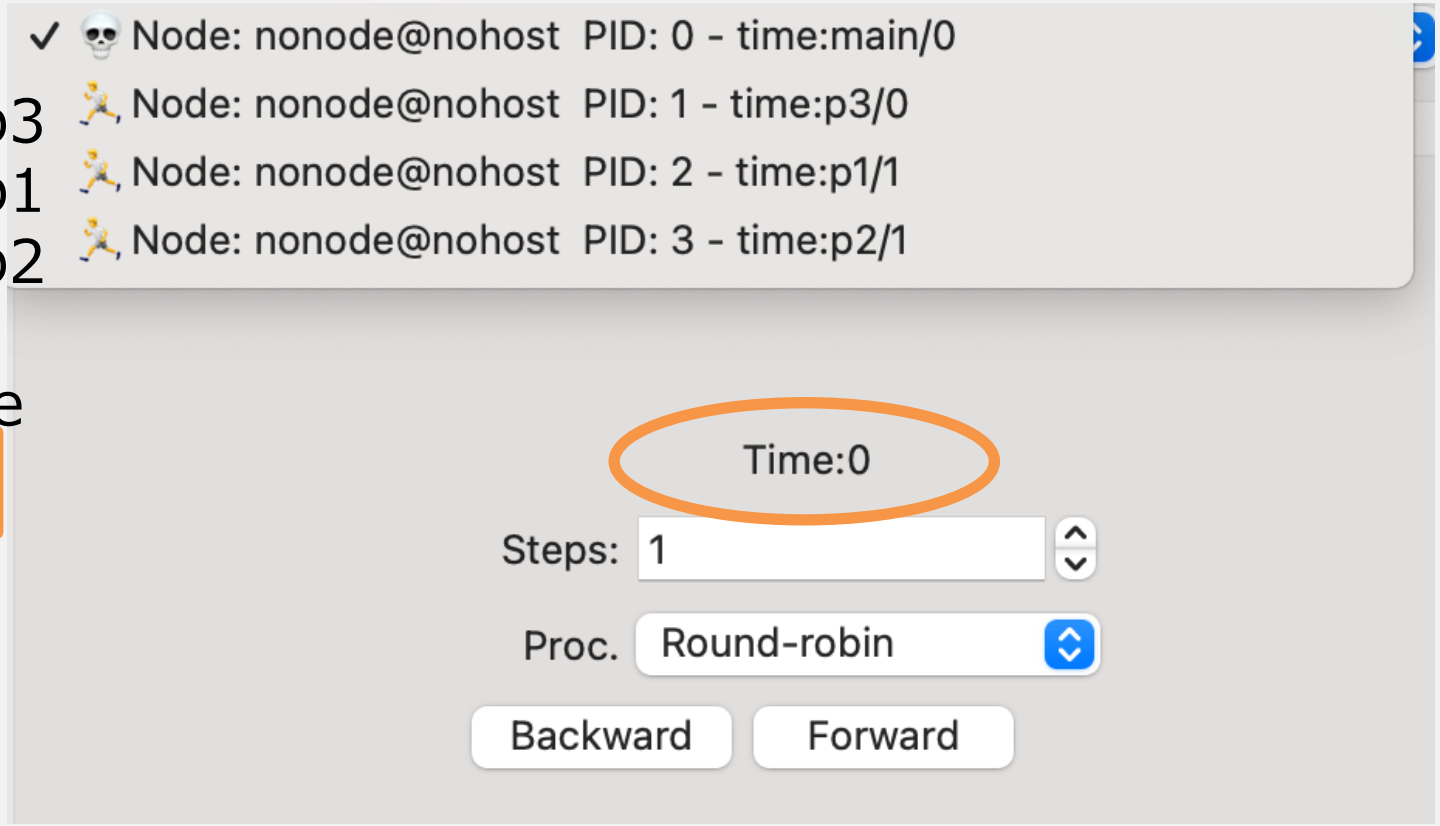


DEMONSTRATION

Backward Execution

- GUI changes to 'Time: 0’

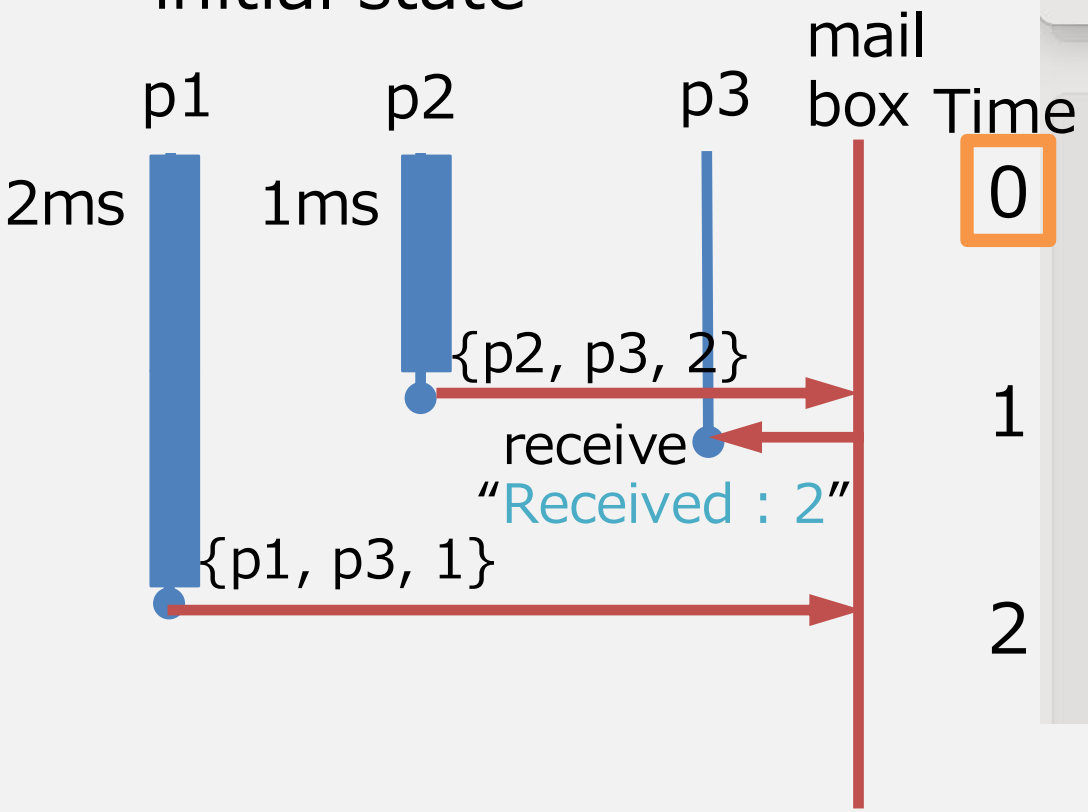


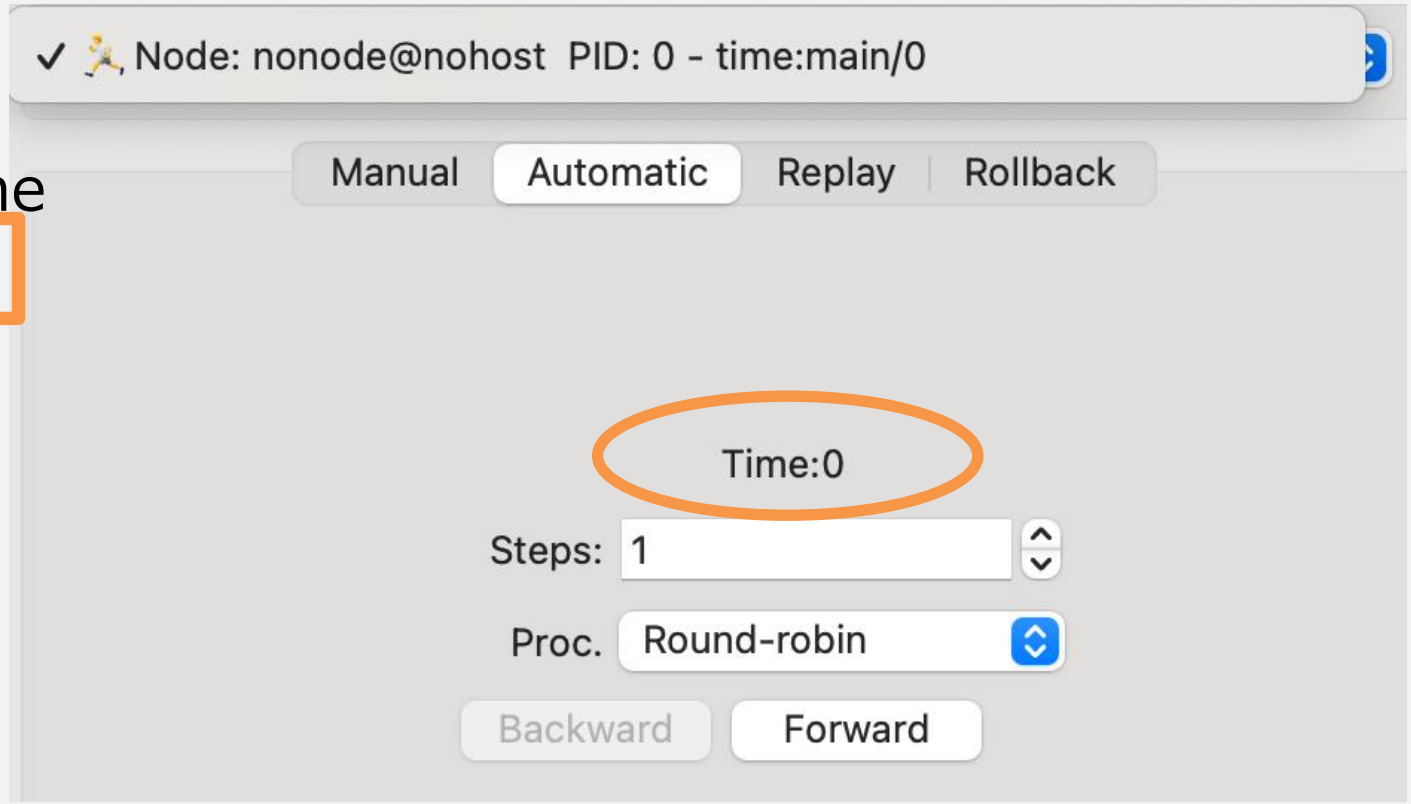


DEMONSTRATION

Backward Execution

- Back to the initial state





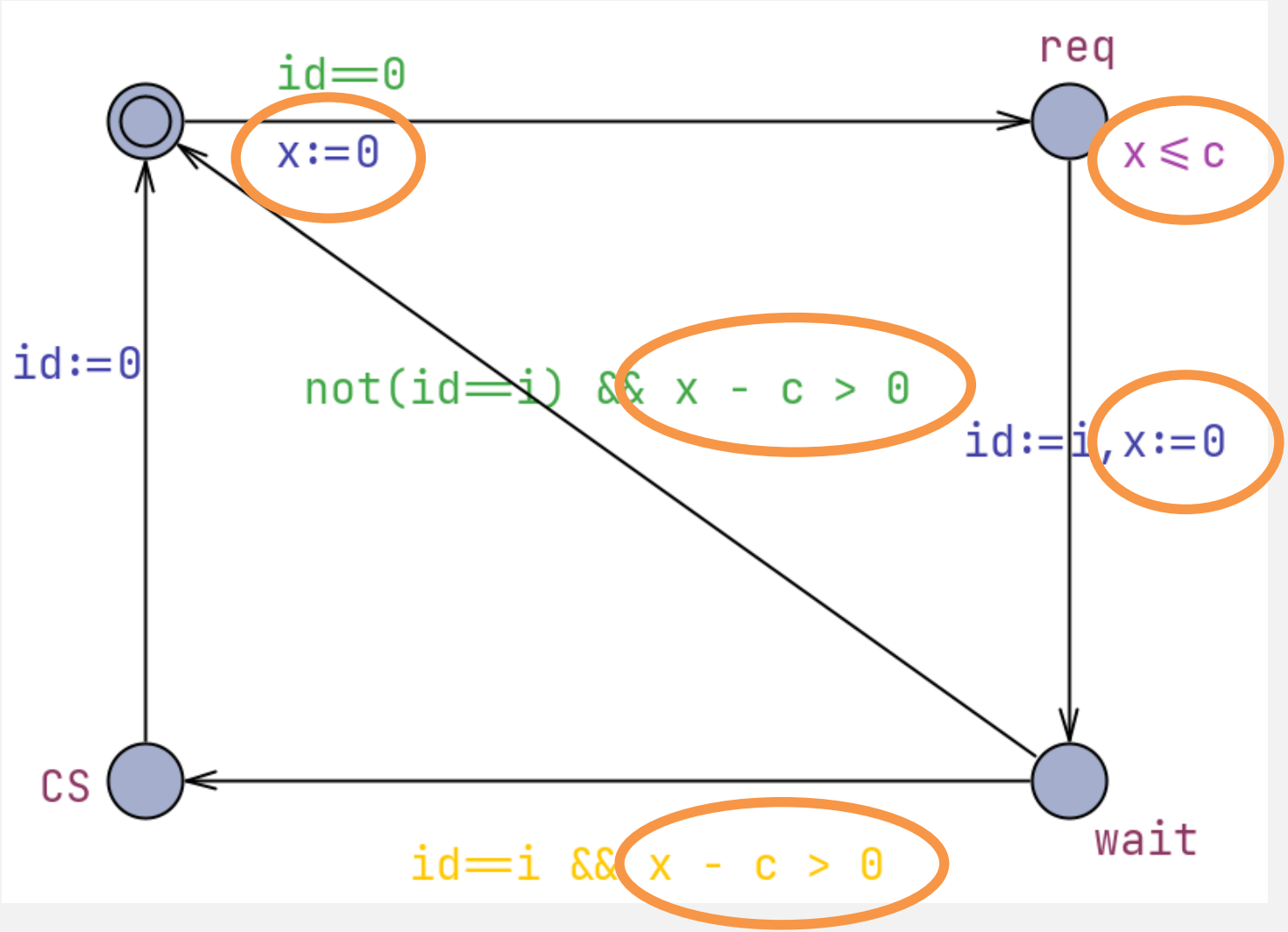
End of Backward execution

DEMONSTRATION

- Airline [Hoey+,20] program is a program in which bookers reserve seats
- Due to a data race, multiple bookers may over sell seats
- Avoid oversell by using Fischer's mutual exclusion
- Fischer's algorithm realizes mutual exclusion with one shared variable and local clocks for multiple processes

DEMONSTRATION

- Uppal model of Fischer's mutual exclusion



DEMONSTRATION

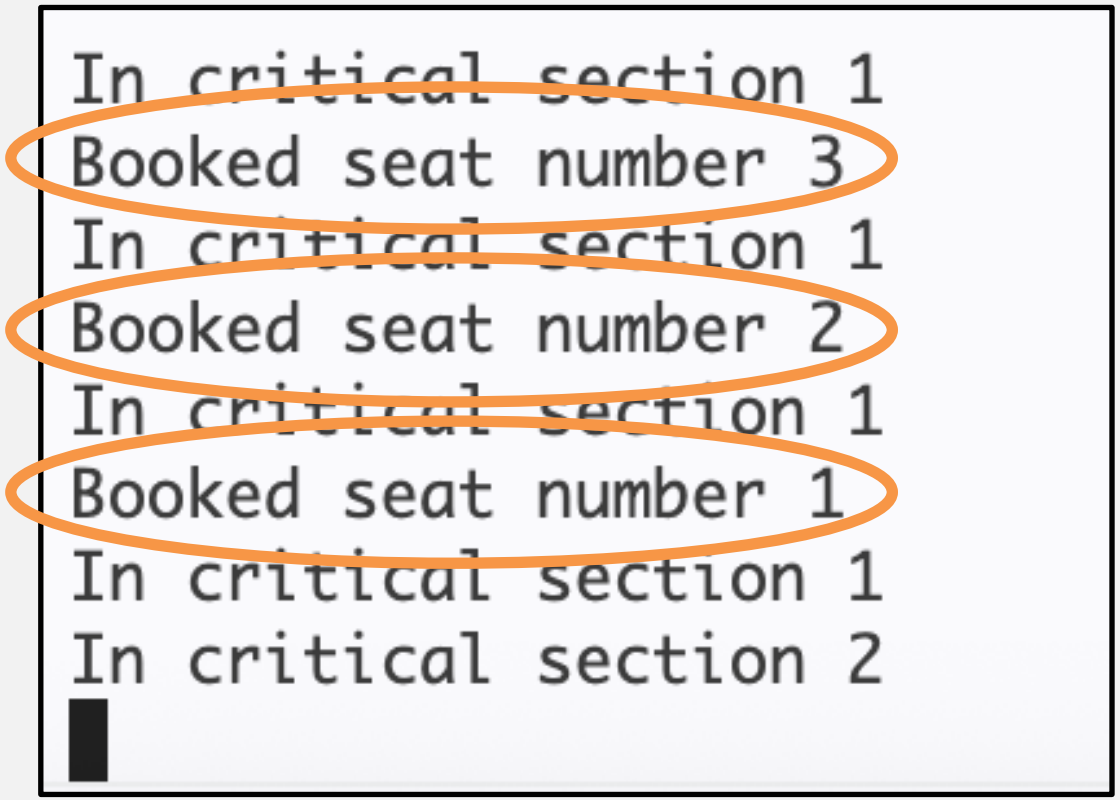
- This is part of the Airline-Fischer program
- This lock function realizes Fischer's algorithm by using the passage of time.

```
53 lock(FischerId, MyId) ->
54     case readId(FischerId) of
55         0 ->
56             timer:sleep(?DELAY),
57             writeId(FischerId, MyId),
58             timer:sleep(50),
59             case readId(FischerId) of
60                 MyId ->
61                     cs(FischerId, MyId);
62                 _OtherId ->
63                     lock(FischerId, MyId)
64             end;
65         _OtherId ->
66             lock(FischerId, MyId)
67     end.
68 cs(FischerId, MyId) ->
69     io:format("In critical section ~p~n", [MyId]),
70     writeId(FischerId, 0).
71
72 unlock(FischerId, MyId) ->
73     writeId(FischerId, 0),
74     lock(FischerId, MyId).
75
```

$x - c > 0$

DEMONSTRATION

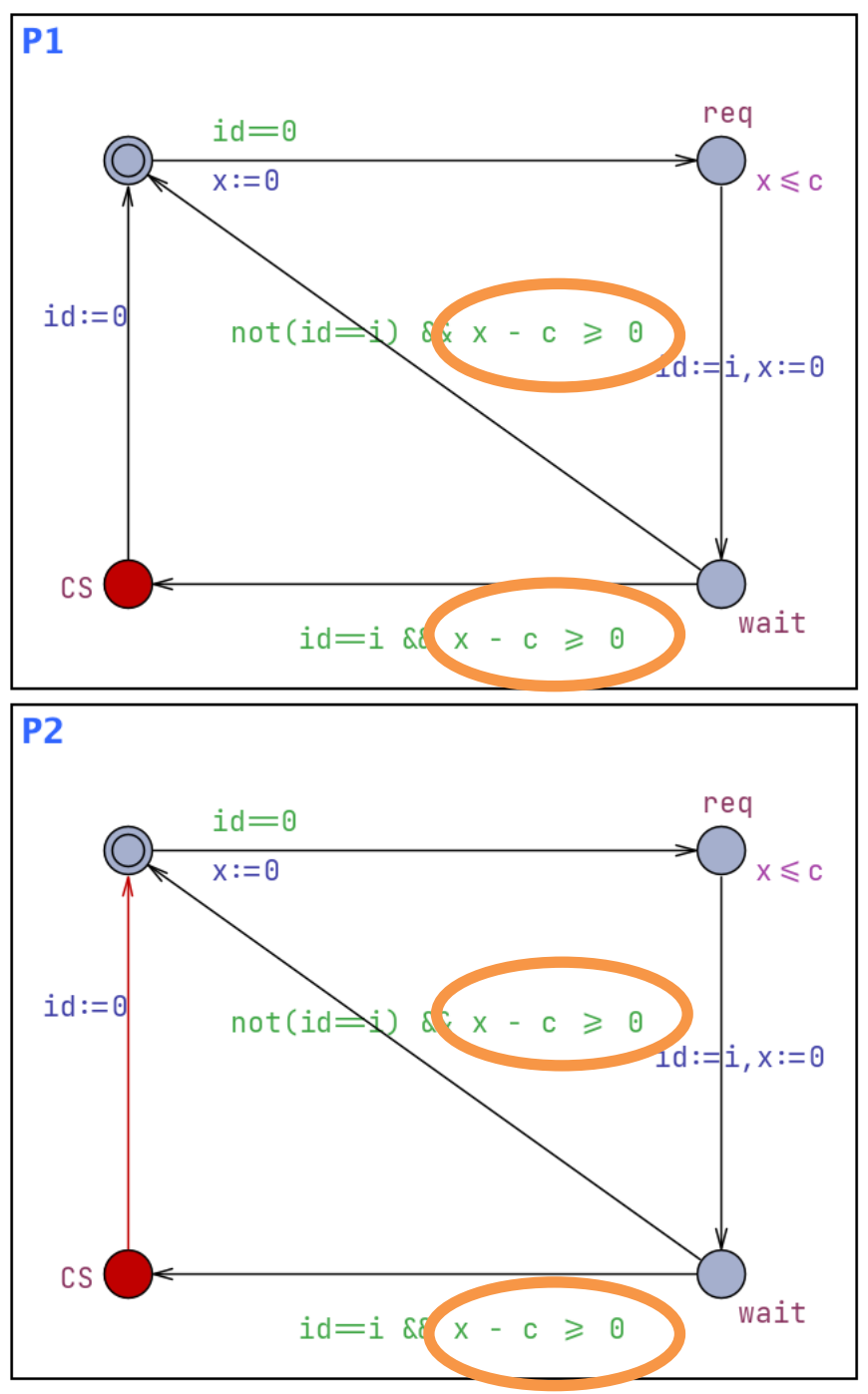
- When executing this program in CauDEr, double-booking of the same seat does not occur



works ok!

bug injection

- This allows two processes to enter the critical section at the same time, violating mutual exclusion



DEMONSTRATION

- This is part of the incorrect Airline-Fischer program
- This lock function allows processes to enter the critical section at the same time

```
50  lock(FischerId, MyId) ->
51      case readId(FischerId) of
52          0 ->
53              timer:sleep(?DELAY),
54              writeId(FischerId, MyId),
55              %timer:sleep(50),
56              case readId(FischerId) of
57                  MyId ->
58                      cs(FischerId, MyId);
59                  _OtherId ->
60                      lock(FischerId, MyId)
61              end;
62          _OtherId ->
63              lock(FischerId, MyId)
64      end.
```

comment out

$x - c \geq 0$

DEMONSTRATION

- When multiple bookers simultaneously check seat availability and then reserve the same seat without proper synchronization, a data race occurs

```
seatManager(NumOfSeats) ->
    receive
        {askForNum, Pid} ->
            Pid ! {num, NumOfSeats},
            seatManager(NumOfSeats);
        {book, Pid} ->
            io:format("Booked seat number ~w ~n", [NumOfSeats]),
            Pid ! {booked, NumOfSeats},
            seatManager(NumOfSeats - 1)
    end.
```

DEMONSTRATION

- When executing incorrect Airline-Fischer program in CauDEr, the 4th seat is booked
- Due to a data race, multiple bookers may reserve the same seat, and the number of remaining seats is -1

```
In critical section 1
In critical section 2
Booked seat number 3 by 3
Remaining seats: 2
Booked seat number 2 by 4
Remaining seats: 1
In critical section 2
In critical section 1
Booked seat number 1 by 4
Remaining seats: 0
Booked seat number 0 by 3
Remaining seats: -1
```

CONTENTS

- Background
- Time passage in the Erlang semantics
- Reversible Timed Semantics
- Integration to CauDEr
- Demonstration
- Conclusion

CONCLUSION

- Extended reversible semantics of Erlang with discrete time passage
- Integrate to CauDEr with timed reversible semantics
- Our timed semantics may not be minimal for integrating into CauDEr (δ(d) is designed for practical use)

Future work

- Timed rollback-and-replay semantics
- Introduce an ε-parameter to capture runtime delay
- Support imperative primitives