LU3IN013

Research Project

Formal Modeling of Algorithms for Distributed Systems

Final Report

Tra My Nguyen
3702069

2019-2020

1 Introduction

Distributed systems are omnipresent nowadays as they help resolving various computing problems with efficiency and high adaptability. However, along these advantages come a lot of difficulties due to the fact that entities in these systems run concurrently and can only communicate with each other via message-passing. As a result, coordinating the behavior of independent components of the system becomes the major challenge in distributed computing and one that is really hard to analyze. Instead of relying on traditional extensive testing, modern day system design uses formal methods based on mathematical techniques to study the reliability and robustness of a system. One proposal is the Carl Adam Petri's seminal work on Petri nets in the early 1960s.

This project is carried out in responding to the Call for Models by the Model Checking Contest handled annually by LIP6, an event where formal verification tools for concurrent systems are evaluated in order to find the best suited techniques for a class of problem (e.g. state space generation, deadlock detection, reachability analysis, causal analysis). The goal is to create formal models with scaling capabilities from distributed algorithms using previously mentioned Petri nets, which may then be used alongside others in the contest as common benchmarks on which all tools will be compared.

In this first report, I will be giving a general view on the notion of Petri nets and their functioning principle, as well as an initial application of Petri nets in modeling a Client-Server communication system. I will also be briefly presenting two distributed algorithms from which I will attempt to create Petri nets models and the steps I will take to work toward that goal.

2 Petri Net

Place/Transition Petri nets (P/T net) will be the main focus in this project.

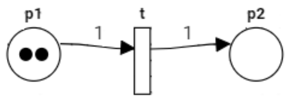


Figure 1: An simple P/T net

The image above shows **a place/transition net (P/T net)**, which, at first glance, resembles a bipartite oriented weighted graph. Its vertices can be separated into a set of rectangle (known as transitions) and a set of circles (places), each circles containing a number of dots (tokens). A state of a P/T net is established by the number of tokens held by each place, formally defined by a marking. Transitions of a P/T net represent actions that can take place and modify its state. The cost of one action is indicated on its input arc (number of tokens taken away from the source place) and its product on the output arc (number of tokens added to the target place).

The models to be produced in the project need to be scalable, which means by increasing a certain type of actors having similar behaviors in a system (parameters), we obtain multiple P/T nets describing the same scenario but of different sizes (number of transitions/places/arcs) or of different initial marking.

2.1 Formal definitions(C.Girault, 2001)

A P/T net can be hereby formally defined as *a tuple* $\mathcal{N} = (P, T, \mathbf{Pre}, \mathbf{Post})$, *where*
• *P is a finite set of <u>places</u>, representing resources in a system, each place holds a number of <u>tokens</u>, representing the number of occurrences of a resource*
• *T is a finite set of <u>transitions</u>, disjoint from P, representing actions that can occur in the system, and*
• $\mathbf{Pre}$,$\mathbf{Post} \in \mathbb{N}^{|P|\times|T|}$ *are matrices (the backward and forward incidence matrices of $\mathcal{N}$), $\boldsymbol{C} = \boldsymbol{Post} - \boldsymbol{Pre}$ is called the incidence matrix of $\mathcal{N}$.*

Pre	t	**Post**	t	**C**	t
p_1	1	p_1	0	p_1	-1
p_2	0	p_2	1	p_2	1

Table 1: The incidence matrices of the P/T net in Figure 1

A marking of a P/T net $\mathcal{N} = (P, T, \mathbf{Pre}, \mathbf{Post})$ is a vector $\mathbf{m} \in \mathbb{N}^{|P|}$. $\mathcal{N}$ *together with a marking* $\mathbf{m}_0$ *(initial marking) is called* **a P/T net system** $\mathcal{S} = (\mathcal{N}, \mathbf{m}_0)$.

It can be said that *a transition $t \in T$ is enabled in a marking* $\mathbf{m}$ *if* $\mathbf{m} \geq \mathbf{Pre}[\bullet, t]$. *In this case* **the successor marking relation** *is defined by* $\mathbf{m} \xrightarrow{t} \boldsymbol{m'} \Leftrightarrow \mathbf{m} \geq \mathbf{Pre}[\bullet, t] \wedge \mathbf{m'} = \mathbf{m} + \mathbf{Post}[\bullet, t] - \mathbf{Pre}[\bullet, t] = \mathbf{m} + \mathbf{C}[\bullet, t]$ *(*$\mathbf{Pre}[\bullet, t]$ *denotes the t-column vector* $\mathbf{Pre}[\bullet, t] = (\mathbf{Pre}[p_1, t], ..., \mathbf{Pre}[p_{|P|}, t])$*) of the* $|P| \times |T|$ *matrix* $\mathbf{Pre}$. *The same holds for* $\mathbf{Post}[\bullet, t]$ *with respect to* $\mathbf{Post}$.

Here, the initial marking is $\mathbf{m}_0 = [2, 0] > \mathbf{Pre}[\bullet, t] = [1, 0]$, the transition t is therefore fireable and firing it will lead to the successor marking $\mathbf{m}_1 = [1, 1]$.

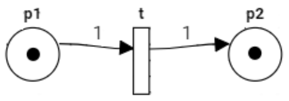


Figure 2: New P/T net after firing transition t

3 The First Model: Client-Server System

Keeping in mind how a P/T net works, we can start modeling an P/T net simulating a simple communication channel between some clients and several servers. Each client can send a request to the channel, which will address it to one of the servers and transfer the reply back to the same client once the chosen server has responded.

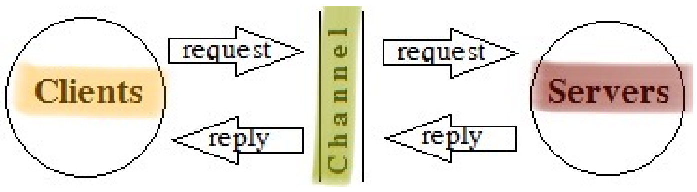


Figure 3: Client-Server communication system

All models basically consist of 3 different parts: the clients, the servers and the channel connecting them. From now on, they will be highlighted respectively in orange, green and red.

Each actor has an initial stage represented by a place holding one token. A client$_i$ sending a request via the transition `send_req`$_{\texttt{i}}$ puts the channel to a new place `sys_req`$_{\texttt{i}}$ where it waits for an available server to help proceeding the request. After obtaining the reply, it enters the `sys_rep`$_{\texttt{i}}$ place, meaning it is ready to response to client$_i$ via transition `rec_rep`$_{\texttt{i}}$. Since the channel deals with one client at a time, there will be one set of these type of transitions and places for each client$_i$. Apart from the initial place, a server$_j$ is composed of as many `req`$_{\texttt{j}}$`_c`$_{\texttt{i}}$ transitions as there are clients because it can take a request from any client and only one at a time.

From this initial attempt, we can move on to expanding the client model by adding a `att`$_{\texttt{i}}$ place where the client would wait for a reply.

For the server model, let's add a place named `sv`$_{\texttt{j}}$`_c`$_{\texttt{i}}$ between a server$_j$ receiving a request from the channel and giving a reply, where it would be handling the request of client$_i$.

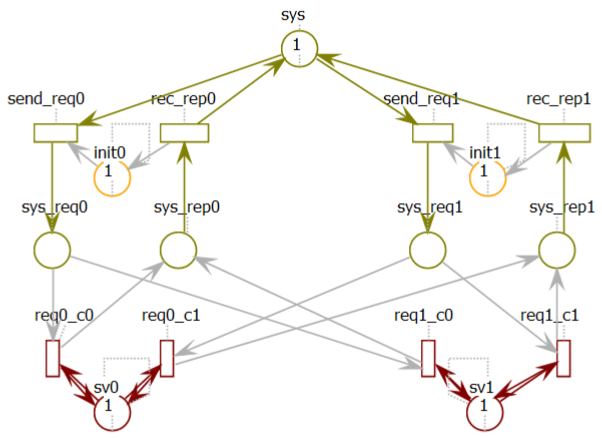


Figure 4: Initial P/T net model of 2 clients and 2 servers

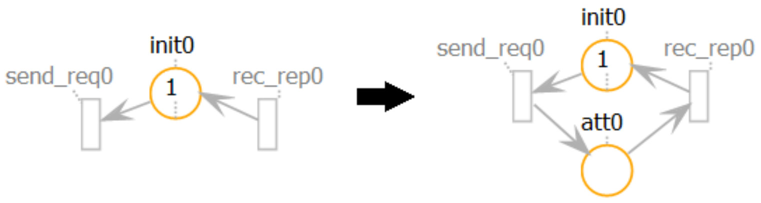


Figure 5: Initial expansion of Client$_0$

The model now looks something like this.

At the end, in order to increase the size of the net's state space, a possibility is to imagine the clients doing an extra step of calculation using their reply (place $\texttt{cc}_\texttt{i}$ and transition $\texttt{calcul}_\texttt{i}$) before returning to sending new requests and also, a resetting phase for the servers (place $\texttt{done}_\texttt{j}$ and transition $\texttt{reset}_\texttt{j}$).

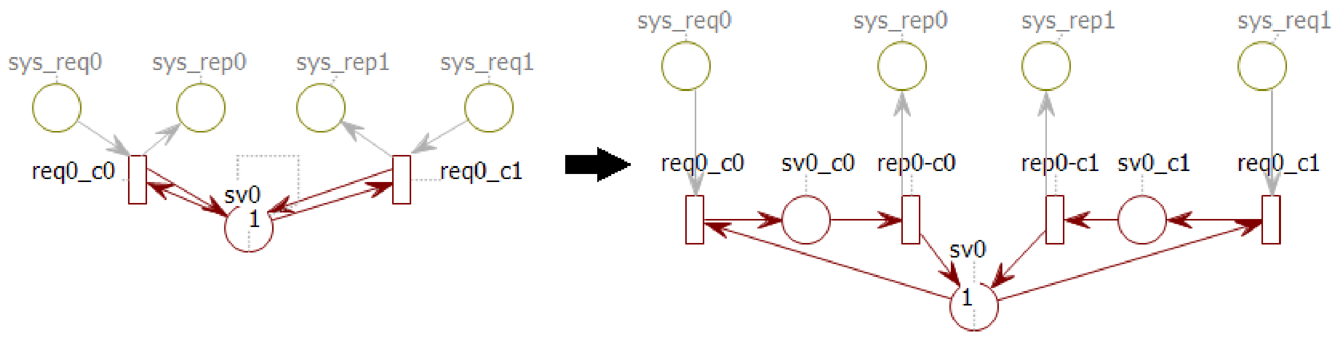


Figure 6: Initial expansion of Server_0

Here is the final model of 2 clients and 2 servers after combining the new client and server models with the initial channel model.

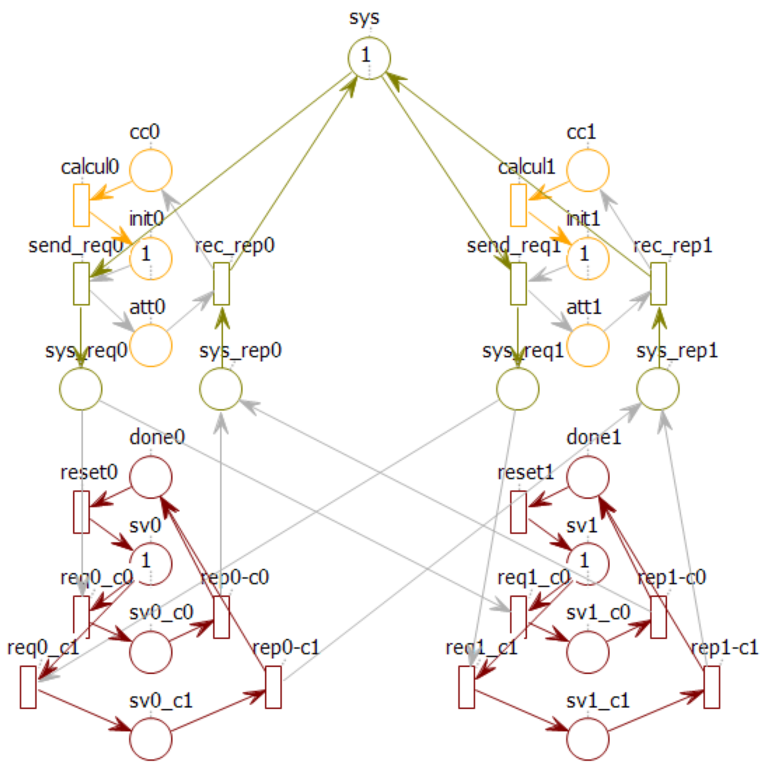


Figure 10: Final P/T net model of 2 clients and 2 servers

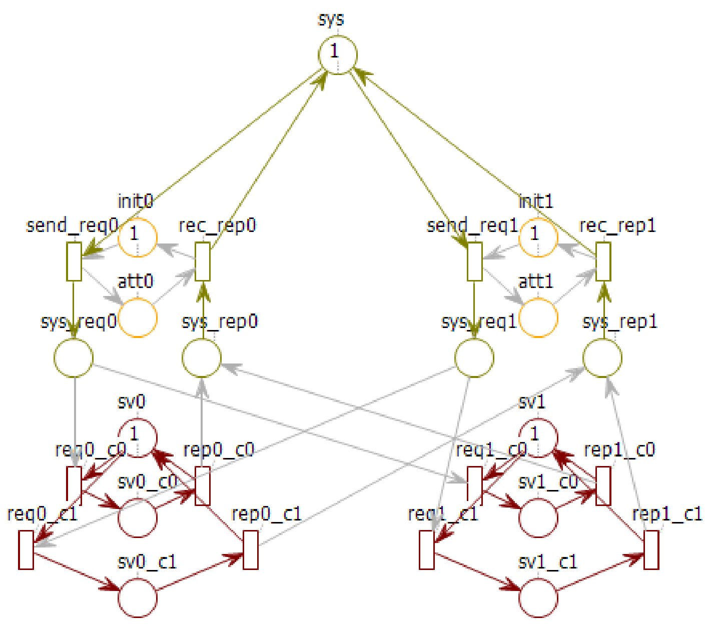


Figure 7: Expanded P/T net model of 2 clients and 2 servers

P/T nets of the same final type can be generated for a given number of clients and servers using a generator basing on the Java object-oriented language and the PNML Framework library. The idea of this generator follows simply each model of different actors explained above, each one implemented in its own class and finally, a common class Generator completes the net by connecting arcs between models (arcs in gray).

Each model generated will possess these types of places and transitions:

Places:

Channel

- sys, waiting for request
- $\texttt{sys\_req}_i$, having request from Client_i
- $\texttt{sys\_rep}_i$, having reply for request of Client_i

Client_i

- $\texttt{init}_i$, wanting to send request
- $\texttt{att}_i$, waiting for reply
- $\texttt{cc}_i$, ready to calculate

Server_i

- $\texttt{sv}_i$, available
- $\texttt{sv}_i\texttt{\_c}_j$, treating request from Client_j
- $\texttt{done}_i$, done treating last request

Transitions:

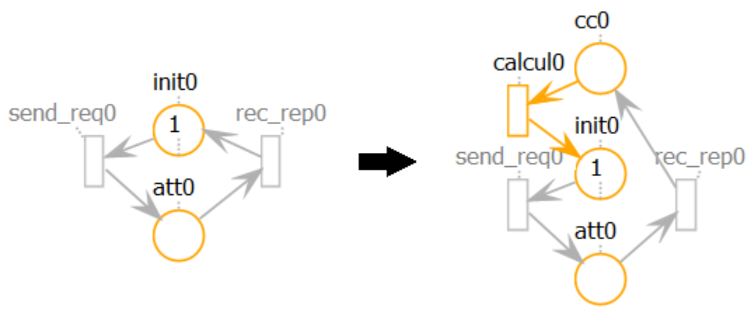


Figure 8: Second expansion of Client_0

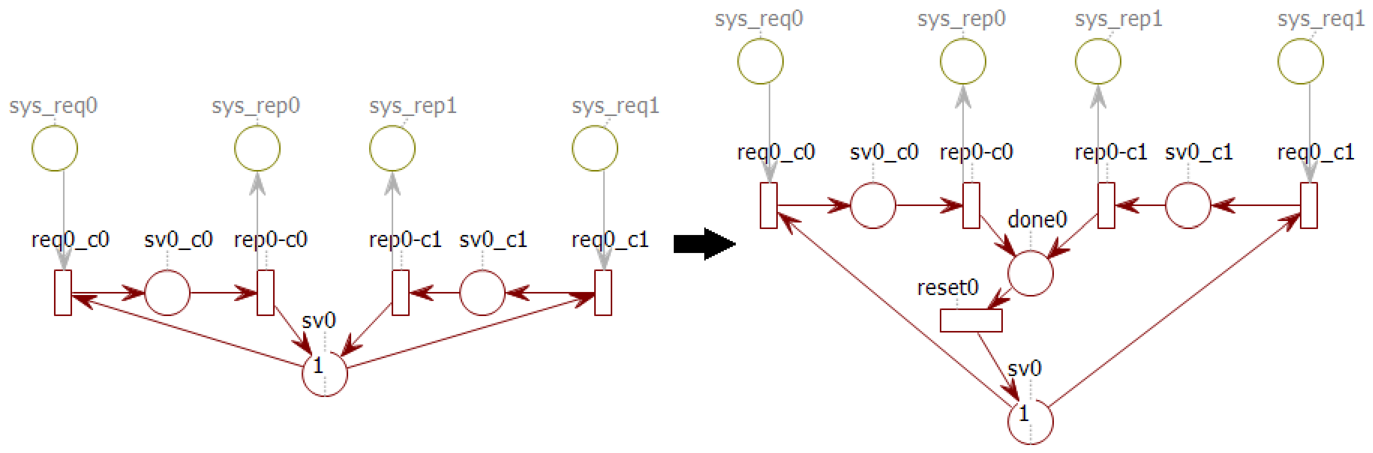


Figure 9: Second expansion of Server_0

<u>Channel</u>

- `send`$_i$, Client_i sending request to channel
- `receive`$_i$, Client_i receiving reply from channel

<u>Client_i</u>

- `calcul`$_i$, calculating

<u>Server_i</u>

- `req`$_i$`_c`$_j$, receiving request of Client_j from channel
- `rep`$_i$`_c`$_j$, sending reply to request of Client_j to channel
- `reset`$_i$, resetting

4 Modeling distributed algorithms

4.1 Leader election algorithm for bidirectional ring

4.1.1 The algorithm (Raynal, 2012)

Firstly, I will be looking at a classic problem in a distributed system: leader election, where node with the highest identity number in a network would be elected as the leader. The algorithm below, presented by D.S.Hirschberg and J.B.Sinclair(1980), shows the procedure of each node in a bidirectional ring network (where each node of identity id_i has two neighbors $left_i$ and $right_i$ with whom it can communicate (send and receive messages)).

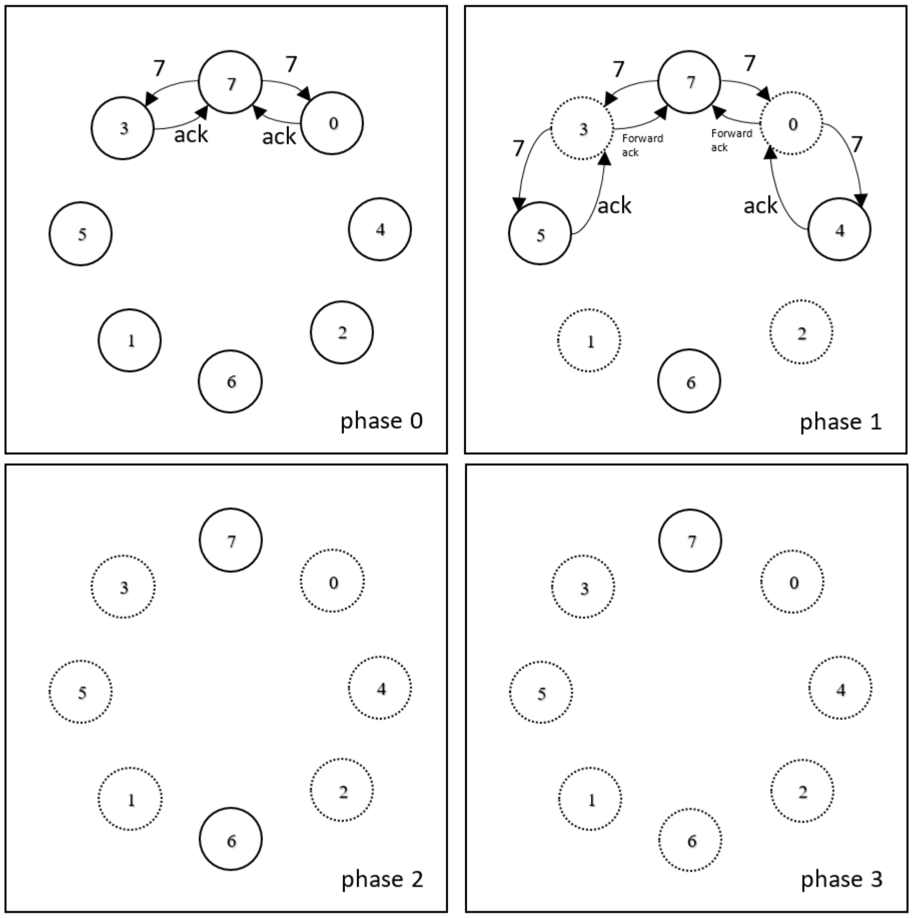


Figure 11: Example of leader election in bidirectional 7-process ring network

I will be modeling the behavior of each node, how they react to a message from another node. All nodes will receive a START message from exterior.

Each node then sends ELECTION$(id_i, 0, 1)$ to both of its neighbors, presenting its own identity, the first round number $r = 0$ and the number of nodes its message has visited $d = 1$. During each round r, each node competes with its 2^r neighbors to the left and to the right, and only the winner of round r can continue to round $r + 1$. After one round, the identity of the winner of a neighborhood will be sent in a REPLY message back to the winner itself, a node receiving two REPLY messages from both sides learns that it is the winner of its $(2^r + 1)$ neighborhood and moves on to the new round. The process ends with the node of highest identity number sending an ELECTION message to itself and becoming the leader, and then, all nodes learning the identity of the leader thanks to the ELECTED message.

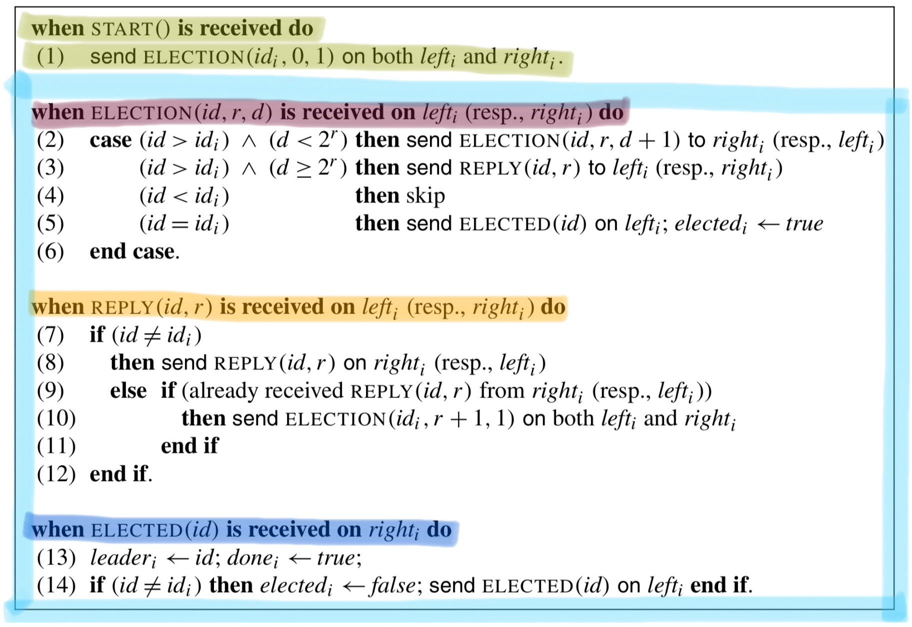
```
when START() is received do
(1)   send ELECTION($id_i$, 0, 1) on both $left_i$ and $right_i$.

when ELECTION($id, r, d$) is received on $left_i$ (resp., $right_i$) do
(2)   case ($id > id_i$) ∧ ($d < 2^r$) then send ELECTION($id, r, d+1$) to $right_i$ (resp., $left_i$)
(3)        ($id > id_i$) ∧ ($d \geq 2^r$) then send REPLY($id, r$) to $left_i$ (resp., $right_i$)
(4)        ($id < id_i$)                  then skip
(5)        ($id = id_i$)                  then send ELECTED($id$) on $left_i$; $elected_i \leftarrow true$
(6)   end case.

when REPLY($id, r$) is received on $left_i$ (resp., $right_i$) do
(7)   if ($id \neq id_i$)
(8)      then send REPLY($id, r$) on $right_i$ (resp., $left_i$)
(9)      else if (already received REPLY($id, r$) from $right_i$ (resp., $left_i$))
(10)             then send ELECTION($id_i, r+1, 1$) on both $left_i$ and $right_i$
(11)          end if
(12)  end if.

when ELECTED($id$) is received on $right_i$ do
(13)  $leader_i \leftarrow id$; $done_i \leftarrow true$;
(14)  if ($id \neq id_i$) then $elected_i \leftarrow false$; send ELECTED($id$) on $left_i$ end if.
```

Figure 12: Leader election algorithm for bidirectional ring network

4.1.2 Modeling process

At the first look, the algorithm can be divided into two big parts: receiving START() and a loop for receiving other messages.

Upon receiving START() from exterior, a process$_i$ can send $ELECTION(id_i, 0, 1)$ to $left_i$ and $right_i$ (places `election_id_0_1_to_left` and `election_id_0_1_to_right`) or it may enter the loop waiting for new messages (place `newmsg`). Generally, there are 3 types of messages that can be sent to a process: ELECTION(), REPLY(), ELECTED(), all of which can be represented by transitions whose input places are `newmsg` of the receptor and, respectively, `election`..., `reply`..., `elected`... of the sender.



```
when START() is received do
(1)  send ELECTION($id_i$, 0, 1) on both $left_i$ and $right_i$.

when ELECTION($id, r, d$) is received on $left_i$ (resp., $right_i$) do
(2)  case ($id > id_i$) ∧ ($d < 2^r$) then send ELECTION($id, r, d+1$) to $right_i$ (resp., $left_i$)
(3)       ($id > id_i$) ∧ ($d \geq 2^r$) then send REPLY($id, r$) to $left_i$ (resp., $right_i$)
(4)       ($id < id_i$)                  then skip
(5)       ($id = id_i$)                  then send ELECTED($id$) on $left_i$; $elected_i \leftarrow true$
(6)  end case.

when REPLY($id, r$) is received on $left_i$ (resp., $right_i$) do
(7)  if ($id \neq id_i$)
(8)    then send REPLY($id, r$) on $right_i$ (resp., $left_i$)
(9)    else if (already received REPLY($id, r$) from $right_i$ (resp., $left_i$))
(10)          then send ELECTION($id_i, r+1, 1$) on both $left_i$ and $right_i$
(11)        end if
(12) end if.

when ELECTED($id$) is received on $right_i$ do
(13) $leader_i \leftarrow id$; $done_i \leftarrow true$;
(14) if ($id \neq id_i$) then $elected_i \leftarrow false$; send ELECTED($id$) on $left_i$ end if.
```

Figure 13: Initial take on the algorithm

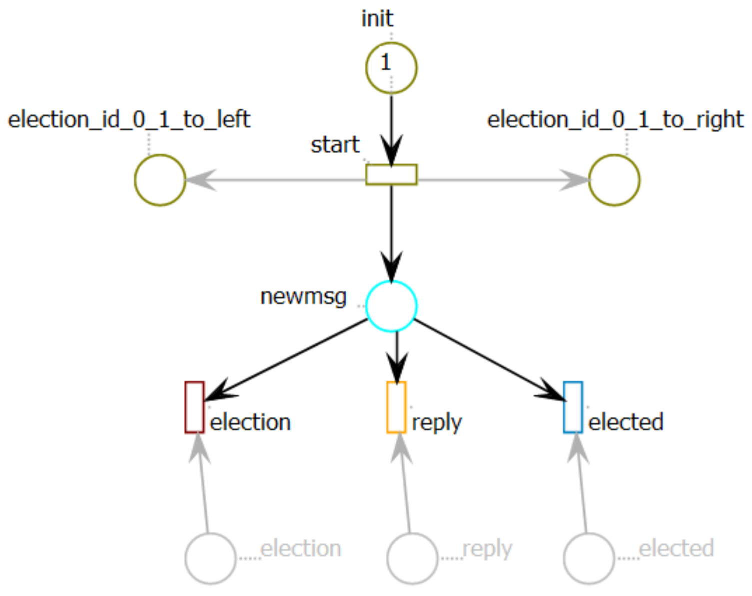
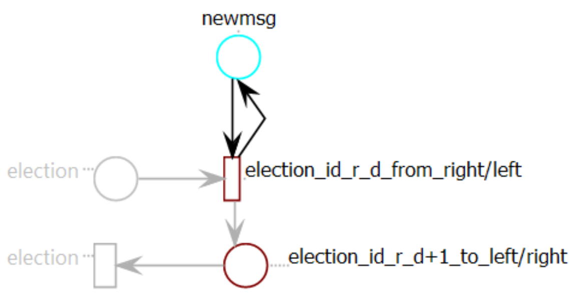


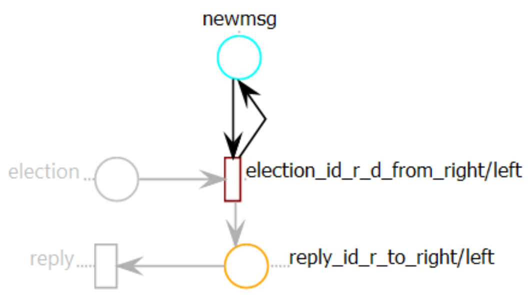
Figure 14: Initial stage of a process

This initial part will be the same for every process.

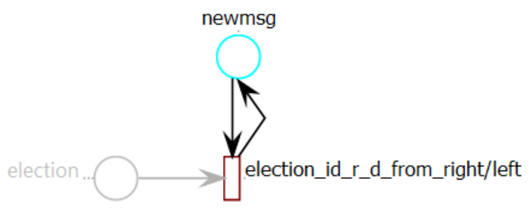
Next up, each message received can be broken down into small cases from which models can be built accordingly. Let's start with ELECTION() messages. In all cases, after dealing with the message, a process$_i$ needs to go back to the loop to continue receiving new messages (because $done_i \leftarrow true$ never happens). The `election` transition can result in a place (for preparing to send a message to another process (case a,b,d)) or nothing (for skipping this ELECTION() message (case c)).


newmsg
election
election_id_r_d_from_right/left
election
election_id_r_d+1_to_left/right

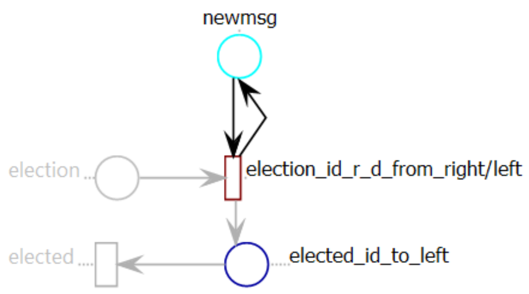
(a) Case $(id > id_i) \wedge (d < 2^r)$



(b) Case $(id > id_i) \wedge (d \geq 2^r)$



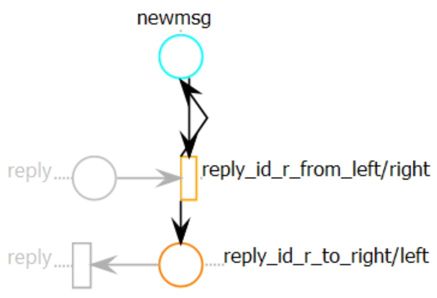
(c) Case $(id < id_i)$



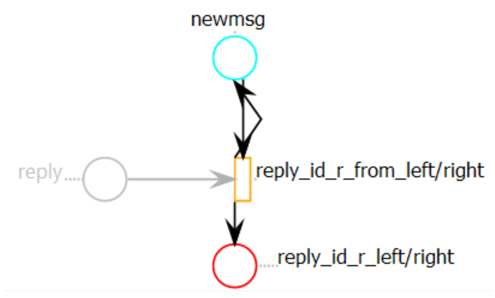
(d) Case $(id = id_i)$

Figure 15: Models of different scenarios upon receiving ELECTION() message

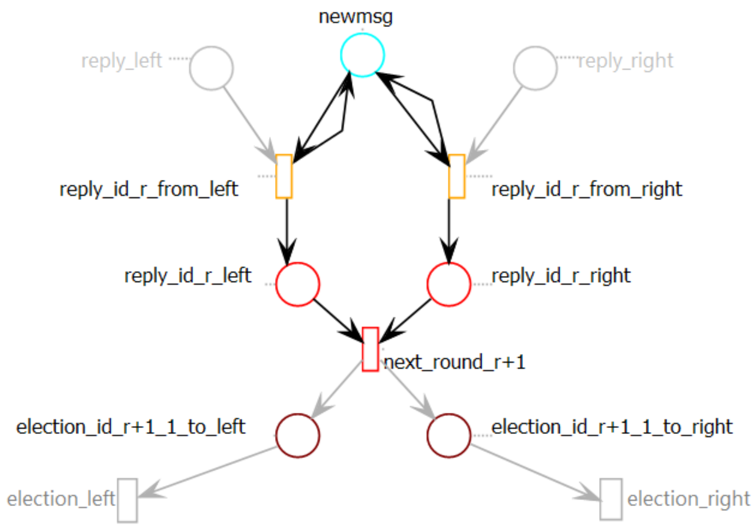
Similarly to ELECTION() messages, a `reply` transition representing a REPLY() message needs to lead back to the `newmsg` place of the loop. There are 3 scenarios possible for the 2 conditions at line 7 and 9:
- case a (true-NA), results in preparing to send a message to another process
- case b (false-false), leads to a flag for keeping this REPLY() in memory
- case c (false-true), results eventually in a transition `next_round`, meaning the process can move on to the next round and continue sending ELECTION() message carrying its own identity



(a) Case ($id \neq id_i$)



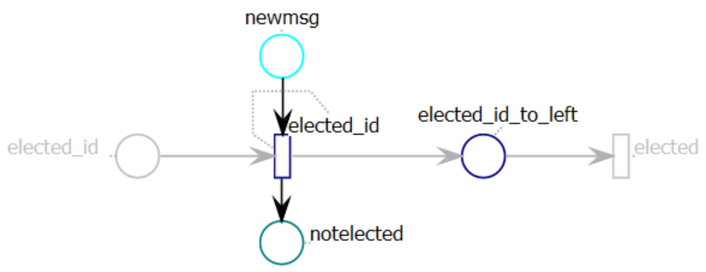
(b) Case ($id = id_i$) and not yet received REPLY() from other side



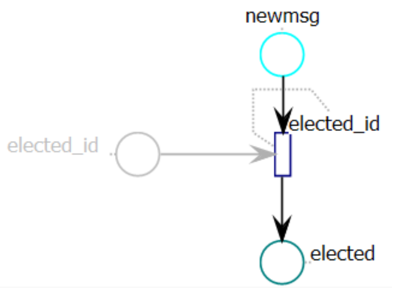
(c) Case($id = id_i$) and already received REPLY() from other side

Figure 16: Models of different scenarios upon receiving REPLY() message

The last scenarios are when a process received ELECTED(), it then stops accepting new messages ($done \leftarrow true$) and for this reason, the transition `elected` does not go back to the `newmsg` loop. At this stage, the process will know whether it is elected, the transition leads then accordingly to the final place `elected` or `notelected`. If a process is not elected, it will transfer the message to its left neighbor.



(a) Case $(id \neq id_i)$



(b) Case $(id = id_i)$

Figure 17: Models of different scenarios upon receiving ELECTED() message

Generally, for a ring of n processes, $r \in \{0, ..., \lceil log_2(n) \rceil\}$ and in each round r, $d \in \{1, ..., 2^r\}$. By the end of the operation, `notelected` will hold $(n - 1)$ tokens and `elected` only one.

The simplest scenario for a 2-process ring network can be obtained by enumerating all the message sent by each process and attaching each one to an above-mentioned scenario. In the model below, process_1 to the left and process_2 to the right both go from their initial stage through 2 rounds of communication to learn their final status at the last stage. However, the messages they receive vary according to their identity.

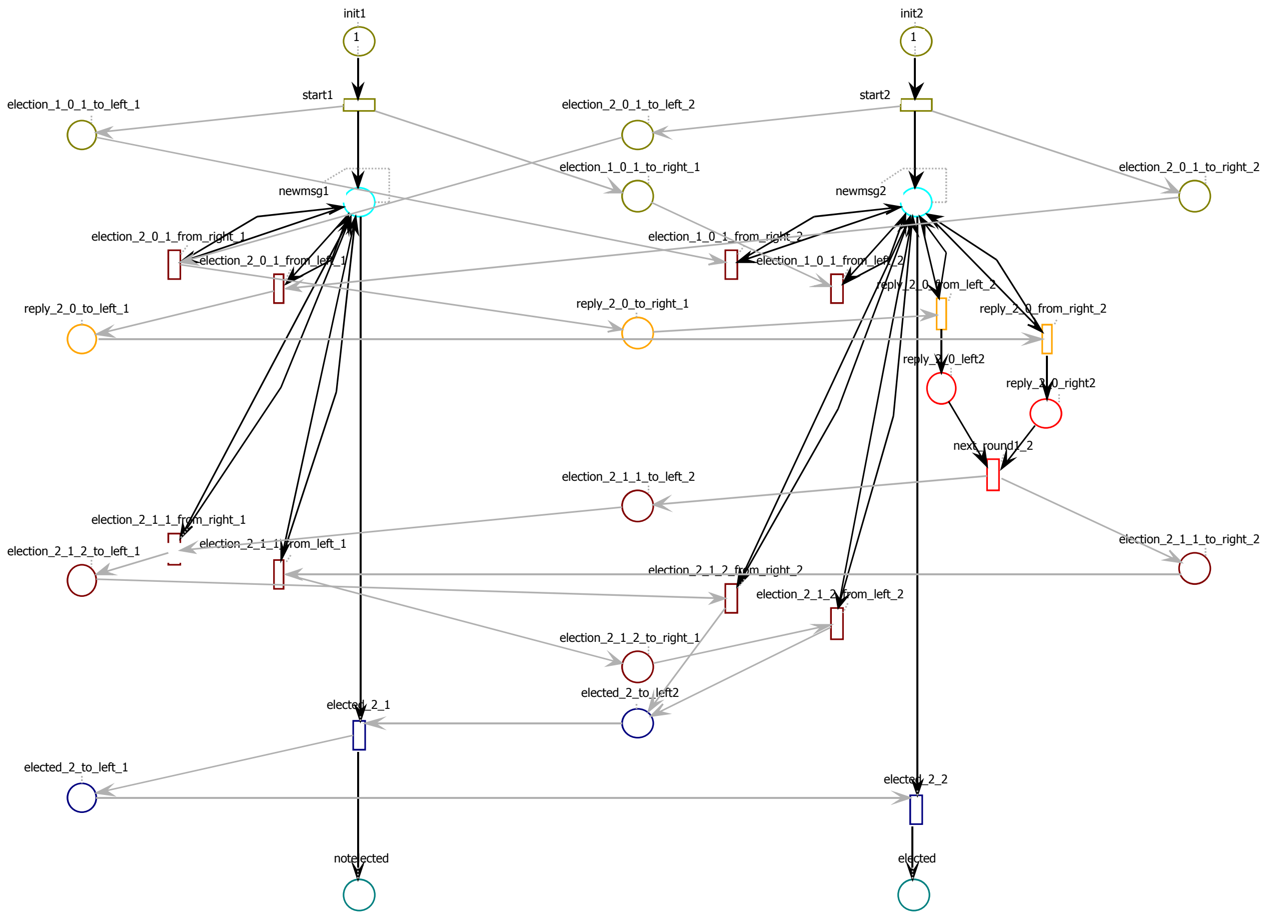


Figure 18: P/T net model of leader election on bidirectional 2-process ring network

Lastly, we can go further by expanding all `election` and `reply` transitions.

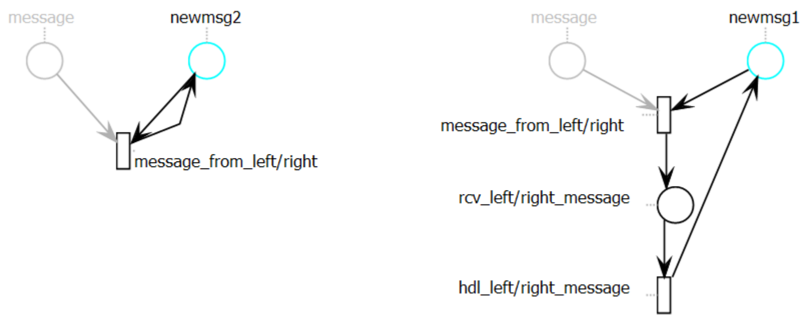


Figure 19: Expansion of `election` and `reply` transitions

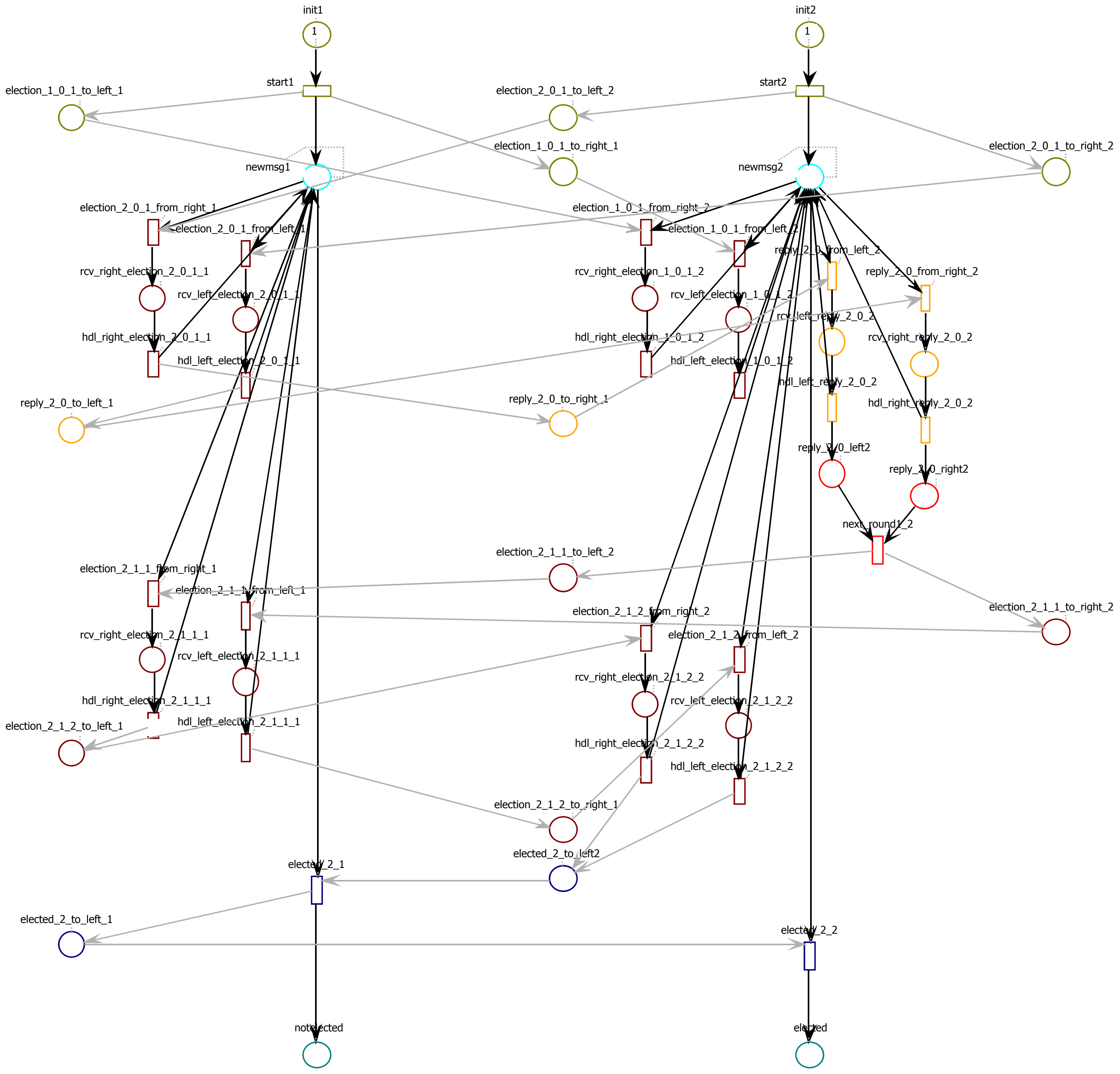

init1
init2
start1
start2
election_1_0_1_to_left_1
election_2_0_1_to_left_2
election_1_0_1_to_right_1
election_2_0_1_to_right_2
newmsg1
newmsg2
election_2_0_1_from_right_1
election_2_0_1_from_left_1
election_1_0_1_from_right_2
election_1_0_1_from_left_2
reply_2_0_from_left_2
reply_2_0_from_right_2
rcv_right_election_2_0_1_1
rcv_left_election_2_0_1_1
rcv_right_election_1_0_1_2
rcv_left_election_1_0_1_2
rcv_left_reply_2_0_2
rcv_right_reply_2_0_2
hdl_right_election_2_0_1_1
hdl_left_election_2_0_1_1
hdl_right_election_1_0_1_2
hdl_left_election_1_0_1_2
hdl_left_reply_2_0_2
hdl_right_reply_2_0_2
reply_2_0_to_left_1
reply_2_0_to_right_1
reply_2_0_left2
reply_2_0_right2
next_round1_2
election_2_1_1_to_left_2
election_2_1_1_from_right_1
election_2_1_1_from_left_1
election_2_1_2_from_right_2
election_2_1_1_to_right_2
election_2_1_2_from_left_2
rcv_right_election_2_1_1_1
rcv_left_election_2_1_1_1
rcv_right_election_2_1_2_2
rcv_left_election_2_1_2_2
hdl_right_election_2_1_1_1
election_2_1_2_to_left_1
hdl_left_election_2_1_1_1
hdl_right_election_2_1_2_2
hdl_left_election_2_1_2_2
election_2_1_2_to_right_1
elected_2_to_left2
elected_2_1
elected_2_to_left_1
elected_2_2
notelected
elected

Figure 20: Final P/T net model of leader election on bidirectional 2-process ring network

The generator for this type of model uses the same principle of enumerating messages and attaching small models onto the whole ring. After generating the initial stage for all processes, it will list, for each round r, all the possible consequences of the ELECTION() messages sent by a competitor process$_i$ to its neighborhood and then, connect all the fitting models with each others and onto the main ring (arcs in gray). The chosen ring configuration for this project is an increasing one (... $\leftrightarrow$ 1 $\leftrightarrow$ 2 $\leftrightarrow$ 3 $\leftrightarrow$ ...), though, the generator can be modified very easily to fit other configurations.
The result will consist of these types of place and transition:
Places
• `init`$_i$, initial state of process p_i
• `newmsg`$_i$, p_i available to receive messages
• `election_id_r_d/reply_id_r_to_right/left_i`, p_i ready to send ELECTION(id,r,d)/ REPLY(id,r) to $right_i/left_i$
• `rcv_right/left_election_id_r_d/reply_id_r_i`, p_i just received ELECTION(id,r,d)/ REPLY(id,r) from $right_i/left_i$
• `reply_i_r_right/left_i`, p_i has already received REPLY(i,r) from $right_i/left_i$
• `elected_id_to_left_i`, p_i ready to send ELECTED(id) to its $left_i$
• `elected`, elected leader
• `not_elected`, not elected processes

Transitions
• `start`$_i$, p_i receiving START() from exterior
• `election_id_r_d/reply_id_r_from_right/left_i`, p_i receiving ELECTION(id,r,d)/ REPLY(id,r) from $right_i/left_i$
• `hdl_right/left_election_id_r_d/reply_id_r_i`, p_i handling ELECTION(id,r,d)/ REPLY(id,r) sent by $right_i/left_i$
• `next_round`$_{r\_}i$, p_i moving to round r
• `elected_id_i`, p_i receiving elected(id) from $right_i$

4.2 Bellman-Ford's shortest path algorithm

4.2.1 The algorithm (Raynal, 2012)

Secondly, I will turn to an algorithm that I have studied in the sequential context: Bellman-Ford's Shortest Path Algorithm, and try to model its distributed adaptation. The goal is that each vertex in a non-oriented weighted graph learns the shortest path from itself to every other vertex. It is based on the principle of dynamic programming where each vertex p_j stores an array $length_i[1..n]$ such that $length_i[k]$ will contain the length of the shortest path from p_i to p_k (initially, $length_i[i] = 0$ and $length_i[k] = +\infty$ for $k \neq i$). When a node p_i updates its $length_i$ (by comparing the length of the newly learned and the old path and updating if the new path is shorter), it will signal all of its neighbors p_j to update their own $length_j$ array. For a node p_i, $l_{g_i}[j]$ denotes the length associated with the channel (i, j) and $routing\_to_i[1...n]$ is such that $routing\_to_i[k] = j$ means that p_j is a neighbor of p_i on a shortest path to p_k.

The algorithm ends when each node acquires the knowledge of the shortest path to other nodes, there will not be anymore UPDATE message as the condition in which $update_i$ becomes *true* can no longer be satisfied.

```
when START() is received do
(1)  for each $j \in neighbors_i$ do send UPDATE($length_i$) to $p_j$ end for.

when UPDATE($length$) is received from $p_j$ do
(2)  $updated_i \leftarrow false$;
(3)  for each $k \in \{1, \dots, n\} \setminus \{i\}$ do
(4)      if ($length_i[k] > \ell g_i[j] + length[k]$)
(5)          then $length_i[k] \leftarrow \ell g_i[j] + length[k]$;
(6)               $routing\_to_i[k] \leftarrow j$;
(7)               $updated_i \leftarrow true$
(8)      end if
(9)  end for;
(10) if ($updated_i$)
(11)     then for each $j \in neighbors_i$ do send UPDATE($length_i$) to $p_j$ end for
(12) end if.
```

Figure 5: Distributed adaptation of Bellman-Ford's Shortest Path algorithm

4.2.2 Modeling process

Unfortunately, the modeling of this algorithm could not be done in time. An naive start would be something resembling the model of leader election, an similar initial stage with a loop for receiving new messages.

5 Conclusion

Through this project, I have learned the concept of Petri nets some of its simple applications, which helped me gain a better understanding of the field of concurrent programming. The work of modeling has taught me to capture the essentials of algorithms, and from that, I was able to visualize and look at them in a new way. Even though, due to lack of time, I could only create two simple models, I hope they are interesting enough and will be of use for the MCC.

References

C.Girault, R. (2001). *Petri nets for systems engineering.* Springer-Verlag.

Raynal, M. (2012). *Distributed algorithms for message-passing systems.* Springer.