Delft University of Technology

Quantitative Evaluation of Embedded Systems

IN4390

Lab 3(Optional Part)

Authors: Jiaxuan Zhang(5258162)
Yiting Li(5281873)
Group ID: (26)

February 3, 2021

1 Question 4

1.1 Modified Petri Nets

The modified Petri Nets is shown in the Figure 3. In this Petri nets,

1. Each server has a 5-size buffer, when buffer is full (place *inactivated*? has one token, place $buffer$ is empty), server will not receive data, the normal processing will be deactivated. That simulates the case that when a server is full, the listener will be informed and not send data to this server.

2. When the server's queue is full, the transition *processing*′ can be fired. After that, place *inactivated*? will be empty, place $buffer$ has one token, then this server can receive data again.

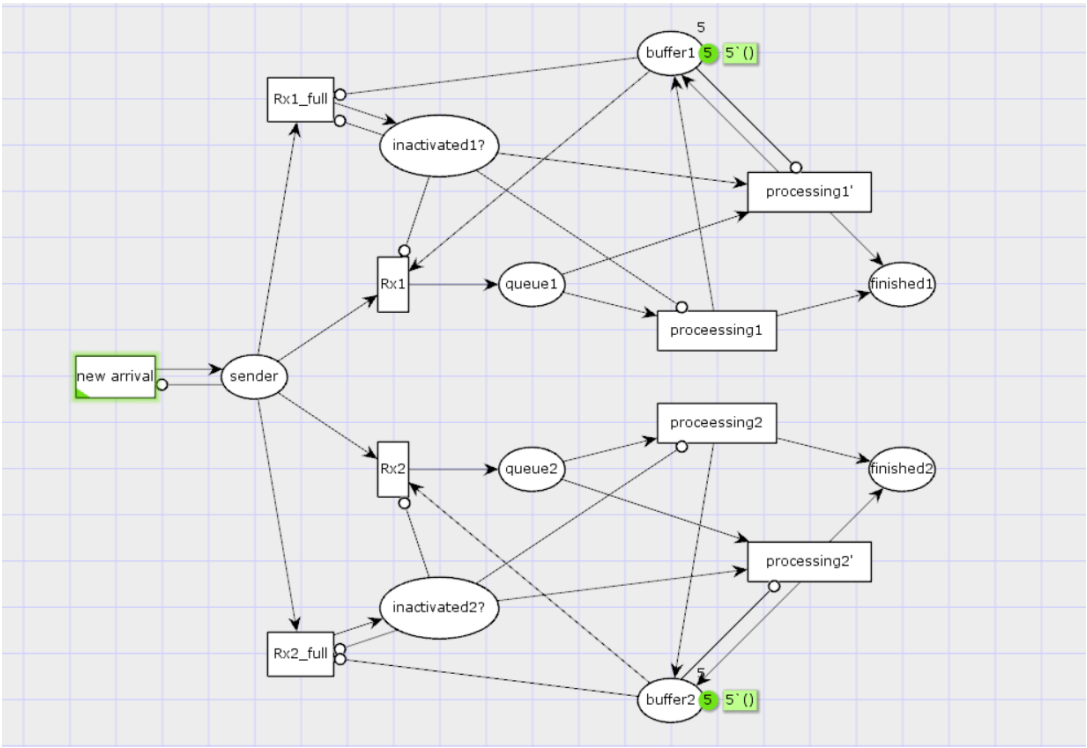

Figure 1: Petri Nets of modified model

1.2 Deadlock Property

In the modified Petri Nets, we can determine whether the whole system has a deadlock by examining whether each server (server 1 or server 2) has a deadlock. The reason for that is:

1. First, we assume that each server has no deadlock,that is each server will not be deactivated after its queue is full. When server has a full queue, it will not receive any data but can process the waiting data, and after a period of time of processing, this server can receive data again.

2. Second, if the assumption holds, we can find that there is at least one server can receive data and both servers can process data, which means that the system will not be stuck and enter deadlock state. Because the average data arriving rate is 10 per sec and the average processing rate is 12 per sec (if both servers will not be deactivated after their queues are full), so when a server is filled with data, the other will be not full.

3. Last, we find that if the assumption holds, the system will not enter into deadlock state, as a result, we can check whether this assumption holds or not.

To check this assumption, we may simplified our Petri Nets, because the original Petri Nets is quite hard to draw a reach-ability graph. We ignore server 1 for example, also we ignore place *sender* and place $finished2$ and related transitions, because deadlock will not appear at this two places. Then we can obtain the following simplified Petri Nets:

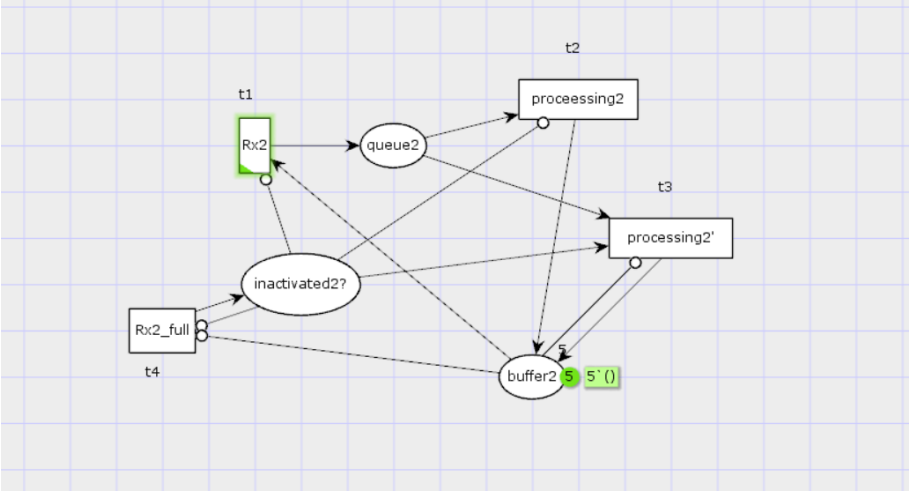

Figure 2: Simplified Petri Nets for checking deadlock

We can obtain its reach-ability graph, where m is ($queue$ 2, $buffer$ 2, $inactivated2?$).

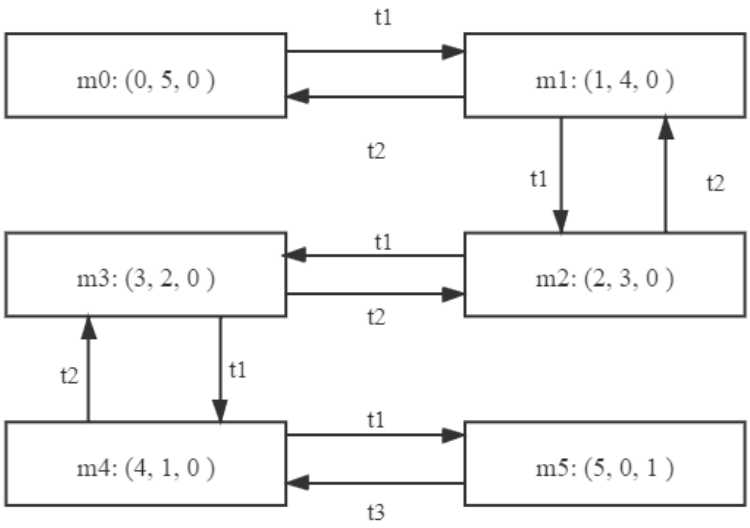

Figure 3: Simplified Petri Nets for checking deadlock

Form this reach-ability graph, we can see that, a transition that convert one marking to another marking exist, therefore, the modified system is deadlock free.

2 Question 5

2.1 Sub-Question 1

We only modified `node.py` to implement the change. We modified `node.py` with these steps.

1. After receiving a token, the listener will check its queue. If token in the queue is equal to the maximum capacity, it will send a message to notify the talker and will use a flag to record the notification;

2. After removing a token, the listener will check the queue. If the token is less than the maximum capacity and the notification flag is true, it will send a message to tell the talker that it is ready to receive tokens again.

3. Talker makes a copy of the listener's weight at the beginning. When sending a token, the target is chosen according to the copy of the weight. When a talker is notified a listener is nearly full, it will change the weight of the listener to 0. This means the listener can still be chosen, but because the talker will check the weight of the chosen node, if the weight is 0, the token will be aborted. When a talker is notified a listener is ready to receive a token again, it will change the weight to the value in the copy of the initial weight. Why we use this mechanism will be explained.

4. we reuse the topic `state` and modified the message in the state. If a listener wants to notify a talker it is nearly full, it will send a message `<listener_id>:<talker_id>:<1>`. If a listener wants to notify a talker it is ready to receive token again, the message will be `<listener_id>:<talker_id>:<0>`. If a talker is deactivated, it will send `<listener_id>:<0>:<2>`. A talker who received a message on this topic will check the talker id part to determine whether it should make some changes.

The reason why we use the third step is to simplify the analysis. The function `random.choices()` in Python will not choose a node with weight 0 if there is another node with non-zero positive weight. Without the third step, we have two ways to analyze this process:

1. With two M/M/1 queue with capacity 5. However, because the property of `random.choics()`, if one node is nearly full and we simply change the weight to 0, all receiving burden will come to another queue, which means the arrival rate will change to 2 times of initial speed. The Markov chain will be dynamically and the parameter is time variant (more specific, event-triggered variant), and it will be very hard to analyze.

2. Model the process in one Markov chain with each state as `{queue 1 number,queue 2 number}`, the Markov chain will have 36 states and we can only calculate it with a really large generation matrix.

Both of these methods are too complex, so we use the above modification. That means the initial Markov chain in Question 2 is still suitable to use with this simplification.

2.2 Sub-Question 2

The Markov Chain is unchanged, which means the utilization, throughput, and response time are unchanged. However, the dependability is improved. That is because with the modification only the situation in which all two queues is full will lead to a crash. The crash probability is :

$$\begin{aligned} P &= P(\text{two listener's token} = 5) \\ &= 0.1008 \cdot 0.1008 \\ &= 0.0102 \end{aligned} \tag{1}$$

Therefore the dependability is improved.