Peterson's Algorithm in a Distributed Environment

Jason Wu Vincent Rahli

November 3, 2011

1 Background Information

There are two processes (P_1 and P_2) and one shared resource (R), each in separate locations. The processes have their own local memory and may communicate with each other through messages. In our environment, messages can be delayed and received in a different order than they were sent. However, messages cannot be lost or duplicated. Access to the shared resource is unsynchronized and this leads to the problem we wish to solve: how can the two processes coordinate with each other to safely access the shared resource?

2 A Message Passing (MP) Version of Peterson's Algorithm

The overall idea is to use a token to control access to the shared resource. Initially, one of the processes is given the token at the start of the algorithm. Whenever a process has the token, it may access the shared resource. Otherwise, it must request the token from the other process.

It may be helpful to think of the token as a key that grants a process access to the shared resource. As time passes with processes accessing the shared resource, the key gets passed back and forth between the two processes.

The behavior of a process may be succinctly described with the following rules:

1. When a process wants to enter its critical section (i.e., access the shared resource):
   (a) If the process has the token, the process may enter its critical section.
   (b) If the process does not have the token, it will send a request to the other process for the token and wait for the token before entering its critical section.
2. When a process receives a request for the token:
   (a) If the process is not in its critical section, it will immediately send the token.
   (b) If the process is in its critical section, it will delay sending the token until it has left its critical section.

2.1 Pseudocode

```
EnterCriticalSection():

busy = true

if (not token)
        send_requestMSG()
        wait_for_tokenMSG()
        token = true

/* Critical Section */
/* Access Shared Resource */

busy = false

if (needToReply)
        token = false
        send_tokenMSG()
        needToReply = false
```

(a) Function called when a process wants to access the shared resource

```
HandleRequestMSG():

if (busy)
        needToReply = true
else
        token = false
        send_tokenMSG()
```

(b) Function called when a process receives a request message

Figure 1: Pseudocode

Pseudocode for the algorithm consists of two functions: *EnterCriticalSection* and *HandleRequestMSG*. *EnterCriticalSection* is called by the process whenever it would like to access the shared resource. *HandleRequestMSG* is used whenever the process receives a request from the other process.

The functions *send_requestMSG* and *send_tokenMSG* respectively send a request message and a token message to the other process. The function *wait_for_tokenMSG* allows the process to block until it receives a token message.

The pseudocode given in Figure 1 assumes the function *HandleRequestMSG* (Figure 1b) is atomic. Each process has three boolean variables: *token*, *busy*, and *needToReply*. *token* is true if and only if the process has the token. *busy* is true only when the process is in (or wants to enter) the critical section. *needToReply* is true if the process has received a request while it was in (or wanted to enter) the critical section. *token* is intialized to true for only one of the processes. Both *busy* and *needToReply* are initially false.

2.2 Correctness, Liveness, and Fairness

We provide informal arguments for the correctness, liveness, and fairness of the MP version.

2.2.1 Correctness

Notice that a process can only enter the critical section if it has the token. Also, notice that a process may only give up the token if it is not in the critical section. Then, the correctness of the algorithm depends on the existence of only one token at any point in time. This unique token may be in the hands of one of the processes or it may be in the network as a message.

Initially, this is true because only one of the processes starts off with the token. Throughout the algorithm, a process can only obtain the token if it receives a token message. This token message must have been sent by the other process (and it must have the token in order to send a token message). When the other process sent the token message, it gave up its token. Also, notice that the token message can neither be duplicated nor lost in the network. Both the actions of the processes and the environment preserve the existence of only one token.

2.2.2 Liveness

Suppose P_1 wants to use the shared resource. If P_1 has the token, progress can be made (by P_1). If P_1 does not have the token, it will send a request to P_2. If P_2 is in its critical section, progress can be made (by P_2). If P_2 is not in its critical section, P_2 will send the token to P_1.

Now, the only way for no progress to be made is if the token is passed back and forth without either process accessing the shared resource. However, since P_1 wants to use the shared resource (i.e., it is in its critical section), it will only send the token after it has made progress (i.e., left its critical section).

2.2.3 Fairness

Suppose P_1 has the token and P_2 would like to access the shared resource. According to the algorithm, P_2 will send a request to P_1 for the token. When P_1 receives the request, it will act depending on whether it is in its critical section or not. If P_1 is not in its critical section, it will send the token immediately. If P_1 is in its critical section, it will send the token as soon as it leaves the critical section. In either case, P_2 will eventually receive the token and be able to access the shared resource.

Consider the case where P_1 sends a request right after sending the token and P_2 receives the request before receiving the token. According to the algorithm, P_2 will delay sending the token until after it has left the critical section, guaranteeing fair access to the shared resource.

2.3 Comparison to the Original Version

In the original version of Peterson's algorithm, if both processes want to use the shared resource, the first process that enters its critical section gets to go first. The second process is guaranteed access before the first process can use the resource again.

In the MP version, if both processes want to use the shared resource, the process with the token gets to go first. The second process is guaranteed access as soon as the first process receives the request and is aware that the second process wants access.

2.4 Fault Tolerence

We make a small note about the fault tolerence in the original version and the MP version of Peterson's algorithm. In the original version, if a process failed while it was in the critical section, the other process will be locked out of the shared resource. In the MP version, the same will happen. However, in addition, if a process failed while it had the token (regardless of whether it was in the critical section or not), the other process will be locked out of the shared resource. Obviously, it appears that this version has weaker fault tolerence than the original version.

We believe that this is the best one can do given the properties of the environment. For any distributed version of Peterson's algorithm that does not use a token, if a process failed at any time, the other process will always be locked out of the shared resource. The other process will attempt to coordinate with the failed process in order to safely access the shared resource, but will end up waiting forever for a reply to arrive. It is impossible for a process to distinguish between the failure of the other process, an infinitely-delayed message, or the fact that the other process is staying in its critical section.