Periodic Processor Group Membership Protocol

Hailey Hultquist
University of Texas at Austin
hultquist@utexas.edu

Nidhi Kadkol
University of Texas at Austin
nidhik@cs.utexas.edu

Abstract

This project aims to implement the periodic broadcast membership protocol as described by Flaviu Cristian [1]. We implement the join handling as well as failure handling cases for this protocol and analyze their performance in terms of number of messages sent. Implementing the join and failure handling of this protocol gives an insight into the nuances of how they work. We had to make many decisions in implementing these protocols, the specifics of which were not mentioned in [1].

1 Introduction

In any computing system or machine setup, there will be a number of interacting processes which work together to seamlessly produce the desired output. Sometimes, failure of a particular component may occur, in which case unpredictable situations may happen. This can lead to disastrous consequences if not accounted for at the time of system design. Failure semantics [2] need to be agreed upon and kept in place for such situations.

In the case of failure of a component, until that component recovers it is unable to perform any useful work. In such a situation, it should not be a part of the group with which it was working. The rest of the processes that belong to this group need to be made aware of this fact, else they will continue to operate as though the said component is able to take on its share of work as usual. This can lead to incorrect or even dangerous outcomes, and thus there is a need for group membership protocols so that processes can keep track of other members in their group. Specific join and failure handling algorithms need to be followed to maintain the correctness of the system. These algorithms have been implemented in a number of important real-world applications, for example in an Air Traffic Control System [3].

In this project, we implement the periodic broadcast membership protocol as explained by Cristian [1]. We assume that all the processor clocks are synchronized, so we do not have to account for clock drift. We analyze the total number of messages sent in different cases of this protocol, as well as implement our own modifications to improve the efficiency of the algorithm.

2 Implementation

All code is written in C++11, making use of the $pthread$ library for simulating real world message sending. We also use the Standard Template Library for implementing data structures in a clean and efficient manner. The code is publicly available on GitHub.

For efficiency of implementation, we deviated slightly from Cristian's specifications. We created an abstract `Group` class to manage process memberships, and set up atomic broadcast as simply a sequence of direct messages to every other process in the system rather than as described in [4]. More details are explained in sections 3 and 4.

3 Modules

In this section we describe how we break down the broadcast protocol into separate modules so we can approach the implementation in an object-oriented manner. We abstract the code into a `Process` class, a `Group` class, and a `Message` class.

3.1 Process

Every process in the system is an object of this class. It has private members `process_id` and `group_id`. In addition, it also has a boolean `active` flag that indicates the process' failure status. Each process also has a list of all the other members in its group which is modified and re-sized depending on failures and new group requests. Lastly, it maintains a boolean `checked` array which is used at the time of failure checking.

3.2 Group

We created a `Group` class that serves to notify each member process when it is time for a failure check to occur. This simplifies the protocol because each `Process` instance will not be required to maintain its own clock but instead can depend on the notification from its group. This does not change the outcome of the protocol in any way since a failed process will ignore the notification. We also add a modification to the original protocol by assigning each `Group` instance an integer to serve as an identifier, rather than the time-stamp $V + \Delta$. We use this `group_id` to guarantee that two groups created concurrently will not create a collision later.

3.3 Message

The Message class has attributes `content`, `group_id`, and `timestamp`. The `content` attribute can be one of `NEW_GROUP` (to request a new group), `PRESENT_ADD` (to join a group in response to a new group request), or `PRESENT_CHECK` (to confirm the process is still active at membership check time). Since all processes receive all atomic broadcasts, `group_id` is used to help the receiving process interpret the message correctly.

4 Events

4.1 New Group

We set up the implementation such that we can specify the number of processes as a program argument. We arbitrarily set the process with process id 0 to make the first request for a new group using an atomic broadcast. This means that process 0 sends out a message `NEW_GROUP` to all processes in the global process list. On receiving this message, if the process is active, it responds with a `PRESENT_ADD` message, indicating its willingness to join this new group. Thus, all the independent processes respond to its atomic broadcast and join the group. As they do so, they collect and store a list of other processes that are joining the same group.

4.2 Failure Simulation

To simulate a failed processor, we assign each process a boolean `active` flag that indicates whether the process has crashed. The simulator executes a "malicious" thread every t seconds that will kill each processor with probability p. Generally p should be very small in a real-world system, since a process crash is an unexpected event.

4.3 Atomic Broadcast

We represent an atomic broadcast as a sequence of direct messages to every other process in the system. As described in [4], an atomic broadcast is much more complicated and thorough in order to guarantee atomicity and arrival. However, since C++ is a reliable language, we feel that it is safe to assume that our message thread will reach its desired recipient without implementing the redundancy required by less reliable systems.

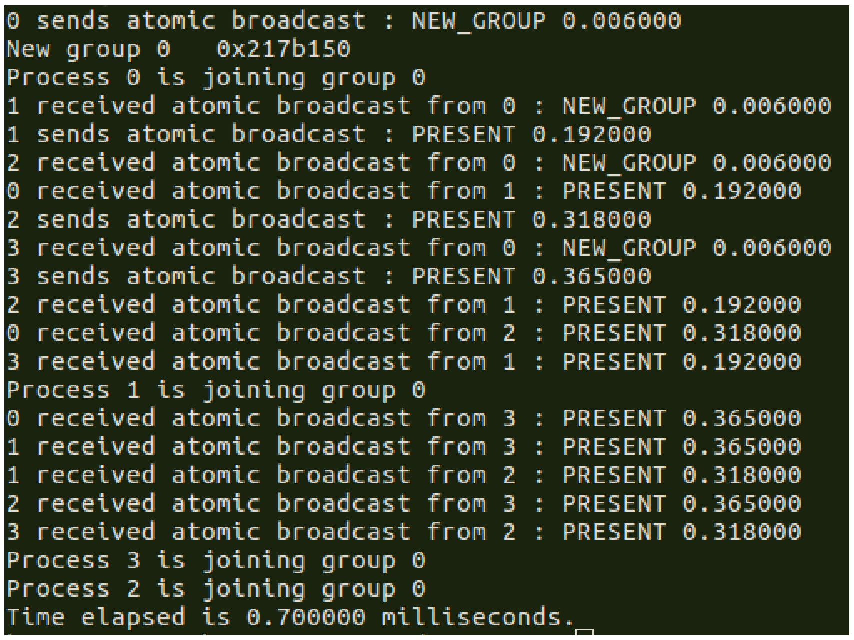
4.4 Failure Checking

After joining a new group, each process sets up a failure checking framework by setting the values of all the members in its boolean `checked` array to False. Following the protocol, it will then send its own atomic broadcast with the message content `PRESENT_CHECK` before beginning to listen for broadcasts from other processes in its group. As and when the process receives a `PRESENT_CHECK` message from another process, it will set the corresponding element of its `checked` array to True. After Δ time units, it will evaluate the results and return its conclusion to the `Group` class object. The `Group` object will confirm that all processes are in agreement (Since the atomicity of the atomic broadcast is guaranteed by design [4], if all correct processes do not return the same results, this is an indication that there is a serious problem with the system).

In case of a failure, the `Group` object will delegate an arbitrary correct process to create a new group. This also deviates from the original protocol description in [1], where all processes that detect a failed process will send out an atomic broadcast to create a new group. We reasoned that this will lead to many redundant atomic broadcasts, since if one process fails, all the other correct processes will send out atomic broadcasts to create a new group. It is important to note that each atomic broadcast is inherently very expensive as described in [4]. Additionally, the resolution of this situation is left ambiguous in [1] as it does not describe the case where multiple processes create new groups concurrently. One solution to this problem involves labelling the new group requests with the `process_id` of the initiator so the receiving processes can settle on a single new group to join. Our implementation decision to create a `Group` abstraction elegantly removes these issues and redundancies.

5 Testing

We ran a series of tests to verify the correctness of our protocol. Shown below is an example of the output we received from a short group creation simulation with 3 processes. The number displayed at the end of each line is the time-stamp of the message that was sent/received.



```
0 sends atomic broadcast : NEW_GROUP 0.006000
New group 0   0x217b150
Process 0 is joining group 0
1 received atomic broadcast from 0 : NEW_GROUP 0.006000
1 sends atomic broadcast : PRESENT 0.192000
2 received atomic broadcast from 0 : NEW_GROUP 0.006000
0 received atomic broadcast from 1 : PRESENT 0.192000
2 sends atomic broadcast : PRESENT 0.318000
3 received atomic broadcast from 0 : NEW_GROUP 0.006000
3 sends atomic broadcast : PRESENT 0.365000
2 received atomic broadcast from 1 : PRESENT 0.192000
0 received atomic broadcast from 2 : PRESENT 0.318000
3 received atomic broadcast from 1 : PRESENT 0.192000
Process 1 is joining group 0
0 received atomic broadcast from 3 : PRESENT 0.365000
1 received atomic broadcast from 3 : PRESENT 0.365000
1 received atomic broadcast from 2 : PRESENT 0.318000
2 received atomic broadcast from 3 : PRESENT 0.365000
3 received atomic broadcast from 2 : PRESENT 0.318000
Process 3 is joining group 0
Process 2 is joining group 0
Time elapsed is 0.700000 milliseconds._
```

Figure 1: Simulation of group creation using 3 processes.

We also recorded the number of atomic broadcasts that were sent during each test. With 50 processes that are killed with probability $p = 0.05$ every 12 seconds and a failure check every 5 seconds, there were a total of **2630** atomic broadcasts sent over the 150 second testing period. When we increased the failure probability to $p = 0.1$, there were **2495** atomic broadcasts sent. As expected, we see that as processes drop out of the group, the number of atomic broadcasts sent decreases since those processes will no longer send PRESENT_CHECK messages. It is important to specify that once a process has failed, we do not allow it to become active again for the remainder of the simulation.

6 Conclusion

We implemented the Periodic Broadcast Protocol and successfully simulated it in a real-world setting by arbitrarily killing processes during the course of message sending. We see that as a higher number of processes fail, the total number of messages sent in the system decreases. This project leads us to conclude that while Periodic Broadcast is an effective membership protocol, it is inefficient in the number of messages sent (and consequently, time taken). Especially as the number of processes increases, we prefer a protocol that does not depend so heavily on atomic broadcasts to reduce the overhead cost of maintaining group membership.

Acknowledgments

We are grateful to Prof. Aloysius K. Mok for his thorough teaching and discussion of this topic during the course lectures, as well as his patience in answering our many questions about different aspects of group membership protocols. We are also thankful to Wei-Ju Chen, the Teaching Assistant for this course who was ready to spend time discussing various scenarios that might deviate from the usual handling of group membership protocols.

References

[1] F. Cristian, "Reaching agreement on processor group membership in synchronous distributed systems," *Distributed Computing*, vol. 4, pp. 175–187, 1991.

[2] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, p. 265–286, Feb 1991.

[3] B. D. Flaviu Cristian and J. Dehn, "Fault-tolerance in air traffic control systems," *ACM Transactions on Computer Systems*, vol. 14, p. 265–286, Aug 1996.

[4] F. Cristian, "Synchronous atomic broadcast for redundant broadcast channels," *The Journal of Real-Time Systems*, vol. 2, pp. 195–212, 1990.

7 Selected Code Snippets

```
void Process::send_atomic_broadcast_p1(Message m) {
    // Filters out processes that have failed
   if (!active) { return; }

   // (We have written a threadsafe print function that calls cout)
   print(to_string(process_id) + " sends atomic broadcast : " + m.get_content_str())
        ;
   atomic_messages_sent++;

   if (m.get_content() == NEW_GROUP) {

      other_members.clear();

      // Attach the newly created group id to the message
      Group* group = new Group(this, m.get_time_stamp() + BIG_DELTA);
      m.set_group_id(group->get_id());

      leave_group();

      group->add_member(this);
      this->group_id = group->get_id();
   }
```

```
    int list_size = process_list.size();

     // setting up the army of threads to be sent out soon
    pthread_t thread_ids[list_size];
    ThreadArgs** thread_args_array = (ThreadArgs**) malloc(list_size * sizeof(
        ThreadArgs*));

    for (int i = 0; i < list_size; i++) {
       // Don't send it to yourself
       if (process_list[i]->get_process_id() == process_id) {
          continue;
       }

       thread_args_array[i] = (ThreadArgs*) malloc(sizeof(ThreadArgs));

       thread_args_array[i]->receiver = process_list[i];
       thread_args_array[i]->sender = this;
       thread_args_array[i]->message = m;
    }

    for (int i = 0; i < list_size; i++) {
       if (process_list[i]->get_process_id() == process_id) {
          continue;
       }

         // atomic_broadcast_thread_helper is a static helper function that
         // is used to allow a thread to access the non-static function
       pthread_create(&thread_ids[i], NULL, atomic_broadcast_thread_helper, (void*)
           thread_args_array[i]);
    }

    for (int i = 0; i < list_size; i++) {
       if (process_list[i]->get_process_id() == process_id) {
          continue;
       }

       pthread_join(thread_ids[i], NULL);
       free(thread_args_array[i]);
    }

    free(thread_args_array);
 }

 void Process::receive_atomic_broadcast_p1(Process* sender, Message m) {
     // Filters out processes that have failed
    if (!active) { return; }

    print(to_string(process_id) + " received atomic broadcast from " +
       to_string(sender->get_process_id()) + " : " + m.get_content_str());

    switch (m.get_content()) {
       case NEW_GROUP: {
           // clear your current members and leave your group
          other_members.clear();
          other_members.push_back(sender->get_process_id());
          leave_group();

          Group* new_group = group_id_table[m.get_group_id()];
          new_group->add_member(this);
          group_id = m.get_group_id();

          send_atomic_broadcast_p1(Message(PRESENT_ADD, group_id));

          break;
       }
```

```
      case PRESENT_ADD: {
         // Make sure this message applies to your group
         if (m.get_group_id() != group_id) { break; }

         other_members.push_back(sender->get_process_id());

         break;
      }
      case PRESENT_CHECK: {
         int sender_id = sender->get_process_id();
         int sender_idx = find(other_members.begin(), other_members.end(), sender_id)
              - other_members.begin();

         checked[sender_idx] = true;

         break;
      }
      default: {
         print("Unknown message.");
      }
   }
}
```

The complete code is available on Github.