Parallel and Distributed Computing
DD2443 - Pardis24
Exercises for Lecture 5

Name: Casper Kristiansson

September 12, 2024

Exercise 1

Question

Prove Lemma 5.1.5, that is, that every n-thread consensus protocol has a bivalent initial state.

Lemma 5.1.5. Every n-thread consensus protocol has a bivalent initial state.

A protocol state is *critical* if:

- **it is bivalent, and**
- **if any thread moves, the protocol state becomes univalent.**

Answer

We have that a state is univalent if the outcome from the consensus protocol that all executions of it will result in the same outcome. A state can be called bivalent where future executions cannot be exactly determined. Lastly, a critical state is a bivalent state where if a thread executes one step its execution reaches a univalent state (determined state).

The goal is to prove that every n-thread consensus protocol has a bivalent initial state meaning that there is an undecided configuration where the current outcome cannot be predetermined.

From the basis of the protocol, we know that the starting state could either lead to a 0-valent state or a 1-valent state. From the basics, we know that the starting state will be either decided as 1 or 0 or a state between these which can be seen as undecided. The undecided state is the bivalent state. This state is determined based on what the threads are doing and deciding if they are going down path 1 or 0. From this, we know that all states can't be univalent because the overall protocol won't even work. But we also know that the state needs to have both 0-valent and 1-valent states meaning that the outcome can't be pre-determined and therefore we understand that the proctol has a bivalent state.

Exercise 2

Question

Consider a distributed system where threads communicate by message passing. A type A broadcast guarantees:

- **every nonfaulty thread eventually gets each message,**
- **if P broadcasts M1 and then M2, then every thread receives M1 before M2, but**
- **messages broadcast by different threads may be received in different orders at different threads.**

A type B broadcast guarantees:

- **every nonfaulty thread eventually gets each message,**
- **if P broadcasts M1 and Q broadcasts M2, then every thread receives M1 and M2 in the same order.**

For each kind of broadcast,

- **give a consensus protocol if possible;**
- **otherwise, sketch an impossibility proof.**

Answer

Consensus Protocol for Type A Broadcast

The proctol for type A messages from the same thread are received in order but messages from other threads can arrive in different orders. This means that there will be inconsistency because we can't guarantee that all other threads know the sequence of events. This means that consensus is impossible.

Consensus Protocol for Type B Broadcast

For Type B every thread receives the message in the same order. This means that there will be consistency across all threads which makes consensus achievable in this scenario. This means that a protocol can be designed so that the order of the messages that come in decides the consensus value. Because they all are consistency they can decide which value to use. Therefore consensus is possible.

Exercise 3

Question

Exercise 5.22. Fig. 5.18 shows a FIFO queue implemented with read(), write(), getAndSet() (that is, swap), and getAndIncrement() methods. You may assume this queue is linearizable, and wait-free as long as deq() is never applied to an empty queue. Consider the following sequence of statements:

- **Both getAndSet() and getAndIncrement() methods have consensus number 2.**
- **We can add a peek() simply by taking a snapshot of the queue (using the methods studied earlier) and returning the item at the head of the queue.**
- **Using the protocol devised for Exercise 5.8, we can use the resulting queue to solve n-consensus for any n.**

We have just constructed an n-thread consensus protocol using only objects with consensus number 2.

Identify the faulty step in this chain of reasoning, and explain what went wrong.

Answer

Based on the FIFO queue implementation in Fig 5.18 the faulty step lies in that the getAndSet() and getAndIncrement() have consensus numbers of 2 where adding the extra method peek() does not increase the queue consensus value. The peek() method does allow us to see the top item the getAndSet() and getAndIncrement() methods limit the system to 2 consensus (2 threads).

Based on this we can understand that the consensus number in any system is always determined by the strongest synchronization point. For the example with peek() and the increase to more than 2 threads, we can't guarantee that all threads are aligned with each other (agreement).