Lecture Notes for Concurrent Algorithms

Lecture of November 9, 2009

Implementing the Consensus object with Timing Assumptions

1 Introduction

So far in the course, we have kept ourselves confined to completely asynchronous systems. These systems provided us with Single Reader Single Writer **(SRSW)** safe binary registers and we were able to construct Multi Reader Multi Writer **(MRMW)** M-valued atomic registers using them. We also showed how we can construct more objects such as a counter or a snapshot object. However, we also faced restrictions on what was achievable (FLP). We also showed that if one had a consensus object, which was impossible to make using only registers, then we could use it as a Universal constructor to construct any other shared object. However, we would need support from hardware to make such an object.

In this lecture we investigate the question whether we can construct such a consensus object exploiting knowledge about the nature of synchronicity (relative speed of processes) the system can minimally guarantee. This indeed is the case in most situations, where such a guarantee may be implied by the nature of scheduling (round robin) or may be provided with probability tending to one with time.

Finally, we consider the question of what is the weakest assumption which still allows us to construct the consensus object.

2 Modular approach

The construction is done by assuming presence of other structures and showing that provided the smaller structures, we can arrive at the final consensus object. Wait-free consensus is constructed using:

- Registers (Atomic), and,
- Lock-free Consensus (**L-Consensus**) which is in turn constructed with:
  - Obstruction-free Consensus (**O-Consensus**)
  - **Leader** abstraction which utilizes certain timing assumptions

The approach is illustrated in figure 1.

2.1 Requirements for Consensus

A consensus object need to meet the following requirements:

1. **Wait-free-termination:** If a correct process proposes, then it eventually decides.

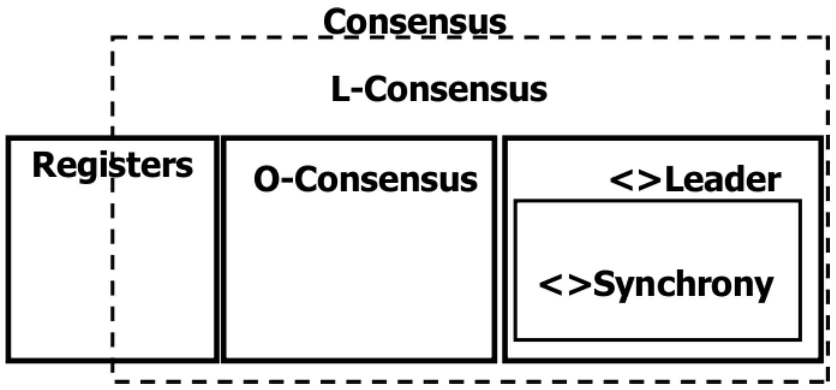

Figure 1: A modular approach

2. **Agreement:** No two processes decide differently.
3. **Validity:** The agreed value is a proposed value.

2.2 Requirements for L-Consensus

For lock-free consensus, we retain the conditions 2 and 3 and weaken condition 1 to the following:

1. **Lock-free-termination:** If a correct process proposes, then *at least one* correct process eventually decides.

This does not ensure that *all* the processes will decide, but instead, *at least* one process will. Hence, it guarantees that the collection of processes collectively proceed forward, without ensuring that all of the processes make progress. To add this capability, we would require a MRMW atomic shared register (see figure 1).

2.3 Requirements for O-Consensus

For the Obstruction free consensus, we further weaken condition 1 to the following:

1. **Obstruction-free-termination:** If a correct process proposes and *eventually executes alone*, then the process eventually decides.

Hence, if a process is allowed to execute alone, with no other process competing with it (reading/writing to the same registers) then the process will be able to eventually arrive at a consensus. It is interesting to note that while FLP proves that an algorithm which implements wait-free consensus is impossible, obstruction free consensus is possible to implement even on a completely asynchronous system.

3 Implementing O-Consensus

The idea behind the implementation is that several processes may keep trying to concurrently decide, until some time (unknown): and agreement and validity must not be violated in this preliminary period.

The processes will keep track of activities of other processes and if it finds that no process is competing to decide a consensus value, it chooses the last suggested value and marks it as available for other processes.

The algorithm requires the following data structures:

- A local time stamp ts for process p_i initialized to i. It will be incremented by N (the number of processes) each time to ensure that each process maintains a unique timestamp, different from all other processes.
- An array of shared registers is maintained **Reg[1 .. N]**. Each register contains a pair of values:
  - **Reg[i].T** contains a timestamp (initialized to 0)
  - **Reg[i].V** contains a pair (proposed value, timestamp). Each pair is initialized to $(\perp, 0)$.

Notice that the array of registers $Reg[1..N]$ contains two distinct copies of timestamps. $Reg[i].T$ will contain the value of the timestamp which the process is at, while $Reg[i].V.ts$ will contain the timestamp when $Reg[i].V.value$ was suggested by process i.

To further simplify the presentation, the following two functions are defined:

- **highestTsp()** : returns the highest time stamp among all elements $Reg[1..N].T$. Defined in 2
- **highestTspValue()** : returns the value $Reg[idx].V.value$ corresponding the the highest timestamps $Reg[1..N].V.ts$. Defined in 1.

```
Output: The highest time stamp
$maxTS := -1$
$maxV := 0$
for $i \leftarrow 1$ to $N$ do
    if $max < Reg[i].V.ts$ then
        $maxTS := Reg[i].V.ts$
        $maxV := Reg[i].V.value$
return $maxV$
```

Function `highestTstValue()`

It is important to note that while **highestTsp()** works on the timestamps **Reg[1 .. N].T**, the function **highestTspValue()** works on the timestamps **Reg[1 .. N].V.ts**. Also, these functions need not be atomic.

```
Output: The highest time stamp
$max := -1$
for $i \leftarrow 1$ to $N$ do
    if $max < Reg[i].T$ then $max := Reg[i].T$
return $max$
```

Function `highestTsp()`

3.1 O-consensus Algorithm

```
Input  : $v$, the value proposed by process $i$
Output: The decided consensus value
while (true) do
1       $Reg[i].T.write(ts)$
2       $val := highestTspValue()$
3       if $val = \perp$ then $val := v$
4       $Reg[i].V.write(val, ts)$
5       if $ts = highestTsp()$ then return $(val)$
6       $ts := ts + N$
```

Function *propose*(*i*,*v*)

In words:

- p_i announces its timestamp in step 1
- p_i selects the value with the highest time stamp in step 2
- p_i announces the value with its (p_i's) new timestamp in step 4
- If p_i's time stamp is the highest, then p_i decides in step 5

It is easy to see that this algorithm can continue to loop indefinitely even if just two processes move in lock steps. And example execution would be if process p_1 is at step 5 and p_2 executes step 1, making p_1 to loop instead of returning. When p_2 reaches step 5, p_1 executes step 1, making p_2 loop. And this process can repeat potentially forever, which will not allow wait free termination.

3.2 Proving correctness

It is easy to see that this simple algorithm implements Obstruction free consensus. To see this, consider a process executing alone. If it does not have the highest timestamp (in step 5), then it continues to loop until its own timestamp becomes the highest and agrees on the value last proposed by it on step 4. However, we need to ensure that in presence of contention, it still preserves the other two properties.

If a process i returns value val, then all processes that execute step 2 after process i returns will find the same proposed value and will eventually return it. The only case *agreement* might be violated is when a process is executing the interim step 4, and changing the val with highest timestamp while process i is on step 5. However, if a process comes to the *then* clause of step 5 successfully, it means that no competing process has executed step 1, which must be executed before step 4. Thus, if another process introduces a *newer* value while this process is on step 5, this process will not return a value at all. Hence, if a value is decided, it will be the same for all processes, satisfying property 2 required for consensus.

Step 3 insures that the first process chooses a proposed value (its own). Also, all values written to $Reg[i].V.value$ are proposed values. Hence, property 3 required for consensus is always satisfied.

Hence, this object satisfies all requirements for a O-Consensus object.

4 Implementing L-Consensus

Consider an object **Leader** which provides us with a function *leader()* with the following properties:

- Does not take any input arguments (uses only local/shared variables)
- Returns a Boolean as output : the process receiving a *true* considers itself a *leader*.
- If a correct process invokes leader, then the invocation returns and *eventually*, some correct process is *permanently* the only leader.

Hence, for an unknown time, more than one process may be returned *true*, but eventually only one process will be returned *true*. Hence, we can condition the execution of processes based on the Boolean returned by the *leader()* function, thereby making sure that eventually, there is only one process which is executing alone. Under these conditions, we can ensure the conditions necessary for Obstruction free consensus.

Implementing this idea, we can implement L-consensus as follows:

4.1 Proving correctness

Properties 2 and 3 hold naturally as the algorithm is the same as that for L-consensus.

Also, since the property of the *leader()* functions ensure that eventually a correct process gets returned true permanently, we have eventually exactly one process which is running the loop and making progress. Hence, the condition for lock-free termination are met, and *at least* one process eventually decides a value.

```
Input  : $v$, the value proposed by process $i$
Output: The decided consensus value
while $true$ do
    if $leader()$ then
1       $Reg[i].T.write(ts)$
2       $val := highestTspValue()$
3       if $val = \bot$ then $val := v$
4       $Reg[i].V.write(val, ts)$
5       if $ts = highestTsp()$ then $return\ (val)$
6       $ts := ts + N$
```

Function $propose_L$ (i,v)

5 Implementing W-Consensus

Actually, exactly one process gets to continue for lock-free consensus, since all other processes remain in the infinite **while** loop. However, this situation can be remedied by making the process which finally reaches step 4 announce to all other processes the value which has been decided. This can be easily achieved by using a shared register **Dec**. The implementation of Consensus would then be the following:

```
Input  : $v$, the value proposed by process $i$
Output: The decided consensus value
while Dec.read() $= \bot$ do
    if $leader()$ then
1       $Reg[i].T.write(ts)$
2       $val := highestTspValue()$
3       if $val = \bot$ then $val := v$
4       $Reg[i].V.write(val, ts)$
5       if $ts = highestTsp()$ then Dec.write(val)
6       $ts := ts + N$
return Dec.read()
```

Function $propose_W$ (i,v)

This ensures that eventually all correct processes terminate, satisfying the final property 1 required for wait-free consensus.

6 Leader object

Hence, if we are able to finally implement the **Leader** object, we will be able to attain wait-free consensus. However, we would require certain assumptions about the synchronicity of the system.

6.1 The Assumption

There is a time after which there is a lower and an upper bound on the delay for a process to execute a local action, a read or a write in shared memory.

On the face of it, it does not seem like a very strong assumption at all, stating only that no process step would take an infinite time to complete and most systems can guarantee the same. However, this assumption is enough for implementing the **Leader** object.

The property ensures that each correct process will eventually take a step, and there is an upper bound on the time it takes. The *leader()* function will aim at choosing the correct process with the lowest id (i). Also, assuming that at least one process keeps making progress, if the *leader()* function notices that no process has made any progress, then it knows that either the processes have crashed, or else, the upper bound is larger than the bound chosen, and it can improve the estimate for the upper bound and check again.

However, as one of the processes themselves would invoke the *leader()* function, we are assured of correctness of this process and if all processes with id less than it are considered crashed, then it should select itself to be the leader. However, if it was the leader and one of the processes with a smaller id has made progress, then it would mean that the estimate for the upper bound of time was not good enough. Hence, the *delay* before the next check is incremented and the other process is elected the leader.

The algorithm is defined in 6 and *elect()* is defined as shown in 7.

```
$currentLeader$ initialized to self
$check$ and $delay$ initialized to 1
$clock$, $last[j]$ and $Reg[j]$ initialized to 0
Input  : $i$, the id of the process
Output: Whether process $i$ is the leader
return ($currentLeader == self$)
Task:
while true do
    if $leader = self$ then $Reg[i].read() + 1$
    $clock := clock + 1$
    if $clock = check$ then elect()
```

Function $leader_i$()

Hence, we see that given such a weak guarantee of an unknown upper bound, we can write a function which will eventually find a good enough estimate of it and help us in choosing a single leader among a group of processes. With this, we conclude our construction of the consensus object.

```
Input : $i$, the id of the process
$noLeader := true$
for $j \leftarrow 1$ to $i-1$ do
    if $Reg[j].read() > last[j]$ then
        $last[j] := Reg[j].read()$
        if $leader \neq p_j$ then $delay := delay \times 2$
        $leader := p_j$
        $noLeader := false$
        break(for)
$check := check + delay$
if $noLeader$ then $leader := self$
```

Function $elect_i$()

7 Looking back on the minimal assumptions

This was the first assumption which asynchronous systems seemed to guarantee, this result was found independently by many groups. Ever since, the search for weaker assumptions has been progressing forward, and much work as been done in attempting to define what one means by an assumption being *weaker* than another.

The set of results we have are:

- Consensus is impossible in an asynchronous system with registers.
- Consensus is possible in an eventually synchronous system with registers

However, there are some questions which are yet to be answered:

- What is the minimal synchrony assumption needed to implement Consensus with registers?
- Is there a weaker timing abstraction then **Leader** that helps registers solve consensus?

The research is ongoing in these areas and has provided many insights in the working of concurrent systems. A direct result of these are *Failure detectors*. However, discussing them is out of the scope of this lecture.