Distributed Algorithms **M.Tech., CSE, 2016**

Lecture 3: Asynchronous Shared memory.

Faculty: K.R. Chowdhary *: Professor of CS*

3.1 Introduction

Distributed computing theory develops computability and complexity theories for models whose computation involves many processors interacting in certain limited unpredictable manner through some communication objects, where a processor is shorthand for a sequential piece of code that includes instructions, some of which involve access to the communication objects.

Distributed computing has focused on a number of models distinguished by the different communication objects and different timing constraints. Among those investigated most extensively are the *asynchronous message-passing model*, the *asynchronous shared-memory model*, *asynchronous models.*

3.2 Models

An *asynchronous distributed computation model* is the set of all sequential interleaving of communication actions performed by sequential processes or processors on shared communication objects. A communication action may be thought of as the invocation of a remote procedure call by a processor at an object. An object may be thought of as a processor that executes the remote procedure call, changes its state, and responds to the invoking processor by returning a value. The returned value causes the invoking processor to change its state, which in turn determines the next parameter for the next access to a communication object. We assume that processors never halt, and therefore after each return of an invocation, they enter a state in which a new invocation is enabled.

In the *network model*, processors access unidirectional point-to-point communication channels. A single communication object is associated with two processors, called sender and receiver, respectively. The sender can invoke an action *send*(m) on the object. The effect of the action is to place a message m in the buffer of the object. After placing the message in its buffer the object responds by returning an *ok* to the sender. The receiver invokes an action receive which moves a message from the buffer of the object to the receiver, if such a message exists, or the object responds by notifying exception otherwise. In the shared-memory model, communication objects are *read/write registers* on which the action of read and write can be invoked.

Here we assume that communication objects do not fail. Yet, in light of the view of a communication object as a "restricted" processor, it is not surprising that when communication failures are taken into account, they give rise to results reminiscent of processor failures.

Given a **protocol** - the instantiation of processors with codes-and the initial conditions of processors and objects, we define a space R of *runs* to be a subset of the infinite sequences of processors names. Since we assume that a processor has a single enabled invocation at a time, such a sequence when interpreted as the order in which enabled invocations were executed completely determines the evolution of the computation.

Before the system starts all runs in which the processor has its current input are possible. As the system evolves, the local state of the processor excludes some runs. Thus with a local state of a processor we associate a view - the set of all runs in R that are not excluded by the local state. By making processors maintain their history in local memory we may assume that consecutive views of a processor are monotonically nondecreasing. Thus, with each run $r \in R$ of a protocol p we can associate a limit view $lim(V_i(r,p))$ of processor P_i. A protocol f is full-information if for all i, r, and p we have $lim(V_i(r,f)) \subseteq lim(Vi(r,p))$. Intuitively, a full-information protocol does not economize on the size of its local state, or the size of the parameter to its object invocation. Models which are oblivious, that is, the sequence of communication objects a processor will access is the same for all protocols, possess a full-information protocol.

One can define the notion of full-information protocol with respect to a specific protocol in a nonoblivious model, but we will not need this notion here. A sequence of runs r_1, $r2, \ldots$ converges to a run r, if r_k and r share a longer and longer prefix as k increases.

It can be observed from the definition of a view, that two views of the same processor, are either disjoint, or related by containment. Given an intermediate view $V_i(r)$ of a processor P_i in run r, we say that a processor outputs its view in r if for all P_j which have infinitely many distinct views in r, $lim(V_j(r)) \subseteq V_i(r)$. Processor P_i is faulty in r if it outputs finitely many views. Processor P_i is participating in r if it outputs any nontrivial view. Otherwise it is sleeping in r. A model A with n processors with communication object O_A wait-free emulates a model B with n processors and communication objects O_B if there is a map m from runs R_A in A to runs R_B in B such that,

1. The sets of sleeping processors and faulty processors in r and $m(r)$ are identical.
2. The map m is continuous with respect to prefixes. That is, if r_1 , r_2, $\ldots$ in A converges to r, then $m(r_1), m(r_2), \ldots$ in B converges to $m(r)$. This captures the idea that the map does not predict the future.
3. The map m does not utilize detailed information about the past of a run, if this detailed information is not available through processors views. Formally, for all P_j nonfaulty and for all r in A, $m(lim(V_j(r))) \subseteq lim(V_j(m(r)))$.

3.3 Asynchronous Model (Two-Processor Shared-Memory)

Consider a two-processor single-writer/multireader (SWMR) shared-memory system. In such a system, there are two processors P_1 and P_0, and two shared-memory cells C_1 and C_0. Processor P_i writes exclusively to C_i, but it can read the other cell. Both shared-memory cells are initialized to null $\perp$. Computation proceeds with each processor alternately writing to its cell and reading the cell of the other processor.

Can this two-processor system 1-resiliently (wait-free in this case, since for n = 2, n - 1 = 1) elect one of the processors as the leader? No one-step full-information protocol, and consequently no one-step protocol at all, for solving this problem exists. Consider the state of processor P_1 after writing and reading. It could have read what processor P_0 wrote (denoted by $P_1 : w_0$), or it could have missed what processor P_0 wrote (denoted by $P_1 : \perp$). Thus, we have four possible views, two for each processor, after one step.

In the graph whose nodes are these views, two views are connected by an undirected edge if there is an execution that gives rise to the two views. The resulting graph appears in figure 3.1. Since a processor has a single view in an execution, edges connect nodes labeled by distinct processor IDs. The two nodes of distinct IDs which do not share an edge are $P_1 : \perp$ and $P_0 : \perp$. This follows from the fact that in shared memory in which processors first write and then read, the processor that writes second must read the value of the processor that writes first.

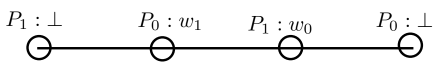

Figure 3.1: One-step view graph.

The edge $\{P_1 : \perp, P_0 : w_1\}$ corresponds to the execution: P_1 writes, P_1 reads, P_0 writes, P_0 reads. If we could map the view of a processor after one step into an output, then processor P_1 in this edge is bound to elect P_1, since the possibility of a solo execution by the processor has not yet been eliminated. Similarly, in the edge $\{P_0 : \perp, P_1 : w_0\}$, processor P_0 is bound to elect P_0. Thus, no matter what processor is elected by P_1 and P_0 in the views $P_1 : w_0$ and $P_0 : w_1$, respectively, we are bound to create an edge where at one end P_1 is elected and at the other end P_0 is elected. Because there is an execution in which both processors are elected, we must conclude that there is no one-step 1-resilient protocol for election with two processors.

References

[1] Allen B. Tucker, Jr., "The computer Science and Engineering Handbook," *CRC Press*, 1997.