1 Introduction

The objective of the next section of the course is to discuss how quorum systems play a role in distributed storage (what used to be called distributed registers). Before we can start talking about quorum systems, we need to set a model of distributed storage. That is the goal of today’s lecture.

2 Timing assumptions in the model

Communication between processors is either synchronous or asynchronous. But we assume (1) internal steps by processors take no time, and (2) there is a global clock. (Although, in asynchronous systems, processors are not aware of the global clock.) This allows our analysis storage algorithms by focusing on the amount of latency or efficiency of inter-processor communication. We now consider the two different communication timing assumptions in turn.

Synchronous communication

In our synchronous model, any message sent from processor $p$ to processor $q$ will arrive at $q$ in at most $\delta$ time steps. Here, $\delta$ is a global constant, and all processors are aware of the value of $\delta$, so a processor can detect if it will not receive a message from a potential sender in the current round of communication. There are two main types of synchronous communication models, Message Passing and Shared Memory.

**Message Passing** A message sent from $p$ to $q$ at time $t$ is received by $q$ at time $\leq t + \delta$.

**Shared Memory** An invocation at time $t$ gets a matching response no later than $t + \delta$.

Asynchronous communication

In our asynchronous model, there are no bounds on message delay, or on response time of shared objects, except a reliability assumption that the message cannot be lost. Every message sent from $p$ to $q$ is eventually received by $q$ in finite time; or, every invocation on a shared object eventually gets a matching response (again, in finite time).
Partial synchrony

We can also consider a model in which there are synchronous periods within an asynchronous system. For example, communication may be synchronous within the time interval \([t, t']\) if there is a \(\delta\) such that any message sent at time \(s\) is received by time \(s + \delta\), as long as \(s \in [t, t']\). Another option is to consider communication that is eventually synchronous: the system may behave asynchronously for some finite initial period of time, but after some time \(t\), the system will behave synchronously.

2.0.1 Time complexity

The worst-case time complexity of an asynchronous algorithm is the worst case number of time units over all executions, from start to finish, assuming that message delay is at most one unit.

3 Model for distributed storage

Definition 1 A distributed READ/WRITE storage (formerly called read/write registers) is a shared object where:

1. the object has a set of allowable values \(D\), and an initial value \(\perp \notin D\).

2. the object has two operations: (a) \(\text{write}(v)\) where \(v \in D\); and (b) \(\text{read}()\). The operation \(\text{write}(v)\) returns an ack, and the operation \(\text{read}()\) returns a value \(v \in D\) (sometimes written \(\text{return}(v)\)).

We focus on wait-free implementation. This is a liveness property, that is to say, it ensures something good will eventually happen. A storage algorithm is wait-free if in every execution, every read/write operation invocation has a matching response. This may sound like a trivial condition, and indeed it is, if the system has no faulty processors. Our goal is to ensure wait-free implementation of storage even in a system where several processors can fail. Intuitively, a correct processor will always be able to proceed with its algorithm, even if some of its neighbors are faulty; the correct processor does not have to wait for them.

As an aside, recall that a safety property is one that ensures that bad things will never happen. (For example, no two processors will ever enter the critical section at the same time. That’s the main safety property of a mutual exclusion algorithm.) On the other hand, an example of a liveness property is fairness: it’s possible to define many different kinds of fairness, but an example of fairness in a mutual exclusion algorithm is that every processor that requests to enter the critical section will eventually be able to do so. (That is to say, there exists no possible execution where a processor might be waiting at the Entry stage forever.)

We’re interested in proving a liveness property about distributed storage algorithms. This will require arguing about infinite executions, and ensuring the appropriate responses to operations always occur.
Definition 2 A storage algorithm execution $ex$ is well-formed if no benign process $c$ invokes a new operation in $ex$ while earlier operations invoked by $c$ are pending.

Informally, if $ex$ is well-formed, then no processor $p$ will issue multiple concurrent reads and writes, but rather, will wait for each operation to terminate before starting the next operation. So the only concurrency in the system arises from different processors invoking operations whose time intervals overlap.

We now define what it means to order the operations in an execution into a well-formed history.

Definition 3 Suppose $op_1$ and $op_2$ are operations in execution $ex$. We say $op_1$ precedes $op_2$ if the response of $op_1$ happens before the invocation of $op_2$. We denote this $op_1 \xrightarrow{ex} op_2$. If neither $op_1 \xrightarrow{ex} op_2$ nor $op_2 \xrightarrow{ex} op_1$, then $op_1$ and $op_2$ are concurrent.

Definition 4 A history of a (possibly partial) execution is a sequence of invocations and responses in the same order as they appear in the (partial) execution.

Definition 5 History $H_1$ completes history $H_2$ if $H_1$ is obtained from $H_2$ by taking each pending $op$ in $H_2$ and either removing the invocation, or appending any valid matching response to the end of $H_2$.

4 Storage consistency

Lamport defined three notions of storage consistency: safe, regular and atomic storage. We will, later on, be interested in how to implement atomic storage using quorum systems. First, let’s define these notions.

Definition 6 Given an execution $ex$, and a read $r$ in $ex$, write $w$ is the last preceding write of $r$ if both of the following hold.

1. $w \xrightarrow{ex} r$.

2. For all writes $w' \neq w$, if $w' \xrightarrow{ex} r$ then $w \xrightarrow{ex} w'$.

Note that this definition is not well defined if there are overlapping writes performed by multiple processors, as that would allow us to trace a chain of earlier and earlier writes, in which even the write that was invoked first is not the last preceding write. So this definition is well defined in the context of single-writer systems: many processors may read from shared memory, but only one processor is allowed to write to locations.

We now present the weakest notion of storage consistency, that of safety, which intuitively requires that a read to a storage when the context is unambiguous (i.e., there is no write overlapping the same time interval) returns what we would want it to return, the last value written to it. And, on the other hand, if a read and write overlap, the system cannot blow up completely: the read cannot return total garbage, but is allowed to return any of the values that can be legally stored in the storage location.
Definition 7 A (partial) execution is safe if every read that doesn’t overlap a write returns the value of the last preceding write. A read concurrent with writes may return any value in $D$.

A somewhat stricter condition on the storage is to limit the range of values that can be returned in the event of overlapping reads and writes. If we require the read to return either a value that is being concurrently written, or the last value written, that is called regular storage.

Definition 8 A (partial) execution is regular if every read returns the value of one of the overlapping writes, or the last preceding write.

We will define atomic storage next time.