1 Introduction

This lecture introduces a model of Distributed Shared Memory. We will use an example of an important and general technique: constructing one model given another one. It is similar to the previous topic, in which we constructed registers of one type (i.e., a model) given registers of a “weaker” type (i.e., a different model). In this case:

- $M_1$ is a model of read/write shared memory (Distributed Shared Memory)
- $M_2$ is a model of a computer network

The motivation is that real world systems are networks, but algorithms that run on distributed shared memory are often much easier to reason about. So it can be worth paying the extra complexity cost to simulate distributed shared memory ($M_1$) over a network ($M_2$), and to run algorithms on that simulation. See Figure 1 for a diagram of this approach.

2 Sequential specifications

We will build a shared memory system that implements read/write atomic objects. Every shared memory object has a sequential specification, which describes what operations can be permitted to happen in which order. For example, WRITE(5) READ(4) does not satisfy the sequential specification, because a read must return the value of the last preceding write. In the abstract, a sequential specification can just be a set of sequences of legal operations – the set of all desired behaviors of the object in the absence of concurrency. (We aren’t worried about what might happen if operations overlap.) So it’s possible to define complex objects – or weird ones – with this same structure.

First, recall that an execution $\alpha$ is a sequence of invocations and responses. In our recent proofs, we moved from $\alpha$ to a permutation $\pi$ which is a sequence of operations obtained from the invocations and responses in $\alpha$. The sequential specification could potentially apply to $\pi$. Note that safe, regular and atomic registers all have the same sequential specification, because their behavior varies only when operations overlap, and a sequential specification only describes nonoverlapping behavior.

Formally, each operation is a pair: (invocation, response). Let $i$ be a processor, and $X$ be a
Figure 1: Building a Memory Consistency System over a network. The network is what physically exists. Each external user talks to a different processor, and each processor maintains a copy of the memory consistency system.
read/write shared object. For operations READ and WRITE, the following table holds.

<table>
<thead>
<tr>
<th>inv</th>
<th>resp</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ\textsubscript{i}(X,v)</td>
<td>read\textsubscript{i}(X)</td>
</tr>
<tr>
<td>WRITE\textsubscript{i}(X,v)</td>
<td>write\textsubscript{i}(X,v)</td>
</tr>
</tbody>
</table>

A sequence of operations is legal if each READ returns the value of the most recent preceding WRITE. If there is no preceding WRITE, the READ returns the initial value of the object. Example: suppose at initialization, $X = 0$. Then $\text{READ}_i(X,0)$ $\text{WRITE}_j(X,1)$ $\text{WRITE}_i(X,2)$ $\text{READ}_j(X,2)$ is a legal operation sequence of object $X$, so it will be part of the sequential specification of $X$.

## 3 Correct interaction and linearizability

### 3.1 Correct interaction

Let $\sigma$ be a sequence of invocations and responses. With the notation $\sigma \upharpoonright i$ we mean: the subsequence of $\sigma$ consisting of all of $p_i$’s invocations and responses. We define correct interaction as: For each $p_i$, $\sigma \upharpoonright i$ consists of alternating invocations and responses (matching the preceding invocation), beginning with an invocation.

So, for example, a user cannot send out two READs without receiving a return in between. This is a simplification, and we could allow more parallelism in the model, but we won’t. Note that this is a safety property. We also assume a related liveness property about $\sigma$, which is: every invocation has a matching response.

## 4 Consistency conditions

Now we look at two conditions of memory consistency. The first, linearizablity, is a strong condition that more or less enforces that the shared objects behave like “classical” objects, with no issues arising from concurrency. The second, sequential consistency, is a relaxation of linearizability. It still provides nice properties of linearizability, but is much less expensive to implement.

### 4.1 Linearizability

One of the nicest niceness properties of an execution is linearizability - informally, the notion that the execution behaves as though it is a completely sequential execution. Formally, we define linearizability as follows.

**Definition 1** Let $\sigma$ be a sequence of invocations and responses. We say $\sigma$ is linearizable if there exists a permutation $\pi$ of all operations in $\sigma$ such that
1. for each object $X$, $\pi|X$ is legal

2. if the response of $op_1$ precedes the invocation of $op_2$ in $\sigma$, then $op_1$ precedes $op_2$ in $\pi$.

For example, suppose there are two processors, 0 and 1; and two read/write shared objects $X$ and $Y$. Then the following sequence $\sigma_1$ is linearizable.

\[
\sigma_1 = \text{write}_0(X, 1) \ \text{write}_1(Y, 1) \ \text{ack}_0(X) \ \text{ack}_1(Y) \ \text{read}_0(Y) \ \text{read}_1(X) \ \text{return}_0(Y, 1) \ \text{return}_1(X, 1)
\]

We can see this by setting

\[
w_0 = \text{WRITE}_0(X, 1) \\
w_1 = \text{WRITE}_1(Y, 1) \\
r_0 = \text{READ}_0(Y, 1) \\
r_1 = \text{READ}_1(X, 1)
\]

and setting $\pi = w_0w_1r_0r_1$ which satisfies linearizability.

On the other hand, if we make just one change to $\sigma_1$ (shown in bold)

\[
\sigma_2 = \text{write}_0(X, 1) \ \text{write}_1(Y, 1) \ \text{ack}_0(X) \ \text{ack}_1(Y) \ \text{read}_0(Y) \ \text{read}_1(X) \ \text{return}_0(Y, 0) \ \text{return}_1(X, 1)
\]

the sequence $\sigma_2$ is not linearizable. We can show this by setting $r_0$ to READ$_0(Y, 0)$ and making the following argument. It must be true that $w_1$ precedes $r_0$, because its response precedes the invocation of $r_0$ in $\sigma_2$. And yet, the value returned by $r_0$ is not the value written by $w_1$. Therefore, $\sigma_2$ is not legal.

Nevertheless $\sigma_2$ obeys a weaker consistency condition, sequential consistency, which is still good enough for many applications. We will consider it now.

### 4.2 Sequential consistency

We now define sequential consistency.

**Definition 2** Let $\sigma$ be a sequence of invocations and responses. We say $\sigma$ is sequentially consistent if there is a permutation $\pi$ of all operations in $\sigma$ such that

1. for each object $X$, $\sigma|X$ is legal

2. if the response of $op_1$ at $p_i$ precedes the invocation of $op_2$ at $p_i$, then $op_1$ precedes $op_2$ in $\pi$.

Condition (2) is a relaxation of the linearizability condition. We no longer require that every operation “be sequential,” but only that every operation of a particular processor “be sequential.” The nonlinearizable $\sigma_2$ from the previous subsection is sequentially consistent. Recall that
\[ \sigma_2 = \text{write}_0(X, 1) \text{write}_1(Y, 1) \text{ack}_0(X) \text{ack}_1(Y) \text{read}_0(Y) \text{read}_1(X) \text{return}_0(Y, 0) \text{return}_1(X, 1) \]

A sequentially consistent permutation of \( \sigma_2 \) is \( w_0 r_0 w_1 r_1 \). Before, we had to place \( w_1 \) before \( r_0 \), which was not legal. Now, since the operations are performed by different processors, we don’t.

Linearizability implies sequential consistency, but the reverse is not true. Both are considered strong memory consistency conditions, since all processors agree on the same view of the order of events. It’s not hard to construct sequences that are not sequentially consistent.

\[ \sigma_3 = \text{write}_0(X, 1) \text{write}_1(Y, 1) \text{ack}_0(X) \text{ack}_1(Y) \text{read}_0(Y) \text{read}_1(X) \text{return}_0(Y, 0) \text{return}_1(X, 0) \]

We can show that \( \sigma_3 \) is not sequentially consistent. Set

\[
\begin{align*}
w_0 &= \text{WRITE}_0(X, 1) \\
w_1 &= \text{WRITE}_1(Y, 1) \\
r_0 &= \text{READ}_0(Y, 0) \\
r_1 &= \text{READ}_1(X, 0)
\end{align*}
\]

We write \( op_1 \prec op_2 \) for “\( op_1 \) precedes \( op_2 \) in the sequence.” Since both \( \sigma|X \) and \( \sigma|Y \) must be legal (condition (1)), \( r_0 \prec w_1 \) and \( r_1 \prec w_0 \). Since, for each processor \( p_0 \) and \( p_1 \), if an operation concludes before another operation starts (condition (2)), the order must be preserved in \( \pi \), \( w_0 < r_0 \) and \( w_1 < r_1 \). But then \( r_0 < w_1 < r_1 < w_0 < r_0 \), so \( r_0 < r_0 \) a contradiction.

The stronger the consistency condition, the better for the programmer. The weaker the consistency condition, the better for the architect.

5 Totally Ordered Broadcast simulates Linearizability

Assume we have an algorithm that provides totally ordered broadcast, which we abbreviate \( \text{tbc} \). Such algorithms exist, for example by using encoded time stamps to ensure that processors know the order in which messages are to be processed, regardless of the time at which the messages arrive. If the network has \( \text{tbc} \) as a primitive, it’s possible to build linearizable read/write shared objects, as stated by the following theorem.

**Theorem 1** The linearizable shared memory system is simulated by a totally ordered broadcast system.

**Proof:** We prove this by providing an algorithm for read-write object \( X \). Each processor has a local copy of \( X \). In order to process a read from, or a write to, \( X \), the system queues each operation in a total broadcast order. That way, if processors \( p \) and \( q \) receive a read and a write out of order, they will nevertheless both process the operations in the same order, because of the \( \text{tbc} \) primitive. So all processors move in lockstep when updating their local copies of \( X \). See Algorithm 1 for the pseudocode.
Algorithm 1 Linearizability of $X$ given tbc

Require: $\text{copy}_i[X]$ contains the initial value of $X$ for each $p_i$

Algorithm for $p_i$
when $\text{read}_i(X)$ occurs
\hspace{1em} tbc-send$_i(\text{read}, X)$

when $\text{write}_i(X,v)$ occurs
\hspace{1em} tbc-send$_i(\text{write}, X, v)$

when tbc-recv$_i(\text{read},X)$ occurs from $p_j$
\hspace{1em} if $j = i$ then return$_i(X, \text{copy}_i[X])$

when tbc-recv$_i(\text{write},X,V)$ occurs from $p_j$
\hspace{1em} $\text{copy}_i[X] = v$
\hspace{1em} if $j = i$ then ack$_i(X)$