1 Introduction

We now turn to implementation of distributed shared memory (DSM), and the building of a DSM on a message passing system. The programmer is building on, for example, read/write objects and atomic snapshot objects. The stronger the memory consistency condition, the better for the programmer, but the worse for the architecture supporting that consistency condition.

We will see there is a complexity gap between Sequential Consistency and Linearizability. We will also see that certain programming paradigms, when combined with a weak consistency condition, give us Sequential Consistency. We will simplify the discussion by talking in terms of reads and writes, but we could speak in general of stacks, queues, etc.

We assume that invocations for each processor are sequential, and invocations/responses are of the following form:

\[
\text{READ} \rightarrow \text{RETURN}(v) \\
\text{WRITE}(v) \rightarrow \text{ACK}
\]

**Definition 1** A sequence \(\sigma\) of operations 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 \(\text{op}_1\) occurs in \(\sigma\) before the invocation of \(\text{op}_2\), then \(\text{op}_1\) occurs before \(\text{op}_2\) in \(\pi\).

**Definition 2** A sequence \(\sigma\) of operations is sequentially consistent 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 \(\text{op}_1\) at \(p_i\) occurs in \(\sigma\) before the invocation of \(\text{op}_2\) at \(p_i\), then \(\text{op}_1\) occurs before \(\text{op}_2\) in \(\pi\).
2 Algorithms for Linearizability and Sequential Consistency

A system in which messages always take the same amount of time to be delivered admits correct execution by simple algorithms. However, if we allow uncertainty in message delay, the algorithms become more complicated, to ensure consistency.

We assume we have a primitive: *Totally Ordered Broadcast*, meaning that every processor receives the messages in the same order. This can be implemented in an asynchronous system using sequence numbers. Note that if $p_1$ sends a message, and later $p_2$ sends a message, all Totally Ordered Broadcast is that either all processors will receive the message from $p_1$ first, or the message from $p_2$ first. Totally Ordered Broadcast does not necessarily respect the timeline.

**Theorem 1** The linearizable shared memory system is simulated by Totally Ordered Broadcast.

**Proof:** Consider the following algorithm. Suppose the message passing model is such that a message takes at most $d$ time to travel. Then a read takes $d$ time, and a write takes $d$ time (in the worst case).

Let $\alpha$ be an admissible execution of this algorithm, and let $\sigma$ be the sequence of invocations and responses. Define $\pi$ by ordering the operations in $\sigma$ according to the total order of their corresponding message broadcasts. This satisfies legality, because each processor receives messages for operations on $X$ in the same order. We can see $\pi$ respects the real-time ordering of operations as follows: suppose $op_i$ ends before $op_j$ starts. Then $p_i$ received the invocation of $op_i$ before receiving the invocation of $op_j$. As all processors get messages in the same order, $p_i$’s order is true of all processors. So the algorithm satisfies linearizability.

The read method in the above algorithm does need to include a send. If it does not include a send, we could construct a counterexample by having $p_i$ write a value to shared object $X$, and then ack before $p_j$ received the tbc message that $p_i$ had written to $X$. If $p_j$ reads the value of $X$ after $p_i$’s ack, but before receiving the tbc message that the value of $X$ has changed, that sequence of invocations and responses is not linearizable.
Algorithm 2 Local read algorithm for sequential consistency (Code for $p_i$)

1: when read$_i(X)$ occurs return$_i(X, \text{copy}[X])$
2: when write$_i(X, v)$ occurs tbc-send($X, v$)
3: when tbc-recv$_i(X, c)$ occurs from $p_j$
4: copy$_i[X] := v$
5: if $j == i$ then
6: ack$_i(X)$

However, Algorithm 1 is sequentially consistent if the reader returns its own value right away, without the tbc step, as one can see from Algorithm 2. The complexity of Algorithm 2 is that a read takes zero time ($R = 0$), while a write takes $d$ time ($W = d$). In general, there is a tradeoff: on one end of the spectrum $R = 0, W = d$; then in the middle $R = \alpha, W = d - \alpha$; and on the other end of the spectrum $R = d, W = 0$.

We will now prove that Algorithm 2 implements Sequential Consistency.

Lemma 2 For every $p_i$

1. $p_i$’s local copies take on all values of writes
2. all updates occur in the same order at all processors
3. this order preserves order of writes by individual processors

Proof: (1) and (2) are clear. To prove (3), we use the same argument as before: if wop$_1$ occurs before wop$_2$ in $p_i$, then by total broadcast, they occur in the same order for all processors. ■

Lemma 3 For all $p_i$, if op$_1$ precedes op$_2$ in $\sigma|i$, then the local read/write for op$_1$ at $p_i$’s copy precedes the local operation for op$_2$ at $p_i$’s copy.

Proof: We have

$$\text{inv(op}_1) \prec_{\sigma|i} \text{resp(op}_1) \prec_{\sigma|i} \text{inv(op}_2) \prec_{\sigma|i} \text{resp(op}_2)$$

and local operations must occur within their invocations and responses, so localop(op$_1$) $\prec_{\sigma|i}$ localop(op$_2$). ■

Theorem 4 Algorithm 2 implements Sequential Consistency.

Define the permutation $\pi$ of operations in $\sigma$. Order all writes in tbc order. Consider each read in invocation order in $\sigma$. Read $r$ by $p_i$ on object $X$ is placed immediately after the latest in $\pi$ of (1) the previous op by $p_i$ in $\alpha$ and (2) the write that caused the latest update in $p_i$’s copy of $X$ before local read $r$.

If $r \prec w$ is it possible that $r$ is placed after $w$? We will show that $\sigma|i = \pi|i$. 3
Lemma 5 \( \sigma|i = \pi|i \).

Proof: Case 1: \( \text{op}_1 = w_1 \) and \( \text{op}_2 = w_2 \). If \( w_1 \prec_{\sigma|i} w_2 \) then \( w_1 \prec_{\pi|i} w_2 \).

Case 2: \( \text{op}_1 = r_1 \) and \( \text{op}_2 = r_2 \). If \( r_1 \prec_{\sigma|i} r_2 \) then \( r_1 \prec_{\pi|i} r_2 \). This is so because \( r_1 \) is inserted first in \( \pi \), and then \( r_2 \) is inserted satisfying condition (1).

Case 3: \( \text{op}_1 = w_i \) and \( \text{op}_2 = r_i \). If \( w_i \prec_{\sigma|i} r_i \) then \( w_i \prec_{\pi|i} r_i \). This is so because \( r_i \) is inserted after \( w_i \) and placed after \( w_i \) in \( \pi \) by condition (1).

Case 4: \( \text{op}_1 = r_i \) and \( \text{op}_2 = w_i \). We claim that \( r_i \prec_{\sigma|i} w_i \) implies that \( r_i \prec_{\pi|i} w_i \). Suppose for contradiction there exist \( r_i, w_i \) such that \( r_i \prec_{\sigma|i} w_i \) but \( w_i \prec_{\pi|i} r_i \). \( r_i \) can only be placed after \( w_i \) because of condition (2). Choose a \( w_i, r_i \) pair with earliest placed \( r_i \) in \( \pi \). There are two cases.

Case (i): \( r_i \) is placed in \( \pi \) immediately after \( \text{op}_i \) the previous op of \( p_i \) in \( \alpha \). Now \( w_i \neq \text{op}_i \) since \( r_i \prec_{\sigma|i} w_i \). So \( w_i \prec_{\pi} \text{op}_i \prec_{\pi} r_i \). Suppose \( \text{op}_i \) is a write. Then this could not happen, as writes are always placed in the correct order. If \( \text{op}_i \) is a read, they are in the wrong order, but this could not happen because we picked the first case that was placed wrongly.

Case (ii): \( r_i \) is placed in \( \pi \) after the write \( w_j \) that caused the latest update of \( p_i \)’s copy before the read \( r_i \). In that case, \( w_i \neq w_j \) since \( r_i \prec_{\sigma|i} w_i \), and because of the order of updates and reads in \( p_i \)’s local copy, it is the case that \( w_j \prec_{\sigma|i} r_j \prec_{\sigma|i} w_i \) occurs. But the order in \( \pi \) must include \( w_j \prec_{\pi} r_i \). So \( w_i \) must be before \( w_j \). This contradicts Lemma 3, which guarantees the updates at local processors are the ordering seen in \( \pi \).

Hence, we have proved \( \sigma|i = \pi|i \) for all \( i \).