1 Introduction

We continue our discussion of 4 different models for memory systems:

- Model 1: Perfect clocks, message delay = d (exactly).
- Model 2: Perfect clocks, message delay ∈ [d – u, d].
- Model 3: Imperfect clocks, message delay = d.
- Model 4: Imperfect clocks, message delay ∈ [d – u, d].

Intuitively (versus formally), model x simulates model y if any algorithm model x can use, model y can use (perhaps with some “extra work”). We define executions, α, as a sequence of one or more operations’ invocations and responses. We define permutations as a sequence of one or more operations.

2 Properties and Proofs

Theorem 1 For any sequentially consistent memory system that provides two read/write objects, then : $t_{\text{read}} + t_{\text{write}} \geq d$.

Note: This proof holds for all models 1-4.

Proof: Let initial values of two registers $x$ and $y$ be 0. Assume for contradiction that $t_{\text{read}} + t_{\text{write}} < d$.

Let $\alpha$, be the execution where only $P_0$ takes steps:

$write_0(x, 1)ack_0(x)read_0(y)return_0(y, 0)$

where $write_0(x, 1)$ takes place after time $t = 0$ and $return_0(y, 0)$ takes place before time $t' < d$.

Let $\alpha_2$ be the execution where only $p_1$ takes steps:

$write_1(y, 1)ack_1(y)read_1(x)return_1(x, 0)$
where \( write_1(y, 1) \) takes place after time \( t = 0 \) and \( return_1(x, 0) \) takes place before time \( t' < d \). It follows that both \( \alpha_1 \) and \( \alpha_2 \) complete before time \( d \). Assume that every message takes exactly time \( d \) in \( \alpha_1 \) and \( \alpha_2 \). So no message is received before the reads return (before \( t' \)).

Consider a new execution \( \alpha_3 \) which is the interleaved combination of \( \alpha_1 \) and \( \alpha_2 \). The new execution is admissible since any messages sent by \( P_0 \) (or \( P_1 \)) are not received by \( P_1 \) (or \( P_0 \)). So, \( P_0 \) cannot distinguish between \( \alpha_1 \) and \( \alpha_3 \), and \( P_1 \) cannot distinguish between \( \alpha_2 \) and \( \alpha_3 \).

\[
t = 0 \quad \frac{[write_0(x, 1)...ack_0(x)]}{[write_1(y, 1)...ack_1(y)]} \quad \frac{[read_0(y)...return_0(y, 0)]}{[read_1(x)...return_1(x, 0)]} \quad t' < d
\]

To ensure legality it follows chronologically:

\[
READ_0(y) < WRITE_1(y, 1) \\
READ_1(x) < WRITE_0(x, 1)
\]

Notice that \( READ \) and \( WRITE \) represent the time where an operation takes place, whereas \( read \) and \( write \) represent the operation itself. Due to the processor’s internal order it follows chronologically:

\[
WRITE_0(x, 1) < READ_0(y) \\
WRITE_1(y, 1) < READ_1(x)
\]

We thus get the following contradiction:

\[
READ_0(y) < WRITE_1(y, 1) < READ_1(x) < WRITE_0(x, 1) < READ_0(y) \\
READ_0(y) < READ_0(y)
\]

We have thus shown that \( t_{\text{read}} + t_{\text{write}} \geq d \).

We can thus have any algorithm where \( t_{\text{read}} + t_{\text{write}} \geq d \), including anything in between:

- Local Write Algorithm: \( t_{\text{read}} = d, t_{\text{write}} = 0 \).
- Local Read Algorithm: \( t_{\text{read}} = 0, t_{\text{write}} = d \).

**Theorem 2** In the perfect clocks model with message delay \( = d \), there exists a linearizable MCS with either local reads or local writes.

**Proof:** 1: procedure Local Read Algorithm

2: When \( read_i(x) \) occurs:

3: \( return_i(x, \text{copy}[x]) \)

4: When \( write_i(x, v) \) occurs:

5: \( tbc - send_i(\text{write}, x, v) \)

6: \( \text{wait } d \text{ time and } ack_i(x) \)

7: If \( tbc - recv_i(\text{write}, x, v) \) occurs from \( p_j \):

\[
\text{...}
\]
copy[x] := v

end procedure

We linearize reads when they happen, and we linearize writes at \( \text{ack}_i(x) \). By construction, the order of any two reads in a permutation \( \pi \) are preserved by construction (since reads are atomic). Furthermore, the order to any two writes in a permutation \( \pi \) is also preserved since the operation \( \text{copy}[x] := v \) takes exactly \( d \) time. If a read is placed before a write, then it will call \( \text{return}_i(x, \text{copy}[x]) \) before the write changes \( \text{copy}[x] \), and thus is only affected by early writes. Finally reads after a write will only read the values of \( \text{copy}[x] \) changed by the earlier write or a write earlier, since the reads are placed after a writes \( \text{ack}_i(x) \).

1: \textbf{procedure} Local Write Algorithm

2: \quad \text{When read}_i(x) \text{ occurs:}
3: \quad \quad tbc - \text{send}_i(\text{read}, x)
4: \quad \quad \text{wait } d \text{ time and } return_i(x, \text{copy}[x])
5: \quad \quad \text{When write}_i(x, v) \text{ occurs:}
6: \quad \quad \quad \text{copy}[x] := v \text{ and } \text{ack}_i(x)
7: \quad \quad \text{If } tbc - \text{recv}_i(\text{read}, x) \text{ occurs from } p_j:
8: \quad \quad \quad \text{copy}[x] := x
9: \textbf{end procedure}

We linearize writes when they happen, and we linearize reads at their invocation. By construction the order of any two writes in a permutation \( \pi \) are preserved by construction (since writes are atomic). Furthermore, the order of any two reads is also preserved since the operation \( \text{copy}[x] := x \) takes exactly \( d \) time. If a write occurs before a read in \( \pi \), then the write happens atomically, and the read will use the value written by the write. If a read occurs before a write, \( tbc - \text{send}_i(\text{read}, x) \) will execute before the write is invoked, and the read must read from an earlier write.

\begin{theorem}
For any linearizable MCS that provides a read/write object \( x \) written by 2 processors and read by a third. Then we know \( t_{\text{write}} \geq \frac{u}{2} \).
\end{theorem}

\textbf{Proof:} \quad \text{We construct an execution } \alpha \text{ where message delays are :}

\begin{itemize}
\item \( p_1 \xrightarrow{d} p_2 \)
\item \( p_2 \xrightarrow{d-u} p_1 \)
\item \text{and all other delays are } d - \frac{u}{2}
\end{itemize}

Let \( x \) be written by \( P_1 \) and \( P_2 \) and read by \( p_0 \). Assume for contradiction that \( t_{\text{write}} < \frac{u}{2} \).

\[
\begin{array}{c|c|c}
\alpha & P_0 & P_1 \\
\hline
0 & [W(x,1)] & [R(x,2)] \\
\hline
P_0 & P_1 & P_2 \\
\hline
[\text{copy}[x]] & [W(x,2)]
\end{array}
\]
We thus have a permutation \( \pi = w_1w_2r \). Let \( \beta = shift(\alpha, < 0, \frac{u}{2}, -\frac{u}{2} >) \):

\[
\begin{array}{c|c|c|c}
\beta & P_0 & P_1 & P_2 \\
\hline
0 & \frac{u}{2} & u \\
\hline
[W(x, 2)] & [W(x, 1)] & [R(x, 2)] \\
\end{array}
\]

This changes the message delays:

- \( p_1 \xrightarrow{d-u} p_2, p_1 \xrightarrow{d-u} p_0, p_0 \xrightarrow{d-u} p_2 \)
- \( p_2 \xrightarrow{d} p_1, p_0 \xrightarrow{d} p_1, p_2 \xrightarrow{d} p_0 \)

All of these delays are within \([d - u, d]\), thus processors cannot tell the difference between executions \( \alpha \) and \( \beta \); however, the execution \( \beta \) does not have a permutation that satisfies legality. Thus we have a contradiction, and \( t_{\text{write}} \geq \frac{u}{2} \).

Intuitively, \( \frac{u}{2} \) is how much time a read needs to change since the uncertainty \( u \) can mean a change of \( \frac{u}{2} \) for writes in either direction in time.