Efficient Implementation of Vector Clocks

Group I

11th March 2016

1 Introduction

A distributed system is a system of processes communicating with each other through message exchanges. The events of one process depends not only on the results of previous computations, but also on the computations carried out by other processes which is communicated via messages to it. Therefore it is important for us to order all the events in a distributed system to know the dependencies among events of different processes. This has several uses in software applications, like in debugging a distributed system where we are given a set of breakpoints and we are required to find the cause of an exception, distributed mutual exclusion, distributed file consistency, causal broadcast etc.

Because of the distributed nature, events in a distributed system do not form a linear sequence but instead have a partial order. A partial order is defined on the set of all events, and two events are related to each other if one can potentially affect the other. The relation is anti symmetric and transitive, but not reflexive because we don't allow any such system where an event can affect itself, even before it has actually occurred. Our aim is to reconstruct the partial order relationship, given a set of observed events.

We face different challenges while trying to solve this problem. We can keep a single global clock with one process and every process can timestamp their events according to this single clock. However this requires communication links between all the processes and the time server, and it also increases the overhead because the processes are always communicating with the time server after every event. Therefore it is better to place local clocks at every process. We can use real physical clocks for this purpose, but then we face the problem of bounding the maximum drift between the different clocks. To not bother ourselves with clock synchronisation algorithms here, we describe all the approaches using logical clocks that can be implemented using simple registers. Even with logical clocks, we face the problem of how much information should we store at a process and how much should we send along with a message, so that we are able to reconstruct the partial order relationship at the end. Secondly, our approach should be flexible enough so that it can get the partial order for any subset of events. Thirdly, these events can be received in any order while constructing the relationship. These three challenges form our criteria of measuring each solution approach and are named intrusion, filtering and consistency respectively.

We have three types of solution approaches depending upon what type of dependency is tracked at the processes while sharing messages. These three dependencies are transitive dependency, direct dependency and pseudo-direct dependency. Logical clocks were first introduced by Lamport who defined the partial order relationship. The same was implemented through vector clocks by Fidge, whose approach was improved upon by Mukesh and Kshemkalyani. Vector clocks maintain transitive dependencies. Direct dependencies were maintained by Fowler and Zwaenepoel when they described causal distributed breakpoints. Later Jard and Jourdan gave a better approach, combining the best of both approaches using adaptive timestamps. The relationship maintained there was defined as pseudo-direct dependency.

The rest of the report is organised as follows - section 2 talks about the problem statement and our criteria of comparison and measurement. Section 3 discusses Lamport's logical clocks and Fidge's vector timestamp approach. It has three subsections, each discussing the three solution approaches with a working example. Section 4 compares all three approaches.

2 Problem Statement

We define a process P as a sequence of events that have a total ordering imposed on them. Events in a process can either be internal events or communication events. Internal events are local events involving some computation. Communication events can be the send event of a message or the receipt event of a message. We only define the partial order relationship on internal events. The send or receipt of a message cannot computationally affect any local operation. Any processing of information done just before sending it, and any processing done with the received message at the other end are considered as internal events. The granularity of events to define what constitutes an internal event is application dependent. We define I as the set of all internal events, and C as the set of all communication events.

A *directly happened before* relation $<_d$ is defined over I. Let $x, y \in I$ be any two events. Then $x <_d y$ iff i) x and y are events in the same process and x occurs immediately before y or ii) x is directly followed by the send of a message from $P_{site(x)}$ to $P_{site(y)}$ and y immediately follows the receipt of the message. $P_{site(x)}$ and $P_{site(y)}$ are the processes where events x and y take place respectively. The transitive closure of $<_d$ defines the partial order $\tilde{I} = (I, <_I)$. Let $O \subseteq I$ be the set of observed events. $\tilde{O} = (O, <_O)$ is the partial order induced on O due to $\tilde{I}$. Our aim is to extract the $<_O$ relation, given the set of events O.

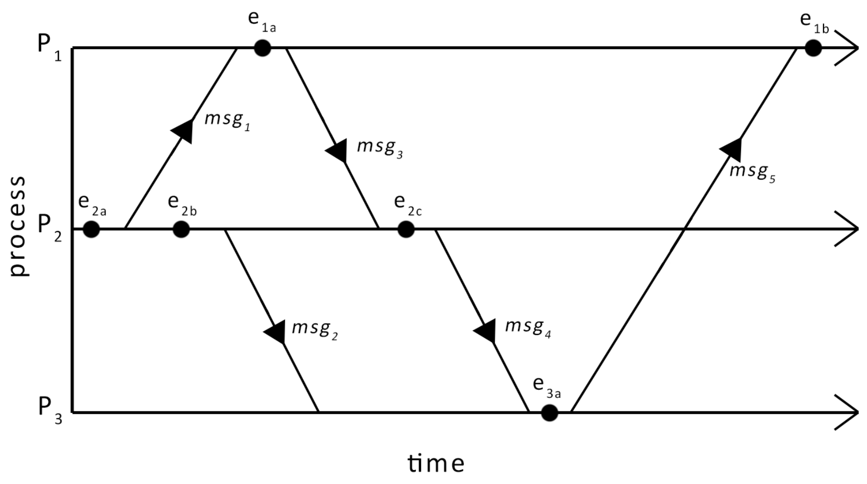

Figure 1: Space Time Diagram

In figure 1 the relation $<_d = \{(e_{2a}, e_{1a}), (e_{2a}, e_{2b}), (e_{1a}, e_{1b}), (e_{1a}, e_{2c}), (e_{2b}, e_{2c}), (e_{2c}, e_{3a}), (e_{3a}, e_{1b})\}$. So the transitive closure comes out to be $<_I = \{(e_{2a}, e_{2b}), (e_{2a}, e_{2c}), (e_{2a}, e_{3a}), (e_{2a}, e_{1a}), (e_{2a}, e_{1b}), (e_{2b}, e_{2c}), (e_{2b}, e_{3a}), (e_{2b}, e_{1b}), (e_{2c}, e_{3a}), (e_{2c}, e_{1b}), (e_{1a}, e_{1b}), (e_{1a}, e_{2c}), (e_{1a}, e_{3a}), (e_{3a}, e_{1b})\}$. The aim is to extract $<_I$. We have the following performance measures to compare the different algorithms. These measures are more qualitative than quantitative. We usually give a value of *strong* or *weak* (sometimes *medium*) to these measures, rather than assigning a particular numerical value.

1. **Intrusion** - Intrusion measures the amount of information maintained at every process and the amount of extra information sent along with each message when a process communicates with another process. This information is later used to reconstruct the partial order relationship. As we will see later if this information, maintained and sent, quantitatively is of the order of the number of processes (n), then we say that there is *strong* intrusion, while if it is of constant size, like a single scalar, then we have *weak* intrusion. Obviously *weak* intrusion is

desired. Jard and Jourdan farther defined another level of intrusion called *medium* intrusion, where the information sent is not of constant size but is bounded by some value k, usually less than n.

2. **Filtering** - Filtering measures the size of the subset O qualitatively whose partial order can be found out. If we can find the partial order relationship for any subset O of I, then we say that we have *strong* filtering while otherwise we have *weak* filtering. *Strong* filtering is desirable.

3. **Consistency** - All the events belonging to O whose partial order needs to be found out may not be available to the process computing the partial order at the same time. They may be sent to that process in any order. Let $H \subseteq O$ be the subset of events available to the process at any time. $\tilde{H} = (H, <_H)$ is the partial order induced due to $\tilde{O}$ on H. If we can find $<_H$ for any H, then we have *strong* consistency, otherwise the consistency is *weak*. *Strong* consistency is desirable.

3 Solution Approaches

The first attempt at ordering events came from Lamport [1]. Lamport defined the *Happened Before* relationship which is same as $<_I$. A logical clock is maintained at every process that is incremented after each internal event. Every internal event and messages sent from a process, is timestamped with the current clock value. Upon receipt of the message the receiver process updates its clock value to the sent timestamp if it is greater than its present clock value.

Algorithm 1 Lamport's Logical Clock for process P_i

```
1: $T_i \leftarrow 0$
2: After every internal event $x$, $T(x) \leftarrow T_i, T_i \leftarrow T_i + 1$
3: When sending message $msg$, $msg.T \leftarrow T_i$
4: Upon receiving message $msg$, $T_i \leftarrow \mathbf{max}(T_i, msg.T)$
```

In figure 1 Lamport timestamps for all the events are $T(e_{1a}) = 1$, $T(e_{1b}) = 4$, $T(e_{2a}) = 0$, $T(e_{2b}) = 1$, $T(e_{2c}) = 2$ and $T(e_{3a}) = 3$. We can define a total ordering $\Leftarrow$ of the events, given Lamport's logical clock values of every event and an ordering of the processes. For any two events x and y occurring in P_i and P_j respectively, $x \Leftarrow y$ if i) $T(x) < T(y)$ or, ii) $T(x) = T(y)$ and $P_i < P_j$. The problem with Lamport is that the total order defined does not reflect the true partial order. $x < y$ implies $x \Leftarrow y$ but the other way round is not true. Concurrent events that do not affect each other might get different timestamps. Thus Lamport's approach only gives one of the possible orderings. A correct implementation of Lamport's *Happened Before* relationship was provided by Fidge.

Instead of a scalar, Fidge [2] used vector timestamps for clock values. Let n be the number of processes. An n dimensional logical clock T is kept at every process, all components initialised to zero. $T(x)$ is the timestamp of an event x. For the process P_i, $T_i[j]$ is the number of events that has happened at P_j before the last communication from P_j to P_i via any number of processes. The array is updated as follows-

Algorithm 2 Fidge's vector timestamp for process P_i

```
1: $n \leftarrow \textit{number of processes}$
2: $T_i[j] \leftarrow 0, \forall j = 1$ to $n$
3: After every internal event $x$, $T(x) \leftarrow T_i, T_i[i] \leftarrow T_i[i] + 1$
4: When sending message $msg$, $msg.T \leftarrow T_i$
5: Upon receiving message $msg$, $T_i[j] \leftarrow \mathbf{max}(T_i[j], msg.T[j]), \forall j = 1$ to $n$
```

In figure 1, vector timestamps to the events are $T(e_{1a}) = [0, 1, 0]$, $T(e_{1b}) = [1, 3, 1]$, $T(e_{2a}) = [0, 0, 0]$, $T(e_{2b}) = [0, 1, 0]$, $T(e_{2c}) = [1, 2, 0]$ and $T(e_{3a}) = [1, 3, 0]$. For any two vector timestamps $T(x)$ and $T(y)$, $T(x) \neq T(y)$ iff $T(x)[k] \neq T(y)[k]$ for any k, and $T(x) \leq T(y)$ iff $T(x)[k] \leq T(y)[k]$ for all k. $T(x) < T(y)$ if $T(x) \leq T(y)$ and $T(x) \neq T(y)$. Ordering the events according to T gives the correct partial order relationship. Like here we get the relation $\{(e_{2a}, e_{1a})$, (e_{2a}, e_{2b}), (e_{1a}, e_{2c}), (e_{2b}, e_{2c}), (e_{2a}, e_{2c}), (e_{1a}, e_{3a}), (e_{2c}, e_{3a}), (e_{2b}, e_{3a}), (e_{2a}, e_{3a}), (e_{1a}, e_{1b}), (e_{3a}, e_{1b}), (e_{2c}, e_{1b}), (e_{2b}, e_{1b}), $(e_{2a}, e_{1b})\}$, which is same as $<_I$ (calculated before), and thus the *Happened Before* relation. The main problem with this approach is the bit complexity of the algorithm because every message incurs $O(n)$ overhead. Even if at any point of time only a handful of processes are interacting with each other, an entry for every process is carried in the message timestamp. This incurs *strong* intrusion, which we know is undesirable. Thus we are faced with the challenge of making the implementation of vector clocks more efficient.

As noted before there are three solution approaches. Let $\tilde{O} = (O, <_O)$ be the partial order induced due to $\tilde{I}$. Let $\downarrow_O x$ be the set of all predecessors of event x in O. Let $O_j \subseteq O$ be the events occurring in P_j.

1. **Transitive Dependency** - Let $V_x[j] = \{\downarrow_O x \cap O_j\}$ for an event x, which is the set of events in P_j that x depends upon. In this approach we maintain this value for every event x.

2. **Direct Dependency** - We define a relation $\lhd$ on O, such that for any two events x and y occurring in processes P_i and P_j respectively, $y \lhd x$ iff i) $i = j$ and $y <_{O_i} x$ or ii) $i \neq j$ and $\exists x', y'$ (y' is immediately followed by the send of a message from P_j to P_i and x' immediately follows the receipt of the message) such that $y <_{O_j} y'$ and $x' <_{O_i} x$. Let $D_x[j] = \{y \in O_j : y \lhd x\}$, which denotes the number of events in P_j preceding the latest message exchange from P_j to P_i before the occurrence of event x in P_i. In this approach we maintain this value for every event x.

3. **Pseudo-direct Dependency** - We define a relation $\ll$ on O, called the pseudo-direct relation, such that for any two events x and y occurring in processes P_i and P_j respectively, $y \ll x$ iff i) $i = j$ and $y <_O x$ or ii) $i \neq j$ and $\exists s_1, r_1, s_2, r_2, \ldots, s_k, r_k$ such that s_i is the sending of a message from $P_{site(s_i)}$ to $P_{site(r_i)}$ and r_i is the corresponding receipt, y happened before s_1 on P_j, r_l directly happened before s_{l+1} and r_k directly happened before x on P_i. Let $M_x[j] = \{y \in O_j : y \ll x\}$, which denotes the number of events that occurred on P_j that precede the existence of a path of interactions from P_j to P_i, starting after y on P_j and ending before x in P_i, without any observed event after leaving P_j. In this approach we make use of this value.

The model we will assume from now on will be asynchronous, with reliable communication. We don't concern ourselves with faulty nodes and every process can send messages directly to any other process. Processes have information about the total number of processes, although every algorithm can be modified to allow for the absence of this pre-requisite.

3.1 Transitive Dependency

Mukesh Singhal and Kshemkalyani's [3] approach tries to reduce the $O(n)$ overhead that is faced in vector timestamping method during message exchanges. The main intuition is to send only that information that has changed since the last message exchange. Every process along with the vector timestamp T, maintain two other arrays LS and LU. For process P_i, $LS_i[j]$ is the local time when P_i last sent a message to P_j and $LU_i[j]$ is the local time when P_i updated $T_i[j]$. So if $LU_i[k] > LS_i[j]$, this means P_i has updated $T_i[k]$ after sending the last message to P_j, and in the next message exchange with P_j, P_i should include $T_i[k]$ in the message overhead. We have tweaked the original algorithm a little so that we update LU_i only when a local internal event occurs, and not during message send or receive. T_i is however incremented for every internal and send event.

Algorithm 3 Singhal and Kshemkalyani's Method for Process P_i

```
1: $n \leftarrow$ *number of processes*
2: $T_i[j] \leftarrow 0, \forall j = 1$ to $n$
3: $LU_i[j] \leftarrow \perp, \forall j = 1$ to $n$
4: $LS_i[j] \leftarrow \perp, \forall j = 1$ to $n$
5:
6: function InternalEvent($x$)    ▷ After completion of internal event $x$
7:     $T(x) \leftarrow T_i$
8:     $LU_i[i] \leftarrow T_i[i]$
9:     $T_i[i] \leftarrow T_i[i] + 1$
10:
11: function SendMsg($msg, j$)    ▷ message send from process $P_i$ to $P_j$
12:     $msg.T \leftarrow []$
13:     for $k \leftarrow 1$ to $n$ do
14:         if $LU_i[k] \neq \perp$ and $(LS_i[j] = \perp$ or $LS_i[j] < LU_i[k])$ then
15:             $msg.T$.append($\{k, T_i[k]\}$)
16:     $LS_i[j] \leftarrow T_i[i]$
17:     $T_i[i] \leftarrow T_i[i] + 1$
18:     send($msg, j$)
19:
20: function ReceiveMsg($msg$)    ▷ message receive at process $P_i$
21:     for $k \in msg.T$ do
22:         if $msg.T[k] > T_i[k]$ then
23:             $T_i[k] \leftarrow msg.T[k]$
24:             $LU_i[k] \leftarrow T_i[i]$
```

In figure 2 msg_3 is sent with contents $\{(2, 10), (3, 4)\}$. After that two events take place in P_2, a message is received from P_1 that updates $T_2[1]$ to 8 and an internal event e_{2c} occurs at P_2. This updates $LU_2[1]$ and $LU_2[2]$, making them larger than $LS_2[3]$, the time P_2 last sent a message to P_3. Thus in the next message exchange from P_2 to P_3, msg_4 carries these new changes as $\{(2, 12), (1, 8)\}$.

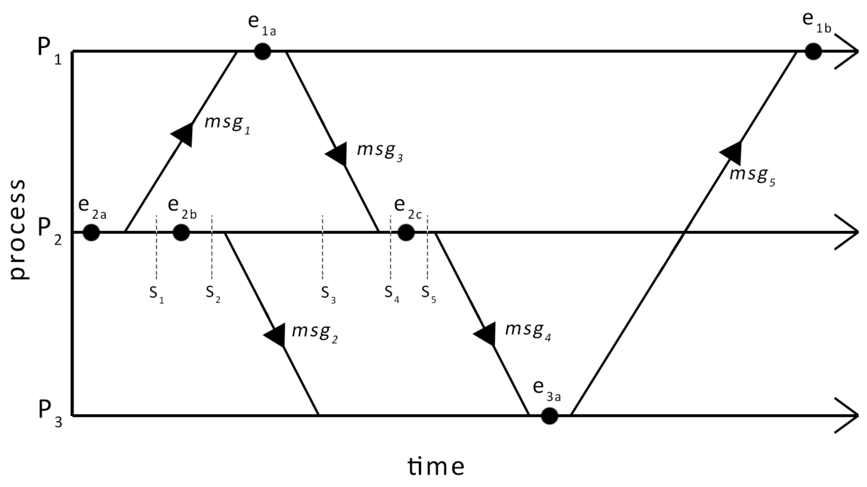

Figure 2: Space Time Diagram

	State 1			State2			State 3			State 4			State 5		
	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
T_2	6	9	4	6	10	4	6	11	4	8	11	4	8	12	4
LS_2	8	-	4	8	-	4	8	-	10	8	-	10	8	-	10
LU_2	2	7	5	2	9	5	2	9	5	11	9	5	11	11	5

EVENTS				MESSAGES	
	P1	P2	P3		
e_{2b}	6	9	4	msg_3	{(2,10), (3,4)}
e_{2c}	8	11	4	msg_4	{(2,12), (1,8)}

Table 1: Change in T_2, LS_2, and LU_2 values, Content of $e_{2b}, e_{2c}, msg_3, msg_4$

An additional requirement of this algorithm is that we need FIFO channels. This is because we always send along information that has changed since the last message exchange. So in the case of non FIFO channels, if newer messages reach the destination before older messages, the receiving process will miss out some information and incorrectly timestamp its events that occur between the receipt of the new and old message. Figure 3 explains this.

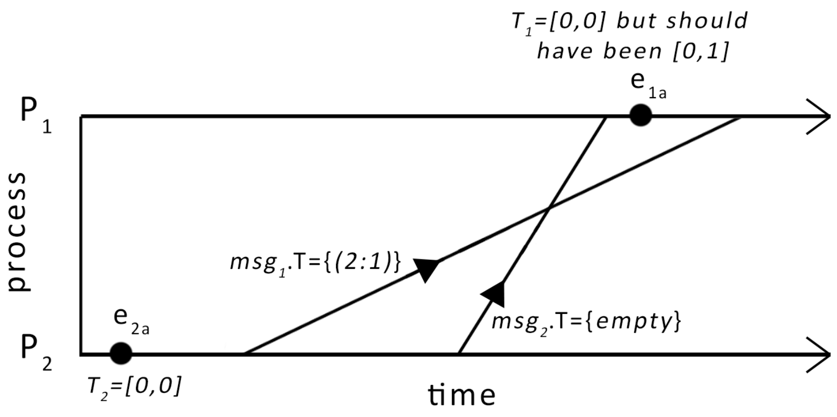

Figure 3: Non-FIFO Channels

Since we maintain the whole *Happened Before* relation at every event, we can reconstruct the partial order relationship for any subset of internal events. Therefore this process has *strong* filtering. Also since the timestamp of every event tells the number of events it depends upon at different processes, we only depend on the timestamp values and not on the order in which these events arrive for computing the partial order relationship. So we also have *strong* consistency. Although the message overhead is less, in the worst case where there are frequent message exchanges we don't benefit from this approach compared to Fidge's method. So the intrusion is still *weak*.

3.2 Direct Dependency

Fowler and Zwaenepoel [4] approach does not maintain the entire *Happened Before* relation but only direct dependencies as defined by the $\triangleleft$ relation. So to construct the entire partial order, we call *VisitEvent* method recursively where the k^{th} recursive invocation updates the causal dependency vector if any process affects the given event through k hop message exchanges. After n recursions, where n is the number of processes, the causal dependency vector has the same value as the direct dependency vector associated with the most recent event at any process that affects the given event. We have made a minor change in the algorithm by introducing a new data structure M_i that is a mapping from sequence number of the event to the event in P_i.

Algorithm 4 Fowler and Zwaenepoel's Method for Process P_i

```
1: $n \leftarrow$ *number of processes*
2: $D_i[j] \leftarrow 0, \forall j = 1$ to $n$
3: $M_i \leftarrow \{\}$
4:
5: function InternalEvent($x$)    ▷ After completion of internal event $x$
6:     $D(x) \leftarrow D_i$
7:     $D_i[i] \leftarrow D_i[i] + 1$
8:     $M_i(D_i[i]) \leftarrow x$
9: function SendMsg($msg$)    ▷ message send from Process $P_i$
10:     $msg.T \leftarrow D_i[i]$
11:     send($msg$)
12: function ReceiveMsg($msg, j$)    ▷ message received from Process $P_j$
13:     $D_i[j] \leftarrow msg.T$
14:
15: function FindCausalDependency($x$)
16:     $CD[j] \leftarrow 0, \forall j = 1$ to $n$
17:     $CD[i] \leftarrow D(x)[i]$
18:     VisitEvent($i, D(x)[i] + 1$)
19: function VisitEvent($j, seq$)    ▷ visit event for $P_j$ on event number $seq$
20:     $x \leftarrow M_j(seq)$
21:     for $\forall k \neq j$ do
22:         $a = D(x)[k]$
23:         if $a > CD[k]$ then
24:             $CD[k] \leftarrow a$
25:             VisitEvent($k, a$)
```

With reference to figure 4, we want to find the causal dependencies of event e_{1b}. We initialise $CD = [1, 0, 0]$ and call *VisitEvent* on P_1 and event e_{1b}. Referring table 2, we see that $D(e_{1b}) = [1, 1, 1]$. Thus we call *VisitEvent* on P_2 and event e_{2a} after updating CD to $[1, 1, 0]$. $D(e_{2a}) = [0, 0, 0]$, so there is no change to CD and we return. *VisitEvent* is called on P_3 and event e_{3a} after updating CD to $[1, 1, 1]$. $D(e_{3a}) = [0, 3, 0]$, so *VisitEvent* is farther called on P_2 and event e_{2c} after updating CD to $[1, 3, 1]$. There is no farther change and we get $CD = [1, 3, 1]$.

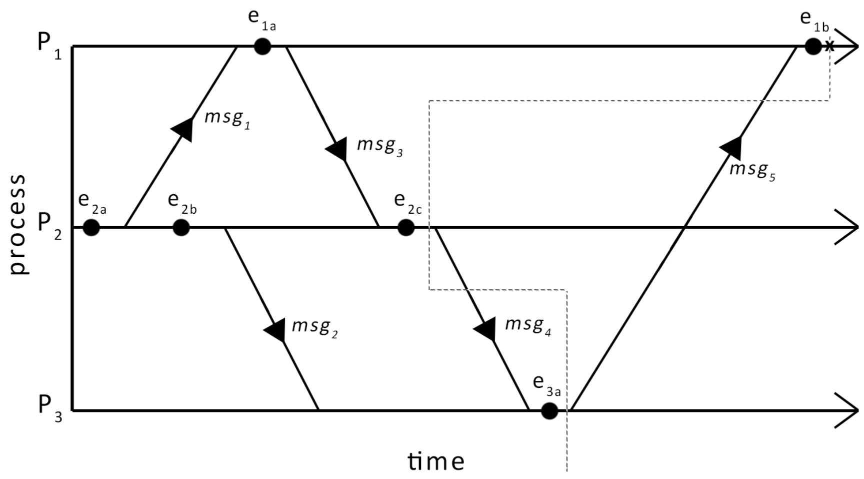

Figure 4: Space Time Diagram

	e_{1a}	e_{1b}	e_{2a}	e_{2b}	e_{2c}	e_{3a}
P1	0	1	0	0	1	0
P2	1	1	0	1	2	3
P3	0	1	0	0	0	0

Table 2: Dependency vectors of all events

The main problem with this approach is that it suffers from both *weak* filtering and *weak* consistency. Consider this example:

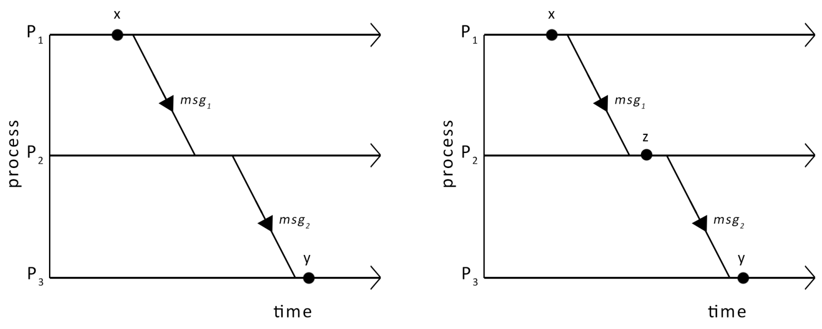

Figure 5: Without *NIVI* and with *NIVI*

In the first diagram of figure 5, $D(x) = \{\}$ and $D(y) = \{\}$. So if we were to take $O = \{x, y\}$ we will not be able to deduce from just $D(x)$ and $D(y)$

that y is causally dependent on x. The main reason for this is the absence of any internal event between receive and send of message exchange occurring between two events of two different processes, over more than one hop. Jard and Jourdan called this interaction as *Invisible Interaction.* The absence of such interactions is called *Non Invisible Interaction*, or *NIVI. Strong* filtering can only be ensured if the subset O satisfies *NIVI*, as in the second figure where we have $D(x) = \{\}$, $D(z) = \{x\}$ and $D(y) = \{z\}$.

This method is *weakly* consistent. Let $H \subseteq O$ be the set of events available at any moment. Let $x \in H$ be an event. If $\exists y \in D(x), y \notin P_{site(x)}$ such that $y \notin H$ then until and unless y is not available we will not be able to call *VisitEvent* on process $P_{site(y)}$ and event y. Thus H has to be downward closed (according to the poset chain of the Hasse diagram) for the method to be *strongly* consistent. This condition was defined as *IDEAL* by Jard and Jourdan on H.

3.3 Adaptive Timestamps

Jard and Jourdan [5] tried to combine the best of both Fowler-Zwaenepoel's direct dependency method and the vector timestamp approach which was improved in Mukesh-Kshemkalyani's work. If you consider the *Hasse* diagram of events, the intuition was to record for every event the just immediate event following it for every distinct poset chain the event lay on. These predecessor events were called pseudo-direct predecessors. For an event x, we record the pseudo-direct predecessors in $PT(x)$. To get all the predecessors of an event x we have to recursively examine again the pseudo-direct predecessors of the events in $PT(x)$. Unlike Fowler-Zwenepoel's approach, that requires the occurrence of an event after the receive of a message and before any message is sent out, Jard and Jourdan's method works without the requirement of *NIVI.*

We also have a bound on the size of message that can be exchanged in Jard-Jourdan's method. Whenever upon the receipt of a message and after the union of timestamps, the size of PT_i becomes more than K, we make a *NULLEVENT* to occur which brings down the size to one. This bounds the message timestamp size to K. Because of this, we say that Jard-Jourdan's method has *medium* intrusion.

We have *strong* filtering because we can find the partial order relation $<_O$ for any subset $O \subseteq I$. However since we are not maintaining the entire set of predecessors, this method has *weak* consistency.

Algorithm 5 Jard-Jourdan Method for Process P_i, with bound K

```
 1: $PT_i \leftarrow [\{i, 0\}]$
 2:
 3: function InternalEvent($x$)          ▷ After completion of internal event $x$
 4:     $PT(x) \leftarrow PT_i$
 5:     $PT_i \leftarrow [\{i, PT_i[i] + 1\}]$
 6:
 7: function SendMsg($msg$)              ▷ message send from process $P_i$
 8:     $msg.T \leftarrow PT_i$
 9:     send($msg$)
10:
11: function ReceiveMsg($msg$)           ▷ message receive at process $P_i$
12:     for $k \in msg.T$ do
13:         if $k \in PT_i$ then
14:             $PT_i[k] \leftarrow$ max($msg.T[k], PT_i[k]$)
15:         else
16:             $PT_i$.append($\{k, msg.T[k]\}$)
17:     if sizeof($PT_i$) $> K$ then
18:         InternalEvent($NULLEVENT$)
```

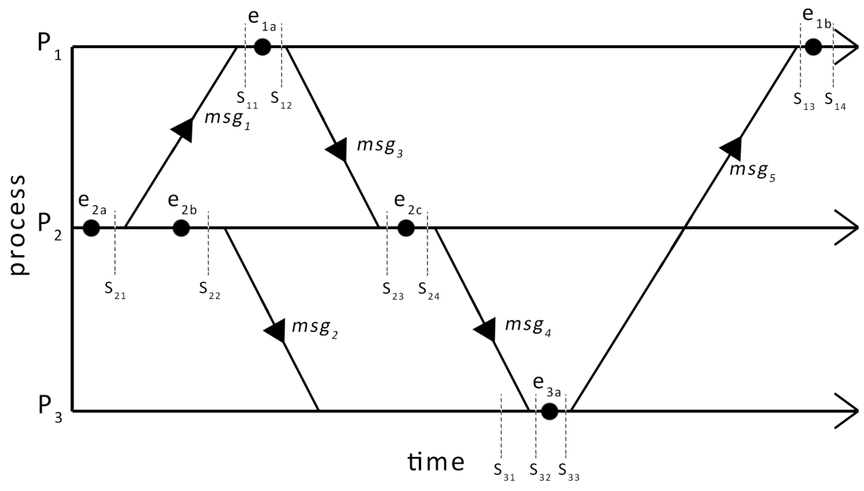

Figure 6: Space Time Diagram

	PT_1		PT_2		PT_3
S_{11}	{(1,0), (2,1)}	e_{2a}	{(2,0)}	S_{31}	{(2,2),(3,0)}
e_{1a}	{(1,0), (2,1)}	S_{21}	{(2,1)}	S_{32}	{(2,3),(3,0)}
S_{12}	{(1,1)}	e_{2b}	{(2,1)}	e_{3a}	{(2,3),(3,0)}
S_{13}	{(1,1),(3,1)}	S_{22}	{(2,2)}	S_{33}	{(3,1)}
e_{1b}	{(1,1),(3,1)}	S_{23}	{(1,1), (2,2)}		
S_{14}	{(1,2)}	e_{2c}	{(1,1), (2,2)}		
		S_{24}	{(2,3)}		

	Timestamp		Timestamp
msg_1	{(2,1)}	msg_4	{(2,3)}
msg_2	{(2,2)}	msg_5	{(3,1)}
msg_3	{(1,1)}		

Table 3: PT values at different states. Event and message timestamps

4 Comparison

We compare the three approaches we have discussed so far in this table.

	Filtering		Consistency		Intrusion		
	Strong	Weak	Strong	Weak	Strong	Medium	Weak
Transitive Dependency	Y		Y		Y		
Direct Dependency		Y		Y			Y
Pseudo-direct Dependency	Y			Y		Y	

Table 4: Table comparing the three approaches

5 Conclusion

Finding dependencies among events is of crucial importance in distributed systems. In this report, we have seen how we can achieve the same by maintaining logical clocks at every process. We have compared the pros and cons of different solution approaches. We have discussed properties and applications of system clocks and how efficiently they can be implemented.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[2] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, pp. 56–66, 1988.

[3] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Information Processing Letters*, vol. 43, no. 1, pp. 47 – 52, 1992.

[4] J. Fowler and W. Zwaenepoel, "Causal distributed breakpoints," in *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pp. 134–141, May 1990.

[5] C. Jard, G. V. Jourdan, T. Jeron, and J. X. Rampon, "A general approach to trace-checking in distributed computing systems," in *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, pp. 396–403, Jun 1994.