Causality and clocks in a dynamic distributed setting

Sai Rithwik M (IMT2018061)

Mandate 1 - AI 704: Multi Agent Systems

Abstract

With the increased adoption of distributed systems in P2P sharing, version tracking and many scenarios, it is common to have a highly variant number of nodes participating in the system. Vector Clocks cannot optimally identify causality in a large dynamic distributed setting. This report covers a few topics related to identifying causality and discusses some optimisations to Vector Clocks. The report further details how causality can be tracked in dynamic systems using Interval Tree Clocks.

Keywords: Lamport Clocks, Vector Clocks, Version Clocks, Interval Trees, Keyspaces, Global Identifiers

1 Introduction

Usage of a single global clock has proven to be highly unreliable in distributed settings due to various synchronisation issues caused by independent tick rates in the distributed system. Hence ordering of events in distributed systems is essentially very important. The system thereby needs an invariant to judge the ordering of the events. Causality is an important theory that loosely states that a cause always precedes the effect for an event. For events to be causally linked to each other Lamport[3] has defined the term Partial Ordering where an event e_1 could be the cause of e_2:

- if they happened on the same node in the system and e_1 happened before e_2 or
- if happened in different nodes, e_2 somehow knows that e_1 has happened before due to some message that some other node has sent about e_1 to the node on e_2

If we cannot deduce the ordering of the two events, it simply means that they are concurrent.

Over the course, various techniques like Vector Clocks [4] and Matrix Clocks have been designed to identify the partial ordering between events. One of the major disadvantages of these causality tracking techniques is that these implementations have been designed under the assumption of a fixed, well known, set of nodes. These causality tracking techniques are unsuitable in a dynamic setting where the number of nodes in the system is highly variant. There have been various proposals for such systems which accommodate dynamic creation and termination of nodes in a system[5, 6]. However, most of them were either practically impossible or did not support localised termination of a node. Localised termination essentially implies that all the active nodes must agree to terminate a specific node. Even if there is a single node that is unreachable, termination stalls.

Hence this situation calls for a model which would generalise the Vector Clocks to a dynamic system. The following section discusses the shortcomings of vector clocks and how they are unsuitable in certain situations. Section 3 discusses the basic architecture of Interval Tree Clocks. Section 4 deals with how causality is determined in this setup. Section 5 addresses a few trivial but interesting questions that I had stumbled on reading the paper.

2 Issues with Vector Clocks and workarounds

To provide a perfect causal ordering we would need a data structure that would essentially capture

- its local progress
- a representation of global time

Vector Clock is a structure where the global time is represented by an n-dimensional vector. Charron-Bost[9] showed that the dimension of vector clocks cannot be less than n, for causality to hold.

If the number of tasks to perform increases to a larger number, this data structure would essentially require carrying a lot of vectors to determine the ordering between two events. This overhead grows linearly with the number of nodes in the system and when there are thousands of nodes in the system, the message size becomes huge even if there are only a few events occurring in a few nodes. There have been various optimisations to reduce the communication bandwidth in this regard. But according to the results by Charron-Bost we would still have to deal with vectors of size n. We can hence conclude that:

- Vector Clocks are an accurate representation of causality.
- Even with the optimisations on the Vector Clock algorithm, identifying causality through event timestamps is a complex task in a dynamic setting.

Peter Bailis[8] in his article describes some solutions to reduce the cost overhead by the vector clocks briefly. One of the primary methods is to reduce the number of comparisons. As discussed earlier, vector clocks are overloaded. There might be situations when it is not required to find out causality relationship between all events in the system. Imagine if we have a 2 node system with nodes labelled A and B. The first event propagates from A to B, so the vector clock changes from [1, 0] to [1, 1]. We can say that [1, 0] $\rightarrow$ [1, 1]. Now imagine if we have 20 further updates only in node B. So according to vector clocks it should change as [1, 2], [1, 3] ... [1, 21]. We observe that we can still say [1, 0] $\rightarrow$ [21, 1]. So updating from [1, 1] to [1, 21] didn't add any value in comparison of causality between two node states. If there was a way where all this could have been skipped it would have been able to reduce the size of unnecessary computations. This might be helpful in a version tracking scenario where nodes represent replicas of a particular file. A new version could be represented as a new node forked from the original node.

This problem is essentially solved by the usage of Version Vectors[14, 15]. Vector clocks and Version Vectors are almost the same, but as discussed earlier, for a certain set of tasks Version Vectors would outperform Vector Clocks. The most distinguishing feature among them is that Vector Clocks are used to track causality among all the events in the server. However, Version Vectors track updates to keys in a given node. So an instance of a version vector is stored per key and not per node. Again, this is useful in scenarios where the number of keys might be lesser than the number of nodes. This thereby reduces the complexity from $O(n)$ to $O(k)$ where k is the number of keys.

A small example[11] is shown in the figure to help understand how Vector Clocks perform better in some scenarios.

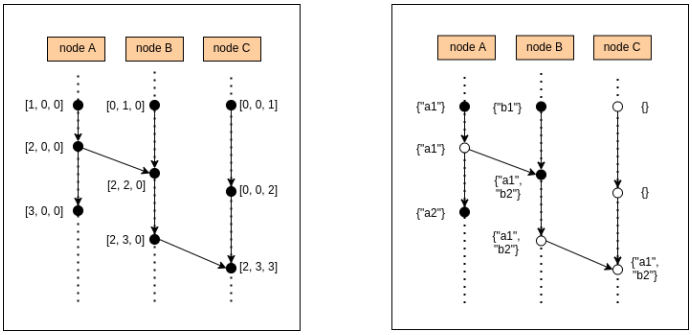

Figure 1: Comparison b/w Vector clocks and Version vectors

Notice how in Figure 1.2 only the keys are being updated. Key "a1" loosely means version 1 in node A. By the end of all the events we can see that Vector Clocks used up a space of 3 units whereas Version Vectors just used up 2 units of space. In fig 1.2 we see that not all updates are depicted using •. Some are ∘, it just means that causality is not tracked in that situation.
Now thinking it for a large scale, we can see that in a scenario where the number of keys are less it would be really helpful to ascertain partial ordering. This was one probable way to reduce the dimension for vector clocks. However, using Version Vectors does not solve the problem of dynamic updates, as, at some point of time, there needs to be garbage collection implemented here too. Now using both the concepts from Vector Clocks and Version Vectors we would define something called Interval Tree Clocks[1] which could help us efficiently perform dynamic operations in a distributed setting.

3 Interval Tree Clocks

From the above sections, we can infer that the main problem with Vector Clocks are they keep on growing linearly as the number of nodes join the system. And in a dynamic setting where there will be addition and deletion of nodes frequently, it is a pretty sticky situation to fall into[7]. Garbage collectors must be used at some instance to prevent the size of vectors to keep from growing[3].

Interval Tree Clock is a data structure that could help us with this scenario. Roughly this acts like Git VCS. The goal of Interval Tree Clock(ITC) is to represent the same information as a vector clock, but in a way that permits nodes to be created and removed on the fly without garbage collectors, thus preventing actor explosions[10].

ITC uses a Fork-Event-Join Model to track causality. These are defined as follows:

- **fork**: It is similar to forks that we use in VCS. You get a similar copy of the history with a different user id. In the case of ITC, it is the same. The node which triggers the fork operation gets a copy of the copy of causally known events till that moment, with a different id. Mathematically fork can be depicted as:

$$fork(id, e) = ((id_1, e), (id_2, e))$$

- **peek**: It is similar to fork but no new id is created. id_2 is depicted as 0. It is used for transmitting causal information of the event

$$peek(id, e) = ((id, e), (0, e))$$

- **event**: It is an operation as we know in Lamport clocks. If a new event e' is created from an event e, then the causal ordering must be e $\rightarrow$ e'.

- **join**: On comparing again with VCS model, this is similar to the merge operation. This operation joins two identifiers to a new one. Event merging happens by comparing the histories and is obtained by union $\mathrm{e}_1 \bigcup e_2$.

$$join((id_1, e_1), (id_2, e_2)) = (id_3, e_3)$$

- **send**: It is depicted by a composition of event and peek.

- **receive**: It's a special case of join where id_2 is 0. This is typically used to receive information

$$receive((id_1, e_1), (0, e_2)) = (id_3, e3)$$

- **sync**: It is depicted by the composition of join and fork. Join first synchronises the event by union and then fork is done to divide them back into two different nodes.

The figure below shows some core operations pictorially

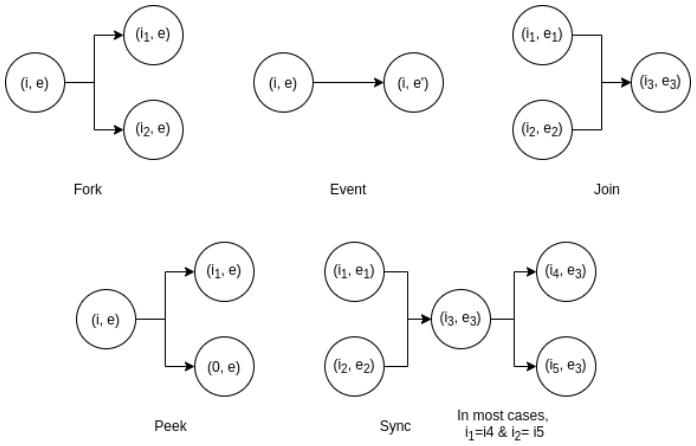

Figure 2: Some core operations in ITC

The next part of the section explains how ITCs essentially function. The data structure is based on the fact that the clock which is used to measure causality or conflicts are based on keyspace regions.

On forking, the parent of the fork gives away part of its keyspace to the children that emerge out of the fork. An example is given below in Fig. 3 to illustrate this. Initially, the system starts with

a single node and a keyspace of {1}. Then on forking, it generates two nodes with keyspaces in the range [0, 1/2) and [1/2, 1), which is depicted as {1, 0} and {0, 1}. Imagine a new node wants to join the system. Let's assume it approaches the node with keyspace of range [0, 1/2). This node now gives a part of keyspace to the new node by forking. So the new children formed by the forks are depicted as {{1, 0}, 0}, {{0, 1}, 0}. Now let's assume that a node wants to retire. Let this node be {{0, 1}, 1}. It needs to merge the events that it has observed till now with some node in the system. The mechanism by which events are tracked in the data structure is explained later. Now this node contacts a random node {0, 1} and merges to form {{0, 1}, 1}. This is explained in the figure given below.

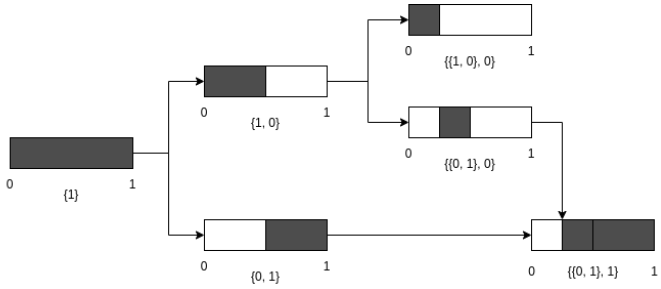

Figure 3: Keyspaces in ITC

With this method of keyspace generation, we can assure that generation, termination and recycling of identifiers for nodes can happen without causing identity conflicts. Once the identifiers are allocated, event tracking should be taken care. We need to understand that any event that occurs can only be added to the current id of the node. As long as the subset of the id is not null, the event can inflate itself into a subspace. A simple example of how ITC functions is as follows.

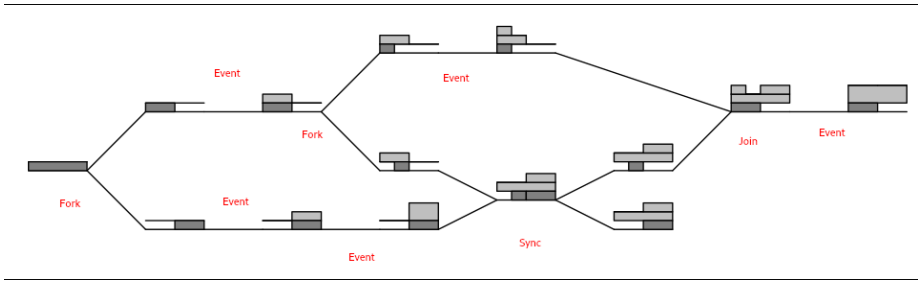

Figure 4: An example of ITC
Image Courtesy [1]

1. Initially the root node forks into two nodes. The top node is represented by {1, 0} and the bottom one as {0, 1}.

2. Events occur in both nodes, {1, 0} and {0, 1}. Hence we see a light grey box placed over the line. The dark grey box represents the keyspace range for a given node and light grey

represents the events. Remember that events can only be inflated within any subset of the node's keyspace.

3. The node {1, 0} forks and becomes {{1, 0}, 0} and {{0, 1}, 0}. Notice how in node {{1, 0}, 0} event is added only within the key-range [0, 1/4).

4. The nodes {{0, 1}, 0} and {0, 1} trigger a synchronise function. As discussed earlier, synchronise is a merge of join and fork. Join merges the keyspace and updates the events in both and they fork back and nodes get their respective keyspaces back. Notice how at the end of the sync operation, the events have merged using a union operation.

5. Again the nodes perform join operations and an event occurs in the keyspace. The last node that is depicted in the image is an example of a normal form. Normal forms and merge conflicts are not discussed in this report.

4 Tracking Causality

There can be implementations for both Vector Clocks as well as Version Vectors using ITC. If the user's intention is to track causal ordering in the system, a vector clock can be used. Otherwise, if just the causal ordering of key modifications are required, Version Vectors can be used. This report discusses only implementing a dynamic Vector Clock using Interval Trees. The figure below shows how causality tracking is similar in both Vector Clocks and ITC. An example of how ordering of two stamps can be determined in ITC graphs is shown in Fig. 5

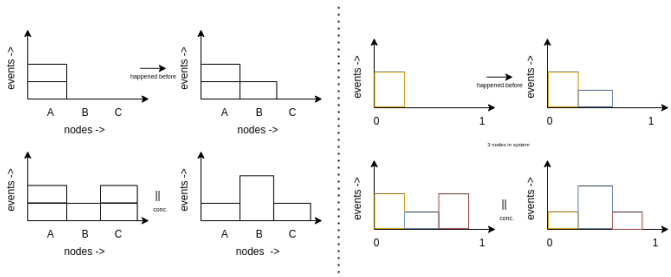

Figure 5: Comparision between two stamps in Vector Clock and ITC

Fig. 6 shows how static Vector Clocks can be implemented using ITC. The top left corner shows the corresponding Vector Clock implementation. In Fig. 6 we can see that whenever a new node enters the system a fork is created from a node and when a node wants to communicate with another node a send operation is performed, and hence the local clock for that node is updated.

In a future when all the nodes want to retire, they can perform a join operation, with some node in system, pass on their history and retire.

Fig. 5 & 6 give a brief description of how events help us determine whether two stamps are concurrent or not. From Fig 5 in the ITC implementation, we see that two stamps are deemed concurrent if their event histories are conflicting just like in Vector Clocks.

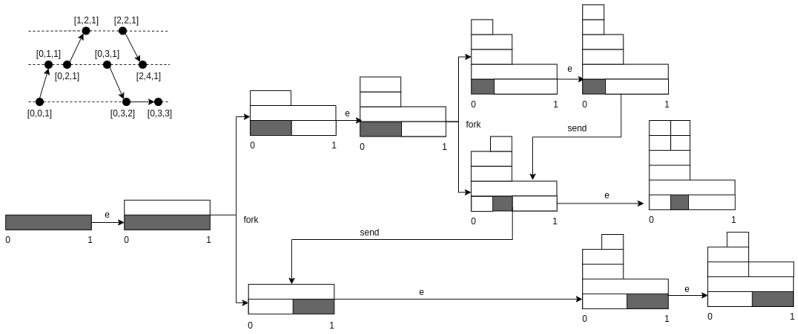

Figure 6: Vector Clock implementation in ITC

5 Some interesting questions

1. Why are we depicting keyspaces using intervals rather than assigning unique identifiers randomly?

   One reason for doing this is because the keyspace logic always assures us with unique intervals and hence unique IDs. We can clearly see from the above that there is no possibility of conflict between IDs. As discussed earlier a counter-based mechanism would be problematic, as the size of the vector would keep increasing and would not stop.

2. What do events represent?

   Just like in Vector Clocks, where events are used to track the logical time, the functionality is similar in this case too. Once an event is triggered, the count of events in the node's keyspace increases.

3. Why do the events have to increase within the keyspace only?

   To track causality as shown in Figure 5. Incrementing within the keyspace helps us understand whether the timestamps in question are concurrent or not.

4. If Charron-Bost[9] has proved that the dimension of vector clocks cannot be less than n, for causality to hold. How is ITC based causality tracking an efficient mechanism?

   Charron-Bost's principle still holds. It's just that the value of the vector size is static in ITC ie. n. In regular Vector Clocks, the size of array keeps growing and at some instance garbage collection needs to happen. ITC makes sure that once a node retires all the event history it had with other nodes is passed on everytime, hence avoiding any sort of garbage collection issues and keeping space constant.

6 Conclusion

Vector Clocks are required to track the progress of events and measure causality. This report discussed about the shortcomings of vector clocks and how version vectors could help in such scenarios. The report then discussed the actor burst problem where the size of the vector grows linearly with the number of nodes in the system. To counter this the concept of Interval Tree Clocks was presented which helped to identify the causality in a dynamic setting. The paper[1] concludes by showing some test results in a dynamic as well as static setting. In the dynamic scenario, space had stabilised after a certain set of operations, instead of linearly growing like the Vector Clocks.

References

[1] Almeida, Paulo Baquero, Carlos Fonte, Victor. (2008). "Interval Tree Clocks: A Logical Clock for Dynamic Systems." 5401. 259-274. 10.1007/978-3-540-92221-6_18.

[2] Fred Herbert "Interval Tree Clocks"

[3] Leslie Lamport. 1978. "Time, clocks, and the ordering of events in a distributed system." *Commun. ACM 21, 7 (July 1978), 558–565.*

[4] M. Raynal and M. Singhal, "Logical time: capturing causality in distributed systems, in Computer", *vol. 29, no. 2, pp. 49-56, Feb. 1996*

[5] David Ratner, Peter Reiher, and Gerald Popek. "Dynamic version vector maintenance." Technical Report *CSD-970022, Department of Computer Science, University of California, Los Angeles, 1997.*

[6] Richard G. Golden, III. "Efficient vector time with dynamic process creation and termination." Journal of Parallel and Distributed Computing (JPDC), 55(1):109–120, 1998.

[7] Mostéfaoui, Achour Raynal, M. Travers, C. Patterson, Stacy Agrawal, Divyakant Abbadi, A.. (2005). From static distributed systems to dynamic systems. *109- 118. 10.1109/RELDIS.2005.19.*

[8] Peter Bailis "Causality Is Expensive (and What To Do About It)"

[9] Bernadette Charron-Bost. 1991. "Concerning the size of logical clocks in distributed systems." *Inf. Process. Lett. 39, 1 (July 12, 1991), 11–16.*

[10] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput. 7, 3 (March 1994), 149–174.*

[11] Carlos Baquero and Nuno Preguiça. 2016. "Why logical clocks are easy." *Commun. ACM 59, 4 (April 2016), 43–47.*

[12] Justin Sheehy "Why Vector Clocks Are Hard"

[13] Ricardo BCL "Interval Tree Clocks in C"

[14] Martin Fowler "Version Vectors"

[15] Carlos Baquero "Version Vectors are not Vector Clocks"