Hierarchical Vector Clock Implementation and Analysis

Matthew Golla, Rithvik Baddam

I. Introduction

A common problem in distributed systems is determining the order of events in order to analyze the system and the computation, and to solve other problems such as consistency, mutual exclusion, recovery and fault tolerance. To collect such causal knowledge, several solutions have been developed including Lamport clock, vector clock, matrix clock and other variations of logical time systems. However, issues with scalability, large clock overheads, and accurate tracking of order of events still persists. This paper presents and reviews an implementation of hierarchical vector clock algorithm by Khomitsky and Zhuklinets that improves message overhead and scalability, while also increasing accuracy of causality detection in large distributed systems with hierarchical structure involving grouping of nodes by geographic, administrative, and physical attributes.

II. Background

A. Posets and Causal Ordering

A partially ordered set or *poset* is a relation P on set X if it is transitive, reflexive, and antisymmetric. *Hasse diagrams* are used to represent finite posets [1]. A computation $(E, \rightarrow)$ can be thought of as a poset modeled by a Hasse diagram. Events during a computation can be partially ordered based on causality using a logical time mechanism. Causality and partial ordering knowledge establishes happened before relationships between events that can be used to solve problems and analyzing distributed systems. Lamport clock, vector clock, and matrix clock are examples of logical time systems that can be used to determine causality in a computation.

B. Vector Clock

In order to graph the *happened-before relations* between events between processes, the *vector clock system* can be used. This mechanism associates timestamps with events to compare them to indicate whether events are or are not causally related [4]. If events are causally related and thus not concurrent, vector clock timestamps can determine which event happened before the other in a distributed system computation [2]. The algorithm works by having each process P_i have a vector of numbers$V[1 ... n]$ initialized to 0 and can be formally expressed as

```
**P**$_i$:
**for all** Events (Send, Receive, Internal) **do**
    $V[i] := V[i] + 1$
**end for**

To send message to **P**$_j$, $j \neq i$:
    Attach current vector clock to message

On receiving message from **P**$_j$, $j \neq i$:
**for** $i = 1$ to $N$ **do**
    $V_j[i] = max(V_j[i], V_i[i])$
**end for**
```

C. Hierarchical Vector Clock

In modern distributed systems, sets of nodes may be grouped together by geographical, physical, and/or administrative concerns, which form a hierarchical structure where nodes may be communicating more heavily within their groups. On such a network, the distributed system running on it may require a large number of nodes with transport channels crossing hierarchical levels and connecting between multiple domains [3]. Causality knowledge for events in this system is critical for analysis of the system and solving problems like mutual exclusion, consistency, recovery and fault tolerance. To enable efficient encoding of information and recording causality of events in the lattice while also keeping message overhead low, logical time is used.

To capture causal relations for events in distributed systems that form posets containing varying processes, the *hierarchical vector clocks* (varying size) is a mechanism that can be used which requires much smaller communication and storage overhead than a vector clock [3]. This clock grows dynamically as its storage and processing necessities "grow gracefully at approximately a logarithmic rate with the number of distributed sites within the distributed system." [3] The mechanism is hierarchical, scalable, and best used for modern message-passing distributed systems with highly hierarchical communication networks.

III. Design

A. Description

Hierarchical systems involve grouping of processes into a tree structure based on categorical, geographical, or administrative concerns. Each grouping has a parent node, except for the highest level grouping which is the root, and children nodes. At the lowest level, each individual process can be thought of as its own grouping. Hierarchical vector clocks exploit the organization of processes in order to use less memory and thus are more ideally suited for hierarchical environments than vector clocks. In a hierarchical system, each process is a logical node on its own at the lowest level. There are multiple levels each with multiple clusters representing logical nodes belonging to a set of grouped nodes. Each of these peer groups are clustered into their own logical nodes on a higher level of hierarchy.

B. Algorithm

Let L define the number of levels for a hierarchical system. In a non-hierarchical system, the number of levels $L = 1$. We can define a function, $G_k(\mathrm{P}_i)$, that maps a process to the group node which contains P_i on the k-th level of hierarchy.

An important concept for using a hierarchical vector clock is the idea of hierarchical distance, $H(\mathrm{P}_i, \mathrm{P}_j)$ between two processes. The distance is equivalent to the lowest level of the hierarchy in which both processes appear in the same peer group. By convention, $H(\mathrm{P}_i, \mathrm{P}_i) = 0$.

The overall size of a hierarchical vector clock for process P_i is the summation of the cardinality of each group on each level of the hierarchy that the process belongs to. Formally, this would be:

$$N = \textstyle\sum_{k=1}^{L} ||G_k(\mathrm{P}_i)||$$

Where k is the hierarchical level, L is the number of levels, and $G_k(\mathrm{P}_i)$ is the function defined previous. Hierarchical vector clock size varies for each process depending on the size of the groups the process belongs to at each hierarchical level. In total, the process's clock value may be described as

$$v_i = [v_1^i, v_2^i, v_3^i, \cdots, v_L^i]$$

Similar to a vector clock, all entries in a hierarchical vector clock are initially set to zero and the local component of the lowest level vector is incremented for internal events that occur.

$$v_1^i[p_i] = v_1^i[p_i] + 1$$

Processes that communicate with each other know the addresses of each other. When process P_i sends a message to P_j, a timestamp is attached to the message. Depending on the hierarchical distance between the two processes, the contents and size of the timestamp varies. When the hierarchical distance, r, is 1 (within the same group on the lowest level), the timestamp s is identical to P_i's local hierarchical vector clock.

$$s = v_1^i, v_2^i, \ldots v_L^i$$

But if $1 < r <= L$, then the timestamp s consists of only the rth vector up to the maximal vector.

$$s = v_r^i, v_{r+1}^i, \ldots v_L^i$$

v_r^i is generated by "substituting level r logical time component by local component of level 1" [3] at the index of the process's ancestor group at that level, represented by g_r^i.

$$v_r^i[g_r^i] = v_1^i[p_i]$$

The message timestamp decreases as the hierarchical distance between two communicating processes increases. When process P$_j$ receives the message timestamp $s = (s_0 s_1 ... s_m)$ from P$_i$, where m = L - $H(P_i, P_j)$. Process P$_j$ updates its own clock by doing a component wise maximum between corresponding vectors. The component wise maximum step can be stated formally as

$$v_k^j := \max(v_k^j, s_{k-r})$$

The receive event timestamp is recorded with the new value of P$_j$'s hierarchical vector clock. The local component of the lowest level vector of the sender process is always included in the timestamp for each message. Recording the index of the most recent event in a corresponding process, the lowest level vector is essentially a vector clock for the lowest level hierarchical group. Furthermore, the vector components at higher levels record events from nodes between processes of the same group at the corresponding hierarchical level.

C. Implementation

https://github.com/rithviknimma/Hierarchical-Vector-Clock

The implementation in Java simulates a computation of processes and events running the hierarchical vector clock. The Process class models the behavior of different processes on different systems and contains three event related methods, $Send()$, $Receive()$, and $InternalEvent()$. The $Send()$ method is called with the destination process. Inside the $Send()$ method, the hierarchical distance between the sender process and the receiver process is calculated in order to build the timestamp that is sent with the message. The message object is returned from the $Send()$ method of the sender process, and then passed to the $Receive()$ method of the receiver process. The receiver process performs the comparison between the message timestamp and its own hierarchical vector clock and takes the maximum value component wise as described in the receive algorithm. The $InternalEvent()$ increments the process' local clock value. With these three core methods, the implementation can simulate a computation of events and messages among processes. Process objects belong to ProcessorGroup objects, which represent the hierarchical structure.

Each Process is represented by a ProcessorGroup object and has a parent ProcessorGroup object. ProcessorGroup objects have parent ProcessorGroup objects, and so on up to the highest level of the hierarchy. To simulate a computation, the test suite can initialize any desired number of processes and describe the hierarchical structure using a two dimensional array. Each row in the array represents a level of the hierarchy, and each integer in the row corresponds to how many ProcessorGroups are children of the current ProcessorGroup node. With an initialized hierarchical system, the test can direct process P_i to send a message to P_j, and direct P_j to receive that message. The implementation can also check if event e happened before event f by using the hierarchical vector clock timestamp at the respective events and comparing them according to the causal precedence detection algorithm.

IV. Complexity Analysis

A. Happened Before Relation

In order to determine if event e happened before event f, $e \to f$, the process is different from determining causality with a vector clock. First, compare the local Lamport clock values for the two timestamps. If equivalent, then $e \mid\mid f$. If not equivalent, must check the following values after computing the hierarchical distance r [3].

```
**for all** $k$ in range $L$ to $r$ **do**
  **if** $v_k^f < v_k^e$ **then**
    $e \not\to f$
  **else if** $v_k^e < v_k^f$ and $v_1^e \leq v_k^f[g_k^i]$ **then**
    $e \to f$
  **end if**
**end for**
$e \mid\mid f$ or $f \to e$
```

B. Time Complexity

The time complexity of building a message timestamp using this hierarchical vector clock implementation is greater than the time complexity using a regular vector clock. In order to build the timestamp for a message using a hierarchical vector clock, the hierarchical distance between the sender and receiver processes must be known. The hierarchical distance can be calculated in $O(\log n)$ time, by iterating up the hierarchical structure until a shared parent group is found. Once the hierarchical distance, r, is calculated, the process can build the timestamp in $O(1)$ time, by selecting the last r rows and replacing the appropriate index of the first row with its local clock value. The time complexity of building a message timestamp using a vector clock is trivially $O(1)$, as the sender process sends its entire vector to the receiver process. The time complexity of receiving a hierarchical vector clock message timestamp is smaller than that of receiving a vector clock timestamp. In both systems, the receiving process performs a component wise maximum. With a hierarchical vector system, there are only $(L - r + 1) * \log n$ values to compare, where L is the number of levels in the hierarchy and r is the hierarchical distance, whereas a vector clock system requires n comparisons. In most hierarchical structures that are relatively balanced, r is of the order $\log n$, so the time complexity of receiving a hierarchical vector clock timestamp can be thought of as roughly $O(\log^2 n)$. This is smaller than the $O(n)$ complexity of receiving a vector clock timestamp. With a logarithmic time complexity, hierarchical vector clocks show similar performance to vector clocks.

C. Memory Complexity

Using a hierarchical vector clock, processes are required to store L arrays of size $\log n$, where L represents the number of levels in the hierarchy. Assuming the hierarchy is relatively balanced, L will be on the order $\log n$. Therefore, the memory complexity for a process using hierarchical vector clock is $O(\log^2 n)$ in most cases. With a vector clock, processes must store an array of size n, therefore the memory complexity in that case is $O(n)$. With a hierarchical vector clock, there is a slight improvement on the vector clock in terms of memory performance.

D. Message Complexity

In distributed systems, message complexity is a more important factor of overall performance than processor runtime and memory usage. In this regard, a computation that uses a hierarchical vector clock has smaller message timestamps compared to a computation that uses a vector clock, hierarchical vector clocks have a superior performance over vector clocks. The two computations send the same number of messages, one per send event. However, the message size is smaller when using a hierarchical vector clock than when using a vector clock. The hierarchical vector clock timestamp size is $(L-r+1)*\log n$ elements, where L is the number of levels in the hierarchy, r is the hierarchical distance between the sender and receiver processes, and n is the number of processes. A vector clock timestamp is always n. The greatest advantage of using the hierarchical vector clock instead of the vector clock is the decrease in the size of message timestamps.

V. Conclusion

This paper presented and reviewed Khotimsky's hierarchical vector clock, a logical time system algorithm that improves scalability and message overhead that persisted with prior solutions, while maintaining accuracy for establishing causal knowledge and partial ordering in distributed systems. A computation that uses hierarchical vector clocks to maintain logical time was implemented in a simulated distributed system in a Java Runtime Environment. With the program, a hierarchy can be built and messages can be sent between processes. The separate processes use hierarchical vector clocks to maintain logical time of events. In addition, the program can determine if an event happened before another event based on the respective hierarchical vector clock timestamps of the events.

References

[1] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. 2nd ed. Cambridge University Press, 2002. DOI: 10.1017/CBO9780511809088.

[2] Colin J Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering,"". In: *Proc. 11th Austral. Comput. Sci. Conf. (ACSC '88)*. 1988, pp. 56–66.

[3] D. A. Khotimsky and I. A. Zhuklinets. "Hierarchical vector clock: scalable plausible clock for detecting causality in large distributed systems". In: *1999 2nd International Conference on ATM. ICATM'99 (Cat. No.99EX284)*. 1999, pp. 156–163. DOI: 10.1109/ICATM.1999.786798.

[4] Michel Raynal. "Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems." In: *IEEE Distributed Systems Online* 3 (Jan. 2002).