Research on Singhal-Kshemkalyani's implementation of vector clocks and previous project

Xinyu Chang

Computer Science Department

Kent State University

Kent, OH, 44242

xchang@kent.edu

**Abstract**

Singhal-Kshemkalyani's algorithm is an efficient way of implement vector clock [3]. It is a optimization of the original vector clock algorithm. In this project, it is implemented in a simulated engine, and 3 experiments were done on it. The experiments includes simulated engine time complexity analyze, simulated engine message complexity measurement, and comparison between message transmission load of original vector clock algorithm and Singhal-Kshemkalyani 's algorithm.

**1 Introduction**

**1.1 Singhal-Kshemkalyani’s algorithm**

In the original algorithm of vector clock, assume there are *N* processes in a computation, there is a problem that each process has to store latest VC sent to each receiver [1], which made it **O(N2)**. However in SK’s Implementation of VC, only elements changed will be sent in the form of entry. The format of an entry is like ***(Process ID, time stamp).*** Figure 1 shows a example of vector clock maintained by each process, *T* denotes a local timestamp, *LU* is latest update time stamp, and *LS* istime stamp of latest sent, an entry is sent only when Processi.***LU >*** Processi.***LS, i*** *is the index of process* where ***0 < i < N****.* Let the index of receiver be ***j,*** and the process ID contained by the entry be ***k***, the timestamp carried by the entry will be used in update by the receiver only when ***Entry.T >* Processk*.T***, when this condition is fulfilled, ***Processk.T*** will be updated by ***Entry.T*** and ***Processk.LU*** will be updated by ***Processj.LU.***



Figure 1: An example of SK’s vector clock

The rule of update vector clock for message sending process ***i*** and message receiving process ***j*** is as below.

**For sender:**

1. Add 1 to ***Processi.T*;**
2. Add 1 to ***Processi.LU***;
3. ***Processj.LS = Processi.LU***;

**For receiver:**

1. Add 1 to ***Processj.T***;
2. Add 1 to ***Processj.LU***;

**1.2 Simulated engine**

The simulated engine is coded in *Java,* the SK vector clock was implemented in a flooding algorithm, in a flooding algorithm, which has applied an all-connected topology, all the processes share one FIFO channel, each process has only one chance to send messages to random set of neighbors, the initiator start the flood by sending the first set of messages, other non-initiator processes could be executed only when there is a message for them in the channel. When multiple processes are enabled, the simulated engine will randomly pick up one to be executed. The flooding algorithm ends when no message is remain in the channel. Figure 2 shows the Pseudo code of the flooding algorithm. *Process 0* in Figure 2 is the initiator, and *process I* denotes the non-initiator processes.



Figure 2: Pseudo code of the flooding algorithm

**2 Experiments**

**2.1 experiment setup**

The number of process ***N*** is designated as the variable parameter, and this parameter was varied from 5 to 60 with 5-process increment. By every time of execution the data is obtained and analyzed.

**2.2 Simulated engine time complexity analyzes and optimize**

According to the time complexity theory [2], If ***f(n)*** is a polynomial of degree ***d*** (that is, ***f(n) = a0 + a1n + adnd***), then ***f(n)*** is ***O(nd)***, so we only need to find out the part have highest degree in the computation. As it is mentioned in the section 1.2, the simulated engine will select a process to execute from the enabled processes randomly, For the worst case, every process send N messages, let ***t*** denotes the number of times of execution, every time a process sent a set of messages, it will receive one message from the channel, so the amount of messages remains in the channel for the worst case should be ***tN – t****,* and due to the feature of each process sent at most one time, after the first message sending the message remain in the channel is ***(N-1),*** and for the last time the amount of message remain in the channel should be ***N2 – N,*** so the total time cost should be ***(N-1) + (2N–2) + …(N2 – N) = (N(N-1)2) /2*** ***= O(N3)***, and channel should be checked for every process, so the time complexity for the whole process should be ***O(N4)*** .

 Figure 3 show the execution result of time complexity, according to the graph, we can draw the conclusion that the time complexity of this algorithm is ***O (N4)***, which is consist with the prediction in the previous paragraph.

****

Figure 3: time complexity data

**2.3 Message complexity**

In this section, I will introduce the experiment on message complexity. As it is mentioned in section 1.2, each process has only one chance to send messages to random set of neighbors, therefore, for the worst case, ***N*** processes will sent ***N\*N*** messages, so the message complexity should be ***O(N2).***

Figure 4 shows the execution result of message complexity, according to the graph, we can draw the conclusion that the time complexity of this algorithm is ***O (N2)***, which is consist with the analyze in the previous paragraph.



Figure 4: message complexity data

**2.4 Entry measurement**

 The Figure 4 also shows the number of entries (yellow line), which increases according to the variation of process number, this will be used in later section 2.5. Experiment setup in section 2.1 contains two variable parameters, the number of process and the number of messages, both of which will affect on the number of entry. But here I like to do research on entry increment affected by single variable, therefore, the experiment setup is modified, the number of process was fixed to 10, then the number of messages has been varied from 0 to 600 with 50-message increment.



Figure 5: entry number according to message number

 As it is shown in Figure 5, the growth of entry number is linear. This result shows that, the increment rate of the entry in the computation is in the same level with the message number increment rate, which means that even in a computation with large number of message transmission, entry will not have big impact on efficiency of the computation.

**2.5 Comparison**

In the previous vector clock algorithm is inefficient because the size of a timestamp is proportional to the number of processes in the system [3]. Let one set of process ID and timestamp be one message unit, for example an entry is one message unit, and a complete vector clock is ***N*** message unit. Message unit will be used in the transmission load comparison. This experiment is done under the experiment setup in section 2.1.



Figure 6: transmission load comparison

**Conclusion** According to the experiment data in Figure 6, the message load increment generated by original vector clock is far more significant than SK vector clock. So I can draw the same conclusion of SK algorithm is an efficient way of maintain Mattern/Fidge vector clocks, which cuts down the communication overhead due to propagation of vector timestamps by sending only incremental changes in the timestamps[3].

**3 Future Work**

In this report I analyzed and researched on the time complexity of the simulated engine. However the result of the experiment is not optimistical, the efficiency for the whole system still needs to be improved. For the single send operation cost a lot of time, I think the future work on this should focus on improving the efficiency of sending operation.

**References**

1. [deneb.cs.kent.edu/~mikhail/classes/aos/](http://deneb.cs.kent.edu/~mikhail/classes/aos/) l04logicalclocks.ppt.
2. Michael T. Goodrich, Roberto Tamassia, Algorithm Design Foundation, Analyze, and Internet Examples, pages 13-23, 2001.
3. Mukesh Singhal, Ajay Kshemkalyani, An efficient implementation of vector clocks.
4. Lamport, L., Time, clocks, and the ordering of events in a distributed system, 1978.