

CS5320 – Assignment #1

Implementing Vector Clocks

By Harsh Agarwal
cs15btech11019

Implementation Details

Each node is represented by a thread.

Every node creates another thread for receiving messages at a port number. Every node executes internal & send events randomly.

All nodes are running on different port numbers on localhost.

The port number for every node identified using
(*PORT_STARTING_RANGE + nodeID*).

vectorClock array is shared among the internal, send & receive events.

VC: For every send event, the entire vectorClock is sent as a string to the receiver.

SK: For every send event, (index,vectorClock-value) is sent as a string to the receiver.

The receiver tokenizes the received data and performs the appropriate modifications to vectorClock.

Consistency Property

```
harsh@harsh-inspiron-5558:~/btech/sem-7/Distributed Computing/assignment...  harsh@harsh-inspiron-5558:~/btech/sem-7/Distributed Computing/assignment...  harsh@harsh-inspiron-5558:~/btech/sem-7/Distributed Computing/assignment...
Process#5 sends message m12 to Process#3 at 1002, vc: [ 0 4 0 2 0 5 0 0 2 0 0 ]
Process#8 sends message m29 to Process#9 at 1002, vc: [ 0 4 0 0 0 0 0 0 4 0 0 ]
Process#1 receives message m18 from Process#9 at 1003, vc: [ 0 5 1 2 0 4 0 0 2 4 0 ]
harsh@harsh-Inspiron-5558:~/btech/sem-7/Distributed Computing/assignments/Implementing Vector Clocks$
harsh@harsh-Inspiron-5558:~/btech/sem-7/Distributed Computing/assignments/Implementing Vector Clocks$
head -n 50 output.txt
Process#0 executes internal event e1 at 1000, vc: [ 1 0 0 0 0 0 0 0 0 0 0 ]
Process#0 executes internal event e2 at 1000, vc: [ 2 0 0 0 0 0 0 0 0 0 0 ]
Process#0 executes internal event e3 at 1000, vc: [ 3 0 0 0 0 0 0 0 0 0 0 ]
Process#0 executes internal event e4 at 1000, vc: [ 4 0 0 0 0 0 0 0 0 0 0 ]
Process#1 executes internal event e1 at 1000, vc: [ 0 1 0 0 0 0 0 0 0 0 0 ]
Process#4 executes internal event e1 at 1000, vc: [ 0 0 0 0 1 0 0 0 0 0 0 ]
Process#1 sends message m21 to Process#4 at 1000, vc: [ 0 2 0 0 0 0 0 0 0 0 0 ]
Process#2 sends message m31 to Process#9 at 1000, vc: [ 0 0 1 0 0 0 0 0 0 0 0 ]
Process#3 sends message m94 to Process#10 at 1000, vc: [ 0 0 0 1 0 0 0 0 0 0 0 ]
Process#1 sends message m43 to Process#4 at 1001, vc: [ 0 3 0 0 0 0 0 0 0 0 0 ]
Process#4 receives message m21 from Process#1 at 1000, vc: [ 0 2 0 0 2 0 0 0 0 0 0 ]
Process#5 executes internal event e1 at 1000, vc: [ 0 0 0 0 0 1 0 0 0 0 0 ]
Process#3 sends message m74 to Process#5 at 1000, vc: [ 0 0 0 2 0 0 0 0 0 0 0 ]
Process#2 sends message m93 to Process#4 at 1001, vc: [ 0 0 2 0 0 0 0 0 0 0 0 ]
Process#6 executes internal event e1 at 1000, vc: [ 0 0 0 0 0 0 1 0 0 0 0 ]
Process#4 receives message m43 from Process#1 at 1001, vc: [ 0 3 0 0 3 0 0 0 0 0 0 ]
Process#4 sends message m57 to Process#7 at 1001, vc: [ 0 3 0 0 4 0 0 0 0 0 0 ]
Process#10 receives message m94 from Process#3 at 1000, vc: [ 0 0 0 1 0 0 0 0 0 0 1 ]
Process#3 sends message m24 to Process#10 at 1001, vc: [ 0 0 0 3 0 0 0 0 0 0 0 ]
Process#5 receives message m74 from Process#3 at 1000, vc: [ 0 0 0 2 0 2 0 0 0 0 0 ]
Process#1 sends message m99 to Process#8 at 1001, vc: [ 0 4 0 0 0 0 0 0 0 0 0 ]
Process#7 receives message m57 from Process#4 at 1000, vc: [ 0 3 0 0 4 0 0 1 0 0 0 ]
```

The red boxes events have a causal relationship.
This is also evident from the vector clocks.

Graphs Section

Topology #1 - Directed Complete Graph

Lambda - 2

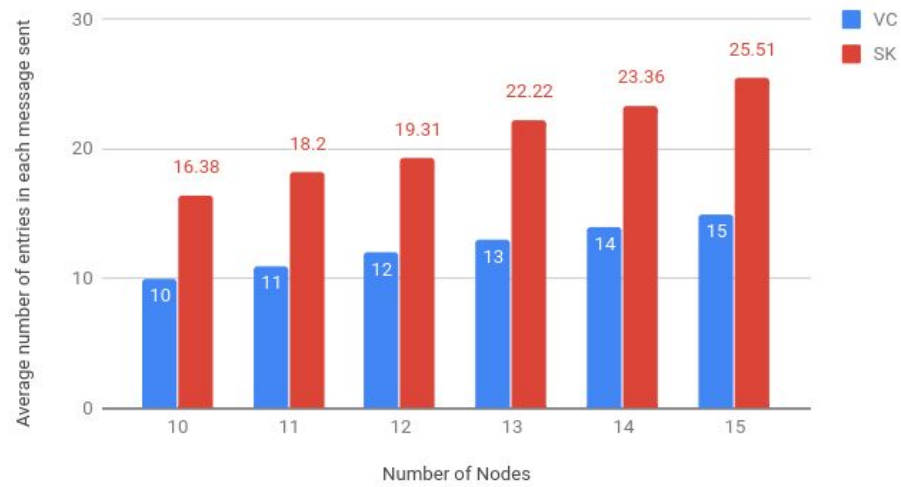
Alpha - 1.5

M - 50

READINGS

Threads	Vector Clock	Singhal-Kshemkalyani
10	10	16.38
11	11	18.20
12	12	19.31
13	13	22.22
14	14	23.36
15	15	25.51

Vector-Clocks vs Singhal-Kshemkalyani



Topology #2 - Directed Linear Graph

Ex: for 4 nodes

Connection is as

1 2

2 3

3 4

4 1

Lambda - 2

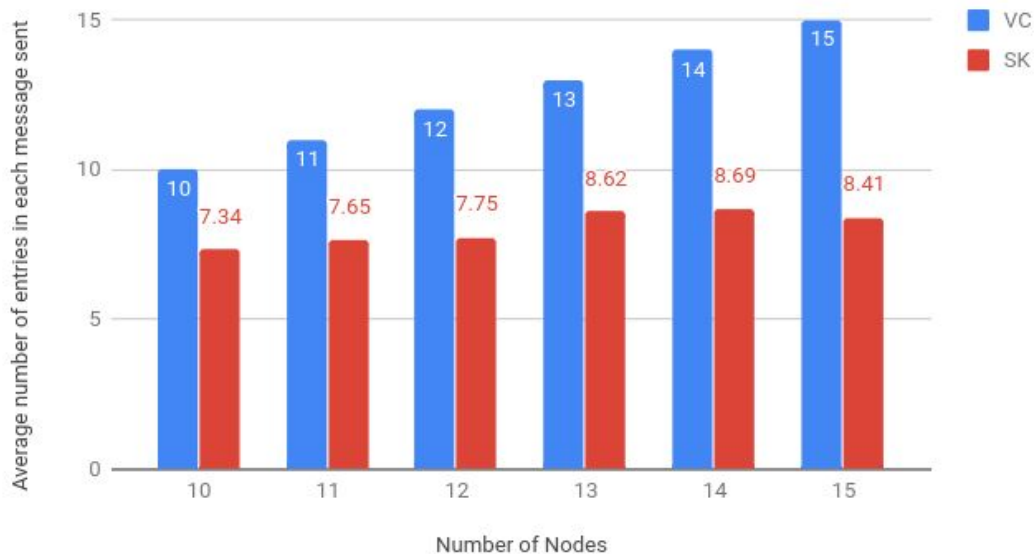
Alpha - 1.5

M - 50

READINGS

Threads	Vector Clock	Singhal-Kshemkalyani
10	10	7.34
11	11	7.65
12	12	7.75
13	13	8.62
14	14	8.69
15	15	8.41

Vector-Clocks vs Singhal-Kshemkalyani



OBSERVATIONS

- According to [*Distributed Computing Principles, Algorithms & Systems* by Ajay D. Kshemkalyani & Mukesh Singhal](#)

Singhal-Kshemkalyani differential technique [25] is based on the observation that between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change. This is more likely when the number of processes is large because only a few of them will interact frequently by passing messages

- For the complete graph, SK is performing poorly than VC. It is taking nearly double the space used by VC. This is because in the complete graph, there are interactions between every 2 processes in the system. Therefore in almost every send event, almost N (number of nodes) vectorClock entries are sent across the channel. Since the index is also piggybacked with the vectorClock value, the message size is almost double than that of VC.

Hence VC is a better algorithm for this topology.

- For the linear graph, VC is performing poorly than SK.
This is because in this graph, a process is interacting with just 2 other processes in the system.
Therefore in almost every send event $< N$ (number of nodes) vectorClock entries are sent across the channel.
Hence in this case, the extra overhead of the index is not slowing down the performance.

Hence SK is a better algorithm for this topology.

- In case #1, the graph is increasing exponentially for SK.
This is because on adding a new node in the topology, an extra neighbor is being added to every other node.
VC is always equal to the number of nodes in the topology.
- In case #2, the graph is almost a straight line for SK.
This is because on adding a new node in the topology, the number of neighbors for every node remains 2. VC is always equal to the number of nodes in the topology.