Download from the ACM Digital Library and read (you may skip Section 3):
Kshemkalyani, A.D., Khokhar, A.A., Shen, M.: **Encoded vector clock: Using primes to characterize causality in distributed systems**. In: Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018, pp. 12:1–12:8 (2018). DOI 10.1145/3154273.3154305. URL http://doi.acm.org/10.1145/3154273.3154305

Experiment Part 1

AIM
A goal of the project is to identify how fast the EVC grows, as a function of the number of events executed by a process, and the number of events executed by all the processes collectively. n is an input parameter. With n processes and assuming 32-bit integer, how many events it takes for the size of the EVC to occupy a number equal to 32n bits long. Once it equals 32n bits long, we can do a system-wide EVC reset.

A POSSIBLE APPROACH
One way to measure this is to simulate asynchronous message-passing among n processes. Generate the first n prime numbers in Primes[1,n] and assign one to each process being simulated. Each process is simulated by a thread and the events of each process (internal, send, and receive events) are also simulated. A process generates internal and send events with a certain probability (a controllable parameter, which can also disallow internal events) and at a certain frequency (rate), say 1 event/ms. The events get queued in the process queue Q_i along with the simulation time timestamp, which is processed by the simulating thread. If it is a send event, its destination P_j is chosen at random from among the other n-1 processes. A corresponding receive event, (along with the sender's EVC timestamp) and along with the simulation time timestamp is enqueued in Q_j and processed (perform EVC operations for a receive event) by the thread simulating P_j. The simulation time timestamp of a receive event, as chosen by the sender, can be set to the sum of the send event simulation time timestamp plus a uniformly chosen value between, say, 0ms and 10ms.

The queue Q_i determines the schedule of events occurring at process P_i. The thread simulating process P_i dequeues events in simulation timestamp order, and simulates the EVC value update for that event. (That is, if an internal or send event, it does a multiply by the prime number associated with that process; if a receive event, it calculates the LCM, and then multiply, etc. etc. as per the EVC rules). In essence, you have to ensure fair scheduling; for example, you can implement round-robin execution among the threads.

In addition, each thread P_i maintains a count of the number of simulated events it has processed after dequeueing from the local queue Q_i. For debugging purposes, it is useful to maintain 3 arrays: Send_simulated[1,n], Receive_simulated[1,n], and Internal_simulated[1,n] to count the corresponding number of send/receive/internal events dequeued and processed from each Q_i. In addition to the required EVC, it is useful to maintain the (traditional) vector clock VC of the latest event simulated by each thread – this will be particularly useful for Part 2. (The sum of all the components of the vector clock at a thread gives the total number of events executed in the causal past in the system.)

A key challenge in the simulation is how to represent and store and manipulate (multiply, divide, and LCM or GCD calculation, etc.) EVCs that are hundreds or thousands of bits long, or "biginteger". Check for example,
https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic

https://en.wikipedia.org/wiki/List_of_arbitrary-precision_arithmetic_software
You may, if you wish, choose to use one of the libraries for arbitrary-precision arithmetic (multiplication and division), and any appropriate programming language. For GCD calculation, you can implement Euclid's recursive algorithm.

DELIVERABLES

- For selected values of *n* (e.g., 10, 40, 100), in separate graphs each, plot the size of EVC in bits as a function of the number of events executed in the system (until the size of the EVC reaches 32n, i.e., overflow occurs earliest, at some one process).

- Also, plot the number of events executed at a process (until EVC overflow (i.e., reaching size 32n) occurs earliest at some process), as a function of *n*.

- Now vary the mix percentage of internal events (baseline case is with no internal events) and communication events. On the X-axis, vary the percentage of internal events, from 0 to 99, say. On the Y-axis, show the number of events in the system until the EVC size overflows (reaches size 32n earliest) at some one process. In the graph, for different values of *n*, say, 10, 20, 40, 60, 80, 100, you will have different curves.

You may plot not just graphs, but other forms of charts such as bar charts and histograms and pie charts, as you think are useful. If necessary, you may also tabulate data.

Analyze and explain the trends and observations you make about your data in all the above cases. Document all the design choices you made in the project, and how you implemented the main procedures.

Submit a detailed project report (hard-copy and soft copy), typeset preferably using Latex. It is recommended that your plots be in Gnuplot or some other professional tool (not Microsoft).

Note: The scientific approach to taking readings in simulations/experiments is as follows. For each setting of the parameters, take the readings for several runs (for example, 5 or 10 runs) and report the average. If there is noticeable variation in the readings (for the identical setting of the parameters), in addition to the average, report the standard deviation also for each setting of the parameters.

Experiment Part 2

AIM
To understand the benefits and limitations of storing and transmitting the logarithms of the EVCs (instead of the EVCs themselves). Due to finite-precision arithmetic, round-off errors may get introduced. Such errors may cause inaccuracies in the comparison test between the EVCs of two events. (Recall the comparison test, for events e and f. e → f if essentially EVC(f) mod EVC(e) = 0; now implement this using logarithms as shown in the paper, Section 4.4.)
An important part of understanding the limitations is being able to quantify the level of accuracy (or its complement, the error rate introduced) in the use of logarithms and anti-logarithms. For each pair of events e and f in the execution simulation, where e occurs at P_i and f occurs at P_j, determine e→f using traditional vector clocks (using 1 comparison: V(e_i)[i] <= V(f_j)[i]) and also using the logarithms of EVCs as shown in **Section 4.4** of the paper. Specifically, we have 4 possible cases for each pair of events:

1. e→f and EVC(f) mod EVC(e) = 0. (true positive)
2. e→f and NOT(EVC(f) mod EVC(e) = 0). (false negative)
3. NOT(e→f) and NOT (EVC(f) mod EVC(e) =0). (true negative)
4. NOT(e→f) and EVC(f) mod EVC(e) = 0. (false positive)

When we pick a pair of events e and f, the log-EVC of either event may be greater. So there is one of 3 relations between e and f: (i) e→f, (ii) f→e, (iii) Not(e→f) AND Not(f→e). Comparing the test using vector clocks with the test using log-EVCs, we then have:

1. (true positive) (e→f AND loginv(y-x) in N) OR (f→e AND loginv(x-y) in N)
2. (false negative) (e→f AND loginv(y-x) not in N) OR (f→e AND loginv(x-y) not in N)
3. (true negative) (NOT(e→f) AND loginv(y-x) not in N) AND (NOT(f→e) AND loginv(x-y) not in N)
4. (false positive) (NOT(e→f) AND NOT(f→e) AND (loginv(y-x) in N OR loginv(x-y) in N)

A POSSIBLE APPROACH
You will need to use a multi-precision binary floating point library.
Generate distributed executions for *n* = 10, 20, 40, say, containing up to, say (for example) *v***n* events totally, where *v*= 50, 100, 200, 500, for example.
Store the vector clock, for each event. Also, for each event, store the logarithm of its EVC as generated by the **algorithm of Figure 3**. In order to store the logarithm, assume a bounded mantissa *m* (the digits after the decimal point). For example, assume *m*=16 bits, 32 bits, 64 bits, 128 bits, 256 bits, 512 bits, etc.

There are (*v***n*)(*v***n* – 1)/2 pairs of events in the total of *v***n* events. For each event pair, determine whether e → f using the vector timestamp (or EVC, if you are storing EVCs instead; assume unbounded storage for EVCs in this case) test, and then using the logarithms of the EVC timestamps of e and f. Tabulate the number of true positives, true negatives, false positives, and false negatives for various combinations of *n* and *m* and *v*.
Calculate the rate of errors for false positives and for false negatives, for various combinations of *n* and *m* and *v*. The formulas are,
FN_rate = #false negatives/(#false negatives + #true positives) and
FP_rate = #false positives/(#false positives + #true negatives)

Note 1: It is sufficient to take logarithms to base 2, but if you wish, you can also experiment with other bases.

Note 2: you may need to exercise some prudence in rounding off while taking the anti-log in the comparison test. For example, assume *m*=4. If the anti-log gives 12.1111, or 13.0001, you may want to consider it as 13 which is a natural number. It may be unlikely that the computer will give you 13 as the exact anti-log. This is because in any execution, the EVCs timestamps are not consecutive integers but rather, spaced far off, thus it may be reasonable to consider the anti-log as 13 (a natural number). *If useful, formulate such rounding-off rules, and apply them consistently to your processing.*

DELIVERABLES
Submit a hard-copy report quantifying the errors introduced by the rounding-offs in using finite-precision mantissas in the logarithms of the EVCs.

Tabulate the number of true positives, true negatives, false positives, and false negatives (use the formulas given) for various combinations of *n* and *m* and *v*. Calculate the rate of errors for false positives and for false negatives (use the given formulas), for various combinations of *n* and *m* and *v*. Use graphs, and other forms of charts liberally. Decide what makes sense to present as results. For example, you may want to plot the percentage error rate (or degree of accuracy) of false negatives and false positives of the causality relation, for various *n* and *m* and *v*. In addition, you can also use tables to present your data about the number and percentages of false positives and false negatives.

For the settings of *n* and *m*, you may have to experiment with the values, to see which ranges give meaningful results. *v* should typically be much larger than *n* for steady-state to set in. If you run out of system resources, you may need to revise the values chosen.

Analyze and explain the trends and observations you make about your data. Document all the design choices you made in the project, and how you implemented the main procedures. Make recommendations for enhancing the algorithm pseudo-code to further improve the accuracy. (For example, what choice of *m* user should choose when using logarithms of EVCs. Or, when taking anti-logarithms, what extent of rounding-off to use.)

SUGGESTED READING (for Part 2)

Torres-Rojas, F.J., Ahamad, M.: Plausible clocks: Constant size logical clocks for distributed systems. Distributed Computing 12(4), 179–195 (1999). DOI 10.1007/s004460050065. URL https://doi.org/10.1007/s004460050065

Roh, H., Jeon, M., Seo, E., Kim, J., Lee, J.: Log' version vector: Logging version vectors concisely in dynamic replication. Inf. Process. Lett. 110(14-15), 614–620 (2010). DOI 10.1016/j.ipl.2010.04.026. URL https://doi.org/10.1016/j.ipl.2010.04.026