Encoded Vector Clocks

Implementation study

Author Bhargav Voleti

Abstract

Vector clocks are used in for keeping track of time in a distributed system. Since a global physical clock is not possible, logical clocks are used instead. Vector clocks are a form of logical clocks. The issue with vector clocks is the size of the payload being transmitted between nodes. Encoded vector clocks are an optimization wherein an encoded version of the vector clock is sent as a part of the payload. This paper talks about the implementation of a hypothetical system and shows the results of using vector clocks to transmit information regarding time in a distributed system.

Architecture

I chose Rust as the language to implement this system. According to its website the Rust programming language provides the following guarantees:

> A systems programming language that runs blazingly fast prevents segfaults, and guarantees thread safety.

This means a whole class of bugs related to multi-threaded programs are eliminated by the compiler. It also encourages message passing as a way of sharing data between threads and provides thread-safe data structures to share state between threads which is also memory safe, the safety of which is guaranteed by the compiler. All of this means that there are no data races in the system and the programmer is very productive in building a multi-threaded system. I personally found the notion of "if it compiles, it runs" very refreshing.

The system is built as an asynchronous message passing system where each thread represents a process in a distributed system. The number of threads can be configured by a configuration file. This configuration file can be used to configure a few other parameters:

- `num_processes`: Number of processes in the system
- `max_bits`: Maximum size in bits of the encoded vector clock.

- `timeout`: Timeouts for the various channels used for message passing
- `float_precision`: Size of mantissa used for arbitrary precision floating values
- `max_events`: Maximum number of events allowed.

Using these 5 values we can control different parameters of the simulation, thus gaining an understanding of the system under different conditions.

The system consists of three main types of processes

1. Processes
2. Dispatcher
3. Collector

Infomation is shared in the system using message passing. Messages are passed on FIFO channels which are thread safe and act as queues for the processes.

Process

A process is a type of thread which simulates a process in the distributed system. It keeps track of its vector clock and has a handle for the receiving end of a channel on which it receives Send messages from other processes. It also has a handle for the sending end of a channel to the collector. This is so that events can be sent to the collector to be processed.

The Process is responsible for maintaining the following clocks:

- The vector clock
- The Encoded vector clock

The process updates these clocks as a part of the steps in its operating loop. It first waits for `timeout` number of milliseconds on the receiving channel for any new send messages dispatched to it. If a timeout occurs, it then randomly selects between an Internal event or a Send event. The probability of this choice is hardcoded but can be changed pretty quickly between different runs of the experiments. For all experiments except otherwise specified the probability for an Internal vs Send event is 50:50. Once an event is generated, it then is handled. The process has a central event handler which updates the clocks according to the type of event which then dispatches to the appropriate event handler based on the type of the event to be acted upon.

If it is an Internal event, the thread sleeps for `timeout` milliseconds as a way of simulating some internal event. Once the thread wakes up, it goes back to waiting on the receiver.

If it a Send event, the thread dispatches the event to the Dispatcher thread to be sent to a random process and goes back to waiting on the receiver.

If the thread receives a message within `timeout` milliseconds, it handles the message received and generates a corresponding receive event which is then sent to the collector to be recorded.

The vector clock and the Encoded vector clocks are updated according to their rules when handling the different events generated by the system.

Dispatcher

These threads simulate the asynchronous message passing layer in the system. All processes when generating a Send event and sending a message to another process actually send the message to the dispatcher thread. This is because the number of channels required would be exponential with relation to the number of the processes if all processes directly communicated with all other processes. Having a single dispatcher thread reduces the number of channels down to a linear scale, that is the number of channels equals the number of processes in the system.

Since we are simulating a distributed system, we don't really care about which process the event is being dispatched to, as long as it is not being dispatched to the process that first created it. Therefore, the dispatcher when selecting a random process to dispatch an event to considers this rule.

We also use the destruction of the sending end of all channels held by processes as a signal that the simulation has ended. This means the dispatcher is where the end condition is monitored for, and when the dispatcher thread exits, the receiving ends of all the channels for the processes in the system have their destructors called. This signals the end of the simulation to all the threads in the system who exit.

Before the dispatcher thread exits, it sends a special End message to the collector thread to signal the end of the simulation. The collector then can start processing all the events it collected.

Collector

The collector thread is responsible for collecting all the events sent from the processes and once it gets the signal for the end of the simulation, processing all the events. The messages sent to the collector are of a different type called Messages. This is so that they can be differentiated from the events also being passed between the different threads in the system.

Using Rust's enums, a form of *algebraic data types*, we can send a message of the type Data which contains the event as the value of the type, or we can send a message of type End which signals the end. Depending on the type of the message received the collector stores events in the appropriate data structures which would make for easier processing of the data.

A shard config object is also a part of the collector, which is used to track the number of End messages to be received to signal the end of the simulation. This number is always `num_processes + 1` since all the simulation is over after all the processes and the dispatcher thread have exited.

The collector all logs various information as the simulation is being run, such as the size of the encoded vector clock vs the total number of events at a given instant. It also logs the results of processing the events after the operation has been performed.

Gathering data

All events are logged and sent to the collector to be processed. The system uses different levels of logging as a way of filtering out data. There are two different loggers configured at different levels for this purpose. A file logger handles all logs over the log level Error and a terminal logger handles all logging logs from the log level Info. This is to separate data from general system information.

The terminal logger is used to monitor the general state of the system during runs and to spot any issues with the current running simulation. The file logger is used to gather information to be processed and plotted out.

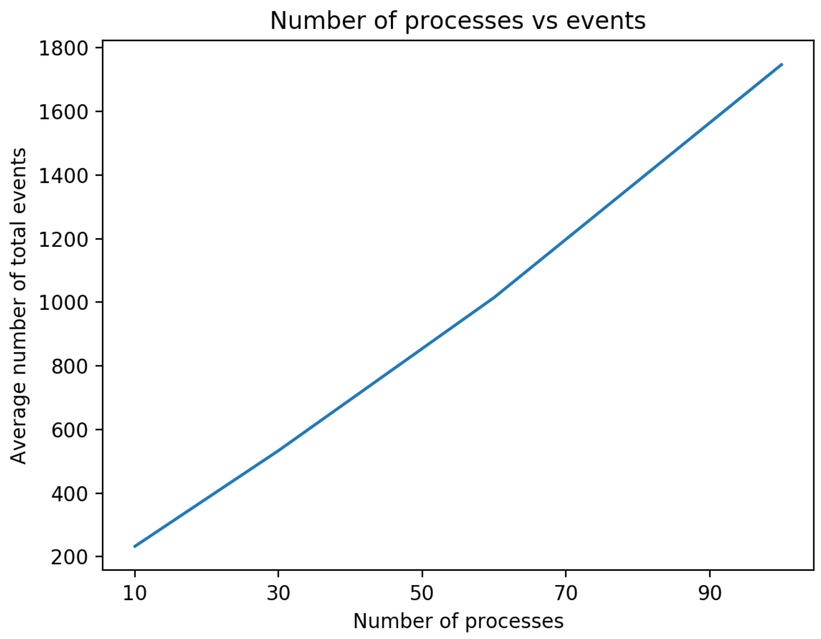
Thus using logging gives us a simple, fast and efficient way to filter data to the appropriate places.

The data is taken from the log file, cleaned and converted to CSV to be further processed by a python script. The python script also uses Matplotlib to plot the data in the appropriate graph.

Findings

All the following graphs were plotted with the mean of the data collected from ten individual runs.

Events vs number of processes

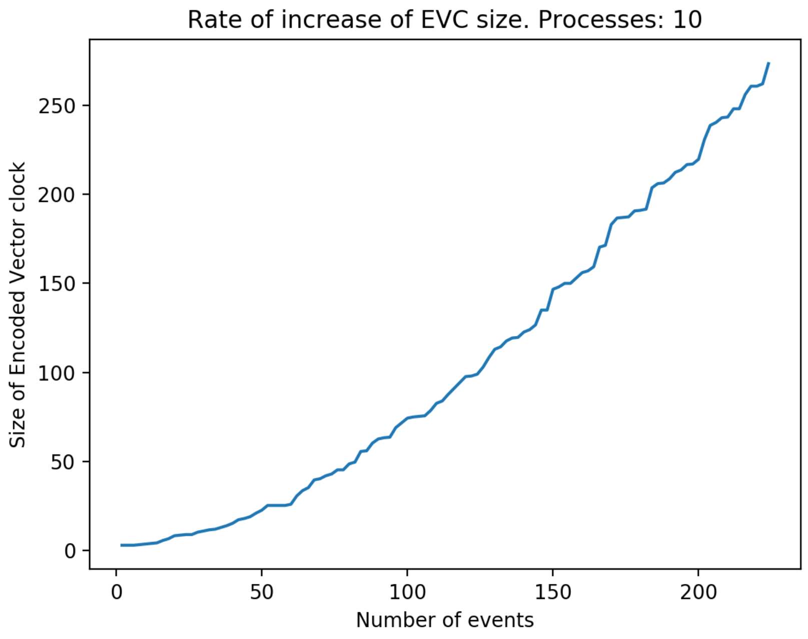


As it can be seen, the relationship between the number of processes and the total number of system events is very linear. As the number of processes in the system grows, the number of events grow with it. This is to be expected since the number of send vs internal events per process stays at the same value for a given `max_bits`(capped to $32 * n$) size of the encoded vector clock.

Size of EVC vs number of processes

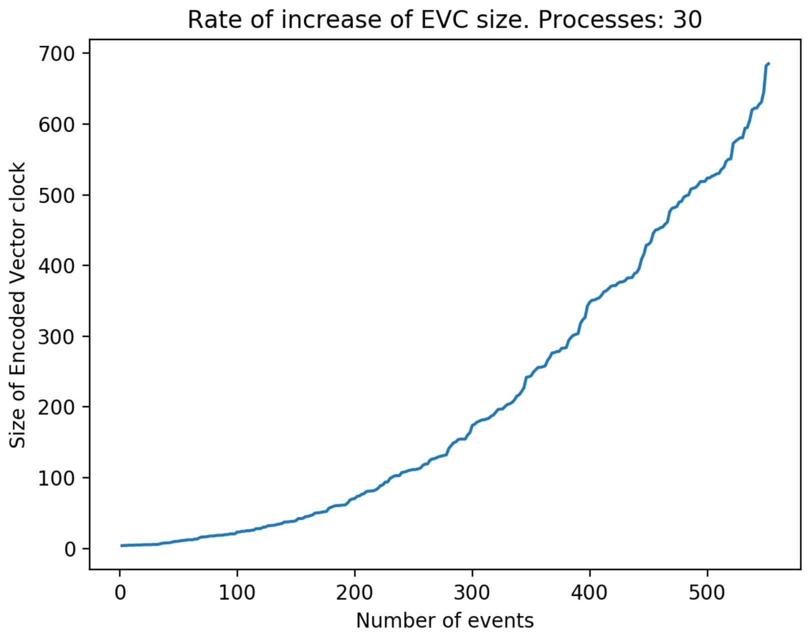
This data was gathered for a various number of processes, once the size of the EVC hit $32 * num_processes$ the simulation stopped.

10 processes



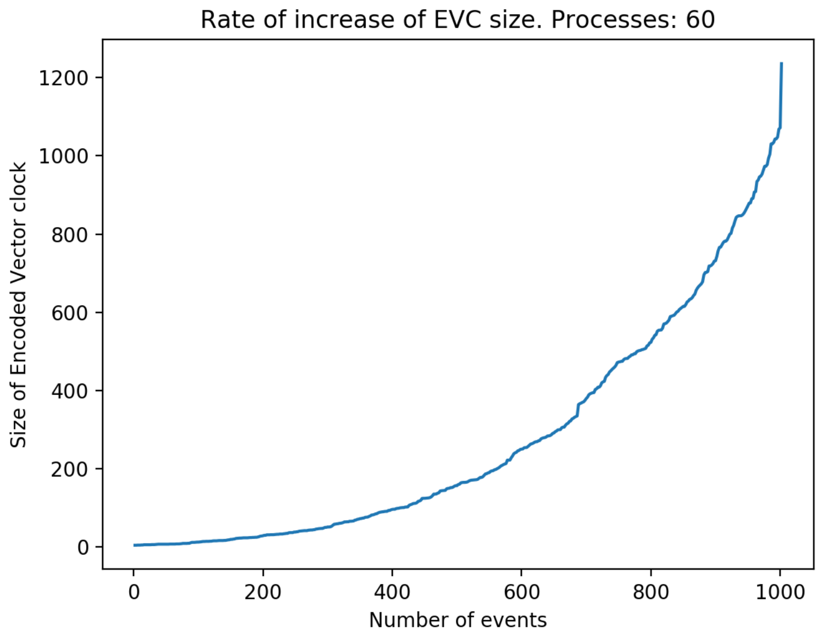
Simulation stops when size of evc becomes 320 bits.

30 processes



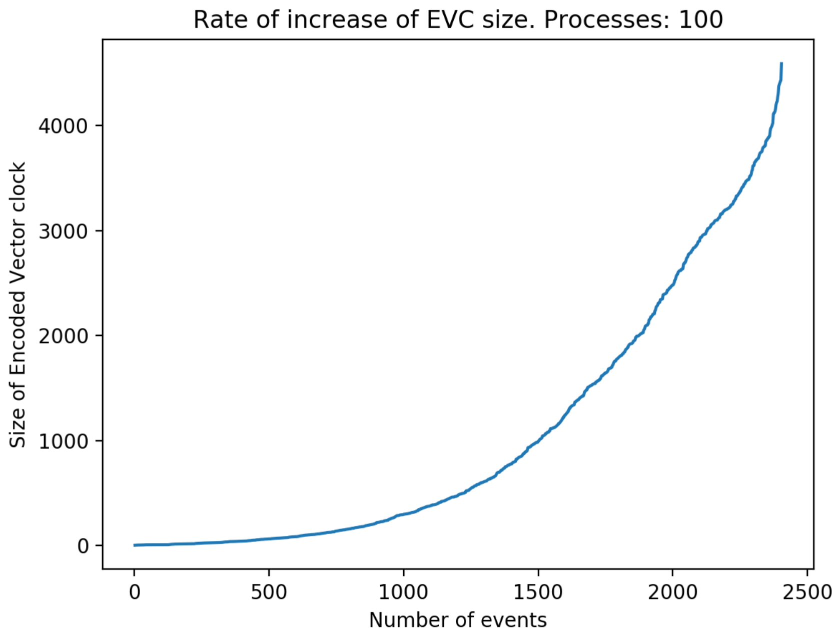
Simulation stops when size of evc becomes 960 bits.

60 processes



Simulation stops when size of evc becomes 1920 bits.

100 processes



The simulation stops when the size of EVC becomes 3200 bits.

As we can see, as the number of events in the system increases, the size of the EVC increases exponentially. In a real-world system, this would be problematic since this would mean each process's memory would see unbounded growth. To negate this, the system can be reset after the EVC hits a certain size. at 3200 bits and 100 processes, on my machine, I saw the memory consumption of my application rise up to 80 MB. Therefore depending on the amount of memory available on the machine, the correct threshold for resetting the logical clocks in the system can be evaluated.

False negatives and false positives with Logarithms

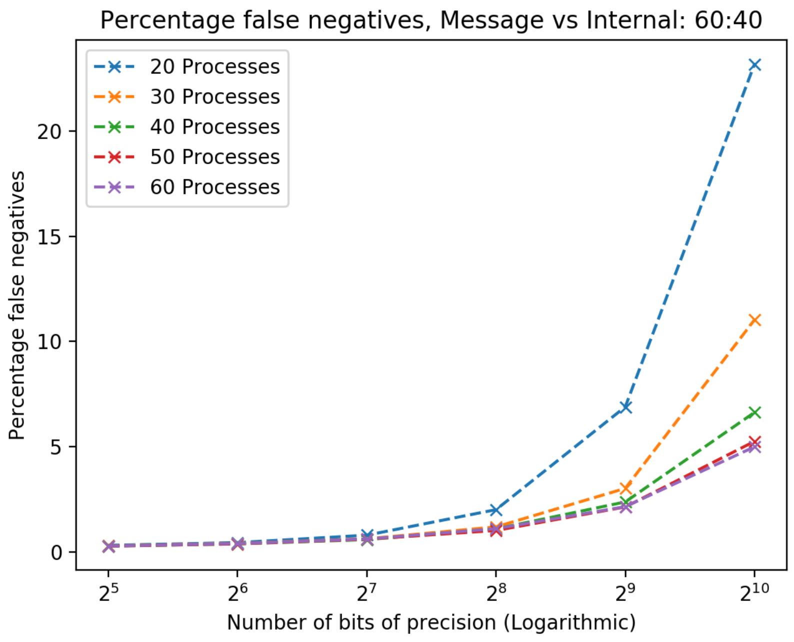
One way to put a bound on the size of the Encoded Vector Clock would be to use its logarithm instead of the encoded clock itself. The Log encoded vector clock can be hooked into the operational loop of the process without a lot of changes and the rules for updating the log clock are pretty straightforward.

The issue with Log clocks is the precision lost during the conversion between arbitrary precision Integer types and Floating point types. This conversion is required because arbitrary precision Integer types do not have a Logarithm method on them. While the Floating point type do not have numerical methods such as finding the greatest common divisor and the least common multiple, thus requiring the conversion between the two types to perform the merge operation. This leads to a lot of precion being lost which subsequently leads to a very high number of false negative causalities between different events in the system.

As of this implementation, the most stable and mature libraries for arbitrary precision operations are GMP (GNU Multiprecision Arithmetic Library) and MPFR(GNU Multiple Precision Floating-Point Reliably). While these libraries are written in C, various languages have bindings for them and the issues stated in the previous paragraph exist for these libraries as well.

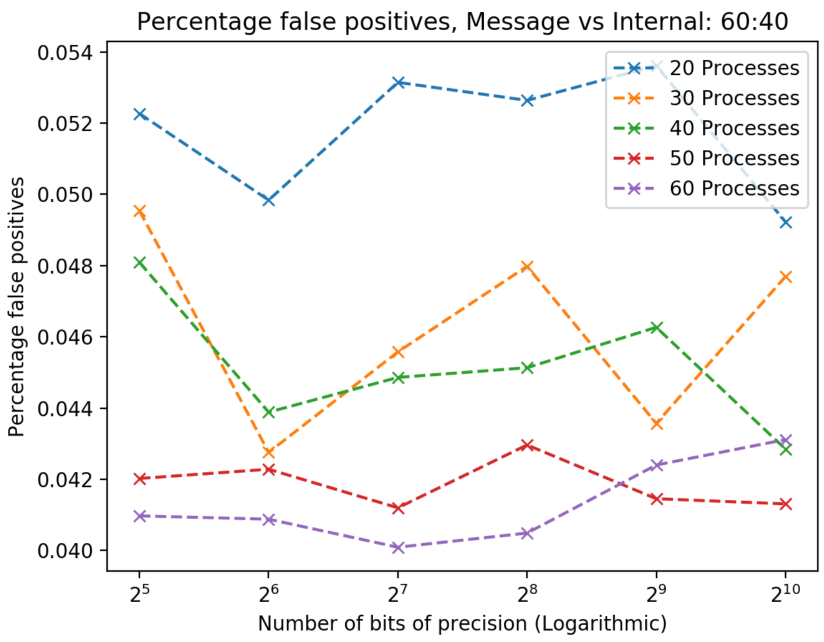
The following results are obtained after the post processnig of all the events. The Log encoded EVC is derived from the Envoded vector clock stored as a part of the events and compared.

False negatives, Send to Internal events: 60:40, total events 3000



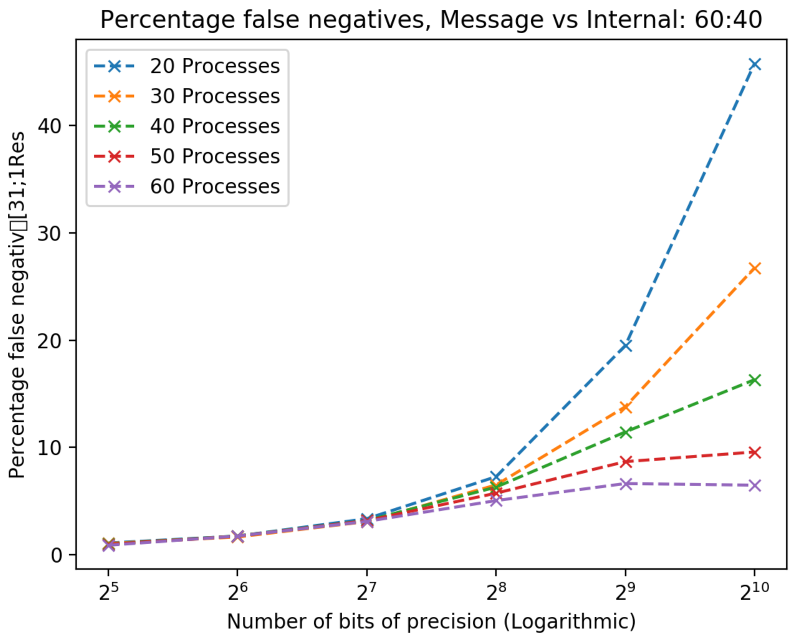
Percentage false negatives vs the the percision of the mantissa.

False positives, Send to Internal events: 60:40, total events 3000



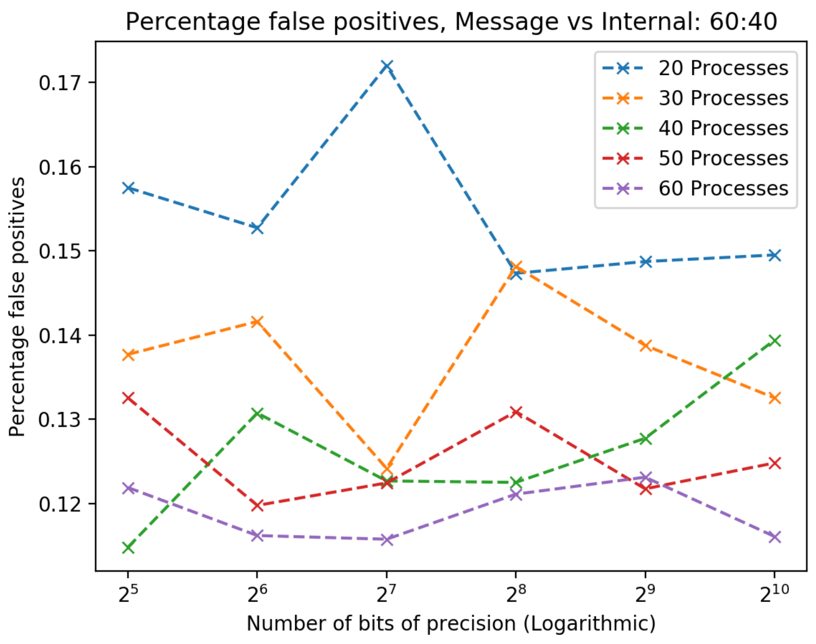
Percentage false positives vs the precision of the mantissa.

False negatives, Send to Internal events: 60:40, total events 1000



Percentage false negatives vs the the percision of the mantissa.

False positives, Send to Internal events: 60:40, total events 1000



Percentage false positives vs the precision of the mantissa.

The ratio of Send to Internal events was chosen to be representative values which could be seen in an application implementing a distributed algorithm. In such algorithms, the number of communication events are higher than the number of internal events.

The data for false negatives and false positives when tabulated can be seen in the following tables

When n: 1000

False Negatives

Processes	1024	512	256	128	64	32
20	45.72595	19.48291	7.28607	3.37226	1.75615	1.11455
30	26.71363	13.78916	6.51393	1.67361	1.07108	3.10915
40	16.31064	11.44639	6.31471	1.77016	1.00017	3.20329
50	9.58151	8.68519	5.74976	1.74742	0.99040	3.19204
60	6.49364	6.64650	5.06011	1.76219	0.90401	3.12227

False Positives

Processes	32	64	128	256	512	1024
20	0.15749	0.15273	0.17198	0.14733	0.14873	0.14951
30	0.13771	0.14161	0.12417	0.14813	0.13878	0.13254
40	0.11481	0.13074	0.12268	0.12251	0.12775	0.13940
50	0.12245	0.13254	0.11978	0.13090	0.12175	0.12482
60	0.12190	0.11620	0.11577	0.12112	0.12313	0.11609

When n: 3000

False Negatives

Processes	32	64	128	256	512	1024
20	0.31181	0.44032	0.79318	2.00681	6.89384	23.18071
30	0.62109	1.18855	0.28539	0.40510	3.01874	11.02982
40	0.27562	0.37763	0.58234	1.09621	2.37356	6.62984
50	0.27079	0.38073	0.60220	1.01646	2.13589	5.25191
60	0.27918	0.39583	0.61224	1.10376	2.14882	4.99340

False Positives

Processes	32	64	128	256	512	1024
20	0.05227	0.04984	0.05314	0.05264	0.05361	0.04923
30	0.04558	0.04954	0.04275	0.04798	0.04357	0.04769
40	0.04486	0.04513	0.04809	0.04388	0.04627	0.04285
50	0.04120	0.04202	0.04228	0.04296	0.04145	0.04131
60	0.04097	0.04088	0.04009	0.04049	0.04240	0.04311

Both tables have the mean values of 10 runs per configuration value.

Conclusion

Vector clocks are used for communicating the notion of logical time between processes in a distributed system. Due to the size of vector clocks increasing linearly with the number of processes in the system, they might not be very scalable depending on the limits of payload size. Encoded vector clocks are a consice alternative to vector clocks which reduce the payload size considerably.

They are also efficient when the system wide reset parameters are tuned well. When implementing Encoded vector clocks in a distributed system, one must ensure that the data type storing the encoded clock is ergonomic to use and provide a good abstraction to interact with the clock since the order of operations is very important in maintaining causality between events. Care must also be taken when determinig the upper bound of the size in bits of the encoded clock depending on the memory requirements of the system

Log encoded vector clocks are a futher extention of Encoded vector clocks, but when implementing Log encoded vector clocks, care must be taken when performing arithmetic operations using arbitrary precision libraries as errors might occur due to precision loss. If a certain threshold of false negative results can be tolerated in the system, Log encoded vector clocks are a great, more concise alternative to vector clocks. When implementing Log encoded vector clocks, the system wide reset must take this threshold in mind when determining the reset condition. Existing libraries for arbitrary precision arithmetic must also be evaluated to ensure all the required operations can be performed and that they are performant and efficient since they can become CPU and memory intensive quickly is care is not take.