TRACING DISTRIBUTED ALGORITHMS USING REPLAY CLOCKS

By

Ishaan Kiran Lagwankar

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science—Master of Science

2024

ABSTRACT

In this thesis, we introduce replay clocks (*RepCl*), a novel clock infrastructure that allows us to do offline analyses of distributed computations. The replay clock structure provides a methodology to *replay* a computation as it happened, with the ability to represent concurrent events effectively. It builds on the structures introduced by vector clocks (*VC*) and the Hybrid Logical Clock (*HLC*), combining their infrastructures to provide efficient replay. With such a clock, a user can replay a computation whilst considering multiple paths of executions, and check for constraint violations and properties that potential pathways could take, especially in the presence of concurrent events. Specifically, if event e must occur before f then the replay clock must ensure that e is replayed before f. On the other hand, if e and f could occur in any order, replay should not force an order between them.

After identifying the limitations of existing clocks to provide the replay primitive, we present the *RepCl* structure and identify an efficient representation for the same. We demonstrate that *RepCl* can be implemented with less than four integers for 64 processes for various system parameters if clocks are synchronized within $1ms$.

Furthermore, the overhead of *RepCl* (for computing/comparing timestamps and message size) is proportional to the size of the clock. Using simulations in a custom distributed system and NS-3, a state-of-the-art network simulator, we identify the expected overhead of *RepCl* based on the given system settings. We also identify how a user can then identify feasibility region for *RepCl*. Specifically, given the desired overhead of *RepCl*, it identifies the region where unabridged replay is possible.

Using the *RepCl*, we provide a tracer for distributed computations, that allows any computation using the *RepCl* to be replayed efficiently. The visualization allows users to analyze specific properties and constraints in an online fashion, with the ability to consider concurrent paths independently. The visualization provides per-process views and an overarching view of the whole computation based on the time recorded by the *RepCl* for each event.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest appreciation to my committee in supporting me through this work in the course of my Master's program. I am deeply indebted to Dr. Sandeep S. Kulkarni, and the ideas he has put forth that allowed me to progress in this research. The completion of this thesis would not have been possible without his support. I am deeply indebted to Dr. Li Xiao and Dr. Philip K. McKinley for providing me guidance through the committee and providing feedback for the work done in this research. I would like to extend my sincere thanks to the Department of Computer Science and Michigan State University, with gratitude to Vincent Mattison and Brenda Hodge, for supporting me with administrative decisions and financial support for the ICDCN '24 conference in which this work was first published. I would also like to thank my peers in the Department, for their constant support and feedback on the work I have done, without which this thesis would not be complete. Lastly, I express deep gratitude to my family for providing me with the opportunities that brought me to this stage, and I will be forever indebted to them for their constant nurturing and support throughout my academic career.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

According to the observer effect, when we try to measure something, we change it to some extent [2]. Therefore, precise measurement is never truly possible. Computer programs suffer from this same difficulty. When you try to measure something in a computation, it changes the underlying computation. In an ideal world, a program may want to make sure that every step that it is taking is correct with respect to any environmental changes. However, the time taken for performing these checks may cause the program to be incorrect. In other words, it is possible that adding excessive safety checks or checks for guaranteeing fairness may cause the system to spend substantial time on those checks, thereby violate system requirements, even though those requirements would never have been violated without those checks in the first place.

This issue is even more complicated in distributed computing where each process (component, node, etc.) relies on partial information. Hence, computing the required safety checks (or checking for the satisfaction of fairness requirements, etc.) would require processes to communicate with each other. In turn, the time for computing them would be even higher.

As an illustration, consider two drones A and B that are cooperating to perform a task. Each drone may take independent actions based on some environment that the other drone cannot see. It is required that the area covered by the drones remains 50% or above at all times (100% of the time). It is also preferred that this area remains at 75% most of the time (75% of the time). Here, we would like to know (1) how frequently a given point x was covered by one of the drones, (2) how frequently a given point was covered by both the drones, (3) what was the minimum coverage at any time, etc. (Assume that they are at two different altitudes so safety measures such as preventing collision are not necessary). One possibility is to require A to notify B of its actions at all times to ensure that B can adjust its plan to account for what A is doing, and vice versa. However, this will require A to unnecessarily spend more time communicating with B, and vice versa. In other words, it may be necessary for A and B to move independently. Doing all these checks during the execution would require that A and B communicate with each other before they make any move. In

turn, it would change the behavior of the drones completely thereby preventing us from making any conclusion about their behavior in the absence of these checks. Furthermore, the problem would be even more complicated if we had a larger number of drones.

One way to address this problem is to log the computation as it is happening so we can evaluate it later for all properties of interest. These properties may include non-critical safety properties or desirable performance criteria, etc. To be beneficial, the amount of storage for the log or the time to create that log should be small enough so that the underlying computation is affected minimally. In other words, the measurement should not change the underlying computation substantially. At the same time, the log should capture the non-determinism that is inherently present in any distributed computation. We also want to make sure that the creation of the logs is performed independently by each process, i.e., each process stores its local state whenever it changes along with a timestamp (discussed next) that identifies when the change was made. We also assume that all messages are logged as well. We consider various approaches for storing the timestamps and their implications.

The simplest approach we can consider is to let the timestamp be the physical time of the relevant process. Here, the storage and computation cost is very low. However, the physical clocks of processes often differ. Hence, drone A may send a message at time 50 (local time of A) but it is received by B at time 40 (B's local time). When we try to reply to this log to evaluate the given properties, it will cause B to receive the message before A has sent it. This is unacceptable, as it violates the system's consistency.

The next approach we can consider is vector clocks. Vector clocks introduce two concerns: Their size of $O(n)$, where n is the number of processes, may be too high. Another challenge is that vector clocks do not have any reference to the physical clock and do not account for communication outside the system. For example, it is possible that drone A activated a green LED at physical time t_1 and B activated a white LED at time t_2 where $t_2 >> t_1$. In other words, an external observer will know that the action of A occurred before B. However, if A and B did not communicate then the corresponding events will be concurrent [3]. Thus, when we replay the log, it is possible that the white LED event could be replayed before the green LED event. This is also unacceptable.

Hybrid logical clocks (HLC) [4] combine logical clocks and physical clocks. Specifically, they rely on a system where physical clocks are synchronized within the acceptable limit of clock skew, $\mathcal{E}$, they guarantee that $hlc.e < hlc.f$ if e happened before f or $pt.e + \mathcal{E} < pt.f$ ([3]). Here, $hlc.e$ denotes the Hybrid Logical Clock of process e, and $pt.e$ denotes the physical time observed on process e. In other words, $hlc.e < hlc.f$ if f causally depends upon e or f occurred substantially after e. They eliminate the problem associated with physical clocks as HLC respects causality. They also eliminate the problem caused by vector clocks as the HLC timestamp of activating the green LED will be less than the HLC timestamp of activating the white LED. HLC does create another problem though. Consider the case where we have events e and f such that $|pt.e - pt.f| < \mathcal{E}$ and $e||f$, i.e., the events are causally concurrent and very close to each other in physical time. Without loss of generality, let $hlc.e < hlc.f$. In this situation, when we replay the log, e will always occur before f. In other words, the log does not have the necessary information that could allow it to replay f before e even though they could have occurred in any order.

An extension of HLC, hybrid vector clocks [1] reduces some of these issues. However, as we highlight in Section 4.6, this overhead is still quite high.

Based on these limitations, in this thesis, we focus on building a new clock, the Replay Clock (*RepCl*), that combines hybrid logical clocks and vector clocks to eliminate their limitations. Our goal is to investigate scenarios under which *RepCl* permits efficient replay of events. To understand why we may need to limit *RepCl* to specific scenarios, observe that if the underlying system was asynchronous (unbounded clock drift) then it is required to have $O(n)$ vector clocks to enable replay of events. Systems that communicate frequently will need more information stored to replay events. Thus, we focus on the following problem: *Given the amount of permissible overhead for logging events, what are the scenarios where perfect replay of events is possible?*

Once we identify the scenarios in which perfect replay is available, we design a trace visualizer, named *RepViz*, that allows us to depict candidate traces with the ability to replay concurrent events in any order of execution. This visualizer takes in a *RepCl*-timestamped trace to generate a visualization depicted in Chapter 7. We provide per-process views with timelines of events to

depict the events occurring while indicating causality between those events. The trace visualizer provides a interactive display to the user with orderings of events, and allows the user to reorder concurrent events to view different candidate traces. The visualizer API is discussed in Chapter 7.

1.1 Contributions

- We present *RepCl*, a replay clock that enables the replay of events in a distributed system. It guarantees that if there is a causal relation [3] or if f occurred far later than e then $RepCl.e < RepCl.f$, i.e., the replay will cause e to replayed before f. On the other hand, if e and f are causally concurrent and occurred close in physical time then they could be replayed in any order.

- By considering various system parameters, clock skew ($\mathcal{E}$), message rate (α), and message delay (δ), we identify the feasibility region for *RepCl*.

- We implement an API for the *RepCl* for NS-3, a widely used distributed network simulator. It provides all the operations and documentation on how it integrates with different network components available to the NS-3 simulations.

- We design *RepViz*, a visualizer that generates traces from *RepCl*-timestamped logs in a distributed computation. The visualizer provides the user with various orders of replay, and allows the user to view different candidate traces and evaluate various constraints along those traces through the visualization.

Organization of the thesis: This thesis is organized as follows. In Chapter 2, we describe the model of computation for distributed systems including the notion of causality and clock synchronization. We move forward to the idea of the replay clock and the problems it solved in Chapter 3. in Chapter 4 we describe the algorithms associated with the *RepCl*, and describe the properties of the *RepCl*. Additionally, we describe the method of representation for the *RepCl* and describe the various overheads that are characteristic of the design of the clock. Chapter 5 talks about the design of the simulators we used to collect metrics for the *RepCl*. These metrics are

discussed in Chapter 6 with analysis on the size of the clock and feasibility of implementation of the clock. Chapter 7 talks about the design of the *RepViz*, the visualization system for trace building using the *RepCl*. Chapter 8 discusses related work and identifies questions raised by *RepCl*. Finally, in Chapter 9, we conclude and discuss future work.

CHAPTER 2

PRELIMINARIES

A distributed system is a set of processes $1..n$. Each process has three types of events (1) $send$, where it sends a message to another process, (2) $receive$, where it receives a message from another process, and (3) $local$, where it performs some local computation.

We define the happened-before (denoted by hb) relation [3] among the events in a distributed computation.

- If e and f happened on the same process and e occurred before f then $e\ hb\ f$.
- If e was a send event and f was the corresponding receive event then $e\ hb\ f$.
- The hb relation is transitive, i.e., if there exist events e, f, and g such that $e\ hb\ g$ and $g\ hb\ f$ then $e\ hb\ f$

We say that $e||f$ iff $\neg(e\ hb\ f) \wedge \neg(f\ hb\ e)$. In other words, e is concurrent with f iff e did not happen before f and f did not happen before e.

A timestamping algorithm assigns a timestamp for every event e in the system as soon as the event is created. Additionally, the timestamping algorithm defines a $<$ relation that identifies how two timestamps are compared.

As an illustration, Lamport's logical clock [5] assigns an integer timestamp $l.e$ to every event e. The $<$ relation for Lamport's logical clocks is the standard $<$ for integers. Likewise, the physical timestamping algorithm assigns $pt.e$ for every event e where $pt.e$ is the physical time of the process where event e occurred when it occurred, and the $<$ relation is the same as that over integers. A vector clock [6][7] assigns event e a timestamp $vc.e$ where $vc.e$ is a vector that includes an entry $vc.e.j$ for every process j. The $<$ relation on two vector clocks $vc.e$ and $vc.f$ requires that each element in $vc.e$ is less than or equal to the corresponding element in f and some element in e is less than the corresponding element in f. In other words, $vc.e < vc.f$ iff $(\forall j :: vc.e.j \leq vc.f.j) \wedge (\exists j :: vc.e.j < vc.f.j)$.

Note that while the $<$ relation is defined by the timestamping algorithm, the properties of the $<$ relation vary. For example, logical clocks provide one-way causality information, i.e., $e \; hb \; f \Rightarrow l.e < l.f$. Vector clocks provide two-way causality information, i.e., $e \; hb \; f \Leftrightarrow vc.e < vc.f$. By contrast, (unsynchronized) physical clocks may not provide any guarantees. For example, it is possible that $(e \; hb \; f)$ and $pt.e \not< pt.f$ are simultaneously true.

We assume that each process j in the system is associated with a physical clock, $pt.j$. Clocks of processes are synchronized with a protocol such as NTP [8] such that the clock of two processes differ by at most $\mathcal{E}$, where $\mathcal{E}$ is a parameter, i.e., $\forall j, k :: |pt.j - pt.k| \leq \mathcal{E}$. We also assume that individual clocks are monotonically increasing. We assume that messages are delivered with a *minimum message delay* of δ. We do not assume maximum message delay; it could be ∞ if messages are permitted to be lost. Since our focus is on the replay of events, if a message is lost, it simply implies that the corresponding received message is never replayed.

CHAPTER 3

REPLAY WITH CLOCKS

In this chapter, we focus on how clocks can be used to replay a given computation. We also discuss some of the limitations of using logical clocks and vector clocks in the replay process. Note that the goal of this chapter is only to illustrate the concept and the goals of the replay; it does not focus on developing an *efficient* algorithm for the same.

As discussed in the introduction, the goal of replay is to order the events so that we can evaluate various properties of interest. To replay a given computation, we begin with the set where each entry is of the form $\langle e, ts.e \rangle$, where e is the event (send/receive/local) and $ts.e$ is the timestamp of e. To replay the given set of events, we first find events e such that all events with smaller timestamps than e have already been replayed. In other words, we find the set $\{e|\neg(\exists f : ts.f < ts.e)\}$. We replay one of these events randomly. Then, we remove event e. The process is continued until all events are replayed. Thus, the algorithm for replay is shown in algorithm 3.1.

Algorithm 3.1 ReplayEvents Operation

1. **Input:** S: Set of Events and timestamp
2. **While** $S \neq \phi$ **do**
3. $FrontLine = \{e|(e, ts.e) \in S \wedge \neg(\exists f : (f, ts.f) \in S \wedge ts.f < ts.e)\}$
4. Choose a random event e from $FrontLine$ and replay it
5. $S = S - \{e\}$
6. **end while**

3.1 Limitations of Existing Clocks for Replay

As an example, consider the execution in Figure 3.1. Here, we have four events A, B, C, and D. Their physical timestamps and logical timestamps are shown in Figure 3.1. If we replay these events using physical clocks then the possible outcomes are $CBAD$ or $CBDA$. Note that both these outcomes are undesirable, as A should occur before B based on the causality (happened-before) relation.

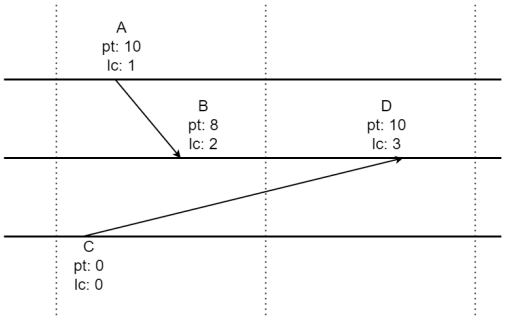


Figure 3.1 Sample Execution Sequence and Application of Replay Algorithm.

If we replay them by logical clocks, the possible outcome is $CABD$. However, there is no option to replay B before C even though $B||C$.

If we replay them with vector clocks, possible orderings are $ABCD$, $ACBD$, or $CABD$. If the clocks were synchronized to be within 5 time units then $ABCD$ and $ACBD$ are incorrect.

3.2 Requirements of Replay Clock (*RepCl*)

In this thesis, we focus on a system where the physical clocks are synchronized to be within $\mathcal{E}$, i.e., for any two processes j and k, $|pt.j - pt.k| \leq \mathcal{E}$. The goal of *RepCl* is to assign a timestamp $RepCl.e$ to event e such that

Requirement 1. If e happened before f then

$RepCl.e < RepCl.f$, i.e., e will always be replayed before f

Requirement 2. If f occurred far after e, i.e., e and f could not have occurred simultaneously under clock drift guarantee of $\mathcal{E}_1$ where $\mathcal{E}_1 \approx \mathcal{E}$ then e will be replayed before f, i.e., $RepCl.e < RepCl.f$

Requirement 3. If e and f could have occurred in any order in a system where clocks were synchronized to be within $\mathcal{E}_2$, where $\mathcal{E}_2 \approx \mathcal{E}$ then $RepCl.e||RepCl.f$ (i.e., $\neg(RepCl.e < RepCl.f) \wedge \neg(RepCl.f < RepCl.e)$)

In the last two requirements, we have chosen $\mathcal{E}_1$ and $\mathcal{E}_2$ instead of $\mathcal{E}$ itself as it can permit more efficient implementation by allowing us to maintain a coarse-grained clock. We discuss this further in section 4.6.

With these requirements in mind, we show that the *RepCl* provides efficient replays of distributed computations, without suffering with the overhead that vector clocks impose, and the shortcomings of replay with logical clocks like the HLC. The *RepCl* combines the best of both these worlds, and provides a better mechanism to replay computations.

CHAPTER 4

ALGORITHM FOR THE REPLAY CLOCK (*RepCl*)

In this chapter, we present our approach for *RepCl*. As discussed earlier, we assume that the physical clocks are synchronized to be within $\mathcal{E}$. We discretize the process execution in terms of epochs, where each epoch corresponds to an increment of the clock by I, $0 < I \leq \mathcal{E}$ such that $\mathcal{E} = \epsilon * I$, where ϵ is an integer. The timeline of a process is split into epochs where each epoch is of size I(in the local process clock). In other words, the epoch of process j is obtained by $\lfloor \frac{pt.j}{I} \rfloor$.

4.1 Structure of *RepCl* Timestamp

With such discretization, the timestamp of process j (or event e) is of the form

$$\langle mx.j, \texttt{bitmap}.j[], \texttt{offset}.j[], \texttt{counter}.j[] \rangle, \tag{4.1}$$

where $mx.j$ is an integer for the approximation of the top-level HLC, and $\texttt{bitmap}.j$, $\texttt{offset}.j$ and $\texttt{counter}.j$ are bitsets [9] that store *at most* one entry $\texttt{bitmap}.j.k$, $\texttt{offset}.j.k$ and $\texttt{counter}.j.k$ for process k. Each of these bitsets is treated as an array but are serialized as integers in packets for efficiency.

The intuition behind these variables is as follows:

- $mx.j$ denotes the maximum epoch process j is aware of (either due to the value of $pt.j$ or the value of epochs learned from messages it receives).
- $\texttt{bitmap}.j.k$ is essentially an array of bits, where each bit with index k denotes whether the $\texttt{offset}.j.k$ is being stored. If the bit is 1, process j is actively maintaining $\texttt{offset}.j.k$. This will come in handy for efficient updates to the clock.
- $mx.j - \texttt{offset}.j.k$ denotes the maximum epoch value of k that j has learnt (either via direct/indirect message from k, clock drift assumption, etc). If there exists an offset between two processes j and k, $\texttt{offset}.j.k$ denotes the difference between $mx.j$ and $mx.k$ as seen on process j.

- Counters are used to deal with the scenario where multiple events happen within the same epoch, and have the same offsets. If two clocks that are not concurrent have the same *mx* and the same `offset` values, then the two clocks differ on the counters. The clock with the lower counter value is replayed first.

For example, the timestamp $\langle 50, [1,1,1], [0,1,2], [4,5,6] \rangle$ denotes that this event is aware of epoch 50 of process 0 (as $50-0$), 49 of process 1 (as $50-1$), and 48 of process 2 (as $50-2$). And, the counter values are $4, 5$ and 6 respectively.

4.2 Efficient traversal and lookup

All computations are optimized using the bitmap. While the bitmap does not serve a purpose to the timestamp itself, it allows us to traverse and update the clock efficiently. Here we describe the traversal and lookup of offsets based on the bitmap. For brevity, we describe the rest of the algorithms as a simple traversal, but it is important to note that each traversal takes O(number of 1s in the bitmap) time complexity, and getters and setters take O(1) complexity. To describe these implementations, we use the integer representations of $\texttt{offset}.j[]$ and $\texttt{counter}.j[]$.

4.2.1 Traversal

The traversal operation is described in algorithm 4.1, which iterates through the `bitmap` to find all processes for which the `offset` is being maintained. In the algorithm, we get every index that has a set bit in the bitmap. The set bit at position k indicates that $\texttt{offset}.j.k$ is being stored by process j.

4.2.2 Extract

The extract operation extracts k bits from position p. The algorithm is described in algorithm 4.2.

4.2.3 GetOffsetAtIndex

GetOffsetAtIndex is an operation that gets the offset stored at a particular index. This is an O(1) lookup operation using the index obtained from the bitmap traversal or a specific offset that the clock may need at any index. The algorithm is described in algorithm 4.3. Here, τ denotes the

Algorithm 4.1 Traversal Operation

1. Define Traversal operation.
2. Traversal(ts)
3. **While** $ts.bitmap > 0$
4. $\quad index = \log_2(\neg(ts.bitmap \oplus (\neg(ts.bitmap - 1)) + 1) >> 1)$
5. $\quad$ // Get or set any offset at index
6. $\quad ts.bitmap = ts.bitmap \wedge (ts.bitmap - 1)$
7. **End While**

Algorithm 4.2 Extract Operation

1. Define Extract operation.
2. Extract($number, k, p$)
3. **Return** $((1 << k) - 1 \wedge (number >> p))$

max offset size allowed by the user (in bits). The max offset size is usually set to $log(\epsilon)$.

Algorithm 4.3 GetOffsetAtIndex Operation

1. Define GetOffsetAtIndex operation.
2. GetOffsetAtIndex($ts, index$)
3. $offset = Extract(offset.j[].ToInteger(), \tau, \tau * index)$
4. **Return** $offset$

4.2.4 SetOffsetAtIndex

SetOffsetAtIndex is an operation that sets the offset at a particular index. This is an O(1) setter operation using the index obtained from the bitmap traversal or a specific offset that the clock may need at any index, like the GetOffsetAtIndex algorithm. The algorithm is described in algorithm 4.4.

Algorithm 4.4 SetOffsetAtIndex Operation

1. Define SetOffsetAtIndex operation.
2. SetOffsetAtIndex($ts, index, newoffset$)
3. $firstpart = Extract(offsets.j, \tau * index, 0)$
4. $res| = firstpart$
5. $res| = newoffset << index * \tau$
6. $lastpart = Extract(offsets.j, \tau * N - (\tau * (index + 1)), \tau * (index + 1))$
7. $res| = lastpart << (index + 1) * \tau$
8. **Return** res

4.2.5 RemoveOffsetAtIndex

The RemoveOffsetAtIndex operation removes an offset given an index. This is an O(1) removal operation once the position is found by the traversal operation. This algorithm is described in Algorithm 4.5.

Algorithm 4.5 RemoveOffsetAtIndex Operation

1. Define RemoveOffsetAtIndex operation.
2. RemoveOffsetAtIndex($ts, index, newoffset$)
3. $firstpart = Extract(offsets.j, \tau * index, 0)$
4. $res| = firstpart$
5. $lastpart = Extract(offsets.j, \tau * N - (\tau * (index + 1)), \tau * (index + 1))$
6. $res| = lastpart << (index + 1) * \tau$
7. **Return** res

4.3 Helper functions

Now that we have described the traversals and auxiliary operations, we move on to the clock helper algorithms. For brevity, we assume all traversals and assignments are the algorithms described in the previous subsection. We omit the bitmap to make it easier to understand the

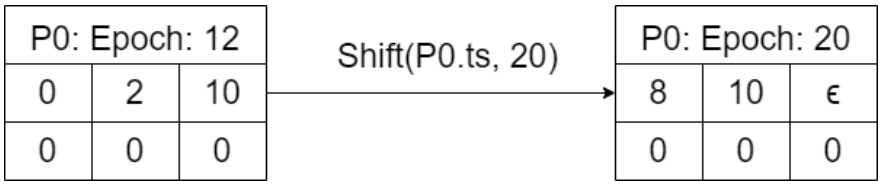


Figure 4.1 Working of Shift() on Process 0. Here, $\epsilon = 15$, and the shift is issued to advance to mx 20. Process 3's `offset` becomes 20, but since $\epsilon = 15$, the `offset` is set to ϵ (due to clock skew limit guarantees).

algorithms that follow. We discuss two helper functions, *Shift* and *MergeSameEpoch*, that will come in handy when we design the main clock processing algorithms.

4.3.1 Shift Operation

The Shift function allows us to change the value of mx. Since $mx.j - \texttt{offset}.j.k$ denotes the knowledge of j has about the epoch of k, if mx is changed to newmx without providing j any additional knowledge of the clock of k then $newmx - \texttt{newoffset}.j.k$ should remain the same as $mx.j - \texttt{offset}.j.k$. Hence, Shift operation changes offset.$j.k$ to be $\texttt{offset}.j.k + (newmx - mx)$. Furthermore, if this value is more than ϵ then we reset it to ϵ, as guaranteed by the clock drift assumption. (Note that process j can learn about the clock of k via clock synchronization assumption even if j and k do not communicate.)

For example, shifting the timestamp $\langle 12, [0, 2, 10]\rangle$ so that mx is changed to 20 will result in $\langle 20, [8, 10, 18]\rangle$. If $\epsilon = 15$, this will change to $\langle 20, [8, 10, \epsilon]\rangle$ (cf. Figure 4.1).

4.3.2 MergeSameEpoch Operation

The MergeSameEpoch function takes two timestamps $t1$ and $t2$ with the same mx value and combines their offsets by setting to be $\texttt{offset}.j.k$ to be the $min(t1.\texttt{offset}.j.k, t2.\texttt{offset}.j.k)$. For example, merging $\langle 50, [0, 1, 2]\rangle$ and $\langle 50, [2, 0, 1]\rangle$ results in $\langle 50, [0, 0, 1]\rangle$.

4.3.3 EqualOffset Operation

The EqualOffset function takes in two timestamps $t1$ and $t2$ and checks whether the `offset` arrays and mx values are the same. This is used particularly to update the `counter` array if the other values are equal.

Algorithm 4.6 Shift Operation

1. Define Shift operation.
2. Shift($ts, newmx$)
3. **For each** k **do**
4. $\quad ts.\texttt{offset}.k = \texttt{offset}.k + (newmx - ts.mx)$
5. $\quad$ **If** $ts.\texttt{offset}.k > \epsilon$ **then**
6. $\quad\quad ts.\texttt{offset}.k = \epsilon$
7. $\quad$ **End If**
8. **End For**
9. **Output:** ts

Algorithm 4.7 MergeSameEpoch Operation

1. **Input:** Timestamp $t1$, Timestamp $t2$
2. Timestamp ts = new Timestamp
3. **For each** k **do**
4. $\quad ts.\texttt{offset}.j.k = \min(t1.\texttt{offset}.j.k, t2.\texttt{offset}.j.k)$
5. **End For**
6. **Return** ts

Algorithm 4.8 EqualOffset Operation

1. **Input:** Timestamp $t1$, Timestamp $t2$
2. **If** $t1.mx \neq t2.mx \vee (\exists j\ t1.\texttt{offset}.j \neq t2.\texttt{offset}.j)$ **then**
3. $\quad$ **Return** false
4. **Else**
5. $\quad$ **Return** true
6. **End If**

4.4 Description of the *RepCl* Algorithm

In this section, we discuss the key clock processing algorithms. These algorithms update the clock based on the type of event observed on the process. We describe two key operations: Send/Local and Receive.

4.4.1 Local/Send event

Here, we describe how $RepCl.j$ is updated when j sends a message. Let the current timestamp of j be

$$\langle mx.j, \texttt{bitmap}.j[], \texttt{offset}.j[], \texttt{counter}.j[] \rangle \tag{4.2}$$

First, $mx.j$ needs to be increased if the clock of j has advanced beyond epoch $mx.j$. Hence, we first compute $newmx.j$ which is equal to $max(mx.j, \texttt{epoch}.j)$. When j sends a message, it does not learn any new information about the clock of process k.

We consider two cases: The first case is for the scenario where the newly created event f is in a new epoch as the previous event, e. This will happen if mx remains unchanged and $\texttt{offset}.j.j$ is unchanged. In this case, we increase $\texttt{counter}.j.j$.

The second case deals with the scenario where f is in a new interval. Thus, the offset associated with k is changed using the Shift operation. Note that the Shift operation computes the shift of all processes except j. $\texttt{offset}.j.j$ should be based on the value of $\texttt{epoch}.j$. Hence, we set it equal to $newmx - \texttt{epoch}.j$. The Shift operation is illustrated in Algorithm 4.6.

4.4.2 Receive event

Next, we describe how *RepCl* is updated when j with timestamp

$$\langle mx.j, \texttt{offset}.j[], \texttt{counter}.j[] \rangle \tag{4.3}$$

receives a message m with timestamp $\langle mx.m, \texttt{offset}.m[], \texttt{counter}.m[] \rangle$.

First, we compute $newmx$ which is the maximum of $mx.j$, $mx.m$ and $\texttt{pt}.j$. Timestamps of j and m are then shifted to $newmx$ using the Shift operation. These timestamps are then merged to obtain the mx and $\texttt{offset}$ values of the new event, say f.

Algorithm 4.9 Send Message

1. $newmx = \max(mx.j, \texttt{pt}.j)$
2. $\texttt{new\_offset} = newmx - \texttt{pt}.j$
3. **If** $(mx.j = newmx \wedge \texttt{offset}.j.j = \texttt{new\_offset})$ **then**
4. $\quad \texttt{counter}.j.j = \texttt{counter}.j.j + 1$
5. **Else**
6. $\quad ts.j = Shift(ts.j, newmx)$
7. $\quad \texttt{offset}.j.j = \min(newmx - \texttt{pt}.j, \epsilon)$
8. $\quad \texttt{counter}.j = [0, 0, \ldots, 0]$
9. **End If**

Now, we check if the knowledge that f has about epochs is the same as that of e (the previous event on j) or m. If all three are in the same epoch then $\texttt{counter}.j.j$ is set to one more than the maximum of $\texttt{counter}.j.k$ and $\texttt{counter}.m.k$. If only e and f are in the same epoch, $\texttt{counter}.j.j$ is incremented by 1. If only m and f are in the same epoch, $\texttt{counter}.j$ is set to $\texttt{counter}.m$ and the value of $\texttt{counter}.j.j$ is incremented by 1. If none of these conditions apply then counters are reset to 0.

4.5 Comparing *RepCl* Timestamps

The *happens-before* relation in *RepCl* is codified in Algorithm 4.11. In this algorithm, we check whether timestamp $t1$ happens-before timestamp $t2$. In this relation, we first compare the HLCs of the two *RepCl* timestamps. Since HLC provides the top-level information of the physical clock of the message, it resolves ties with clocks having different HLCs. If the HLCs are equal we move to the offsets. For $t1$ to be strictly happening before $t2$, we follow the comparison used in traditional vector clocks, where $vc.e < vc.f$ iff $(\forall j :: vc.e.j \leq vc.f.j) \wedge (\exists j :: vc.e.j < vc.f.j)$. If the offsets for the two timestamps are also equal, we check for the counters.

We say two events are concurrent by the definition introduced in Requirement 3, which basically states that if $\neg(RepCl.e < RepCl.f) \wedge \neg(RepCl.f < RepCl.e)$, then $RepCl.e || RepCl.f$.

Algorithm 4.10 Receive Message

1. **Input:** Received Message m
2. $newmx = \max(mx.j, mx.m, \texttt{pt}.j)$
3. $ts.a = \texttt{Shift}(ts.j, newmx)$
4. $ts.b = \texttt{Shift}(ts.m, newmx)$
5. $ts.c = \texttt{MergeSameEpoch}(ts.a, ts.b)$
6. **If** $\texttt{EqualOffset}(ts.j, ts.c) \wedge \texttt{EqualOffset}(ts.m, ts.c)$ **then**
7. **For each** k **do**
8. $\texttt{counter}.j.k = \max(\texttt{counter}.j.k, \texttt{counter}.m.k)$
9. **End For**
10. $\texttt{counter}.j.j = \texttt{counter}.j.j + 1$
11. **End If**
12. **If** $\texttt{EqualOffset}(ts.j, ts.c) \wedge \neg\, \texttt{EqualOffset}(ts.m, ts.c)$ **then**
13. $\texttt{counter}.j.j = \texttt{counter}.j.j + 1$
14. **If** $\neg\, \texttt{EqualOffset}(ts.j, ts.c) \wedge \texttt{EqualOffset}(ts.m, ts.c)$ **then**
15. $\texttt{counter}.j = \texttt{counter}.m$
16. $\texttt{counter}.j.j = \texttt{counter}.j.j + 1$
17. **If** $\neg\, \texttt{EqualOffset}(ts.j, ts.c) \wedge \neg\, \texttt{EqualOffset}(ts.m, ts.c)$ **then**
18. $\texttt{counter}.j = [0, 0, \ldots, 0]$

Algorithm 4.11 Compare Operation

```
1.  Input: Timestamp t1, Timestamp t2
2.    If t1.mx < t2.mx then
3.      Return True
4.    Else If t1.mx > t2.mx then
5.      Return False
6.    Else then
7.      For i, j in t1.offsets, t2.offsets
8.        If t1.offsets.i > t2.offsets.j
9.        Return False
10.     If t1.counters <= t2.counters
11.       Return True
12.   End For
13.     Return False
14.   End If
```

As an illustration, consider the execution of the program in Figure 3.1. Assuming that $\epsilon = 5, I = 1$, and $\mathcal{E} = 5$, the *RepCl* timestamps will be as shown in Figure 4.2. Here, event A has physical time of 50. Since process P1 has not heard from anyone else so far, the offsets for P2 and P3 will be ϵ. The offset for process P1 will be 0. Regarding event C, the situation is similar except that the offset for process P3 is 0. When event B is created upon receiving message m_1, process P2 is aware of times 50 from $P1$. And, it is the maximum epoch it is aware of. Hence, offsets are $[0, 2, \epsilon]$ respectively. When event D is created, process P2 is aware of epoch 52 (from P2) and epoch 50 (from P1). It is aware of timestamp 40 from P3. However, this information is overridden by the clock synchronization guarantee that says that the clock of P3 is at least 47. Thus, the offsets are set to $[3, e, \epsilon]$ Here, the permissible ordering is $CABD$.

In this figure, if ϵ were 20 then the timestamp of D would be changed to $[3, 2, 12]$. Furthermore,

B and C could be replayed in any order. Thus, the permissible replays would be $CABD$ or $ABCD$ or $ACBD$.

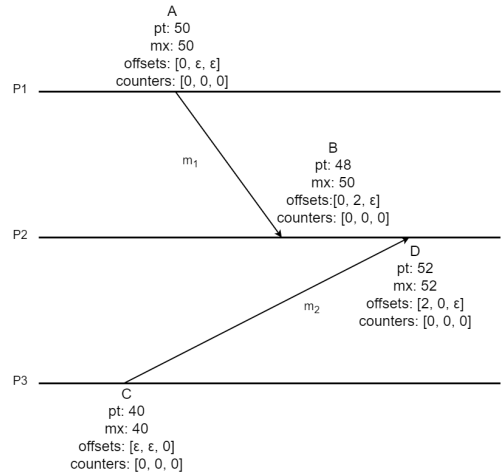


Figure 4.2 Replay of the Execution in Figure 3.1 with *RepCl*.

4.6 Properties of *RepCl*

In this section, first, we define the $<$ relation on two timestamps $RepCl.e$ and $RepCl.f$. Then, we identify the properties of this $<$ relation and the happened-before relation.

Given timestamps

$RepCl.e = \langle mx.e, \texttt{offset}.e[], \texttt{counter}.e[] \rangle$ and

$RepCl.f = \langle mx.f, \texttt{offset}.f[], \texttt{counter}.f[] \rangle$,

we say that $RepCl.e < RepCl.f$ iff

$$
\begin{aligned}
& mx.f > mx.e + \mathcal{E} \\
& \quad \vee \Bigg(|mx.f - mx.e| \leq \mathcal{E} \\
& \qquad \wedge \Bigg(\Big(\forall l (mx.e - \texttt{offset}.e.l) \leq (mx.f - \texttt{offset}.f.l) \Big) \\
& \qquad\quad \wedge \Big(\exists l (mx.e - \texttt{offset}.e.l) < (mx.f - \texttt{offset}.f.l) \Big) \Bigg) \\
& \qquad \vee \Bigg(\forall l (mx.e - \texttt{offset}.e.l) = (mx.f - \texttt{offset}.f.l) \\
& \qquad\quad \wedge \Bigg(\forall l (\texttt{counter}.e.l) \leq (mx.f - \texttt{counter}.f.l) \\
& \qquad\qquad \wedge \exists l (\texttt{counter}.e.l) < (\texttt{counter}.f.l) \Bigg) \Bigg) \Bigg)
\end{aligned}
$$

The above $<$ relation first compares if $mx.f$ and $mx.e$ are far apart. If that is the case, we define $RepCl.e < RepCl.f$. If they are close, i.e., $|mx.f - mx.e| \leq \epsilon$, then, we compare the offsets. Since $mx.e-$offset$.e.k$ identifies the knowledge e had about the epoch of process k, we use a comparison that is similar to vector clocks to determine if $<$ relation holds between $RepCl.e$ and $RepCl.f$. Finally, if the offsets are also equal then we use the comparison of counters (again in the same fashion as vector clocks).

We overload the $||$ relation for comparing timestamps as well. Specifically, given timestamps $RepCl.e$ and $RepCl.f$, we say that $RepCl.e||RepCl.f$ iff

$$\neg(RepCl.e < RepCl.f) \wedge \neg(RepCl.f < RepCl.e) \tag{4.4}$$

From the construction of the timestamp algorithm, we have the following two lemmas:

Lemma 1: (e happened before f) $\Rightarrow RepCl.e < RepCl.f$

Lemma 2: $|mx.e - maxt.f| \leq \mathcal{E} \wedge (e||f) \quad \Rightarrow \quad RepCl.e||RepCl.f$

Requirement 1 of *RepCl*: Observe that Lemma 1 satisfies the first requirement of *RepCl*; if e happened before f then e must be replayed before f.

Requirement 2 of *RepCl*: Now, we focus on the second requirement. Specifically, we show that by letting $\mathcal{E}_1 = \mathcal{E} + I$, the second requirement is satisfied.

Observe that in the *RepCl* algorithm, messages carry the epoch values of multiple processes. This allows a process to learn epoch information about other processes. For the subsequent discussion, imagine that the messages also carried the actual physical time as well. In this case, j will learn about the clock of a process k via such messages. Additionally, j will also learn about the clock of a process k based on the assumption of clock synchronization. Likewise, when event e is created, it will have some information about the clock of each process. Let $mxph.e$ and $mxph.f$ be the maximum clock (of any process) that e and f are aware of when they occurred. If $mxph.f > mxph.e + \mathcal{E}_1$ then f cannot occur before e under the clock synchronization guarantee of $\mathcal{E}_1$. Now, we show that in this situation, it is guaranteed that $RepCl.e < RepCl.f$.

By definition of mx, $mx.f = \lfloor \frac{mxph.f}{I} \rfloor$ and $mx.e = \lfloor \frac{mpt.e}{I} \rfloor$. Additionally, we have

$$
\begin{aligned}
& -1 < (x - \lfloor x \rfloor) - (y - \lfloor y \rfloor) < 1 \\
\implies & -1 < (\tfrac{mxph.f}{I} - \lfloor \tfrac{mxph.f}{I} \rfloor) - (\tfrac{mxph.e}{I} - \lfloor \tfrac{mxph.e}{I} \rfloor) < 1 \\
\implies & -1 < (\tfrac{mxph.f}{I} - mx.f) - (\tfrac{mxph.e}{I} - mx.e) < 1 \\
\implies & -1 < (\tfrac{mxph.f - mxph.e}{I}) - (mx.f - mx.e) < 1 \\
\implies & -I \leq ((mxph.f - mxph.e) - (mx.f - mx.e)I \leq I
\end{aligned}
$$

Now, if $mxph.f - mxph.e > \mathcal{E} + I$ then we can rewrite the second inequality as $(\frac{\mathcal{E}+I}{I} - (mx.f - mx.e) \leq 1$. Using the fact that $\mathcal{E} = \epsilon * I$, we have $\epsilon < (mx.f - mx.e)$. In other words, $mxph.f - mxph.e > \mathcal{E} + I \Rightarrow (mx.f > mx.e + \epsilon)$. which gives us $RepCl.e < RepCl.f$. In other words, we have

Lemma 3: If f occurred far after e, i.e., f could not have occurred before e in a system that guarantees that clocks are synchronized within $\mathcal{E}_1 = \mathcal{E} + I$ then e will be replayed before f, i.e., $RepCl.e < RepCl.f$.

Requirement 3 of *RepCl*: Next, we consider the case where e and f could have occurred in any order if the underlying system guaranteed that clocks were synchronized to be within $\mathcal{E}_2 = \mathcal{E} - I$. Letting the maximum clock that event e (respectively, f) was aware of to be $mxph.e$ (respectively, $mxph.f$), we observe that $|mxph.e - mxph.f| \leq \mathcal{E}_2$. Furthermore, e and f must be causally concurrent. Under this scenario, we show that $RepCl.e||RepCl.f$.

If $|mxph.e - mxph.f| \leq \mathcal{E} - I$, we have

$$\begin{aligned}
&\implies |\tfrac{mpt.e}{I} - \tfrac{mpt.f}{I}| \leq \epsilon - 1 && \text{// since } \mathcal{E} = \epsilon * I \\
&\implies |\lfloor \tfrac{mpt.e}{I} \rfloor - \lfloor \tfrac{mpt.f}{I} \rfloor| \leq \epsilon && \text{since } |(x - \lfloor x \rfloor) - (y - \lfloor y \rfloor)| < 1 \\
&\implies |mx.e - mx.f| \leq \epsilon && \text{by definition of } mx
\end{aligned}$$

Now, from Lemma 2, $RepCl.e||RepCl.f$. In other words,

Lemma 4: If e and f could have occurred in any order in a system where clocks were synchronized to be within $\mathcal{E} - I$ then $RepCl.e||RepCl.f$.

4.7 Effect of discretization and comparison with Hybrid Vector Clocks [1]

We note that the discretization of the clock via I has caused the bounds used for clock synchronization in Lemmas 1 and 2 to be different. We could have eliminated this if we had not discretized the clocks. (Discretization with I was not done in [1].) However, without discretization, the values of offsets will be very large. Without discretization, we will need to rely on just the physical clocks which have a granularity of under 1 nanosecond. Now, if $\mathcal{E} = 1ms$ then the value of the offset could be as large as 10^6. By discretizing the clock, it would be possible to keep offsets to be very small. We expect that the discretization will not seriously impact the replay. For example, if $\mathcal{E} = 1ms$ and $I = 0.1ms$ then our algorithm will guarantee that causally concurrent events within $0.9ms$ can be replayed in any order. And, events that could not occur simultaneously under a clock

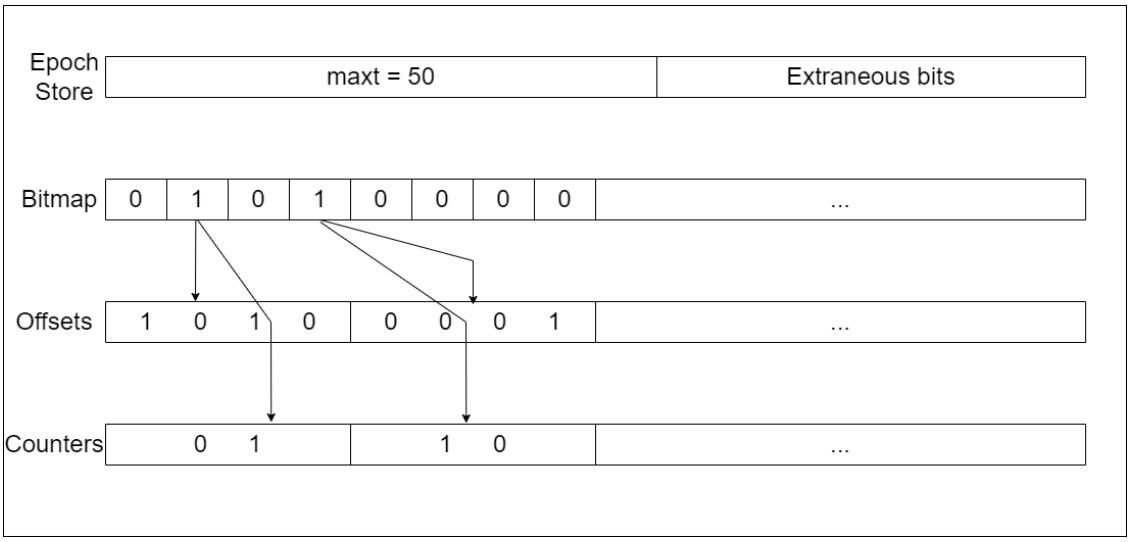


Figure 4.3 *RepCl* representation.

synchronization guarantee of $1.1ms$ will be replayed only in one order. Additionally, if e happened before f then e will always be replayed before f.

4.8 Representation of the *RepCl* and its Overhead

In this section, we identify how *RepCl* can be stored to permit efficient computation. As written, *RepCl* will require $2n + 1$ integers. However, a more compact representation will be possible when we account for the fact that it is being used in a system where the clocks are synchronized to be within $\mathcal{E}$. Thus, if j does not hear from k (directly or indirectly) for a long time then the knowledge j would have about the clock of k is the same that is provided by the clock synchronization assumption. In this case, offset.$j.k = \epsilon$. It follows that there is no need to store this value if we interpret *no information about the offset of k* to mean that offset.$j.k = \epsilon$.

With this intuition, we represent $RepCl.j$ as shown in Figure 4.3. Here, the value of $mx.j$ (represented by the first word) is 50. The next word identifies the bitmap. Since the bit corresponding to process 1 is 0, it implies that offset.$j.0 = \epsilon$. The offset for process 2 is 10 (first 4 bits of the offset) and $counter.j.2$ is 2 (first 2 bits of the counter). We note that the bits for each offset and counter are hard-coded based on the system parameters (cf. Chapter 6).

The second word is a bitmap that identifies whether offset.$j.k$ is stored for process k. Each offset is stored with a fixed number of bits in the subsequent word(s).

Next, we show that this representation allows us to reduce the cost of storage as well as the cost of computing timestamps or comparing them (using $<$ relation). Specifically, all these costs are proportional to the number of processes that have communicated with j recently.

With representation in Figure 4.3, first, we note that finding the location of the 1s in the given bitmap can be done in time that is proportional to the number of 1s in the bitmap. [$((n-(n\&(n-1)))$ will return the number with only the rightmost 1]. Thus, we have

Observation 1: Shift and MergeSameEpoch can be implemented using $O(x)$ time where x is the number of bits set to 1 in the bitmap.

Note that this means that the time to compute the timestamp for send/receive at process j is not dependent upon the number of processes in the system. But only processes that have recently communicated with process j. In turn, this means that

Observation 2: Send and Receive can be implemented using $O(x)$ time where x is the number of bits set to 1 in the bitmap.

Observation 3: Given two timestamps, $RepCl.e$ and $RepCl.f$, we can determine if $RepCl.e < RepCl.f$ in $O(x)$ time where x is the number of bits set to 1 in e and f.

It follows that the number of bits that are 1 in a given timestamp identifies not only the storage cost of the timestamps but also the time to compute these timestamps at run time. Effectively, this also identifies the overhead of the timestamps that enable the replay of the computation. Hence, in Chapter 6, we focus on identifying scenarios where the cost of storing these offsets is within the limits identified by the user.

We note that the above approach will work as long as the number of processes is less than the number of bits in a word (typically, 64 in today's systems). We expect that this will be more than sufficient for many systems in practice. If there are more than 64 processes, we expect that process j is communicating with only a subset of these processes. And, if process j does not communicate with someone there is no need to store offsets for them. Thus, this approach can be extended for

the case where the number of processes is larger. However, the details are out of the scope of this thesis.

CHAPTER 5

SIMULATOR SETUP

In this chapter, we discuss the construction of a custom discrete event simulator, to serve as a validation to state-of-the-art simulators available for research. We first design a baseline simulator build for natively supporting the *RepCl* infrastructure. We call this the Custom Discrete Event Simulator, or CDES. The goal of the *RepCl* was to serve as a plug-and-play structure that allows replays to be supported natively by virtue of the clock's algorithm. In other words, the clock should be *self-contained*, where replay should only require *RepCl*-timestamped logs to function correctly, and any simulator should be able to incorporate the visualization just by implementing the clock in its infrastructure. Details of this simulator are further discussed in Section 5.1.

However, building a simulator for the *RepCl* introduces a bias in the design. To validate the results to more real-world scenarios, we considered using a state-of-the-art (SOTA) simulator, NS-3 [10]. NS-3 proved to be an effective simulator to implement the *RepCl* structure. However, NS-3 posed challenges in making node-local noisy clocks, which is discussed further in Section 5.2. For this reason, we revise NS-3 to implement structures that would aid us in approximating what a node-local clock would look like.

Since the NS-3 team expressed the desire to enhance NS-3 with node-local noisy clocks, we decided to build a custom implementation of the same. We designed the node-local clock in a way to permit any arbitrary node level clock (e.g., HLC [11], Vector clock [6][7], Logical Clocks [5], etc.).

The following chapter is organized as follows. Section 5.1 describes the custom simulator we designed to validate the results obtained by any generic simulator, and identify key differences in the results. During our research, we chose to implement the custom simulator (CDES) first, as the choice of the SOTA simulator was not apparent. Additionally, the SOTA simulator should produce the same results as the CDES, as CDES was built solely for the *RepCl*. The CDES served as a ground truth system, and any architecture we chose would be validated against the results of this simulator. Section 5.2 talks about NS-3, and the revisions that needed to be made to implement

our clock infrastructure. We discuss the application implemented, and the complexities that the application had to handle to provide correct results.

5.1 *CDES, A Custom Discrete Event Simulator*

In this section, we detail the design of our own custom discrete event simulator (CDES). We modeled processes containing a physical clock pt and the *RepCl*. The design of the CDES is discussed below.

Each process maintained a vector msg_queue, that queued messages sent to that process. The messages contained the *RepCl* of the sending process, along with the time the message was to be processed by the receiving process. This receiving time was configured by the sending process by reading the clock of the receiver and adding the message delay to the time. When the receiving process obtained this message, it would compare its physical time pt with the receiving time, and process the message. Each process had a skewing node-local physical clock. We implemented this by randomly advancing the clock of a process based on a seed, and chose not to advance clocks. We maintained the invariant that for no two processes i and j, $|pt.i - pt.j| > \mathcal{E} * I$, same as the case in NS-3.

To test the clock, we used the same parameters for n, the number of processes, $\mathcal{E}$, I, δ and α. The message delay is modeled as the average delay experienced in the simulated network, and can vary by some nonzero Δ in production. In the simulation, at every *clock tick*, a process delivers any messages it is expected to deliver at that clock tick. It also sends a message to other processes based on the message rate α. When a message is sent, the corresponding receive event is added to the receiver's queue based on the value of δ. We also compute the actual value of the maximum clock skew observed in the simulation to ensure that if $\mathcal{E} = 1ms$ then the worst-case clock skew is indeed $1ms$. The simulation was initialized such that each process started with the same starting clock and at each microsecond time step, each process made a decision to send a message. The process first generated a random number in the range [0, 100], and if this number was lower than α, the process elected to send a message to any other process in the simulation or perform a local event. Each process in the simulation had a uniform chance to send a message with constant delay

Parameter	Minimum Value	Maximum Value	Increments
N	32 processes	64 processes	32 processes
$\mathcal{E}$	10 units	1000 units	50 units
I	100 microseconds	1000 microseconds	50 microseconds
δ	1 microsecond	8 microseconds	2x microseconds
α	10 messages/s	160 messages/s	2x messages/s

Table 5.1 **Parameter configurations for the NS-3 Simulations. We only selected the configurations where $\mathcal{E} * I$ % 1000 == 0 to give us acceptable clock skew limits of [1ms, 6ms].**

δ to any other process in the system. The total messages sent varied with alpha, within the range of [140, 10208] messages over 10,000 steps of the simulation. The parameters we varied are detailed in Table 5.1.

5.2 *NS-3 Simulator*

Network Simulator-3 (NS-3) [10] provides a generic discrete event simulator, that works on top of different devices implementing applications in different topologies. What makes NS-3 an attractive option to test clock infrastructures is its versatility in types of nodes available, its ease of use in topology configuration and application design, and most importantly, the configurability it offers in designing simulations. The NS-3 infrastructure describes a generic Simulator, that allows different network configurations to be supported and tested by changing a few parameters of this Simulator class. These factors made NS-3 an attractive option to test and incorporate the *RepCl*.

However, NS-3 posed a few challenges. NS-3, as of the time of writing this thesis, does not provide support for node-local noisy clocks. Clocks in NS-3 are synchronized with the top level Simulator, and do not contain their own clock implementations. There have been attempts in creating a node-local noisy clock, but have not been incorporated into the NS-3 infrastructure. Another challenge stems from this. Due to the absence of node-local noisy clocks, NS-3 does not handle clock drifts. Due to the absence of clock drifts, the *RepCl* would not store any offsets, as all processes would tick in sync each time with the Simulator class.

Hence, we devised an API to overcome these key challenges. We implemented a node-local noisy clock, which would approximate a node reading its physical time, and added a value δ to approximate the noise produced by skewing physical clocks. The algorithm for the node-local

noisy clock is described in Algorithm 5.1. Here ,$nt.i$ denotes the node-local time of process i.

Algorithm 5.1 Node-Local Noisy Clock: Get Operation

1. **Input:** $SimulatorTime$
2. $nt.i = random(nt.i, SimulatorTime + (\mathcal{E} * I))$
3. **Return** $nt.i$

As described in Algorithm 5.1, we receive a clock that maintains the relation that for no two processes i and j, is $|pt.i - pt.j| > \mathcal{E} * I$, but produces clocks that skew with respect to each other. This helps us produce offsets between different processes in a dynamic fashion, and allows us to handle clock drifts.

Using this node-local clock, we design an application in NS-3 called the ReplaySimulatorApplication, with nodes implementing the node-local clocks and the *RepCl*. The *RepCl* uses the local clock to perform updates on itself. The ReplaySimulatorApplication picks a random node candidate for each nodes and sends a message at intervals defined by the message rate α. The channels implemented provide a maximum data rate of 500 Mbps, and a message delay defined by δ. We also provided the option to choose $\mathcal{E}$ and I for the purposes of testing the clock sizes. In a more real-world implementation, only the I would be changeable by the user. All other parameters would be specified by the distributed system's operating constraints.

We simulated a distributed environment to test the clock with five parameters - the number of processes (n), the maximum allowed clock skew ($\mathcal{E}$), the interval size (I), the message rate (α), and the message delay in microseconds (δ). We collected results for about 20 seconds for each run. Table 5.1 defines the variation statistics of each of these parameters.

A key change we made in the NS-3 simulator is the counter storage. We store **only the sum of counters** in the counter array space. This is explained further in section 7.3.1. The key idea is that there are very few events that actually store counters for all processes, and by storing just the sum of counters of all processes helps us condense the information needed, without the loss of generality in most cases. We do risk missing a few orderings, but it is an acceptable tradeoff as the

cases where a lot of counters are stored are rare.

CHAPTER 6

SIMULATION RESULTS

As demonstrated in Section 4.6, the overhead of *RepCl* depends upon the number of offsets/counters that need to be stored. And, this value depends upon the number of processes that communicate with a given process in the $\mathcal{E}$ time. In other words, the system parameters will determine the size of *RepCl*. In this section, we evaluate the overhead of *RepCl* via simulation. For the purposes of the results, we denote the offset array size as θ and the counter array size as σ.

In the following sections, we outline the effects of changing different parameters both in the custom simulator and in NS-3. Note that the results for the CDES are reported in 32-bit word lengths, and the results for NS-3 are reported in bits, due to the different data collection techniques used for each.

6.1 Effect of Clock Skew ($\mathcal{E}$)

In this section, we measure the trends in $\mathcal{E}$ while varying the other parameters to see how the θ and σ are affected. We compare each parameter pair-wise with $\mathcal{E}$ to see the effect of the parameter on the clock skew trend with θ.

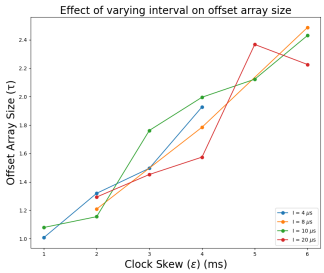
6.1.1 Analysis of Varying Interval Size (*I*) with the CDES

Here, we keep the δ and α constant to see how θ and σ change with $\mathcal{E}$.

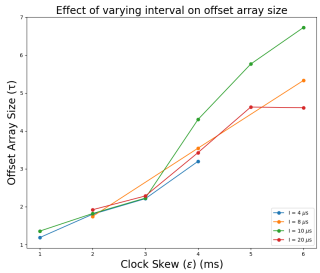
- In case of θ vs $\mathcal{E}$ curve, we notice that the value of I has little bearing on θ. As expected, as the value of $\mathcal{E}$ increases, θ increases with it. This is true for all values of α, and we consistently store more offsets as α increases. For any given value of α, however, the value of I can be chosen to set the granularity of the user's choice and would allow more flexibility in the clock information. Regardless of the choice of I by the user, the offset sizes increase with roughly the same trend. These trends are illustrated in Figure 6.1.

- In the case of σ vs $\mathcal{E}$ curve, we see not too much of a variation in σ, as most events reach a different epoch. On average, we do not see many events storing counters; roughly 0.78% of events store counters, and the values of such counters do not exceed 5 in most cases. Hence,

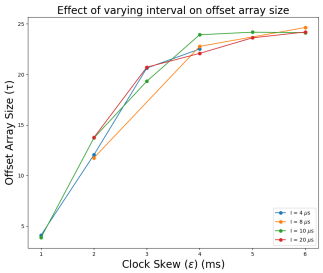
we would need very little space to store these counters. These trends are illustrated in Figure 6.2. Since this observation is true for all simulations in this paper, we do not discuss the analysis for σ in the subsequent sections.



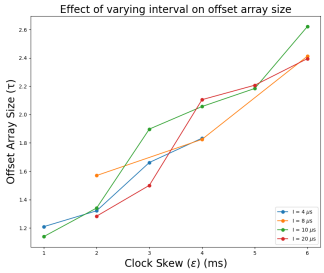
(a) α = 20 msgs/s, n = 32.



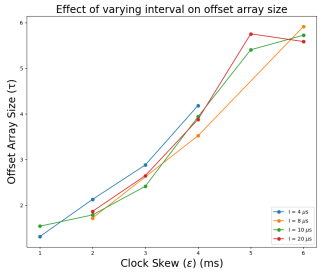
(b) α = 40 msgs/s, n = 32.



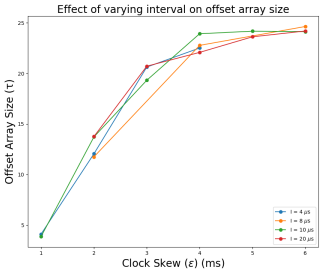
(c) α = 160 msgs/s, n = 32.



(d) α = 20 msgs/s, n = 64.

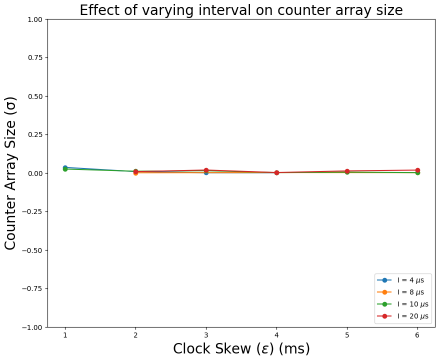


(e) α = 40 msgs/s, n= 64.

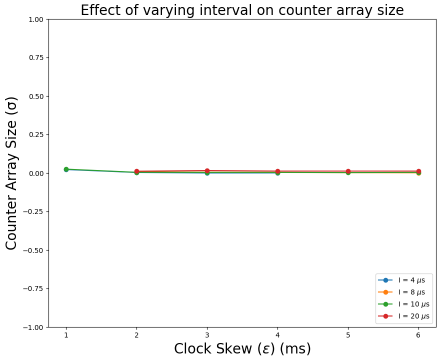


(f) α = 160 msgs/s, n = 64.

Figure 6.1 Custom Simulator: θ vs $\mathcal{E}$ when varying I, $\delta = 8\mu s$.



(a) n = 32.



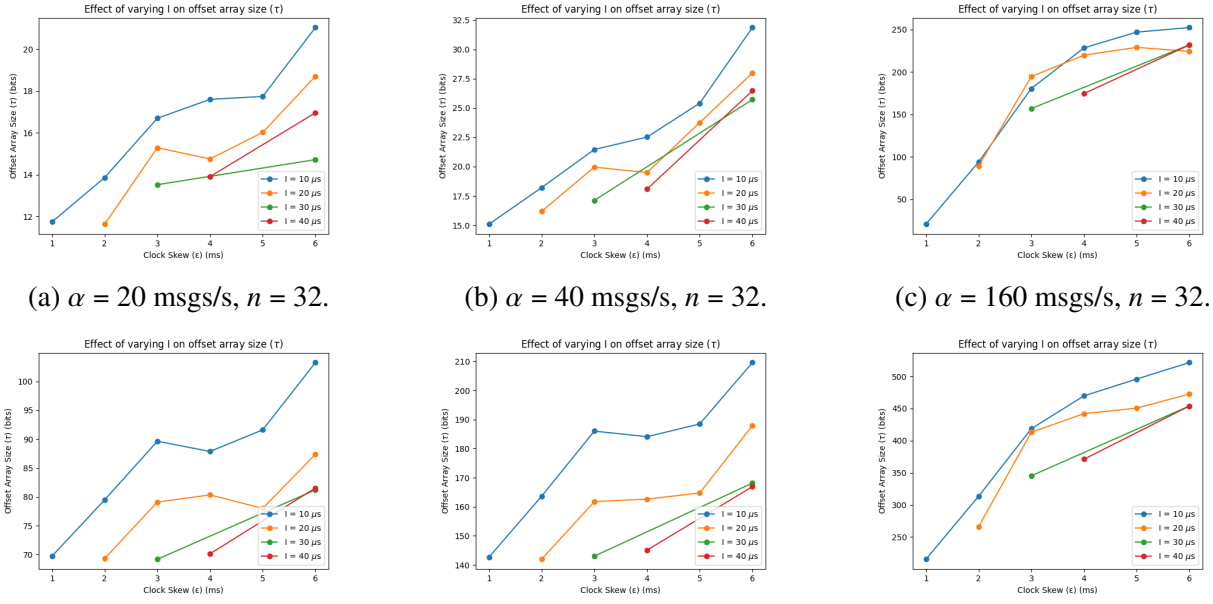
(b) n = 64.

Figure 6.2 Custom Simulator: σ vs $\mathcal{E}$ when varying I, δ = 8 μs, α = 160 msgs/s.

6.1.2 Analysis of Varying Interval Size (I) with NS-3

- In the case of the θ vs $\mathcal{E}$ curve in the NS-3 simulation, we observe a similar trend of θ increasing with $\mathcal{E}$. This is in agreement with our findings from the custom simulator, and true for all values of α. These trends are illustrated in Figure 6.3. While we see a decrease in number of bits stored as I decreases, the difference is only significant in some cases, notably in the case of lower message rates. At higher message rates, the gap reduces. This is due to the amount of communication happening, and most processes tend to store close to the acceptable limit of the number of offsets we want to store.

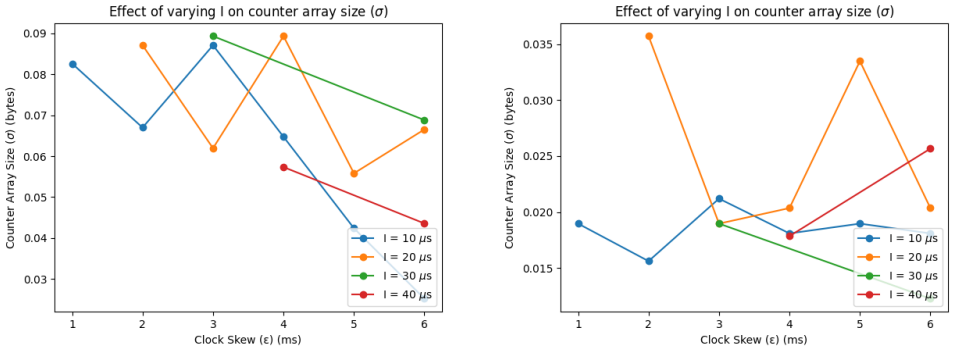
- In the case of the σ vs $\mathcal{E}$ curve in the NS-3 simulation, we store only the sum of counters. Hence, we see some variations in counter size. With the increase of epsilon, we store slightly higher counters, and this does not change with variation in I. However, the sizes of the counters vary only from 0.03 bytes in the lowest case of n = 32 to 0.1 bytes in the highest case. Hence, the number stored as the counter value is not too large. This is true even for the case of n = 64, and is depicted in Figure 6.4.



(a) α = 20 msgs/s, n = 32. (b) α = 40 msgs/s, n = 32. (c) α = 160 msgs/s, n = 32.

(d) α = 20 msgs/s, n = 64. (e) α = 40 msgs/s, n= 64. (f) α = 160 msgs/s, n = 64.

Figure 6.3 NS-3 Simulator: θ vs $\mathcal{E}$ when varying I, $\delta = 1\mu s$.



(a) $n = 32$. (b) $n = 64$.

Figure 6.4 NS-3 Simulator: σ vs $\mathcal{E}$ when varying I, $\delta = 1\ \mu s$, $\alpha = 20$ msgs/s.

Since the total size of *RepCl* depends upon the number of bits for each offset and the total number of offsets, we consider a specific example here. For Figure 4.3, the number of offsets is 2 and the size of each offset is 4 bits. Therefore, one word is sufficient to store offsets. Likewise, one word is enough for counters. Thus, we need a total of 4 words to store this timestamp.

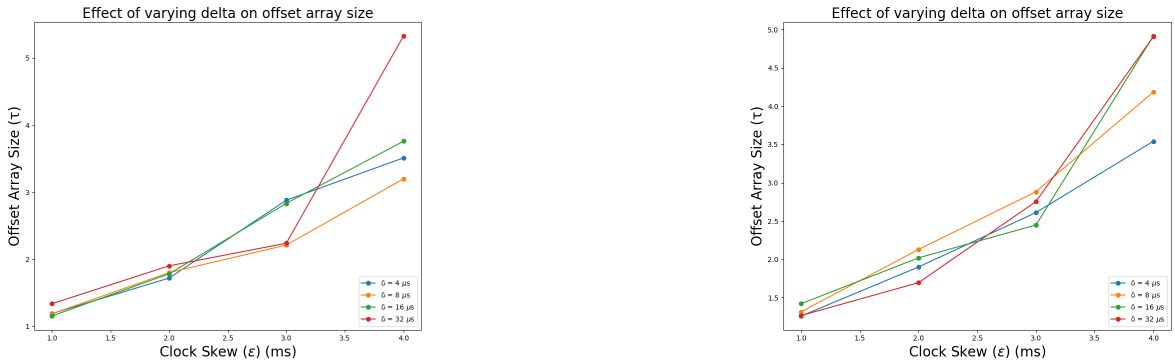
(Note that the counters can be stored in the same amount of memory as we have a number of extraneous bits in this representation, specifically in the max epoch word if we elect to store the sum of all these counters here. By doing this, we would lose some information, but considering that the number of events that record meaningful counters is low, this may be an acceptable trade-off.) It is straightforward to observe that the size of *RepCl* grows linearly with the number of offsets in it. And, the total size of *RepCl* will require the use of the floor function to identify the number of words necessary to store it. Since the floor operation loses some of the relevant data, we present the value of θ in this section.

6.1.3 Analysis of Varying Message Delay (δ) for the CDES

Here, we fix the I and α, and for different δ values, and we identify how θ changes with $\mathcal{E}$.

As $\mathcal{E}$ increases, we see higher values of θ, implying a higher number of offsets stored on average. We observe that higher values of δ produce a lower number of offsets in each case, barring some noise. This is expected as an increase in δ implies messages would reach in a delayed fashion, and would lead to processes setting other process offsets to ϵ due to non-receipt of messages. As the $\mathcal{E}$ increases, a process hears from more processes (directly or indirectly) within time $\mathcal{E}$. Hence, the

number of offsets increases. This is illustrated in Figure 6.5.



(a) $n = 32$.

(b) $n = 64$.

Figure 6.5 Custom Simulator: θ vs $\mathcal{E}$ when varying δ, $I = 8\ \mu s$, $\alpha = 40$ msgs/second.

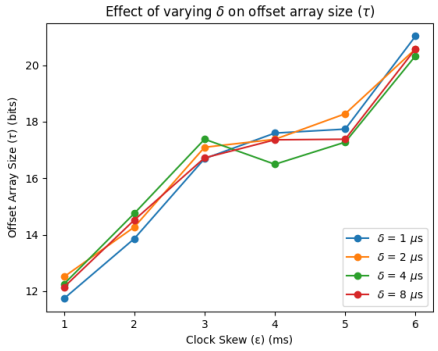
6.1.4 Analysis of Varying Message Delay (δ) for the NS-3

As in the case of the Custom simulator, we see higher values of θ as the $\mathcal{E}$ increases. We do not see too much of a difference as δ varies however, where the number of bits stored for each δ value are between $\pm$ 1 bit. The increase in offset size is somewhat linear for the most part as the $\mathcal{E}$ increases, which is the same as observed in the analysis of varying I. The δ variations are not pronounced as much due to the fact that the clocks implicitly synchronize when messages are sent between each other. A message sent from far back into the past effectively does not change the clock of the receiver. If a sender gets a clock from the future (in its local observation), it modifies its own clock to push forward to this future timestamp to guarantee the acceptable $\mathcal{E}$ limit. This allows different δ values to show close to no variation. This is illustrated in Figure 6.6. It is important to note, however, that when δ exceeds the $\mathcal{E}$ limit, no process stores any offsets.

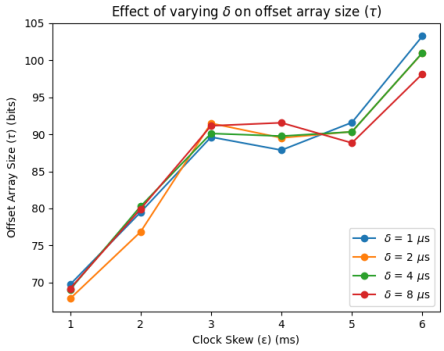
6.1.5 Analysis of Varying Message Rate (α) for the CDES

Here, we fix the I and δ, and for different α values, we identify how θ changes with $\mathcal{E}$.

As expected, for lower values of α, we consistently store lower offsets, as communication between processes is sporadic. As the $\mathcal{E}$ increases, θ increases linearly until the bound of n is reached. This is due to the same reason mentioned earlier, as the bound lengths on epochs are larger, even sporadic messages tend to store more offsets on other processes, causing the overall



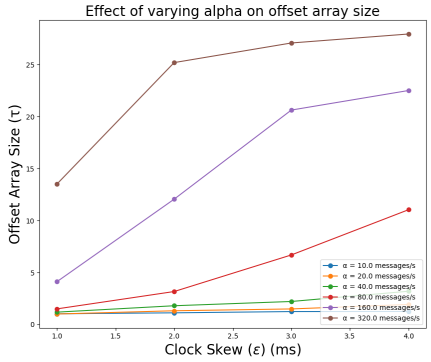
(a) $n = 32$.



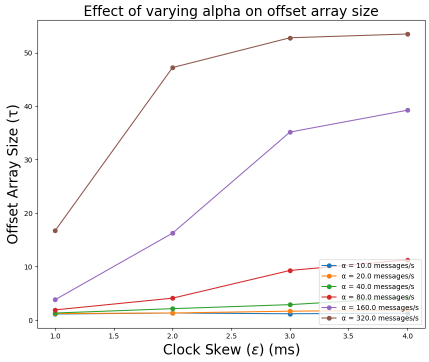
(b) $n = 64$.

Figure 6.6 NS-3 Simulator: θ vs $\mathcal{E}$ when varying δ, $I = 20\ \mu s$, $\alpha = 160$ msgs/second.

value of θ to increase. This is illustrated in Figure 6.7.



(a) $n = 32$.



(b) $n = 64$.

Figure 6.7 Custom Simulator: θ vs $\mathcal{E}$ when varying α, $I = 4\ \mu s$, $\delta = 8\ \mu s$.

6.1.6 Analysis of Varying Message Rate (α) for NS-3

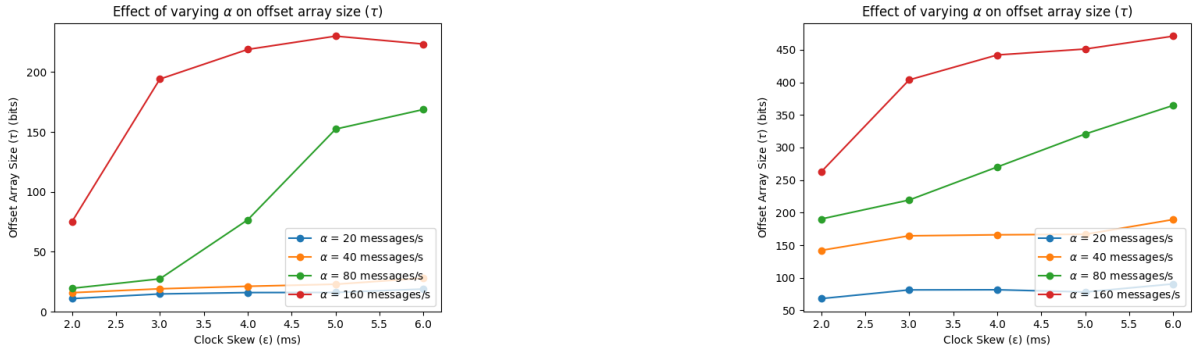
Our results from the custom simulator are confirmed by the experiments in NS-3, where lower values of α store lower offsets due to lower communication. In the case of NS-3, higher values of α store many more offsets than what we would like our upper limit to be (about one word of offsets stored per clock). This is guaranteed by having lower values of α. This is illustrated in Figure 6.8.

6.2 Effect of Interval Size(I)

In this section, we observe the trends in I with respect to δ and α.

6.2.1 Analysis of Varying Message Delay (δ) for the CDES

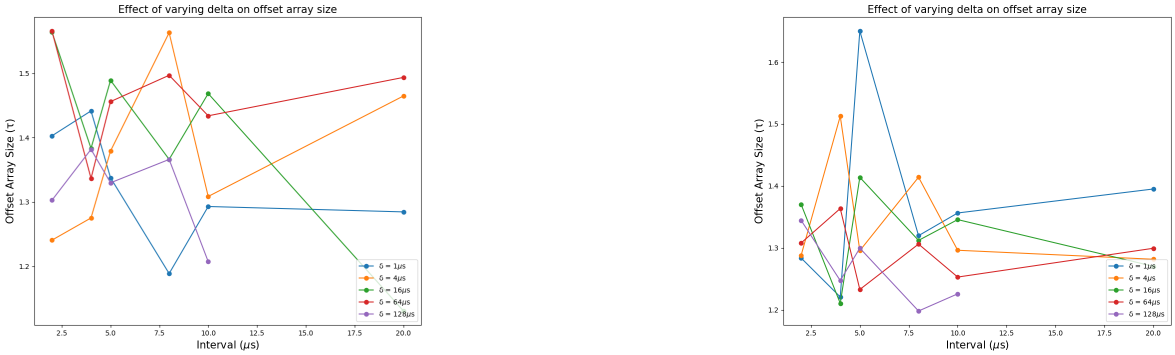
Here, we fix the $\mathcal{E}$ and α and check how θ changes with I.



(a) $n = 32$. (b) $n = 64$.

Figure 6.8 NS-3 Simulator: θ vs $\mathcal{E}$ when varying α, $I = 20\ \mu s$, $\delta = 4\ ms$.

From Figure 6.9, we observe that the value of I does not really have a significant effect on θ (Note that the Y axis of this figure varies only from 1.2 to 1.5.) This means that the selection of I does not affect the number of offsets maintained by a process. However, it affects the size of each offset. Specifically, the max value of the offset is $\epsilon = \frac{\mathcal{E}}{I}$ and the number of bits required for each offset is $\log_2 \epsilon$. Hence, a larger value of I is better for reducing the size of the *RepCl*. However, with larger I, the guarantees provided by *RepCl* are lower. Specifically, Lemma 3 shows that some unforced reordering may occur when events e and f differ by time $\mathcal{E} + I$. Users should therefore choose the value of I based on the desired guarantees of *RepCl* or the maximum desirable offset.

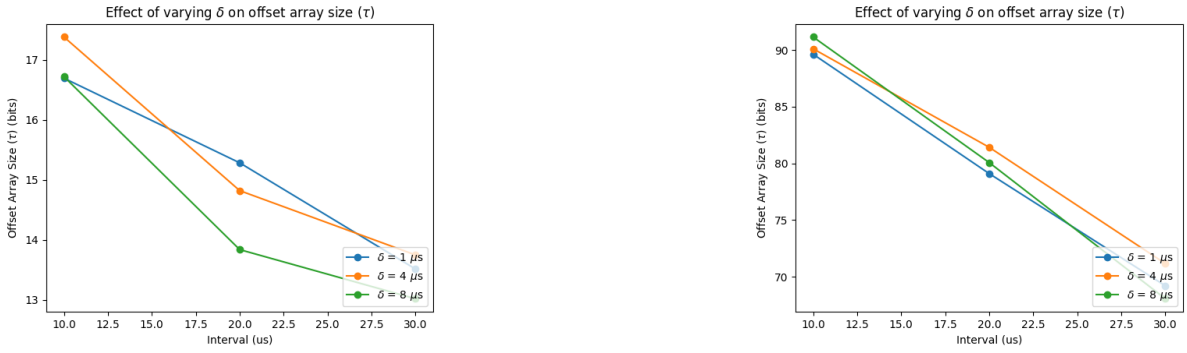


(a) $n = 32$. (b) $n = 64$.

Figure 6.9 Custom Simulator: θ vs I when varying δ, $\mathcal{E} = 2$ ms, $\alpha = 20$ msgs/s.

6.2.2 Analysis of Varying Message Delay (δ) for NS-3

In the case of the NS-3 simulator, we observed that the size of offsets decreased linearly with increase in I. This is attributed to more information being stored in counters, as the length of the interval increases, and less offsets being stored. The likelihood that all processes are in the same epoch increases as the I increases, leading to lower number of offsets stored. The variation with δ is negligible, and seemingly random, which is confirmed with the custom simulator results.



(a) $n = 32$. (b) $n = 64$.

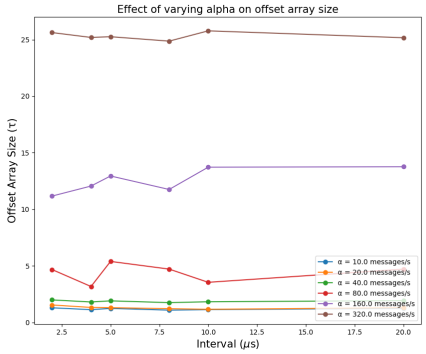
Figure 6.10 NS-3 Simulator: θ vs I when varying δ, $\mathcal{E}$ = 3 ms, α = 20 msgs/s.

6.2.3 Analysis of Varying Message Rate (α) for the CDES

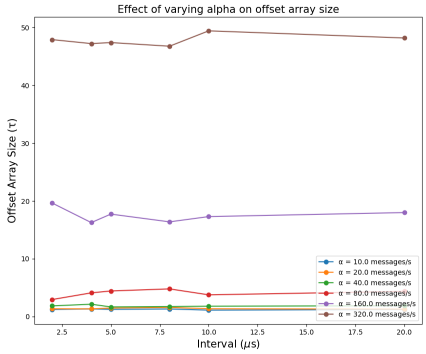
For every point in the $I - \theta$ trend, lower α values produce lower θ. This is consistent with our observations so far as communication is infrequent and processes tend to not hear from other processes, subsequently not storing their offsets. As I increases, the θ remains the same, as the δ remains the same. When δ is constant, and $\mathcal{E}$ is constant, messages being sent either remain in the same epoch (mx) as they would in the previous I chosen, or a message that was between different intervals in a lower I, now would be in the same interval. Overall, this does not change the total θ. This is illustrated in Figure 6.11.

6.2.4 Analysis of Varying Message Rate (α) for NS-3

As observed in the custom simulator, lower α values produce lower θ, due to infrequency in communication. We additionally observe that the θ remains constant with increasing I, consistent with our results from the CDES simulator in the previous section. This is illustrated in Figure 6.12.

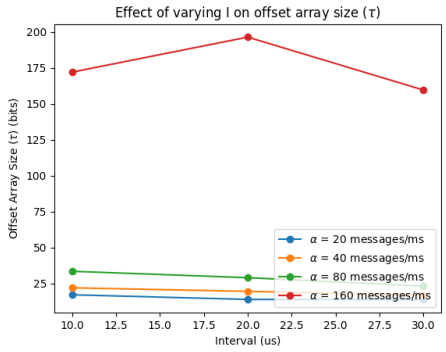


(a) $n = 32$.

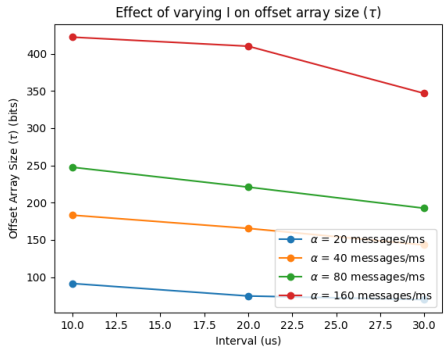


(b) $n = 64$.

Figure 6.11 Custom Simulator: θ vs I when varying α, $\mathcal{E} = 2$ ms, $\delta = 8\ \mu s$.



(a) $n = 32$.



(b) $n = 64$.

Figure 6.12 NS-3 Simulator: θ vs I when varying α, $\mathcal{E} = 3\ ms$, $\delta = 2\ ms$.

6.3 Effect of Message Delay(δ)

In this section, we observe the effect of δ on θ while fixing $\mathcal{E}$ and I.

6.3.1 Analysis of Varying Message Delay (δ) for the CDES

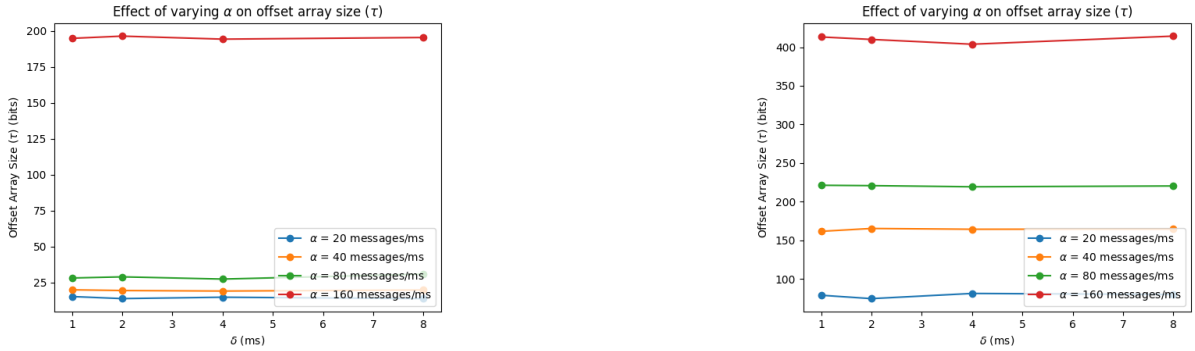
Here, we observe that the value of δ has minimal effect on θ. Specifically, as shown in Figure 6.13, we observe that the value of θ increases as the value of α increases. However, for a fixed value of α, the θ remains the same. When the $\mathcal{E}$ is fixed, and the I is fixed, the only way the θ would go down is when processes would send messages that were received after the $\mathcal{E}$ limit. Because we enforce the limit as $\delta \leq \mathcal{E}$, this limit is never exceeded, and the θ hence, remains the same. This is illustrated in Figure 6.13.

(a) $n = 32$. (b) $n = 64$.

Figure 6.13 Custom Simulator: θ vs δ when varying α, $\mathcal{E} = 4$ ms, $I = 16\ \mu s$.

6.3.2 Analysis of Varying Message Delay (δ) for NS-3

We gather similar results in the case of NS-3 as we did in the CDES, which validates our observations. This is illustrated in Figure 6.14. We also notice that for higher values of α, the θ increases drastically. Hence, it is important to note the feasibility of the *RepCl*, mainly that it is advantageous to use it in a setting of low α (message rates). This is covered further in the following section.



(a) $n = 32$. (b) $n = 64$.

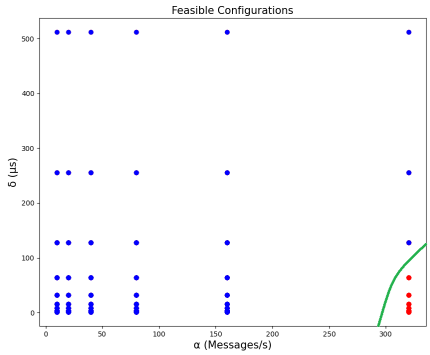
Figure 6.14 NS-3 Simulator: θ vs δ when varying α, $\mathcal{E} = 3$ ms, $I = 20\ \mu s$.

6.4 Feasibility Regions

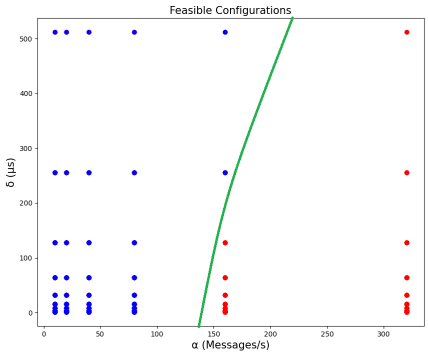
In this section, we review the simulations to define the notion of feasible regions. As discussed earlier, the goal of *RepCl* is to enable the replay of a distributed computation with a small overhead. Here, we consider the case where the user identifies the expected overhead of *RepCl* to identify

scenarios under which *RepCl* can be used to provide a *perfect-replay* that meets all the requirements from Section 4.6. Since the overhead of the counters remains virtually unchanged, we only focus on the overhead of the number of offsets, i.e., the value of θ.

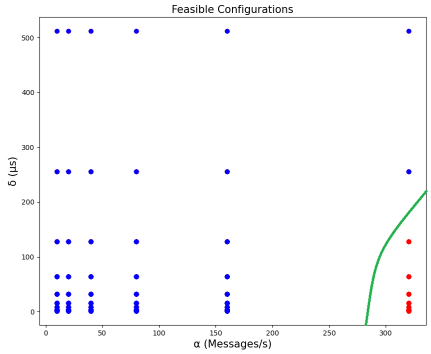
For $\theta = 8$, the feasibility regions are shown in Figure 6.15a. Here, the blue dots identify the data points where $\theta = 8$ is feasible and the red dots represent the data points where $\theta = 8$ is not feasible. The green line identifies the bounds where $\theta = 8$ is feasible. We find that the size of the feasible region remains fairly unchanged with the value of n. However, it shrinks when the value of $\mathcal{E}$ is increased. This is expected based on how θ changes with $\mathcal{E}$. We note that the feasibility region only identifies the case where *perfect-replay* meets all the requirements from Chapter 4.6. If the user needs to utilize *RepCl* in an infeasible region, the user can obtain *partial-replay*. To understand this, consider the case where the actual value of $\mathcal{E}$ is $4ms$ but the user specifies it to be $2ms$ while constructing *RepCl*. In this case, if e and f are within $2ms$ then *RepCl* will allow them to be replayed in any order. However, if f occurred $3ms$ after e then e will always be replayed before f. We anticipate that even in a system where the clock skew is $4ms$, the actual clock skew at a given moment is likely to be smaller than $4ms$. This implies that the forced order between e and f will be quite infrequent. Hence, we anticipate that *RepCl* will be applicable even in domains where the system parameters cause it to fall in an infeasible region, and is referred to in Section 8.3.



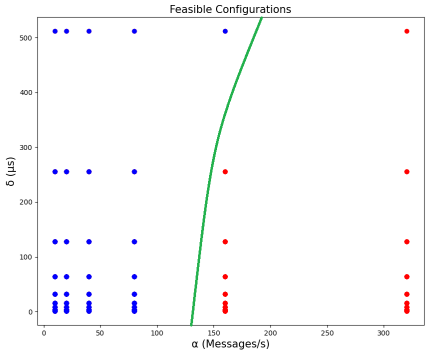
(a) $\mathcal{E}$ = 1 ms, n = 32, and θ = 8.



(b) $\mathcal{E}$ = 2 ms, n = 32, and θ = 8.



(c) $\mathcal{E}$ = 1 ms, n = 64, and θ = 8.



(d) $\mathcal{E}$ = 2 ms, n = 64, and θ = 8.

Figure 6.15 Feasibility regions for α and δ settings.

CHAPTER 7

VISUALIZING TRACES WITH *REPVIZ*

We have now seen the various intricacies of the *RepCl* infrastructure, and we now describe how to apply it to visualization systems. The goal of a visualization is simple, to show the user a view of the computation both from an overall system perspective and a per-process view of how the computation looked on that specific node. Various implementations of visualizers (ref. Section 8.2) achieve this, but in all of them, the underlying timestamping algorithm enforces an order between concurrent events. We therefore, need a visualizer that allows the user to choose the order of concurrent events, and view multiple execution traces simultaneously.

In this Chapter, we introduce *RepViz*, the third infrastructure component of this work. *RepViz* is a visualizer that works on top of a log generated by the *RepCl*. It takes in a *RepCl*-timestamped log, and generates a web-based visualization of the traces the algorithm generates. The user has the option to replay events that are concurrent in any order, while the other events are ordered according to the *RepCl*. Once the user has selected a replay order, a web visualization is displayed for that trace. The web visualization is under progress, but a sample representation is depicted in Figure 7.1. In the following sections, we will go over the different methods that went into implementing *RepViz*.

7.1 Implementation

The visualizer defines a top level component, called *Tracer*. This component contains the following functions amd members:

- SortEvents(): This function sorts all events according to their *RepCl* timestamp.
- RunReplay(): The top level function that provides an interactive view to the user to replay events.
- EventList: The list of events obtained from a *RepCl* timestamped log.

A *Tracer* object contains a set of *Event* objects. Each *Event* object has the following properties:

- EventID: The Message ID.
- EventType: The type of event, i.e., Send, Local or Recv.
- EventTime: The*RepCl* timestamped to this event.
- Sender: The IP address of the sender.
- Receiver: The IP address of the receiver.

The Sender and Receiver fields are populated as the (Sender, Receiver) when its a Send/Local event, and as (Receiver, Sender) for a Recv event.

Now, we will go into detail on how each of the *Tracer* methods are implemented.

7.1.1 SortEvents

The *Tracer* first orders all the events according to the *RepCl* timestamp. Events are sorted based on their *RepCl* timestamps fed by the logs of the algorithm. The sorting algorithm sorts all timestamps by the happens-before (hb) relation discussed in Chapter 2. Once the ordering is set, the events are then given to the matching function. The sorting rules are described in Algorithm 4.11. The algorithm compares two *RepCl* timestamps $t1$ and $t2$, and returns True if $t1 < t2$ and False otherwise. The other comparisons can be implicitly derived from the same algorithm.

7.1.2 RunReplay

Once the events are ordered, the *Tracer* runs the replay according to the event timeline generated by the sorter. Every concurrent event pool is given to the user as a choice to replay any one of the outstanding events waiting to be replayed. Once an event has been replayed, that event is removed from the replay pool. This follows Algorithm 3.1.

7.2 User View

In the current iteration of *RepViz*, the prototype is run on an ASCII terminal. Here is a brief run output of a sample trace snippet generated through an NS-3 simulation:

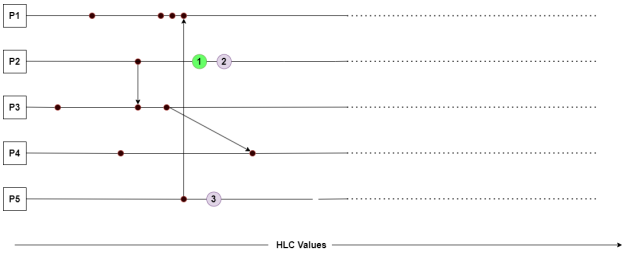
```
[(EventID=1, EventType=SEND, EventTime=[(NodeId=10.1.1.3, HLC=21, Offsets=[-15,
    -15, 0, -15, -15], Counters=0)], Sender=10.1.1.3, Receiver=10.1.1.4)]
```

```
[(EventID=2, EventType=SEND, EventTime=[(NodeId=10.1.1.1, HLC=42, Offsets=[0,
    -15, -15, -15, -15], Counters=0)], Sender=10.1.1.1, Receiver=10.1.1.2)]
[(EventID=3, EventType=SEND, EventTime=[(NodeId=10.1.1.4, HLC=44, Offsets=[-15,
    -15, -15, 0, -15], Counters=0)], Sender=10.1.1.4, Receiver=10.1.1.5)]
[(EventID=4, EventType=SEND, EventTime=[(NodeId=10.1.1.2, HLC=55, Offsets=[-15,
    0, -15, -15, -15], Counters=0)], Sender=10.1.1.2, Receiver=10.1.1.3)]
[(EventID=4, EventType=RECV, EventTime=[(NodeId=10.1.1.3, HLC=55, Offsets=[-15,
    0, 0, -15, -15], Counters=1)], Sender=10.1.1.3, Receiver=10.1.1.2)]
[(EventID=5, EventType=SEND, EventTime=[(NodeId=10.1.1.1, HLC=57, Offsets=[0,
    -15, -15, -15, -15], Counters=0)], Sender=10.1.1.1, Receiver=10.1.1.2)]
[(EventID=6, EventType=SEND, EventTime=[(NodeId=10.1.1.3, HLC=59, Offsets=[-15,
    4, 0, -15, -15], Counters=0)], Sender=10.1.1.3, Receiver=10.1.1.4)]
[(EventID=7, EventType=SEND, EventTime=[(NodeId=10.1.1.5, HLC=61, Offsets=[-15,
    -15, -15, -15, 0], Counters=0)], Sender=10.1.1.5, Receiver=10.1.1.1)]
[(EventID=7, EventType=RECV, EventTime=[(NodeId=10.1.1.1, HLC=61, Offsets=[0,
    -15, -15, -15, 0], Counters=0)], Sender=10.1.1.1, Receiver=10.1.1.5)]
[(EventID=8, EventType=SEND, EventTime=[(NodeId=10.1.1.1, HLC=61, Offsets=[0,
    -15, -15, -15, 0], Counters=1)], Sender=10.1.1.1, Receiver=10.1.1.2)]
Concurrent events detected!
0. [(EventID=9, EventType=SEND, EventTime=[(NodeId=10.1.1.2, HLC=62,
    Offsets=[-15, 0, -15, -15, -15], Counters=0)], Sender=10.1.1.2,
    Receiver=10.1.1.3)]
1. [(EventID=10, EventType=SEND, EventTime=[(NodeId=10.1.1.5, HLC=62,
    Offsets=[-15, -15, -15, -15, 0], Counters=0)], Sender=10.1.1.5,
    Receiver=10.1.1.1)]
2. [(EventID=2, EventType=RECV, EventTime=[(NodeId=10.1.1.2, HLC=62, Offsets=[0,
    0, -15, -15, -15], Counters=0)], Sender=10.1.1.2, Receiver=10.1.1.1)]
Please choose the event to replay: 0
[(EventID=9, EventType=SEND, EventTime=[(NodeId=10.1.1.2, HLC=62, Offsets=[-15,
```

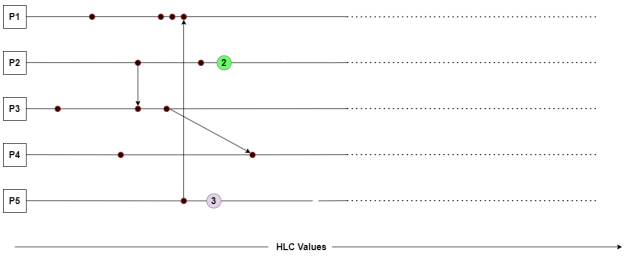
```
    0, -15, -15, -15], Counters=0)], Sender=10.1.1.2, Receiver=10.1.1.3)]
Please choose the event to replay: 2
[(EventID=2, EventType=RECV, EventTime=[(NodeId=10.1.1.2, HLC=62, Offsets=[-15,
    0, -15, -15, -15], Counters=1)], Sender=10.1.1.2, Receiver=10.1.1.1)]
Please choose the event to replay: 1
[(EventID=10, EventType=SEND, EventTime=[(NodeId=10.1.1.5, HLC=62, Offsets=[-15,
    -15, -15, -15, 0], Counters=0)], Sender=10.1.1.5, Receiver=10.1.1.1)]
```

The above log would transform to the visualization on a WebUI as shown in Figure 7.1. In Figure 7.1a, the user does not have a choice in the replay, as all events are ordered with the *hb* relation detailed in Section 4.5 in the absence of concurrency. The user simply taps the right arrow key to move forward in the replay. At some point in the replay, the user encounters concurrent events, depicted in Figure 7.1b. Here, the user elects to replay event 1 by providing an input of 1 through the keyboard. The event marked 1 is added to the replay. Next the user has to choose between events 2 and 3, depicted in Figure 7.1c. The user chooses event 2, and the last event left to replay is event 3, which is replayed in Figure 7.1d. With this visualization, it is easy for a user to try different combinations of replay and analyse how parameters change with the event order chosen. The visualization can also generate exhaustive logs of all possible replays should the user require it.

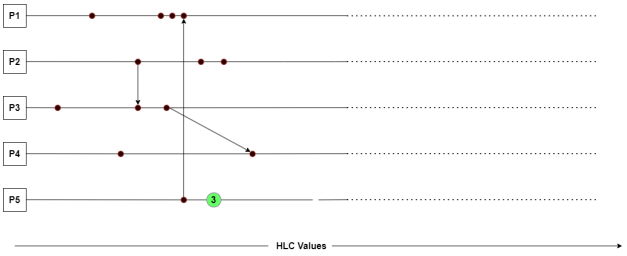
(a) No concurrency conflicts, push the right arrow key.



(b) Concurrency conflicts detected, replay event 1 first.



(c) Event 1 replayed, replay event 2 next.



(d) Event 2 replayed, replay event 3 next.

Figure 7.1 Sample Visualization: Here the user uses arrow keys to replay events that do not have concurrency conflicts, and then uses number keys to input which events to replay in which order.

CHAPTER 8

RELATED WORK AND DISCUSSION

8.1 Clocks in Distributed Systems

Logical clocks were proposed in 1978 by Leslie Lamport [3] to trace the ordering of events in a distributed system. Vector time was designed independently by multiple researchers [12][7][13], and they proposed the idea of representing time in a distributed system as a set of n-dimensional non-negative integer vectors. According to [14], Vector clocks are defined by three properties: Isomorphism, Strong consistency, and Event Counting. Isomorphism suggests that if two events x and y have timestamps vh and vk, respectively, then $x \longrightarrow y \Longleftrightarrow vh < vk$. Here, $\longrightarrow$ implies a partial ordering between a set of events. Strong consistency implies that by examining the vector timestamp of two events, we can determine the causal relationship between the two events. Event counting suggests that if d is always 1 in the rule $R1$, then the i^{th} component of the vector clock at process p_i, $vt_i[i]$, denotes the number of events that have occurred at p_i until that instant.

There have been several prior implementations of vector clocks including Singhal-Kshemkalyani's differential technique [15] and Fowler-Zwaenepoel direct dependency technique [16]. While vector clocks are upper bounded by $O(n)$ complexity in terms of both time and memory complexity, the different implementations of the past have tried to reduce this complexity and generate more efficient representations, with some success. Singham-Kshemkalyani's differential technique relied on piggybacking using the last sent and last update, without updating every vector clock. This method relies on the assumption that even though the number of processes is large, only a few key processes in a system would interact frequently by passing messages. A benefit of this method is that it cuts down storage overhead at each process to $O(n)$. However, this method doesn't make a substantial contribution to reducing the time complexity incurred when updating the vector clock, as it relies on piggybacking to work.

Fowler-Zwaenepoel's direct-dependency technique cuts down storage complexity again by reducing the message size during transmission by transmitting only a scalar value in the messages. Here, a process only maintains information regarding direct dependencies on other processes. The

downside of this method is that it has a high computational overhead as it has to trace dependencies and update the vector clock, especially in systems where a few key processes may have a large number of events.

Clock synchronization using Network Time Protocol (NTP) uses the Offset Delay Estimation method to ensure physical clocks are synchronized across the internet. Clock offsets and delays are calculated, and timestamps are issued between different machines within a system accordingly. The system then attempts to establish a causal relationship by using the corrected timestamps. This, however, can be computationally expensive and is open to error, as the delay estimation may not always be accurate and result in a violation of the causal relationship between processes issued by different machines.

One existing limitation between vector clocks representing logical time and physical clock synchronization is the difficulty in reconciling one with the other. To overcome this challenge, Hybrid Logical clocks were introduced by Kulkarni et al. [4] to capture the causality relationship of a logical clock with the characteristics of a physical clock embedded into it. Another variant of the hybrid clock is the Hybrid Vector Clock [17], [1], which, unlike the Hybrid Logical Clock, can provide all possible/potential consistent snapshots for a given time. For this experiment, we use the Hybrid Vector Clock design presented in Yingchareonthawornchai et al. as it provides desirable characteristics to build our visualization framework.

8.2 Visualizing Traces

Mattern [7] talks about how distributed systems use the concept of global state to communicate information, and the need to characterize this global state. They talk about how a process can only approximate the global view of the system, and no process can have, at any given instant, a consistent view of the global state. To verify a distributed system, the author provides a comparison between three key approaches: simulating a synchronous distributed system given an asynchronous system, simulating a common clock or simulating a global state. They highlight the need of a vector clock system to provide a consistent snapshot of the global state, as each process having a clock that stores only its own state is not enough to describe the global state of the computation.

PARAVER [18] uses the PVM message passing library to analyze traces generated from a computation. PVM primarily uses parallel message passing, and PARAVER analyses these parallel traces using data analytics and provides a graphical description of the analysis. This was one of the earliest visualization works on distributed systems simulated only for parallel traces. It used a parser to parse through the logs of the PVM-generated traces and analyze CPU activity, communications, and user events. This however required the addition of functionality to the PVM itself and was not generalized to any distributed system interface. It also did not provide generic support and incurred a larger overhead to profile system resources while computation went on.

VAMPIR [12] provided analysis of MPI programs by generating timeline traces by profiling MPI applications. It used different visualization metrics to show whether processes were still active or not. It also provided views of system activities and aggregated statistics about the system itself. However, it was made specifically only for MPI applications and added to the profiling interfaces of MPI.

D3S [19] allowed developers to specify predicates on distributed properties of the system. These predicates can vary depending on what consistency checks one would require on the distributed system. They modeled the tracing as a consistency checker and generated traces of predicate evaluation. The predicates are injected dynamically at compile time into the system and are evaluated based on the customization provided by the user. However, we believe this approach would add overhead to running the distributed computation due to complex predicate checking.

Zinsight [20] provides hierarchies of tasks and provides aggregated metrics to show timeline visualizations of events. It also provides users with changing the granularity of the metric they want to see with sequences of computations per process.

Trumper et al. [21] present a dynamic analysis tool that uses boundary tracing and post-processing to analyze system behavior through a distributed computation. These are task-based visualizations, where tasks are mapped to memory resources. However, this may not always be the case where processes share the same memory, as in the case of OpenMP-based infrastructures.

Dapper [22] is Google's tracing software for distributed systems where they provided low

overhead, application-level transparency, and scalability. Dapper uses annotations and *spans* to generate traces through RPCs. However, the authors mention that Dapper cannot correctly point to causal history, as it uses annotations in non-standard control primitives, that may mislead the causality calculations. Our approach would overcome this, as causality is enforced through a lattice of clocks, rather than the events itself.

Isaacs et al. [23] provides a comprehensive survey of different distributed monitoring and tracing tools in the past decade, providing detailed descriptions and categorizations based on task parallelism, causality information, and so on.

Isaacs, Bremer et al. [24] design a trace visualization system purely relying on logical clocks and then transposing those clocks back to real-time clocks in the visualization. Processes are also clustered based on logical behavior. However, this would incur more overhead than our solution and may cause conflicts in enforcing causality, due to the usage of a standard logical clock.

Verdi [25] provides developers with choosing the fault system to diagnose, and verify the implementation of the system. This is a formal verification system where it provides the developer with an idealized fault model, and once this is verified, it applies the correctness to a more realistic fault model.

ShiViz [26] uses vector clocks in generating distributed system traces using happens-before relationships. By using vector clocks, it provides a verifiable and accurate notion of causality. However, since it uses traditional vector clocks, it uses a higher complexity than our proposed model.

8.3 Discussion

In Section 6, we identified feasible regions for the given permissible overhead for *RepCl*. Thus, the natural question is: *what can a user do if the given system parameters fall into the infeasible region?* Here, observe that $\mathcal{E}$ provides one way to reduce the overhead if we accept some imperfect replay. To explain this, consider the case where we are using a system with clock skew to be $\mathcal{E}_a$. If the user implements *RepCl* with $\mathcal{E} < \mathcal{E}_a$ then the resulting replay will still satisfy requirements 1 and 2 (cf. Section 4.6). Requirement 3 will be satisfied with $\mathcal{E}_2 = \mathcal{E} - I$.

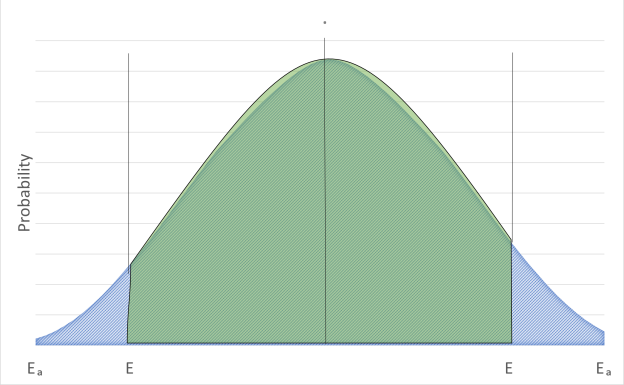


Figure 8.1 Effect of using $\mathcal{E}$ instead of $\mathcal{E}_a$ in *RepCl*.

Looking at this situation closely, we observe that the clock skew between two processes follows a structure shown in Figure 8.1. Specifically, at a given instance, the clocks of two processes j and k differ by some amount that is less than $\mathcal{E}_a$. However, the actual clock difference at a fixed point in time (that is not visible to either j and k) is often less than $\mathcal{E}_a$. Hence, if e and f occurred at the same global time, the probability that the respective clocks differed by $\mathcal{E}$ depends upon the area of the shaded part (cf. Figure 8.1). In this case, e and f would still be replayed in arbitrary order. Only if the clocks fall in the non-shaded area then the replay will force an order between e and f. In other words, even if the system parameters fall in the infeasible region, it would be possible to use *RepCl* that provides a valid replay. It is just that it will not be able to reproduce all possible replays.

CHAPTER 9

CONCLUSION AND FUTURE WORK

In this paper, we focused on the problem of replay clocks in systems where clocks are synchronized to be within $\mathcal{E}$. The purpose of these clocks is to reproduce a distributed computation with all its certainties and uncertainties. By certainty, we mean that if event e must have happened before f then the replay must ensure that e is replayed before f. Specifically, this required that if e happened before f (as defined in [3]) or f occurred $\mathcal{E}_1 \approx \mathcal{E}$ time after e then e must occur before f. And, by uncertainty, we mean that if e and f could occur in any order then the replay should not force an order between them. Specifically, if $e||f$ (as defined in [3]) and e and f occurred within time $\mathcal{E}_2 \approx \mathcal{E}$ then the replay permits them to be replayed in any order. We presented *RepCl* to solve the replay problem with $\mathcal{E}_1 = \mathcal{E}+I$ and $\mathcal{E}_2 = \mathcal{E}-I$, where I is a parameter to *RepCl*. We analyzed *RepCl* for various system parameters (clock skew ($\mathcal{E}$), message rate (α), message delay (δ)). We find that for various system parameters, the size of *RepCl* and the overhead to create timestamps and/or compare them is small. For the purpose of replay, *RepCl* provides several advantages over existing approaches. For example, unlike logical clocks, they do not force certain unneeded event ordering. They have a significantly lower overhead compared to vector clocks. Also, they do not generate illegitimate replays that can occur with the user of vector clocks. The overhead of *RepCl*.j depends upon the number of processes that communicate with j (directly or indirectly) in $\mathcal{E}$ window. This is different from the case in vector clocks where the overhead is always proportional to the number of processes in the system.

With the design of the *RepCl*, we ensured that the clock size is not a leading factor in slowing down computation. The *RepCl* is a non-invasive method to ensure that causality is maintained in the presence of skewing clocks, and this is particularly useful in various applications. To facilitate the ease of development with this clock, we provide an API and a sample implementation of the API in NS-3, a widely used distributed network simulator. We illustrate with the help of NS-3, the various invariants our clock provides, such as the size scaling while varying various parameters. We also identify feasibility regions that would provide perfect replay through the clock. For systems out of

this feasible region, we additionally provide techniques to approximate replay that is acceptable.

We have utilized *RepCl* to enable users to visualize a distributed computation using our tool, *RepViz*. The goal of the visualization is to allow the user to identify an event f where a failure occurred. Then, they can use a replay of events just preceding f to determine whether the error would go away. If it does, it would imply that it is a synchronization error. Likewise, a user can replay some portion of the computation. Since the replay of events may occur in a different order, it will help identify potential synchronization errors.

RepCl is designed mainly for offline analysis, where the event data is stored during execution and analyzed at a later time. However, *RepCl* can be used for run-time monitoring/analysis as well if the data related to the timestamps is sent to a monitor, that monitor could analyze it for potential properties of interest if the analysis can be done *quickly*. However, a key challenge in this context will be whether the run-time monitors can keep up with the execution of the system. There are several future directions for *RepCl*. If the size of *RepCl* needed for perfect replay is too large, the user can reduce the size of *RepCl* by choosing a lower value of $\mathcal{E}$. In this case, the resulting replay will force some ordering between concurrent events. One of the future works is to identify the effect of reducing $\mathcal{E}$ in this manner.

Another potential future extension would be the ability to evaluate different veins of execution for different properties. Currently, it is difficult to compare and contrast different traces of execution for a specific invariant. With the help of *RepViz*, users can identify key characteristics on how data is moved between processes, and if there is an efficient way to coordinate movement. Another possible characteristic that could be measured is resource utilization and how it is affected by different veins of execution. Specifically, we aim to answer the question: Are there stark differences when an event is replayed first before another based on the amount of work needed to perform that event? This would give better methodologies to evaluate data movement operations.

Furthermore, we intend to expand this clock structure to beyond 64 processes. A potential avenue to do this is to implement it on a hierarchical structure. If a network is structured as a network of switches, with each switch connected to a cluster of nodes, we can implement a replay

clock for each level independently. All we would need is a mechanism to merge the clocks coming in from the cluster to the switch and have the switch relay clock information from its cluster to the other clustered nodes on the network.

BIBLIOGRAPHY

[1] S. Yingchareonthawornchai, D. N. Nguyen, S. S. Kulkarni, and M. Demirbas, "Analysis of bounds on hybrid vector clocks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 1947–1960, 2018.

[2] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan, "Observer effect and measurement bias in performance analysis," *Computer Science Technical Reports CU-CS-1042-08, University of Colorado, Boulder*, 2008.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[4] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*, pp. 17–32, Springer, 2014.

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[6] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proceedings of the 11th Australian Computer Science Conference (ACSC)* (K. Raymond, ed.), pp. 56–66, 1988.

[7] F. Mattern *et al.*, *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[8] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[9] H. Schildt, "C++ complete reference," 1998.

[10] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*, pp. 15–34, Springer, 2010.

[11] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *International Conference on Principles of Distributed Systems*, pp. 17–32, Springer, 2014.

[12] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," 1996.

[13] F. B. Schmuck, "The use of efficient broadcast protocols in asynchronous distributed systems," tech. rep., Cornell University, 1988.

[14] A. D. Kshemkalyani and M. Singhal, *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

[15] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Information Processing Letters*, vol. 43, no. 1, pp. 47–52, 1992.

[16] J. Fowler and W. Zwaenepoel, "Causal distributed breakpoints," in *Proceedings of the Tenth International Conference on Distributed Computer Systems*, 1990.

[17] M. Demirbas and S. Kulkarni, "Beyond truetime: Using augmentedtime for improving spanner," *Aug*, vol. 23, pp. 1–5, 2013.

[18] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, pp. 17–31, 1995.

[19] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3s: Debugging deployed distributed systems," in *NSDI*, 2008.

[20] W. De Pauw and S. Heisig, "Zinsight: A visual and analytic environment for exploring large event traces," in *Proceedings of the 5th international symposium on Software visualization*, pp. 143–152, 2010.

[21] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multithreaded software systems by using trace visualization," in *Proceedings of the 5th international symposium on Software visualization*, pp. 133–142, 2010.

[22] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[23] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the art of performance visualization.," *EuroVis (STARs)*, 2014.

[24] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing parallel execution traces using logical time," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2349–2358, 2014.

[25] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 357–368, 2015.

[26] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Communications of the ACM*, vol. 59, no. 8, pp. 32–37, 2016.