SGLC: A LOGICAL CLOCK USING SUCCINCT GRAPHS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SAIDU ALIYU ISA SOKOTO

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2022

Approval of the thesis:

SGLC: A LOGICAL CLOCK USING SUCCINCT GRAPHS

submitted by **SAIDU ALIYU ISA SOKOTO** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Ertan Onur
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Halit Oğuztüzün
Computer Engineering, METU _____

Prof. Dr. Ertan Onur
Computer Engineering, METU _____

Prof. Dr. İbrahim Körpeoğlu
Computer Engineering, Bilkent University _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Saidu Aliyu Isa Sokoto

Signature :

ABSTRACT

SGLC: A LOGICAL CLOCK USING SUCCINCT GRAPHS

Sokoto, Saidu Aliyu Isa
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Ertan Onur

July 2022, 56 pages

This thesis presents a new logical clock, SGLC, capable of capturing causality relationships in distributed systems. SGLC relies on a succinct graph representation codable and decodable in polynomial time while storing the graphs as integers. What makes SGLC feasible is that directed graphs can be used to implement logical clocks. Consequently, the main goal of introducing SGLC is to reduce the communication overhead of transporting causal history graphs by encapsulating the causality relationships of events as graphs that are decoded at the receiving process. We implemented the new protocol in an ad hoc computing framework and conducted an extensive benchmarking campaign comparing it with the vector clock, which is the most well-established type of logical clock. In addition, we evaluated other ways of further reducing the communication overhead and the overall storage complexity of the proposed clock. Finally, we also studied the application of SGLC in two distributed algorithms. Results obtained from experiments performed on the bare implementation of SGLC show that a reduction of up to 85% is attainable for a limit of 100 events and 32 processes in terms of overall bits exchanged compared to the vector clock. Applying further optimizations to SGLC can result in a further reduction of 63% for the same number of events.

Keywords: Logical Clocks, Linear Clocks, Succinct Representation, Succinct Graphs

ÖZ

SGLC: ÖZ ÇİZGE GÖSTERİMLERİ İLE BİR MANTIKSAL SAAT

Sokoto, Saidu Aliyu Isa
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ertan Onur

Temmuz 2022 , 56 sayfa

Bu tez, dağıtık sistemlerde nedensellik ilişkilerini yakalayabilen yeni bir mantıksal saat olan SGLC'yi sunmaktadır. SGLC, grafları tamsayı olarak depolarken polinom zamanda kodlanabilen ve çözülebilen özlü bir grafik gösterimine dayanır. SGLC'yi uygulanabilir kılan şey, yönlendirilmiş grafların mantıksal saatleri uygulamak için kullanılabilmesidir. Sonuç olarak, SGLC'yi tanıtmanın temel amacı, olayların nedensellik ilişkilerini alıcı süreçte kodu çözülen grafikler olarak kapsülleyerek nedensel geçmiş grafiklerinin taşınmasının iletişim yükünü azaltmaktır. Yeni protokolü bir ad hoc hesaplama çerçevesinde uyguladık ve en köklü mantıksal saat türü olan vektör saati ile karşılaştıran kapsamlı bir kıyaslama kampanyası yürüttük. Buna ek olarak, önerilen saatin iletişim yükünü ve genel depolama karmaşıklığını daha da azaltmanın diğer yollarını değerlendirdik. Son olarak, SGLC'nin iki dağıtık algoritmada uygulanmasını da inceledik. SGLC'nin çıplak uygulaması üzerinde yapılan deneylerden elde edilen sonuçlar, vektör saatine kıyasla değiş tokuş edilen toplam bitler açısından 100 olay ve 32 işlem sınırı için %85'e varan bir azalma elde edilebileceğini göstermektedir. SGLC'ye daha fazla optimizasyon uygulamak, aynı sayıda olay için %63'lük bir azalma ile sonuçlanabilir.

Anahtar Kelimeler: Lamport saati, mantıksal saatler, öz çizge gösterimi

To my parents

ACKNOWLEDGMENTS

I want to express my deep appreciation to my thesis supervisor, Prof. Dr. Ertan Onur for introducing me to distributed systems and suggesting this research topic to me. His academic and psychological support made my time at METU memorable.

Finally, this thesis wouldn't have been possible without the sponsorship of YTB to whom I am grateful.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ABBREVIATIONS xvi

CHAPTERS

1 INTRODUCTION 1

1.1 Causality Tracking 1

1.2 Contributions and Novelties 2

1.3 The Outline of the Thesis 3

2 BACKGROUND AND RELATED WORK 5

2.1 Events in Distributed Systems 5

2.1.1 Distributed Execution and Partial Order 6

2.1.2 Assigning Dates/Time to Events 7

2.2 Related Work 9

2.3 Scalar/Linear Clock 12

2.4 Vector Clock 14
2.4.1 Properties of Vector Clocks 16
2.5 Graph Representations 17
3 SGLC: A SUCCINCT GRAPH LOGICAL CLOCK 27
3.1 SGLC Algorithm 27
3.2 An Example 28
4 SIMULATION ENVIRONMENT AND EVALUATION OF SGLC 31
4.1 Simulation Environment 31
4.2 System Model 33
4.3 Evaluation of SGLC 34
4.4 Scalability Issues With SGLC 38
4.4.1 Incremental Piggybacking 39
4.4.2 Relevant Events 41
4.4.3 Resetting SGLC 42
5 CASE STUDIES 45
5.1 Predecessor Determination 45
5.2 Causal Order Broadcast 46
6 CONCLUSIONS AND FUTURE WORK 49
REFERENCES 51

LIST OF TABLES

TABLES

Table 2.1 Decoding time for graphs based on number of edges. 25

Table 4.1 Complexity of some operations performed on vector clock and SGLC. 39

Table 4.2 Result of applying incremental piggybacking to SGLC. 41

LIST OF FIGURES

FIGURES

Figure 2.1 An example space/time diagram for a distributed execution. . . . 8

Figure 2.2 The positioning of logical clocks in distributed systems. 14

Figure 2.3 Behavior of the Cost Function J. 22

Figure 2.4 Behavior of the Cost Function J. 22

Figure 2.5 Graph decoding time based on the number of edges. 24

Figure 3.1 An example interaction between events. 30

Figure 4.1 Aggregate message sizes of vector clock and SGLC for 4 processes. 35

Figure 4.2 Aggregate message sizes of vector clock and SGLC for 8 processes. 36

Figure 4.3 Aggregate message sizes of vector clock and SGLC for 16 processes. 36

Figure 4.4 Aggregate message sizes of vector clock and SGLC for various 32 processes. 37

Figure 4.5 Aggregate message sizes of vector clock and SGLC for various number of processes and events. 37

Figure 4.6 Cross-section view of aggregate message sizes of vector clock and SGLC for various processes. 38

Figure 4.7 An example interaction between events. 40

Figure 4.8 Application of incremental piggybacking. 40

LIST OF ABBREVIATIONS

3D	3 Dimensional
SGLC	Succinct Graph Logical Clock

CHAPTER 1

INTRODUCTION

In this chapter, we briefly introduce causality tracking and how logical clocks are used for implementing it. In addition, we also give the application domain of logical clocks, the aim of the thesis, and what we hope to overcome by introducing a new type of logical clock.

1.1 Causality Tracking

The ability to capture causality plays an important role in distributed computing applications. It can be used to determine consistent recovery points in distributed databases, prevent deadlocks and false termination. It also plays a critical role in distributed debugging, detection of race conditions, and other synchronization errors[1, 2]. Furthermore, it can be used in ensuring maximum parallelism as pointed out by Raynal in [3].

However, in comparison to their sequential counterparts, distributed applications are more difficult to design. This is mainly due to additional problems such as parallelism, non-determinism, absence of a precise global time, delays, and so on [1]. All of which one way or the other can be attributed to the lack of a global clock. This is especially the case with modern distributed systems, which have a higher need for keeping track of causal precedence due to replicas, unreliable hardware, and the use of proprietary hardware [4]. As a result of this, and in order to keep track of the ordering of events and their causal relationships in distributed systems, logical clocks are used. They do not require any form of physical clocks, more so their synchronization. The first solution in this regard is the Lamport clock [5], which, despite its

efficient implementation, cannot indicate causality or concurrency. This deficiency in Lamport clocks gave room for the introduction of vector clocks, which can keep track of the order of events. However, the storage complexity of vector clocks and the need to send a vector of size proportional to the number of processes make their use unattractive. This is the case, especially in large distributed systems with many nodes/processes.

Therefore, the main aim of this thesis is to adopt a succinct representation of graphs to keep track of causal relationships and consequently reduce the overall bandwidth requirements on channels. This representation is codable and decodable in polynomial time and precedence determination is a constant time operation since partial order is transitively closed and an edge exists between any ordered events [6].

Furthermore, we explore additional techniques to further reduce the bit complexity of the graphs by only sending a subset of the whole graph (incremental piggybacking [7]). We also propose other ways of taming the local storage of the graphs by keeping track of only relevant events [8, 9, 10] and briefly touch on the possibility of resetting the SGLC as inspired by [11, 12].

SGLC will not be useful without possible use cases, consequently, we make a case for the use of SGLC for causal order broadcasting and immediate predecessor determination, which are both used for building more meaningful applications.

1.2 Contributions and Novelties

Our contributions are as follows:

- SGLC: A logical clock using succinct graphs
- Evaluation and validation of SGLC
- Optimization of SGLC
- Case studies/use cases of SGLC

1.3 The Outline of the Thesis

In addition to the introduction, this thesis consists of 5 additional chapters organized as follows; in Chapter 2, we give some formal foundation underlying logical clocks and a brief overview of some logical clocks already present in the literature. In Chapter 3, we present our proposed succinct graph logical clock (SGLC). In Chapter 4, we give an overview of the simulation environment used for experimenting and evaluating the performance of SGLC along with methodological issues, research design, and a description of procedures followed in order to empirically compare SGLC with the most well-established clock (the vector clock). In addition, we explore some techniques for further optimizing SGLC to reduce both its storage and bandwidth requirement. In Chapter 5, we give an overview of possible use cases before finally concluding and discussing avenues for further research in Chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, a brief introduction to events, partially ordered sets (POSETs), and other relevant information necessary to understand the rest of this thesis is presented. Furthermore, we elaborate on two types of clocks, linear and vector clocks. We also make a case for how graphs are related to causality tracking through POSETs and then go on to give an overview of the succinct graph representation we make use of in the rest of the thesis.

2.1 Events in Distributed Systems

When dealing with distributed systems, the concept of an event plays an important role in understanding the system's inner workings. However, what is an event? An event is simply the execution of a statement [13]. Consequently, since a process is sequential, it consists of a sequence of events, which can be categorized into three in a distributed system:

- Send events
- Receive events
- Internal events

A send event takes place when a process sends a message to another process while a receive event occurs when a process receives a message from another process. Both are considered to be communication events, which take place between processes that are connected through a communication channel (medium). Internal events on the

other hand take place within the same process and do not involve any form of external communication. They range from the execution of machine instructions to the execution of subprograms and so on as further elaborated in [13].

Since each process produces events sequentially, then events occurring at a process say p_i are said to be totally ordered. This sequence can also be called the history of p_i, which can be denoted as h_i. Therefore, if e_{ix} is the x^{th} event produced by process p_i then the event history can be denoted as $h_i = \{e_{ix}\}$ where x denotes the number of events that have taken place at process p_i.

2.1.1 Distributed Execution and Partial Order

A distributed execution can be viewed as a partial order of local events, which can be extended to include all the events in a distributed computation as defined in definition 2.1.1. This is synonymous with the union of the event histories of all processes.

Definition 2.1.1. *Given that h_i is the sequence of events that have taken place at process p_i, the set of all events produced by a distributed computation (execution) will be a union of the events of all processes. Denoting this set as H, it is equivalent to $h_1 \cup h_2 \cup h_3 \ldots \cup h_n$.*

Furthermore, a relationship exists between communication events known as message relation [13], which is defined in definition 2.1.2.

Definition 2.1.2. *[13] Given that M is the set of all messages exchanged in a distributed computation, a message order relation between send and receive events can be defined as follows: given any message $m \in M$, with $s(m)$ denoting the sending of that message and $r(m)$ its reception.*

In other words, definition 2.1.2 is equivalent to saying that message m has to be sent before it is received ($s(m) \longrightarrow r(m)$). This can be generalized as done in definition 2.1.3 to include other causal relationships as first envisioned by Lamport [5] and not just messages.

Definition 2.1.3. *[5, 13, 14] The smallest partial order relation denoted $\rightarrow$, is capable of capturing the flow of a distributed computation if one of the following properties*

is met:

1. *Process order: events e_x and e_y belong to the same process and $x > y$ then $e_x \rightarrow e_y$.*
2. *Message order: $\exists m \in M : e_x = s(m)$ and $e_y = r(m)$, if e_x is the sending of message m by p_i and e_y is the reception of the same message by p_j then $e_x \rightarrow e_y$.*
3. *Transitive closure: if $e_x \rightarrow e_y \wedge e_y \rightarrow e_z$ then $e_x \rightarrow e_z$ (results from combining the above two properties).*

Relying on the definition of a causal relation, we can model a distributed computation as a partial order set (POSET) on a set of events that is independent of physical time and enables the capturing of the core of asynchronous distributed computation. This gives a framework for the explanation, analysis, and simplified reasoning of distributed event-driven computations [13]. Basically, the POSET $\hat{H} = (h_i, \rightarrow)$ will come to represent the main model for representing causality we adopt in this thesis, such that the relation "$\rightarrow$" known also as the "happened before" or causal precedence relation will serve as a building block for explaining and building tracking mechanisms/schemes. These mechanisms can be viewed as operations, which make it possible to extract relevant information from events in order to build more useful applications and algorithms.

2.1.2 Assigning Dates/Time to Events

The aim at hand is to associate with each event that is part of a distributed execution a logical time independent of physical time. This is usually achieved by creating a mapping between causality-related events and the set of non-negative integers (this can also be achieved using functions as done in [15]). This mapping in essence respects the causal precedence relation $\rightarrow$ between events, which is the building block of a causal path defined below.

Definition 2.1.4. *A sequence of events $e_1, e_2, e_3, \ldots, e_z$ form a **causal path** if there is*

a unique and (usually) increasing event identifier such that

$$\forall x : 1 \leq x < z : e_x \rightarrow e_{x+1}.$$

Since a causal path is a sequence of equally increasing events connected by $\rightarrow$, it can be observed that the process history earlier introduced is equivalent to a causal path. This can be more easily observed and depicted using a graphical representation capable of capturing a distributed computation such as a space/time diagram. In such a representation, individual processes are represented by an arrow from left to right representing the direction of the flow of events (time in other words) while a message is represented by an arrow from the sending process to the destination process.

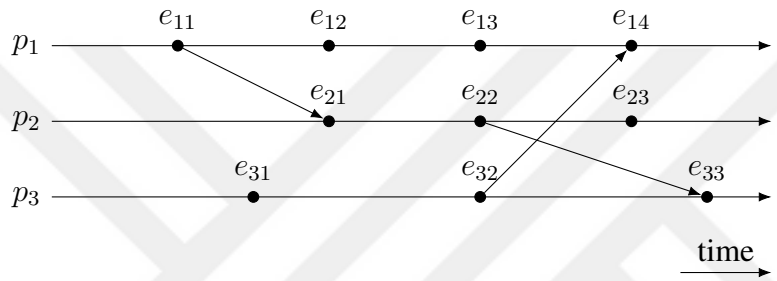


Figure 2.1: An example space/time diagram for a distributed execution.

An example of such space/time diagram is shown in Fig 2.1. The sequence of events $e_{11}, e_{21}, e_{22}, e_{33}$ form a causal path between event e_{11} and event e_{33}. It is also important to observe that the causal relationship relates processes that might not otherwise directly communicate with each other. For instance, p_1 never sent a message directly to p_3, but a causal path exists between the two as earlier observed. The absence of a causal path between events signifies that the two events are concurrent. For example, event e_{11} and e_{31} can be said to be concurrent, which is usually denoted as $e_{11}||e_{31}$ resulting simply because there are no causal paths in either direction. More formally, this can be denoted as:

$$e_{11}||e_{31} = \neg(e_{11} \rightarrow e_{31}) \wedge \neg(e_{31} \rightarrow e_{11}).$$

In summary, there are three possible relationships between any two events, say e and e':

1. e could have preceded e', if $e \rightarrow e'$.

2. e' could have preceded e, if $e' \rightarrow e$.

3. Neither e nor e' precede one another.

With these three relationships, it becomes possible to categorize the set of possible events into three based on their logical time of occurrence as done in [13],

1. past(e') $= \{f \mid e \rightarrow e'\}$,

2. future(e) $= \{f \mid e \rightarrow e'\}$,

3. concurrent(e) $= \{e' \mid e' \notin \text{past}(e) \cup \text{future}(e)\}$.

2.2 Related Work

The framework described above makes it possible to not only explain, but also build mechanisms known as logical clocks that are capable of keeping track of causal relationships in a distributed computation. This is achieved by maintaining sets of events at processes and piggybacking event histories on outgoing messages (send events). In other words, it is possible to make a connection between the aforementioned notions of consistent dates with events in a distributed computation. Consistent dates require that dates generated by a clock mechanism must conform to the "causality" generated by the distributed execution being considered. Based on the view of a distributed execution we are interested in, this causality is the causal precedence order on events (relation $\rightarrow$), and every "cause" has to precede an "effect". Since we consider distributed systems that are asynchronous and have no notion of time, the dates do not correlate with physical time.

The first of such solutions was proposed by Lamport in his seminal paper [5] where he used the term "happened before". This paved the way for many schemes geared toward capturing causal relationships. Each scheme has its trade-offs, which can be evaluated based on time, space, or bit complexity. Generally, they all aim at determining whether an event preceded, succeeded, or was concurrent with another event.

For the sake of this thesis, we will focus on algorithms that require attaching a data structure with messages exchanged between processes. By far the most well-known scheme in this regard is the vector clock, which was independently introduced in [16, 17, 18]. Multiple attempts have been made to make the vector clock more efficient. Meldal, Sankar, and Vera [19] achieved this by assuming that communication paths are known beforehand and only focusing on messages received by the same destination. Singhal and Kshemkalyani [20] came up with the idea of sending only entries that have been updated since the most recent message. This with some trade-offs, such as making use of an additional vector at each process results in a reduction in the size of messages. Further improvements were made to that scheme by Hélary et al. [21] who introduced multiple algorithms that provide varying trade-offs between space and communication overhead but eventually make use of a vector of size less than the number of processes. Similarly, there is the work of Agarwal and Garg [22] who introduced variants of the vector clock called chain clocks. Their algorithms mainly focus on keeping track of relevant events and using those events to set an upper bound on the number of components required to capture causality. Nonetheless, a shared data structure is necessary, making it more practical for shared memory systems. Another similar attempt at reducing the size of vector clocks is the work of Garg, Skawratananond, and Mittal [23]. They reduce the number of components required by grouping the distributed system into stars or triangles. Each component then serves a particular group avoiding the need to have a component for each process. Similarly, Ward and Taylor [24] proposed a solution for processes organized into hierarchical clusters. A dynamic and self-organizing version of this is also available [25]. More recently, we have seen the introduction of the mix-vector clock in [26]. This clock is meant for shared memory systems and as a result, makes use of a combination of thread and object components represented as a bipartite graph. Finding a vertex cover for this graph then gives the minimum number of components necessary in the vector clock. Lastly, the use of an encoded vector clock can also be found [27].

A generalization of vector clocks in the form of matrix clocks can also be found in the literature. For instance, in [28, 29] messages are accompanied by a matrix, which conveys the causal relationships between events. Similar to vector clocks, the memory

and communication cost of the matrix makes their usage impractical. Nonetheless, Drummond and Barbosa have made improvements to this type of clock in [30].

Apart from vectors, other data structures have also been used to capture causality. There is the plausible clock [31], which is a probabilistic solution that makes use of a fixed number of components (tuples) to keep track of causality. For greater accuracy, multiple plausible clocks can be combined. Nonetheless, the results presented show that it is not 100% accurate. Recently the Bloom clock has been proposed by Rambaja [32]. It makes use of a probabilistic and space-efficient data structure known as a bloom filter. The complexity of the algorithm is independent of the number of processes and relies on a set of chosen parameters that determine its confidence intervals. However, with no quantitative analysis, it will be hard to conclude how well it performs.

Despite the abundance of techniques in the literature on tracking causality, we have only come across a handful that treads the path we have chosen i.e. the use of graphs to keep track of causal relationships. The most evident is the antecedence graph used in the Manetho system [7], which is used to propagate causal histories in the form of graphs. Although no attempt was made to reduce the size of the antecedence graph, other optimization techniques were put forth to reduce the overall communication overhead. Similarly, Psynch [33] and Transis [34] also keep track of causality by maintaining graphs. However, Psych, as noted in [7] is more or less a tool for imposing order on message delivery and requires that each process or a subset of processes (multicast) receive every message. Similarly, the Lansis protocol in Transis uses directed acyclic graphs to keep track of all the messages in the system and maintain order on message delivery.

Unfortunately, most of these solutions do not give a general-purpose solution for analyzing causal relationships. Many are isomorphic variants of the vector clock with specific systems in mind. Other proposals, despite being clever, lack the intuition that should ease the understanding of causal precedence and do not even have any analysis to back them up. On the contrary, the flexibility and generalizability provided by graphs make them an attractive solution as we shall show.

Below we give further details on two types of logical clocks that play an important

role in understanding logical clocks, namely: scalar (linear) clocks and vector clocks.

2.3 Scalar/Linear Clock

As mentioned earlier, the aim is to bind each event with a unique logical timestamp. Akin to the notation in [31] we let $\tau(e)$ be the logical time associated with event e. Linear clocks capture the order between causally related events by creating a correspondence τ between events and the set of non-negative integers that respects the causal precedence relation between the events. Linear clocks exhibit the weak clock condition such that

$$\forall e, e' : (e \rightarrow e') \implies \tau(e) < \tau(e').$$

Because a linear clock relies on a sequence of increasing integers, it is the simplest time domain capable of respecting causality. A process p_i achieves this by increasing its local clock c_i before producing an internal event, sending a message, or after receiving a message thereby making this new clock value the time of the corresponding receive, send, or internal event. Additionally, every message contains its sending time and upon receiving a message, a process updates its local clock. As a result, the receiving process has a time larger than both the time of the corresponding send event and the time of its last internal event. Algorithm 1 captures these operations. Each process p_i maintains a local counter C_i, which is updated accordingly based on the type of event taking place. Messages from the lower level layers serve as the receipt of a message, a message from the upper layers triggers the send message while internal events occur on the same layer without affecting another process. Further details about this architecture can be found in Chapter 4 where we introduce the simulation environment, but the corresponding structural view can be seen in Fig 2.2. We use the middleware as a controller to observe the execution, which does not in any way affect the execution being observed [35, 36].

From the algorithm, it follows that the linear clock is consistent with the causal relation since it increases along all causal paths [13]. This is the case because an incremental value of 1 is used. Regardless, any value ≥ 1 should suffice. Hence, each

Algorithm 1 Linear Clock algorithm

Implements: Linear Clock **Instance**: LC

Uses: Logicalclock **Instance**: lc

Events: Init, Internal, MessageFromBottom, MessageFromTop

OnInit: () **do**

1: $C_i = 0$

OnInternal: () **do**

2: $C_i = C_i + 1$

OnMessageFromBottom: (m, c) **do**

3: $C_i = max\{C_i, c\}$

4: $C_i = C_i + 1$

5: **Trigger** lc.Sendup (m)

OnMessageFromTop: $(m\ )$ **do**

6: $C_i = C_i + 1$

7: **Trigger** lc.SendDown (m, C)

process p_i manages a local integer variable C_i (initialized to 0) that increases in respect to the relation.

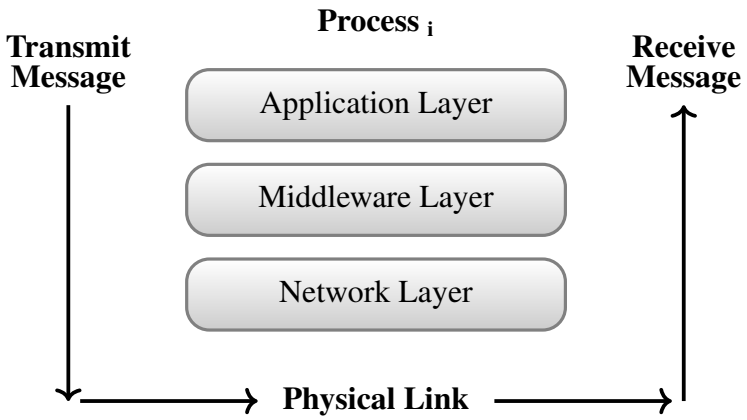


Figure 2.2: The positioning of logical clocks in distributed systems.

Property 2.3.1. *[13] Since logical clocks are consistent with the "happened before" relation, the following two properties follow accordingly, assuming e and e' are two events.*

1. $\tau(e) \leq \tau(e') \implies \neg(e' \rightarrow e)$
2. $\tau(e) = \tau(e') \implies (e \mid\mid e')$.

Although linear clocks capture the order between casually related events, they are unable to keep track of both causality and concurrency. Therefore, with linear clocks it is possible to have $\tau(e) < \tau(e') \wedge \neq (e \rightarrow e')$. In other words, simply because event e has a smaller time than event e' it does not mean that e precedes e'. A stronger scheme is needed to detect this, leading us to the vector clock.

2.4 Vector Clock

Vector clocks make it possible to determine the causal relationship between events by providing a time domain that enables the accurate comparison of two events. This is possible because each event is uniquely associated with a specific process/node and therefore uniquely identifiable. More precisely, let $\tau(e)$ be the time associated with an

event e, vector clocks provide a time domain in which any two events are comparable in such a way that the following properties are satisfied:

1. $\forall e, e' : (e \rightarrow e') \leftrightarrow \tau(e) < \tau(e')$,
2. $\forall e, e' : (e||e') \leftrightarrow \tau(e)||\tau(e')$.

Algorithm 2 Vector Clock algorithm

Implements: Vector Clock **Instance**: vc

Events: MessageFromTop, MessageFromBottom, Internal

OnInit: () **do**

1: $V_i[j] = 0$ for all $j = 1, \ldots, n$.

OnInternal: () **do**

2: $V_i[j] := V_i[j] + 1$

OnMessageFromTop: (m) **do**

3: $V_i[j] := V_i[j] + 1$

4: **Trigger** vc.SendDown (m, V_i)

OnMessageFromBottom: (m, V) **do**

5: $V_i[j] := V_i[j] + 1$

6: $V_i[j] := max\{V_i[j], V)\}$ for all j$= 1, \ldots, n$.

7: **Trigger** vc.SendUp (m)

In order to achieve this, each process p is assigned a unique identifier p_i such that $0 < p_i \leq N$ where N is the number of processes in the computation. Furthermore, each individual process p_i maintains a vector of positive integers $V_i[1 \ldots N]$ with each element initially set to zero. This vector enables the tracking of the following:

1. The number of events produced at p_i through $V_i[i]$
2. The number of event produced at p_j through p_i's perspective using $V_i[j]$, with $j \neq i$.

Therefore, a vector clock keeps track of events that happen both locally and at other processes thereby guaranteeing that the scheme is in tandem with the $\rightarrow$ relation. Algorithm 2 gives the implementation of the vector clock in an event-driven simulator (see Chapter 4). It can be observed that by incrementing a local counter and keeping track of events that occur in another process through an n-dimensional vector every process can keep track of causal relationships. The vector helps to keep track of global time and is updated based on messages received from other processes. In other words by knowing $V_i[j] = k$, process p_i knows that k events occurred at process p_j. Thus, p_i has knowledge of the time at p_j.

More formally, if e' is an event produced by a process p_i before producing e' we have the following as reproduced from [13]:

$$V_i[k] = e' \mid (e' \text{ produced by } p_i) \wedge (e \rightarrow e') \mid +1(k, i)$$

where the value of $V_i[k]$ corresponds to the number of events produced at process p_i that are in the causal past of event e', $1(k, i)$ is the clock value producing event e' at process p_i, and $V_i[1..N]$ is the vector date of event e' [13]. Therefore, the relationship between two vectors, say V_x and V_y can be deduced as follows:

1. $V_x \leq V_y$ **iff** $\forall k \in [1, \ldots, n] \mid V_x[k] \leq V_y[k]$,
2. $V_x < V_y$ **iff** $(V_x < V_y) \wedge (V_x \neq V_y)$,
3. $V_x \mid\mid V_y$ **iff** $\neg(V_x \leq V_y) \wedge \neg(V_y \leq V_x)$.

2.4.1 Properties of Vector Clocks

The above three comparisons serve as the basis for an important property of vector clocks stated in theorem 1 below.

Theorem 1. *Let $e.V$ be the vector clock associated with event e, by Algorithm 2 the meaning of time is such that for any two distinguishable events e and e' either $e \rightarrow e'$ meaning $e.V < e'.V$, or $e.V \mid\mid e'.V$.*

Relying on this theorem, the proof of which can be found in [1], it is possible to compare events based on their vector clocks. However, this will require $O(n)$ comparisons. With additional information about the process that produces an event, this can be reduced as shown in lemma 1. Consequently, checking whether or not two events are causally related becomes a comparison of two integers.

Lemma 1. *Given two events e and e' such that $e \neq e'$, then*

- $e < e' \quad \textbf{\textit{iff}} \quad e.V[i] \leq e'.V[i]$,
- $e \mid\mid e' \quad \textbf{\textit{iff}} \quad e.V[i] > e'.V[j] \wedge e.V[j] > e_1.V[i]$.

The proof of this lemma can be found in [37, 35]

Despite the strength of vector clocks and the insight they give into causal relationships, the need to maintain vectors of size proportional to the number of processes at each process is a major drawback. This is further exacerbated by the need to exchange such a vector during communication events.

2.5 Graph Representations

Relying on a succinct graph representation [38] this communication cost can be reduced. The encoding scheme proposed therein provides a solution that is both one to one and onto between integer numbers and labeled directed graphs enabling a succinct representation of labeled directed graphs that can be stored close to the information-theoretical lower bound. Furthermore, the speed of the decoding and encoding algorithm they propose is independent of the number of nodes.

It is possible to track causality through causal histories. The requirement is that at the occurrence of a new event a process assigns a unique name to it. For instance, using a combination of the process id and its local increasing counter. The causal history is then the union of the new event and the causal history of previous events. A process can then send this causal history to other processes in order to capture the known past. Using Fig 2.1 as an example, the third event at process p_1 is assigned the name e_{13} and the causal history at that point will be $H = \{e_{11}, e_{12}, e_{13}\}$. Afterward,

a process sending a message also sends its causal history (or the causal history of the event that triggers the send event). In the same vein, when a process receives a message, it merges its local causal history with the received causal history. As a result of this, checking whether an event e causally precedes an event e' can be done by checking whether $H(e) \subset H(e')$ or even simply whether $e \in H(e')$ [35] provided some assumption are satisfied. This is the same as the message relation defined earlier and a directed acyclic graph is capable of capturing this.

Earlier, we mentioned that space-time diagrams can be viewed as directed graphs with each vertex representing an event and the edges the relationship between the events. To make this statement more concrete, it should be noted that a POSET $P = (X, \leq)$ can be represented using a directed graph with X being the vertex set and $\leq$ being the set containing the edges [39]. But for this to be true it has to be a strict or strong [40] partial order. Meaning it has to be irreflexive, asymmetric, and transitive as elaborated in definition 2.5.1.

Definition 2.5.1. *A strict partial order set is a pair of sets X and $<$, where X is a set and the relation $<$ is irreflexive, asymmetric, and transitive.*

1. *For all $x \in X$, $\neg(x < x)$ (irreflexive)*
2. *if $x < y$, then $\neg(y < x)$ (asymmetric)*
3. *if $x \leq y$ and $y \leq z$, then $x \leq z$ (transitivite).*

It should also be noted that "$\rightarrow$", "happened before", and message relations are synonymous and equivalent to the strict partial order which in turn can be represented using a directed acyclic graph (DAG). For completeness, we give the definition of a directed graph below followed by a theorem characterizing DAGs.

Definition 2.5.2. *A directed graph $G = (V, E)$ consists of a set of vertices V and a set of directed edges E between the vertices.*

Theorem 2. *A direct acyclic graph (DAG) G has at least one vertex with an in-degree of zero and at least one vertex with an out-degree of zero.*

In simple terms, theorem 2, the proof of which can be found in [41] simply states that DAGs are devoid of loops.

There are numerous ways of representing directed graphs including but not limited to adjacency matrices, adjacency lists, incidence matrices, and succinct representations. In this thesis, we will be relying on the succinct representation, specifically, the one from [38]. With this representation it is possible to represent a directed graph G as a tuple consisting of $(X, CODE, DECODE)$, where X is an integer between 0 and $C(n(n-1),\ m)$ that can be encoded and decoded in polynomial time and C here stands for combinations, $n = |V|$, and $m = |E|$. By adopting this succinct representation from [38] we have the encoding and decoding Algorithms 3 and 4 respectively.

The crux of the decoding function is the minimization of the cost function,

$$J(x, c, g) = \sum_{p=1}^{c} \left[\log_b \left(\frac{x-c}{p} + 1 \right) \right] - \log_b(g). \tag{2.1}$$

This is posed as a constrained optimization problem in [38]. However, as we will show shortly, it is better approached as a problem of determining the x-intercept occurs subject to $x \geq c$. Here $x \in \mathbb{R}$, $c \in \{1, 2, \ldots, m\}$, representing the current edge label being determined, g is an integer representing the graph and the binomial coefficient $\binom{x}{c} \leq g$. Additionally, the base of the $\log$ can be arbitrarily chosen depending on the expected size of the integer representation (preferably 10).

We observed that the interpretation of the absolute value with regards to the cost function $|J|$ in [38, 42] can be a bit ambiguous. We therefore summarize all the possibilities in Fig. 2.3 and 2.4. From Fig. 2.3, we see that J (No absolute value) is defined for only values ranging from $[c, +\infty)$. On the other hand, $|J|$ results in a convex function with one global minimum. Fig 2.4, shows the behavior of the function on applying the absolute value on only $\log_b(\frac{x-c}{p})$ and to both J and $\log_b(\frac{x-c}{p})$. Using $|\log_b(\frac{x-c}{p})|$ results in a convex function with one global minimum between $x \in [c, +\infty)$, although not a root. On the other hand, applying the absolute value to both $|J|$ and $\log_b(\frac{x-c}{p})$ results in multiple minima (including for values of $x < c$) and a global maximum. In summary, there are four possibilities:

Algorithm 3 Succinct Graph Encoding (adopted from [38])

Input: G : a graph

Output: g: integer representing G, m: number of edges in G

function ENCODE(G)
    $g \leftarrow 0$
    $j \leftarrow 0$
    $links \leftarrow []$
    $m \leftarrow$ number of edges in G
    for each vertex $u \; in \; G$ **do**
        for each edge v of u **do**
            $u \leftarrow u$
            $v \leftarrow v$
            if $u < v$ **then**
                $i = (2u) + (v-1)(v-2)$
            else
                $i = (2v) + u(u-3) + 1$
            end if
            links.append(i)
        end for
    end for
    links.sortdescending()
    for n in $links$ **do**
        $g \leftarrow g + (-1)^{(m-j)} * (C(n,\, j) - 1)$
        $j \leftarrow j + 1$
    end for
return $g,\, m$
end function

Algorithm 4 Succinct Graph Decoding (adopted from [38])

Input: g: Integer representing graph G

Input: m: Number of edges in G

Output: graph: decoded graph

function DECODE(g, m)
    graph $\leftarrow \{\}$
    for c $\leftarrow m$ to 1 **do**
        $x = \lfloor \underset{x}{\text{minimize}}\ J(x, c, g)$ subject to $x \geq c \rfloor$
        $i = x + 1$
        if i odd **then**
            $u = 1 + \left\lceil (-1/2) + \sqrt{1/4 + i} \right\rceil$
            $v = ((i + 1)/2) - C(u - 1,\ 2)$
            if u not a vertex of graph **then**
                graph[u] = []
                graph[u].append(v)
            else
                graph[u].append(v)
            end if
        end if
        if i is even **then**
            $v = 1 + \left\lceil (-1/2) + \sqrt{1/4 + i} \right\rceil$
            $u = (i/2) - C(v - 1,\ 2)$
            if u not a vertex of graph **then**
                graph[u] = []
                graph[u].append(v)
            else
                graph[u].append(v)
            end if
        end if
        $g = C(i,\ c) - g - 1$
    end for
return $graph$
end function

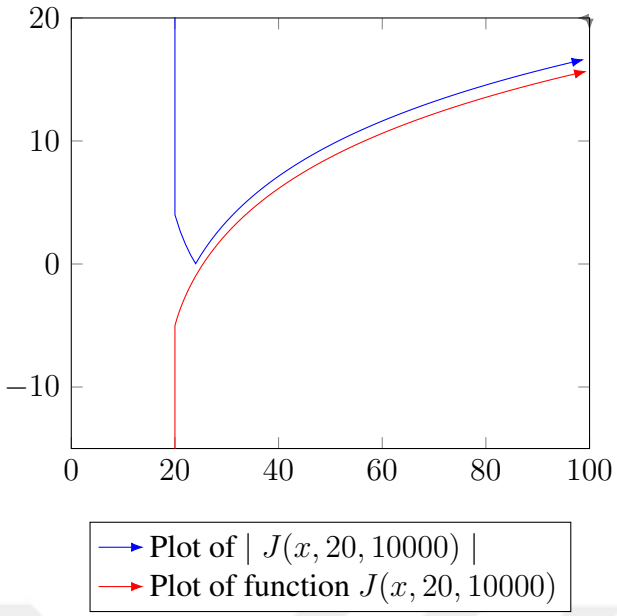


Figure 2.3: Behavior of the Cost Function J.

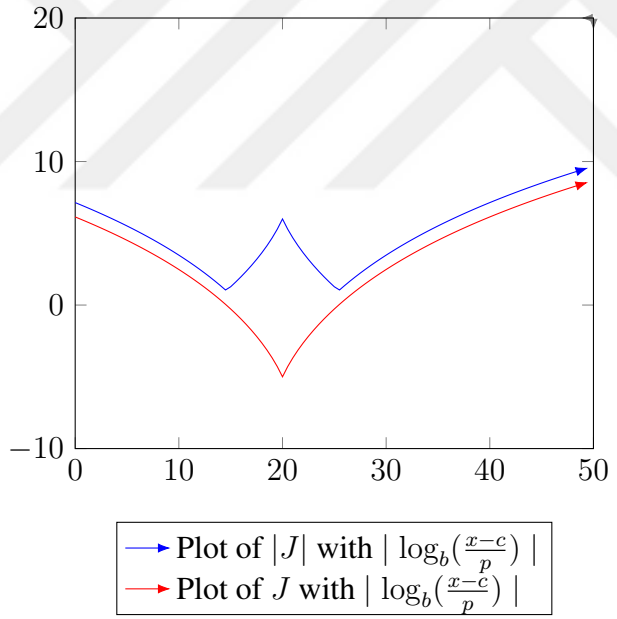


Figure 2.4: Behavior of the Cost Function J.

1. J defined from $[c, +\infty)$. No minima, see Fig. 2.3,

2. $|J|$ defined from $[c, +\infty)$. One root, see Fig. 2.3,

3. J and $|\log_b(\frac{x-c}{p})|$ defined from $(-\infty, +\infty)$. Global minimum with no root, see

Fig. 2.4,

4. $|J|$ and $|\log_b(\frac{x-c}{p})|$ defined from $(-\infty, +\infty)$. Multiple minima, see Fig. 2.4.

Evidently, with the constraint that $x \geq c$ in place, we are left with only the first two options, and Fig. 2.4 becomes exactly Fig. 2.3 and only the second options become a viable solution. Nevertheless, as we will see shortly it doesn't work.

Following the authors in [38], we use the Newton-Method as shown in (2.2) as our optimization technique mainly due to its fast convergence rate. For the initial guess of x_i we set it to the number of edges, m. Subsequently, we set x_i to the previous solution, thus ensuring fast convergence owing to the fact that $x_1 < x_2 < \ldots < x_m$ as shown in (2.4).

$$x_{i+1} = x_i + \frac{J}{J'} \tag{2.2}$$

$$J'(x, c) = \frac{1}{\ln(b)} \sum_{p=0}^{c-1} \left(\frac{1}{x - p} \right) \tag{2.3}$$

$$x_i^0 = \begin{cases} m & \text{if i = m} \\ x_{i-1} & \text{otherwise} \end{cases} \tag{2.4}$$

Throughout our experiments, the maximum number of iterations is set to 12. Additionally, we make use of three other stopping criteria, if $|\ f(x_n)\ | < \epsilon$, if $(|\ f'(x_n)\ |) < \epsilon)$ and if $(relativeerror < \epsilon)$, where $\epsilon = 10^{-11}$. The tolerance is set very small in order to accurately decode large graphs, otherwise, a higher tolerance should be sufficient. In certain situations, during the simulation, we do use a higher tolerance when two acceptable roots are very close to each other. As an example of this, suppose both 4.99999 and 5.00000 are roots for different integer representations of a graph. Since we use the floor function in determining the edge label, both 4 and 5 should be genuine labels. However, using a small tolerance would usually skip 4.99999 and only return 5.00000. To circumvent this, we increase the tolerance when such a number is witnessed and reevaluate the root with $\epsilon = 10^{-6}$. By doing this we ensure that 4 is not skipped and 5 is determined subsequently.

Furthermore, each iteration has two function calls, one for determining the value of $f(x_n)$ and the other for $f'(x_n)$. The use of the analytical first derivative (2.3) does not converge at all when using the absolute value of $|J|$. However, it does converge for J. Remarkably, an approximation of the first derivative works in both situations, but a backtracking line search such as the one in [43] has to be used to determine the step size. Of course, if we view the problem as an optimization problem it will fail in areas around the minima due to the cusp. Therefore, unlike what was mentioned in [38] we use Newton's method to find the root or the x-intercept of the cost function instead and not a minimum point even though the two occur at the same point.

It is also worth mentioning that although in [38] it is stated that the main bottleneck of the decoding algorithm is the optimization function, we have observed that the combination function (binomial coefficient) present in both the encoding and decoding function is also worthy of attention. An efficient combination function can result in a significant speed-up while more traditional approaches can render the algorithm impractical. With this in mind, in Fig. 2.5 and Table 2.1 we present an extended version of the result presented in [38] for the speed of decoding graphs based on the number of edges. The experimental setup is the same as the one used in Chapter 4 and the results represent the average of 100 test runs.

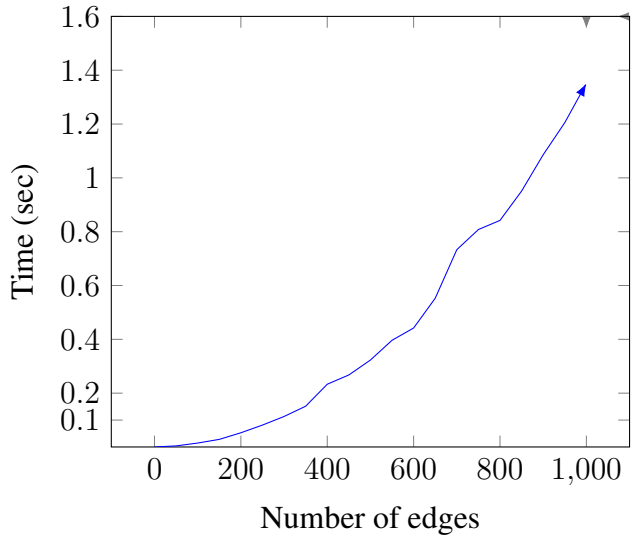


Figure 2.5: Graph decoding time based on the number of edges.

Table 2.1: Decoding time for graphs based on number of edges.

Number of edges	Time (msec)
50	3.986
100	14.583
150	28.219
200	52.760
250	81.374
300	113.294
350	151.430
400	233.13
450	267.624
500	323.068
550	396.435
600	442.087
650	552.681
700	733.098
750	808.002
800	841.869
850	950.945
900	1085.884
950	1205.199
1000	1350.885

CHAPTER 3

SGLC: A SUCCINCT GRAPH LOGICAL CLOCK

In this chapter, we introduce the algorithm for our proposed succinct graph logical clock (SGLC) and provide an example. The algorithm presented in this chapter first appeared in the conference proceedings of SIU 2022, the 30^{th} IEEE Conference On Signal Processing and Communications Applications.

3.1 SGLC Algorithm

The most intuitive and perhaps simplest way to track causality is by using directed acyclic graphs (DAGs). In such a case precedence determination is a constant time operation since the partial order is transitively closed and an edge exists for any ordered events[6]. Nonetheless, the standard representations of graphs are not efficient and quickly grow out of proportion in terms of not only storage overhead but also bandwidth exchange.

Relying on the coding and decoding algorithms from the previous chapter, we present the algorithm for SGLC. For clarity, we break it down into the following three scenarios:

1. What happens when there is an internal event?
2. What happens when a process sends a message?
3. What happens when a process receives a message?

The algorithm responds to the above three questions as follows:

1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), p_i increments its internal counter C_i. This is equivalent to recording a new event that happened at p_i.

2. When there is an internal event simply execute the first step.

3. Before a process p_i sends a message m to p_j, it adds/records the event in the form of adding a vertex to its causal history graph then encodes the graph (compresses the graph) using Algorithm 3.

4. Upon the receipt of a message m, process p_j decompresses the received graph using Algorithm 4 and then appends a new vertex to the decoded graph to capture the reception of the message.

Algorithm 5 gives the implementation of SGLC in an event drive simulator [44] to be introduced shortly.

3.2 An Example

As an example of how the algorithm works, an analysis of Fig. 3.1 is provided. Suppose that it is a distributed system with three processes, p_1, p_2 and p_3, events that occur within the same process form a chain. Therefore, at process p_1 the chain of events is $e_{11} \rightarrow e_{12} \rightarrow e_{13} \rightarrow e_{14}$. Taking a look at the causal history of event e_{22} which we denote as $H_{21} = \{e_{11}, e_{12}, e_{21}\}$ or in other words event e_{11} influences event e_{1w} which in turn influences event e_{21} and so on This is captured by the graph because before any process communicates with another process it compresses its causal precedence graph (a directed acyclic graph) as an integer (g) and sends it along with the number of edges of the graph (m). The receiving process has to first acknowledge the receipt of the message by incrementing its local counter and then decodes the graph. After decoding the graph it can then append this event to the causal history graph by adding a unique vertex v to the most recent event in the graph it just received. For example, at process 2 an edge is created between e_{21} and e_{12} where e_{12} is the last event in the received graph and e_{21} is the reception of the new message. This process is captured by the dashed lines in Fig. 3.1.

Algorithm 5 SGLC Algorithm

Implements: SGLC **Instance**: SC

OnInit: () **do**

1: $C_i = 0$

2: graph $LG = \{\}$

OnInternal: () **do**

3: $C_i = C_i + 1$

OnMessageFromBottom: $(m,\ g)$ **do**

4: $C_i = C_i + 1$

5: $G = decode(g)$

6: append G to LG

7: create vertex v with unique label based on C_i

8: find a vertex u in LG such that outdegree u = 0, add an edge from u to v in LG

9: **Trigger** SC.Sendup (m)

OnMessageFromTop: (m) **do**

10: $C_i = C_i + 1$

11: create vertex v with unique label based on C_i

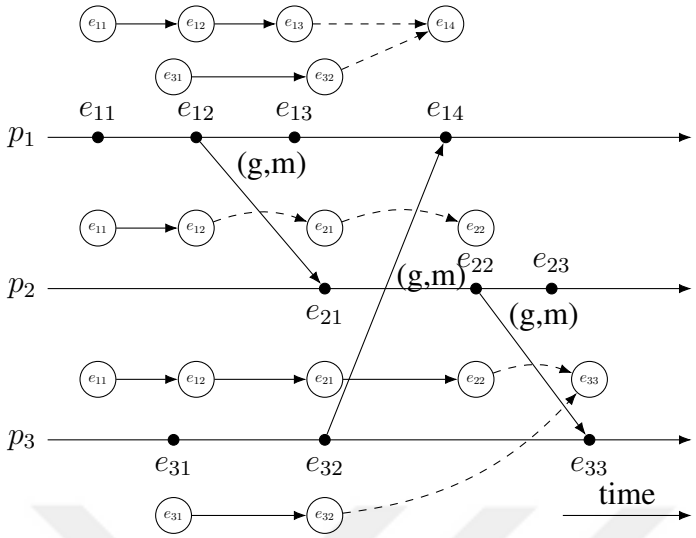
12: find a vertex u in in LG such that outdegree u = 0,

13: add an edge from u to v in LG

14: $g = encode(LG)$

15: **Trigger** SC.Senddown $(m,\ g)$

Figure 3.1: An example interaction between events.



CHAPTER 4

SIMULATION ENVIRONMENT AND EVALUATION OF SGLC

In this chapter, we present the simulation environment used to implement and test SGLC. We then give the results obtained after comparing SGLC with the vector clock. Finally, we discuss some challenges with using SGLC and propose some ways of overcoming those challenges.

4.1 Simulation Environment

To implement and test SGLC we relied on an Ad Hoc Computing Framework (AHC) [44]. This is a simple and flexible yet powerful open-source python event-driven simulation environment. The fundamental building block for all abstractions in AHC is a component, which is a single-threaded process that interacts with other processes through queues. This interaction is event-driven and all other blocks are built (and/or inherit) from the component model. Each component defines event handlers that respond accordingly to events relevant to them. The events can be anything such as a message from a higher-level layer (more on that later) or another component on the same layer. Furthermore, each component has initialization handlers (constructors) that dictate what happens when a component is initialized for the first time. The relationship between events and components is that components produce events, therefore becoming their source. Each event object has a set of attributes that characterize it: the event source mentioned previously, a name, creation time, and so on. A component relies on queue-handlers to trigger the right event handler at the occurrence of an event. AHC enables the stacking of components either one on top of another or side by side to create more complex components. These "connected" components can

connect with each other through connectors. AHC provides three default connectors, namely, “UP”, “DOWN””, and “PEER”. As the names imply, up connectors make it possible for a component to interact with components immediately above it while the “DOWN” connectors (or channels) make it possible for a connector to communicate with components that are immediately below it. The “PEER” connector enables the exchange of information between peers that are on the same level. These connectors make it possible to create a hierarchical stack such as the Open Systems Interconnection (OSI) model or the IP reference model.

The ability to stack components either vertically or horizontally in AHC makes it flexible and enables the user to generate a variety of topologies for different applications such as Ad hoc, distributed, and wireless systems. This is further simplified by the integration of the Networkx python package enabling the conversion of a wide variety of graphs to a topology meeting different requirements. To keep track of components, AHC also provides a Registry function that keeps track of the creation of all components from the start of an experiment/simulation.

To better understand how our clocks will make use of the connectors to exchange messages, it is important to understand the message structure provided by AHC. Each message exchanged between components or nodes has a structured pattern, similar to what you get in a packet-switched network. They have a message header and payload, which can be inherited from the GenericMessageHeader and GenericmessagePayload classes readily available. If there are more specific needs, custom message headers and payload classes can be implemented.

In addition, AHC also provides channel models, which can be more intuitively viewed as the stages messages pass through before reaching their destination. The stages provided by default are three: OnMessageFromTop, INCH, and DLVR. Each stage is controlled by a separate event handler and different operations can be performed on messages as they pass through these stages. However, it should be noted that there is no guarantee of keeping the order of messages generated by different components since they are handled by different threads.

The above description of AHC is merely a recapitulation of what it provides to show that it serves as a suitable development environment to properly test SGLC. The in-

terested reader is referred to `https://github.com/cengwins/ahc` for more details.

4.2 System Model

From the range of topologies available to us in AHC, it is straightforward to model a distributed system. Therefore, we model it as a connected undirected graph $G(V, E)$ where V represents the set of distributed nodes and E the set of the edges which serve as the communication channels. A process p_i is able to communicate with another process p_j if there is a channel between p_i and p_j. We implement the succinct clock as a service in the middleware that can be utilized by both higher and lower-level layers. Events that make use of the succinct clock are classified into three: internal, send, and receive events. The simulation while keeping track of internal events also keeps track of the sending and receiving of messages at each process. Furthermore, each event is uniquely identifiable globally through the formula $(k * n) + j$, with k being the process ID, n the maximum number of events possible at each process, and j the current event (clock counter) at a process. Below we elaborate further on some of the abstractions we make use of.

1. **Channels**: processes/nodes leverage on channels to exchange messages between each other. The channels we make use of are point-to-point, bidirectional, and reliable. Nevertheless, as pointed out earlier, AHC makes it possible to easily manipulate messages passing through channels.

2. **Processes**: a distributed system comprises a collection of computing units, which we build using components. Each of these units is abstracted through the notion of a process. Multiple processes together form a node. Processes from the same node or different nodes interact with each other in different ways.

From the onset, we have used processes and nodes interchangeably. Generally, nodes are larger and are made up of many processes (which in turn are made up of components). A computation can be made up of many nodes. As we consider asynchronous systems made up of n processes $p_1, \ldots, p_n$ where the identity of process p_i is its index

i. This enables us to distinguish between processes since we assume that each process has its own identifier. Furthermore, each process p_i has a set of neighbors, denoted $neighbors_i$. Based on the functions provided by AHC, this set contains the identities of these processes. In summary, a node can be viewed as a Turing machine with the added capability of being able to send and receive messages.

On a more holistic note, it is worth mentioning that in our scheme a node is not required to maintain the entire causal (history) graph of the distributed computation, but only the one relevant to it. Lastly, the following assumptions have been made:

1. There is no interaction between processes except through the exchange of messages
2. Before each experiment, the number of events that can happen at each process is determined by the user i.e the simulation is bounded by the number of events.
3. There isn't a global clock.
4. Communication between processes is point-to-point and reliable.
5. A process randomly determines the process it is going to communicate with from its neighbors.
6. Both messages and internal events experience random delays.
7. Each event has a globally unique identifier generated using only local information.

4.3 Evaluation of SGLC

Based on the above system model, we implement SGLC using the AHC framework and compare it with the vector clock. All experiments were conducted on an Intel dual-core i5 2.4 GHz PC running macOS Big Sur with 8GB memory using python 3.9.

The comparison of SGLC to the vector clock is not straightforward due to a fundamental difference between them. While the vector clock depends on the number

of processes, it is independent of the number of events. SGLC on the other hand is independent of the number of processes and dependent on the number of events. Consequently, the most reasonable way to compare the two is through the aggregate size of messages exchanged over the whole distributed computation. With this in mind, we performed different experiments with a varying number of processes ranging from 4 to 32 while adjusting the number of events from 20 to 100. Figs. 4.1–4.4 show the average results obtained over 100 test runs [45]. From Fig. 4.1 it can be seen that SGLC initially consumes less bandwidth than the vector clock, but then as the number of events increases, the vector clock consumes fewer bits. This is the overall trend that is expected especially since SGLC depends on the number of events. Nevertheless, how quickly this happens continues to decrease as the number of processes is increased as can be seen in Figs. 4.2– 4.4. In order to clearly see this, we combine the 4 figures using a 3 dimensional plot as seen in Fig. 4.5 and provide a cross-section of the plot in Fig. 4.6. The z-axis represents the bits exchanged, the y-axis represents the number of events, and the x-axis the number of processes. The change in color intensity reflects the increase in bits exchanged for vector clocks as the number of processes increases. On the other hand, the intensity stays the same throughout for the SGLC as seen underneath.

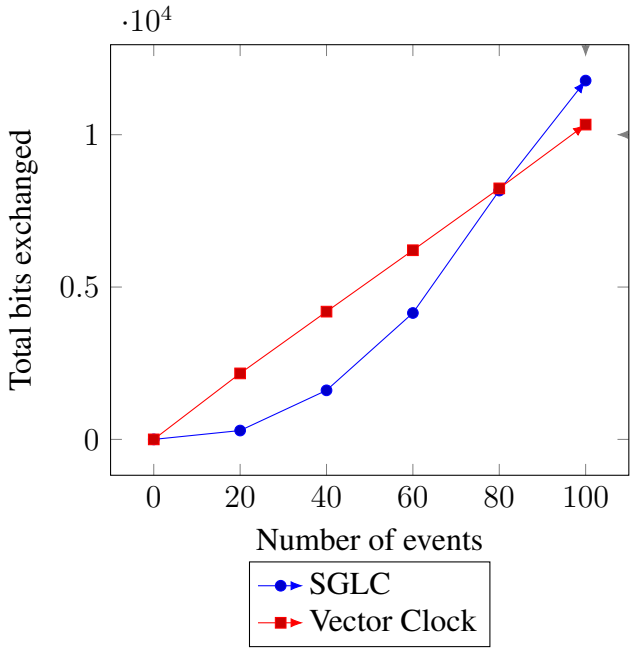


Figure 4.1: Aggregate message sizes of vector clock and SGLC for 4 processes.

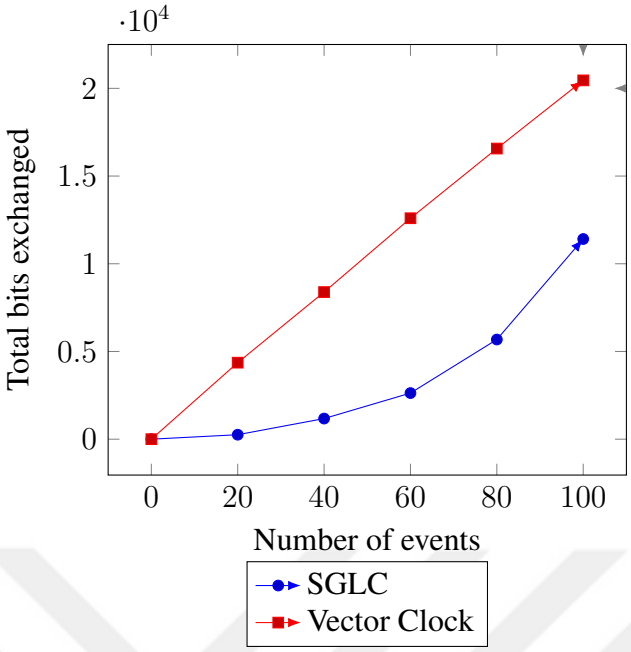


Figure 4.2: Aggregate message sizes of vector clock and SGLC for 8 processes.

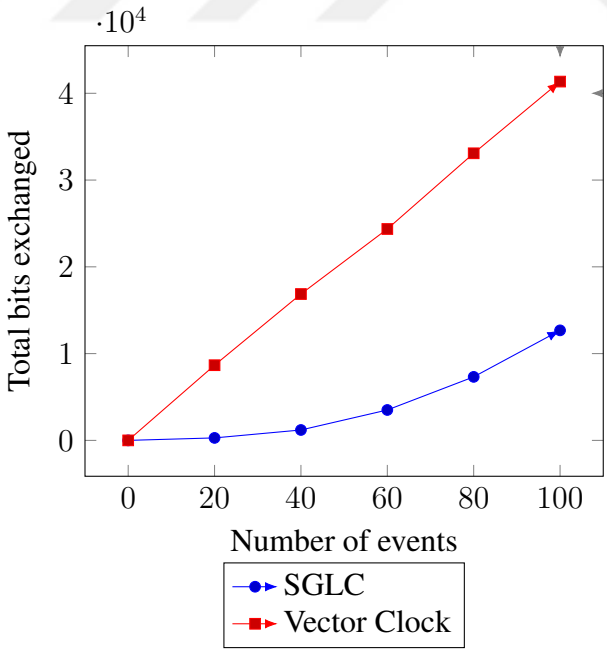


Figure 4.3: Aggregate message sizes of vector clock and SGLC for 16 processes.

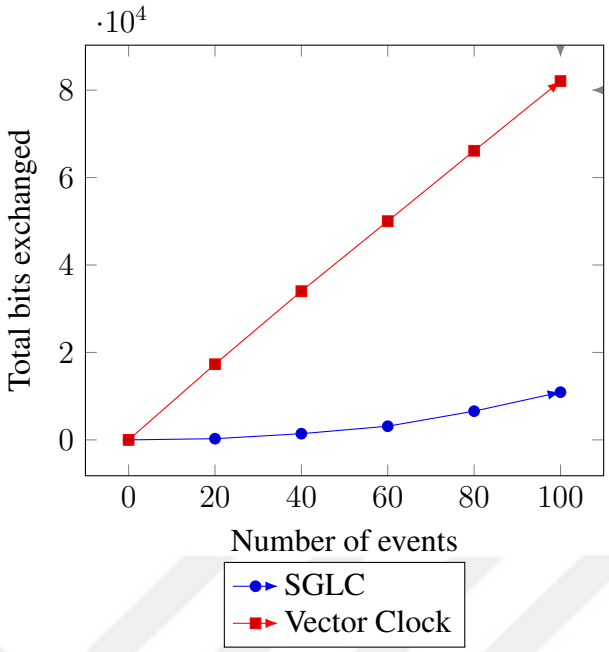


Figure 4.4: Aggregate message sizes of vector clock and SGLC for various 32 processes.

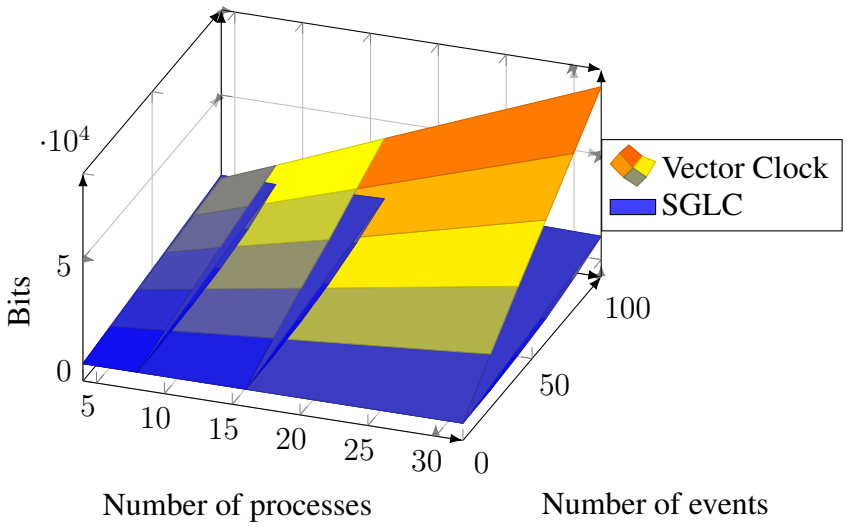


Figure 4.5: Aggregate message sizes of vector clock and SGLC for various number of processes and events.

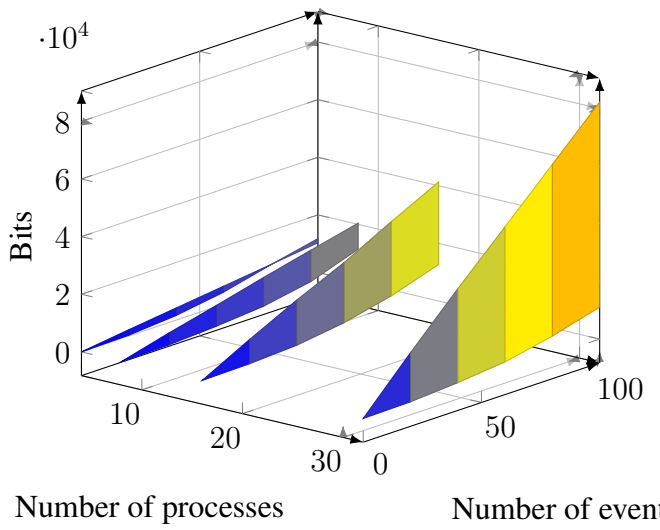


Figure 4.6: Cross-section view of aggregate message sizes of vector clock and SGLC for various processes.

When proposing a new distributed algorithm, factors that are usually considered are time, space, message, and bit complexity. The main focus of our proposal is on reducing the overall message sizes, which we achieved at the expense of greater time complexity incurred due to the graph encoding and decoding that takes place before sending or receiving a message. Nevertheless, it should be noted that it takes around 0.15 ms to decode a graph with 100 edges as shown in Table 2.1. Furthermore, the decoding algorithm can be parallelized for greater speed as mentioned in [38, 46].

4.4 Scalability Issues With SGLC

The vector clock is suitable if the number of processes is small. However, as pointed out earlier, it becomes unattractive as the number of processes increases. It is known to increase linearly in size for each additional node and has a storage overhead of $O(N)$ with respect to the number of nodes. On the other hand, the advantage provided by SGLC is in terms of bandwidth (information piggybacked on messages). It encodes causal graphs as integers that require $\log_2 \binom{n(n-1)}{m}$ bits, where n here is the number of events in the graph and m the number of edges. It is both an advantage and disadvantage that SGLC is dependent on the number of events. If there is a large number of processes with sparse interaction then the graph will not be very large.

However, as the number of events becomes larger the graph will also become larger eventually making SGLC worse than the vector clock even in terms of bit complexity. To have a better understanding of this we give the storage complexity of the vector clock and SGLC together with the complexity of some common operations performed on them in Table 4.1. Although we restrict the complexities to only the adjacency list and adjacency matrix, it should be clear that the storage will be a problem irrespective of the graph representation provided it allows for meaningful graph operations. In light of this, in the upcoming sections, we explore multiple techniques similar to those presented in [7, 27] that will not only reduce the bit complexity but also reduce the storage size of the graphs.

Action	Vector clock	SGLC	
		Adjacency list	Adjacency matrix
Clock increment	$O(1)$	$O(1)$	$O(1)$
Comparison	$O(N)$	–	–
Vertex addition	–	$O(1)$	$O(V^2)$
Edge addition	–	$O(1)$	$O(1)$
Edge query	–	$O(V)$	$O(1)$
Decoding	–	$O(E)$	
Storage	$O(N)$	$O(V+E)$	$O(V^2)$

Table 4.1: Complexity of some operations performed on vector clock and SGLC.

4.4.1 Incremental Piggybacking

The term incremental piggybacking was first used in [7]. The idea is that the sender does not always include a complete version of its causal history graph in all the messages sent. If the sender say, p_i, has previously interacted with a node p_j, it needs to only send the new events since the last interaction i.e. for any node p_i with graph $graph_i(e_x)$, this graph is a proper subset of any graph $graph_i(e_{x+1})$. Thus, each node p_i that communicates with another node p_j stores the last event that was included in the graph sent to p_j. Later, when p_i communicates with p_j it

piggybacks only $graph_i(e_{x+1})$ - $graph_i(e_x)$ on the message being sent. Considering Fig. 4.7 and focusing only on the interaction between process p_1 and p_2. The first time p_1 communicates with p_2 it sends a graph with a node set of $\{e_{11}, e_{22}\}$, therefore the next time it communicates with process p_2 its causal history consists of $\{e_{11}, e_{22}, e_{13}, e_{31}, e_{32}, e_{31}, e_{14}\}$. Since $\{e_{11}, e_{22}\}$ was sent in the previous message it does not have to be included. Fig. 4.8a gives the graph at process p_1 while Fig. 4.8b shows the actual graph sent to process p_2.

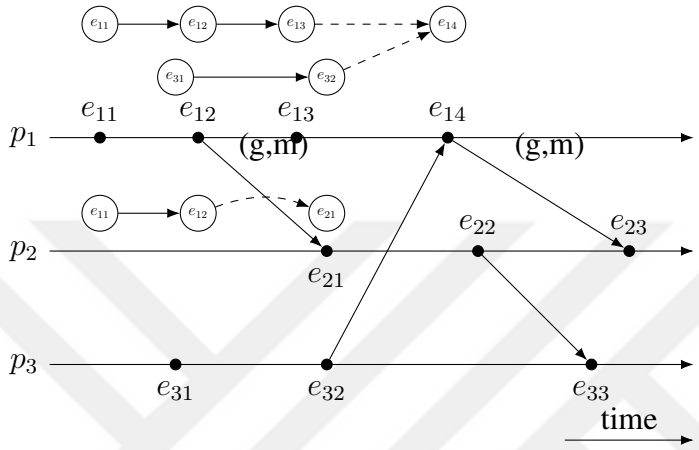
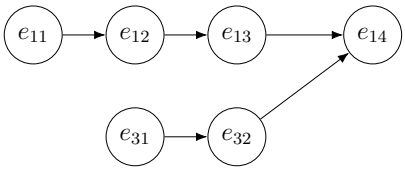
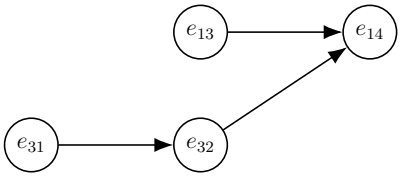


Figure 4.7: An example interaction between events.



(a) Graph at process p_1.



(b) Graph sent to process p_2.

Figure 4.8: Application of incremental piggybacking.

To see the impact of incremental piggybacking on the bit complexity of messages we reconducted the experiments from Section 4.3. Since we know that SGLC is

independent of the number of processes, it suffices to present only the results obtained for 4 processes while varying the number of events from 20 to 100. Similar to earlier experiments, we present the average of 100 test runs in Table 4.2. From the results, it can be seen that incremental piggybacking is more effective as the number of events increases and with 100 events a further reduction of 62.7% was realized compared to using SGLC without any further optimization.

Number of events	SGLC (bits)	SGLC with incremental piggybacking (bits)	Reduction percentage (%)
20	215.3	179.24	16.7%
40	1160.33	740.58	36.2%
60	2660.46	1448.66	45.6%
80	4664.1	2187.40	53%
100	10199.28	3800.31	62.7%

Table 4.2: Result of applying incremental piggybacking to SGLC.

4.4.2 Relevant Events

Another solution to reducing the size of the causal history graph is for SGLC to keep track of only those events that are relevant to the application. This strategy is utilized in the context of the immediate predecessor problem in [13] and revisited in the next chapter. Such an idea can also be found in the context of predicate detection [8]. Another example in the context of synchronization events in the MPI application in which the clock ticks at the occurrence of synchronization events such as lock, unlock, fork, and join events is provided in [27]. Without loss of generality, Algorithm 6 provides a simplified version of SGLC capable of capturing relevant events with the assumption that relevant events are comprised of only internal events.

Algorithm 6 SGLC for relevant events

Implements: SGLC **Instance**: SC

OnInit: () **do**

1: $C_i = 0$

2: graph $LG = \{\}$

OnInternal: () **do**

3: $C_i = C_i + 1$

OnMessageFromBottom: $(m,\ g)$ **do**

4: $G = decode(g)$

5: append G to LG

6: **Trigger** SC.Sendup (m)

OnMessageFromTop: (m) **do**

7: $g = encode(LG)$

8: **Trigger** SC.Senddown $(m,\ g)$

4.4.3 Resetting SGLC

Despite the effectiveness of both incremental piggybacking and keeping track of only relevant events, the most effective way to reduce the storage overhead of causal history graphs might be through resetting SGLC. The aim is to have bounded clock counters to tame the growing storage requirements of SGLC without affecting the correctness of applications. Therefore, the clock resetting techniques from [11, 47, 12] can be adapted. Depending on how this is achieved, it is up to the program designer to determine when and how the resetting takes place. For instance, using the protocol from [11], the reset is based on a "reset rule" that is enforced by a "reset line" which helps to prevent incomparable (clocks from different views) clocks from working side by side and leading to inconsistent clocks. Relying on [47] requires that an upper bound/limit be set on the clocks and the introduction of phase numbers to accompany timestamps. These phase numbers keep track of the number of times clocks have

been reset in order to achieve distinct non-overlapping clocks for each phase. Finally, using the solution [12] makes it possible to reuse timestamps through non-blocking resets in phase-based applications provided certain conditions termed as "contracts" are met.

These three possible solutions despite being geared towards vector clocks should also apply to SGLC. Therefore, it might even be more reasonable to set the limit as being the size of the vector clocks i.e. either $32n$ or $64n$ bits depending on the encoding of the vector. Doing this will ensure that the storage of SGLC does not exceed that of the vector clock. Eventually, the choice depends on whether you want a network-wide reset of all clocks or only independent local resets.

CHAPTER 5

CASE STUDIES

In this chapter, we use our proposed SGLC in two applications, namely, predecessor determination and causal order broadcast.

5.1 Predecessor Determination

The notion of finding immediate predecessors play an important role in the analysis of distributed computations. The term immediate predecessor tracking first appeared in [10], however, before that other solutions for the problem were proposed in [48, 49]. These solutions all make use of the vector clock. With our proposed SGLC, this is simplified and there is no need to send a vector of bits along with the vector clock or other such solutions.

Definition 5.1.1. *[10] The immediate predecessor Tracking (IPT) problem has to do with the on the fly determination of an event's immediate predecessors without relying on any additional control information.*

In other words, we say that x is covered by y in P if $\forall z \in P$ such that$(x \leq z \leq y) \implies (z = y)$. x is an immediate predecessor of y in P and y an immediate successor of x in P. The directed graph associated with this covering is equivalent to the transitive reduction of P. The notion of relevant events discussed in the previous chapter simplifies the computation of the transitive reduction. Nevertheless, even with the transitive reduction, we need to create a topological ordering of the graph to determine the immediate predecessor. This is an $O(n + v)$ operation that can be tolerated if the number of events is small. But as the number of events increases, it

becomes impractical. In light of this, it would be better to send the identifier of the last produced event along with the graph.

5.2 Causal Order Broadcast

Many applications rely on a node sending a message to all other nodes in a distributed system. This makes the use of point-to-point communication links impractical. As a result, two other models of communication are available: broadcast and multicast. In broadcast, a message from a source is sent to all other nodes in the system while in multicast a message from a source is sent to a subset of the nodes in the system. When messages are broadcast by nodes in the system, there are three properties of interest, namely; reliability, consistent ordering, and causality preservation [50]. These three properties bring forth three different types of broadcast primitives: reliable, atomic, and causal broadcast. Broadcast serves as a building block for many applications. In this sense, a broadcast is an abstraction that is not an end in itself but is needed for building other applications mostly related to fault tolerance and reliability. Our focus will be on causal order broadcast or causal broadcast in short. Causal broadcast ensures that all processes in a distributed system deliver a broadcast message per the happened before relation. Its application can be found in [51, 52, 53, 54, 55]. For completeness, we give a glimpse into one such usage by making use of a distributed database from [50] as an example. Let a node p_i send a broadcast request followed by a normal message to another node p_j. After the reception of this message, node p_j makes its request for a certain operation on the database. It might be the case that node p_j uses the information it received from p_i or possibly on a database operation already performed by p_i. Consequently, the request made by p_j should be performed on the version of the database after the request by p_i has taken effect. In such a situation, causal broadcast is required. Causal broadcast requires that the order in which the messages are delivered is consistent with the partial order defined by the "happened before" relation to the events corresponding to the sending of the messages. That is, if node p_i delivers m_1, then p_i must have delivered every message causally preceding $(\rightarrow)$ m_1 before delivering m_1. Likewise, if $S(m_1)$ is the sending of a message m_1, causal broadcast requires that if $s(m_1) \rightarrow s(m_2)$, then m_1 is delivered before m_2 by

all nodes.

A message is received by a node when it arrives over the communication medium. The received message is then delivered, for instance to another process or layer. It is important to distinguish between receiving a message and delivering it. A node has control over when it delivers a message to an application and can therefore control the order of delivery, but it has no control over the order in which messages are received. As a result, the causal order broadcast protocol we propose below ensures that the order in which messages are delivered is consistent with the causal ordering captured by SGLC. However, the algorithm below only ensures a weak causal order. It does not guarantee the total ordering of all messages at all nodes. That is, if two messages m and m' are such that $m \rightarrow m'$, then m is delivered before m' at all nodes. However, if there is no relation between m and m', then m and m' may be delivered in any order, which may be different for different nodes.

To support broadcast, a node p_i maintains a buffer that contains all the messages sent to it. From the buffer, messages are put on a delivery queue from which the application process can receive the messages. The use of SGLC ensures that messages are added to the delivery queue in an order which is consistent with the causal order of the messages. To see this, say a node p_i performs a broadcast of a message m. This message is sent along with SGLC to all other nodes. When a node p_j receives the message m, it adds it to its buffer which is ordered based on SGLC. Messages in the buffer are then put onto the delivery queues of p_j. We assume that lower layers (this is actually the case) take care of duplicate packets and retransmissions, therefore allowing us to focus on only the causal broadcast.

In order to see how this supports causal order broadcast, suppose a message m is sent by a node p_i and a message m' is sent by a node p_j which could be the same as p_i. If m precedes m', it implies that there is a sequence of broadcast messages $m_0, m_1, \ldots, m_n$ such that $m = m_0$ and $m' = m_n$, which appear in the order in which they occur. The buffers used to store (broadcast) messages are arranged based on this order.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The most intuitive way to explain, view, and reason about causal relationships is through directed acyclic graphs as is commonly done using space/time diagrams. Consequently, this thesis explores the possibility of using a succinct representation of graphs to implement logical clocks to keep track of causal relationships. A quantitative comparison between our proposed SGLC and the vector clock shows a reduction in overall bits exchanged especially as the number of processes/nodes increases in the distributed system. Going further, we also explore further optimization techniques to lower the bandwidth consumption of our proposed clock and ways to overcome the storage cost associated with using graphs as logical clocks.

Detailed analysis and implementation of the proposed ways of reducing the local storage complexity of SGLC should be performed as the next step to consolidating the work done in this thesis. Other application areas of SGLC should also be explored.

REFERENCES

[1] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distrib. Comput.*, vol. 7, no. 3, p. 149–174, Mar. 1994. [Online]. Available: https://doi.org/10.1007/BF02277859

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

[3] M. Raynal, "About logical clocks for distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 26, no. 1, p. 41–48, Jan. 1992. [Online]. Available: https://doi.org/10.1145/130704.130708

[4] R. Hoettger, B. Igel, and E. Kamsties, "Vector clock tracing and model based partitioning for distributed embedded systems," *International Journal of Computing*, vol. 12, no. 4, pp. 324–332, 2013.

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, jul 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[6] P. A. S. Ward, "A framework algorithm for dynamic, centralized dimension-bounded timestamps," in *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '00, 2000, p. 14.

[7] E. N. Elnozahy, *Manetho: fault tolerance in distributed systems using rollback-recovery and process replication*. Rice University, 1994.

[8] M. Shen, A. Kshemkalyani, and A. Khokhar, "Detecting unstable conjunctive locality-aware predicates in large-scale systems," in *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*, 2013, pp. 127–134.

[9] T.-D. Diep, K. T. Pham, K. Fürlinger, and N. Thoai, "A time-stamping system to detect memory consistency errors in mpi one-sided applications,"

Parallel Computing, vol. 86, pp. 36–44, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819118303235

[10] E. Anceaume, J.-M. Hélary, and M. Raynal, "A note on the determination of the immediate predecessors in a distributed computation," *International Journal of Foundations of Computer Science*, vol. 13, no. 06, pp. 865–872, 2002.

[11] L.-H. Yen and T.-L. Huang, "Resetting vector clocks in distributed systems," *Journal of Parallel and Distributed Computing*, vol. 43, no. 1, pp. 15–20, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731597913307

[12] A. Arora, S. S. Kulkarni, and M. Demirbas, "Resettable vector clocks," *Journal of Parallel and Distributed Computing*, vol. 66, no. 2, pp. 221–237, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731505001759

[13] M. Raynal, *Distributed algorithms for message-passing systems*. Springer, 2013, vol. 500.

[14] S. E. P. Hernández, "The minimal dependency relation for causal event ordering in distributed computing," *Applied Mathematics & Information Sciences*, vol. 9, no. 1, pp. pp–57, 2015.

[15] P. Sérgio, A. Carlos, and B. V. Fonte, "Interval tree clocks: A logical clock for dynamic systems."

[16] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, 1991.

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

[18] F. B. Schmuck, "The use of efficient broadcast protocols in asynchronous distributed systems," CORNELL UNIV ITHACA NY DEPT OF COMPUTER SCIENCE, Tech. Rep., 1988.

[19] S. Meldal, S. Sankar, and J. Vera, "Exploiting locality in maintaining potential

causality," in *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, 1991, pp. 231–239.

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

[21] J.-M. Hélary, M. Raynal, G. Melideo, and R. Baldoni, "Efficient causality-tracking timestamping," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1239–1250, 2003.

[22] A. Agarwal and V. K. Garg, "Efficient dependency tracking for relevant events in shared-memory systems," in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 19–28. [Online]. Available: https://doi.org/10.1145/1073814.1073818

[23] V. K. Garg, C. Skawratananond, and N. Mittal, "Timestamping messages and events in a distributed system using synchronous communication," *Distributed Computing*, vol. 19, no. 5-6, pp. 387–402, 2007.

[24] P. A. Ward and D. J. Taylor, "A hierarchical cluster algorithm for dynamic, centralized timestamps," in *Proceedings 21st International Conference on Distributed Computing Systems*. IEEE, 2001, pp. 585–593.

[25] P. A. S. Ward and D. J. Taylor, "Self-organizing hierarchical cluster timestamps," in *Euro-Par 2001 Parallel Processing*, R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 46–56.

[26] X. Zheng and V. Garg, "An optimal vector clock algorithm for multithreaded systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 2188–2194.

[27] A. D. Kshemkalyani, M. Shen, and B. Voleti, "Prime clock: Encoded vector clock to characterize causality in distributed systems," *Journal of Parallel and Distributed Computing*, vol. 140, pp. 37–51, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731519304939

[28] G. T. Wuu and A. J. Bernstein, "Efficient solutions to the replicated log and dictionary problems," in *Proceedings of the third annual ACM symposium on Principles of distributed computing*, 1984, pp. 233–242.

[29] R. Baldoni, F. Quaglia, and B. Ciciani, "A vp-accordant checkpointing protocol preventing useless checkpoints," in *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*. IEEE, 1998, pp. 61–67.

[30] L. M. Drummond and V. C. Barbosa, "On reducing the complexity of matrix clocks," *Parallel Computing*, vol. 29, no. 7, pp. 895–905, 2003.

[31] F. J. Torres-Rojas and M. Ahamad, "Plausible clocks: constant size logical clocks for distributed systems," *Distributed Computing*, vol. 12, no. 4, pp. 179–195, 1999.

[32] L. Ramabaja, "The bloom clock," *arXiv preprint arXiv:1905.13064*, 2019.

[33] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting, "Preserving and using context information in interprocess communication," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 3, pp. 217–246, 1989.

[34] Y. Amir, D. Dolev, S. Kramer, and D. Malki, *Transis: A communication subsystem for high availability*. Citeseer, 1991.

[35] A. S. Tanenbaum and M. Van Steen, *Distributed Systems*. Pearson Education, 2017.

[36] R. Baldoni, R. Beraldi, R. Friedman, and R. Van Renesse, "The hierarchical daisy architecture for causal delivery," *Distributed Systems Engineering*, vol. 6, no. 2, p. 71, 1999.

[37] D. Haban and W. Weigel, "Global events and global breakpoints in distributed systems," in *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Volume II: Software track*, vol. 2. IEEE Computer Society, 1988, pp. 166–167.

[38] V. Parque and T. Miyashita, "On succinct representation of directed graphs," in

2017 IEEE International Conference on Big Data and Smart Computing (BigComp), Feb 2017, pp. 199–205.

[39] G. Kawatani, "Graph representation of partially ordered sets," Ph.D. dissertation, National Informatics Institute, 2017.

[40] W. D. Wallis, *A beginner's guide to discrete mathematics*. Springer Science & Business Media, 2011.

[41] K. Thulasiraman and M. N. Swamy, *Directed Graphs*. John Wiley & Sons, Ltd, 1992. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118033104.ch5

[42] V. Parque and T. Miyashita, "Unranking combinations using gradient-based optimization," in *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2018, pp. 579–586.

[43] T. T. Truong and H.-T. Nguyen, "Backtracking gradient descent method and some applications in large scale optimisation. part 2: Algorithms and experiments," *Applied Mathematics & Optimization*, vol. 84, no. 3, pp. 2557–2586, 2021.

[44] E. Onur, "Ad hoc computing (ahc) framework," https://github.com/cengwins/ahc, 2022.

[45] S. Sokoto and E. Onur, "Sglc: A logical clock using succinct graphs," in *2022 30th Signal Processing and Communications Applications Conference (SIU)*, 2022.

[46] V. Parque and T. Miyashita, "Towards the succinct representation of m out of n," in *Internet and Distributed Computing Systems*, Y. Xiang, J. Sun, G. Fortino, A. Guerrieri, and J. J. Jung, Eds. Cham: Springer International Publishing, 2018, pp. 16–26.

[47] A. Mostefaoui and O. Theel, *Reduction of timestamp sizes for causal event ordering*. IRISA, 1996.

[48] C. Jard and G.-V. Jourdan, “Incremental transitive dependency tracking in distributed computations,” *Parallel Processing Letters*, vol. 6, no. 03, pp. 427–435, 1996.

[49] C. Diehl, C. Jard, and J.-X. Rampon, “Reachability analysis on distributed executions,” in *Colloquium on Trees in Algebra and Programming*. Springer, 1993, pp. 629–643.

[50] P. Jalote, *Fault tolerance in distributed systems*. Prentice-Hall, Inc., 1994.

[51] B. Nédelec, P. Molli, and A. Mostefaoui, “Crate: Writing stories together with our browsers,” in *Proceedings of the 25th International Conference Companion on World Wide Web*, ser. WWW ’16 Companion. International World Wide Web Conferences Steering Committee, 2016, p. 231–234. [Online]. Available: https://doi.org/10.1145/2872518.2890539

[52] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, “Epidemic algorithms for replicated database maintenance,” in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 1–12.

[53] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of convergent and commutative replicated data types,” Ph.D. dissertation, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[54] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Bolt-on causal consistency,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 761–772.

[55] M. Bravo, L. Rodrigues, and P. Van Roy, “Saturn: A distributed metadata service for causal consistency,” in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 111–126.