POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	On Line Trace Synchronization for Large Scale Distributed Systems
Auteur: Author:	Masoume Jabbarifar
Date:	2013
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Jabbarifar, M. (2013). On Line Trace Synchronization for Large Scale Distributed Systems [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1312/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1312/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

ON LINE TRACE SYNCHRONIZATION FOR LARGE SCALE DISTRIBUTED SYSTEMS

MASOUME JABBARIFAR
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
NOVEMBRE 2013

© Masoume Jabbarifar, 2013.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

ON LINE TRACE SYNCHRONIZATION FOR LARGE SCALE DISTRIBUTED SYSTEMS

présentée par : JABBARIFAR Masoume
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

Mme BOUCHENEB Hanifa, Doctorat, présidente
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. GAGNON Michel, Ph.D., membre
M. GOSWANI Dhrubajyoti, Ph.D., membre

I dedicate my dissertation work to :

my loving parents, Batoul and Norouzali
whose words of encouragement while
I am far away of them and they miss me a lot,

my husband, Alireza,
has never left my side and supported me
throughout the process with his special love,

my beautiful princess daughter, Liana,
whose smile is my everything

I would have never been here without your support.

ACKNOWLEDGEMENTS

I would like to express my deep and sincere gratitude to my supervisor Professor Dr. Michel Dagenais for his endless support, understanding, kindness, and great supervision. His guidance and ideas opened new doors into new aspects of distributed systems tracing for me. He also gently added some constraints and restrictions into the method of thinking which have been very beneficial and accelerated my research process. During my PhD studies in Ecole Polythechnique de Montreal, I learned a lot from him not only about research but also about life.

Many thanks go to the committee members who were more than generous with their expertise and precious time. A special thanks to Dr. Hanifa Boucheneb, committee chairman, for her countless hours of reflecting, reading, encouraging, and most of all patience throughout the entire process. Thank you Dr. Michel Gagnon, Dr. Dhrubajyoti Goswami, and Dr. Robert Legros for agreeing to serve on my committee.

Thanks to Ericsson and the Natural Sciences and Engineering Research Council of Canada for funding this research.

I would like to thank my friends and colleagues at the DORSAL laboratory of the department of Computer and Software Engineering. All of you have been my best cheerleaders.

I wish to thank my mother Batoul Tabaghlou-Sorkhab and my father Norouzali Jabbarifar. No words can express and no act of gratitude can relay what their love and support have meant to me. I hope they accept this as an indication of my heartfelt appreciation for everything they are.

I would like to thank my brothers Rasoul and Davoud, my sisters Farkhonde, Soraya, and Fahime and their lovely families, my nephews Vahid and Saeed and their nice spouses, Ali, and Mohammad, and finally my nieces Maryam, Pegah, Maedeh, Mahya, Zahra, Parastou, and Dina. Without their support and encouragement, my efforts to complete this dissertation would not have been possible.

I am very thankful to my mother-in-law and father-in-law, Parvin and Khanali Shameli-Sendi. I am also grateful to my brothers-in-law Mohammad and his family, and Sajad, and my sisters-in-law Zahra and Fatemeh for their unconditionally loving and supportive energy at all times.

Finally, but most importantly, I am indefinitely indebted to my husband, Alireza, who has put up with me for reasons not always obvious, endured countless sacrifices so that I can follow my dreams and supported me in any way that he could. I owe all of my accomplishments to him.

RÉSUMÉ

Les systèmes distribués en réseau fournissent une plate-forme informatique polyvalente pour soutenir diverses applications, telles que des algorithmes de routage dans les réseaux de télécommunication, les systèmes bancaires dans les applications de réseau, les systèmes de contrôle d'aéronefs dans le contrôle de processus en temps réel, ou le calcul scientifique, y compris les grilles et grappes de calcul en calcul parallèle. Ces systèmes sont généralement supervisés afin de détecter, de déboguer et d'éviter les problèmes de sécurité ou de performance. Un outil de traçage est une des méthodes les plus efficaces et précises, avec laquelle toutes les informations détaillées pour chaque noeud individuel dans le système peuvent être extraites et étudiées.

Typiquement, une tâche énorme est divisée en de nombreuses tâches, qui sont distribuées et exécutées sur plusieurs ordinateurs coopérant en réseau. Ainsi, afin de contrôler la fonctionnalité des systèmes distribués actuels, toutes les informations sont collectées à partir de plusieurs systèmes et appareils embarqués pour une analyse et une visualisation à la fois en ligne et hors ligne. Cette information de traçage, générée à un rythme effarant, est livrée avec estampilles de temps générées localement sur chaque noeud. Ces estampilles sont généralement fondées sur des compteurs de cycle, avec une granularité du niveau de la nanoseconde. Toutefois, les horloges de chaque noeud sont indépendantes et donc asynchrones les unes des autres. Néanmoins, les utilisateurs s'attendent à voir la sortie de l'analyse en temps réel, sur un axe de référence de temps commun, afin d'être en mesure de diagnostiquer les problèmes plus facilement.

La portée de l'oeuvre proposée ici est la synchronisation efficace et en direct de traces générées dans un environnement de grande grappe d'ordinateurs avec des estampilles de temps de granularité du niveau de la nanoseconde, produites par des horloges non synchronisées. Par ailleurs, le modèle de trafic du réseau, le nombre de noeuds informatiques disponibles et même la topologie du réseau peuvent changer. En effet, les grands centres de données roulent un ensemble diversifié et en constante évolution d'applications. Les noeuds peuvent échouer ou revenir en ligne à tout moment, et même le réseau peut être reconfiguré dynamiquement. Ainsi, motivé par la grande échelle des systèmes ciblés, le volume élevé de flux de traces de données associés, la limitation des tampons mémoire et la nécessité d'une analyse en direct, et la haute précision de synchronisation requise, nous avons conçu une nouvelle approche incrémentale pour synchroniser les traces de plusieurs ordinateurs connectés à un réseau dynamique à grande échelle.

Tout d'abord, nous présentons une nouvelle technique de synchronisation en direct des

connexions individuelles basée sur la classification rapide des paquets échangés, soit comme des paquets précis ou des paquets inintéressants. Cette méthode permet d'obtenir à la fois le plus bas coût de calcul, une latence minimale et une meilleure précision. Deuxièmement, nous avons proposé un algorithme efficace pour calculer incrémentalement l'arbre couvrant minimum des liaisons réseau avec la meilleure précision (plus faible inexactitude) afin de permettre le calcul efficace de paramètres de synchronisation transitive entre deux noeuds qui ne sont pas connectés directement. Ce problème est un défi multiple puisque l'exactitude des liens change au fur et à mesure que des paquets sont échangés entre deux noeuds, de nouveaux liens peuvent apparaître lorsque les noeuds commencent à échanger des paquets, et de nouveaux noeuds peuvent aussi apparaître. Enfin, nous avons proposé un nouvel algorithme pour identifier efficacement et mettre à jour le noeud de référence optimal dans l'arbre couvrant minimum, afin d'utiliser ce noeud comme référence de temps pour l'analyse et la visualisation des traces de plusieurs noeuds. En résumé, nous avons conçu et mis en oeuvre une nouvelle procédure efficace et complète pour la synchronisation de trace optimale, dans un environnement de très grande grappe d'ordinateurs, en direct.

Le Linux Trace Toolkit next generation (LTTng), développé à l'École Polytechnique de Montréal, offre une trace d'exécution détaillée des systèmes Linux avec faible surcharge. Notre nouvelle procédure a été programmée et validée par la synchronisation en ligne d'énormes traces LTTng dans de grands réseaux dynamiques.

ABSTRACT

Networked distributed systems provide a versatile computing platform for supporting various applications, such as routing algorithms in telecommunication networks, banking systems in network applications, aircraft control systems in real-time process control, or scientific computing including cluster and grid computing in parallel computation [61]. These systems are typically monitored to detect, debug and avoid security or performance problems. A tracing tool is one of the most efficient and precise methods, in which all the detailed information for every individual node in the system can be extracted and studied. Typically, a particular huge task is divided into many tasks, which are distributed and run on several cooperating networked computers. Hence, in order to monitor the functionality of current distributed systems, all information is collected, from multiple systems and embedded devices, for both online and a posteriori offline analysis and viewing. This tracing information, generated at a staggering rate, comes with timestamps locally generated on each node. These timestamps are typically based on cycle counters, with a nanosecond level granularity. However, the clocks in each node are independent and thus asynchronous from one another. Nonetheless, users expect to see the analysis output in real-time, on a common time reference axis, in order to be able to diagnose problems more easily.

The scope of the work proposed here is the efficient and live synchronization of traces generated in distributed systems with nanosecond granularity timestamps produced by unsynchronized clocks. Moreover, the pattern of network traffic, the number of available computer nodes and even the network topology can change. Indeed, distributed systems run a diverse and changing set of applications, nodes may fail or come back online at any time, and even the network can be reconfigured dynamically. Thus, motivated by the large scale of targeted systems, the high volume of associated trace data streams, the data buffering limitations, and the need for live analysis and high synchronization precision, we designed a new incremental approach to synchronize traces from multiple connected computers in a large scale dynamic network.

First, we present a novel schema for live synchronization of individual connections based on the fast classification of exchanged packets as either accurate packets or uninteresting packets. This method achieves at the same time the lowest computing cost, lowest latency and best accuracy. Secondly, we proposed an efficient algorithm to incrementally compute the minimum spanning tree of network links with the best precision (lowest inaccuracy) in order to allow the efficient computation of synchronization parameters transitively between two nodes which are not connected directly. This problem is a multiple challenge since the

accuracy of links changes as more packets are exchanged between two nodes, new links may appear when nodes start exchanging packets, and new nodes may appear as well. Finally, we proposed a new algorithm to efficiently identify and update the optimal reference node in the minimum spanning tree, in order to use this node as time reference when analyzing and visualizing traces from multiple nodes. In summary, we designed and implemented a new efficient procedure for optimum trace synchronization in a live distributed systems.

The Linux Trace Toolkit next generation (LTTng), developed at Polytechnique Montreal, provides a detailed execution trace of Linux systems with low overhead. Our new procedure was programmed and validated through the online synchronization of huge LTTng traces in large dynamic networks.

CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vii

CONTENTS ix

LIST OF TABLES xii

LIST OF FIGURES xiii

LIST OF SIGNS AND ABBREVIATIONS xv

CHAPTER 1 INTRODUCTION 1
- 1.1 LTTng 1
  - 1.1.1 LTTV and TMF 2
  - 1.1.2 Synchronization Architecture in LTTng 4
- 1.2 The Contributions of this thesis 6
- 1.3 General organization of the thesis 8

CHAPTER 2 LITERATURE REVIEW : A Comprehensive Survey of Techniques and Challenges in Distributed Systems Time Synchronization 9
- 2.1 Abstract 9
- 2.2 Introduction 9
- 2.3 Clock and Synchronization Protocols 11
  - 2.3.1 Time Keeping Hardware 12
  - 2.3.2 Packet-based Clock Offset Calculation 13
  - 2.3.3 Logical clock Synchronization 15
- 2.4 Synchronization techniques to compute clock offset and drift 15
- 2.5 Synchronization Applications 20
  - 2.5.1 Offline Clock Synchronization 20
  - 2.5.2 Online Clock Synchronization 22

2.6 Evaluation of protocols 23
2.6.1 Evaluation factors 23
2.6.2 Protocols comparison 25
2.7 Conclusion 26

CHAPTER 3 Paper 1 : Streaming Mode Incremental Clock Synchronization 28
3.1 Abstract 28
3.2 Introduction 28
3.3 Related Work 29
3.3.1 Offline Clock Synchronization 29
3.3.2 Online Clock Synchronization 31
3.4 Kernel-Level Event Tracing 32
3.4.1 Tracer 32
3.4.2 Time Stamp Counter 32
3.5 Terminology and background 33
3.6 Proposed Model 35
3.6.1 Model 36
3.6.2 Convex-Hull 38
3.6.3 Window-based Approach 40
3.6.4 Fully Incremental Approach 45
3.7 Experiments and evaluation 50
3.7.1 Experimental setup 50
3.7.2 Packet matching and Convex-Hull points 52
3.7.3 Accuracy and Cost 55
3.7.4 Delay and Packet loss effects on the Fully Incremental approach 58
3.8 Conclusion 60

CHAPTER 4 Paper 2 : Reference Node Selection in Dynamic Tree 61
4.1 Abstract 61
4.2 Introduction 61
4.3 Related Work 62
4.4 Data Structure 63
4.5 Methodology 65
4.5.1 Reference Node 65
4.5.2 Independent trees 65
4.5.3 Adding a single vertex and edge 66
4.5.4 Replacing an edge in a tree 66

4.5.5 Inserting an edge between two independent trees 73
4.6 Algorithm complexity 76
4.7 Experiments and evaluation 76
4.7.1 Experimental setup 76
4.7.2 Results 77
4.7.3 Performance evaluation 81
4.8 Conclusion 83

CHAPTER 5 Paper 3 : LIANA : Live Incremental Time Synchronization of Traces for Distributed Systems Analysis 86
5.1 Abstract 86
5.2 Introduction 86
5.3 Related Work 88
5.4 Terminology and background 89
5.5 Methodology 91
5.5.1 Two-node synchronization 91
5.5.2 Multi-hop synchronization 95
5.5.3 Dynamic Reference Node 98
5.5.4 Synchronization Factor Propagation 99
5.6 Experiments and evaluation 99
5.6.1 Simulation experiments 99
5.6.2 Real world traced network 101
5.6.3 Discussion 108
5.7 Conclusion 115

CHAPTER 6 GENERAL DISCUSSION 116

CHAPTER 7 CONCLUSION 120
7.1 Concluding Remarks 120
7.2 Future Research 121
7.2.1 Data integration from Virtual Machine 122
7.2.2 Hardware tracing 123

LIST OF REFERENCES 124

LIST OF TABLES

Table 2.1 Performance comparison of synchronization protocols 26
Table 3.1 The packet loss affection on Fully Incremental approach 58
Table 4.1 Number for each operation, from a total of one million operations . . . 79
Table 4.2 The result of proposed algorithm for six datasets in term of *RN* changes 80
Table 4.3 The status of join operation 81
Table 4.4 Number of descendantSize update in each operation 82
Table 5.1 Number of operations by type, out of a total of one million operations . 102
Table 5.2 Number of operations which affect and update the *MST*, out of a total of one million operations 102
Table 5.3 Dataset features and number of *RN* changes 106
Table 5.4 Time evaluation with the Non Incremental method 109
Table 5.5 The MST, RN, and conversion parameter update computation time with the Non Incremental [85] method applied on 2-second windows . . 110
Table 5.6 Decomposition of the execution time for the proposed method 113

LIST OF FIGURES

Figure 1.1 Synchronization view of LTTV . . . 3
Figure 1.2 Synchronization view of TMF . . . 5
Figure 1.3 Synchronization architecture . . . 6
Figure 2.1 SYNC message . . . 14
Figure 2.2 Convex-hull method. . . . 19
Figure 3.1 Two different approaches for online synchronization. . . . 37
Figure 3.2 Convex-hull method. . . . 39
Figure 3.3 The local clock values used for traces T0 and T1 may be highly desynchronized. Two traces starting about at the same time may see start times of 600sec. and 800sec. on their local clocks, respectively. With a window size of 3sec., the first window, W1, will go from 600sec. (minimum start time) to 803sec. (maximum start time plus window size). After processing the first time window, and analyzing matching events, it may be computed that trace T0 should be offset by -200sec., using T1 as time reference. The second time window, W2, is from 803sec. to 806sec., based on the reference time of T1. This corresponds to 603sec. to 606sec. in T0 based on its local clock. After synchronization, we realize that events in T0 for time range W2 have already been processed as part of W1. These already read events are skipped. . . . 44
Figure 3.4 Correlated sliding window. . . . 46
Figure 3.5 Fully Incremental Approach . . . 47
Figure 3.6 Geometric movement state in upper and lower hulls . . . 48
Figure 3.7 The number of matched packets in each window. . . . 53
Figure 3.8 The number of pairs in Convex-Hull in each window (Correlated approach). . . . 53
Figure 3.9 The number of pairs in Convex-Hull in each synchronization (Fully Incremental approach). . . . 54
Figure 3.10 Comparison of total pairs in Convex-Hull. . . . 54
Figure 3.11 Accurate packet rate . . . 56
Figure 3.12 Accurate packet distribution vs. time window enhancement . . . 56
Figure 3.13 Comparison of time synchronization approaches in streaming mode. . . . 59
Figure 3.14 Zoom on the accuracy dimension of Figure 3.13 from $1.2e^{-06}$ to $1.9e^{-06}$ and the trace time dimension from 120 to 170 second. . . . 59

Figure 4.1 The DescendantSize operation in insertion mode with no tree cycle . . 67
Figure 4.2 The position of add() and cut() 69
Figure 4.3 One way the previous reference node can remain an *RN* 72
Figure 4.4 One case of joining two trees 74
Figure 4.5 Execution time for recomputing the *RN* as a graph with an increasing number of updated vertices. The updating sequences contain one million operations, consisting of *Insertion*, *Join*, *Cycle*, and *updateEdge*, in a forest. The previous algorithm measured here has a complexity of $O(n^2)$ 83
Figure 4.6 Dynamic Time *RN* : running time on a random graph with an increasing number of vertices plotted using an algorithm with a complexity of $O(log\ n)$. Updating sequences contained one million operations including *Insertion*, *Join*, *Cycle*, and *updateEdge*, in a forest 84
Figure 4.7 The rate of page faults with the proposed method : running time increases linearly with the number of nodes. 85
Figure 4.8 The memory usage of the proposed method ; the running time increases linearly with the number of nodes. 85
Figure 5.1 Convex-Hull method. 93
Figure 5.2 Fully Incremental Approach 94
Figure 5.3 Fully Incremental Approach 96
Figure 5.4 A general example of a resynchronization area when the MST changes . 100
Figure 5.5 Execution time for recomputing the *MST* as a graph with an increasing number of updated vertices and edges. The updating sequences contain one million operations, consisting of *Insertion*, *Join*, *Cycle*, and *updateEdge*, in a forest. The proposed algorithm measured here has a time complexity of $O(log\ n)$ 103
Figure 5.6 Dynamic Time *RN* : running time on a random graph with an increasing number of nodes plotted using an algorithm with a complexity of $O(log\ n)$. Updating sequences contained one million operations including *Insertion*, *Join*, *Cycle*, and *updateEdge*, in a dynamic network . . 103
Figure 5.7 Map of the computer cluster used in the experiment 105
Figure 5.8 Comparison between the Fully Incremental and Non Incremental methods for pairwise computer time synchronization 111
Figure 5.9 Comparison between the two methods for the complete network time synchronization computation 112
Figure 5.10 Accuracy after 25 minutes for each node statically defined as RN . . . 112

LIST OF SIGNS AND ABBREVIATIONS

ACK	Acknowledge
API	Application Programming Interface
BTS	Branch Trace Store
CPU	Central Processing Unit
CTF	Common Trace Format
DSB	Data Synchronization Barrier
GPS	Global Positioning Satellites
ID	IDentification
IBM	International Business Machines
I/O	Input/Output
IP	Internet Protocol
IRQ	Interrupt Request
KVM	Kernel-based Virtual Machine
LIANA	Live Incremental Asynchronous Network Analysis
Log	Logarithm
LTTng	Linux Trace Toolkit Next Generation
LTTV	Linux Trace Toolkit Viewer
MPI	Message Passing Interface
MST	Minimum Spanning Tree
NTP	Network Time Protocol
NTPD	Network Time Protocol Daemon
OS	Operating System
PIT	Programmable Interrupt Timers
PTP	Precision Time Protocol
QoS	Quality of Service
QEMU	Quick EMUlator
RN	Reference Node
RT	Real-Time
RTT	Round Trip Time
SNTP	Simple Network Time Protocol
ST	Splay Tree
SYNC	Synchronization
TCP	Transmission Control Protocol

TMF	Tracing and Monitoring Framework in the Eclipse framework
TSC	Time Stamp Counter
UDP	User Datagram Protocolor
UTC	Coordinated Universal Time
UST	User-Space Tracer
VM	Virtual Machine

CHAPTER 1

INTRODUCTION

The arrival of multi-core processors in computer clusters represents an evolutionary change in conventional computing to obtain high performance computing. However, these systems may exhibit coherency problems when parallel programs access shared resources, thus creating hard to debug timing related problems. It is therefore crucial to have proper tools to monitor, trace and analyze system execution, in order to identify functional and performance problems. A trace facility aims to keep track of functional flow and report relevant changes at certain times. An efficient and accurate system level tracing is required to monitor and maintain distributed systems.

Over the years, different tools have been implemented to trace operating system behavior by recording kernel events. Some of the most interesting tracing tools are *Ftrace* [5], Dtrace [4], Systemtap [9], and *LTTng* [8]. The currently available trace visualization tools have often targeted detailed traces for small real-time embedded systems, or much less detailed system logs for larger systems. Moreover, existing tracing tools for distributed systems often use coarse higher level events, at the message passing programming interface layer, for which local clock differences may not be a problem ; using a time sychronization service daemon may provide sufficient accuracy in that case, to combine timestamps from several nodes as if their clocks were synchronized.

In newer distributed systems, with shorter and more frequent interactions between nodes, higher accuracy is desirable, especially for measuring and debugging low latency operations. This is the case, for example, for telecom servers, and high performance web sites such as search engines. This explains the high interest for accurate traces synchronization, providing higher accuracy and avoiding the requirement for a time synchronization service in the system under study. Indeed, a major challenge, in monitoring and debugging tools for live systems, is to minimize the impact of tracing on the traced computer.

1.1 LTTng

LTTng, developed at Ecole Polytechnique de Montreal, provides a detailed execution trace of the Linux operating system with low overhead. *LTTng*, like other tracers such as *Perf* [27], *Xtrace* [43], and etc.[28], uses probes to track system events. The probes fetch some information, and write it in event records. An event record contains an event identifier,

a timestamp, and optionally an event specific payload. Probes, when currently enabled, are called when the associated instrumentation is encountered during execution.

LTTng is a prime example of low overhead tracing used for measuring small intervals, for instance system call entry and exit, which may happen within one microsecond. *LTTng* is thus capable of handling huge traces of several gigabytes or more [29]. *LTTng* not only has a very low overhead but it is also able to trace kernel space and user space activities simultaneously. These specific characteristics of *LTTng* help monitoring an ample range of activities in a computer [8].

However, to handle huge detailed traces collected from multiple system nodes and embedded devices, for both online and a posteriori offline analysis and viewing, a new approach is required. Furthermore, while *LTTng* started as a tool for a posteriori analysis, the latest improvements now enable the live tracing and streaming of traces from multiple nodes. In a computer cluster, multiple nodes produce separate trace streams independently. Events in the traces come with a timestamp. Since timestamps are recorded based on a local clock that runs asynchronously on each node, the logical order of events cannot be guaranteed. Global trace analysis, therefore, faces the problem of converting the local timestamp values to a common reference time. Consequently, the aim of this work is to provide a new, efficient and accurate traces synchronization algorithm for live trace viewing and analysis. Indeed, *LTTng* should be able to visualize traces from several distributed systems on a common reference time base.

1.1.1 LTTV and TMF

LTTV, Linux Trace Toolkit Viewer, is the stand-alone viewer for kernel and userspace traces. It is written in C/C++ using *Glib* and *GTK+*. It uses libbabeltrace to read the *LTTng* *CTF* traces. Figure 1.1 shows a screenshot of *LTTV*.

TMF, the Tracing and Monitoring Framework, is an Eclipse plug-in to view *LTTng* kernel and userspace traces. It is part of the Linux Tools project at Eclipse and was used to prototype the new proposed approach. *TMF* provides different types of detailed trace analysis. It offers different views such as the "Control flow view" and "Statistic view", which facilitate trace analysis [7]. Figure 1.2 shows a screenshot of *TMF* where two trace files are shown with a common time base. Meanwhile, the synchronization parameters, relating each local clock to the common time reference, are shown in the synchronization view at the bottom of this Figure. Two traces with the names of scp_dest and scp_src are illustrated and their connection status is shown in row *Quality*. The *Accurate* label for this row indicates that the synchronization was achieved correctly at this moment. Hence, the drift and offset of these two traces are shown in the next two rows (alpha and beta) respectively. In addition, other

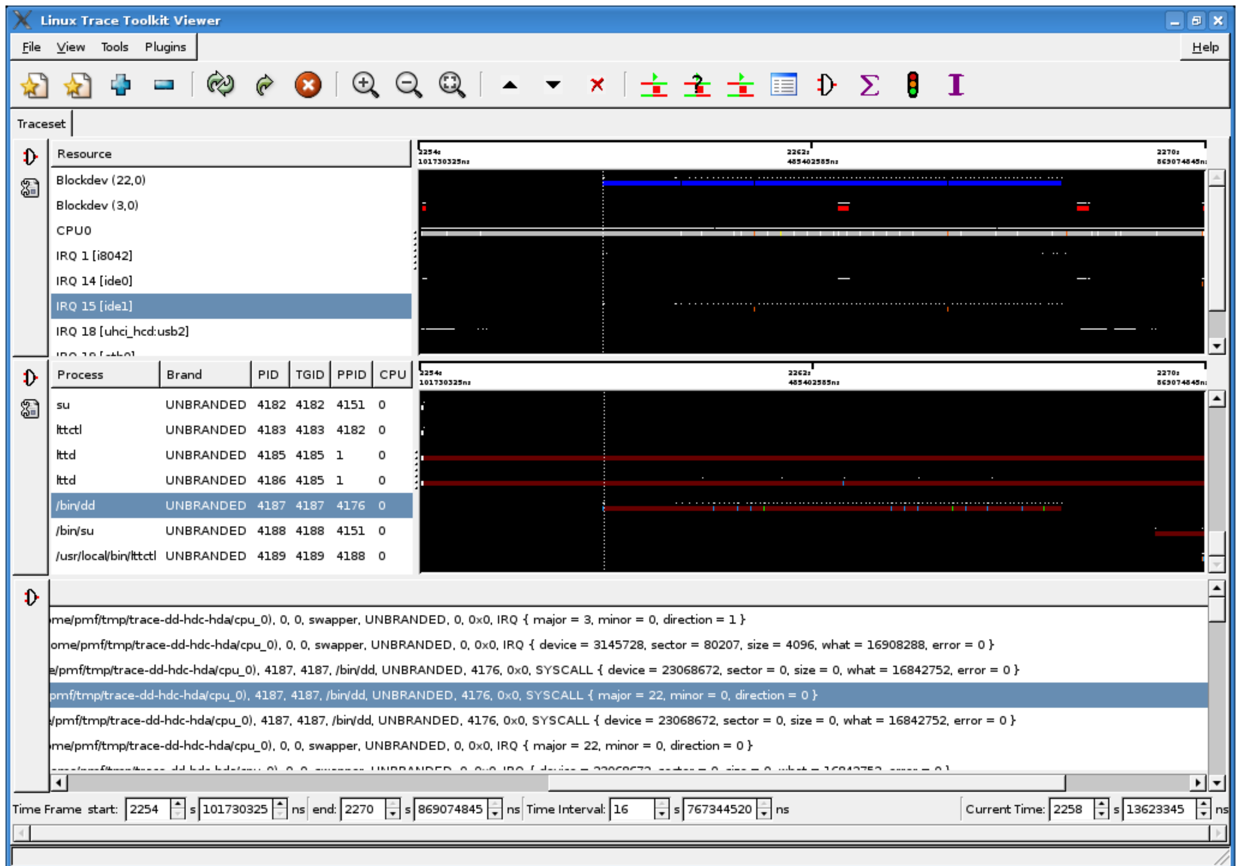


Figure 1.1 Synchronization view of LTTV

synchronization parameters are represented in following rows.

1.1.2 Synchronization Architecture in LTTng

Prior to the work proposed here, the synchronization of two traces in offline mode (tracing is completed and saved in a file on each computer) had been implemented recently in *LTTng* [85]. The main concern was to achieve a high accuracy with this trace analysis enhancement. It consisted in a post-processing step called offline synchronization [85]. This method was applied to traces recorded at the kernel level with low intrusiveness in offline mode. Figure 1.3 illustrates the general architecture of the synchronization model, showing the synchronization steps. There are four connected modules in it. Each module receives input from one or more modules and sends output to other modules.

The input of this architecture is fed by *LTTng* and consists of two or more unsynchronized trace files. The output is two or more synchronized traces. The output format is compatible with *LTTV* and *TMF*, which are able to show synchronized traces. The following modules are present in this architecture :

Processing module : Traces are gathered from all distributed nodes in distributed systems, and are ready to be analyzed. In order to synchronize traces from two nodes, network traffic exchanges between them are required. Packet exchange events are extracted and dispatched to the next module. Thus, this module captures network traffic and computer activity and extracts the necessary information for the matching module.

Matching module : Event processing feeds the events one by one. However, the *Analysis module* works on groups of events. Consequently, the Matching module is responsible for forming these groups. The relations between the packets are of different types (”one to one”, ”one to many”, or a mix) and this will influence the overall behavior of the module for TCP, UDP or MPI. This module must match the sent and receive events for a same packet and group them. For example, for linear regression, the round trip time (RTT) is needed, so this module makes a group after finding an acknowledgment packet, and the acknowledge time will be assumed as reference time.

Analysis module : There are two methods to synchronize time ; *Linear Regression* and *Convex-Hull*. In this module, the user can choose any of these methods to synchronize traces. The *Convex-Hull* method synchronizes traces with better accuracy. Consequently, it is chosen as the default option.

Reduction module : Matched packets are sent to the Analysis module without any processing and are then used to synchronize each node pair based on the *Convex-Hull*. The reference time is computed and all nodes can be synchronized with this reference time. Then, the reference node, the node which has the most accurate links to other nodes, is selected

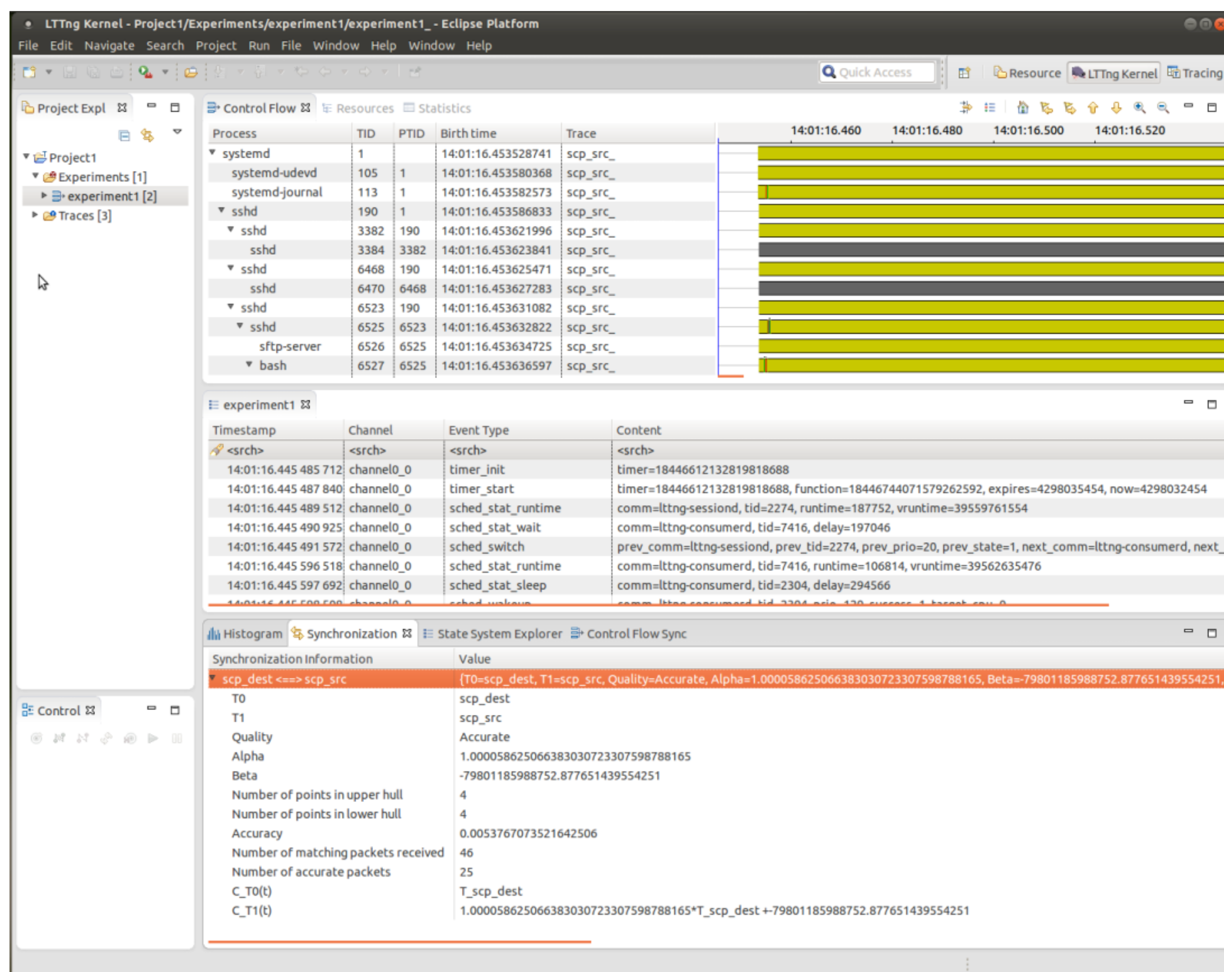


Figure 1.2 Synchronization view of TMF

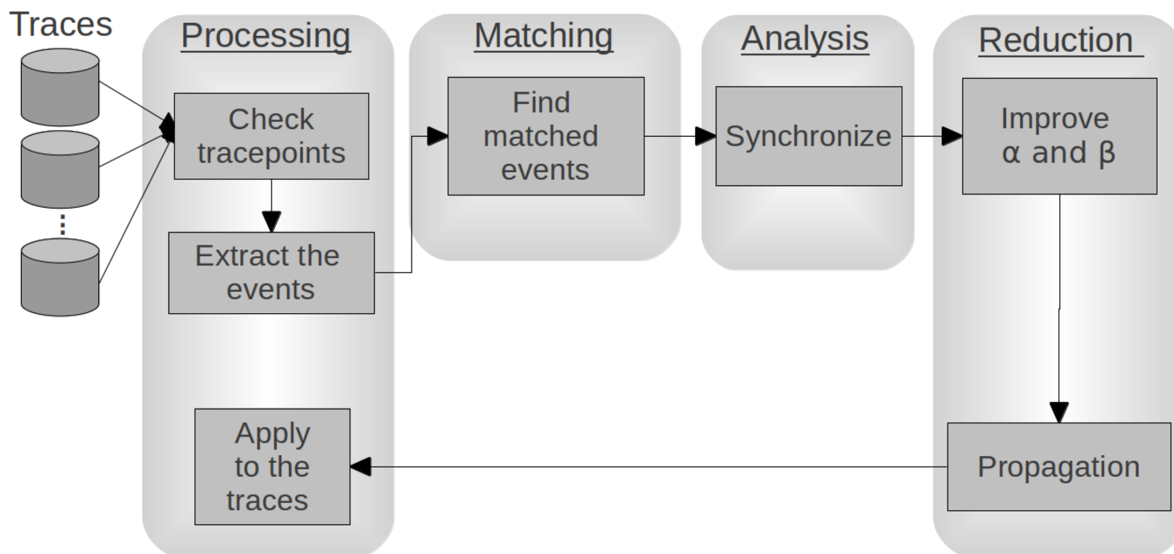


Figure 1.3 Synchronization architecture

and each node pair is synchronized. However, sometimes an indirect path between two nodes has better accuracy (less drift and offset) and is chosen. To find those precise links (and drift/offset) between all node pairs, a Minimum Spanning Tree (MST) ordered by accuracy is computed. It ensures a minimal synchronization tree and the best accuracy in distributed systems. Using a spanning tree has been useful in other similar applications such as wireless sensor networks. The last part in this module is propagation. All nodes should be synchronized based on the reference node time, and drift and offset factors (α and β), propagated through the paths to the reference node. At the end, the resulting times, converted to a common reference node, become available to the user interface and trace events are shown in *LTTV* and *TMF* with right time order.

The work proposed here retains in large part this architecture but transforms each step to efficient incremental algorithms, while maintaining the same accuracy.

1.2 The Contributions of this thesis

Our goal was to achieve an incremental synchronization scheme with high accuracy and low impact. The results of this research were evaluated in the context of a tracing environment, and showed excellent performance. Therefore, the presented approaches can be used for online time synchronization in computer networks even under the most demanding conditions. Our work guarantees the best accuracy, taking into account all the *Convex-Hull* constraints generated by matched packets as they arise, as well as optimal performance and scalability.

First, we proposed a new incremental and efficient technique to synchronize two live

connected systems. As soon as two computers start exchanging messages, this method starts computing the clock offset and drift between the two, based on an optimized *Convex-Hull* algorithm. Since the *Convex-Hull* algorithm relies on the packets with lowest latency, it insures the best time synchronization accuracy. This method updates the synchronization factors when an accurate packet is exchanged and incrementally improves the time synchronization. This method not only does not need any buffering, but also takes $O(1)$ time on average for updating the synchronization, when a new accurate packet is found, which is ideal for live online analysis.

Secondly, we presented a novel incremental method to compute the synchronization parameters at the link level and maintain a Minimum Spanning Tree formed by the most accurate links. In a dynamic network, where computers connect/disconnect to/from the network, we efficiently maintain a dynamic MST. The proposed method is based on splay trees, in which every operation on the tree, such as computer connection, joining separate networks, and so on, takes $O(log\ n)$ where n represents the total number of computers in the network. Therefore, instead of updating the whole MST, the network tree splays on one of updated nodes and the computation is performed upon a portion of the network.

We finally proposed a new method to select and maintain a central reference node in a dynamic network and then update the synchronization parameters. This work is performed for tracing and monitoring purposes, where a time Reference Node is required to synchronize the traces from all the nodes in a dynamic network. In the proposed technique, a novel schema analyzes new vertex insertions, tree merging, and cycle handling in a forest, minimizing average time complexity per operation in the dynamic network. What distinguishes this work from previous work is that it investigates only the altered path with respect to the Reference Node, once an alteration has occurred in the network. The proposed method incrementally processes updates in evolving trees in the forest and thus improves performance.

This new live approach to traces synchronization is fully incremental and most efficient. It not only does not degrade the accuracy of the results, but it also does not delay the synchronization improvement updates. Moreover, it minimizes buffering, an important feature for scaling to large computer clusters and distributed systems. We tested the proposed approach on extremely large clusters, with a real network containing more than 55 physical computers, and in a simulated network containing 60,000 nodes. The results demonstrated that the proposed method addresses the accuracy, performance and scalability needs. Hence, this can be used in all cases where an efficient online time synchronization is desired.

1.3 General organization of the thesis

This dissertation is organized in seven chapters and submitted as a "thesis by articles". This first chapter, introduction, clarifies the context and framework of the research project. It is then followed by the body of this doctoral thesis which consists of four main articles presented in Chapters two to five. The detailed literature review is represented in a survey paper in Chapter 2 with title *"A Comprehensive Survey of Techniques and Challenges in Distributed Systems Time Synchronization"*, submitted to Journal of Network and Computer Applications. This second chapter aims to bring a sufficient understanding of the issues and methods used in this research project [53]. Chapter three presents the second article entitled : *"Streaming Mode Incremental Clock Synchronization"*, submitted to Springer Journal : Network and System Management (JONS). This scientific article introduces an efficient and fully incremental, continuous time synchronization approach for links between two computers. It offers high precision and low intrusiveness for online applications and constitutes the first main original contribution [57]. The fourth chapter contains the scientific article entitled : *"Reference Node Selection in Dynamic Tree"*, submitted to the Journal of Network Management. This work presents an efficient incremental method to select and update the optimum reference node in dynamic networks where nodes connect/disconnect frequently. This is used to efficiently select the time reference node in order to achieve high synchronization accuracy, as required for real-time process-level tracing in a live distributed system. It constitutes the second main original contribution [55]. The fifth chapter presents the last article entitled : *"LIANA : Live Incremental Time Synchronization of Traces for Distributed Systems Analysis"*, submitted to the Journal of Network and Computer Applications. It addresses the complete process of achieving online distributed trace synchronization in real-world live distributed system tracing. It proposes new algorithms to update the time synchronization MST, based on the link level synchronization of Chapter 3, and passes the updated MST to the incremental reference node selection algorithm of Chapter 4 [54]. A general discussion on this research area, and the results obtained, is presented in Chapter 6. This is followed in Chapter 7 by a conclusion and recommendations for future work.

CHAPTER 2

LITERATURE REVIEW : A Comprehensive Survey of Techniques and Challenges in Distributed Systems Time Synchronization

Masoume Jabbarifar and Michel Dagenais

2.1 Abstract

With the appearance of a new generation of distributed systems applications and cloud computing environments, the need arises to revisit the discussion of time synchronization. In such environments, individual physical nodes in large data centers come and go while applications and virtual machines migrate from one physical node to another. A significant problem in this context is to trace and monitor events, with a common reference time, on interacting applications and systems. Yet, tracing and monitoring tools are more important than ever to properly analyze problems in online applications under high load. Thus, time synchronization between interacting nodes is highly desirable. This paper presents a survey and classification of time synchronization protocols according to a variety of factors such as accuracy, scalability, and cost. The provided context helps designers to select the most practical synchronization protocol for their own purposes. The detailed analysis of the characteristics of each approach guides developers to design and characterize new protocols with the desired feature set for a distributed system application. Furthermore, this paper presents a comparison framework through which designers can correlate and analyze the features of new and existing synchronization protocols.

2.2 Introduction

Time synchronization plays an important role for many applications in distributed systems and networks, where many nodes may interact or observe the same events. The information is collected at each individual node in the network, yet it may need to be assembled to build a coherent observation in order to achieve higher-level analysis. One of the key and fundamental ingredients for a coherent observation is a common time reference. For example, when nodes trace the actions related to a problem or a cyber-attack, then higher-level information (such as system performance analysis, attack sources and destinations) can be extracted by correlating data from multiple nodes.

Researchers have created several time synchronization protocols for wired and wireless networks over the past years. Since the challenges posed by wireless networks, such as energy efficiency and movement, are different, time synchronization in wireless sensor networks is considered a separate branch and is outside the scope of this article. On the other hand, wired network applications are growing rapidly with new applications and associated challenges appearing. In particular, online applications deployed in distributed systems require a fast and precise protocol to synchronize streaming data such as execution traces. While many applications insist on either accuracy or cost, applications such as tracing and monitoring need both at the same time. Indeed, tracing tools need to be minimally invasive to not change the system behavior under study, yet high accuracy is required to solve the most difficult problems. This is similar to electronic test and measurement equipment requiring much higher performance than the systems under test.

This paper addresses three main objectives. First, it surveys traditional clock synchronization protocols in distributed systems, and looks at new requirements and solutions for applications such as traces synchronization. This review is based on several factors such as accuracy, scalability, and cost, and will help designers to select the most efficient protocol for their application. Secondly, the analysis of the features, protocols and usecases of the different approaches presented should enable developers to combine or adjust existing approaches for specific application goals. Finally, this paper provides a framework to present different functional aspects and comparisons of the existing synchronization approaches.

Although there are several survey papers on time synchronization, they either predate many of the new applications exposed here [37, 66, 100] or focus on specific usecases such as wireless sensor networks [10, 11, 46, 96, 99]. Wireless sensor networks have several specific concerns of their own, such as energy and movement, and form a somewhat separate branch. In this work, we rather focus on distributed systems which bring different and new usecases, such as telecommunication services, online business applications, online gaming and etc.

This paper is organized as follows. Section 2.3 presents the concepts and foundations of clocks synchronization protocols. Section 2.4 provides a classification of synchronization approaches into online and offline classes. Section 2.5 contains an evaluation of several protocols and a comparison among them based on several factors, which are discussed separately. Section 2.6 lists different challenges that influence the time synchronization accuracy and cost. Finally, we conclude our survey paper with Section 2.7.

2.3 Clock and Synchronization Protocols

The ultimate common reference time usually is the Coordinated Universal Time (*UTC*) [90]. *UTC* is a standard global time source provided by several governmental laboratories around the world and normally based on atomic clocks. It is available as primary time servers on the Internet and as over-the-air electromagnetic signals. Another important precise common time reference is the Global Positioning Satellites (*GPS*) system [71], built around a large number of satellites, each containing an atomic clock. The two time referentials differ by several seconds since *UTC* adds leap seconds every few years to synchronize with the earth rotation, while *GPS* clocks simply follow their atomic clocks with no attempt to remain synchronized with terrestrial days.

While most systems ultimately synchronize with *UTC*, the major concern usually is to have a precise common reference time within a distributed system in order to compare related events arising on different nodes. In that context, the accuracy of the synchronization between the different nodes in the distributed system is often much more important than the accuracy of the synchronization with *UTC*.

The problem arises from the fact that each computer node has an independent clock. In theory, it would be possible to distribute a common clock to several computers in a data center. This could be achieved by a light source feeding several fiber optics cables with carefully measured cable lengths. Indeed, signals in either electrical wires or fiber optics travel at approximately 2/3 of the speed of light or about 6ns per meter. All cables could have the same length or compensation factors could be computed based on the cable length to each node. Then, this signal should be connected directly to the processor internal clock, a facility not available on most systems. Specialized telecommunication equipment may use similar schemes because of their stringent needs on clock synchronization for high speed protocols such as *SONET* [36].

A common mechanism is to have a timing signal cable providing a pulse per second signal at the beginning of each second. This cable is connected to an input signal that can generate interrupts on the node (e.g. a pin on the serial or parallel port of many computers). For example, a very precise pulse per second signal may be provided by relatively inexpensive *GPS* timing receivers. The main source of inaccuracy then often becomes the interrupt latency, for the computer node to be notified of the interrupt generated by the pulse per second signal [22, 33, 74].

When no such special purpose hardware is available, time synchronization is achieved through the existing network hardware. Packet exchanges between nodes can be used to estimate the clock offsets. This will be the focus of the remainder of this article. Different

strategies may be used in order to compare events on a common time reference. The most prevalent, [75], attempts to keep the clock at each node as close as possible to a reference clock, typically *UTC*. Then, events at each node are loosely based on the same time reference. A second approach, used for tracing, is [85] where the clock at each node does not need to be especially well synchronized. Instead, the a posteriori analysis of packet send and receive events in traces is used to deduce with high accuracy the offsets between the clocks in each node. Finally, a third approach is used when tracing information is displayed live, in quasi real time, [57]. The approaches are similar to a posteriori analysis except that the computation must be performed incrementally and that the offset estimation may improve over time as more packet send and receive events are analyzed.

2.3.1 Time Keeping Hardware

When a computer node is started, it needs to initialize its internal clock. This is usually achieved by reading the current time from a special battery-backed circuit called Real Time Clock. This circuit is typically not used thereafter during normal operation because of the delay to read from this slow peripheral. Instead, the operating system uses a regular interrupt to update its internal clock. Most processors contain at least one and often several Programmable Interrupt Timers (PIT) that can be used for that purpose [21]. Unix and early Linux systems received interrupts to update their internal clock every 10ms. This was later changed to each 1 ms.

When a finer granularity is required, a cycle counter can often be used, such as the Time Stamp Counter (*TSC*) on Intel processors. The *TSC* is a special hardware register incremented at each clock cycle. It thus provides a high-resolution, low-overhead, and fine-grained (about 1 nanosecond or better) time source [31]. However, if the processor frequency changes, the *TSC* rate will change as well on most systems. When the processor goes into idle or halted mode, the *TSC* may stop altogether. With the advent of multiple processor cores on a single chip or motherboard, the *TSC*s on different processor cores may drift away from each other over time, and there is no guarantee that they will ever resynchronize.

A new generation of multicore processors from Intel include a constant rate *TSC*, which provides for synchronization of the cores even though their frequency may vary over time [81]. Indeed, a constantly running clock, shared by many processor cores, is used to update these *TSCs* [64]. The *TSC* can then be used as the only time source for kernel and user space needs.

The interaction between these multiple clock sources can be problematic. Linux systems have a system clock tracking *UTC*, typically through the Network Time Protocol. However, the time adjustments can distort time measurements needed by the operating system,

for instance in device drivers. While it is not recommended, the system clock may even be adjusted to go back in time. For this reason, another clock source is defined in the kernel, CLOCK_MONOTONIC, which is never adjusted during the execution and progresses regardless of *UTC*. It represents the absolute elapsed time since an arbitrary point. It keeps increasing and is synchronized between all processor cores in a node [26]. It may run slightly faster or slower than real time, its rate may vary slightly due to environmental conditions like temperature or voltage, and it may jump ahead when in a virtual machine and coming back from being scheduled out. This clock source is typically built from the regular timer interrupts at every milisecond, with the TSC being used on Intel processors to interpolate between two timer interrupts and provide the needed high resolution. The time update at each timer interrupt is synchronized among the multiple processor cores, enabling the CLOCK_MONOTONIC to be fairly well synchronized between cores. CLOCK_MONOTONIC is used in the LTTng tracer [8], insuring a high resolution common reference time among the processor cores on a single node [20, 62, 84].

2.3.2 Packet-based Clock Offset Calculation

When a packet is sent from time server node S to client node C, C's clock could be set to S's clock plus the delay for the packet to travel between the two nodes. The problem with this simplistic approach is that the nominal network delay may not be known. Moreover, network congestion or operating system latency may delay almost indefinitely the delivery of the packet to the clock synchronization application.

A better approach was proposed by Cristian [25]. This method is based on the round-trip time (RTT) between a computer C and a time server S. C sends a time request to S. Once S receives the request, it responds by appending its current time T_S to the message. Then, C calculates the time with Formula 2.1 and updates its clock accordingly. This technique assumes that the network delay is the same for the request and the response. When this is the case, perfect accuracy is obtained. Otherwise, the computed time value is within $\pm RTT/2$ of the real value. To improve the accuracy, C can send multiple requests to S and retain the response with the smallest round-trip time (RTT).

$$T_C = T_S + (RTT/2) \tag{2.1}$$

This was further improved upon by incorporating into the equation the processing time of the server. Unlike the network delay, this processing time, albeit small, is easily obtained. Figure 2.1 illustrates that the sender issues to the receiver a message at *T1*. The receiver notes *T2* as the reception time. Then, the receiver returns an ACK message to the sender

at time *T3*. The sender receives the *ACK* message with *T4* as reception time. Finally, at the end of the message exchange, the sender can compute the offset and accuracy from the four timestamps embedded in the Response message with Formula 2.2. The sender adjusts its clock according to the offset and the synchronization is performed.

Here again, the method assumes that the sender-receiver and receiver-sender propagation times are exactly equal, in which case perfect accuracy is obtained. Otherwise, the bound on accuracy is provided. Since the inaccuracy is related to the time elapsed between T1 and *T2*, and between *T3* and *T4*, one optimization is to get these time values as close as possible to the packet send and receive points in the kernel. For example, in the *LTTng* tracer, which uses this technique to synchronize traces, the packet send and receive events in the trace are generated in the kernel at the lowest possible point in the network stack, just ahead of the interface with the *NIC* driver [85]. The values of timestamps are thus obtained much closer to the packet send and receive points. This is used in order to obtain a better accuracy than what can be achieved when generating timestamps at the application level in a time synchronization daemon. Another interesting aspect of *LTTng* is that the information from existing packets is used to compute the offset, keeping the tracer minimally invasive since it is not necessary to send additional packets for time synchronization.

This synchronization method, based on Equation 2.2, is the basis for the Network Time Protocol (*NTP*), a standard Internet protocol for clock synchronization. It proposes an organisation with two levels of time servers : Primary and Secondary time servers. A primary time server synchronizes directly with a reference time source, usually a *UTC* atomic clock. Secondary time servers synchronize with primary time servers or other secondary servers. A client typically synchronizes with its nearest secondary time server. *NTP*, depending on the network latencies, typically achieves an accuracy between one and ten milliseconds in local networks and tens or even hundreds of milliseconds in wide area Internet. The Simple

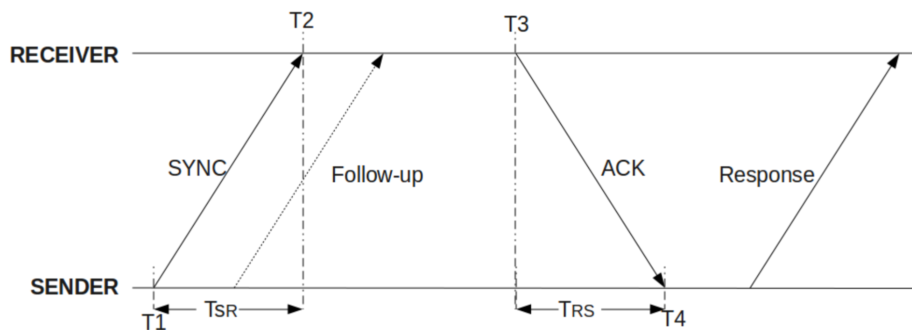


Figure 2.1 SYNC message

Network Time Protocol ($SNTP$) is a simplified version of NTP for users [80].

$$
\begin{aligned}
Offset &= [(T2 - T1) + (T3 - T4)]/2 \\
Accuracy &= \pm[(T2 - T1) - (T3 - T4)]/2
\end{aligned}
\tag{2.2}
$$

The Precision Time Protocol (PTP) [38] uses the same equation to estimate the network delay between each node and the time server. However, hardware support in the networking equipment can be used to insert the reference time at the network switch level in a broadcast packet, insuring that all clients receive the reference time in parallel and with a very short network delay. Furthermore, networking equipment is optimized to minimize network delay variations and asymmetry. As a result, PTP can achieve clock synchronization accuracy at the microsecond level.

When no time server is available, several computer nodes can communicate their clock values and compute an hopefully more precise average value as the reference. This is the main idea behing the Berkeley protocol [24, 66, 100].

2.3.3 Logical clock Synchronization

Some applications only require causal ordering of events. Hence, they use logical clocks to order events. Each pair of related events is ordered by causality relations (such as the send event for a packet necessarily happening before the receive event). This type of synchronization is called logical time. Lamport proposed algorithms to compose logical clocks. The limitation of his algorithm is that it cannot necessarily specify which one is executed first when $timestamp_{e_1} < timestamp_{e_2}$, unless they refer to the same logical clock [65, 69]. Mattern and Fidge proposed a method based on Vector Clocks to address this problem and determine precedence. Landes et al. use a tree structure to improve Vector Clocks limitation. However, the storage and message size increase with the number of nodes [70]. Logical clocks suffer from two important limitations. First, they cannot provide precedence relationships for events without explicit causality relationships, for example two computer nodes interacting with the same shared storage device (thus indirectly but not directly interacting). Secondly, there is no notion of absolute time, which is important in several applications.

2.4 Synchronization techniques to compute clock offset and drift

As seen in the previous section, network packets sent among nodes are used to compute the clock offset between communicating nodes. Packets may be sent explicitly for time synchronization, carrying the send and receive time values for the exchange, or a trace of ordinary

packets sent and received at each node can be taken and sent later to an analysis node. In either case, it is interesting to combine the information from several packet exchanges to get a more precise estimate of the offset, and its variation over time. The exact frequency of the clock at a node can vary somewhat. Several computers with a 2 GHz nominal frequency will in practice all have a slightly different frequency. That frequency, for a given node, is however extremely stable over a time as long as factors such as the operating temperature or the supply voltage do not vary significantly [75]. For this reason, the clock offset between two nodes varies linearly with time as a factor of the ratio of their respective frequency. It will therefore be represented as a linear function with the initial offset and drift as coefficients. The two main methods to estimate this linear function, based on the data from several packet exchanges, are the Linear Regression and the *Convex-Hull*.

Linear Regression

The Linear Regression models a response variable, y, as a linear function of a single predictor variable, x. The formula for this definition is : $y = a \times x + b$, where *"a"* and *"b"* are the regression coefficients. Coefficient *"a"* is the slope of the line and *"b"* defines the Y-intercept. In our case, *"a"* is the clock drift, *"x"* is the time and *"b"* is the initial offset.

In a two dimensional space, based on timestamps of node A and B, the Linear Regression algorithm tries to fit a line among the points. Packet A sent from node A has a timestamp that provides the x coordinate. As soon as it is received in node B, its timestamp is obtained and becomes the y coordinate. These two coordinates define a point in the two dimensional space. Since there are many packet exchanges in a regular connection, many points will be obtained in this way [95]. It is obvious that a line can be drawn through two points. However, there are typically many more than two points and just one line should model the whole information. The solution is based on the method of least squares. This model estimates the best-fitting line that minimizes the differences with actual data. If each point in the two dimensional space is represented by its x and y values, and there are N points (total number of exchanged packets), the average of x and y values are obtained as shown in Formula 2.3.

$$\begin{aligned} \overline{x} &= \frac{\Sigma_{i=1}^{n} x_i}{n} \\ \overline{y} &= \frac{\Sigma_{i=1}^{n} y_i}{n} \end{aligned} \tag{2.3}$$

Both regression coefficients are estimated with Formulas 2.4 and 2.5.

$$a = \frac{\Sigma_{i=1}^{n}(x_i - \overline{x}) \times (y_i - \overline{y})}{\Sigma_{i=1}^{n}(x_i - \overline{x})^2} \tag{2.4}$$

$$b = \overline{y} - a \times \overline{x} \tag{2.5}$$

In these equations, *"a"* and *"b"* are the drift and offset respectively between the two clocks. While the linear regression gives adequate results, there are significant weaknesses with this approach. The first is that the measurement error is biased. When everything works as expected, a minimal delay is achieved and a very good point is obtained. The system and network delay cannot go under this minimum, corresponding to the ideal case. However, if the system is preempted by a high priority interrupt or there is network congestion, a much longer delay may be obtained creating a bad point. The problem is that the linear regression takes into account all points when calculating the best fit. To alleviate this problem, some have proposed to detect and eliminate outliers, not using them for the linear regression [22] and obtaining more precise results. A second problem is that the x and y values, the time at nodes A and B, are not sampled simultaneously ; they are separated by the packet propagation time. However, if the number of packets, used for the linear regression, is the same in each direction, the errors in each direction should mostly compensate for one another.

Convex-Hull

The *Convex-Hull* algorithm is based on the fact that each packet implies that the receiving time is later than the sending time. Thus, it does not suffer from outliers since they bring weaker constraints which have no effect on the result. Similarly, the fact that there is a network delay between the time at which x and y are measured does not contradict the inequality indicating that the receiving time (even with the network delay added) is after the sending time.

As shown in Figure 2.2, pairs are divided into two sets, based on the message direction. Consequently, the synchronization estimation line should be below all the pairs $\{\overrightarrow{(\theta_i^S, \xi_i^R)}$, $\overrightarrow{(\theta_{i+1}^S, \xi_{i+1}^R)}$, ...$\}$ and above all pairs $\{\overleftarrow{(\theta_j^R, \xi_j^S)}$, $\overleftarrow{(\theta_{j+}^R, \xi_{j+1}^S)}$, ...$\}$.

The packets with minimum latency are those of interest in the *Convex-Hull* synchronization algorithm. Packets sent from θ (horizontal axis) to ξ (vertical axis) occupy the upper left half-plane and are shifted higher when more network latency was encountered. Therefore, the lower half-hull, of the *Convex-Hull* formed by those points, is a lower bound for the packets sent from θ and identifies the packets with the lowest latency. Similarly, packets sent from ξ (vertical axis) to θ (horizontal axis) occupy the lower right half-plane and are shifted to the right when more network latency was encountered. Therefore, the upper half-hull, of the *Convex-Hull* formed by those points, is an upper bound for the packets sent from ξ and identifies the packets with the lowest latency. The possible synchronization lines lie below the

lower half-hull of packets sent from θ and above the upper half-hull of packets sent from ξ.

The synchronization accuracy is limited by the delay between the send and receive events. Any packet delayed by interrupts, network switch delay, or some other cause will lead to an inaccurate pair and a long delay. The *Convex-Hull* algorithm selects the pairs on the inside envelope between the send time and the receive time, and so identifies the most accurate pairs with the shortest delay. In other words, it finds the area that has minimum latency and ignores outlying pairs, as shown in Figure 2.2 ($(\overrightarrow{(\theta_2^S, \xi_2^R)}, \overrightarrow{(\theta_5^S, \xi_5^R)}, \overleftarrow{(\theta_2^R, \xi_2^S)}, \overleftarrow{(\theta_3^R, \xi_3^S)})$). Therefore, the estimated line is more accurate than with Linear Regression, not being affected by outliers.

According to the above definition, we have two completely separate sets. Otherwise, this would imply a message inversion (receive before send). In each set, the optimal separator is computed (the solid line in each hull) from the points in each set nearest to the separator space. Graham's scan algorithm selects the points forming the *Convex-hull* in these two separated sets. The bounds of the *Convex-hull*, shown in Figure 2.2, are :

$$\begin{aligned} UpperBound &= \{\overleftarrow{(\theta_1^R, \xi_1^S)}, \overleftarrow{(\theta_4^R, \xi_4^S)}, \overleftarrow{(\theta_5^R, \xi_5^S)}\} \\ LowerBound &= \{\overrightarrow{(\theta_1^S, \xi_1^R)}, \overrightarrow{(\theta_3^S, \xi_3^R)}, \overrightarrow{(\theta_4^S, \xi_4^R)}, \overrightarrow{(\theta_6^S, \xi_6^R)}\} \end{aligned} \tag{2.6}$$

Thus, the maximum likelihood estimators are between the following conditions :

$$\begin{aligned} &\alpha\theta_i^S + \beta < \xi_i^R \\ &\alpha\xi_j^R + \beta < \theta_j^S \\ &i, j = 1, 2, ..., n \end{aligned} \tag{2.7}$$

The next step is to find two lines, one with maximum slope (L_{max}) and another with minimum slope (L_{min}) :

$$\begin{aligned} L_{max} &= \alpha_{max}\theta + \beta_{min} \\ L_{min} &= \alpha_{min}\theta + \beta_{max} \end{aligned} \tag{2.8}$$

As a result, the final estimated α and β are certainly limited to the area that is enclosed between $(\alpha_{min}, \alpha_{max})$ and $(\beta_{min}, \beta_{max})$, and the selected synchronization line is the bisector of L_{max} and L_{min}.

When synchronizing two traces, the relationship between them can be one of the following :

Definition 1) An accurate relationship : this is the expected case, shown in Figure 2.2, where L_{max} and L_{min} are available and their middle can be computed. If the relationship between two clocks is of the accurate type, we can define the accuracy metric as the difference between the minimum and maximum possible drifts between the two clocks.

$$Accuracy(i) = L_{max}.drift - L_{min}.drift; \tag{2.9}$$

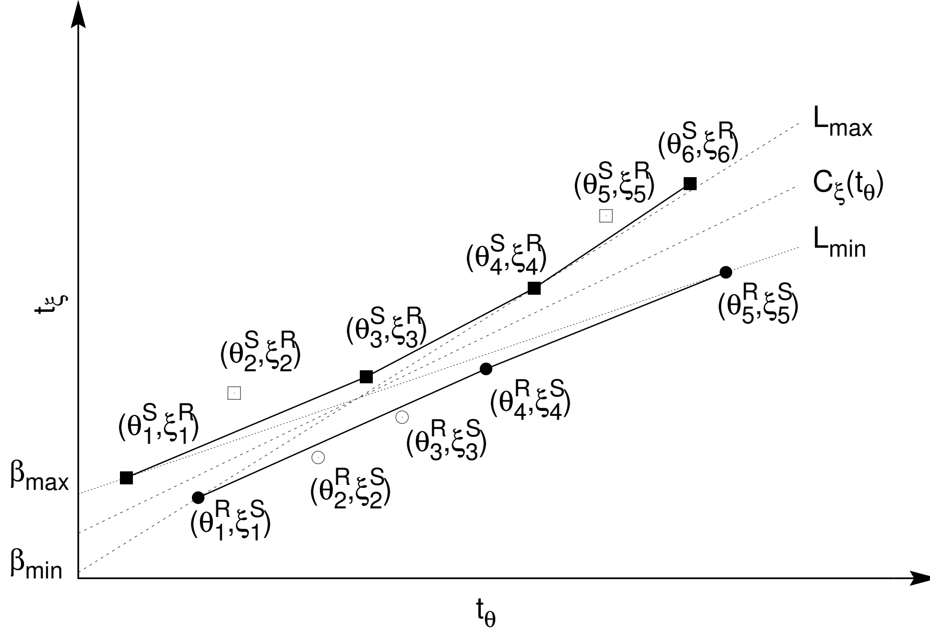


Figure 2.2 Convex-hull method.

Definition 2) An approximate relationship : L_{max} and L_{min} are not available because the hulls do not satisfy the hypothesis that the upper half-hull should be below the lower half-hull and not intersect with it. This may be caused by a deviation from the assumed constant clock frequency, causing a higher-order (e.g. quadratic) relation between the two clocks. Other possible causes include a problem with the time measurement computation. In that case, the approximation is a "best effort".

Definition 3) An incomplete relationship : only one of the L_{max} and L_{min} lines is available. There is communication in only one direction, which is insufficient to obtain a proper bounded synchronization.

Definition 4) An absent relationship : there is no communication between the nodes in either direction, and nothing can be deduced about their relative time.

Definition 5) Fail relationship : none of the L_{max} and L_{min} lines is available. This is because the hulls completely intersect each other or are reversed.

Based on each of these definitions, two nodes either are synchronized, not synchronized, or partially synchronized. When there is no connection between two nodes, it may be possible to compute their offset and dritf indirectly through other nodes with which both are communicating.

2.5 Synchronization Applications

In traditional clock synchronization, the aim is to adjust the clock offset immediately with respect to a time server [73, 77, 78]. Information from successive synchronization points is used to compute the clock drift, better correlate these synchronization points over a long period of time, filtering out less accurate values, and improving the accuracy between synchronization points using a correction factor. The other form of time synchronization is traces synchronization [34, 83, 86]. In that context, traces from several distributed nodes are brought together on an analysis node. The basic algorithms for synchronization remain the same but several constraints are different and the algorithms can be adapted and optimized accordingly. For instance, it can select the best path to compute the clock differences between two nodes among several indirect paths. Moreover, the most interesting node to use as time reference can be selected dynamically. Finally, in some cases, the trace analysis is only performed *offline* at the end of data collection. In that case, the clock differences can be optimally computed based on the complete data set.

In this section, we will therefore examine the specificities of traces synchronization and in particular offline *A posteriori Trace Synchronization* and live streaming *Online Trace Synchronization*.

2.5.1 Offline Clock Synchronization

Recently, many time synchronization algorithms have been suggested. The main goal of these algorithms is to increase the time synchronization accuracy. All algorithms tried to estimate a function that models the time on the clock of a computer versus the time on the clock of another computer, and then propagate the estimation to other computers in a cluster.

When a message is exchanged between a pair of nodes, the receiving and sending times will not be directly comparable because the clocks of two nodes are not synchronized [12, 64, 72]. However, by the principle of causality, the receive time must be later that the send time. This constraint is used to compute the clock drift between two nodes [25].

An interesting offline clock synchronization method has been proposed by Duda et al. [37], which consists of two synchronization algorithms, *Linear Regression* and *Convex-Hull*. These algorithms estimate a conversion function between the clocks in a pair of nodes. In both algorithms, the conversion function is linear, and the drift and offset between the two clocks are extracted from this linear model [58].

It is also used to estimate the one-way delay between two nodes by Moon et al. [82]. In a two-dimensional space, based on the time values at nodes A and B, the *Linear Regression*

algorithm attempts to map all points to a line. Thus, every point will affect the position of that line. In reality, network latency and similar events between two nodes can cause problematic outlying points, biased being only late. These outliers should ideally not affect the drift or offset computed from the *Linear Regression*. They should, in fact, be ignored in order to increase accuracy [14].

The *Convex-Hull* algorithm does not suffer from this problem and insures the highest synchronization accuracy [59]. The *Convex-Hull* is a precise algorithm that assumes upper and lower bounds (sending time and receiving time) separated by the network delay. In this way, it finds the area that has minimum latency and ignores outlying points. As a result, the estimated line is more accurate with this algorithm than with *Linear Regression*.

Among these two algorithms, *Linear Regression* and *Convex-Hull*, the *Linear Regression* algorithm can use existing functions from a statistical package and is thus easier to implement. However, the *Convex-Hull* algorithm can model clocks with higher accuracy while still requiring a modest computational complexity. The following two subsections provide more details about using each of these two algorithms.

Khlifi et al. [63] proposed two algorithms, which they call *the average* and *direct skew removal* techniques for offline skew removal. The *average* algorithm calculates the average delay for a fixed number of consecutive packets at the beginning and the end of a trace, yielding a constant *O(1)* complexity. The *direct skew removal* technique has the interesting property of being able to account for low clock resolution, where the clock granularity may be larger (e.g. 1ms) than the packet delay. For this purpose, the whole trace is analyzed for a linear *O(n)* complexity.

Clement et al. [22] have evaluated the impact of system characteristics on trace synchronization accuracy. First, they studied the tracing duration impact. They propose dividing long duration traces into 30 minute segments, since the error in the clock drift linear approximation begins to increase significantly after approximately 45 minutes of tracing, while it is quite stable during the first 30 minutes. The error increases because of the variation in the clock drift with time, as shown by the *Allan deviation* [2], and because of environmental effects on the clock circuit frequency, such as temperature and supply voltage variations.

Then, they studied the impact of the system load parameter, when there is a heavy load on major subsystems, CPU, memory, and hard disk. They found that the transmission time and response time measurement variations, caused by interrupt latency in a loaded system, influence the clock drift computation directly, and subsequently the time synchronization accuracy. The third parameter studied was hop count, when there is more than one path between two nodes. In that case, the offset between the two indirectly linked nodes may be computed by adding the offsets along a path, from one intermediate hop to the next. A path

with fewer hops generally provides higher accuracy. If there is a direct path, it is normally better to choose that one to synchronize two nodes.

Poirier et al. [85] presented an accurate method for synchronizing distributed traces. This method is applied to traces recorded at the kernel level with low intrusiveness. They applied the *Convex-Hull* algorithm to the clocks of traced nodes as a conversion function. If collected traces are huge and involve numerous nodes, their method is time consuming. Since their algorithm was designed for post-processing, the analysis delay was not major concern. However, for a live display of traces, the latency should be minimized. In [56], the proposed method estimates accurate paths in large computer clusters and improves the performance of offline distributed trace synchronization.

2.5.2 Online Clock Synchronization

Online synchronization works in streaming mode. Several researchers have proposed algorithms for this application. The standard clock synchronization method, widely used today, are the Network Time Protocol (NTP) [79] and NTP Daemon (NTPD) [80]. It sets and maintains the kernel system clock, used to measure packet send and receive time, based on feedback from exchanges with the server. Veitch et al. and Ridoux et al. [87, 102] proposed the RAD clock (Robust Absolute and Difference Clock), which provides alternative clock synchronization algorithms. The timing packets are timestamped using raw packet timestamps. They estimate the clock skew based on the difference between the system clock and the timestamps received from the server, and maintain the clock skew correction without changing the raw system clock, in a feed-forward approach.

Khlifi et al. [63] presented two techniques for online skew estimation and removal. The first one, *sliding window*, monitors the minimum delays to reduce the gap between the true and the corrected delays, (the correction being the estimated skew). To improve the accuracy, they present a second technique, the combined approach. They perform the sliding window algorithm to quickly estimate the skew, in the first interval. Then, they use the *Convex-Hull* algorithm during subsequent intervals to improve the accuracy.

A particularly efficient algorithm is proposed in [57], which is based on *Converx-Hull* algorithm combined with lines with minimum and maximum possible slopes between the hulls. This can provide skew estimates very early, obviating the need for a different method in the first interval. Furthermore, the proposed algorithm can identify accurate packets (those few that can improve the estimate) with a simple test and recomputes the drift and offset incrementally in $O(1)$ upon identifying an accurate packet.

2.6 Evaluation of protocols

In this Section, the evaluation criteria are detailed. Then, the various synchronization protocols are compared and evaluated based on the comparison criteria. The presented classification helps selecting the most efficient synchronization protocol in terms of performance and applicability.

2.6.1 Evaluation factors

The evaluation factors are listed in this section. Moreover, the influence of each factor on the synchronization is explained.

Synchronization Accuracy

A hardware oscillator circuits provide the physical clock in a system. Since the frequency of hardware oscillators varies, clocks operate at slightly different rates on different systems. Therefore, the physical clock values should be synchronized when an application needs to accurately compare the time of events on different nodes. As discussed earlier, many other applications are satisfied with a causal ordering of related events.

Synchronization accuracy is a factor that shows the time difference at a node. The accuracy measurement defines how well the synchronization is performed. Accuracy shows the deviation of the synchronized time at a node from an external reference node on the network, or from the time of another node. When the deviation between the clocks at two nodes is smaller, we have better accuracy. When synchronizing a pair of nodes, the accuracy is the deviation between the two, and when synchronizing a network, the accuracy is the maximum deviation over the network.

1. Absolute synchronization accuracy : The maximum deviation of the logical/physical clock of the node from an external standard, for instance UTC.

2. Implicit synchronization accuracy : The maximum deviation among all pairs of logical/physical clocks of two connected nodes in a network.

Computational Complexity

In a large computer cluster with numerous nodes, the computational complexity in both time and memory is an important concern to select an efficient synchronization protocol for a specific application. It is different from the message complexity, discussed in the next subsection. Beside the computation time, the buffering requirements is another significant factor to evaluate the behavior of the protocols. A protocol may be impractical if its memory requirements relative to the number of computers being synchronized are disproportionate.

Convergence time

The total synchronization time for a network is named convergence time. Usually, protocols with a large number of messages exchanged for synchronization have longer convergence time, as compared to protocols with no *Sync* messages. The protocols with no *Sync* messages use the timing information from regular messages. Hence, there is no need to wait for the next round of synchronization messages to get synchronization data. Beside the number of messages, the network bandwidth directly affects the convergence time. A protocol with minimum convergence time is preferred for many applications.

Overall cost

Algorithmic complexity and communication overhead are combined into a metric called *overall complexity*. Overall complexity is shown as a numerical value even though it is qualitative metric. However, the *computational complexity* and *convergence time* are two quantitative factors.

Fault tolerance

A real-world network is influenced by many disturbances caused by either the environment or human intervention. A packet loss impacts time synchronization where synchronization protocols use explicit message exchanges. Packet loss directly affects the overhead in terms of both network traffic and synchronization accuracy. Furthermore, it reduces the performance significantly.

Scalability

Scalability plays an important role in current distributed systems. In many cases, the protocol is impractical when the network contains a large number of nodes. As large computer clouds are employed for *big data* computations, limitations on time synchronization scalability become important. A time synchronization protocol is expected to work efficiently in a huge cluster. Thus, the scalability becomes an important feature of a protocol.

Real-time application

There are two types of applications in which the protocols provide synchronized data for a distributed system. The first is when the applications need synchronized information for further a posteriori analysis. Other applications require online analysis and thus *Real-time* online synchronization. In such cases, a key factor is to minimize the synchronization

computational complexity and latency at each incremental update.

2.6.2 Protocols comparison

Table 2.1 presents a comparison among synchronization protocols based on the factors introduced. Most protocols are highly accurate. The desired level of accuracy varies from one application to another. The most accurate protocols are based on the *Convex-Hull*, and in particular those presented by Poirier et al. [85] and Jabbarifar et al. [54]. The protocol obtains a synchronization accuracy among connected nodes in the network of around 10 μs, as compared to *ms* accuracy in other protocols. As discussed earlier, however, hardware assisted synchronization can obtain better accuracy.

All [63, 68, 85] protocols takes $O(n)$ for synchronizing two nodes, n being the number of packets used for the computation. Since these protocols do not have an optimized mode for whole network synchronization, they require $O(n^2)$ for network synchronization where n is the number of nodes. Therefore, they are all problematic to use in huge computer clusters. However, the protocol in [57] provides a streaming and incremental approach to analyze exchanged packets where the drift and offset are updated when new packets are received with $O(1)$ time complexity. Jabbarifar et al. presented a procedure for whole network synchronization in [54], which takes $O(n \ log \ n)$ for a network with n nodes. Consequently, it is the only method optimized for very large clusters, and the best choice to synchronize a huge network with adequate computational complexity.

The protocol proposed by Khlifi et al. [63] uses specific messages for synchronization. Therefore, this adds to network traffic and also typically provides fewer and more distant synchronization points, incurring a higher convergence time. On the other hand, Kuhn et al. [68], Poirier et al. [85], Jabbarifar et al. [54], and Scheuermann et al. [89] use messages from normal network traffic to synchronize the nodes, avoiding the need for additional synchronization messages. Hence, they normally provide a better convergence time.

As discussed earlier, the protocols by Kuhn et al. [68] and Poirier et al. [85] have higher computational, storage and message costs as compared to Khlifi et al. [63]. The best choice for a resource-restricted application is probably the protocol proposed in [63]. However, it is not very accurate and thus not suitable in many applications. The lowest synchronization cost belongs to the protocol proposed by Jabbarifar et al. [54]. In addition, it has minimal buffering requirements. It only stores a very limited number of packets, even for a very long trace, typically less than 10, those likely to be on the final *Convex-Hull*.

The protocols presented in [54, 63, 85] do not suffer from message losses, as long as there is a sufficient number of successful packets. Other protocols have not addressed the issue of their sensitivity to message losses. For protocols requiring explicit time synchronization

messages, lost messages require retransmission and care to prevent incorrect matches between duplicate (retransmitted) send or receive messages.

Based on the experimental results presented for each of the protocols, the protocol presented in [89] scales to large networks. However, it achieves synchronization accuracy in *ms*, which is not considered very accurate. The other protocol tested with large networks (more than 20 physical nodes in one experiment, and with 60000 simulated nodes in another experiment) is the one presented in [54]. All other protocols have been tested in small networks and, while they may offer good accuracy, their scalability is a major concern.

As shown in the last column of Table 2.1, only the protocols in [54, 63, 68] support online synchronization, which is important for online monitoring. Other protocols collect data first and then apply synchronization algorithms. Using repeatedly an offline algorithm to achieve online synchronization is a costly proposition, or may induce latency in obtaining results when used in small duration batches.

2.7 Conclusion

Some distributed systems only require maintaining a logical order among events and are satisfied with logical clocks. There is however a large proportion of applications, especially for debugging and monitoring purpose where accurate timestamps, on a common time referential, is required. In some cases, a hardware solution can provide the needed synchronization with minimal computation cost and very high accuracy. In the general case, such hardware is not available and efficient time synchronization algorithms must be used. The needs in such applications vary greatly in terms of accuracy, network size and online versus offline.

This paper surveyed and evaluated existing clock synchronization protocols according to a number of important parameters such as accuracy, scalability, cost and etc. Each parameter

Table 2.1 Performance comparison of synchronization protocols

Protocol	Synchronization Accuracy	Complexity of Calculation	Convergence Time	Overall Cost	Fault Tolerance	Scalability	Real-time Application
Khlifi et al. [63]	50 ms	$O(n^2)$	High	Medium	Yes	Poor	Online/Offline
Kuhn et al. [68]	Unknown	$O(n^2)$	Low	High	No	Poor	Online
Poirier et al. [85]	10 μs	$O(n^2)$	Low	High	Yes	Poor	Offline
Scheuermann et al. [89]	0.1 ms	N/A	Low	N/A	Unknown	Good	Offline
Salyers et al. [88]	17.2 ms	N/A	Low	N/A	Yes	Poor	Offline
Jabbarifar et al. [54]	10 μs	$O(log\ n)$	Low	Low	Yes	Excellent	Online/Offline

was explained as well as its impact on distributed systems. The article thus provides a comprehensive and detailed review of the existing protocols and helps developers to analyze and compare the different approaches. Finally, the analytical structure of the survey facilitates the selection by developers of the most suitable and efficient protocol, or a combination or extension to these protocols, to satisfy their specific application.

CHAPTER 3

Paper 1 : Streaming Mode Incremental Clock Synchronization

Masoume Jabbarifar, Michel Dagenais, and Alireza Shameli-Sendi

3.1 Abstract

It is crucial to have appropriate tools to monitor, trace, and analyze system execution, so that functional and performance problems in distributed systems can be identified. A trace facility is aimed at keeping track of functional flow and reporting relevant changes at certain times. In distributed mode, each node produces individual trace streams independently. Times are recorded by a local clock which runs natively on each node. Traces from all the computers in a network are gathered and analyzed online. One of the most significant requirements for analyzing traces is online synchronization accuracy. The aim of this paper is to demonstrate an efficient implementation of time synchronization in streaming mode. We propose several approaches based on sliding window and non-sliding window techniques to resynchronize the traces at regular intervals. We compare these approaches, and introduce a fully incremental, continuous synchronization approach.

3.2 Introduction

The advent of multicore processors in computer clusters represents an evolutionary change in conventional computing to achieve high performance computing. However, these systems may exhibit coherency problems when parallel programs access shared resources, creating timing-related problems that are hard to debug. It is therefore crucial to have appropriate tools to monitor, trace, and analyze system execution, in order to identify functional and performance problems. The trace facility is aimed at keeping track of the functional flow globally, and at reporting relevant changes at certain times. The problem with global trace analysis is that the cores of each node in a cluster have their own clock, which is not synchronized with the others [75]. Dealing with this problem becomes even more complicated in multilevel tracing (tracing in virtual machines, middleware, and application layers).

The Linux Trace Toolkit next generation (LTTng) [17], developed at the École Polytechnique de Montréal, provides a detailed execution trace of the Linux operating system with low overhead. LTTng is capable of handling huge traces, in the order of several gigabytes. However, a new method is required to handle these large traces, while at the same time

allowing the traces from multiple systems and embedded devices to be collected for online analysis and viewing. Furthermore, the LTTng user expects to see the output analysis in real time, in order to diagnose problems live. Consequently, LTTng should be capable of visualizing traces from several distributed systems on a common reference time base. In a computer cluster, multiple nodes produce separate trace streams independently, and there are timestamps associated with each event. Since timestamps are recorded based on a local time that runs natively on each node, a logical ordering of events cannot be guaranteed. The objective of trace synchronization, with a high degree of precision and a low level of intrusiveness has been achieved for a posteriori analysis [85].

Our objective in this paper is to improve the time synchronization algorithms for the live analysis of streaming mode traces recorded on distributed nodes. The main contributions of this work can be summarized as follows :

- Various sliding window approaches are proposed and compared. The functional and performance problems related to these approaches are identified, along with the best approach for real-time analysis.
- We introduce a new algorithm for online time synchronization. This new algorithm efficiently filters new packet send-receive pairs with a simple test to only retain those that can change the synchronization parameters. These remaining packet pairs are processed in constant average time. Furthermore, the synchronization parameters are incrementally updated in constant time as these packet pairs are processed.
- The proposed algorithm improves the streaming time synchronization of traces recorded on distributed nodes with high precision and low intrusiveness.

The paper is organized as follows : first, we discuss related work, and several existing methods for synchronization are introduced. In Section 3.4, we present the details of an open source implementation that uses kernel-level tracing. In Section 3.5, we define the terms needed to describe clock behavior, and introduce the notation used in the remainder of the paper. The proposed model is illustrated in Section 3.6. In Section 3.7, experimental results are provided. In Section 3.8, our conclusion is presented and future work is discussed.

3.3 Related Work

3.3.1 Offline Clock Synchronization

Synchronization algorithms are classified as either *offline* or *online*. Offline synchronization focuses on trace synchronization at the end of tracing and after the traces have been collected. A major offline clock synchronization method has been proposed by Duda et al. [37], which consists of two synchronization algorithms, *Linear Regression* and *Convex-Hull*.

These algorithms estimate a conversion function between a pair of clocks. In both algorithms, the conversion function is linear, and the drift and offset between the two clocks are extracted from this linear model [58]. It also is used to estimate the one-way delay between two nodes by Moon et al. [82]. However, the *Convex-Hull* algorithm guarantees the highest synchronization accuracy [59].

In a two-dimensional space, based on the times of nodes A and B, the Linear Regression algorithm attempts to map all the points on a line. Thus, every point will affect the position of that line. In reality, network latency and similar events between two nodes can cause problematic outlying points, biased being only late. These outliers should ideally not affect the delay or offset computed from the linear regression. They should, in fact, be ignored in order to increase accuracy [14]. The *Convex-Hull* is an accurate algorithm that assumes minimum and maximum delay (sending time and receiving time). In this way, it finds the area that has minimum latency and ignores outlying points. As a result, the estimated line is more accurate with this algorithm than with Linear Regression.

Khlifi et al. [63] propose two algorithms, which they call *the average* and *direct skew removal* techniques for offline skew removal. The *average* algorithm calculates the average delay for a fixed number of consecutive packets at the beginning and the end of a trace, yielding a constant $O(1)$ complexity. The *direct skew removal* technique has the interesting property of being able to account for low clock resolution, where the clock granularity may be larger (e.g. 1ms) than the packet delay. For this purpose, the whole trace is analyzed for a linear $O(n)$ complexity.

They also present two techniques for online skew estimation and removal. The first one, *sliding window*, monitors the minimum delays to reduce the gap between the true and the corrected delays, (the correction being the estimated skew). To improve the accuracy, they present a second technique, the combined approach. They perform the sliding window algorithm to quickly estimate the skew, in the first interval. Then, they use the *Convex-Hull* algorithm during subsequent intervals to improve the accuracy. By contrast, our proposed approach is based on the *Convex-Hull* algorithm combined with lines with minimum and maximum possible slopes between the hulls. This can provide skew estimates very early, obviating the need for a different method in the first interval. Furthermore, our proposed algorithm recomputes the skew incrementally in $O(1)$.

Clement et al. [22] have evaluated the impact of system characteristics on trace synchronization accuracy. First, they studied the tracing duration impact. They propose dividing long duration traces into 30 minute segments, since the error in the clock drift linear approximation begins to increase significantly after approximately 45 minutes of tracing, while it is almost stable during the first 30 minutes. The error increases because of the variation in

the clock drift with time, as shown by the *Allan deviation* [2], and because of environmental effects on the clock circuit frequency, such as temperature and supply voltage variations.

Then, they studied the impact of the system load parameter, when there is a heavy load on major subsystems, CPU, memory, and hard disk. They found that the transmission time and response time measurement variations, caused by interrupt latency in a loaded system, influence the clock drift computation directly, and subsequently the time synchronization accuracy. The third parameter studied was hop count, when there is more than one path between two nodes. In that case, the offset between the two indirectly linked nodes may be computed by adding the offsets along a path, from one intermediate hop to the next. A path with fewer hops generally provides higher accuracy. If there is a direct path, it is normally better to choose that one to synchronize two nodes.

Poirier et al. [85] present an accurate method for synchronizing distributed traces. This method is applied to traces recorded at the kernel level with low intrusiveness. They apply the *Convex-Hull* algorithm to the clocks of traced nodes as a conversion function. If collected traces are huge and involve numerous nodes, their method is time consuming. In [56], the proposed method estimates accurate paths in large computer clusters and improves the performance of offline distributed trace synchronization. Since their algorithm was designed for post-processing, the analysis delay was not major concern. However, for a live display of traces, the latency should be minimized.

3.3.2 Online Clock Synchronization

By contrast, online synchronization works in streaming mode. Several researchers have proposed algorithms for this application. The standard clock synchronization method, widely used today, is the Network Time Protocol Daemon (NTPD) [80]. It sets and maintains the kernel system clock, used to measure packet send and receive time, based on feedback from exchanges with the server. Veitch et al. and Ridoux et al. [87, 102] proposed the RAD clock (Robust Absolute and Difference Clock), which provides alternative clock synchronization algorithms. The timing packets are timestamped using raw packet timestamps. They estimate the clock skew based on the difference between the system clock and the timestamps received from the server, and maintain the clock skew correction without changing the raw system clock, in a feed-forward approach.

In some cases, where the casual ordering of events is sufficient to achieve the application objectives, logical clocks have been proposed. Synchronization based on logical time considers each pair of related events (such as the send event for a packet, which necessarily occurs before the receive event) and orders them. Lamport [69], a pioneer in this area, formalized this concept and proposed algorithms to compose logical clocks. The Lamport algorithm

is fairly limited in what it can order, however. To tackle this weakness, *vector clocks* were independently proposed by Fidge [42] and Mattern [76]. The problem with vector clocks is their use of vectors to track events, since the size of each vector grows with the number of nodes.

However, in tracing and monitoring systems, the exact time of occurrence of each event, and the speed of identification of the relationships between events, are sufficient to isolate the probable problem and measure performance [94]. Consequently, the elapsed time on a physical clock is considered necessary for synchronizing nodes and analyzing their behavior with LTTng. Cristian [25] proposes using a timeserver to synchronize physical clocks. However, the problem with the timeserver is that it must tolerate occasional client readings. The Berkeley algorithm [47] assumes that no machine has an accurate time source. The server estimates the clients' local time and yields an average overall time, informing each client of their offset. However, it does not provide the accuracy required for trace analysis. Consequently, other physical time synchronization algorithms are being considered, which focus on correcting the difference between each pair of clocks, and relating each clock to a uniform reference time base. We provide further details about these algorithms in the next Section.

3.4 Kernel-Level Event Tracing

3.4.1 Tracer

Over the years, different tools have been implemented to enhance trace operating system behavior by recording kernel events. Some of the most applicable tracing tools are *Ftrace*, *Dtrace*, *Systemtap*, and *LTTng* [29]. The trace visualization tools currently available have often targeted detailed traces for small real time embedded systems, or much less detailed system logs for larger systems [16]. All tracing tools rely on local clock synchronization, and incur a significant loss of accuracy in the process. The proposed model is for *LTTng* tracer in online mode. The most significant challenge for all tracing tools is to minimize the impact of tracing on the traced computer. *LTTng* not only has very low overhead, but it is also able to trace kernel space and user space activities. These specific characteristics of *LTTng* help it monitor a broad range of activities in a computer.

3.4.2 Time Stamp Counter

The Time Stamp Counter (TSC) is a special register in the hardware that counts the number of ticks in the computer, which provides high-resolution, low-overhead, and fine-grained processor timing information [31]. If the processor frequency changes, the *TSC* rate will change as well on most systems. When the processor goes into idle or halted mode, the

TSC may stop altogether. With the advent of multiple processors on a single motherboard, the *TSC*s on different processors may drift away from each other over time, and there is no guarantee that they will ever resynchronize. A new generation of multicore processors includes a constant rate *TSC*, which provides for synchronization of the cores even though their frequency may vary over time [29]. For the kernel and user space tracer, the *TSC* must be the only time source. The trace clock is implemented based on TSC in *LTTng*, which is able to detect various problems that may arise from improperly synchronized *TSC*s in multicore systems and react appropriately. We generate events at the lowest possible point in the network stack, just ahead of the interface with the *NIC* driver [85]. Using kernel-level event tracing allows us to timestamp a packet transmission after data have been transferred from an application to the operating system, and after they have been processed by the networking stack. This results in lower timestamping delay than when messages are recorded at the application level, and means that every combination of application and hardware is supported. Since trace events are recorded locally, there is no need to modify the packets. This contributes to keeping the intrusiveness of tracing low.

3.5 Terminology and background

In this section, we introduce the terminology used in the remainder of the paper, and we formalize the definition of the clock skew. Time offset, frequency offset, and frequency offset rate are parameters that describe the behavior of a clock, and they differ from one clock to another. The trajectory of the time offset can be modeled by the following equation [39] :

$$\Delta T(t) = \beta(t_0) + \alpha(t_0)(t - t_0) + \ell(t - t_0)^2 + \in(t) \tag{3.1}$$

$\Delta T(t)$	Time offset at time t
$\beta(t_0)$	Initial offset
$\alpha(t_0)$	Frequency offset
ℓ	Frequency drift
$\in (t)$	Other factors, particularly random perturbations

Equation 3.1 shows that clock inaccuracies are caused by a combination of various factors. Over relatively short intervals, many algorithms consider that only the initial offset and the frequency offset are significant. We will refer to this as the ”linear clock approximation”. Taking this approximation into account, equation 3.1 can be simplified to :

$$\Delta T(t) = \beta(t_0) + \alpha(t_0)(t - t_0) \tag{3.2}$$

Finding the time offset between a node clock and a virtual perfect clock becomes a matter of identifying two factors in a linear equation. It follows that the offset between two real clocks can also be modeled as a linear function. For the rest of this paper, we estimate a function that maps the time on clock A to the time on clock B as follows :

$$C_A(t) = \alpha_0 + \alpha_1 C_B(t) \tag{3.3}$$

Moreover, the structure of a trace can be illustrated as follows :

$$\begin{aligned} &T = (drift, offset, start_time_from_TSC, events) \\ &events = (e_1, e_2, e_3, ..., e_n) \end{aligned} \tag{3.4}$$

Let us assume that there are two traces in a distributed system, T_0 and T_1, on computers C_0 and C_1 respectively. Two event types are considered for time synchronization : (i) sending a message, and (ii) receiving a message. Let us denote by θ_i the time when C_0 sends message i to C_1, and by ξ_i the time when C_1 receives message i from C_0.

$$m(i) : T_0(\theta_i) \mapsto T_1(\xi_i) \tag{3.5}$$

The timestamp for the sent message is stored in T_0 and the timestamp for the received message is stored in T_1. θ_i and ξ_i are based on the local time of C_0 and C_1 respectively. In addition, C_1 sends message j to C_0, and θ_j is the time of when C_0 receives message j from C_1.

$$m(j) : T_1(\xi_j) \mapsto T_0(\theta_j) \tag{3.6}$$

Each trace contains sent (S) and received (R) message timestamps, based on local time, as expressed by the following sets :

$$\begin{aligned} &T_0.events = (\theta_i^S, \theta_j^R, ...) \\ &T_1.events = (\xi_i^R, \xi_j^S, ...) \\ &i, j = 1, 2, 3, ... \end{aligned} \tag{3.7}$$

As shown in sets T_0 and T_1, $\overrightarrow{(\theta_i^S, \xi_i^R)}$ is the first pair of send-receive times for the message sent by C_0 to C_1, and $\overleftarrow{(\theta_j^R, \xi_j^S)}$ is the second pair of send-receive times for the message sent by C_1 to C_0. If the event timestamps of T_0 are considered as references times, this gives us the following equation :

$$\begin{aligned} &C_{T_0}(t) = \theta^S \\ &C_{T_1}(t) = \alpha\theta^S + \beta \end{aligned} \tag{3.8}$$

3.6 Proposed Model

In online time synchronization, we are dealing with streaming data, the distinguishing feature of which is the speed of the stream flow. Thus, it is not practical to scan the data stream more than once. Buffering the data stream for a long time is another challenge that we have to address, because of the huge amount of data usually in the stream. Consequently, because of the limited amount of space for storing stream data, there is a trade-off between memory and accuracy [35]. Ideally, an online synchronization algorithm should be efficient in terms of both time and memory. In addition, the synchronization algorithm should be scalable, have a consistently low synchronization computation latency, and maintain its accuracy over time. Below, we introduce some common techniques for dealing with streaming data :

Random Sampling analysis

One solution to working with a large dataset is to sample the stream at periodic intervals, instead of handling all the data at once. An unbiased sampling requires some information in advance, such as the length of the stream. Reservoir sampling selects an unbiased random sample containing N elements without replacement. The idea is to maintain a sample, called the *reservoir*, from the random samples generated (each sample uniformly has N elements) [44].

Sliding Window analysis

Rather than taking a random sample of the streamed data, a *sliding window* model can be used to analyze them. The idea is to use recent data to make a decision, instead of working with all the data seen so far. This means that, if we have a window of size L, an element that arrives at t will expire at $t + L$. Limiting the amount of data in this way reduces memory space requirements [48], and is efficient in terms of both time and memory.

Incremental analysis

The idea behind this approach is to analyze the data as soon as it is received and then discard most of it from memory. Here, the analysis speed must be the same as, or higher than, the rate at which the data is received. Otherwise, memory buffer overflow will result.

We propose and evaluate three approaches based on sliding windows and one incremental approach.

3.6.1 Model

Figure 3.1 illustrates the basic architecture of the proposed model, which supports both window-based approaches and the Fully Incremental approach. The following steps would be performed in this organization.

Processing module : traces T_0 and T_1 are gathered from two distributed systems in a computer cluster for online analysis. In order to compute the clock differences between the two nodes, there must be network traffic between them. The packet exchange events e_i are extracted and dispatched to the next module. So, this module captures both network traffic and computer activity, and extracts the necessary information for the matching module.

$$TraceSetContext = (T_0, T_1) \tag{3.9}$$

As shown, there are two approaches : (i) *Window-based approaches* : In these approaches, we have to read the traces for a particular time window. Each window is completely disjoint, i.e. the windows do not overlap. At the window end, we finalize synchronization to obtain the synchronization factors (drift and offset) per analysis module. (ii) *Fully Incremental approach :* In this approach, events are read continuously. When the analysis module finds an accurate pair, it updates the synchronization parameters. In the subsections below, we define the term accurate pair.

Matching module : event processing feeds the events into the matching module one by one. In this module, an attempt is made to match one event from a trace to the corresponding event in another trace. Once we have a send or receive event, the matching module looks in the associated table for an unmatched receive/send packet, or enters the send/receive packet as unmatched. The cost of this operation is $O(1)$. This involves a direct lookup in a hash table using a so-called SegmentKey, which is based on TCP and IP packet headers (source and destination IP and port, as well as sequence number). Care is required in order to limit memory usage in the matching module, by removing unmatched packets periodically.

Events may be unmatched for several reasons, as illustrated in the following two scenarios. In scenario *(i)*, the tracing start and end times in the systems are not the same. Suppose that we start with two traces, A and B, A starting 2 minutes before B. If we scan A first, inserting all the unmatched send/receive events that occurred during these 2 minutes, when we arrive at B, we could try to match all the send/receive events from B, but without success. In scenario *(ii)*, lost events cause corresponding events to remain unmatched. Events are lost either when the tracing buffers are not large enough to handle bursts, or when the rate of events arrival is larger than the available bandwidth to send/save the buffers. It is also possible for a packet to be dropped by the network. When a send or a receive event is lost, the

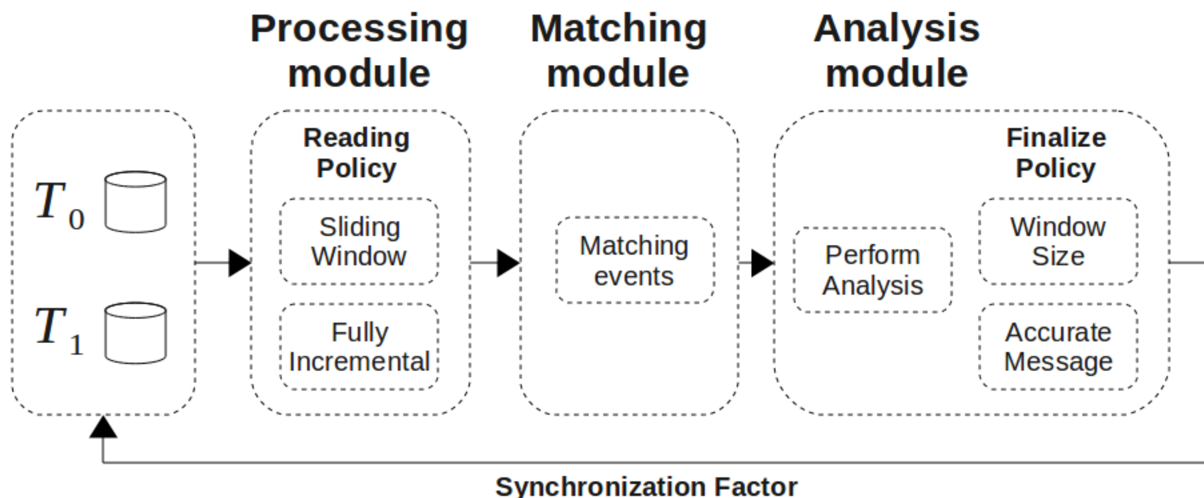


Figure 3.1 Two different approaches for online synchronization.

corresponding receive/send event will remain unmatched, and this may make the matching procedure inefficient, as memory is taken up by needlessly storing the event. Indeed, while it may be possible to match the event at a later time, the corresponding packet will have been significantly delayed (e.g. by taking a different route), and high latency exchanges are inaccurate. Moreover, even if we only have communication in a single direction (unlikely because of TCP ACK packets), once several later packets have been matched, we can deduce that a late match would not create a data point of interest and so we can clear that packet.

Our proposed strategy for addressing these problems is to scan all the events available to date from both traces, and so obtain a first time offset estimate. Subsequently, after a threshold (T_buffer) time, a new hash table is created, the existing one is kept as a previous table, and the former previous table is discarded. The new packets are inserted into the new table, but matching is attempted in the new and previous tables. The T_buffer can be very large, but should be sufficiently small that unmatched packets cannot accumulate to the extent of causing a memory problem. As a result, the proposed strategy flushes the older packets, while keeping the $O(1)$ complexity per packet examined.

If one trace started much earlier than the other, this may overload the memory by storing packets that will never be matched. If we have no hint of the relative timing of the traces, the following are possible : trace 1 may be completed before trace 2, or performed after it. If we do have a hint of the relative timing of the traces, this could provide us with a good estimate of where to start. The timing can come from : *(i)* trace headers or trace buffer headers relating TSC to real-time clocks, assuming that these clocks are synchronized to

within a few seconds ; *(ii)* in streaming mode, if the trace is live, the last buffer of events is received from each trace at the same time, or within a few seconds. If we have no hint as to timing (the traces are already completed, or they are streaming, but with a long delay), and are afraid that inserting all the packet events will cause the memory to overflow, we could use the following strategy. If the traces are disjoint, no synchronization is possible/useful. If the traces intersect, the beginning or end of one trace must intersect with a part of the other trace. In the latter case, we could insert a constant number of packets from the beginning and end of one trace, and then try to match all the packets from the other trace. This would allow us to find an initial time estimate in linear time and constant memory.

Analysis module : the analysis module receives a matched send/receive pair and applies the *Convex-Hull* algorithm in different ways, using one of the various approaches presented in this paper. This module should return synchronization factors for each trace pair of communicating computers.

3.6.2 Convex-Hull

As shown in Figure 3.2, pairs are divided into two sets, based on the message direction. Consequently, the synchronization estimation line should be below all the pairs $\{\overrightarrow{(\theta_i^S, \xi_i^R)}$, $\overrightarrow{(\theta_{i+1}^S, \xi_{i+1}^R)}$, ...} and above all pairs $\{\overleftarrow{(\theta_j^R, \xi_j^S)}$, $\overleftarrow{(\theta_{j+}^R, \xi_{j+1}^S)}$, ...}.

The packets with minimum latency are those of interest in the *Convex-Hull* synchronization algorithm. Packets sent from θ (horizontal axis) to ξ (vertical axis) occupy the upper left half-plane and are shifted higher when more network latency was encountered. Therefore, the lower half-hull, of the *Convex-Hull* formed by those points, is a lower bound for the packets sent from θ and identifies the packets with the lowest latency. Similarly, packets sent from ξ (vertical axis) to θ (horizontal axis) occupy the lower right half-plane and are shifted to the right when more network latency was encountered. Therefore, the upper half-hull, of the *Convex-Hull* formed by those points, is an upper bound for the packets sent from ξ and identifies the packets with the lowest latency. The possible synchronization lines lie below the lower half-hull of packets sent from θ and above the upper half-hull of packets sent from ξ.

Synchronization accuracy is limited by the delay between the send and receive events. Any packet delayed by interrupts, network switch delay, or some other cause will lead to an inaccurate pair and a long delay. The *Convex-Hull* algorithm selects the pairs on the inside envelope between the sent time and the received time, and so identifies the most accurate pairs with the shortest delay. In other words, it finds the area that has minimum latency and ignores outlying pairs, as shown in Figure 3.2 ($\overrightarrow{(\theta_2^S, \xi_2^R)}$, $\overrightarrow{(\theta_5^S, \xi_5^R)}$, $\overleftarrow{(\theta_2^R, \xi_2^S)}$, $\overleftarrow{(\theta_3^R, \xi_3^S)}$). This means that the estimated line is more accurate than with Linear Regression, which is affected by all pairs, including delayed packets.

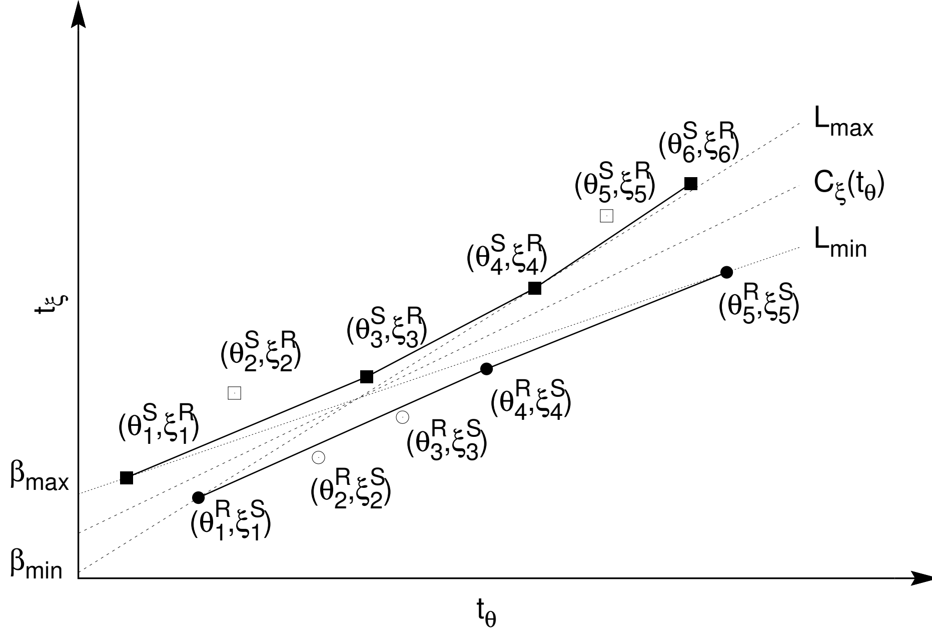


Figure 3.2 Convex-hull method.

According to the above definition, we have two completely separate sets, because there is no message inversion (receive before send). Each set attempts to find the optimal separator (the solid line in each hull), as points in each set are nearest to the separator space. Graham's scan algorithm forms these two separated sets. The bounds of the *Convex-Hull*, shown in Figure 3.2, are :

$$\begin{aligned} UpperBound &= \{\overleftarrow{(\theta_1^R, \xi_1^S)}, \overleftarrow{(\theta_4^R, \xi_4^S)}, \overleftarrow{(\theta_5^R, \xi_5^S)}\} \\ LowerBound &= \{\overrightarrow{(\theta_1^S, \xi_1^R)}, \overrightarrow{(\theta_3^S, \xi_3^R)}, \overrightarrow{(\theta_4^S, \xi_4^R)}, \overrightarrow{(\theta_6^S, \xi_6^R)}\} \end{aligned} \tag{3.10}$$

Thus, the maximum likelihood estimators are between the following conditions :

$$\begin{aligned} &\alpha\theta_i^S + \beta < \xi_i^R \\ &\alpha\xi_j^R + \beta < \theta_j^S \\ &i, j = 1, 2, ..., n \end{aligned} \tag{3.11}$$

The next step is to find two lines, one with maximum slope (L_{max}) and another with minimum slope (L_{min}) :

$$\begin{aligned} L_{max} &= \alpha_{max}\theta + \beta_{min} \\ L_{min} &= \alpha_{min}\theta + \beta_{max} \end{aligned} \tag{3.12}$$

As a result, the final estimated α and β are certainly limited to the area that is enclosed

between $(\alpha_{min}, \alpha_{max})$ and $(\beta_{min}, \beta_{max})$, and the selected synchronization line is the bisector of L_{max} and L_{min}.

When synchronizing two traces, the relationship between them can be one of the following :

Definition 1) An accurate relationship : this is the expected case, shown in Figure 3.2, where L_{max} and L_{min} are available and their middle can be computed. If the relationship between two clocks is of the accurate type, we can define the accuracy metric as the difference between the minimum and maximum possible drifts between the two clocks.

$$Accuracy(i) = L_B^{max}.drift - L_B^{min}.drift; \tag{3.13}$$

Definition 2) An approximate relationship : L_{max} and L_{min} are not available because the hulls do not satisfy the hypothesis that the upper half-hull should be below the lower half-hull and not intersect with it. The approximation is a "best effort".

Definition 3) An incomplete relationship : only one of the L_{max} and L_{min} lines is available. There is communication in only one direction, which is insufficient to obtain a proper bounded synchronization.

Definition 4) An absent relationship : there is no communication between the nodes in either direction, and nothing can be deduced about their relative time.

Definition 5) Fail relationship : none of the L_{max} and L_{min} lines is available. This is because the hulls intersect each other or are reversed.

3.6.3 Window-based Approach

As mentioned, one of the applicable methods for streaming data is the window-based technique. Each window is completely disjoint, i.e. windows do not overlap. With this method, the analysis is performed on the data in the current window instead of the whole data. It is, however, possible to reuse synchronization parameters or even accurate pairs from previous windows. The advantage of this method is to use the most accurate packets that were detected in previous windows for synchronizing time in the current window. Thus, the synchronization results for each window are stored to be potentially used in subsequent windows.

In some cases, after refining the synchronization at the end of one window, the synchronized time in the next window for events in one trace may change, relative to the events in the other trace. As a consequence, the window start time must be adjusted to avoid skipping some events in one trace, and events from the other trace that may already have been read can be ignored.

Algorithm 1 illustrates the pseudocode of the window-based approach. One of the inputs to this pseudocode is window size (l). There is a tradeoff in determining l. If l is too large,

ALGORITHM 1: Window-based approach

```
Require: L : window-size
Require: TtoS() : convert from Trace time to Synchronized time
Require: StoT() : convert from Synchronized time to Trace time
 1: T[0].offset= 0
 2: T[0].drift= 1
 3: T[0].prevWindowTime_t= T[0].startTime_t
 4: T[0].startTime_s= TtoS(T[0].offset, T[0].drift, T[0].startTime_t)
 5: T[1].offset= 0
 6: T[1].drift= 1
 7: T[1].prevWindowTime_t= T[1].startTime_t
 8: T[1].startTime_s= TtoS(T[1].offset, T[1].drift, T[1].startTime_t)
 9: w.startTime= min(T[0].startTime_s, T[1].startTime_s)
10: w.endTime= max(T[0].startTime_s, T[1].startTime_s) + L
11: loop
12:   T[0].endWindowTime_t= StoT(T[0].offset, T[0].drift, w.endTime)
13:   T[1].endWindowTime_t= StoT(T[1].offset, T[1].drift, w.endTime)
14:   for (e = readEvent() and e.time < TSC.T[e.index].endWindowTime_t) do
15:     if e.time > T[e.index].preWindowTime_t then
16:       P= matching(e)
17:       if P is not null then
18:         performAnalysis(P)
19:       end if
20:     end if
21:   end for
22:   finalize-sync()
23:   UpdateViewer()
24:   T[0].endWindowTime_s= TtoS(T[0].offset, T[0].drift, TSC.T[0].endWindowTime_t)
25:   T[1].endWindowTime_s= TtoS(T[1].offset, T[1].drift, TSC.T[1].endWindowTime_t)
26:   w.startTime= min(T[0].endWindowTime_s, T[1].endWindowTime_s)
27:   w.endTime= w.startTime + L
28:   T[0].prevWindowTime_t= T[0].endWindowTime_t
29:   T[1].prevWindowTime_t= T[1].endWindowTime_t
30: end loop
```

there is a latency period before an accurate synchronization is obtained. If l is too small, the precision of the algorithm suffers. However, the total computation time should be unaffected. We used heuristics to determine the window size.

Having a sufficiently large window is essential for a synchronization. If the window is too small, then the traces are being synchronized onto a space of insufficient dimension, in which there is not any accurate packet to improve synchronization. Thus, the delay caused by running several useless synchronizations impacts online synchronization performance. Moreover, a too large window also produces problems : postponing synchronization to the end of each window delays the user visible reaction time. These algorithms also store more data for analysis, impacting memory usage. We tested different window sizes and the experiment results lead us to select a window size equal to three seconds for the experimented datasets.

Other inputs to this algorithm are the two traces, T_0 and T_1. First, all drifts and offsets are set to 1 and 0 respectively, and the traces are ready for the first window. The start and end of a window are updated based on the reference time. We define two boundaries for each trace : *prevWindowTime_t* and *endWindowTime_t*. The *prevWindowTime_t* boundary first refers to the trace start time (line 4 and line 7) and then refers to the previous *endWindowTime_t*, and the *endWindowTime_t* boundary is updated based on *w.endTime*. To calculate *endWindowTime_t* we use *StoT* function because *w.endTime* is based on the reference time and has to be converted to the local time of the trace.

These boundaries not only update the window end time, but also avoid events being read which were read earlier. First, the boundaries of each trace are converted by the *TtoS* function to define the first window. This function converts local time to synchronized time by applying the current synchronization factors of the trace. Then, the minimum and maximum start time between two traces are extracted. The first window starts from the minimum time to avoid skipping events and ends at the maximum time plus l. This means that the first window is the largest. The important point is that new synchronization factors may change the relative trace time, as Figure 3.3b illustrates. Therefore, the *w.startTime* of the new window may not refer to the end of the previous window. This means that, as line 26 indicates, the start time of the new window is the minimum of two previous end window times of traces, to avoid any gap. In fact, we should avoid reading events that were read earlier. To tackle this challenge, we compute two boundaries for each trace. Event reading is performed based on these two boundaries in each trace (line 15), and in this way duplicate reading is avoided. Function *matching* pairs one event from a trace with the corresponding event in the other trace (p). The matched events are sent to the analysis module as a packet to analyze based on the *Convex-Hull* algorithm (line 18). At the window end, we finalize the synchronization to obtain new synchronization factors after reading all the events (line 22). Finally, we update

the trace viewer [85].

Different models for window-based approach

We have compared three different approaches for analyzing information one window at a time, and accordingly recompute the time synchronization.

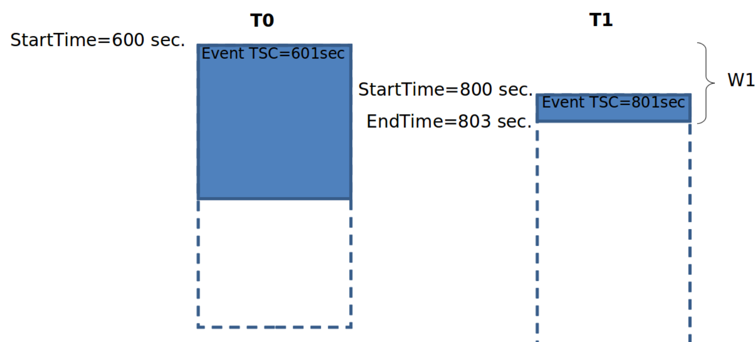
Independent windows : The idea behind synchronizing the computers in streaming mode is to consider only the traffic information in the current window. No relationship is maintained between windows. One of the advantages of this approach is that no buffered data or computations are passed from the previous window to the current window, which minimizes analysis time and complexity. The disadvantage of this method is that it is not capable of achieving a satisfactory level of accuracy, not only in each window, but also at the end of the process.

Replace : Reusing the accurate results from the *Convex-Hull* analysis of the previous windows is the objective of the *Replace* approach. As mentioned before, the result of trace synchronization may be : "accurate", "approximate", "incomplete", "absent", or "fail". The idea behind this approach is to compare the current window's accuracy with that of the previous window. In this way, we compare the accuracies that we have so far. $Accuracy_i$ represents the accuracy of window i and $Accuracy_{(i-1)}$ represents the accuracy of the previous window in Eq. 3.14.

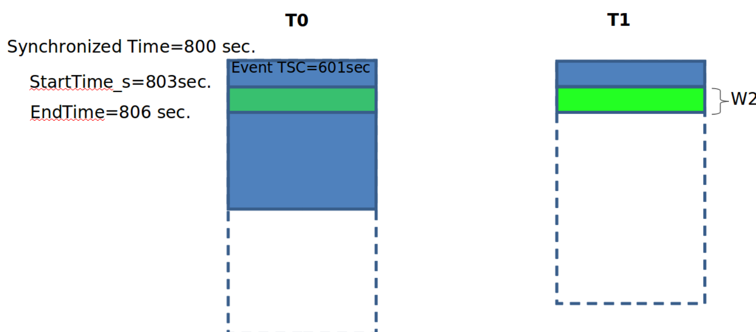
$$\begin{aligned} &\varepsilon_i = Accuracy_{(i-1)} - Accuracy_i \\ &if \quad \varepsilon_i > 0 \quad replace \end{aligned} \tag{3.14}$$

If $\varepsilon_i < 0$, the current window packets have not improved the accuracy and should be ignored. If $\varepsilon_i > 0$, the current window packets have improved in accuracy. In that case, the current accuracy becomes the new best accuracy, and all the points in the *Convex-Hull* are replaced by the current window points. If the current synchronization state is "accurate" while the previous state was "approximate", "incomplete", "absent", or "fail", the replacement takes place automatically. The same applies if the current window synchronization is "approximate" and the previous window is "approximate" (conditional replacement) or one of the inferior states (automatic replacement). This approach is somewhat incremental, and improves accuracy over time, but relatively slowly.

Correlated : This approach selects the accurate packets in each window and transfers them to the next window to initialize the *Convex-Hull*. When a new packet with a small delay is detected, it is added to the set of accurate packets. As shown in Figure 3.4, this method stores pairs $\{\overleftarrow{(\theta_1^R, \xi_1^S)}, \overleftarrow{(\theta_2^R, \xi_2^S)}, \overleftarrow{(\theta_3^R, \xi_3^S)}\}$ in the upper bound list, and pairs $\{\overrightarrow{(\theta_1^S, \xi_1^R)}, \overrightarrow{(\theta_2^S, \xi_2^R)}, \overrightarrow{(\theta_3^S, \xi_3^R)}, \overrightarrow{(\theta_4^S, \xi_4^R)}\}$ in the lower bound list of the first window. Once it finds $\{\overleftarrow{(\theta_5^R, \xi_5^S)}, \overrightarrow{(\theta_7^S, \xi_7^R)}\}$



(a) before sync. (first window)



(b) after sync. (second window)

Figure 3.3 The local clock values used for traces T0 and T1 may be highly desynchronized. Two traces starting about at the same time may see start times of 600sec. and 800sec. on their local clocks, respectively. With a window size of 3sec., the first window, W1, will go from 600sec. (minimum start time) to 803sec. (maximum start time plus window size). After processing the first time window, and analyzing matching events, it may be computed that trace T0 should be offset by -200sec., using T1 as time reference. The second time window, W2, is from 803sec. to 806sec., based on the reference time of T1. This corresponds to 603sec. to 606sec. in T0 based on its local clock. After synchronization, we realize that events in T0 for time range W2 have already been processed as part of W1. These already read events are skipped.

more accurate pairs in the second window, producing a narrower channel between the hulls, it replaces them as the most accurate pairs and saves them for future use in the next windows. In the case where there is no pair as accurate as any of the previous stored pairs in the new window, the method keeps the previous synchronization factors for that window. In this way, the correlated sliding windows keep the performance of the overall analysis. The advantage of this method is that it uses the history of the accurate packets and improves the set over time, based on new exchanged packet information. Moreover, it stores little information about the most accurate points kept for future use, which means that, for the most part, buffering problems are avoided.

This method postpones the recomputation of the trace synchronization to the end of each time window, which reduces the time synchronization costs, in turn limiting the number of recomputations to the number of time windows. Still, it reaches the ideal accuracy at the end of each time window. There are many applications where this small latency period, waiting until the time window ends before obtaining the improved accuracy, is not a problem. The Correlated sliding window approach is suitable for such applications. For example, many system administration tools refresh the analysis view once every few seconds, and selection of the time window can match the display refresh cycle. By contrast, there are some applications that are time sensitive and require the resynchronization of traces as soon as the accuracy can be improved. This would be the case in several real-time applications, particularly for detecting security attacks and generating alerts based on distributed trace analysis. Any delay in such applications is to be avoided. It is this scenario that motivated our additional effort to develop a fully incremental algorithm.

3.6.4 Fully Incremental Approach

The *Convex-Hull* algorithm looks for the smallest difference between the sent and received timestamps. It finds the send and receive events exhibiting the minimum latency and ignores points further apart, and so restricts its computation space as much as possible. The first step of the proposed method for online synchronization is to synchronize the two traces as soon as possible. When the type of relationship is accurate, we have reached the first synchronization between traces. As illustrated in Figure 3.5a, pairs in the upper bound and pairs $(\overleftarrow{(\theta_1^R, \xi_1^S)}, \overleftarrow{(\theta_2^R, \xi_2^S)}, \overleftarrow{(\theta_3^R, \xi_3^S)})$ in upper bound and pairs $(\overrightarrow{(\theta_1^S, \xi_1^R)}, \overrightarrow{(\theta_3^S, \xi_3^R)}, \overrightarrow{(\theta_4^S, \xi_4^R)})$ in lower bound help to estimate the first time synchronization between two local times, C_0 and C_1.

This first estimation is not very accurate, but, as more packets are received, accuracy should improve over time. Accuracy is determined by the difference between α_{min} and α_{max}. The idea is to react only to matched packets that affect the slopes $(\alpha_{min}, \alpha_{max})$, and then

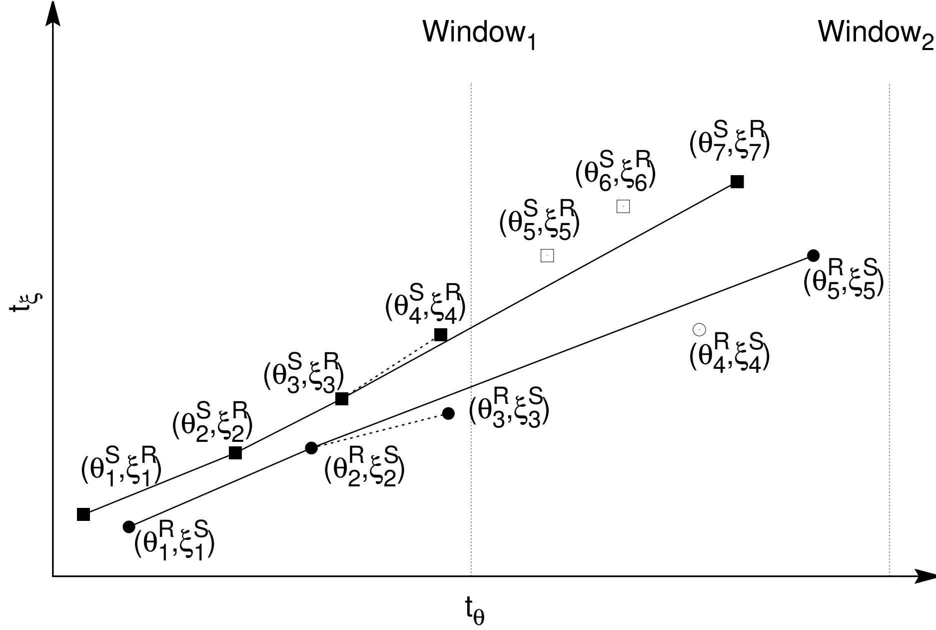


Figure 3.4 Correlated sliding window.

update the synchronization parameters in $O(1)$ time.

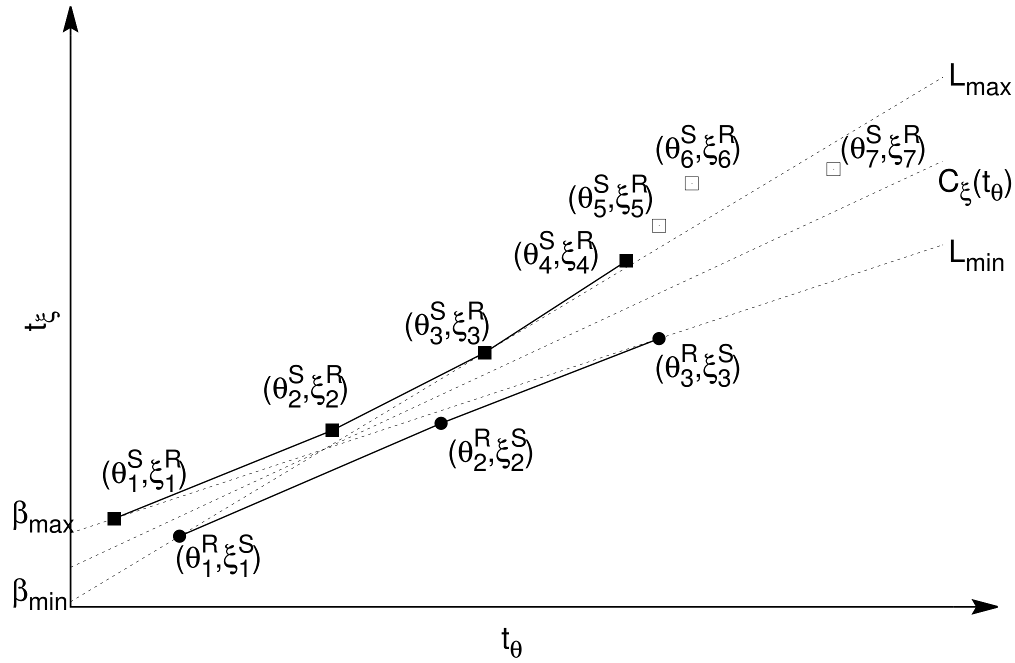
Definition 6) Accurate message : a matched pair will improve the synchronization accuracy if and only if it is below the line L_{max} in the lower hull or above the line L_{min} in the upper hull.

Removal of the right-hand pair from the upper or lower bound list is shown in Figure 3.6. Figure 3.6a illustrates the removal of a pair from the lower hull (θ_2^S, ξ_2^R), and Figure 3.6b illustrates the removal of a pair from the upper hull (θ_2^R, ξ_2^S). This simple test (above or below a line) is applied to every incoming matched pair, and quickly identifies the few pairs that actually change the slope of L_{min} and L_{max}, and improve accuracy, moving towards their center space.

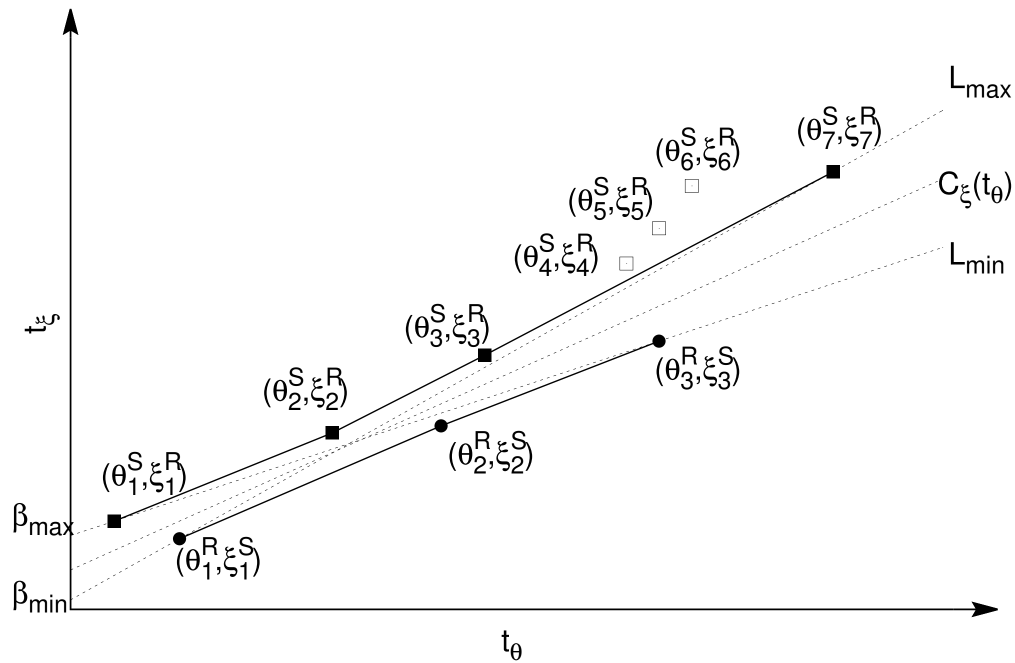
As shown in Figure 3.5b, most packet pairs are outside L_{min} and L_{max}, and cannot affect accuracy, and so are ignored. For example, pairs $\{(\theta_5^S, \xi_5^R), (\theta_6^S, \xi_6^R)\}$ do not change L_{max}, which means that accuracy cannot change. However, the new pair (θ_7^S, ξ_7^R) affects L_{max}, and accuracy is improved.

$$
\begin{aligned}
&\alpha_{max} > \tilde{\alpha}_{max} \\
&\alpha_{max} - \alpha_{min} > \tilde{\alpha}_{max} - \alpha_{min}
\end{aligned}
\tag{3.15}
$$

The same reasoning applies for L_{min}.

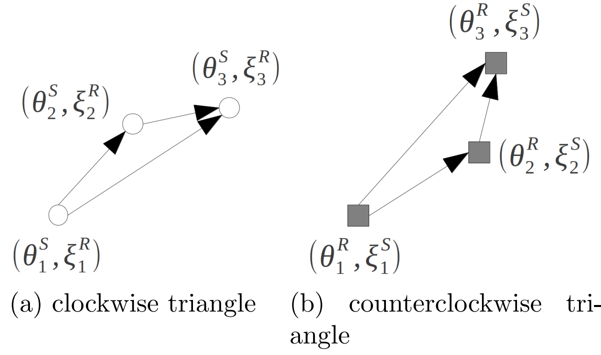


(a) The accurate packet, (θ_7^S, ξ_7^R), position before updating the synchronization



(b) Synchronization based on accurate packet

Figure 3.5 Fully Incremental Approach



(a) clockwise triangle (b) counterclockwise triangle

Figure 3.6 Geometric movement state in upper and lower hulls

$$\begin{aligned} &\alpha_{min} < \tilde{\alpha}_{min} \\ &\alpha_{max} - \alpha_{min} > \alpha_{max} - \tilde{\alpha}_{min} \end{aligned} \tag{3.16}$$

Therefore, our Fully Incremental approach calculates the new drift and offset between two traces on a pair located between lines L_{max} or L_{min}, after the first synchronization. It guarantees the best accuracy without waiting for a window end time. Thus, to manage L_{max} and L_{min}, the two points currently defining each of them must be stored.

$$\begin{aligned} &L_{max} = \{(\theta^R_{i_{min}}, \xi^S_{i_{min}}), (\theta^S_{i_{max}}, \xi^R_{i_{max}})\} \\ &L_{min} = \{(\theta^S_{i_{min}}, \xi^R_{i_{min}}), (\theta^R_{i_{max}}, \xi^S_{i_{max}})\} \\ &i_{min} \ \epsilon \ \{1, ..., n-1\} \\ &i_{max} \ \epsilon \ \{2, ..., n\} \end{aligned} \tag{3.17}$$

Unlike in the classic *Convex-Hull* algorithm, the pairs of points defining L_{max} and L_{min} are incrementally updated as new points are added.

Theorem. Each synchronization in the Fully Incremental approach requires $O(1)$ time, on average.

Proof 1. Algorithm 2 illustrates the Fully Incremental pseudocode. The Fully Incremental approach takes a packet as input (consisting of a pair of matched send-receive events from two traces). Lines 5 to 15 check whether the new packet belongs to the upper or the lower bound. Line 16 calls upon the *Qualify-message* function, to verify whether or not the new packet is an accurate one. L_{min} has a first point in the lower hull (delimiting the upper plane), $L_{min}.point_1$, such that the slope of the hull edge ahead of the point is smaller, and

the slope of the edge after it is larger. The second point, $L_{min}.point_2$, is in the upper hull (delimiting the lower plane), with the slope of the hull edge ahead of the point larger and the slope of the edge after it smaller. L_{min} has a smaller slope than L_{max}, and therefore $L_{min}.point_1$ must be ahead of $L_{max}.point_2$ on the lower hull, and $L_{max}.point_1$ must be ahead of $L_{min}.point_2$ on the upper hull. When a new packet arrives for the lower hull, if it is above L_{max}, it cannot affect L_{max} and is not of interest, even though it could be on the *Convex-Hull*. We call these points neglected hull points. However, if another point comes later which is below L_{max}, it will be considered, included in the hull, and change L_{max}, becoming the new $L_{max}.point_2$. It is important to note that any neglected hull point between the previous and the new $L_{max}.point_2$ would have formed a concave section (pairs $\{(\theta_5^S, \xi_5^R), (\theta_6^S, \xi_6^R)\}$ in Figure 3.5b) and been removed. Thus, behind $L_{max}.point_2$, we have the complete *Convex-Hull*. This is important because it ensures that the optimization that neglects some hull points ahead of $L_{max}.point_2$ does not affect the integrity of the hull behind $L_{max}.point_2$, and so does not interfere with the computation of $L_{min}.point_1$ (pair $\{(\theta_1^R, \xi_1^S)\}$) in Figure 3.5b), which is ahead of $L_{max}.point_2$ on the lower hull. Let (θ_i^S, ξ_i^R) and (θ_i^R, ξ_i^S) be the new pair positioned against the lower and upper bound respectively. The qualification can be performed by a *cross-product* function in the lower and upper bounds, as follows :

$$cross - product((\theta_i^S, \xi_i^R), (\theta_{i_{min}}^R, \xi_{i_{min}}^S), (\theta_{i_{max}}^S, \xi_{i_{max}}^R)) =$$

$$(\theta_{i_{min}}^R - \theta_i^S)(\xi_{i_{max}}^R - \xi_i^R) - (\xi_{i_{min}}^S - \xi_i^R)(\theta_{i_{max}}^S - \theta_i^S)$$

$$cross - product((\theta_i^R, \xi_i^S), (\theta_{i_{min}}^S, \xi_{i_{min}}^R), (\theta_{i_{max}}^R, \xi_{i_{max}}^S)) =$$

$$(\theta_{i_{min}}^S - \theta_i^R)(\xi_{i_{max}}^S - \xi_i^S) - (\xi_{i_{min}}^R - xi_i^S)(\theta_{i_{max}}^R - \theta_i^R)$$

The complexity of the cross-product function is $O(1)$. If the pair is not qualified, it is dropped, because it does not improve accuracy. Otherwise, it is added at the end of the bound list. Let p be the new matched packet. $< l_1, l_2, l_3, ..., l_{m-1}, l_m >$ and $< u_1, u_2, u_3, ..., u_{k-1}, u_k >$ denote the remaining points on the lower and upper bound lists respectively. If p qualifies as an accurate packet for the upper bound list, it becomes u_{k+1}. The next step, line 17, is running the Graham scan vertex addition procedure to performing the cross-product $cross - product(u_{k-1}, u_k, u_{k+1})$. If u_k is removed, then the $cross - product(u_{k-2}, u_{k-1}, u_{k+1})$ is performed. Each point is examined in turn, and is either removed and the processing continues, or is kept and the processing stops. The total number of points (matched pairs) to process being n, the number of iterations in the algorithm is n and the number of points that may reside in the upper or lower list is bounded by n. At every iteration, when a new point is added and several points from the hull may be removed, the number of operations required is in the order of the number of points removed, and may approach n. However, the total

number of points removed over all iterations is also bounded by n. Thus, for the n iterations, the average complexity for one iteration is $O(1)$. It is interesting to note that the number of points in the lower and upper lists is typically much lower than n, and often no more than 8.

The last step is *adjust-bounds* procedure. As mentioned, either L_{max} or L_{min} is necessarily affected when a packet pair qualifies. For instance, if the accurate packet is related to the upper bound, L_{min} has to be recomputed (but does not affect L_{max}). In the proposed approach, we update only one of the two lines, and the received accurate point simply replaces $L_{max}.point_2$ or $L_{min}.point_2$ (line 18). Algorithm 3 illustrates the pseudocode for updating the first point of the line L_{min} or L_{max}. For every new accurate point, when the slope of line L_{min} or L_{max} changes, its first point may advance, skipping a few points to reach one with a higher number in the bound list. Let $l.pos1$ denote the position of point l_{pos1} in the lower bound list, which is currently used as $L_{min}.point_1$. When the slope of L_{min} is updated (increased), it may become larger than the slope defined by points l_{pos1} and l_{pos1+1}. In that case, $l.pos1$ must be incrementally updated until the slope defined by points l_{pos1} and l_{pos1+1} becomes larger than the slope of the updated L_{min}. The number of operations required is in the order of the number of points skipped in the list. Here again, the number of points skipped in one slope update is bounded by n. However, the total number of points skipped over the whole incremental procedure is also bounded by n, and so the average complexity for the slope update per iteration is $O(1)$.

Line 20 in Algorithm 2 reveals an exceptional situation which may occur when $point_1$ of L_{max} or L_{min} is removed by the Graham scan. In this case, we also have to update the first point of the other line in the same way. Even doubling the number of operations in that worst-case situation does not change the algorithm complexity.

The Fully Incremental approach introduced here is particularly interesting, because it incrementally updates the synchronization immediately upon receiving more accurate packets, yet it has a worst case average complexity of $O(1)$ per packet. Moreover, very few packets qualify for synchronization updating, and a very small subset of points is typically retained to define the *Convex-Hull*. This makes the algorithm ideally suited for the targeted high performance trace analysis tools.

3.7 Experiments and evaluation

3.7.1 Experimental setup

In our model, we instrumented the Linux kernel version 2.6.26 using *LTTng*, and the tests were performed on seven Pentium III computers (4 CPUs) with 4 GB of RAM. The window size for the Correlated, Replace, and Independent approaches is 3 seconds.

ALGORITHM 2: Fully Incremental approach

```
Require: p : new packet
1: Upper_bound_list= < u_1, u_2, u_3, ..., u_{k-1}, u_k >
2: Lower_bound_list= < l_1, l_2, l_3, ..., l_{m-1}, l_m >
3: L_max =< point_1, point_2 >
4: L_min =< point_1, point_2 >
5: if p related to the upper bound then
6:    BoundList= Upper_bound_list
7:    OtherBoundList= Lower_bound_list
8:    Line= L_min
9:    OtherLine= L_max
10: else
11:    BoundList= Lower_bound_list
12:    OtherBoundList= Upper_bound_list
13:    Line= L_max
14:    OtherLine= L_min
15: end if
16: Qualify-message(Line, p)
17: Remove-useless-points(BoundList)
18: Line.point_2= p
19: Adjust-bounds(Line, OtherBoundList)
20: if OtherLine.point_1 has been removed then
21:    OtherLine.point_1= BoundList[0]
22:    Adjust-bounds(OtherLine, BoundList)
23: end if
```

ALGORITHM 3: Adjust-bounds

```
Require: Line
Require: OtherBoundList
1: pos1= Line.point_1
2: i = OtherBoundList_pos1
3: j= Line.point_2
4: while i ≤ OtherBoundList.length-1 do
5:    rotation= cross − product(i, i + 1, j)
6:    if the rotation is not optimal then
7:       if i+1.x < j.x then
8:          i++
9:       else
10:         Line.point_1 = NULL
11:      end if
12:   else
13:      Line.point_1 = i
14:   end if
15: end while
```

3.7.2 Packet matching and Convex-Hull points

Figure 3.7 illustrates the number of matched packets in each window for the window-based approaches. As shown, the maximum number is associated with the 198th window. As explained in Figure 3.3, we have two traces with two different S-TSCs that are far apart. We read more events from trace T_0 than from trace T_1 in the first window. Consequently, the rate of matched events depends on T_1, and we have many unmatched read events from T_0 that will be matched in subsequent (second and third) windows. Thus, we have a higher rate of matched events in the initial windows, the third window in this experiment. Since traces are synchronized at the end of the first window, the S-TSCs quickly move closer together over time. This means that the rate of matching will be back to normal after the third window, and subsequently depend on the number of exchanged packets in each window.

We analyzed 239 windows from the data stream in this experiment. According to Figure 3.3, the total number of matched packets is 39786. Figure 3.8 illustrates the number of event pairs that represent the bound points of each hull in the Correlated sliding window approach. As shown in Figure 3.8, there are 12.7 points, on average, in the lower hull, and 11.8 points, on average, in the upper hull. These numbers become 8.5 points in the lower hull and 6.4 points in the upper hull for the Fully Incremental approach, as shown in Figure 3.9. According to this figure, there are 226 accurate pairs, and therefore there were 226 synchronization updates during this experiment. By contrast, the Correlated approach performs synchronization at most once per time window, that is, 239 for this experiment. When compared to the total number of matched pairs, which is 39786, both these numbers, 226 and 239, are much lower, and so incur a low cost. Figure 3.10 illustrates the comparison between the number of pairs in the two hulls in the Correlated and Fully Incremental approaches. As discussed earlier, the number of pairs is minimal in the Fully Incremental approach.

Figure 3.11 shows the accurate packet rate during the trace. It shows that at the start of tracing around 30 percent of matched packets are accurate packets and able to improve the synchronization since initial synchronizations is performed with low precision and the L_{max} and L_{min} drifts are away from each other. This difference decreases and the synchronization accuracy improves while receiving the accurate packets. Consequently, the chances of receiving a packet placed between L_{max} and L_{min} diminish with time.

Figure 3.12 illustrates this concept by comparing the number of received accurate packets versus the number of windows in the Correlated approach. At the start of tracing, the rate of accuracy improvement in the Fully Incremental approach is superior to the window based Correlated approach. After some time, this rate grows slower. For instance, there are 217 received accurate packets until 654 second into the trace, which is comparable to the number of time windows. However, the accurate packet flow reduces after this point and there are

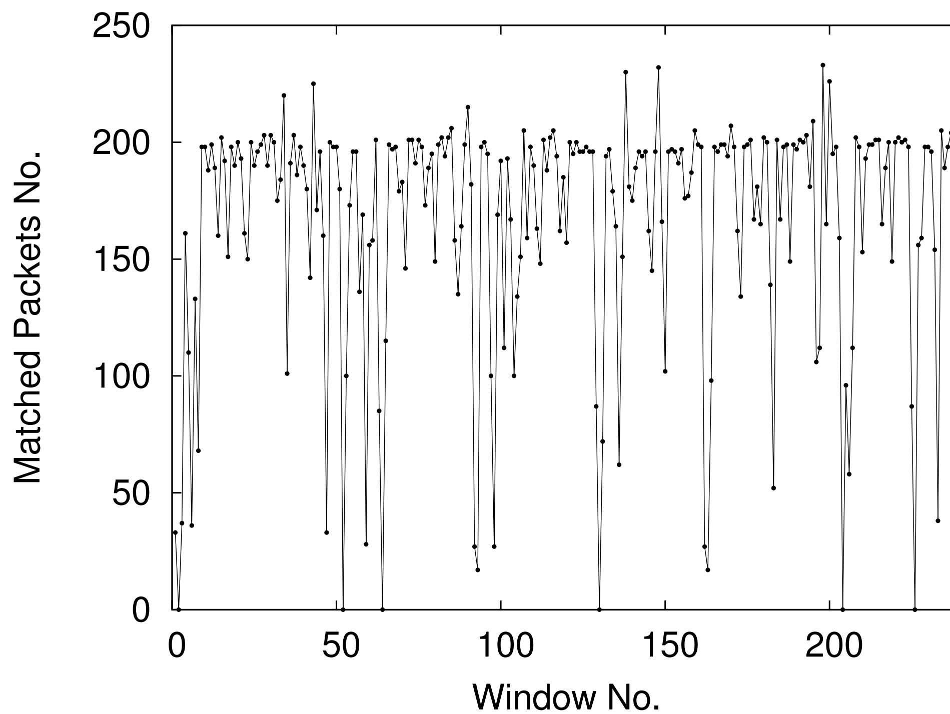


Figure 3.7 The number of matched packets in each window.

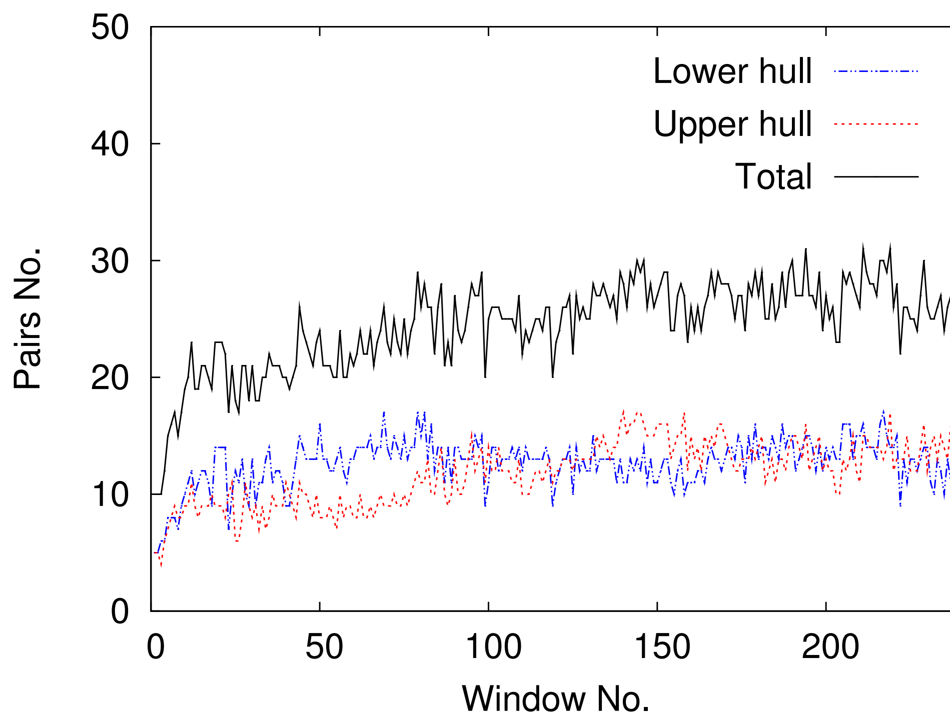


Figure 3.8 The number of pairs in Convex-Hull in each window (Correlated approach).

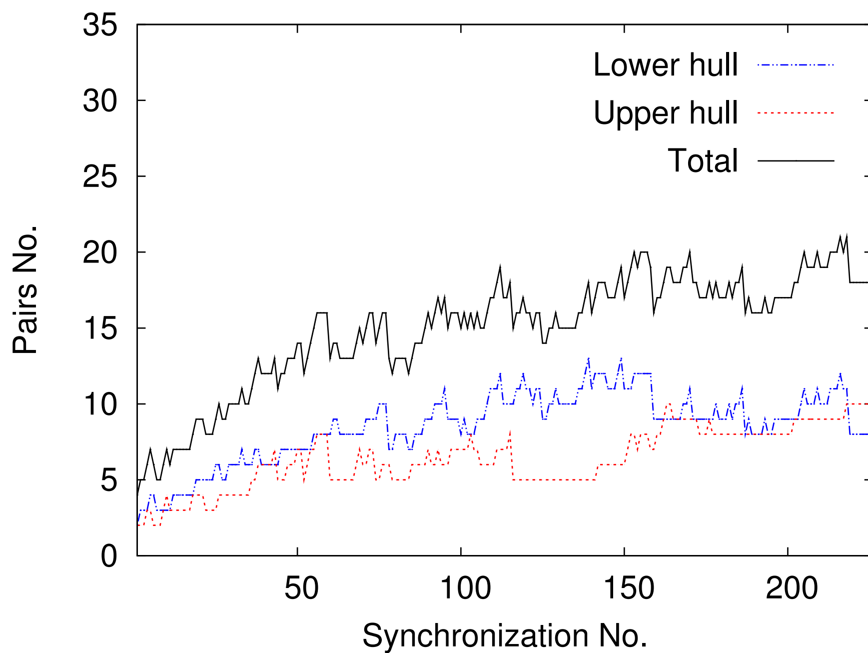


Figure 3.9 The number of pairs in Convex-Hull in each synchronization (Fully Incremental approach).

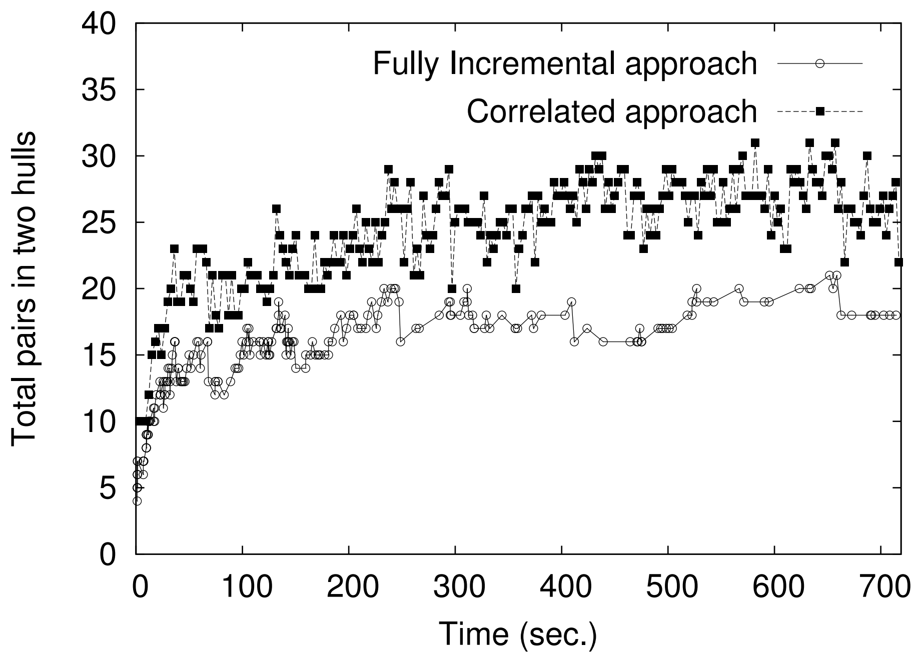


Figure 3.10 Comparison of total pairs in Convex-Hull.

nine subsequent synchronizations performed by the Fully Incremental approach versus 21 synchronizations performed in the Correlated approach.

3.7.3 Accuracy and Cost

Figure 3.13 illustrates the synchronization results of the Fully Incremental vs. the time window-based approaches. As expected, the Independent approach has the worst accuracy, because it ignores the pairs of interest in previous windows. Moreover, the Replace approach changes the results when it is able to improve the total accuracy in a window relative to that of the previous window. Nonetheless, it still ignores the accurate pairs in previous windows. Therefore, as shown in Figure 3.13, it cannot improve on the total accuracy after the second window, and it keeps that accuracy until the end of the experiment. Consequently, we will focus on the other two approaches. As expected, the Fully Incremental approach performs synchronization as soon as it finds an accurate pair. The first synchronization of all window-based approaches was performed at time 3.42. By contrast, the Fully Incremental approach synchronizes these two traces 22 times before time 3.42. This illustrates why the Fully Incremental approach has the highest accuracy at all times, since it does not postpone the calculation of the synchronization factors. We now examine the synchronization cost of the two approaches in further detail. Formula 3.18 represents the synchronization cost of the window-based approaches ($Cost_w$) for each window :

$$
\begin{aligned}
& l = window_size \\
& R = Read_events(l) \\
& M = Matching(R) \\
& P = Perform_analysis(M) \\
& F = Finalize_sync.(P) \\
& \quad = Cal(L_{max}) + Cal(L_{min}) + Cal(\alpha, \beta)) \\
& \\
& Cost_w = Cost(R) + Cost(M) + Cost(P) + Cost(F)
\end{aligned}
\tag{3.18}
$$

The number of events in each window is different, and depends on the rate of event occurrence. The reading time depends on the number of events (Cost(R)). The number of matched and unmatched pairs is different in each window (Cost(M)), but the matching time does not depend on the number of events in the hash table, since it has $O(1)$ complexity. As mentioned, with the *Convex-Hull* algorithm, we perform vertex removal when adding the latest received pairs, and possibly replace some of the related bound pairs. Thus, there is a cost (Cost(P)) in dealing with new pairs, when the Matching module sends a new pair to the Analysis module, in the window-based approaches. The cost of the finalize_sync is to find

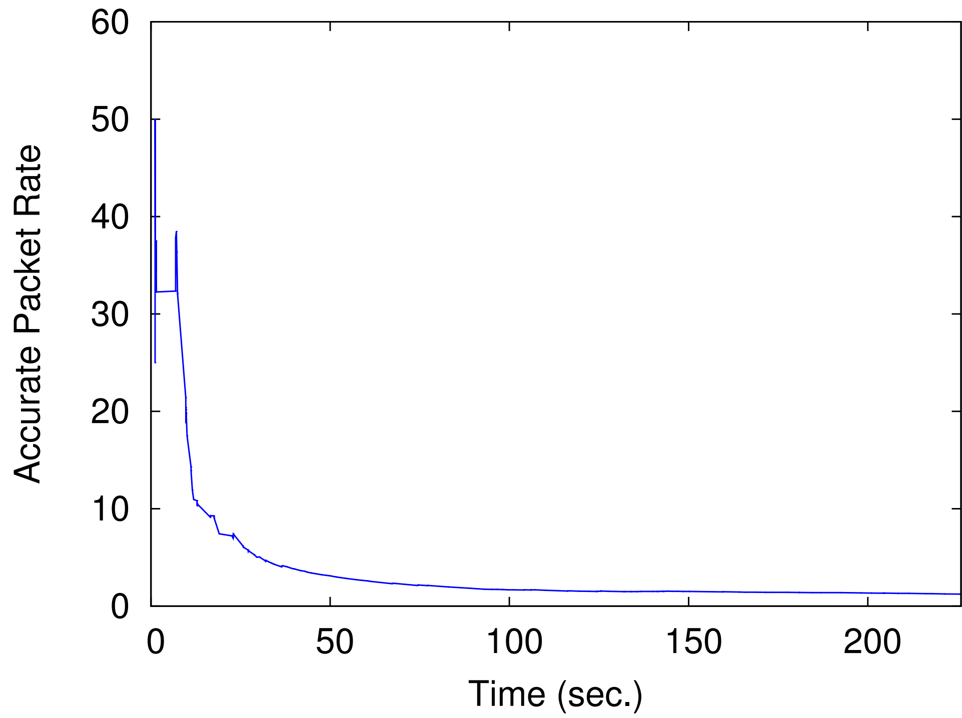


Figure 3.11 Accurate packet rate

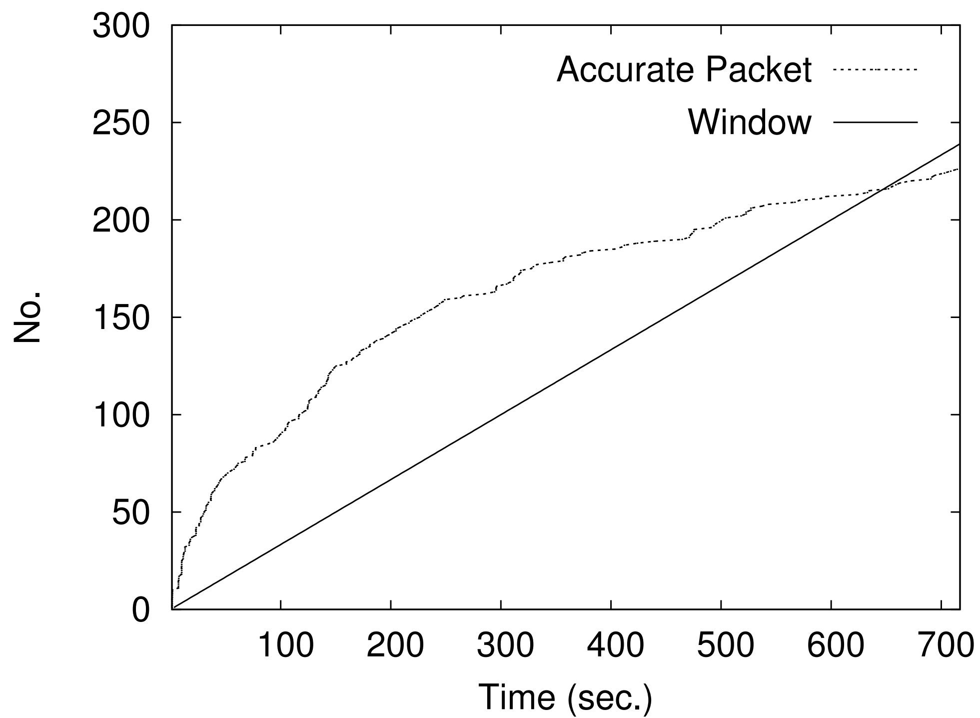


Figure 3.12 Accurate packet distribution vs. time window enhancement

L_{max} and L_{min}, and compute the synchronization factors.

Formula 3.19 represents the synchronization cost for the Fully Incremental approach ($Cost_f$) :

$$\begin{aligned}
R &= Read_events() \\
M &= Matching(R) \\
C &= Check_accurate_pair(M) \\
F &= Finalize_sync.(P) \\
&= Cal(L_{max})|Cal(L_{min}) + Cal(\alpha, \beta))
\end{aligned} \tag{3.19}$$

$$Cost_f = Cost(R) + Cost(M) + Cost(C) + Cost(F)$$

As mentioned in Formula 3.19, the costs of reading and matching events (Cost(R) and Cost(M)) in the Fully Incremental approach are same as for the window-based approaches (Formula 3.18).

In the Fully Incremental approach, there is no comparison algorithm for the new pair and the existing pairs in the hull bound lists. For most points, the procedure merely involves checking the location of the pair and L_{max} or L_{min}, which costs less than the cost comparison required in the window-based approaches. Since we do not save inaccurate pairs that are located higher than L_{max} or lower than L_{min}, we save not only in terms of memory, but also in terms of time, as just explained, since inaccurate pairs do not improve synchronization accuracy (Cost(C)).

Since only L_{max} or L_{min} is changed when an accurate pair is received, the cost of finalizing the synchronization algorithm is lower as well (Cost(F)). Therefore, the total cost of the Fully Incremental approach is lower than the total cost of the window-based approaches.

Figure 3.14 illustrates the zoom on the accuracy dimension from $1.2e^{-06}$ to $1.9e^{-06}$ and the trace time dimension from 120 to 170 second in Figure 3.13. It shows that the Fully Incremental approach always yields the lowest (best) accuracy bound, while the Correlated approach achieves the best accuracy as well, but with some delay, at the end of each window.

For example, $1.35e^{-06}$ is the accuracy reached by Correlated approach at 153 second, while Fully Incremental approach had already achieved this accuracy at 150.69 second of the trace. This illustrates the first disadvantage of the Correlated approach, not synchronizing the traces until the end of windows. This delay (2.31 seconds for this short trace) causes problems for critical applications. The worst case happens when accurate packet is found at the start of window, in which the delay is approximately equal to the length of window.

Moreover, there is no improvement in two windows between 153 and 159 second. Thus, the Correlated approach recomputes needlessly the synchronization parameters. The Fully

Incremental approach does not recompute anything until the next accurate packet. This illustrates the second disadvantage of the Correlated approach running at the end of each window, even when there is no chance to improve the synchronization accuracy.

Four techniques for online synchronization have been discussed and tested in this section. The interesting feature of the Fully Incremental approach is that it continuously updates the time synchronization factors with the best available data. Yet, each step involved : matching the send-receive pair, adding a point to the *Convex-Hull*, and updating the slope of the min, max, and median lines, takes a constant time, on average. Furthermore, for a large proportion of the new points, we can quickly determine that no update is necessary.

3.7.4 Delay and Packet loss effects on the Fully Incremental approach

The presented Fully Incremental algorithm for online clock synchronization is robust even in the presence of network delay uncertainties and internal kernel delays. It also guarantees accuracy improvements as soon as transmission latency improves.

To test the effect of packet loss, we traced a network containing two computers, and generated TCP/IP packet exchange traffic. The target duration of the traces is 12 minutes. Initially, we used a traffic with less than 10 percent packet loss. Then, the experiments were repeated with increasing packet loss. This is achieved by using iptables and the statistic module (iptables -A INPUT -m statistic –mode random –probability 0.1 -j DROP). Since the packet loss grows in the different experiments, the duration of the traced applications increases accordingly. Indeed, the TCP/IP protocol retransmits lost packets when it does not receive acknowledgments. Although this retransmission causes the overall throughput of the connection to drop, the number of sent/received packets increases. Therefore, we choose the first 270 seconds after 30 seconds warm-up period of each trace for comparison purposes. When the proportion of lost packets rises, the number of matched packets and accurate packets diminish and consequently the accuracy slowly declines, as shown in Table 3.1.

Table 3.1 The packet loss affection on Fully Incremental approach

Packet loss	Trace duration (sec.)	Matched Packet No.	Accurate Packet No.	Accuracy
0.1	270	5797	105	7.535×10^{-7}
0.2	270	4027	103	6.736×10^{-7}
0.3	270	2063	87	5.042×10^{-7}

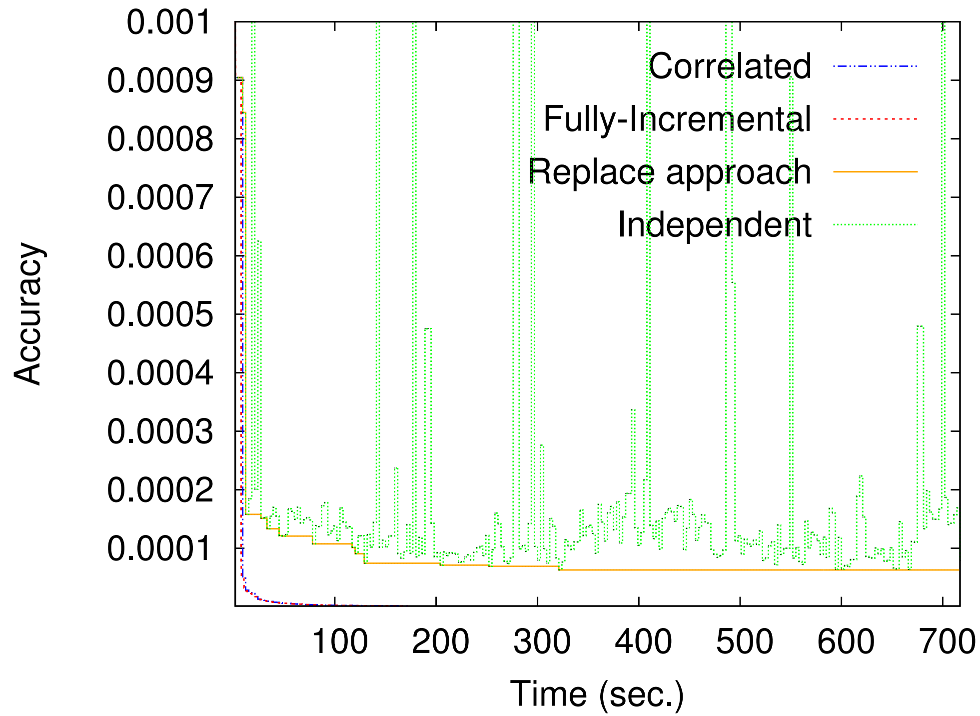


Figure 3.13 Comparison of time synchronization approaches in streaming mode.

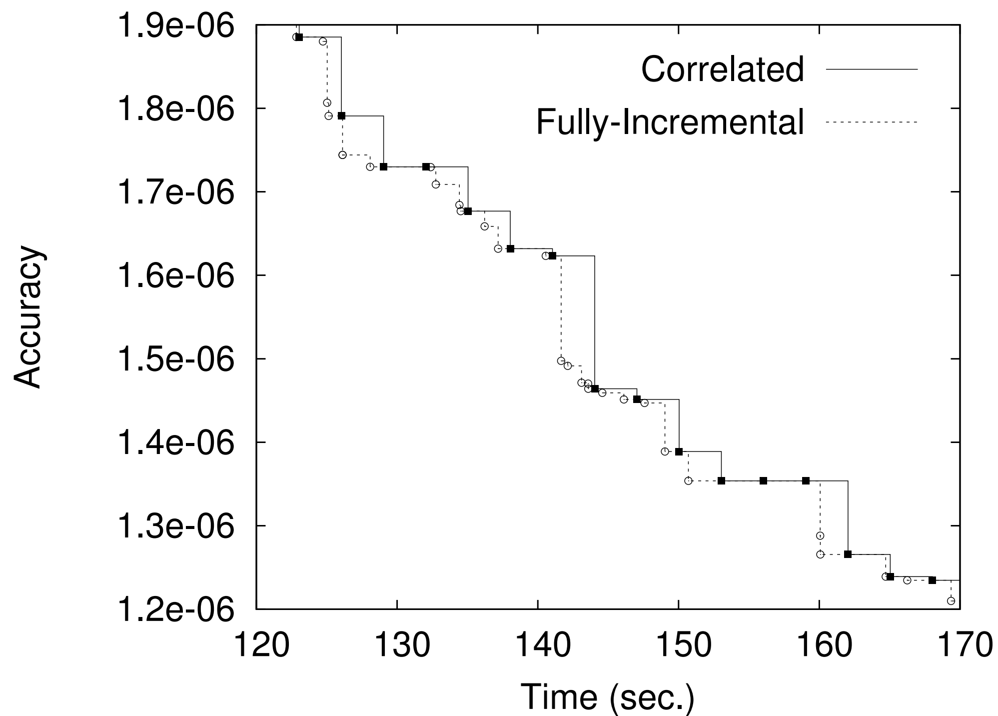


Figure 3.14 Zoom on the accuracy dimension of Figure 3.13 from $1.2e^{-06}$ to $1.9e^{-06}$ and the trace time dimension from 120 to 170 second.

3.8 Conclusion

In this paper, we have presented a framework for online time synchronization, which applies to data streaming. The most notable feature of streaming data is the speed of the stream flow, which makes it impractical to scan the data stream more than once. Buffering the data stream for a long time is not practical either. The proposed model is not only efficient in terms of both time and memory, it is also scalable and yet maintains and improves its accuracy over time. The novel approach proposed is of particular interest for two reasons : it is fully incremental, with $O(1)$ complexity on average per event in the trace, and it does not add latency to the time synchronization computation. Moreover, the algorithm can very quickly eliminate inaccurate packets without loss of accuracy or added latency. It only needs to examine in detail, a very small fraction of the matched packets, and eventually store them as *Convex-Hull* vertices. This makes the proposed algorithm ideally suited for the high performance live analysis of detailed distributed system traces in clusters and clouds.

We also introduced several window-based approaches in this paper. The experimental results demonstrated that our proposed Fully Incremental approach is more accurate and generates no latency for synchronization computation. In future work, the new incremental algorithm will be studied in the context of large scale clusters, and extended for finding the optimal time reference node and synchronization spanning tree.

CHAPTER 4

Paper 2 : Reference Node Selection in Dynamic Tree

Masoume Jabbarifar and Michel Dagenais

4.1 Abstract

Several dynamic network tools require an efficient incremental algorithm to calculate hop counts and choose a central point for the network. For example, in a dynamic network, a time Reference Node is needed to synchronize all the nodes. However, computer connections to the network and disconnections from it frequently occur in a dynamic network, and this affects all network activities. The model proposed in this paper improves the performance of Reference Node maintenance live mode in a dynamic network, and the new method investigates only the altered path with respect to the Reference Node once an alteration has occurred in the network spanning tree. This method provides an efficient way to find and maintain a Reference Node incrementally in an average time complexity of *O(log n)*, where n is the total number of nodes in the network.

4.2 Introduction

In the past few years, intensive research efforts have been devoted to dynamic graphs and forests where edges are added or removed and weights change [67, 103], and a particular and ongoing focus of these efforts is incremental dynamic tree maintenance. As well, various data structures have been presented to improve dynamic maintenance performance [19, 51, 52]. The purpose of this paper is to present a method to incrementally update Reference Nodes (RN) in a dynamic forest. As will be demonstrated, RN should be the nodes with a minimum hop count between them and all the other connected nodes in the tree.

Although we are using this graph-based algorithm to solve a dynamic network synchronization problem, it can also be used in other applications, such as electronics, medical science, etc. In a dynamic distributed system, where communication links appear and disappear, analysis of the applications critically depends on time synchronization [40]. Our objective in this work is not only to provide the best time synchronization accuracy, but also to improve synchronization performance.

Recently, several time synchronization algorithms have been presented for wired and wireless networks [45, 58, 85, 101]. The general idea behind these algorithms is to maintain a

spanning tree (ST) of all the nodes in the network, updating it at each network change, and computing pairwise synchronization for all the ST edges. Once all the clock drifts and offsets have been obtained, a node must be found to act as a time reference [59]. For any node in ST, the shortest paths to every other node are computed. The best time reference node (RN) is the node with the shortest paths to all the nodes [22]. Eventually, all the nodes in the network synchronize their time with the RN through those paths. The position of RN is critical to decreasing the total time conversion error through all paths. Since it takes a time of $O(n^2)$ to find the RN with a naive approach, most applications favor synchronization based on a fixed RN chosen in the network in advance [56]. However, a static RN has many disadvantages. The main problem is that the number of node connections may change, leading to fewer hops for time conversion in a path in the dynamic network over time. Since having fewer hops improves synchronization accuracy, it is beneficial to choose a new RN [58]. The second problem is RN robustness. The synchronization algorithm should be robust to node failures [101] ; however, a static RN is a single point of failure.

The work in this paper is primarily motivated by the online *LTTng* (Linux Trace Toolkit Next Generation) time synchronization project [30]. *LTTng* is capable of handling huge traces of several gigabytes or more. However, a new architecture is required to do so while at the same time allowing traces to be collected from multiple systems and embedded devices, for both online and a posteriori offline analysis and viewing. Moreover, *LTTng* users expect to see the analysis output in real time, in order to be able to diagnose problems more easily. Therefore, *LTTng* should be able to visualize traces from several distributed systems, on a common reference time base, and in streaming mode.

In this paper, we propose an efficient algorithm to identify dynamic RN with $O(\log n)$ time complexity in a dynamic forest. The paper begins with a study of the problem and a review of related work in the 4.3 section. The proposed data structure and methodology are detailed in the 4.4 and 4.5 Sections, and the evaluation of the complexity of the proposed method is explained in the 4.6 Section. Noteworthy results, including performance analysis, are presented in the 4.7 Section. Finally, we present our concluding remarks and discuss future work in the 4.8 Section.

4.3 Related Work

In this section, we discuss the most interesting synchronization algorithms in wired and wireless networks, and see how they use time RN.

Greunen et al. [101] proposed their Lightweight Time Synchronization for Sensor Networks, in which nodes are placed uniformly and at random within a 2-dimensional area. The

network contains a single RN which keeps accurate time in an ST of nodes. For better synchronization precision, the RN is located in the area center. In dynamic networks, nodes are mobile, and can join and leave the network. Thus, efficient dynamic calculation of RN (as proposed in this paper) in ST can increase precision. Ganeriwal et al. [45] propose a protocol to synchronize nodes in sensor networks called TPSN, which first creates an ST of the network and then performs pairwise synchronization along the edges. This algorithm does not handle dynamic topology changes, however, and the RN is static.

Many use cases for dynamic RN are possible in wired networks as well. There is an interesting one in tracing software [29]. Tracing is similar to logging, and consists in recording events that occur in a system, usually more detailed lower-level events as compared to logging) [30]. The tracer software user expects to see the analysis output in real time, in order to diagnose problems live. Consequently, the software should be capable of visualizing traces from several distributed systems on a common reference time base.

In a computer cluster, multiple nodes produce separate trace streams independently, and there are timestamps associated with each event [32]. Synchronization starts online, as soon as two nodes begin to exchange messages. This connection creates an edge between those two vertices and establishes the first tree. The tree then grows, or new trees are established, by adding new nodes and new connections. Synchronization accuracy is the weight of all the edges in the dynamic network. Obviously, when a connection is added or disappears, the dynamic graph changes and must be checked to determine whether or not the change affects the Minimum Spanning Tree (MST) [98]. Clearly, any change that occurs may affect the choice of time RN.

Many interesting algorithms with a $O(\log n)$ running time have been proposed to maintain a dynamic MST [19, 51]. The algorithm proposed in this paper builds on these efficient dynamic MST algorithms, with a view to improving the time synchronization algorithms for the analysis of streaming mode traces recorded on distributed systems.

4.4 Data Structure

Let $G=(V,E)$ be an undirected graph containing a set V of vertices and a set E of edges. An edge ($e = \{\upsilon, \nu\}$) is related to two vertices. Let τ be an ST for G. In streaming mode, many separate trees ($\tau = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$) may be joined together over time.

As more and more messages are sent between the nodes, clock synchronization can be computed between the newly communicating nodes, and edges are dynamically added to the communication graph. The weight of each edge can change over time (e.g. reduced error, if a better synchronization is achieved with more messages). The ST forest computed from the

communication graph can only see trees being merged as time progresses and more messages are sent. It is possible, however, that edges will be added to the ST in the forest, or removed from them, as new edges become favored over others.

Definition 1 : DescendantSize is a factor that illustrates whether or not a vertex is a better balance point than the existing *RN*.

To find an optimal dynamic *RN*, we compute a *DescendantSize* attribute for each vertex. This attribute shows the number of children there are for each vertex, not in the "parent" direction to the *RN*. When a vertex ν is added to a tree τ, its *DescendantSize* is initialized to 1, and this value is propagated along the parent path.

Six types of operations are defined on a tree. All the operations, except *reverse(v)* and *treeId(**vertex** v)*,take time $O(1)$.

- *parent(**vertex** v)* : Return the parent of v. If v is the root of its tree, it returns a null value.
- *reverse(**vertex** v)* : Reverse the direction of the tree, making the child the parent. It is proportional to the length of the tree path from v to the previous root. The number of operations depends on the depth of that branch. Since the average depth of a tree is log n, the average complexity of this function is $O(log\ n)$.
- *DescendantSize(**vertex** v)* : Return the DescendantSize of the vertex.
- *update_DescendantSize(**vertex** v, **int** x)* : Add x to the current value of vertex DescendantSize.
- *treeId(**vertex** v)* : Since there are many separate trees in a forest, this function returns the ID of the tree to which the vertex v belongs. When two trees are joined together, we retain the lowest of the two IDs as the new tree ID.
- *treeSize(**vertex** v)* : Return the number of vertices in the tree to which the vertex v belongs.
- *referenceNode(**treeId** id)* : Return the *RN* in the tree *id*.

The *RN* is the topmost node in our tree. Each node in a tree has zero or more child nodes. All directions in the tree are defined relative to the *RN*. When the *RN* changes in a dynamic tree, all the directions in the altered path(s) change towards the new *RN*.

4.5 Methodology

4.5.1 Reference Node

The RN is a vertex which has a strategic role in a graph. For example, in a distributed system, the RN can be the time reference for all the other nodes for clock synchronization purposes. In such a context, the best RN could be defined as having the minimal sum of time synchronization errors between itself and every other node. The synchronization error between two connected node is the edge weight, and the path sum of weights for indirectly connected nodes.

Lemma 1. *The best RN has the minimum hop count to all the nodes in a tree.*

Let τ be a tree in the forest with $< \upsilon_1, ..., \upsilon_n >$ vertices. Further, $P_{\upsilon_i} < \upsilon_i, \upsilon_{i+1}, ..., RN >$ is the set of vertices met by υ_i on the path reaching the RN.

The Formula 4.1 illustrates a total tree cost that corresponds to the summation of all the vertex weights with respect to the RN. The weight of each vertex to reach the RN corresponds to the summation of P_{υ_i} edge weights.

$$Total\ \ Cost_\tau = \sum_{i=1}^{n} \sum_{j=1}^{l}\ \text{weight}(edge_j \text{ on } P_{\upsilon_i}) \tag{4.1}$$

Thus, for each edge, multiply its weight by the number of children opposite RN, as shown in the formula 4.2, as follows :

$$Total\ \ Cost_\tau = \sum_{i=1}^{n} DescendantSize(\upsilon_i) \times w_{e_i} \tag{4.2}$$

$$Cost = \begin{cases} DescendantSize(\upsilon_l) \times w_e : RN \text{ is on the } \upsilon_k \text{ side} \\ DescendantSize(\upsilon_k) \times w_e : RN \text{ is on the } \upsilon_l \text{ side} \end{cases} \tag{4.3}$$

The contribution of the other edges does not change, whether the RN is υ_k or υ_l. Moreover, they change independently of w_e's contribution when they are further away from e. Therefore, the optimal choice is to place the RN on the side with the larger DescendantSize, irrespective of w_e. This property will be used to incrementally compute the best RN. □

Cost is a concept applicable to different use cases, such as power usage, error, and so on. As discussed, a prime example of the use of RN is in distributed system time synchronization.

4.5.2 Independent trees

In a dynamic environment, new vertices and edges appear (for example, as new computers join and communicate with existing computers) and when they do the weight of edges can

change (a new communication can provide better synchronization accuracy, or the accuracy can be reevaluated when no message has been received from a computer for a long time). Vertices and edges are never removed from the communication graph, since inactive nodes may simply be assigned an updated weight representative of their long period of inactivity. Similarly, the associated ST will only see vertices being added. However, edges can either be added to the ST, connecting and merging two previously independent trees, or replaced. An edge is replaced when a better edge connects two vertices that are in the same tree already, causing the minimum ST algorithm to remove an edge that is less good.

Initially, there is no edge and each vertex forms an independent tree $(\tau_1, \tau_2, \tau_3, ..., \tau_n)$, and the single vertex is trivially the RN for its tree. Let us examine the operations required to update the dynamic RN in each independent tree when the edges are added or replaced.

4.5.3 Adding a single vertex and edge

As Figure 4.1 illustrates, this situation arises when a new vertex υ is connected to the current tree τ_i with a new edge. This occurs frequently when the algorithm starts. In Figure 4.1, the number inside each vertex shows the *DescendantSize*. We denote the *DescendantSize* attribute of each node as ξ. Since a new single vertex does not have a child, its *DescendantSize* is 1. The *DescendantSize* of parent(υ) is increased by 1 (line 12 Algorithm 4), and this new increase ($\Delta\xi = 1$) is propagated along the path from the parent of υ to the RN. The best candidate for the new RN is the latest vertex in the propagation path to the RN. $\xi(\upsilon_n)$ is the number of children that vertex υ_n has. It compares its *DescendantSize* with RN (χ) (line 8 Algorithm 4) to determine which of them will be the next RN. The number of operations required is proportional to the length of the path between the new vertex and the RN.

4.5.4 Replacing an edge in a tree

To maintain an ST structure when an edge is removed from a tree and replaced by another, one subtree is disconnected and reconnected elsewhere, possibly through another vertex. This tree reorganization may change the whole balance. Since the RN is a node with a minimum number of hop counts to all nodes, it should be recomputed for the modified tree. We consider two possibilities, depending on the location of the cut/insertion in the tree :

As Figure 4.2a illustrates, in the first situation, the $cut(s_1, o_1)$ and $add(o_n, e_1)$ operations occur on different sides of the RN. We introduce three paths along which the *DescendantSize* of all vertices has to be updated :

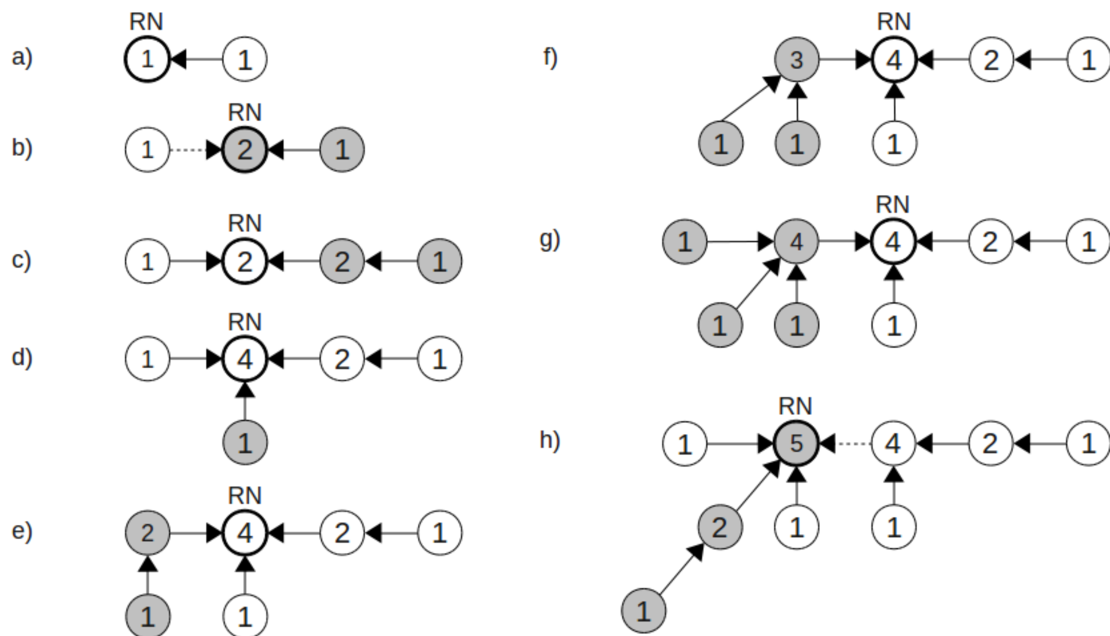


Figure 4.1 The DescendantSize operation in insertion mode with no tree cycle

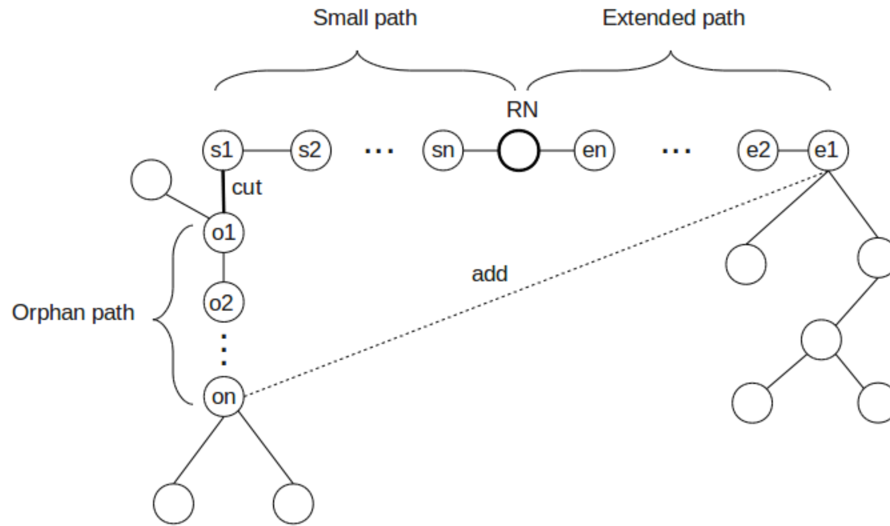
---

ALGORITHM 4: Update_RN_Insertion()

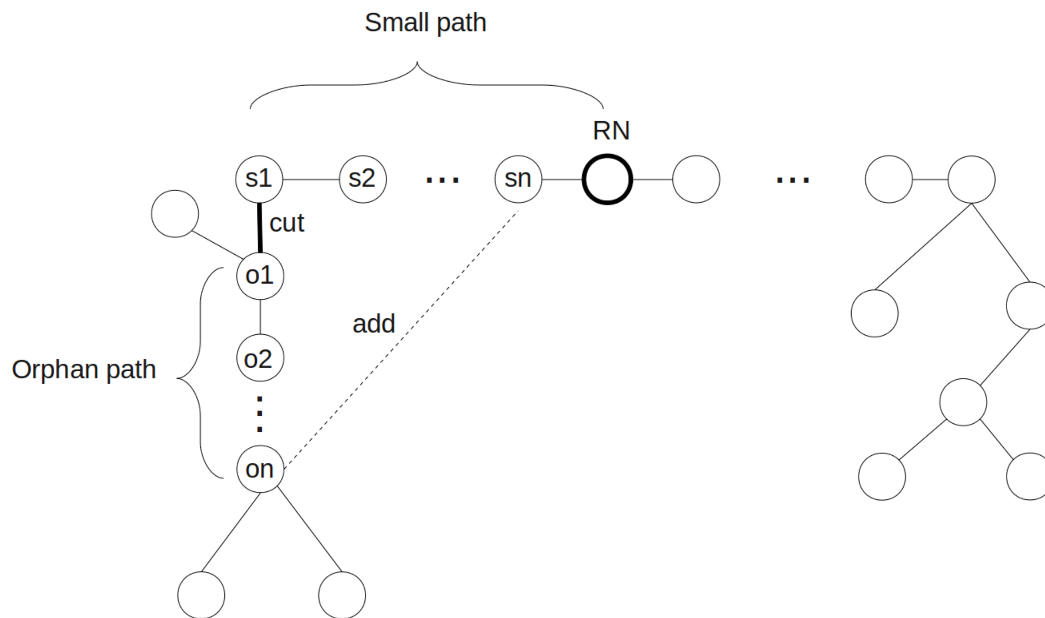
1: χ : an RN
2: υ_1 : a new vertex is added to an existing tree
3: **Begin**
4: propagationPath=$\{\upsilon_1$, ..., υ_n, $\chi\}$
5: **for each** $\upsilon \in$ propagationPath **do**
6:     **if** $\upsilon = \chi$ **then**
7:         DescendantSize(χ)= treeSize(υ_n)-DescendantSize(υ_n)
8:         **if** $DescendantSize(\upsilon_n) > DescendantSize(\chi)$ **then**
9:             The RN is υ_n
10:         **end if**
11:     **else**
12:         update_DescendantSize(υ, DescendantSize(υ)+1)
13:     **end if**
14: **end for**
15: **End**

ALGORITHM 5: Update_RN_Cycle()

```
1:  χ : RN
2:  modification : cut(s_1, o_1) and add(o_n, e_1)
3:  Begin
4:  smallPath={s_1, s_2, ..., s_n, χ}
5:  orphanPath={o_1, o_2, ..., o_n}
6:  extendedPath={e_1, e_2, ..., e_n, χ}
7:  updateDescendantSize(orphanPath)
8:  updateDescendantSize(smallPath)
9:  if e_1 was visited in updateDescendantSize(smallPath) then
10:     tree is still balanced
11: else
12:     if DescendantSize(χ) > treeSize(χ)/2 then
13:         χ is still a RN
14:         updateDescendantSize(extendedPath)
15:     else
16:         while DescendantSize(e_i ∈ extendedPath) < treeSize(χ)/2 do
17:             updateDescendantSize(extendedPath)
18:         end while
19:         e_i = a new RN
20:         reverse(e_i)
21:         betweenRNsPath={e_{i+1}, e_{i+2}, ..., e_n}
22:         updateDescendantSize(betweenRNsPath)
23:     end if
24: end if
25: End
```



(a) add and cut occur on different sides of an RN



(b) add and cut occur on the same side

Figure 4.2 The position of add() and cut()

$$\begin{aligned}&\text{Orphan path} :< o_2, ..., o_{n-1} >\\&\text{small path} :< s_1, s_2, ..., s_n >\\&\text{Extended path} :< e_1, e_2, ..., e_n >\end{aligned} \tag{4.4}$$

Lemma 2. *The candidate RN in (i) below must be one of the vertices in the path* $< e_1, e_2, ..., e_n, RN >$*. In (ii) below, the total number of vertices is unchanged on the side where both the cut and the insertion occurred. Thus, the DescendantSize of the children of the previous RN remain the same and the algorithm keeps that RN.*

(i) Assume that the RN is connected to the n paths : $< p_1, ..., p_n >$. Moreover, there is a minimum number of hops from the RN to all the other nodes in the tree. The candidate list of RN for the next operation is all vertices in all paths close to $RN < v_{p_1}, ..., v_{p_n} >$. The *cut()* operation causes the vertex v_{p_l}, at the end of a path p_l, to have no chance of becoming the *RN*. Indeed, its *DescendantSize* has been reduced and the RN must have the maximum *DescendantSize* in the tree. However, the *add*(o_n, e_1) operation causes m vertices from path p_l to join path p_k and increase the *DescendantSize* of all the vertices in this path. Thus, one of the vertices in $p_k = \{e_1, e_2, ..., e_n\}$, where $\xi(\chi) > treeSize(\chi)/2$, becomes *RN*.

(ii) As Figure 4.3 illustrates, in the special case where the RN (χ) itself has many children, the RN does not change. To calculate how many children the RN has, we use the following formula (Eq. 4.5) :

$$\begin{aligned}&reverse(e_1)\\&\triangle = \underbrace{\xi(o_1)}_{\text{cut impact}}\\&\xi(\chi) = \overbrace{treeSize(\chi) - [\underbrace{\xi(parent(\chi))}_{e_n} + \triangle]}^{\text{the new DescendantSize for the previous } RN}\\&RN(\tau_i) = \begin{cases}\chi & \text{if } \xi(\chi) > \underbrace{treeSize(\chi)/2}_{\text{tree balance value}}\\ \text{search in extendedPath} & \text{otherwise}\end{cases}\end{aligned} \tag{4.5}$$

Since *parent*(χ) is e_n, to obtain its *DescendantSize*, we have to run the *reverse*(e_1) operation, since we do not know who e_n is. If the *RN DescendantSize* is greater than half the *DescendantSize* of its updated child, the previous *RN* can remain as it is (lines 12-15 Algorithm 5). □

Orphan path : *path* $< o_1, o_2, ..., o_n >$: this path was structured for the previous *RN*. Therefore, not only should the *DescendantSize* attribute of each node be updated, but the direction of the edges should be oriented towards the new probable *RN* as well. Since *edge*(s_1, o_1) has been cut, we are reversing the parental structure of this path. Previously,

the *DescendantSize* of o_1 was the sum of $DescendantSize(o_2)$ and how many children it has. Thus, the update algorithm in this step is (line 7 Algorithm 5) :

$$\begin{aligned}
&reverse(o_n) \\
&\quad \underbrace{\xi(o_1)}_{\text{new DescendantSize}} = \underbrace{\xi(o_1)}_{\text{old DescendantSize}} - \underbrace{\xi(parent(o_1))}_{o_2} \\
&\forall x \in path < o_2, ..., o_{n-1} > \\
&\quad \xi(x) = \xi(x) - \underbrace{\xi(parent(x))}_{\text{new parent}} + \underbrace{\xi(x-1))}_{\text{new child or previous parent}} \\
&\xi(o_n) = \underbrace{\xi(o_n)}_{\text{old DescendantSize}} + \underbrace{\xi(o_{n-1})}_{\text{new child or previous parent}}
\end{aligned} \tag{4.6}$$

Small path : $path < s_1, s_2, ..., s_n >$: the direction of this path has not changed, and so updating *DescendantSize* is the only step that has to be taken. Previously, the vertex o_1 was one of the children of s_1. In the new tree, there is no connection between them, and so the updated algorithm for the small path is (line 8 Algorithm 5) :

$$\begin{aligned}
&\triangle = \underbrace{\xi(o_1)}_{\text{cut impact}} \\
&\forall x \in path < s_1, s_2, ..., s_n > \\
&\quad \xi(x) = \xi(x) - \triangle
\end{aligned} \tag{4.7}$$

Extended path : $path < e_1, e_2, ..., e_n >$: we update the *DescendantSize* of the vertices in this path and check a condition for the new *RN* (lines 16-18 Algorithm 5). If we cannot find the *RN* in the previous path, the new *RN* is chosen from the vertices in this path. If the *DescendantSize* of a vertex e_i is greater than the $treeSize(\chi)/2$, which is the half the number of vertices in the tree, e_i will be the new *RN* (line 19 Algorithm 5). Updating *DescendantSize* follows the 4.8 formula.

$$\begin{aligned}
&reverse(\chi) \\
&\triangle = \underbrace{\xi(o_n)}_{\text{new children for the extended path}} \\
&\forall x \in path < e_1, e_2, ..., e_i > \\
&\quad \xi(x) = \xi(x) + \triangle
\end{aligned} \tag{4.8}$$

If we find the new reference (e_i) in the extended path, we have to reverse the direction of the path between the previous *RN* and the new one, e_i, and update each node *DescendantSize* in $path < e_n, e_{n-1}, ..., e_{i+1} >$ (lines 20-22 Algorithm 5) as follows :

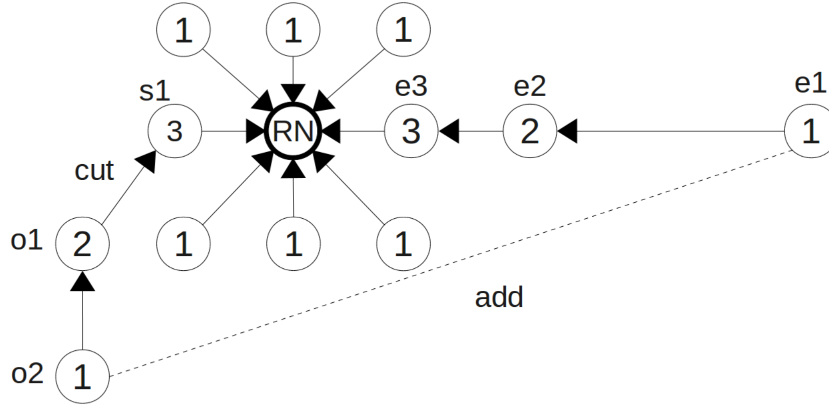


Figure 4.3 One way the previous reference node can remain an *RN*

$$
\begin{aligned}
& reverse(e_i) \\
& \xi(e_n) = \xi(e_n) - \underbrace{\xi(parent(e_n))}_{e_{n-1}} + \underbrace{\xi(\chi)}_{\text{calculated in small path update}} \\
& \forall x \in path < e_{n-1}, ..., e_{i+2} > \\
& \quad \xi(x) = \xi(x) - \xi(parent(x)) + \xi(x+1) \\
& \xi(e_{i+1}) = treeSize(\chi) - \underbrace{\xi(e_i)}_{\text{new } RN}
\end{aligned}
\tag{4.9}
$$

As Figure 4.2b illustrates, in the second situation, the $cut(s_1, o_2)$ and $add(o_n, e_1)$ operations are performed on the same side.

Lemma 3. *When the cut() and add() operations are performed on the same side of the RN, the RN does not change.*

All the changes occur on one side of the *RN*, and the tree size does not change. For example, vertex s_m is the new connection to vertices (n_1, n_2) in the small path. Previously, the children of $DescendantSize(s_m)$ were $[o_n, o_{n-1}, ..., o_1, s_1, s_2, ..., s_{m-1}]$. When we update the orphan path, the children of $DescendantSize(o_n)$ are $[o_1, o_2, ..., o_{n-1}]$. Then, when we update the small path, the children of $DescendantSize(s_m)$ are $[s_1, s_2, ..., s_{m-1}]$ plus the children of $DescendantSize(o_n)$. Thus, when we see the newly connected vertices (n_1, n_2) in the small path, we stop updating the *DescendantSize* and the previous *RN* remains the same. Also, there is no extended path in the second situation (lines 9-11 Algorithm 5). □

In the worst case, the orphan path needs to have its direction reversed and *DescendentSize* updated, the small path needs to have its *DescendentSize* updated, and the extended path needs to have its *DescendentSize* updated and checked for a new *RN* along the extended

path. The number of operations required is proportional to the length of each of these paths (orphan, small, and extended).

4.5.5 Inserting an edge between two independent trees

Every tree in the forest has its own *RN*. Thus, when an edge is inserted between two independent trees, the two trees become connected and merged, and four cases can arise :

1. $edge(RN_{\tau_s}, RN_{\tau_b})$: We denote the small and big tree as τ_s and τ_b respectively. In this case, the new edge is between two *RN*. So, RN_{τ_b} is still the *RN* (line 8 Algorithm 6).
2. $edge(RN_{\tau_s}, e_i)$: In this case, the vertex *RN* from the small tree is joined to a vertex e_i in the big tree. No computation is needed in the small tree. The new *RN* will be in the tree that has more vertices. The candidate *RN* list is in $path < e_{\tau_b}, ..., RN_{\tau_b} >$. If we denote the new *RN* as RN_n, we first have to update the DescendantSize in $path < e_1, e_2, ..., e_i, ..., RN_n >$, as follows :

$$\begin{aligned} &\xi(RN_{\tau_s}) = \underbrace{treeSize(RN_{\tau_s})}_{\text{small tree size}} \\ &\triangle = \xi(RN_{\tau_s}) \\ &\forall x \in path < e_1, e_2, ..., RN_n > \\ &\quad \xi(x) = \xi(x) + \triangle \end{aligned} \tag{4.10}$$

   and then reverse the direction of the path between the previous *RN* and RN_n, and then update $path < RN_{\tau_b}, ..., RN_n >$ (lines 16-25 Algorithm 6), as follows :

$$\begin{aligned} &reverse(RN_n) \\ &\xi(RN_n) = \xi(RN_n) - \xi(parent(RN_n)) + \xi(RN_{\tau_b}) \\ &\forall x \in path < RN_{\tau_b}, ..., RN_n > \\ &\quad \xi(x) = \xi(x) - \xi(parent(x)) + \xi(x+1)) \end{aligned} \tag{4.11}$$

3. $edge(m_i, RN_{\tau_b})$: In this case, a vertex m_i from the small tree joins the *RN* in the big tree, and RN_{τ_b} is still the *RN*. The only computation is in the small tree, where we have to run operation $reverse(m_i)$ and update *DescendantSize* along $path < RN_{\tau_s}, ...,$ $m_i, ..., m_2, m_1, m_{\tau_s} >$ in the small tree before joining it to the big tree. Thus, the update algorithm in the small tree is run as follows (lines 26-30 Algorithm 6) :

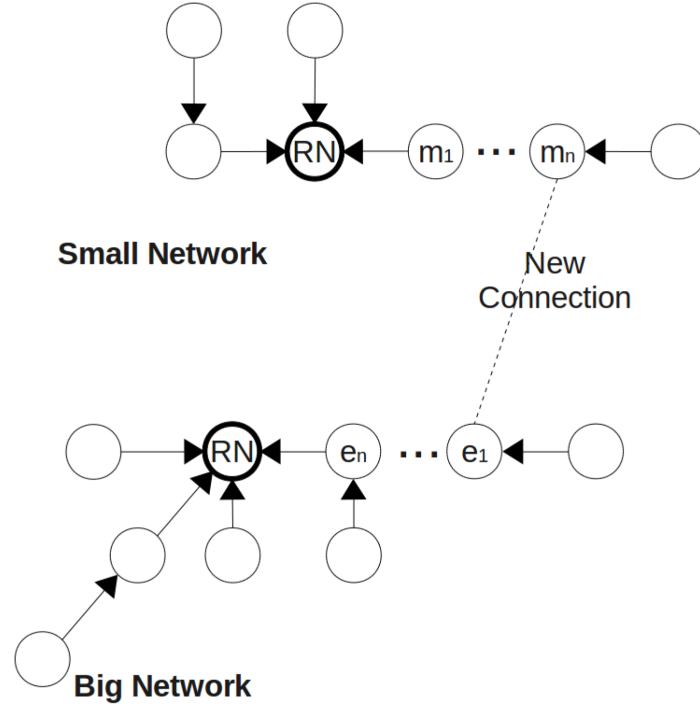


Figure 4.4 One case of joining two trees

$$
\begin{aligned}
&reverse(m_i) \\
&\xi(RN_{\tau_s}) = treeSize(RN_{\tau_s}) - \underbrace{\xi(parent(RN_{\tau_s}))}_{m_1} \\
&\forall x \in path < m_1, m_2, ..., m_{n-1} > \\
&\quad \xi(x) = \xi(x) - \xi(parent(x)) + \xi(x-1)) \\
&\xi(m_n) = \xi(m_n) + \xi(m_{n-1})
\end{aligned}
\tag{4.12}
$$

4. $edge(m_{\tau_s}, e_{\tau_b})$: In this case, a vertex from the small tree is joined to a vertex in the big tree, as shown in Figure 4.4. The candidate RN list is in $path < e_{\tau_b}, ..., RN_{\tau_b} >$. We have to update the *DescendantSize* in three paths at most. In the small tree, we run the update algorithm as mentioned in the third case. In the big tree, we run the update algorithm as mentioned in the second case.

Algorithm 6 illustrates the pseudocode for finding the RN when joining two trees. When they are joined, the ID of the new tree is set to the lowest tree ID of the two trees.

The complexity of joining two independent trees is similar to the complexity of the reconnection phase when replacing an edge in a tree. In the smaller tree, the path between the previous RN and the vertex with the new edge, like the orphan path, needs to have its direction reversed and its *DescendentSize* updated. Then, in the larger tree, the extended

ALGORITHM 6: Update_RN_Join()

1: ν : vertex
2: ξ : vertex
3: **Begin**
4: **if** treeSize(ν) < treeSize(ξ) **then**
5: $e_{\tau_s} = \nu$
6: $e_{\tau_b} = \xi$
7: **else**
8: $e_{\tau_s} = \xi$
9: $e_{\tau_b} = \nu$
10: **end if**
11: $RN_{\tau_s} = referenceNode[treeId(e_{\tau_s})]$
12: $RN_{\tau_b} = referenceNode[treeId(e_{\tau_b})]$
13: **if** $e_{\tau_b} = RN_{\tau_b}$ and $e_{\tau_s} = RN_{\tau_s}$ **then**
14: RN_{τ_b} is still RN
15: **else**
16: **if** $e_{\tau_b}! = RN_{\tau_b}$ and $e_{\tau_s} = RN_{\tau_s}$ **then**
17: $updatePath = \{e_{\tau_b}, ..., RN_{\tau_b}\}$
18: **while** $DescendantSize(e_i \in updatePath) < treeSize(RN_{\tau_b})/2$ **do**
19: updateDescendantSize(extendedPath)
20: **end while**
21: e_i = new RN
22: reverse(e_i)
23: betweenRNsPath={e_{i+1}, e_{i+2}, ..., e_n, RN_{τ_b}}
24: updateDescendantSize(betweenRNsPath)
25: **else**
26: **if** $e_{\tau_b} = RN_{\tau_b}$ and $e_{\tau_s}! = RN_{\tau_s}$ **then**
27: reverse(e_{τ_s})
28: $smalltreePath = \{RN_{\tau_s}, ..., e_{\tau_s}\}$
29: updateDescendantSize(smalltreePath)
30: **else**
31: reverse(e_{τ_s})
32: $smalltreePath = \{RN_{\tau_s}, ..., e_{\tau_s}\}$
33: updateDescendantSize(smalltreePath)
34: $bigtreePath = \{e_{\tau_b}, ..., RN_{\tau_b}\}$
35: **while** $DescendantSize(e_i \in updatePath) < treeSize(RN_{\tau_b})/2$ **do**
36: updateDescendantSize(bigtreePath)
37: **end while**
38: e_i = new RN
39: reverse(e_i)
40: betweenRNsPath={e_{i+1}, e_{i+2}, ..., e_n, RN_{τ_b}}
41: updateDescendantSize(betweenRNsPath)
42: **end if**
43: **end if**
44: **end if**
45: **End**

path needs to have its *DescendentSize* updated and checked for a new *RN*. Here again, the number of operations required is proportional to the length each of these paths (orphan and extended).

4.6 Algorithm complexity

For all the possible tree updates, and associated dynamic recomputation of *RN*, a small constant number of path updates were required, with each path update requiring a number of operations proportional to its length. The worst-case is the edge replacement with between one and two operations required on three different paths (orphan, small and extended). In the worst case, a degenerate tree, the path length can be $O(n)$. In a perfectly balanced tree, the worst case and average path length is $O(\log n)$. It has been demonstrated that the average depth of a tree with n vertices is $O(\log n)$ [23]. Therefore, the average complexity of our proposed method is no larger than $O(\log n)$.

4.7 Experiments and evaluation

4.7.1 Experimental setup

First, let see what happens in a dynamic network. A dynamic network consists of many individual computers. As the computers begin communicate, one after another, small networks appear. Then, messages join the networks, forming a huge network. At that point, the network has reached its full size and the associated graph only changes when we have a new connection between nodes, or when the edge weights decrease.

We simulate this typical situation of a huge dynamic network. Random forests are gradually generated by adding vertices incrementally. Trees are joined together and finally form a huge tree in each forest. Moreover, since cycles are forbidden in a tree, the *MST* algorithm eliminates one of the connections in the cycle when a connection adds an edge that creates a cycle. To follow this model, our simulation supports four operations : (i) *Insertion* : a new vertex connects to the graph, or two new vertices connect to each other and create a new tree ; (ii) *Join* : two trees are connected with a new link ; (iii) *Cycle* : a new link generates a cycle ; and (iv) *updateEdge* : the weight of an existing edge increases, possibly affecting the *MST* computation.

Each line in the dataset represents three numbers : ν, ξ, and ω. These numbers denote the first vertex, the second vertex, and the edge weight respectively. Algorithm 7 illustrates the data analysis of the dataset : (i) if ν and ξ are two new vertices, and they generate a new tree (line 6) ; (ii) if either ν or ξ is a new vertex, it connects to the existing tree. Since

this vertex is new, it cannot create a cycle in the tree, and is therefore not required to call the *MST* algorithm (lines 8-18) ; (iii) if both ν and ξ exist in the forest and are located in two separate trees, this implies that those trees are joining together and forming a bigger tree (line 20), and the *MST* algorithm is not required in this case either ; and (iv) if ν and ξ belong to a tree and there is no edge between them (line 23), this new edge creates a cycle in the tree and the *MST* function must be run to eliminate one of the edges in the cycle (line 32). In the case of new edge elimination, the *RN* does not have to be recomputed. Otherwise, the information about the *add()* and *cut()* operations are passed to the *Update_RN_Cycle* function to find new *RN* (line 35). If ν and ξ belong to a tree, there is an edge between them, and if the previous weight is greater than the new one, the edge weight is updated (line 30).

The dataset consists of one million operations (*Insertion*, *Join*, *Cycle*, and *updateEdge*). Six datasets are used in this simulation, ranging from 10,000 to 60,000 vertices in the forest. We consider a reasonable number of situations that stress this test. Table 4.1 presents the statistics for our datasets.

4.7.2 Results

Table 4.2 illustrates the simulation results. Each category has a *Number* column, which indicates : (1) how many insertions there are, i.e. when a new vertex is added to an existing tree in the forest ; and (2) how many times two vertices make a new tree. The win_{RN} column gives the number of *RN* changes in each category, and $lose_{RN}$ indicates how many unchanged *RN* there are, i.e. when a new vertex connects to the tree. Statistically, 15% of the new vertex insertions change the number of *RN*.

The number in the Join column indicates the situations where two existing vertices from two separate trees connect and the two trees are merged. There are four possibilities with this type of connection : (1) two *RN* from two trees connect ; (2) one vertex of the small tree connects to the *RN* of the big tree ; (3) the *RN* of the small tree connects to one vertex of the big tree ; and (4) two vertices from two trees connect. The *RN* changes only in the third and fourth possibilities. Table 4.3 shows these four possibilities in the datasets. The third and fourth columns constitute the major part of the tree merging operations. Therefore, as shown in the second section of Table 4.2, the *RN* of the big tree changes in 50% of cases.

The *Number* column in the *Cycle* section indicates when a new edge creates a cycle in the tree. Most cycles occur when all the trees are established and merged into a big tree. The *MST* algorithm eliminates one edge in the cycle. If the tree changes, the *RN* algorithm must be run. Statistics in the *Cycle* section demonstrate that in less than 3% of cases the *RN* changes. Indeed, in most cases the balance of the big tree does not change.

Thereafter, we analyze the position of a new vertex insertion in the existing tree and a new

ALGORITHM 7: ReadDataSet()

```
1:  ν : vertex
2:  ξ : vertex
3:  ω : edge weight
4:  Begin
5:  if exist(ν)= FALSE and exist(ξ)=FALSE then
6:    createTree(ν,ξ,ω)
7:  else
8:    if exist(ν)= TRUE and exist(ξ)=FALSE then
9:      addExistTree(ν,ξ,ω)
10:     id= treeId(ν)
11:     χ= referenceNode(id)
12:     Update_RN_Insertion(χ,ν)
13:   else
14:     if exist(ν)= FALSE and exist(ξ)=TRUE then
15:       addExistTree(ν,ξ,ω)
16:       id= treeId(ξ)
17:       χ= referenceNode(id)
18:       Update_RN_Insertion(χ,ξ)
19:     else
20:       if exist(ν)= TRUE and exist(ξ)=TRUE and treeId(ν) != treeId(ξ) then
21:         Update_RN_Join(ν,ξ)
22:       else
23:         if exist(edge(ν,ξ))= FALSE then
24:           modification= MST(ν,ξ,ω)
25:           id= treeId(ν)
26:           χ= referenceNode(id)
27:           Update_RN_Cycle(χ,modification)
28:         else
29:           if weight(edge(ν,ξ)) > ω then
30:             updateWeight(ν,ξ,ω)
31:           end if
32:         end if
33:       end if
34:     end if
35:   end if
36: end if
37: End
```

Table 4.1 Number for each operation, from a total of one million operations

	Nodes	Insertion	Join	Cycle		updateEdge
				Stay[1]	Remove[2]	
$Dataset_1$	10000	4991	2503	45449	946879	178
$Dataset_2$	20000	9892	5052	76404	908556	96
$Dataset_3$	30000	15005	7496	102672	874761	66
$Dataset_4$	40000	19955	10021	125650	844322	52
$Dataset_5$	50000	24959	12519	145733	816753	36
$Dataset_6$	60000	29953	15022	164104	790885	36

[1] the new connection stays in the loop and the other edge in the cycle is removed by the MST algorithm.
[2] the new connection has the highest weight in the cycle and is removed by the MST algorithm

edge insertion in the join/cycle. When a new vertex connects to the RN of an existing tree directly, the RN does not change (Eq. 4.5). Therefore, the number of RN changes depends on the insertion position of the new vertex. In Table 4.4, the *Insertion* section presents the total of number of insertions, the number of updated vertices, and the average distance from the new connection to the RN. As shown, for the first dataset of 10000 nodes, a new vertex insertion causes 18 updates along the propagation path to RN, on average. As Table 4.2 illustrates, the RN changes in only 15% of cases.

As shown in Table 4.3, most cases are associated with trees joining, where two vertices from two trees connect ($m_s - e_b$). In this case, the update is performed from the small tree RN to m_s and from e_b to the big tree RN. As Table 4.4 illustrates, the range of average updates for $Dataset_1$ to $Dataset_6$ is 20-36.

As mentioned, in the *Cycle* case, updates are performed along three paths : *Small*, *Orphan*, and *Extended*. When the *cut()* and *add()* operations are performed on the same side of the RN, there is no ”extended path”. As shown in Table 4.4, on average, 75.5 and 145 updates are performed when there are 10000 and 60000 nodes respectively.

Table 4.2 The result of proposed algorithm for six datasets in term of RN changes

	Insertion					Join					Cycle				
	Number[1]	win_{RN}[2]	%	$lose_{RN}$[3]	%	Number[4]	win_{RN}	%	$lose_{RN}$	%	Number[5]	win_{RN}	%	$lose_{RN}$	%
$Dataset_1$	4991	732	15%	4259	85%	2503	1464	58%	1039	42%	45449	1186	3%	44263	97%
$Dataset_2$	9892	1435	14%	8457	86%	5052	2972	70%	2080	30%	76404	1259	2%	75145	98%
$Dataset_3$	15005	2270	15%	12735	85%	7496	4439	59%	3057	41%	102672	1508	2%	101164	98%
$Dataset_4$	19955	2986	15%	16969	85%	10021	5847	58%	4174	42%	125650	1365	1%	124285	99%
$Dataset_5$	24959	3675	15%	21284	85%	12519	7298	58%	5221	42%	145733	1432	1%	144301	99%
$Dataset_6$	29953	4373	15%	25580	85%	15022	8839	59%	6183	41%	164104	1776	1%	162328	99%

[1] The total number of cases where a vertex is added to an existing tree. Note that other cases belong to two new vertex connections, which form a new tree. In a recent case, one of the vertices was selected as the RN and there was no computation performed to find it.
[2] The number of cases where RN changes.
[3] The number of cases where RN does not change.
[4] The number of cases where two trees merge in the forest.
[5] The number of cases where an edge makes a cycle in one of the trees in the forest.

Table 4.3 The status of join operation

	$RN_s - RN_b$	$m_i - RN_b$	$RN_s - e_i$	$m_i - e_i$
$Dataset_1$	0	190	80	2233
$Dataset_2$	0	440	161	4451
$Dataset_3$	0	595	232	6669
$Dataset_4$	0	819	339	8863
$Dataset_5$	0	1035	420	11064
$Dataset_6$	0	1238	474	13310

4.7.3 Performance evaluation

Firstly, we analyze and compare the previous and the proposed incremental approaches in terms of execution time. The previous approach calculated the summation of shortest paths from each node to each other node. Eventually it selects the node with minimum sum as the Reference Node. Although, it is applicable for offline analysis, where the reference node is found only once, it would be costly for online analysis where the reference node should be recomputed for every change in the network. As shown in Figure 4.5, updating the reference node in a dynamic network with 10000 nodes and one million operations takes 32.53 seconds with the previous approach. The same algorithm takes 1549.92 seconds to update the reference node in a larger network with 60000 nodes and one million operations. This amount of the time is unacceptable for live data analysis.

The proposed new method improves the performance of live mode RN maintenance in a dynamic network. When the tree is modified, the method incrementally propagates the consequences of the update and recomputes the RN efficiently, ignoring unaltered parts to find the new RN.

The proposed method has been tested under the same conditions with one million changes in the live network. We applied the same operations on the same forests, and calculated the proposed method's execution time for updating the RN. As shown in Figure 4.6, our method takes 0.34 seconds to recompute the RN in a forest with 10000 nodes. This is compared with 32.53 seconds for the previous approach on the same network. Also, our method takes 4.93 seconds to update the RN in a forest with 60000 nodes, as compared with 25 minutes and 49.92 seconds with the previous approach.

LTTngTop [8] was used to analyze the performance of the proposed method in terms of page fault rate. This tool displays various system metrics in real time extracted from a detailed operating system trace. The trace is produced with low overhead by *LTTng* [30] and analyzed directly in the shared memory buffers by *LTTngTop*, without the need to write the trace to disk. Figure 4.7 illustrates a gradual increase in the number of page faults with an

Table 4.4 Number of descendantSize update in each operation

	Insertion			Join			Cycle		
	Number	Update	Avg. Distance to RN	Number	Update	Avg. Distance to RN	Number	Update	Avg. Distance to RN
$Dataset_1$	4991	89985	18	2503	50930	20	45449	3431773	75.5
$Dataset_2$	9892	246827	25	5052	128818	25.5	76404	7614996	100
$Dataset_3$	15005	366967	24	7496	191811	26	102672	12250493	119
$Dataset_4$	19955	513153	26	10021	275419	27.5	125650	15740773	127
$Dataset_5$	24959	762460	31	12519	398419	32	145733	18804027	129
$Dataset_6$	29953	1018792	34	15022	540992	36	164104	23822448	145

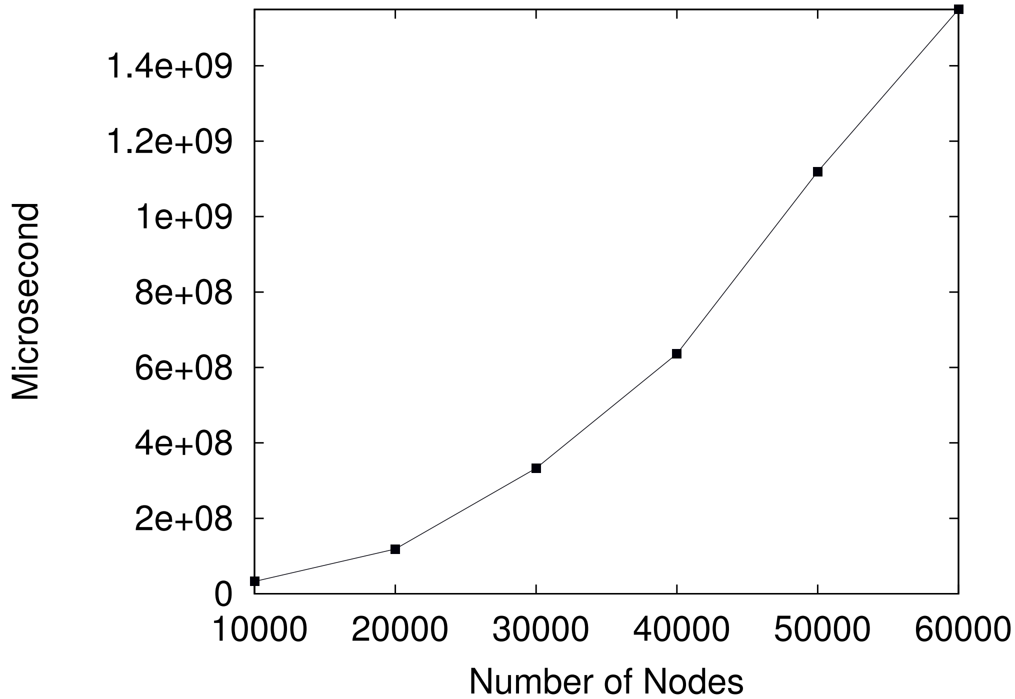


Figure 4.5 Execution time for recomputing the RN as a graph with an increasing number of updated vertices. The updating sequences contain one million operations, consisting of *Insertion*, *Join*, *Cycle*, and *updateEdge*, in a forest. The previous algorithm measured here has a complexity of $O(n^2)$

.

increase in the number of nodes from 10000 to 60000. The minor page faults go from 9090 for 10000 nodes to 12894 for 60000 simulated nodes with the proposed algorithm. During RN computation, the proposed approach incurs almost no major page faults, while the previous approach sustained more than 5 major page faults per second.

Figure 4.8 illustrates the memory usage of the proposed method. It shows a gradual increase in memory usage with the number of simulated nodes, growing from 9.2 MB for 10000 nodes to 12.2 MB for 60000 simulated nodes. The amount of memory scales nicely with the size of the problem.

4.8 Conclusion

In this paper, we have presented a method to maintain the reference nodes in a dynamic forest. A valuable research contribution is presented by introducing a novel method for the online analysis of new vertex insertion, tree merging, and cycle handling in a forest, with $O(\log n)$ average time complexity per operation, where n is the number of nodes in

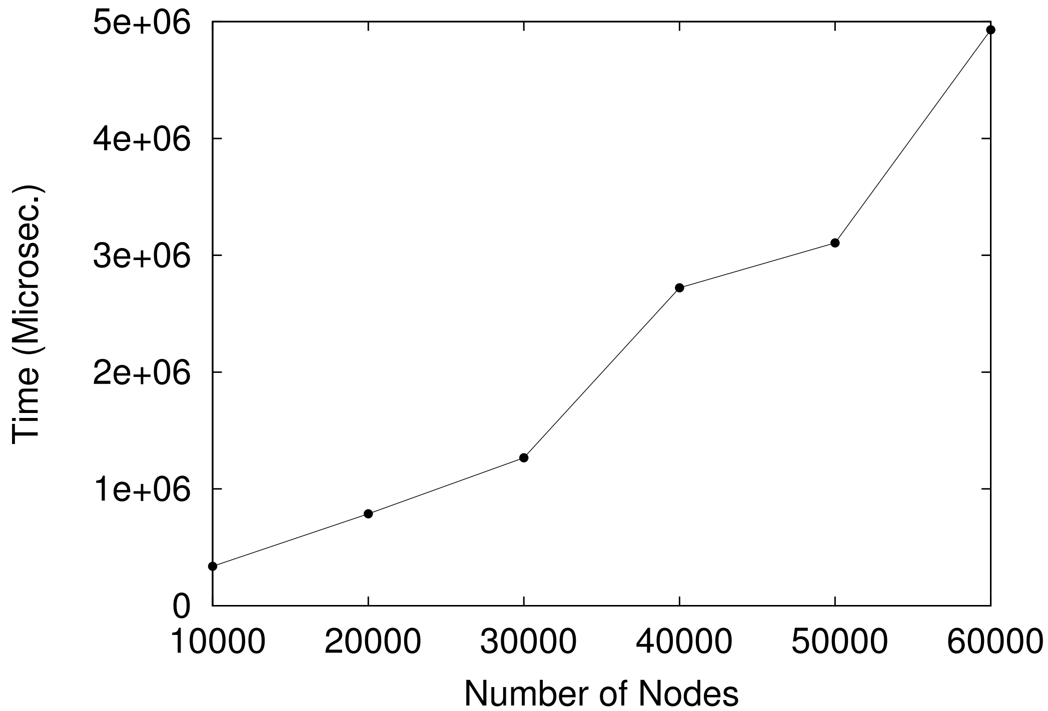


Figure 4.6 Dynamic Time *RN* : running time on a random graph with an increasing number of vertices plotted using an algorithm with a complexity of $O(log\ n)$. Updating sequences contained one million operations including *Insertion*, *Join*, *Cycle*, and *updateEdge*, in a forest

the network. In short, the proposed method improves performance, thanks to its ability to incrementally process updates in evolving trees in the forest. The previous method suffers from the cost of checking the whole forest, even when there has been no change after tree modification has been performed several times. After a large number of modifications have been made, and a sparse forest has grown into a large tree, the performance improvements are even greater. One of the most interesting use cases of the proposed method is synchronization in dynamic wired or wireless networks.

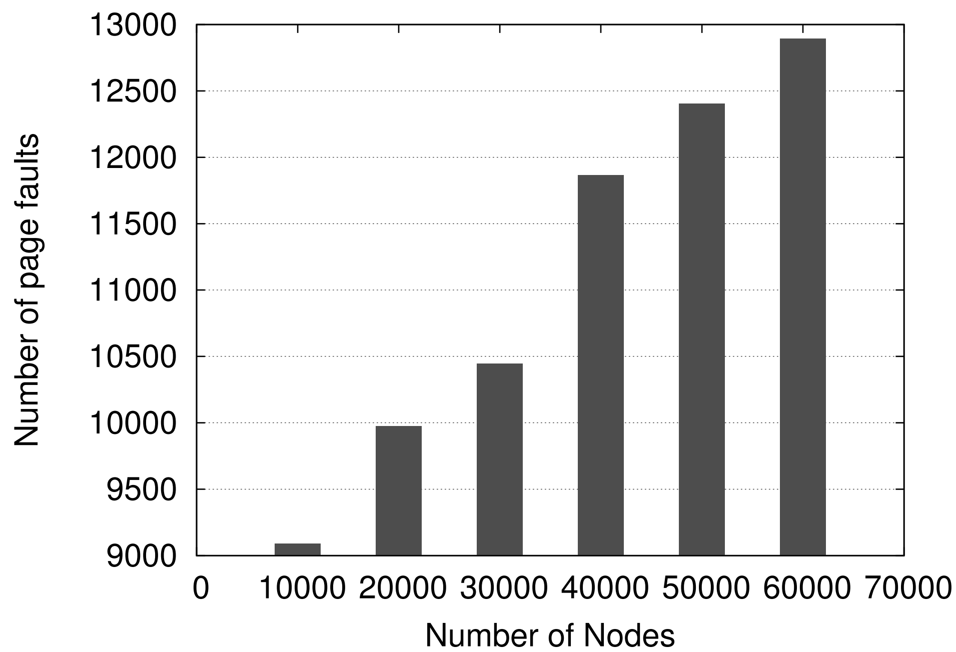


Figure 4.7 The rate of page faults with the proposed method : running time increases linearly with the number of nodes.

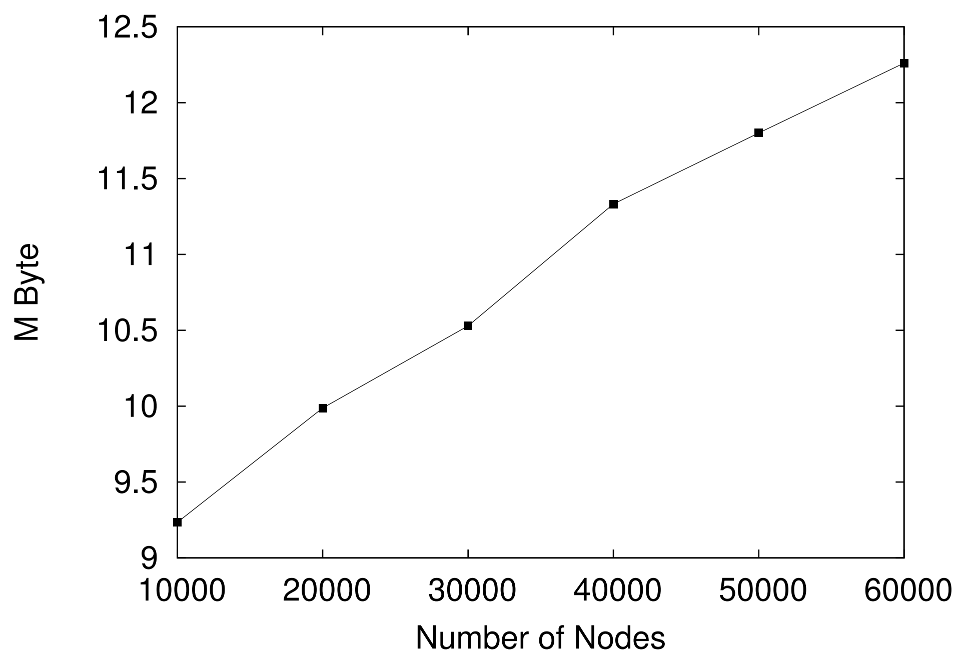


Figure 4.8 The memory usage of the proposed method ; the running time increases linearly with the number of nodes.

CHAPTER 5

Paper 3 : LIANA : Live Incremental Time Synchronization of Traces for Distributed Systems Analysis

Masoume Jabbarifar and Michel Dagenais

5.1 Abstract

Tracing and monitoring tools, and other similar analysis tools, add new requirements to the old problem of coping with asynchronous clocks in distributed systems. Existing approaches based on the convex hull can achieve excellent accuracy for a posteriori analysis, but impose a significant cost and latency when used in live mode and over large clusters. We propose a novel method, LIANA (Live Incremental Asynchronous Network Analysis), for incrementally computing the clock offset, and updating it as the network evolves, along each communication link, as well as selecting the best synchronization paths and time reference node. Each connection in a network requires message exchanges to compute the clock skew and offset between two connected nodes. This method relies on the trace events recorded for the existing TCP/IP traffic between nodes. After computing the offset and its accuracy for every connection in the network graph, a minimum spanning tree is computed. The edges with the best accuracy are selected and form the spanning tree. Then, a central node is selected as the time reference to optimally compute the offset from any node to this reference node. LIANA is efficient, both in terms of synchronization accuracy and time complexity. The method, which is used for online distributed trace synchronization, has been evaluated in realistic scenarios with a diverse set of network topologies and traffic. We show that LIANA generates precise results highly efficiently, which makes it suitable for large cloud-distributed systems.

5.2 Introduction

Distributed systems provide a versatile computing platform for a number of applications, such as routing algorithms in telecommunications networks, banking systems, and aircraft control systems, as well as in scientific computing, including cluster and grid computing. These systems are typically monitored to detect and debug problems. Tracing tools are often the preferred monitoring method, since they can record detailed information for each individual system which can then be analyzed. In clusters, a large task is often divided into many

smaller tasks distributed throughout the cluster. So, in order to monitor the functionality of a distributed system, information from each node in the cluster is collected, merged, and then processed. The result is a huge flow of traced events, timestamped at the nanosecond level, but using each node's independent clock. Synchronizing events from a stream of this scale is the main challenge addressed in this paper.

Because of the large scale of the data stream, data buffering space limitations, and the need for timely results, live analysis is required in many cases. This explains the need for the efficient and accurate streaming mode synchronization of distributed traces. The method presented in this paper is motivated by a novel online tracing and monitoring system called *LTTng*. This system requires the accurate synchronization of traces from nodes in large infrastructures at cloud computer scale.

A computer cluster consists of many individual computers from which traces can be extracted. As each computer starts to communicate with other computers, the network links become visible in the traces through packet send and receive events. Initially, these links form several subgraphs and, as new links are exercised and appear in traces, subgraphs are joined together and may eventually become one large connected graph for the whole cluster. For each link, a clock offset and skew between two connected nodes can be computed, along with bounds on its inaccuracy. For distributed trace analysis purposes, a reference node needs to be identified, and the offset and skew for each node with respect to the reference node needs to be computed. A Spanning Tree (ST) of computers is formed incrementally, with edges (links with their associated inaccuracy) being added as packets are exchanged between new pairs of computers. The ST algorithm eliminates highly inaccurate redundant links and prevents cycling. A time reference node is selected, and may change as links are added. Then, events from different nodes can be compared by looking up the clock offset and skew with the reference node along the ST.

Our objective was to design and implement a high-speed method for synchronizing distributed traces. We use splay trees to store the dynamic graph of traced nodes. This dynamic graph handles variations over time in the inaccuracy of the offset computation between nodes. As new packets are exchanged, either new links are added to the graph or the inaccuracy associated with a link decreases. As a result, the ST may be updated and a new reference node selected.

The main contribution of this work is that the new method works incrementally, supporting the live analysis of streaming mode traces. Moreover, it achieves higher performance than previous methods in both streaming mode and batch mode, while retaining the property of computing skew and offset with optimal inaccuracy bounds. This new scheme quickly identifies the few accurate packets that will improve the inaccuracy bound in $O(1)$ average

time for processing each packet receive or send event. Once the accuracy of a link has been updated after one of the few accurate packets has been found, the ST is updated as needed in $O(log\ n)$ amortized time, n being the number of nodes, using the splay tree algorithms. Finally, dynamic time reference node selection is updated incrementally, which takes $O(log\ n)$, on average.

In section 5.3, we examine related work in this area, and in section 5.4 we provide terminology and background. In section 5.5, we detail the new proposed algorithms, and in section 5.6, we present our experimental results. Finally, in section 5.7, we conclude the paper and discuss possible future work.

5.3 Related Work

Duda et al. [37] proposed the *Linear Regression* and *Convex-Hull* algorithms for offline clock synchronization. The Linear Regression algorithm provides a fairly accurate synchronization approach [58] ; however, because the *Convex-Hull* algorithm is based on the fact that packet send and arrival times impose bounds on the clock offset and skew, it guarantees the highest level of synchronization accuracy [59]. In [14], the timestamps are corrected using the shortest round trip delays between the packets exchanged. Moon et al. [82] use a Linear Programming algorithm to estimate the one-way delay between two nodes.

Zhang et al. [104] discuss the estimation and removal of the relative clock skew based on delay measurements. Their method, like ours, uses the *Convex-Hull* algorithm for online and offline clock synchronization. However, they scan through the measurement points in increasing order, store the lower *Convex-Hull* points, and then estimate the clock skew at the end of each interval. Since they use a time interval to gather the connection information for skew removal, their approach takes a time $O(n)$ at each interval, or $O(n^2)$ globally. However, it also adds a latency of up to one interval, since the computation is postponed to the end of each time interval. By contrast, our scheme updates the bounding hull and clock skew incrementally, and filters out points that may lie on the *Convex-Hull* temporarily, but cannot affect the skew computation, leading to $O(1)$ time complexity per packet processed and $O(n)$ globally, without postponing the computation.

Khlifi et al. [63] proposes two algorithms to remove the skew during offline trace analysis. The first algorithm, *average*, computes the average delay for a fixed number of consecutive packets at the beginning and end of a trace. This algorithm works with a constant $O(1)$ complexity. The second technique, *direct skew removal*, has the interesting property of being able to account for low clock resolution, where the clock granularity may be larger (e.g. 1ms) than the packet delay. To achieve this, it analyzes the whole trace for a linear $O(n)$

complexity. While efficient, these two algorithms do not provide the same accuracy as the *Convex-Hull* method.

None of these synchronization approaches, including our proposed method, requires additional network messages to estimate the linear clock deviations between two nodes. A number of other online synchronization methods, such as Elson et al.'s schema [40], rely on broadcasting synchronizing packets. Likewise, in [89], the proposed algorithm generates additional network traffic to estimate and compensate for the timestamping delay and the network latencies. However, for most real-time applications, a synchronization method without additional network traffic load is preferred.

In a distributed system with more than two nodes, synchronization is performed along a Minimum Spanning Tree (MST), in order to decrease time conversion errors. Kruskal's algorithm [23] computes the MST in a time of $O(m \log n)$, where m and n are the number of edges and vertices respectively. The edges are placed in a priority queue in this algorithm. The algorithm extracts the lowest edge from the queue and adds it to the *MST*, unless it forms a cycle, in which case it is discarded. This procedure is repeated until the *MST* has n-1 edges, which means that all the vertices are reached through the *MST*. For online purposes, various algorithms are available to maintain changes in a dynamic tree [23, 98]. Among them, the algorithms presented in [19, 51] update the *MST* in a time of $O(\log n)$. We use these algorithms in our approach to synchronize distributed traces.

The next generation Linux Trace Toolkit (LTTng) tracks performance and debugging problems. Tracing across multiple systems in a cluster helps uncover various problems which are hard to find [29, 31]. Trace events are recorded based on the local system clock. Since every system has its own clock in a distributed system, a practical synchronization approach is required to order events based on a single reference time. Properly ordered events simplify the analysis of distributed systems [56]. The characteristics of clock skew and drift, and their estimation, have been studied in [22, 29, 75]. Then, Poirier et al. [85] proposed an efficient and accurate algorithm for offline trace synchronization based on the *Convex-Hull* algorithm. However, their method is not efficient for online analysis purposes.

5.4 Terminology and background

In this section, we introduce the terminology used in the remainder of the paper, and we formalize the definition of the clock skew.

Time offset, frequency offset, and frequency offset rate are parameters that describe the behavior of a clock, and these differ from one clock to another. The trajectory of the time offset can be modeled by the following equation [39] :

$$\Delta T(t) = \beta(t_0) + \alpha(t_0)(t - t_0) + \ell(t - t_0)^2 + \in(t) \tag{5.1}$$

$\Delta T(t)$	Time offset at time t
$\beta(t_0)$	Initial offset
$\alpha(t_0)$	Frequency offset
ℓ	Frequency drift
$\in(t)$	Other factors, particularly random perturbations

Equation 5.1 shows that clock inaccuracies are caused by a combination of various factors. Over relatively short intervals, many algorithms consider that only the initial offset and the frequency offset are significant. We refer to this as the "linear clock approximation". Taking this approximation into account, equation 5.1 can be simplified to :

$$\Delta T(t) = \beta(t_0) + \alpha(t_0)(t - t_0) \tag{5.2}$$

Finding the time offset between a node clock and a virtual perfect clock becomes a matter of identifying two factors in a linear equation. It follows that the offset between two real clocks can also be modeled as a linear function. For the rest of this paper, we estimate a function that maps the time on clock A to the time on clock B as follows :

$$C_A(t) = \alpha_0 + \alpha_1 C_B(t) \tag{5.3}$$

Moreover, the structure of a trace can be illustrated as follows :

$$\begin{aligned} &T = (drift, offset, start_time_from_TSC, events) \\ &events = (e_1, e_2, e_3, ..., e_n) \end{aligned} \tag{5.4}$$

Let us assume that there are two traces in a distributed system, T_0 and T_1, on computers C_0 and C_1 respectively. Two event types are considered for time synchronization : (i) sending a message ; and (ii) receiving a message. Let us denote by θ_i the time when C_0 sends message i to C_1, and by ξ_i the time when C_1 receives message i from C_0.

$$m(i) : T_0(\theta_i) \mapsto T_1(\xi_i) \tag{5.5}$$

The timestamp for the sent message is stored in T_0 and the timestamp for the received message is stored in T_1. θ_i and ξ_i are based on the local time of C_0 and C_1 respectively. In addition, C_1 sends message j to C_0, and θ_j is the time when C_0 receives message j from C_1.

$$m(j) : T_1(\xi_j) \mapsto T_0(\theta_j) \tag{5.6}$$

Each trace contains sent (S) and received (R) message timestamps, based on local time, as expressed by the following sets :

$$\begin{aligned} &T_0.events = (\theta_i^S, \theta_j^R, ...) \\ &T_1.events = (\xi_i^R, \xi_j^S, ...) \\ &i, j = 1, 2, 3, ... \end{aligned} \tag{5.7}$$

As shown in sets T_0 and T_1, $\overrightarrow{(\theta_i^S, \xi_i^R)}$ is the first pair of send-receive times for the message sent by C_0 to C_1, and $\overleftarrow{(\theta_j^R, \xi_j^S)}$ is the second pair of send-receive times for the message sent by C_1 to C_0. If the event timestamps of T_0 are considered as reference times, this gives us the following equation :

$$\begin{aligned} &C_{T_0}(t) = \theta^S \\ &C_{T_1}(t) = \alpha\theta^S + \beta \end{aligned} \tag{5.8}$$

5.5 Methodology

Dealing with streaming data involves many challenges. Since long-term buffering of streamed data incurs an unacceptable cost in many cases, the stream should be scanned and analyzed in a timely fashion. Ideally, an online synchronization method should be efficient, in terms of both *time* and *memory.* Another challenge is to ensure that the method is scalable to a large number of nodes, has low latency, and so generates its results quickly and maintains good synchronization accuracy over time.

5.5.1 Two-node synchronization

The method proposed in [57] synchronizes every connection between two nodes incrementally with a method based on the *Convex Hull* algorithm, which estimates a conversion function between the clocks of a pair of traced computers. Figure 5.1 illustrates the sent and received packets in a two-dimensional chart based on the source and destination clocks. The main features of this incremental approach to synchronizing two nodes are summarized in this subsection, in order to explain how it fits into the complete cluster synchronization process proposed in this article.

The set $\{\overrightarrow{(\theta_i^S, \xi_i^R)}, \overrightarrow{(\theta_{i+1}^S, \xi_{i+1}^R)}, ...\}$ shows the sent packets from computer θ to computer ξ, and the set $\{\overleftarrow{(\theta_j^R, \xi_j^S)}, \overleftarrow{(\theta_{j+}^R, \xi_{j+1}^S)}, ...\}$ shows the sent packets from computer ξ to computer θ. Since there is no message inversion in a normal connection, these two sets are completely separate.

The *Convex-Hull* algorithm uses maximum received times and minimum sent times, i.e.

packets with minimum latency, in order to accurately synchronize connections. The packets with minimum latency are those of interest in the *Convex-Hull* synchronization algorithm. In Figure 5.1, packets sent from θ (horizontal axis) to ξ (vertical axis) occupy the upper left half-plane and are shifted higher when more network latency was encountered. Therefore, the lower half-hull, of the *Convex-Hull* formed by those points, is a lower bound for the packets sent from θ and identifies the packets with the lowest latency. Similarly, packets sent from ξ (vertical axis) to θ (horizontal axis) occupy the lower right half-plane and are shifted to the right when more network latency was encountered. Therefore, the upper half-hull, of the *Convex-Hull* formed by those points, is an upper bound for the packets sent from ξ and identifies the packets with the lowest latency. The possible synchronization lines lie below the lower half-hull of packets sent from θ and above the upper half-hull of packets sent from ξ.

This means that the lower bound of the sent packets and the upper bound of the packets received by computer θ determine the possible range for the linear clock function. Graham's scan forms these two sets and their bounds. The formula 5.9 shows the corresponding pairs in each bound in Figure 5.1.

$$\begin{aligned} UpperBound &= \{\overleftarrow{(\theta_1^R, \xi_1^S)}, \overleftarrow{(\theta_4^R, \xi_4^S)}, \overleftarrow{(\theta_5^R, \xi_5^S)}\} \\ LowerBound &= \{\overrightarrow{(\theta_1^S, \xi_1^R)}, \overrightarrow{(\theta_3^S, \xi_3^R)}, \overrightarrow{(\theta_4^S, \xi_4^R)}, \overrightarrow{(\theta_6^S, \xi_6^R)}\} \end{aligned} \tag{5.9}$$

We see that this algorithm ignores inaccurate pairs $(\overrightarrow{(\theta_2^S, \xi_2^R)}, \overrightarrow{(\theta_5^S, \xi_5^R)}, \overleftarrow{(\theta_2^R, \xi_2^S)}, \overleftarrow{(\theta_3^R, \xi_3^S)})$, which are packets delayed by interrupts, network switches, etc.

In the next step, the *Convex-Hull* algorithm attempts to draw two lines. One line has a maximal slope, L_{max}, and the other has a minimal slope, L_{min}, as illustrated in Eq. 5.10. The final estimation line is the line that bisects L_{max} and L_{min}.

$$\begin{aligned} L_{max} &= \alpha_{max}\theta + \beta_{min} \\ L_{min} &= \alpha_{min}\theta + \beta_{max} \end{aligned} \tag{5.10}$$

As mentioned, the *Convex-Hull* algorithm uses the exchanged packets with minimum latencies and ignores the other packets. The basic idea of the *Fully Incremental Approach* [57] is to benefit from this specific feature of the *Convex-Hull algorithm*, by selecting accurate packets that strengthen the bounds. It proposes a novel online synchronization method for two connected computers. Online synchronization starts as soon as enough exchanged packets are found. Synchronization accuracy then improves over time, as more accurate packets are received.

Synchronization accuracy is the difference between the drift of two lines with maximum and minimum slopes (L_{max} and L_{min}).

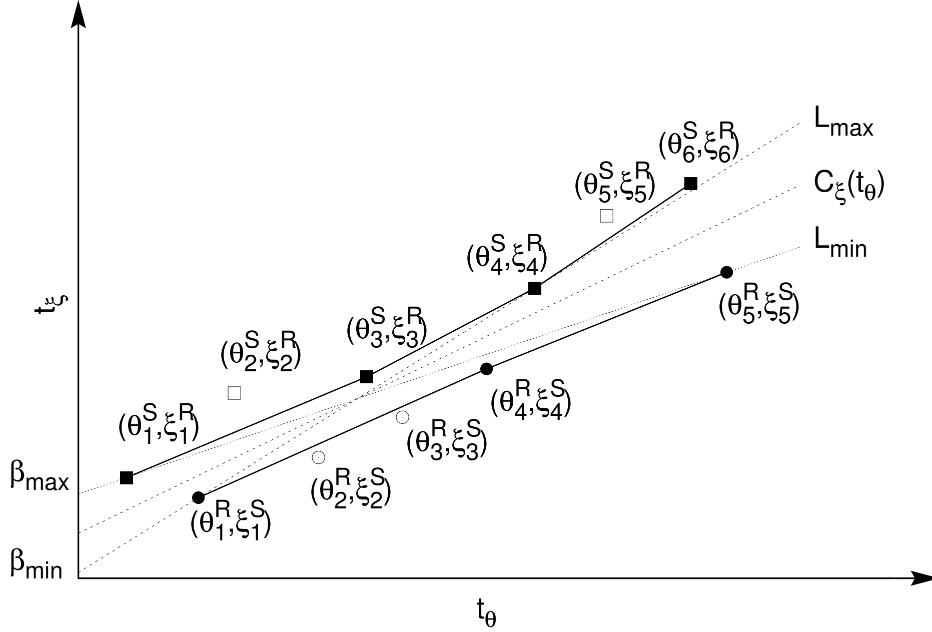
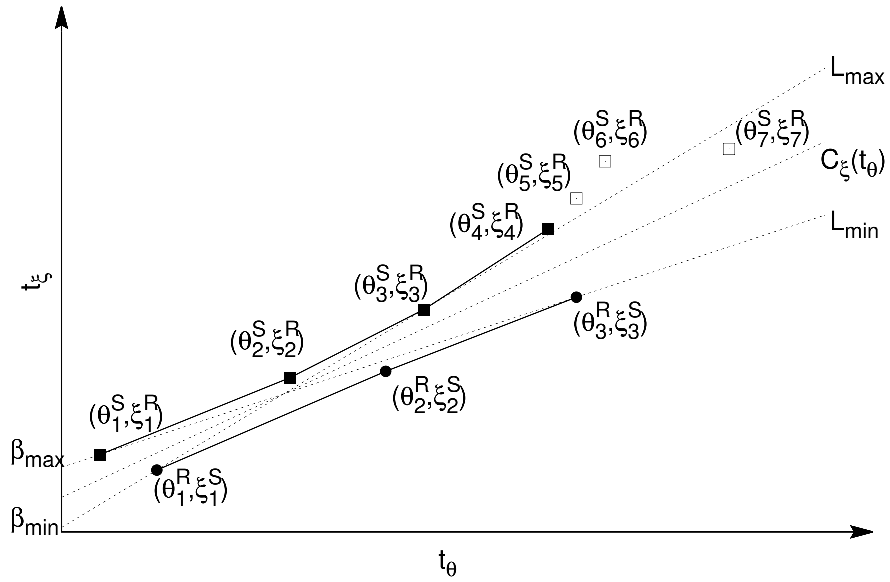


Figure 5.1 Convex-Hull method.

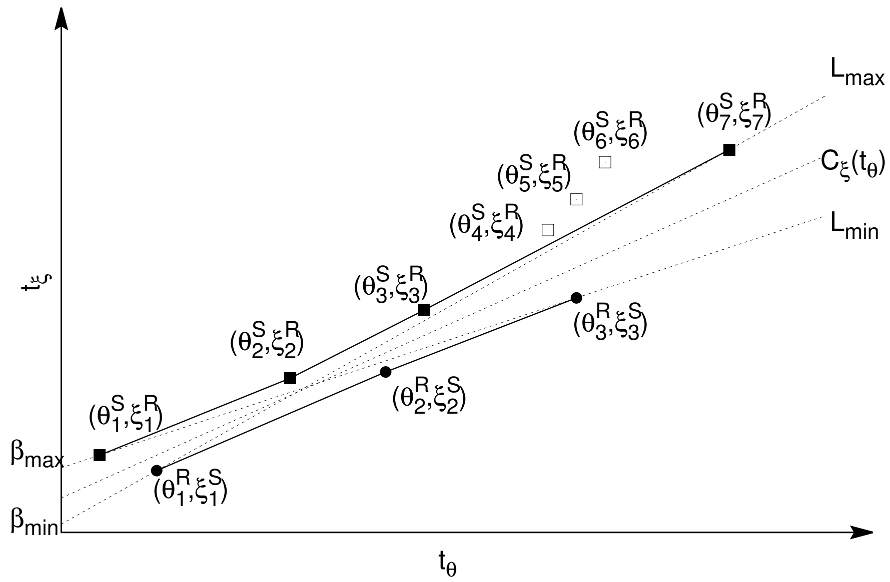
$$Accuracy = L_{max}.drift - L_{min}.drift \tag{5.11}$$

Consequently, the *Fully Incremental Approach* only examines the exchanged packets that impact one of these lines, narrowing the gap between the minimum and maximum lines. These packets are called *accurate packets* in this approach. A packet is accurate when it is placed either below line L_{max} or above line L_{min}. For example, in Figure 5.2a, $\overrightarrow{(\theta_7^S, \xi_7^R)}$ is an accurate packet sent from computer θ to computer ξ, which moves L_{max} to the new position in Figure 5.2b. However, packets $\overrightarrow{(\theta_5^S, \xi_5^R)}, \overrightarrow{(\theta_6^S, \xi_6^R)})$ suffered increased latency in reaching computer ξ, and they do not lower L_{max} or improve accuracy, and so they are ignored.

In summary, the *Fully Incremental Approach* updates the drift and offset, and the associated accuracy, between the traces on two nodes, and always guarantees the best accuracy during live tracing. Moreover, unlike the classic *Convex-Hull algorithm*, it not only retains the packets that lie on the *Convex-Hull*, but also affects L_{max} or L_{min}. The updated drift and offset are computed immediately when L_{max} or L_{min} are affected, which is not the case in the window-based approaches, where the evaluation is postponed to the end of each window [57]. The immediately available updates on the drift, offset, and accuracy of this approach are fed into the proposed new cluster synchronization approach, as detailed in the next subsections.



(a) The accurate packet position before updating the synchronization



(b) Synchronization based on the accurate packet position

Figure 5.2 Fully Incremental Approach

5.5.2 Multi-hop synchronization

Since synchronization is performed using streaming data from a cluster, the idea is to maintain a synchronization graph dynamically. Edges are added when new nodes start communicating, and edge weights (synchronization accuracy between two nodes) are updated when more accurate packets are received.

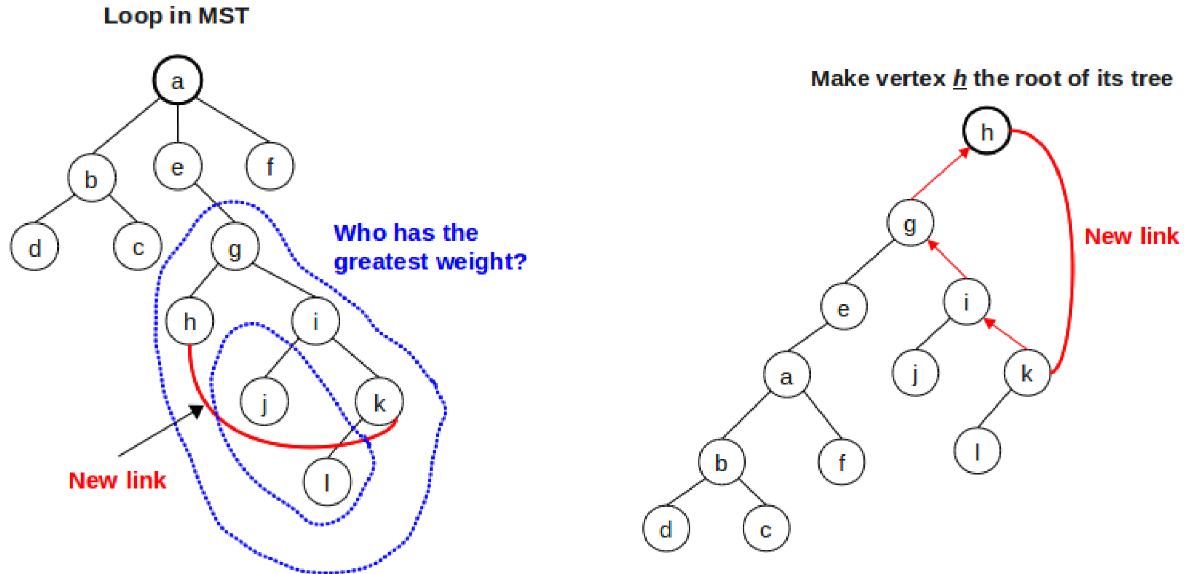
A *Minimum Spanning Tree (MST)* in the graph is maintained, and a root with the minimum cost path to all the other nodes is selected as the time reference node. This way, links with the best accuracy are retained and used to compute the drift and offset from any node to the reference node.

The *MST* can change in three different situations : (i) a new computer connects to the network and starts tracing, adding a vertex ; (ii) a new connection is created between two existing computers in the network, and a new edge is added and may affect the *MST*, and possibly the time reference node as well ; (iii) a more accurate packet is received, improving the accuracy of the weight of an edge, which can lead to updating of the *MST* and possibly of the time reference node as well.

MST algorithms, such as Kruskal's algorithm, work efficiently with static trees and take a time of $O(m \; log \; n)$, on average, where m is the number of edges and n is the number of vertices. We call this a non incremental approach.

Changes occur frequently in online analysis, and the *MST* should be updated accordingly. Repeatedly applying a non incremental approach would be inefficient, leading to a complexity of $O(m^2 \; log \; n)$. The performance of streaming synchronization is improved using an incremental *MST* algorithm. When a change to the graph is made, a new edge may be added or the weight of an edge not on the *MST* may be reduced. In either case, the edge is added to the *MST* for consideration (as shown in Figure 5.3a). The *MST* is splayed, so that one of the nodes connected to the new edge becomes the root of the tree, as shown in Figure 5.3b ($Splay(h) : parent(g) = h, parent(e) = g, parent(a) = e$). In this way, the cycle created by the new edge is quickly identified. Then, the edge with maximum weight in the cycle is removed from the *MST*. This process is performed in an amortized time of $O(log \; n)$.

A data structure for the *Minimum Spanning Tree (MST)* is maintained throughout the sequence of updates. We designed it based on the dynamic tree data structure proposed by Sleator and Tarjan [98]. The proposed structure is implemented as a *splay tree (ST-tree)* [97]. These authors also consider another data structure, the None Tree (NT), which they use to account for increasing and decreasing weights in their graph. However, this structure is not applicable in our case, where the clock skew estimation can only improve as more accurate packets are received.



(a) A new connection between nodes h and k in MST, which creates a cycle

(b) Splaying on node h to move it to the root

Figure 5.3 Fully Incremental Approach

Splaying Process

The splaying process is designed to optimize the updating of the *MST* based on the frequent changes in a dynamic tree. It starts from an arbitrary state and restructures itself when a new operation occurs. This process runs from either of the two nodes with the updated connection. To splay a tree, we run a recursive schema on the specific node until it becomes the root of the tree.

We start from node x and check whether or not *parent(x)* is the root of the tree. When *parent(x)* is not the root, we splay at *parent(x)* and check whether or not *grandparent(x)* is the root of the tree. This procedure continues until we find an *ancestor(x)* in the path between x and the root of the tree and splay at the link from ancestor*(x)* to the root.

When *parent(x)* is the root, splaying is completed by everything the edge linking x and *parent(x)*. Since any node in the tree may have more than two connections, we keep the same children for nodes x and *parent(x)*, and only the parent-child edge of node x is reversed.

In Figure 5.3a, we start from node h and find node e, *ancestor(h)*, whose parent (node a) is the root of the tree. Splaying is performed between e and *parent(e)*, and node e becomes the root. Then, splaying is pursued between g and *parent(g)*, which is the new root (node e). In this step, node g becomes the root of the tree. Splaying is not completed yet, however. In the last step of the splay process, the parent of the specified node becomes the root. Here, parent(h), node g, is the root. Therefore, the last splaying step consists in splaying the edge

linking node h and *parent(h)*, node h becoming the root.

ALGORITHM 8: Dynamic Minimum Spanning Tree Maintenance

Require: ν : vertex
Require: ξ : vertex
Require: ω : edge weight (sync. accuracy)

```
1:  if exist_link(ν, ξ) then
2:    update_weight(ν,ξ,ω)
3:  else
4:    if ¬exist_vertex(ν) and ¬exist_vertex(ξ) then
5:      Id= create_Tree()
6:      add_Exist_Tree(ν,ξ,ω,Id)
7:      RN= ν
8:    else
9:      if
        (exist_vertex(ν) and ¬exist_vertex(ξ)) or (¬exist_vertex(ν) and exist_vertex(ξ))
        then
10:       if exist_vertex(ν) then
11:         id= tree_Id(ν)
12:       else
13:         id= tree_Id(ξ)
14:       end if
15:       add_Exist_Tree(ν,ξ,ω,Id)
16:       update_RN_Insertion()
17:     else
18:       if treeId(ν) = treeId(ξ) then
19:         cycle(ν,ξ,ω,tree_Id(ξ))
20:         update_RN_Cycle()
21:       else
22:         join(ν,ξ,ω,tree_Id(ν),tree_Id(ξ))
23:         update_RN_Join()
24:       end if
25:     end if
26:   end if
27: end if
```

The algorithm 8 illustrates the method for dynamically maintaining the *MST*. First, it checks the existence of the (ν, ξ) link in the data structure (line 1). Then, we update that weight (ω_t). In fact, the receipt of a new, accurate packet between two nodes can improve accuracy, and so diminish the inaccuracy in the weight of the corresponding edge. The new weight must surely be less than it was previously. Moreover, if the link is already in the *MST* and its weight improves, it will remain in the *MST*, which stays unchanged. So, we always have :

$$t > t - 1 : \omega(\nu, \xi)_t < \omega(\nu, \xi)_{t-1} \tag{5.12}$$

If edge (ν, ξ) is not in the *MST*, there are four possibilities. The first is that neither of the two vertices in the *MST* exists, in which case a new single edge disconnected tree is formed (line 4-8). The second is that one of the two vertices is already in the *MST*. The new edge connects the new vertex, resulting in an enlarged *MST* (line 9-17). The third is that both vertices exist in the *MST*. If the two vertices belong to the same tree, the new edge forms a cycle, which requires the removal of the edge within the cycle with the largest weight (line 18-21). Otherwise, when those vertices are not in same tree, the new edge connects two separate trees (line 21-24).

5.5.3 Dynamic Reference Node

In a distributed system, computers are often synchronized with a time server using a protocol such as NTP. The granularity of that synchronization is usually coarser than what is required to compare events with nanosecond-level timestamps between different traces. Moreover, the clients and servers under study and being traced may be using different time servers. For this reason, we derive the clock offset and drift directly from the trace information, and build a synchronization *MST* structure based on the network links with the best accuracy. To do so, we also need to dynamically select a time reference node (*RN*) in the *MST*, which is used to display all the events and state timelines in trace viewing tools using a common time reference. We use an efficient incremental algorithm to update the *RN*, as proposed in [55]. We summarize this approach here and explain its integration into the new proposed cluster synchronization approach.

The incremental *RN* selection must handle three different cases, depending on the possible updates to the *MST*. First, when a new node connects to the current network, the balance of the network may change (Algorithm 8 line 16), and the *RN* selection may be affected. Second, when an edge is added to existing nodes in the *MST*, a cycle is generated and an edge along the cycle must be removed, altering the *MST* topology and the selection of the *RN*. This may happen either because of a new connection between two existing nodes, or a change in the accuracy of an old connection (Algorithm 8 line 20). In fact, when the accuracy of an old connection improves over time, the *MST* algorithm adds it as a new edge and eliminates a link with maximum inaccuracy in the related cycle. Third, when two unconnected networks are joined by a new connection, the *RN* selection is also affected (Algorithm 8 line 23).

For all possible network updates, and the associated dynamic recomputation of *RN*, a small and constant number of operations is required at each node along the path affected

(from the new edge to the existing RN). In a perfectly balanced network, the worst case and average path length is $O(\log n)$. It has been demonstrated that the average depth of a tree with n computers is $O(\log n)$ [23]. Therefore, the average complexity of our proposed method is no larger than $O(\log n)$.

5.5.4 Synchronization Factor Propagation

As described earlier, some MST updates cause RN changes. With such a change, the time conversion parameters with respect to the reference node may change, for all the traced nodes in the network, because of these updates. In some applications, the MST is used *as is* as needed. In that case, when the time difference between a node and the RN is requested, the time conversion parameters along the path from the node to the RN are combined. In other applications, we must store and update the time conversion parameters with respect to the RN at each node. This latter case is examined here.

The algorithm 9 illustrates the three possible cases for updating the time conversion parameters. In the first case (lines 1-3), the MST and RN are fixed, but the accuracy associated with one edge is improved. This update also affects the children of the modified edge.

In the second case, the MST changes while either a new node connects to the network or the accuracy of a current connection improves, but the RN remains the same. When a new node or a new subtree connects to the network, the new node or subtree requires updates to their conversion parameters. Otherwise, a new connection between two existing nodes in the MST results in a cycle. Figure 5.4 shows this situation and the cut required to update the MST. The update area is outlined in Figure 5.4. The path from c_{i+1} to a_i and all the children of a_is have their conversion parameters updated (lines 4-7). Moreover, the nodes from a_{i+1} to the node before the new RN has been inserted also update their synchronization parameters with respect to the new RN. However, other paths with c_i as starting point do not require updates.

In the third case (lines 8-10), both the MST and the RN change. This is the worst case, where all the nodes in the related graph update their conversion parameters with respect to the new RN.

5.6 Experiments and evaluation

5.6.1 Simulation experiments

The proposed schema is applicable to large-scale computer clusters, including cloud and grid computer environments. To validate the proposed approach, we simulated a large scale

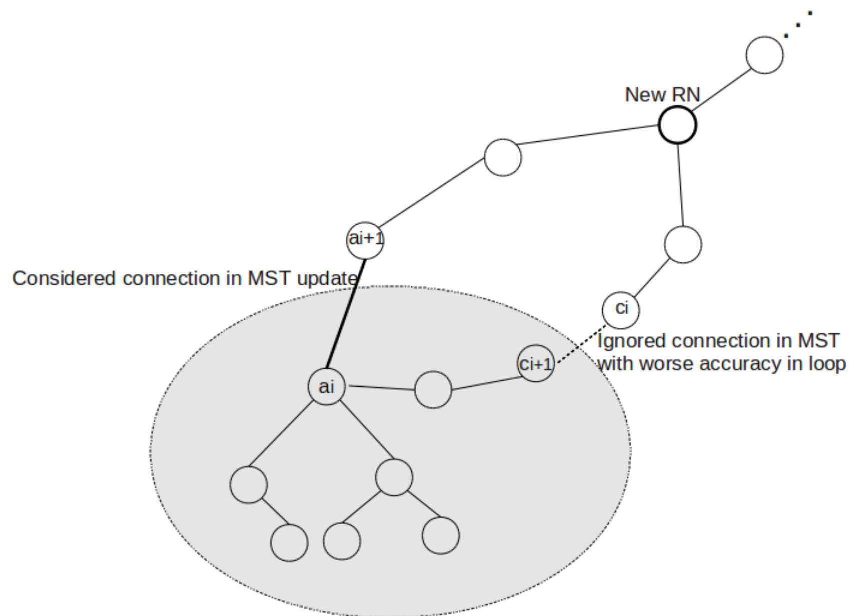


Figure 5.4 A general example of a resynchronization area when the MST changes

ALGORITHM 9: Dynamic Synchronization Factor Propagation

Require: $\Im$: MST status
Require: $\Re$: RN status
1: **if** $\Im = Static$ and $\Re = Static$ **then**
2:     convertPath($child(e_i)$)
3: **else**
4:     **if** $\Im = changed$ and $\Re = Static$ **then**
5:         reversePath(c_{i+1})
6:         convertPath($child(a_i)$)
7:     **else**
8:         **if** $\Im = changed$ and $\Re = changed$ **then**
9:             convertPath(all)
10:         **end if**
11:     **end if**
12: **end if**

consisting of 60,000 nodes with a dataset containing one million operations. Then, we experimented with the schema in this environment. The following subsections present the results of applying the proposed method to this cloud-scale computer cluster.

Simulation setup

The dataset consists of one million operations (*New computer connection*, *Network join*, *Cycle*, and *updateLink*). Six datasets are used in this simulation, ranging from 10000 to 60000 simulated nodes, which constitutes a dynamic network. Tables 5.2 and 5.1 present the operation statistics and MST changes.

Evaluation of schema performance in the simulated network

During online analysis, changes can occur rapidly, and the MST should be updated after every change. As mentioned, using a non incremental method would be costly, as it requires 1.440216 seconds to find the MST once in the simulated network with 10000 simulated nodes. As shown in Table 5.2, 992328 operations (out of one million) result in changes to the MST for this network. This means that a non incremental approach would require more than sixteen days to follow up on the changes.

As shown in Figure 5.5, our Fully Incremental Approach takes 0.36 seconds to update the MST in a network with 10000 simulated nodes, compared to 16.5 days for the non incremental approach on the same network. Moreover, the Fully Incremental Approach takes 7.79 seconds to recompute the MST in a dynamic network with 60000 simulated nodes.

The same test conditions, which include 992328 changes to the live network, are subjected to the Fully Incremental Approach. The same operations are tested on the same forests, and the time to update the RN with the proposed method is measured. Figure 5.6 illustrates that the Fully Incremental Approach takes 0.34 seconds to recompute the RN in a dynamic network with 10000 nodes, compared with 32.53 seconds for the non incremental method with complexity $O(n^2)$ on the same network [55]. Furthermore, it takes 4.93 seconds to update the RN in a dynamic network with 60000 nodes, compared with 25 minutes and 46 seconds with the non incremental RN selection approach.

5.6.2 Real world traced network

We tested our method with traces gathered in a real network cluster. This cluster is used in many applications, such as network monitoring tools, network debugging use cases, and so on. The traces were obtained with the next generation Linux Trace Toolkit (LTTng), which provides the most precise traces, with low overhead, in kernel- and user space-level execution

Table 5.1 Number of operations by type, out of a total of one million operations

	Nodes	Insertion	Join	Cycle		updateEdge
				Examined-Link[1]	Ignored-Link[2]	
$Dataset_1$	10000	4991	2503	45449	946879	178
$Dataset_2$	20000	9892	5052	76404	908556	96
$Dataset_3$	30000	15005	7496	102672	874761	66
$Dataset_4$	40000	19955	10021	125650	844322	52
$Dataset_5$	50000	24959	12519	145733	816753	36
$Dataset_6$	60000	29953	15022	164104	790885	36

[1] The new connection is evaluated by the *MST* algorithm, and one of the other edges in the cycle is ignored.
[2] The new connection has the highest weight in the cycle and is ignored by the *MST* algorithm.

Table 5.2 Number of operations which affect and update the *MST*, out of a total of one million operations

	Nodes	Examined-Link[1]	Ignored-Link[2]	Total
$Dataset_1$	10000	45449	946879	992328
$Dataset_2$	20000	76404	908556	984960
$Dataset_3$	30000	102672	874761	977433
$Dataset_4$	40000	125650	844322	969972
$Dataset_5$	50000	145733	816753	962486
$Dataset_6$	60000	164104	790885	954989

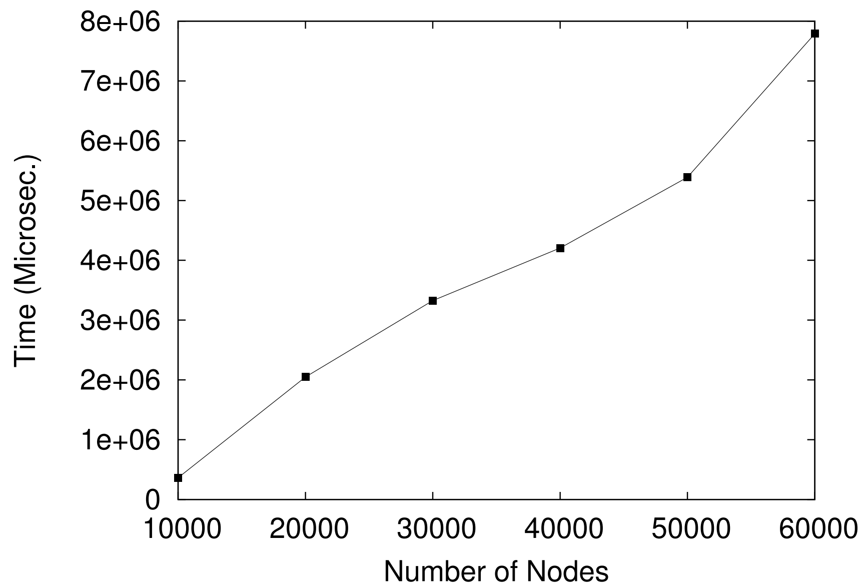


Figure 5.5 Execution time for recomputing the MST as a graph with an increasing number of updated vertices and edges. The updating sequences contain one million operations, consisting of *Insertion*, *Join*, *Cycle*, and *updateEdge*, in a forest. The proposed algorithm measured here has a time complexity of $O(log\ n)$

.

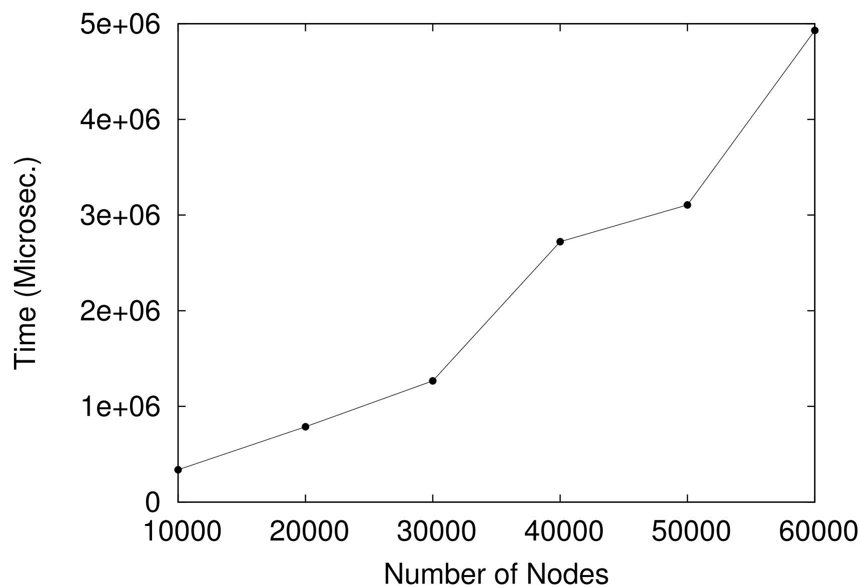


Figure 5.6 Dynamic Time RN : running time on a random graph with an increasing number of nodes plotted using an algorithm with a complexity of $O(log\ n)$. Updating sequences contained one million operations including *Insertion*, *Join*, *Cycle*, and *updateEdge*, in a dynamic network

in live streaming mode. The proposed method synchronizes the trace events from different machines in the network with a high degree of accuracy. The synchronized information can then be analyzed for many applications.

Experimental environment

For our method, we installed the Linux kernel, version $2.6.38-279.22.1.el6.centos.plus.lttng.x86_64$ (LTTng), and the tests were performed with Linux Trace Toolkit Trace Control, version 0.89-05122011. Our experimental setup consisted of 20 machines, each equipped with two Quad-Core 2.00 GHz Intel Xeon processors with 8 GB of RAM. All the systems had one 160 GB HDD and a 6144 KB cache.

We provided 12 datasets, containing traces of different duration, ranging from 3 to 25 minutes. We examined a number of different situations in this test.

Figure 5.6 shows a map of the cluster of 20 computers used for this experiment, and their 61 connections. This graph includes all the connections exercised, which were deduced from the traces, after 25 minutes had elapsed in the experiment. At that moment, node number 12 was selected as *RN*. The solid lines show the accurate connections in the current *MST*, which are used to synchronize the nodes with the current *RN*. As the other connections, shown with broken lines, did not belong to the current *MST*, they were ignored for synchronization purposes, although some may have been part of the *MST* for a while and been removed when more accurate connections appeared.

Analysis and evaluation

The 20 computers in the cluster were traced, and the results of our experiments are illustrated in Table 5.3. Twelve datasets are presented, each of which contains 20 traces from 20 machines and with durations ranging from 3 to 25 minutes. The Node Insertion column shows the number of new nodes connected to the existing trees. When two nodes connect and create a new tree, they are not counted. For example, in the Total Operations column, the first row of the Node Insertion and Tree Join columns shows 10 and 4 respectively, which indicates that five networks have been established. Each network contains two nodes initially, and then 10 other nodes connect to one of these 5 networks one by one. This is why the number shown in the Node Insertion column is 10. Five separate networks join together one by one, and eventually form one big network, so there are 4 Tree Join operations in the related column. The *MST* Changes column shows the total number of MST changes (42) during the 3 minutes of tracing.

The RN Changes column separates the reference node updates into 2 network situations :

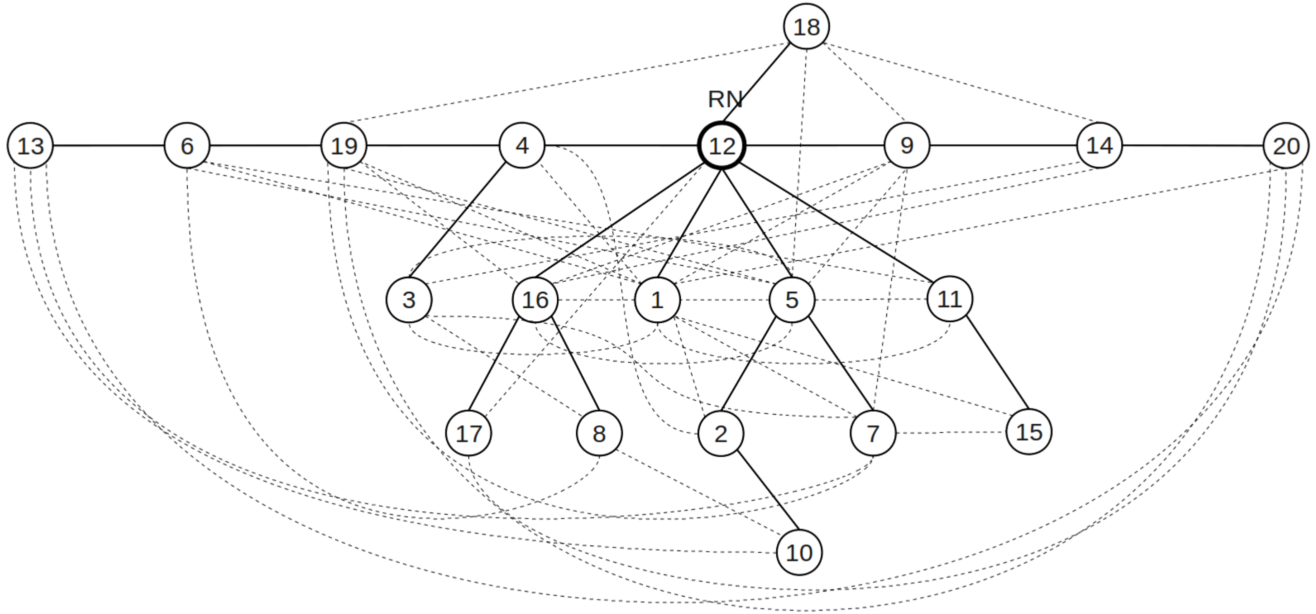


Figure 5.7 Map of the computer cluster used in the experiment

Unstable States, when not all the nodes are connected together ; and Stable States, when all the nodes connect to a single network. For instance, in the unstable state, the *RN* changes twice, once, and twice for updates respectively, as a result of Node Insertion, Tree Join, and *MST* Changes. In the stable states, the RN changes 14 times, as a result of *MST* changes, since no more Node Insertion or Tree Join operations take place once a single network exists.

We perform the experiments with three different methods of increasing sophistication and effectiveness.

– Non Incremental Method

  The Non Incremental method, presented in [85], synchronizes traces collected a posteriori from a computer network with the *Convex-Hull* algorithm. This method starts reading events only when tracing has finished. Then, it matches related packet send/receive events. *Convex-Hull*s are constructed from these packet pairs, and the time conversion parameters are computed as the middle line between the upper and lower *Convex-Hull* for each pair of communicating computers. A *MST* of the links with the smallest inaccuracy is built and a *RN* providing the best synchronization accuracy is selected. Finally, the trace from each computer has its time conversion parameters computed with respect to the *RN* through the *MST*.

  To apply this algorithm to streaming traces, we run the method on small intervals. Smaller intervals reduce the latency required to obtain the synchronization parameters,

Table 5.3 Dataset features and number of RN changes

	Trace Duration (Minutes)	Total Operations				RN Changes					
						Unstable States			Stable States		
		Node Insertion	Tree Join	MST Changes	Update Operations	Node Insertion	Tree Join	MST Changes	Node Insertion	Tree Join	MST Changes
$Dataset_1$	3	10	4	42	8141	2	1	2	0	0	14
$Dataset_2$	5	10	4	15	9926	2	1	0	0	0	3
$Dataset_3$	7	14	2	32	11657	3	1	2	0	0	5
$Dataset_4$	9	14	2	26	10227	4	2	0	0	0	3
$Dataset_5$	11	16	1	24	11780	4	1	2	0	0	11
$Dataset_6$	13	16	1	28	12793	4	1	0	0	0	4
$Dataset_7$	15	16	1	29	13067	4	1	0	0	0	7
$Dataset_8$	17	16	1	68	12293	3	0	4	0	0	24
$Dataset_9$	19	16	1	40	13670	1	0	0	0	0	4
$Dataset_{10}$	21	16	1	18	12646	1	0	1	0	0	8
$Dataset_{11}$	23	16	1	13	13195	3	0	3	0	0	5
$Dataset_{12}$	25	16	1	31	14394	3	0	1	0	0	8

but increase the total running time, since the *Convex-Hull* algorithm is run more often. Furthermore, smaller intervals contain fewer packets and have a lower synchronization accuracy. This method, when run once at the end of 25 minutes, takes 107.24 seconds. With 2-second intervals, the method is run 750 times and requires a total execution time of 141.161651 seconds. The 2-second interval was deemed the maximum value acceptable for monitoring purposes in live streaming mode.

Table 5.4 illustrates the results collected from running the Non Incremental method in streaming mode. For example, in $Dataset_{12}$ with a 25 minute trace duration, there are 750 windows. Thus, *MST*, *RN*, and *Conversion* run 750 times. Since they run with $O(n^3)$, $O(n^2)$, and $O(n\ log\ n)$ respectively, and with the results obtained for $n = 20$ (number of nodes in the network), the time consumption will grow rapidly for larger values of n and will not be suitable for live analysis.

– Incremental method

  In this second approach, the *Convex-Hull* is built incrementally. However, when the synchronization parameters are updated, the *MST* and *RN* are recomputed from scratch, as in the Non Incremental method. Table 5.5 shows that the non incremental computation is very costly for the *MST*, *RN*, and time conversion parameters. In offline trace synchronization, these three phases are computed only once, at an acceptable cost.

  For example, when 20 traces of 25 minutes' duration are synchronized with this method, as shown in the last row of Table 5.5, 31 *MST* updates take more than 23 seconds. Moreover, the *RN* selection phase takes over 500 milliseconds for just 12 updates. In total, these three phases take more than 24.5 seconds in execution time. This is for a very small network, and the time burden will rapidly become unbearable for a very large network.

– Fully Incremental method

  We now show the results of using the proposed method to synchronize the same traces. Table 5.6 illustrates that we can significantly reduce the cost of synchronizing pairs and updating the *MST*, *RN*, and time conversion parameters. For example, after 25-minute traces, as presented in the last row, a negligible delay, around 1 millisecond in total, is needed to update the *MST*, *RN*, and time conversion parameters. As could be expected, these computation times seem directly correlated to the number of changes. Indeed, in cases where there are more *MST* updates, the computation time increases.

  Figure 5.8 shows a comparison between the Fully Incremental and the Non Incremental methods, and the considerable improvement in pairwise synchronization performance. Figure 5.9 compares the two methods for the complete processing, including the up-

dating of the *MST*, *RN*, and conversion time parameters. The slopes of two methods grow differently over time, with the Non Incremental method being at least twice as fast. However, for a larger number of nodes, the difference is much more dramatic, as seen with the large simulated clusters.

Note that the number of runs is not equal in the two methods. The Non Incremental method runs at the end of each window. There are 750 windows of 2 seconds each during the 25 minutes. The Fully Incremental method runs whenever it finds an accurate packet, 3012 in this 25 minute trace set. In some cases, users may want to statically define the time reference node (RN). This brings up the interesting question of how much accuracy is gained by dynamically computing the optimal *RN*. It is interesting to see that the gain in accuracy is considerable, while the computation cost is almost negligible, as shown in Table 5.6. Figure 5.10 shows the total synchronization accuracy after 25 minutes when each of the 20 nodes is statically defined as *RN*. When the preselected node matches the dynamically selected *RN*, at the 25th minute, there is obviously no difference in accuracy at this point.

5.6.3 Discussion

Formula 5.14 represents the online synchronization cost for distributed traces :

$$
\begin{aligned}
&R = Read_events() \\
&M = Matching(R) \\
&C = Check_accurate_pair(M) /* \text{ Current connection } */ \\
&\quad or \\
&\quad Check_enough_points(M) /* \text{ New connection } */ \\
&F = Finalize_sync.(C) \\
&\quad = Cal(L_{max}) | Cal(L_{min}) + Cal(\alpha, \beta))
\end{aligned}
\tag{5.13}
$$

Table 5.4 Time evaluation with the Non Incremental method

Trace Duration (3-25 Min.)	Reading (sec.)	Matching (sec.)	Analyzing (sec.)	MST (micro-sec.)	RN (micro-sec.)	Conversion (micro-sec.)
$Dataset_1$	5.160105	2.574805	7.307259	38312	829	18793
$Dataset_2$	11.089841	2.138869	12.60183	70052	1519	21610
$Dataset_3$	15.097491	3.63815	17.221264	121066	2397	23971
$Dataset_4$	19.302863	5.292982	22.026885	152329	3201	27932
$Dataset_5$	22.531641	8.029082	26.642928	184209	4096	29210
$Dataset_6$	25.499023	9.827137	31.94265	196593	4948	30883
$Dataset_7$	30.334039	10.98167	36.183022	248372	5770	32915
$Dataset_8$	33.164242	13.167896	46.622513	300733	6735	40593
$Dataset_9$	38.76134	13.567527	47.379368	312125	7443	46232
$Dataset_{10}$	43.394643	14.412027	57.448351	325389	8287	52912
$Dataset_{11}$	47.506583	16.527386	60.294533	394914	9244	61756
$Dataset_{12}$	51.2547709	21.9105391	68.486419	409035	10060	70983

Table 5.5 The MST, RN, and conversion parameter update computation time with the Non Incremental [85] method applied on 2-second windows

Trace Duration	MST (sec.)	RN (micro-sec.)	Conversion (micro-sec.)
$Dataset_1$	13.411856	286235	142327
$Dataset_2$	16.998048	300845	183944
$Dataset_3$	17.062387	356732	235296
$Dataset_4$	15.936221	329215	264010
$Dataset_5$	17.411254	370429	260477
$Dataset_6$	17.736352	392539	297339
$Dataset_7$	19.068075	408757	305357
$Dataset_8$	18.984110	411445	322278
$Dataset_9$	19.338771	436489	346175
$Dataset_{10}$	18.734489	457006	353306
$Dataset_{11}$	21.744413	458938	365856
$Dataset_{12}$	23.867056	510076	380507

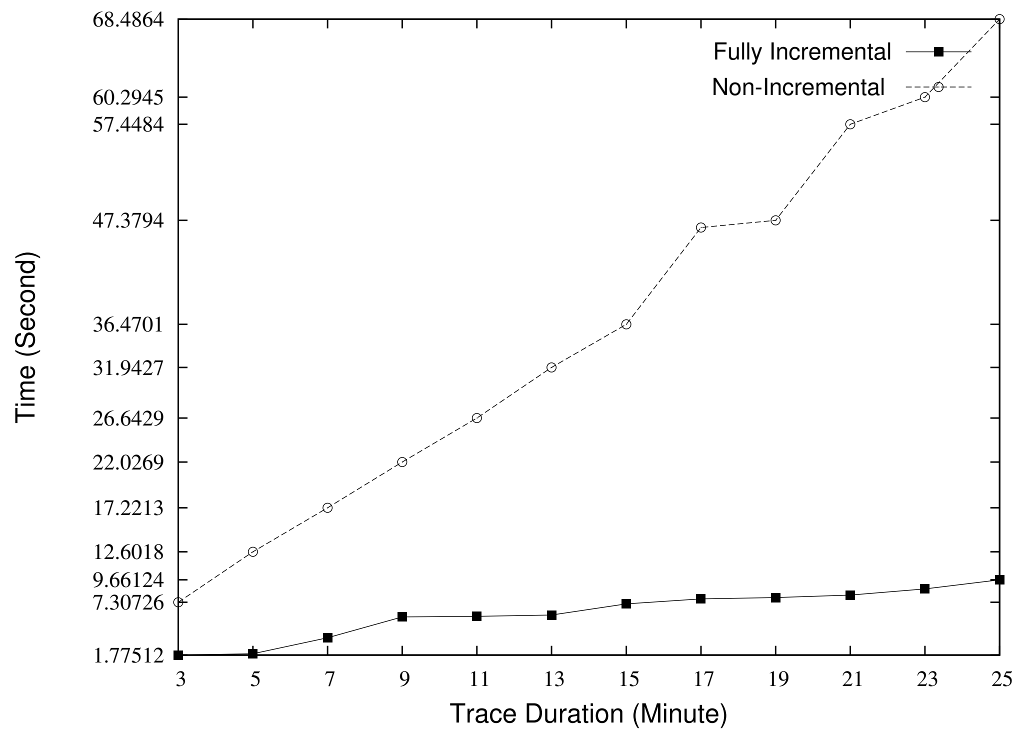


Figure 5.8 Comparison between the Fully Incremental and Non Incremental methods for pairwise computer time synchronization

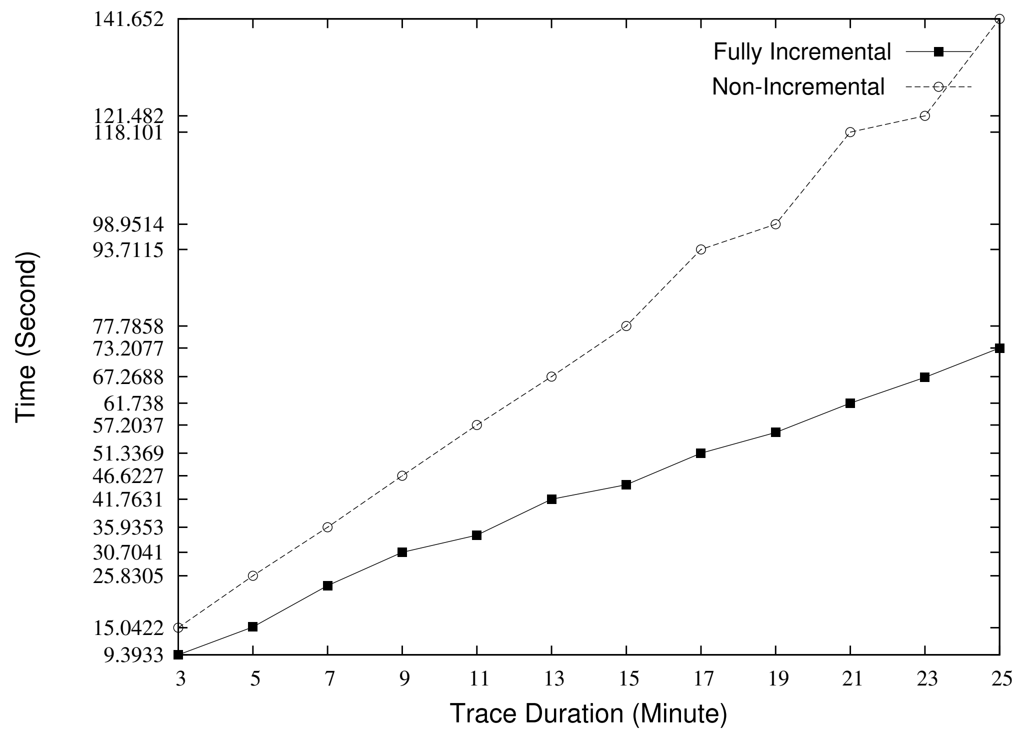


Figure 5.9 Comparison between the two methods for the complete network time synchronization computation

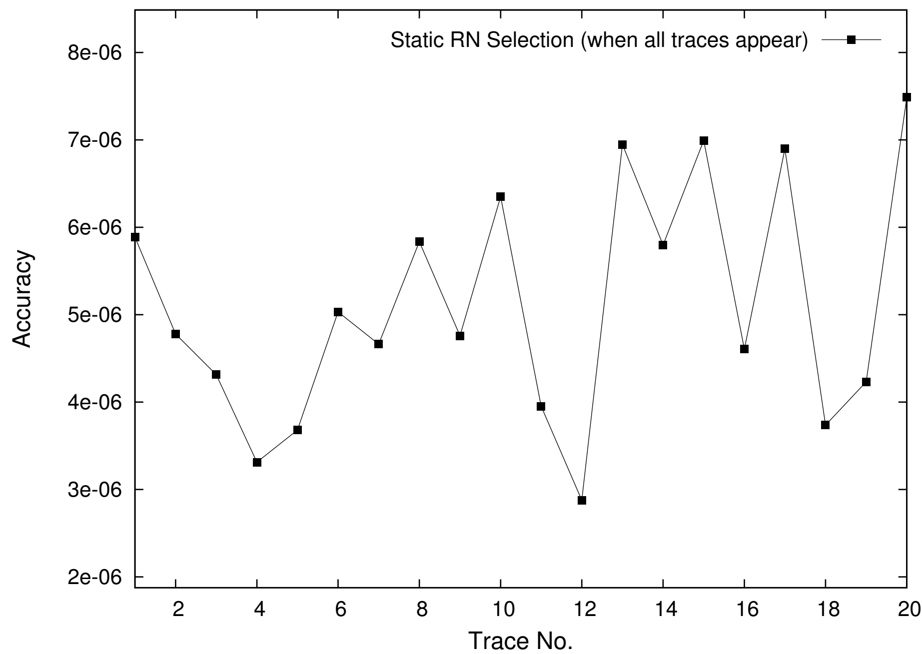


Figure 5.10 Accuracy after 25 minutes for each node statically defined as RN

Table 5.6 Decomposition of the execution time for the proposed method

Trace Duration (3-25 Min.)	No. of Accurate Packets	Total No. of RN Changes	Total No. of MST Changes	Analyzing (sec.)	MST (micro-sec.)	RN (micro-sec.)	Conversion (micro-sec.)
$Dataset_1$	21395	19	42	1.775123	899	940	120
$Dataset_2$	25769	7	15	1.921271	286	342	134
$Dataset_3$	29842	11	32	3.594395	689	587	141
$Dataset_4$	27826	9	26	5.773544	505	538	158
$Dataset_5$	31395	18	24	5.837151	491	914	166
$Dataset_6$	34555	9	28	5.976978	532	404	173
$Dataset_7$	34580	12	29	7.146467	540	626	187
$Dataset_8$	30899	31	68	7.666715	1202	1339	192
$Dataset_9$	35399	5	40	7.808655	847	213	195
$Dataset_{10}$	35200	10	18	8.06981	313	518	201
$Dataset_{11}$	37557	11	13	8.709675	264	594	206
$Dataset_{12}$	39012	12	31	9.661242	672	609	211

$$\begin{aligned}
&MST = updateMST() \\
&RN = updateRN() \\
&S = ConvertSyncFactors() \\
&Cost = Cost(R) + Cost(M) + Cost(C) + Cost(F) + \\
&\qquad Cost(MST) + Cost(RN) + Cost(S)
\end{aligned} \tag{5.14}$$

The number of events depends on the rate at which events are generated in the trace, which means that the reading time depends on the number of events (Cost(R)). The matching time (Cost(M)) does not depend on the number of events in the hash table, since it has $O(1)$ complexity.

Two cases arise in calculating the synchronization factors between two nodes. In the first case, there is a connection between the nodes. We already have synchronization factors for this link, and are awaiting new, more accurate packets to improve accuracy (Cost(C)). As mentioned earlier, in the Fully Incremental method, the procedure for the most matched packets merely involves checking the location of the point with respect to L_{max} or L_{min}, an important improvement over [63, 85]. In the second case, there is no existing connection between the nodes, and we are trying to establish the first synchronization factors for the new link. This requires at least two points in the lower and upper bounding hulls (Cost(C)).

Since only L_{max} or L_{min} is changed when an accurate pair is received, the new conversion factors are easily computed in constant time, again at a lower cost than that of the previous approach (Cost(F)).

As mentioned in the 5.5.2 section, the computation of the conversion parameters with respect to RN is performed only in a subtree of the network graph, the Minimum Spanning Tree. The Minimum Spanning Tree proposed in this work is updated incrementally throughout the sequence of updates in an amortized time of $O(log\ n)$ per update (Cost(MST)).

The reference node for the MST is updated in $O(log\ n)$, on average (Cost(RN)). The MST with a time reference node forms a tree of the best path to relate each node's clock to the reference node based on the link's accuracy. We have proposed an efficient algorithm to incrementally compute the MST and RN, and to propagate the conversion parameters (Cost(S)).

5.7 Conclusion

The online synchronization of distributed traces is important for the live diagnosis of complex problems in distributed systems. The existing methods for trace synchronization cannot efficiently address the problem of live synchronization. As a first step, an incremental approach for synchronizing each communication link was presented. This method requires $O(1)$ time, on average, to process each new network packet and update the synchronization factors, while maintaining optimal accuracy without latency.

New algorithms were also presented to build on the link level synchronization factors and incrementally compute a Minimum Spanning Tree formed by the most accurate links, as well as to find the best time reference node. The new approach presented takes a time of $O(log\ n)$ on average, when the synchronization factors for a link change.

In summary, this new approach efficiently updates the synchronization parameters incrementally, without sacrificing the accuracy of the results or delaying their computation. Furthermore, it is shown to scale extremely well to large networks and traces of long duration. This makes our scheme applicable to all types of online time synchronization.

CHAPTER 6

GENERAL DISCUSSION

New online applications bring new time synchronization needs : online operation, and high accuracy and performance. Traditional clock synchronization algorithms relied on explicit time synchronization network messages. Not only does that add supplementary traffic to the network but, more importantly for tracing and monitoring tools, it needlessly modifies the behavior of the network and nodes being traced.

The main motivation for the new algorithms contributed and presented in this thesis is to optimize traces synchronization for online operation in very large networked clusters. The *LTTng* kernel and user-space tracers can collect extensive traces of execution events on each node with very fine granularity. The traces can then be accumulated at each node for later analysis, adding a small burden to the disk subsystem. Alternatively, they may be streamed to an analysis host on the network. In that case, the traces streaming will add to the network traffic, but using efficient batching of events and compact binary encoding. At least, it does not require, on top of the streaming traces, additional time synchronization messages and running time synchronization daemons on every host.

The events collected at each node are timestamped with a granularity in the order of the nanosecond. Successive events in a trace are sometimes separated one from another by less than a microsecond. The aim is therefore to achieve highly accurate traces synchronization to be able to compare, on a common time referential, events with such time granularity. The second challenge adressed in this thesis is to enable online traces synchronization for live monitoring and analysis. This requires the ability to synchronize on the fly multiple traces without needing extensive buffering. Finally, the third goal of this work was to insure the scalability, to huge clusters, of these online algorithms.

In Chapter 2, a comprehensive review of the existing approaches was presented. It clearly presented the strengths and limitations of each approach and explained why none of the existing time synchronization methods could provide the accuracy and performance required for distributed online tracing in very large clusters. This evaluation and comparison is presented in the form of a survey paper to help designers choosing a practical protocol matching their application. Existing surveys predate large clusters or focus on specialized areas such as wireless sensor networks. Wireless sensor networks have become a field in itself with very specific requirements, for instance in terms of low power and self configuration.

Chapter 3 is a research article proposing a new and extremely efficient incremental ap-

proach that can quickly process packet send and receive events, and update when needed the offset and drift values associated with a connection. Every received packet is checked with a simple formula to determine if it can improve the synchronization accuracy. If not, it is simply ignored. In this way, this method minimizes the number of time points requiring further processing or buffering space. When the received packet lies between the L_{max} and L_{min} lines, it can improve the synchronization accuracy between those two nodes. Therefore, the algorithm takes it into account and updates incrementally the synchronization parameters. Since the packet evaluation and synchronization update takes $O(1)$ time, this method can easily process high volume traces on the fly. Nonetheless, this method retains the accuracy of the *Convex-Hull* approach, and insures the best synchronization accuracy. Thus, this new method is ideally suited for online traces synchronization, addressing both synchronization speed and accuracy requirements.

Any time synchronization algorithm in streaming mode has to be robust even in the presence of network delay variations, internal kernel delays and even lost packets. The main concern is whether such erratic network or system behavior can adversely impact the synchronization algorithm speed, accuracy or reliability.

To test the effect of lost packets, three different traffic tests with 10, 20 and 30 percent lost packets have been conducted between two computers. The packet loss ratio is set as the fraction of packets randomly dropped by iptables. An increasing packet loss ratio models an increasing network congestion. The direct effect is to make packet matching more difficult (finding the corresponding packet send and receive events), having many packets sent but never received. It also reduces the number of packets available for synchronization. The results in this paper illustrate that even with 30 percent packet loss, the fully incremental approach provides appropriate synchronization accuracy. In fact, in spite of the high packet loss ratio, our approach could reach an acceptable precision even with few accurate packets.

Another interesting consideration is how to cope with incremental adjustments to the estimated offset and drift values. When a monitoring tool is currently showing traces from two nodes, it may happen that the exact scale and alignment of one trace with respect to the other gets updated following the reception of more accurate packets. One possibility is to simply redraw the trace with the updated, improved, offset and drift values. However, from an ergonomic point of view, users may feel more comfortable if the already shown portion of the trace is not changed, (i.e. the past is not rewritten), but the view gradually transitions the scale and alignment of the trace to the updated values, when continuing to draw the trace.

Another issue is the initial delay to obtain the first estimation of the offset and drift, when a synchronized view is required as soon as possible for online traces. The synchronization ac-

curacy improves over time as new accurate packets are received. The first values are obtained from as little as 4 packets, enough to define the two lines (L_{max} and L_{min}). Thereafter, as accurate packets are received, the slope of either L_{max} or L_{min} changes, narrowing the gap and improving accuracy. At the beginning, the rate of accurate packets is much higher, the initial values have a relatively high error margin, but are improved upon quickly as more packets are received. After a while, the gap between L_{max} and L_{min} becomes much smaller, as the accuracy increases, and the chances of receiving a packet placed between L_{max} and L_{min} diminish. Thus, the synchronization update rate is high at the start, and decreases rapidly over time. If such updates are discomforting for the users of a live graphical view, a strategy could be to delay the initial drawing by a few seconds, until a sufficient level of accuracy is obtained.

The impact of network traffic on the synchronization performance was observed with a particularly interesting experiment run on the Mammouth cluster. Mammouth is one of the largest Linux clusters in Canada, and is located at the Centre de Calcul Scientifique in Sherbrooke University, funded by the RQCHP [1]. The results have shown that the accuracy depends not only on the traffic volume, but also on the latency of the traffic. Thus, when the latency of the network is smaller, even with low traffic volume, the accuracy will be higher [56].

For our target applications, the online time synchronization algorithms needed to scale to huge computer clusters, and to grid and cloud environments. To this end, the pairwise node synchronizations needed to be combined into network level synchronization, in the presence of a large number of dynamic nodes entering or leaving the network. Indeed, a cluster tracing and monitoring system needs to analyze and maintain a live tracing view in a dynamic network. Chapter 5 presents a new technique to efficiently and incrementally build a Minimum Spanning Tree of links with the best synchronization accuracy. This tree may be used to quickly compute the offset and drift, between any pair of nodes in the network, by navigating the tree and composing the drift and offset values along the path between the two nodes. The idea was to minimize the amount of recomputation needed at each update. The Minimum Spanning Tree updates are based on the very efficient splay tree structure.

To complete the network level synchronization, for instance to show a live tracing view of the activity at several nodes, the final step is to select a reference node to act as time referential. In small to medium networks, the user may sometimes want to select manually the reference node. However, in very large networks, the selection of the reference node is better left automated. Indeed, with a static reference node, the total synchronization error increases because the reference node may not be the optimal one, in the updated synchronization graph. Moreover, it is generally impractical to have a statically fixed reference node in a

dynamic network where any node may come and go.

A new and efficient algorithm was proposed in Chapter 4 to dynamically select and update the reference node in a synchronization Minimum Spanning Tree. The proposed schema dynamically maintains the reference node selection at a minimal cost in processing time. In some tests, it achieved a tenfold accuracy improvement over predefining a specific node as reference. We evaluated our algorithms in dynamic simulated networks with six data sets, each containing one million operations affecting graphs with between 10,000 to 60,000 vertices. It would however be interesting to test and evaluate our algorithms in real-world dynamic networks, such as those formed by the largest data centers in operation on the Internet.

As for the updates to the drift and offset parameters, changing the reference node should not create a disturbing discontinuity in the online view of traces. To this end, in most cases, it will be better not to change already drawn timelines of traces. Moreover, the transition to the time of the new reference node should be gradual and almost imperceptible to the viewer. A temporary offset, representing the time difference between the old and new reference node, can be maintained to insure continuity, and be gradually decreased over time.

The algorithms presented in this thesis have been incorporated into the Eclipse Tracing and Monitoring Framework (TMF) [7] and will appear in upcoming versions. TMF is used throughout the world by large high technology companies such as Ericsson, Intel, Mentor and WindRiver. This confirms that the new algorithms proposed here are not only original, computationally efficient and elegant, they solve actual industrial problems and bring new efficiency to real applications.

CHAPTER 7

CONCLUSION

This chapter presents a few concluding remarks, summarizes the original contributions in this thesis, and proposes possible directions for future research in distributed trace synchronization.

7.1 Concluding Remarks

In this thesis, we investigated several issues that are crucial for tracing when deploying real-time distributed systems. We studied approaches for online synchronization of distributed traces in order to monitor distributed systems and diagnose complex problems. Existing approaches based on Convex-Hulls achieve excellent accuracy for a posterior analysis, but impose a significant cost and latency when used in live mode and over large clusters.

Our objective was to maintain the same synchronization accuracy, without latency for the availability of updated synchronization parameters, while providing an extremely efficient incremental algorithm. Indeed, in a large network, traced events are generated at an enormous rate. We initially relied on repeated applications of the *Convex-Hull* algorithm because of its accuracy. We then used several models using time intervals to support live analysis of stream data. Storing events through a time interval, and then applying the *Convex-Hull* algorithm at the end of the interval, provides accurate results but only at the end of the interval. Furthermore, the repeated application of the *Convex-Hull* is inefficient, and storing events for the interval needlessly consumes memory for the buffers.

We then proposed a novel and efficient method to compute synchronization factors incrementally. It calculates the clock offset and drift, and provides updates as packets are exchanged along a communication link. This method observes every exchanged packet and quickly verifies a basic and simple condition to identify accurate packets. It then uses that packet to incrementally update the synchronization parameters in $O(1)$ time. Therefore, this proposed method avoids needlessly storing packets, and very few packets (accurate packets) actually require further computation and are kept to build the *Convex-Hull*.

The second important contribution of this thesis was to expand live tracing from the link to the network level. Hence, we proposed a new method to build upon the link level synchronization parameters, and incrementally compute a Minimum Spanning Tree formed by the most accurate links. Here, the goal was to achieve the best performance in a dynamic

network where computers connect/disconnect to/from the network and thus update the *MST* efficiently. The presented approach takes $O(log\ n)$ time on average (where n represents the total number of computers in the network) when the synchronization factors for a link change sufficiently to affect the *MST*.

Finally, as third important contribution, we proposed a new algorithm to update the best time reference node incrementally upon network changes. This method provides an efficient way to find and maintain a Reference Node incrementally in an average time complexity of $O\ (log\ n)$, where n is the total number of nodes in the network. In this work, the online analysis of new vertex insertion, tree merging, and cycle handling in a forest are handled. The previous method suffers from the cost of checking the whole forest, even when there has been no change, after each tree modification. After a large number of modifications have been made, and a sparse forest has grown into a large tree, the performance improvements brought by the new approach are even greater.

Our work focused on the fact that, for most application, scalability, accuracy and performance are the main objectives. The new approaches presented can efficiently and incrementally update the synchronization parameters, without sacrificing the accuracy of the results or delaying their computation. The proposed methods were tested on extremely large clusters, a real network containing more than 55 physical computers and a simulated network with 60,000 nodes. Furthermore, long duration traces were generated and analyzed to demonstrate the scalability with respect to trace size and time duration. The results showed that the presented work achieves the accuracy, performance and scalability objectives. These algorithms are thus applicable to all types of online time synchronization.

7.2 Future Research

Our online time synchronization technique can be applied to synchronize several types of traces where accuracy, performance and scalability are required. Although we have only provided results for tracing networked computers, we can extend this method to many data synchronization applications where messages are exchanged. It also can be used effectively in distributed system applications where online applications need accurate data adjustments. For example, in fault tolerant redundant storage applications, our method may be used on every redundant storage server to analyze the history on each node and improve adjustments such as conflict resolutions [60]. This work also can be useful to synchronize events coming from virtual machines on a physical system. This will be discussed in the next Section.

7.2.1 Data integration from Virtual Machine

An increasingly important aspect of trace synchronization is virtual machines (*VM*) tracing. For example, it is very useful to trace complete systems, Linux host and *KVM* guests, to represent graphically the interactions between the host and the guest(s). Each virtual machine is a guest software environment, which consists of an operating system and many software applications. Multiple virtual machines may run on a physical host and each *VM* has its own view of the system time, stored internally as an offset to the Time Stamp Counter (*TSC*). Since the offset to the underlying *TSC* (physical machine *TSC*) can change during operations such as CPU migration, physical machine migration or pause/resume, we need to keep track of the *TSC* offset every time it changes, in order to synchronize the traces. Physical machine migration happens when a virtual machine instance is moved to new physical machine while it is running. It can happen based on an administrative decision to control load balancing, for security purposes and etc. Moreover, a virtual machine may be paused and then resumed after a while. In these two scenarios, a virtual machine should see the time growing monotonically. Consequently, for synchronization purposes, a virtual machine trace may need to be realigned multiple times since there may be more than one offset for a particular *TSC*. Hence, the *TSC* offset on the physical machine has to be updated for these operations. Multi-level traces are required to trace a *VM*. First, tracing is performed at the physical host user-space level (*QEMU* for *KVM*) [75], secondly at the physical host kernel-space level, thirdly at the virtual machine kernel-space level and finally at the virtual machine user-space applications level. All these traces are gathered to fully understand the operation of a virtual machine. To record a consistent trace across these layers efficiently, multi-level time synchronization is essential [18]. Traces recorded in the host and in the guest are not directly aligned since there are multiple time sources in multi-level tracing :

- *TSC* begins at machine boot time
- *LTTng* relies mostly on the *TSC* (cycles + freq)
- *UST* relies on clock_gettime *vDSO* (sec.nsec) : by default it returns the linux uptime which starts when linux boots instead of when the machine boots
- Offset in the traces kernel/user-space : those have been synchronized already using the trace clock in *LTTng*

There are three types of offsets to the *TSC* : 1) Linux boot offset, 2) virtual machine boot offset 3) Linux boot offset on the virtual machine. To tackle this problem, an efficient *TSC* based clock source is required. An efficient *TSC* ensures that the *TSC* is synchronized across all cores, host kernel/user space and guest(s) kernel/user space [74].

7.2.2 Hardware tracing

A number of newer processors now contain hardware-assisted tracing. This has been offered in several ARM processors for some time [3] and similar facilities have been announced for Intel processors recently [6]. The problem here again is synchronizing these hardware generated traces with the kernel and UST software generated traces. For example, the Intel Branch Trace Store (BTS) provides a trace of all control flow changes on a CPU, but without timestamps. When a BTS buffer is full, the associated interrupt service routine can record the current time, leading to synchronization points at each buffer switch. Analyzing the control flow and the relative size of code blocks may be used to estimate interpolated values for each element in this branch trace. Different hardware tracing facilities will present similar challenges and constitute an extremely interesting and useful field for future research.

LIST OF REFERENCES

[1] (2010). Réseau québécois de calcul de haute performance. https://rqchp.ca/?mod=cms&pageId=0&lang=EN&. [Online ; accessed 22-June-2010].

[2] (2012). Time and frequency from a to z : A to al. http://tf.nist.gov/general/glossary.htm. [Online ; accessed 18-October-2012].

[3] (2013). Arm : The architecture for the digital world. http://www.arm.com/products/processors/technologies/biglittleprocessing.php. [Online ; accessed 20-September-2013].

[4] (2013). Dtrace : Dynamic tracing framework. http://dtrace.org/blogs/. [Online ; accessed 20-September-2013].

[5] (2013). Ftrace : the linux kernel internal tracer. http://elinux.org/Ftrace. [Online ; accessed 20-September-2013].

[6] (2013). Intel® processor comparison. http://www.intel.com/content/www/us/en/processor-comparison/compare-intel-processors.html. [Online ; accessed 20-October-2013].

[7] (2013). Linux tools project - lttng integration (tracing and monitoring framework). http://www.eclipse.org/linuxtools/projectPages/lttng/. [Online ; accessed 2-October-2013].

[8] (2013). Lttng project : Linux trace toolkit-next generation. http://lttng.org/. [Online ; accessed 20-September-2013].

[9] (2013). Systemtap. http://sourceware.org/systemtap/. [Online ; accessed 20-September-2013].

[10] AKYILDIZ, I. F., MELODIA, T. and CHOWDURY, K. R. (2007). Wireless multimedia sensor networks : A survey. *Wireless Communications, IEEE*, 14, 32–39.

[11] AKYILDIZ, I. F., SU, W., SANKARASUBRAMANIAM, Y. and CAYIRCI, E. (2002). A survey on sensor networks. *Communications magazine, IEEE*, 40, 102–114.

[12] ARVIND, K. (1994). Probabilistic clock synchronization in distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 5, 474–487.

[13] ASHLEY, S. (2011). Introduction to amba® 4 ace™ and big.little™ processing technology. Rapport technique, ARM.

[14] ASHTON, P. (1995). Algorithms for off-line clock synchronization. Rapport technique, Department of Computer Science, University of Canterbury.

[15] BEAMONTE, R., GIRALDEAU, F. and DAGENAIS, M. (2012). High Performance Tracing Tools for Multicore Linux Hard Real-Time Systems. *Proceedings of the 14th Real-Time Linux Workshop*. OSADL.

[16] BETTI, E., CESATI, M., GIOIOSA, R. and PIERMARIA, F. (2009). A global operating system for hpc clusters. *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 1–10.

[17] BLIGH, M., DESNOYERS, M. and SCHULTZ, R. (2007). Linux kernel debugging on google-sized clusters. *Proceedings of the Linux Symposium*. 29–40.

[18] BLUNCK, J., DESNOYERS, M. and FOURNIER, P.-M. (2009). Userspace application tracing with markers and tracepoints. *Proceedings of the 2009 Linux Kongress*.

[19] CATTANEO, G., FARUOLO, P., PETRILLO, U. F. and ITALIANO, G. (2010). Maintaining dynamic minimum spanning trees : An experimental study. *Discrete Applied Mathematics*, 158, 404 – 425.

[20] CHAI, L., GAO, Q. and PANDA, D. K. (2007). Understanding the impact of multi-core architecture in cluster computing : A case study with intel dual-core system. *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*. IEEE, 471–478.

[21] CHAN, E. M., CARLYLE, J. C., DAVID, F. M., FARIVAR, R. and CAMPBELL, R. H. (2008). Bootjacker : compromising computers using forced restarts. *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 555–564.

[22] CLEMENT, E. and DAGENAIS, M. (2009). Traces synchronization in distributed networks. *Journal of Computer Systems, Networks, and Communications*, 2009, 5.

[23] CORMEN, T. H., STEIN, C., RIVEST, R. L. and LEISERSON, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, seconde édition.

[24] COULOURIS, G. F. and DOLLIMORE, J. (1988). *Distributed systems : concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[25] CRISTIAN, F. (1989). Probabilistic clock synchronization. *Distributed computing*, 3, 146–158.

[26] CUCINOTTA, T. and FAGGIOLI, D. (2010). An exception based approach to timing constraints violations in real-time and multimedia applications. *Industrial Embedded Systems (SIES), 2010 International Symposium on*. IEEE, 136–145.

[27] DE MELO, A. C. (2010). The new linux'perf'tools. *Slides from Linux Kongress*.

[28] DESCHÊNES, J.-H., DESNOYERS, M. and DAGENAIS, M. R. (2008). Tracing time operating system state determination. *The Open Software Engineering Journal*, 40–44.

[29] DESNOYERS, M. (2009). *Low-impact operating system tracing.* Thèse de doctorat, École Polytechnique de Montréal.

[30] DESNOYERS, M. and DAGENAIS, M. (2008). Lttng : Tracing across execution layers, from the hypervisor to user-space. *Linux Symposium*. 101.

[31] DESNOYERS, M. and DAGENAIS, M. (2009). Deploying lttng on exotic embedded architectures. *ELC (Embedded Linux Conference).*

[32] DIETZ, M. A. (1996). *Gathering And Using Time Measurements In Distributed Systems.* Thèse de doctorat, Duke University.

[33] DOLESCHAL, J., KNÜPFER, A., MÜLLER, M. S. and NAGEL, W. E. (2008). Internal timer synchronization for parallel event tracing. *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, Berlin, Heidelberg, 202–209.

[34] DOLEV, D., LYNCH, N. A., PINTER, S. S., STARK, E. W. and WEIHL, W. E. (1986). Reaching approximate agreement in the presence of faults. *J. ACM*, 33, 499–516.

[35] DOMINGOS, P. and HULTEN, G. (2000). Mining high-speed data streams. *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, New York, NY, USA, KDD '00, 71–80.

[36] DOMMETY, G. and JAIN, R. (1998). Potential networking applications of global positioning systems (gps). *arXiv preprint cs/9809079.*

[37] DUDA, A., HARRUS, G., HADDAD, Y. and BERNARD, G. (1987). Estimating global time in distributed systems. *ICDCS*. vol. 87, 299–306.

[38] EIDSON, J. and LEE, K. (2002). Ieee 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*. IEEE, 98–105.

[39] ELLINGSON, C. and KULPINSKI, R. (1973). Dissemination of system time. *Communications, IEEE Transactions on*, 21, 605–624.

[40] ELSON, J., GIROD, L. and ESTRIN, D. (2002). Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36, 147–163.

[41] FAN, R. and LYNCH, N. (2006). Gradient clock synchronization. *Distributed Computing*, 18, 255–266.

[42] FIDGE, C. J. (1988). Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*. vol. 10, 56–66.

[43] FONSECA, R. L. C. (2009). *Improving visibility of distributed systems through execution tracing*. ProQuest.

[44] GABER, M. M., ZASLAVSKY, A. and KRISHNASWAMY, S. (2005). Mining data streams : a review. *SIGMOD Rec.*, 34, 18–26.

[45] GANERIWAL, S., KUMAR, R. and SRIVASTAVA, M. B. (2003). Timing-sync protocol for sensor networks. *Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM, New York, NY, USA, SenSys '03, 138–149.

[46] GOODMAN, D. J., VALENZUELA, R. A., GAYLIARD, K. and RAMAMURTHI, B. (1989). Packet reservation multiple access for local wireless communications. *Communications, IEEE Transactions on*, 37, 885–890.

[47] GUSELLA, R. and ZATTI, S. (1989). The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3 bsd. *Software Engineering, IEEE Transactions on*, 15, 847–853.

[48] HAN, J. (2005). *Data Mining : Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[49] HAN, L. and HUA, N. (2013). A distributed time synchronization solution without satellite time reference for mobile communication. *Communications Letters, IEEE*, 17, 1447–1450.

[50] HE, L.-M. (2008). Time synchronization based on spanning tree for wireless sensor networks. *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM'08. 4th International Conference on*. IEEE, 1–4.

[51] HENZINGER, M. R. and KING, V. (2001). Maintaining minimum spanning forests in dynamic graphs. *SIAM J. COMPUT*, 31, 2001.

[52] HOLM, J., DE LICHTENBERG, K. and THORUP, M. (2001). Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48, 723–760.

[53] JABBARIFAR, M. and DAGENAIS, M. (2013). A comprehensive survey of techniques and challenges in distributed systems time synchronization. *Submitted to the Journal of Network and Computer Applications*.

[54] JABBARIFAR, M. and DAGENAIS, M. (2013). Liana : Live incremental time synchronization of traces for distributed systems. *Submitted to the Journal of Network and Computer Applications*.

[55] JABBARIFAR, M. and DAGENAIS, M. (2013). Reference node selection in dynamic tree. *Submitted to the Journal of Network Management*.

[56] JABBARIFAR, M., DAGENAIS, M., ROY, R. and SENDI, A. S. (2012). Optimum off-line trace synchronization of computer clusters. *Journal of Physics : Conference Series*. IOP Publishing, vol. 341, 012029.

[57] JABBARIFAR, M., DAGENAIS, M. and SENDI, A. S. (2013). Streaming mode incremental clock synchronization. *Submitted to the Journal of Network and Systems Management (JONS).*

[58] JABBARIFAR, M., SENDI, A. S., PEDRAM, H., DEHGHAN, M. and DAGENAIS, M. (2010). L-sync : Larger degree clustering based time-synchronisation for wireless sensor network. *Software Engineering Research, Management and Applications (SERA), 2010 Eighth ACIS International Conference on*. IEEE, 171–178.

[59] JABBARIFAR, M., SENDI, A. S., SADIGHIAN, A., JIVAN, N. E. and DAGENAIS, M. (2010). A reliable and efficient time synchronization protocol for heterogeneous wireless sensor network. *Wireless Sensor Network*, 2.

[60] JÉZÉQUEL, J.-M. and JARD, C. (1996). Building a global clock for observing computations in distributed memory parallel computers. *Concurrency : Practice and Experience*, 8, 71–89.

[61] JOSEPH, J. and FELLENSTEIN, C. (2003). *Grid Computing*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[62] KELTCHER, C. N., MCGRATH, K. J., AHMED, A. and CONWAY, P. (2003). The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23, 66–76.

[63] KHLIFI, H. and GRÉGOIRE, J.-C. (2006). Low-complexity offline and online clock skew estimation and removal. *Computer Networks*, 50, 1872–1884.

[64] KOCH, B., KOCH, R., MOSER, L. E. and MELLIAR-SMITH, P. M. (1998). Timestamp acknowledgments for determining message stability. *In Proceedings of the 2nd International Conference on Parallel and Distributed Computing and Networks.*

[65] KSHEMKALYANI, A. D. (2004). The power of logical clock abstractions. *Distributed Computing*, 17, 131–150.

[66] KSHEMKALYANI, A. D. and SINGHAL, M. (2008). *Distributed Computing : Principles, Algorithms, and Systems*. Cambridge University Press, New York, NY, USA, première édition.

[67] KUHN, F., LENZEN, C., LOCHER, T. and OSHMAN, R. (2010). Optimal gradient clock synchronization in dynamic networks. *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, New York, NY, USA, PODC '10, 430–439.

[68] KUHN, F., LOCHER, T. and OSHMAN, R. (2011). Gradient clock synchronization in dynamic networks. *Theory of Computing Systems*, 49, 781–816.

[69] LAMPORT, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, 558–565.

[70] LANDES, T. (2007). Tree clocks : an efficient and entirely dynamic logical time system. *Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference : parallel and distributed computing and networks*. ACTA Press, Anaheim, CA, USA, PDCN'07, 375–380.

[71] LEICK, A. (2004). *GPS satellite surveying*. Wiley. com.

[72] LEMMON, M. D., GANGULY, J. and XIA, L. (2000). Model-based clock synchronization in networks with drifting clocks. *Dependable Computing, 2000. Proceedings. 2000 Pacific Rim International Symposium on*. IEEE, 177–184.

[73] LISKOV, B. (1993). Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, 6, 211–219.

[74] MAROUANI, H. and DAGENAIS, M. R. (2005). Comparing high resolution timestamps in computer clusters. *Electrical and Computer Engineering, 2005. Canadian Conference on*. IEEE, 400–403.

[75] MAROUANI, H. and DAGENAIS, M. R. (2008). Internal clock drift estimation in computer clusters. *J. Comp. Sys., Netw., and Comm.*, 2008, 9 :1–9 :7.

[76] MATTERN, F. (1989). Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1, 215–226.

[77] MILLS, D. (1992). Modelling and analysis of computer network clocks. *Electrical Engineering Department Report*, 9252.

[78] MILLS, D. (1992). Network time protocol (version 3) specification, implementation and analysis.

[79] MILLS, D. L. (1991). Internet time synchronization : the network time protocol. *Communications, IEEE Transactions on*, 39, 1482–1493.

[80] MILLS, D. L. (1997). Computer network time synchronization. *Report Dagstuhl Seminar on Time Services Schloß Dagstuhl, March 11.–March 15. 1996*. Springer, vol. 12, 332.

[81] MOLKA, D., HACKENBERG, D., SCHONE, R. and MULLER, M. S. (2009). Memory performance and cache coherency effects on an intel nehalem multiprocessor system. *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 261–270.

[82] MOON, S. B., SKELLY, P. and TOWSLEY, D. (1999). Estimation and removal of clock skew from network delay measurements. *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.* IEEE, vol. 1, 227–234.

[83] OLSON, A. and SHIN, K. G. (1994). Fault-tolerant clock synchronization in large multicomputer systems. *Parallel and Distributed Systems, IEEE Transactions on*, 5, 912–923.

[84] PAPAKIPOS, M. (2007). The peakstream platform : High-productivity software development for multi-core processors. *Proceedings of Windows Hardware Engineering Conference (WinHEC), Industry Papers.*

[85] POIRIER, B., ROY, R. and DAGENAIS, M. (2010). Accurate offline synchronization of distributed traces using kernel-level events. *SIGOPS Oper. Syst. Rev.*, 44, 75–87.

[86] RAMANATHAN, P., SHIN, K. G. and BUTLER, R. W. (1990). Fault-tolerant clock synchronization in distributed systems. *Computer*, 23, 33–42.

[87] RIDOUX, J., VEITCH, D. and BROOMHEAD, T. (2012). The case for feed-forward clock synchronization. *Networking, IEEE/ACM Transactions on*, 20, 231–242.

[88] SALYERS, D., STRIEGEL, A. and POELLABAUER, C. (2008). A light weight method for maintaining clock synchronization for networked systems. *Computer Communications and Networks, 2008. ICCCN'08. Proceedings of 17th International Conference on.* IEEE, 1–5.

[89] SCHEUERMANN, B., KIESS, W., ROOS, M., JARRE, F. and MAUVE, M. (2009). On the time synchronization of distributed log files in networks with local broadcast media. *IEEE/ACM Trans. Netw.*, 17, 431–444.

[90] SCHMID, U. (1994). Synchronized utc for distributed real-time systems. *Annual Review in Automatic Programming*, 18, 101–107.

[91] SENDI, A. S., JABBARIFAR, M., SHAJARI, M. and DAGENAIS, M. (2010). Femra : fuzzy expert model for risk assessment. *Internet Monitoring and Protection (ICIMP), 2010 Fifth International Conference on.* IEEE, 48–53.

[92] SHAMELI SENDI, A. and DAGENAIS, M. (2013). Arito : Cyber-attack response system using accurate risk impact tolerance. *International Journal of Information Security.*

[93] SHAMELI SENDI, A., DAGENAIS, M., JABBARIFAR, M. and COUTURE, M. (2012). Real time intrusion prediction based on optimized alerts with hidden markov model. *Journal of Networks*, 7, 311–321.

[94] SHAMELI-SENDI, A., EZZATI-JIVAN, N., JABBARIFAR, M. and DAGENAIS, M. (2012). Intrusion response systems : survey and taxonomy. *International Journal Computer Science Network Security (IJCSNS). i1*, 12, 1–14.

[95] SIRDEY, R. and MAURICE, F. (2008). A linear programming approach to highly precise clock synchronization over a packet network. *4OR*, 6, 393–401.

[96] SIVRIKAYA, F. and YENER, B. (2004). Time synchronization in sensor networks : a survey. *Network, IEEE*, 18, 45–50.

[97] SLEATOR, D. D. and ENDRE TARJAN, R. (1983). A data structure for dynamic trees. *Journal of computer and system sciences*, 26, 362–391.

[98] SLEATOR, D. D. and TARJAN, R. E. (1985). Self-adjusting binary search trees. *J. ACM*, 32, 652–686.

[99] SUNDARARAMAN, B., BUY, U. and KSHEMKALYANI, A. D. (2005). Clock synchronization for wireless sensor networks : a survey. *Ad Hoc Networks*, 3, 281–323.

[100] TANENBAUM, A. S. and STEEN, M. V. (2006). *Distributed Systems : Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[101] VAN GREUNEN, J. and RABAEY, J. (2003). Lightweight time synchronization for sensor networks. *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. ACM, New York, NY, USA, WSNA '03, 11–19.

[102] VEITCH, D., RIDOUX, J. and KORADA, S. B. (2009). Robust synchronization of absolute and difference clocks over networks. *IEEE/ACM Transactions on Networking (TON)*, 17, 417–430.

[103] XUAN, B. B., FERREIRA, A. and JARRY, A. (2003). Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14, 267–285.

[104] ZHANG, L., LIU, Z. and HONGHUI XIA, C. (2002). Clock synchronization algorithms for network measurements. *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. IEEE, vol. 1, 160–169.