Improved Techniques for Time Synchronisation over WiFi and Wireless Sensor Networks

Jonathan Shannon, B.Sc.

Discipline of Information Technology

College of Engineering and Informatics

National University of Ireland, Galway

Head of Discipline: Dr. Michael Madden

Dean of College: Prof. Gerry Lyons

Research Supervisor: Dr. Hugh Melvin

Internal Examiner: Prof. Gerry Lyons

External Examiner: Dr. Marc Weiss

This thesis is submitted to NUI Galway for the degree of

Ph.D.

Acknowledgements

The pursuit of this research doctorial degree has been a long, exciting and, at times, rather difficult journey. Without the support and encouragement of the following people it would not have been possible.

- My supervisor Dr. Hugh Melvin for his continuous guidance throughout. His thorough knowledge, expertise and, in particular, his attention to detail has undoubtedly improved every aspect of my research. His positivity was very much welcomed during the more challenging times which encouraged me to persevere. Also a man with whom I share a somewhat similar background, he has been the source of countless entertaining stories about rural life and hopefully many more to come.
- Each and every one of my colleagues based in the Information Technology department at NUI, Galway. A special thanks to Dr. Michael Schukat for his additional support and guidance and my peers with whom I have enjoyed countless cups of caffeine with.
- My fellow national and international researchers with whom I have collaborated with over the course of my research and who have proved an endless source of knowledge and expertise. These include Dr. Peter Pocta, Dr. Antonio Ruzzelli, Prof. Liam Murphy and Prof. John Murphy.
- My family and friends for their relentless support on this journey, in particular, my mother and father who continuously encouraged me and accommodated my every need. I am certain that without their support I would not have completed this research.
- And finally my two dare friends Flo and John. Their hospitality towards me made possible the undertaking of a significant portion of this research while their friendship with me ensured its completion.

Abstract

Time synchronisation plays a key and ever increasing role in this progressively networked world. Whilst time synchronisation has always been important in industrial and telecommunication systems, its importance has now extended to more diverse applications such as consumer multimedia, medical informatics, SmartGrid, and environmental monitoring. As the need for time synchronisation has grown, so too has the need for such time-sensitive applications to be deployed over wireless networks. It is estimated that there are more than twelve and a half billion Internet-enabled devices connected at the moment - this is expected to reach somewhere between twenty and fifty billion by 2020, as the so-called *'Internet-of-things'* evolves. Much of this connectivity will be via wireless.

The rapid expansion of wireless networks in the last two decades has drastically transformed the architecture of many traditional packet-switched networks. This is particularly the case with TCP/IP based *Local Area Networks (LANs)* which form a significant building block of the internet. Wireless networks have become ubiquitous within the internet, key amongst them being 802.11 based technologies that now commonly represent the last hop. Whilst hugely convenient, such networks present many new challenges in terms of time synchronisation. In particular, 802.11 or WiFi marks a return to contention based access which can lead to significant delays that are often asymmetric. This greatly degrades the performance of common synchronisation protocols that are designed for largely symmetric wired networks.

Another wireless based network has experienced significant, albeit less spectacular growth in the last decade, namely, *Wireless Sensor Networks (WSNs)*. They provide a means to gather physical data via

a wireless network of low-powered, inexpensive computing devices. These miniature devices, although limited in terms of power and computing resources, have huge potential when used in a distributed fashion. They can be used to collect sensory data from large geographical areas which, when collated, can provide invaluable insight into a phenomenon under observation. Data collected via WSNs often requires precise time stamping, and thus, in such cases, WSNs require time synchronisation protocols in order to operate effectively. Since the computing devices that compose a WSN are typically power-limited, the overhead of a time synchronisation protocol can use up valuable energy, thus, limiting node life.

The research contributions in this thesis address key synchronisation problems within both 802.11 and WSN domains. The significant problem concerning the degradation of time protocols operating over 802.11 networks is remedied via a novel mechanism that exploits the standard operation of 802.11 networks in order to reduce the error in datasets employed by such protocols. Validation experiments performed confirm that the mechanism can deliver synchronisation accuracies akin to those achievable over wired networks.

In relation to WSNs, the research contribution addresses the key problem of communication (and thus energy) overhead for synchronisation protocols. This contribution is presented by means of a new protocol, termed the *Dynamic Flooding Time Synchronisation Protocol (D-FTSP)* which offers an energy efficient means of synchronising WSNs deployed in dynamic environments. Experiments reveal that D-FTSP, in particular scenarios, can result in significant energy savings with respect to alternative time protocols, thus, greatly extending the life of a WSN.

Contents

List of Figures

National and International Conferences

[1] Hugh Melvin, Padraig O' Flaithearta, Jonathan Shannon, and Lourdes B. Yuste. Time synchronization within multimedia applications: Opportunities and challenges. In *IEEE Digital Technologies (DT)*, Zilina, Slovak Republic, Jun 2013.

[2] Jonathan Shannon, Hugh Melvin, and Antonio G. Ruzzelli. Dynamic flooding time synchronisation protocol for wsns. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, Anaheim, California, Dec 2012.

[3] Jonathan Shannon, Hugh Melvin, and Antonio G. Ruzzelli. Wireless sensor network synchronisation (d-ftsp). In *Intl. Telecom Synchronisation Forum (ITSF)*, Edinburgh, Scotland, Nov 2011.

[4] Jonathan Shannon and Hugh Melvin. A dynamic wireless sensor network synchronisation protocol. In *Digital Technologies (DT)*, Zilina, Slovak Republic, Nov 2011.

[5] Jonathan Shannon, Hugh Melvin, Ronan O hOgartaigh, and Antonio G. Ruzzelli. Synchronisation challenges within future smart grid infrastructure. In *ENERGY, The First International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, Venice, Italy, May 2011.

[6] Hugh Melvin, Jonathan Shannon, Padraig O' Flaithearta, and Lourdes B. Yuste. Role of synchronisation in the emerging smartgrid infrastructure. In *Intl. Telecom Synchronisation Forum (ITSF)*, Dublin, Ireland, Nov 2010.

[7] Jonathan Shannon and Hugh Melvin. Synchronisation issues over wireless networks. In *Globe Forum*, Dublin, Ireland, Nov 2010.

[8] Hugh Melvin, Padraig O' Flaithearta, Jonathan Shannon, and Lourdes B. Yuste Time synchronisation at application level: Potential benefits. In *Intl. Telecom Synchronisation Forum (ITSF)*, Rome, Italy, Nov 2009.

[9] Jonathan Shannon and Hugh Melvin. Synchronisation challenges for wireless networks. In *Digital Technologies (DT)*, Zilina, Slovak Republic, Nov 2009.

Other Publications

[1] Jonathan Shannon, Hugh Melvin, and Ronan O hOgartaigh. Smart grid synchronisation: Wireless challenges and solutions. Energy Night, NUI Galway, Jan 2011. Poster.

[2] Mark White, Jonathan Hanely, Jonathan Shannon, Hugh Melvin, Michael Schukat, Marcus Keane, Maria Linnane, and Wesley Reilly. Data centre energy efficiency. Energy Night, NUI Galway, Jan 2011.

[3] Jonathan Shannon and Hugh Melvin. Synchronisation challenges for wireless networks. IT Seminar Series, NUI Galway, Feb 2010. Seminar.

[4] Jonathan Shannon and Hugh Melvin. Improving synchronisation within wireless sensor networks. Joint ECI-MRI Research Day, NUI Galway, Jun 2010.

[5] Jonathan Shannon and Hugh Melvin. Improving synchronisation within wireless sensor networks. ECI Research Day Programme, NUI Galway, Jun 2009. Poster.

[6] Jonathan Shannon and Hugh Melvin. Improving synchronised time over wireless networks. University College Dublin seminar, Nov 2008. Seminar.

Other Contributions

[1] Jonathan Shannon and Hugh Melvin. Improving time synchronisation over 802.11 networks. Unpublished work, 2013.

[2] Jonathan Shannon and Hugh Melvin. Improving time synchronisation over 802.11 networks. Timing over IP Connection and Transfer of Clock (Tictoc), 2013. Internet Engineering Task Force (IETF) working group.

Part I

Background

Chapter 1

Introduction

It is estimated that there are over twelve and a half billion devices connected to the Internet [1]. Continuous advancements in digital fabrication techniques have facilitated this through the development of more powerful and advanced computing systems with ever decreasing scale/form factors. In recent years, the term *'Internet-of-things'* or *IoT* has been coined to represent the expectation that the number of such devices will undergo the next phase of rapid growth and miniaturisation, with some predictions of fifty billion by 2020 [1, 2]. Fig. 1.1 illustrates the rate that this connectivity is predicated to evolve [1]. Much of this connectivity will be facilitated via a range of wireless technologies which include *cellular*, *WiFi (802.11)*, *Bluetooth*, and *802.15.4 (Wireless Sensor Networks)*.

Time synchronisation plays a key and ever increasing role in this progressively networked world. While time synchronisation has always been important in industrial and telecommunication systems, its importance has now extended to more diverse applications. These include consumer multimedia applications/services such as *VoIP (Voice over IP)*, *IPTV (Internet Protocol Television)*, *MMOG (Massive Multiplayer Online Games)*, *energy & environmental informatics*, *medical informatics*, and *SmartGrid*. This thesis addresses two key yet distinctly different challenges in the delivery of time synchronisation over two of the above wireless technologies, namely *IEEE* standards *802.11* [3] and *802.15.4* [4]. The deployment of 802.11 networks has seen huge growth in the last decade and often represents the last hop in network connectivity. This growth is expected to increase as large chip manufactures begin to invest in low-powered Wi-Fi tech-

nologies [5, 6] in place of alternative low-powered wireless technologies, such as ZigBee and Z-Wave, that lack the maturity and bandwidth offered by WiFi. Studies that investigate the feasibility of employing low-powered Wi-Fi chips in battery powered devices have shown that multiple years of battery lifetime is achievable for most real-world scenarios [7].

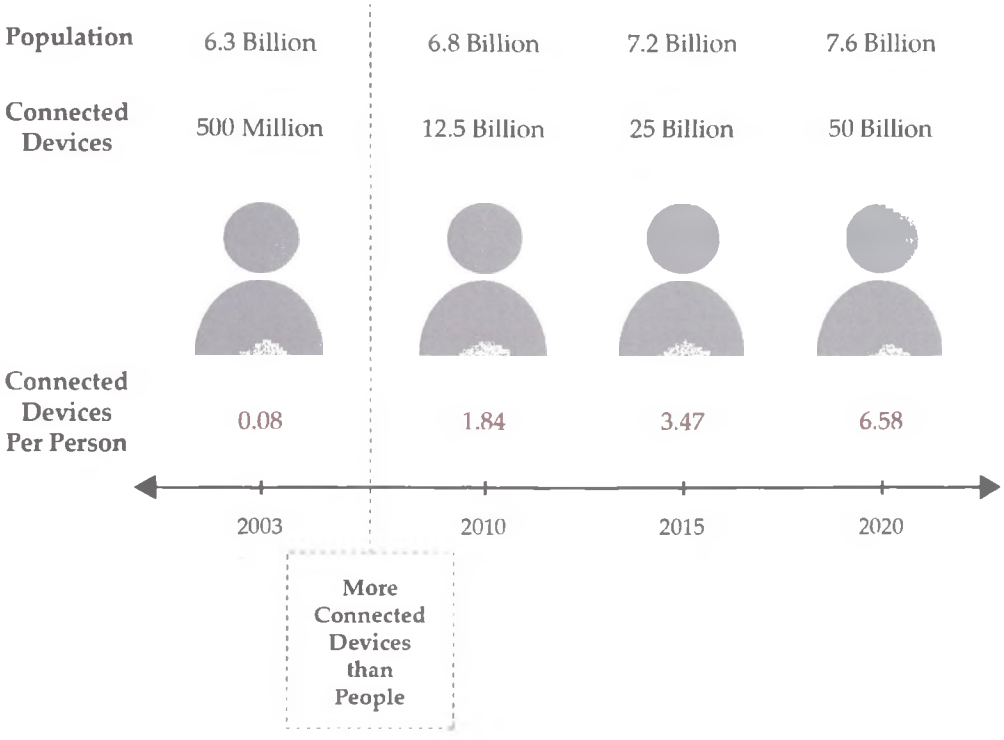


Figure 1.1: Connected devices

Advanced fabrication techniques have also led to the development of low-power, miniature computing systems with comparatively sophisticated processing and communication capabilities. These relatively inexpensive devices when equipped with sensory hardware and subsequently networked together form networks termed *WSNs (Wireless Sensor Networks)* which have numerous applications. The development of the *IEEE 802.15.4* [4] standard in 2003 was specifically aimed at resource-constrained devices requiring low power and low data rate functionality.

As outlined above, time-sensitive applications form a particular subset of network applications. Such applications generally require time synchronisation services as the host's local time-scale must possess a particular degree of accuracy in order to guarantee their correct operation. The accuracy required is very much application dependent, ranging from seconds to nanoseconds. To ensure a particular degree of accuracy, time-sensitive applications rely on a set of auxiliary applications termed *time synchronisation protocols*. They employ various synchronisation techniques and methods in order to improve the accuracy of a host's local clock. These protocols have evolved over the years and prove effective. However, as time-sensitive applications are forced to migrate to wireless networks and are required to run on devices with very limited power reserves, they place new demands on time synchronisation protocols and techniques.

1.1 Time Synchronisation Techniques

Time synchronisation techniques form the cornerstone of most time-sensitive applications, in particular distributed network applications. Such techniques aim to ensure that the time-scale of a host adheres to some reference time-scale. This reference scale can be sourced locally via highly precise clocks or remotely. In the latter case, network connectivity facilitates synchronisation. The potential accuracy provided by the latter technique is highly influenced by the characteristics of the host, the reference device, and the communication link that connects them. The communication link generally proves to be the greatest source of error. Thus, some techniques employ numerous communication exchanges and statistical analysis to alleviate this problem. The combination of such methods and techniques typically form what is termed a *time synchronisation protocol*.

1.2 Time Synchronisation Protocols

Time synchronisation techniques detail the sequence and order of communications between a host and reference, as well as the core data exchanged between them. Specific message structure and other operational details are not specified. The

actual implementation of a synchronisation technique in combination with the message structure, and additional operations and methods, form a time protocol.

Time protocols are typically designed to target specific network types. For example, the *Network Time Protocol (NTP)*, by Mills [8], is designed to operate over large, dynamic and variable latency *Packet Switched Networks (PSNs)*. It does so effectively by employing sophisticated statistical techniques to minimise errors introduced by such networks. Conversely, the *Precision Time Protocol (PTP)* [9] is designed for privately managed and well controlled PSNs that employ specialised hardware and, therefore, can provide much greater accuracies. Each protocol performs well in its targeted environment. The accuracy requirements of a time-sensitive application will ultimately dictate which protocol is employed and which network(s) it should be deployed over.

Both NTP and PTP are designed to operate within traditional packet switched networks. Such networks typically span large geographical areas and consist of devices that have considerable processing, memory, and energy resources. As detailed above, this is not the case in a WSN where resources are very limited and the geographical area it spans is typically small. Although a WSN's composite nodes are relatively powerful for their size, they pale in comparison to a typical PSN node. To elaborate, a typical WSN node's microcontroller operates in the low MHz range and may possess only several hundred kilobytes of short term memory. In comparison, a typical PSN node's CPU will operate in the GHz range, and the node may possess several gigabytes of short term memory. Consequently, alternative time protocols have emerged to deal with the limited capabilities of WSN nodes. These include the *Flooding Time Synchronisation Protocol (FTSP)*, by Maróti et al. [10]; *Reference Broadcast Synchronisation (RBS)*, by Elson et al. [11]; and the *Timing-sync Protocol for Sensor Networks (TPSN)*, by Ganeriwal et al. [12]. Such protocols operate in a fashion so as to limit processor, memory, and, more importantly, energy use while still providing a high degree of time accuracy.

1.3 Research Motivation

The research in this thesis addresses two distinct challenges within the time synchronisation space. The context for both and the motivation behind the research in each area is detailed in the following subsections.

1.3.1 Time Synchronisation in 802.11 Networks

The extensive deployment of wireless networks in recent years has dramatically altered the structure of local and wide area networks. Wireless links now represent a large subset of the extremities of such networks. From a user perspective, wireless links offer the advantage of mobility, though often at a cost of bandwidth and other *Quality of Service (QoS)* issues. Nonetheless, users are increasingly demanding and expect QoS similar to that achievable over wired networks. For many of these applications this is not an issue since modern wireless technologies, such as IEEE 802.11, provide sufficient bandwidth and data rates to accommodate large, sporadic traffic loads. This, however, is not true for various time-sensitive applications, that is, applications that require a notion of real time. In such cases, time synchronisation protocols must be employed in order to ensure accurate time stamping. It is known that the performance of synchronisation protocols, such as NTP, can degrade significantly when they operate over a variable latency network [13], and an 802.11 wireless network represent such a network. This is particularly true when an 802.11 link must accommodate significant traffic loads.

This issue stems from the fact that 802.11 networks employ a shared medium and, thus, are inherently contentious. Nodes must content for the medium with competing nodes before data transmission can begin. When traffic loads are high, this can introduce large non-deterministic network delays. This reality, coupled with the fact that wireless networks operate at relatively low data rates in comparison to wired networks, degrades the performance of time protocols. This degradation in turn, greatly restricts the deployment of applications that either require or can benefit from time synchronisation. Accordingly, techniques and methods must be designed and developed to alleviate this issue. If it were possible to provide time synchronisation accuracies akin to that achievable of wired networks, then it would not only permit the migration of time-sensitive applica-

tions to the wireless domain but also open up a window for new applications and techniques to be developed.

1.3.2 Time Synchronisation in Wireless Sensor Networks

Time synchronisation protocols currently deployed in WSNs can provide a relatively high degree of accuracy, some in the order of microseconds. Although such networks also employ contention based wireless technologies, network access delays can be determined with greater ease. This is in part due to their architecture. The architectural layers in a WSN node are generally more coupled and, thus, accessing lower layers in the communication stack is trivial. Obtaining timestamps recorded at lower layers in the stack eliminates much of the non-determinism associated with message latencies over 802.15.4 and similar networks. Thus, higher accuracies can be achieved with less effort. Nonetheless, the limited memory capacity, processing power, and energy reserves of nodes present different challenges to time protocols. The most limited resource is typically energy, as battery replacement is expensive, time consuming, and often impractical. Since time protocols act as auxiliary background service applications to higher level WSN applications (e.g. environmental monitoring applications), their energy consumption must be restricted. Thus, in a WSN context, a good time protocol does not just provide a high level of synchronisation but does so in an energy efficient manner.

Many of the time protocols employed by WSNs are reasonably effective in this respect, yet they are generally only effective in particular environments. Dynamic environments, that is, environments that subject nodes to temperature or pressure fluctuations, can have a significant influence on the behaviour of a node's clock. Such changes can alter the operational frequency of a clock and if not detected quickly and compensated for, will lead to cumulative time error. This problem can be alleviated by selecting a suitable communication rate based on a priori knowledge of the environment, in particular, its temperature. While this is effective in terms of achieving the required degree of accuracy, it results in significant energy wastage due to unnecessary communication overhead. This is particularly true when temperature fluctuations occur at infrequent intervals.

There are further more complex protocols that are designed to tackle such problems such as *Rate Adaptive Time Synchronisation (RATS)*, by Ganeriwal et al. [14]; and *Temperature Compensated Time Synchronisation (TCTS)*, by Schmid et al. [15]. However, such protocols either require additional hardware or do not react to environmental changes rapidly enough. Thus, there remains a gap that must be filled. What is required is an energy efficient WSN time protocol that is targeted at WSNs that must operate in dynamic environments. Such a protocol would react rapidly to environmental changes, ensuring a continuous application-specific degree of accuracy with minimum communication overhead and no additional hardware.

1.4 Problem Statement

The current limitations regarding synchronisation accuracy in 802.11 networks and synchronisation communication/energy overhead in wireless sensor networks pose a number of key questions that are addressed in this thesis:

- To what degree do high traffic loads in 802.11 wireless links impact time synchronisation techniques employed by PSN time protocols? Is it possible to provide a technique or method that would permit time protocols operating over contentious 802.11 wireless links to achieve time synchronisation accuracies similar to that currently achievable over wired links?
- In relation to time synchronisation over wireless sensor networks, is it possible to provide an alternative synchronisation protocol, targeted at WSNs operating in dynamic environments, that would provide similar accuracies to that of current WSN time protocols but with a significant reduction in communication overhead? If so, would this reduction in communication result in significant energy savings?

1.5 Solution Approach

1.5.1 Improved Time Synchronisation in 802.11 Networks

In order to determine the extent to which the performance of time synchronisation techniques can be degraded by 802.11 links, in particular, contentious links, one must determine the magnitude of delays that packets traversing such links may be subjected to. This thesis presents such an analysis using both an experimental and simulated environment.

A subsequent analysis of the source of such delays with respect to the operation of 802.11 networks is used to formulate a technique that can be used to measure/estimate such delays and, thus, alleviate resulting time errors. This novel technique employs facilities provided by modern 802.11 network interface cards and exploits the operation of 802.11 networks in order to determine the delays incurred by time-related packets traversing such networks. This information is subsequently used to improve the quality of temporal data employed by time protocols. The technique is validated and its effectiveness is quantified in a real world testbed.

1.5.2 Energy Efficient Time Synchronisation in WSNs

In order to provide an energy efficient time protocol targeted at WSNs operating in dynamic environments, this thesis builds upon an existing time protocol, namely, the *Flooding Time Synchronisation Protocol (FTSP)*, by Maróti et al. [10]. FTSP has been widely deployed as an effective time protocol in WSNs. It is capable of providing a high degree of synchronisation accuracy with relatively low processing and memory overhead. It is, however, more suited to stable environments, since it is configured with a static transmission rate. This transmission rate is manually chosen based on both the characteristics of an operating environment and an application's accuracy requirements. In dynamic environments, this can lead to sub-optimal communication overhead and energy usage.

The research contribution presented in this thesis extends FTSP by enhancing it with dynamic capabilities and, thus, is appropriately termed the *Dynamic Flooding Time Synchronisation Protocol (D-FTSP)*. D-FTSP gives WSN nodes

the ability to monitor clock frequency changes in real time, and this allows them to determine suitable transmission rates so that application-specific accuracy requirements are adhered to. This dynamic alteration of a node's transmission rate in response to environmental factors can lead to significant transmission reductions, thus, prolonging the life of a WSN. In addition, the need for a priori manual configuration of a suitable transmission rate is eliminated.

The design, implementation, and validation of D-FTSP are presented in detail. Its effectiveness is verified via experimentation in a real world testbed by comparing it to FTSP in both a stable and dynamic environment. Comparative analysis of both protocols is based on the accuracies they achieve in addition to the number of message transmissions required to achieve their respective accuracies. A subsequent analysis of the potential energy savings of D-FTSP is performed using the results. This analysis factors in the additional energy consuming operations performed by D-FTSP relative to FTSP.

1.6 Contributions

1.6.1 Improved Time Synchronisation in 802.11 Networks

- A low overhead and scalable technique that delivers accurate time synchronisation over contentious 802.11 networks.
- In particular, the technique allows for the determination of the up-link and down-link delay of time packets as they traverse the link between an 802.11 wireless node and its associated access point and vice versa. Knowledge of such delays can be used to significantly improve the quality of a temporal dataset that is employed by PSN time protocols, such as NTP. Experiments reveal a 90% reduction in the average error in a dataset.
- This technique, when used in conjunction with a time synchronisation protocol, can potentially yield accuracies akin to those achievable over a wired link.
- This contribution facilitates the effective deployment of time sensitive applications over 802.11 networks, thus, removing a significant barrier that

currently limits their use.

1.6.2 Dynamic Flooding Time Synchronisation Protocol (D-FTSP)

- D-FTSP provides WSN nodes with the capability to directly detect clock drift in neighbouring nodes. This allows nodes to self-configure an optimal transmission rate while ensuring a WSN application's accuracy requirements are adhered to.
- Automatic configuration of a nodes transmission rate eliminates the need to determine a suitable value at the pre-deployment stage of a WSN. Thus, nodes can be deployed rapidly without detailed knowledge of the dynamics of the operating environment.
- In a dynamic/un-stable environment, D-FTSP is capable of achieving similar network-wide accuracies to that of FTSP with a much reduced number of transmissions, thus, conserving the limited energy supplies of WSN nodes and extending a WSNs life.

1.7 Thesis Outline

The current chapter has served as an introductory chapter in part I of this thesis. It has outlined the problems that exist with respect to time synchronisation in both wireless sensor networks and WiFi/802.11 networks. Additionally, it has presented an overview of the core contributions of this thesis which address those problems.

Chapter 2, which follows the current chapter, provides the reader with fundamental details related to clock oscillators and time synchronisation. It commences with an outline of the architecture and operation of a system clock, as employed in typically computing systems. This is followed by a detailed analysis of the quartz crystal, the frequency standard used in most inexpensive oscillators. An analysis of the quartz crystal is followed by an explanation of time error and the global time standards used to determine such error. The chapter concludes with a

discussion on time synchronisation, in particular, the core techniques used to synchronise distributed computing systems and the main sources of synchronisation error.

Part II of this thesis focuses on synchronisation in wireless sensor networks and presents the first core contribution of this thesis. The first chapter, chapter 3, focuses on state of the art time synchronisation techniques/protocols that are employed in current wireless sensor networks. A detailed description of each protocol's design and operation is presented. This is followed by a comparative analysis. The chapter concludes with a discussion that highlights the need for more energy efficient and responsive time protocols in wireless sensor networks.

Chapter 4 of part II follows on from chapter 3 and focuses on the first major research contribution of this thesis, namely D-FTSP. D-FTSP represents an extension of the FTSP protocol described in chapter 3. Chapter 4 commences by detailing the design and operation of D-FTSP with respect to FTSP. This is followed by an experimental analysis of its performance with respect to FTSP. The chapter concludes with an analysis of the potential energy savings of D-FTSP.

Part III of this thesis diverges from the domain of wireless sensor networks and focuses on synchronisation in packet-switched networks, with a particular focus on WiFi networks. Chapter 5 serves as a tutorial of sorts which aims to provide the reader with a solid technical foundation required to understand the significance of the second major contribution of this thesis. It provides the reader with a comprehensive description of the two most dominant protocols deployed in packet-switched networks, namely NTP and PTP. The chapter concludes with a detailed technical analysis of 802.11 network operation and highlights how its operation can impact the performance of PSN time protocols.

Chapter 6 of part III follows on from chapter 5 and details the second major research contribution of this thesis, namely a technique that can improve time synchronisation in 802.11 networks. The extent of the problem with regard to synchronisation in busy 802.11 networks is analysed via experimental simulations. This is followed by a detailed explanation of a novel mechanism that can be used to improve the performance of PSN time protocols that must operate in highly active 802.11 networks. Finally, a description of the experimental setup used to validate the effectiveness of the mechanism is presented, and the chapter concludes

with an analysis of the results.

Part IV of this thesis presents the conclusions. Chapter 7 of part IV concludes the thesis by summarising the core research contributions. Finally, some potential future work is outlined.

Chapter 2

Computer Clocks and Time Synchronisation

The purpose of any clock employed by a computing system is to generate timing signals. A general purpose computing system will typically possess three core clock oscillators, and each is designed to deal with a specific task. The most familiar of these is the *CPU clock oscillator* which generates the timing signals required to drive a CPU's circuitry. Such oscillators typically generate timing signals at gigahertz (GHz) frequencies permitting CPU's to execute hundreds of millions of instructions per second.

The *system clock oscillator*, which operates at a much reduced frequency, sometimes three orders of magnitude lower, generates the timing signals required to track real-time. In this context, real-time refers to the value of a global time standard such as *Coordinated Universal Time (UTC)*. Such a clock will generally have a dedicated timer or counter which continuously counts and stores the number of pulses generated by the oscillator. It is the responsibility of higher layer system software to use this facility to track real time, since the capacity of the counter is limited and. therefore, overflows at regular and short intervals. The system clock facilitates timekeeping tasks required for process scheduling, CPU utilisation tracking, and file system operations.

In order to provide a computing system with the ability to track real-time when powered off. a battery-powered clock oscillator that utilises low powered

circuitry akin to that used in wrist-watches is employed. Such a clock is termed a *real-time clock*, and the value of this clock is read by system software at start-up. After start-up, subsequent time tracking is performed by the system clock.

The CPU clock, system clock, and real-time clock are by no means the only clocks found in a general purpose computing system. Virtually any device that requires a separate timing signal, such as a network interface card (NIC) or high performance display adapter, generally contains a dedicated clock oscillator. The characteristics of each clock oscillator depend on the task it must perform, and, therefore, the clock oscillators may differ significantly. While the proceeding text focuses on the system clock, the reader should be aware that the terms used to describe the characteristics of the system clock can be applied to virtually any clock.

2.1 The System Clock

The illustration in fig. 2.1 depicts the core components involved in the process of tracking real-time or the time of day. A general programmable system clock consists of an *oscillator*, a *holding register*, and a *clock register*. The holding register stores a system defined constant which, at system start-up, is loaded into the clock register. Subsequently, the oscillator begins to generate timing signals at a specific frequency, each of which decrements the value of the clock register. When the value of the clock register reaches zero, it generates an interrupt after which the value of the holding register is loaded back into the clock register. This process repeats indefinitely resulting in an extremely regular occurrence of interrupts termed *clock ticks*. One of the many tasks undertaken by the operating system in response to a clock tick is to increment a counter in main memory which is used to track the system time. Generally, the value of the system time represents the number of seconds since a particular epoch and, ideally, agrees with an accepted time standard such as *Coordinated Universal Time (UTC)* (detailed in section 2.3.1). It is for this reason it is also referred to as real-time and represents the primary source of time for both the operating system and user processes.

The time interval between system clock interrupts, or clock ticks, is dependent

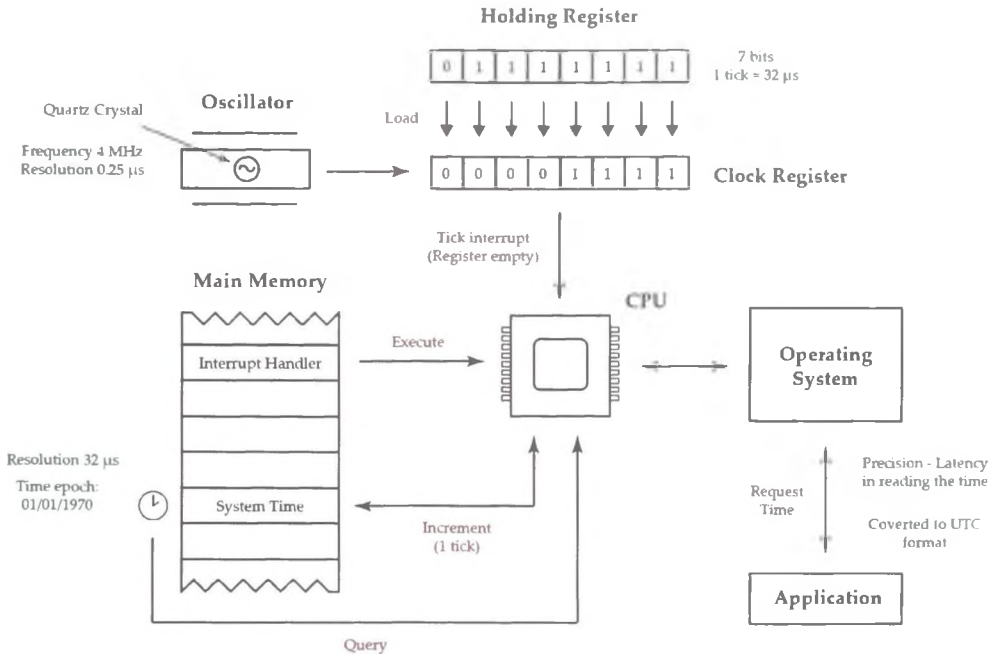


Figure 2.1: System clock operation

on two factors: the frequency of the oscillator and the value of the holding register. For example, if the oscillator generates a signal at a rate of $1MHz$ and the holding register contains a value of 1000_{10}, then the clock register will generate an interrupt every 1ms. In addition to incrementing the system time, other tasks that must be completed at each interrupt event include decrementing the process quantum, tracking CPU utilisation, and checking for watchdog timer expirations (see Tanenbaum and Woodhull [16]). Since clock interrupts are so numerous, the instructions that handle these tasks must be few and extremely efficient in order to avoid degrading the system's performance. The interrupt handler itself is usually responsible for handling the majority of clock interrupts so as to avoid the expense of a context switch. Other more computationally expensive clock tasks, such as those required to handle an expired timer or process quantum, are typically handled by a dedicated clock handler. It is worth noting that the occurrence of clock interrupts during intervals when CPU interrupts are disabled

can result in lost ticks. To counteract this, some systems employ a global variable that is provided to code modules that disable interrupts in order to track lost ticks.

The resolution of a clock represents the degree to which one clock reading can be distinguished from another. The maximum possible time resolution that a system clock is capable of providing is dependent on its composite oscillator's nominal frequency. For instance, a $1MHz$ oscillator can be used to track time in increments of $1\mu s$ but no less. If this same oscillator is employed by a programmable system clock and its holding register set to 1_{10}, then the clock will generate interrupts at $1\mu s$ intervals. Consequently, the resolution of the system time as tracked by the operating system will also be $1\mu s$. However, in general, for performance reasons, the holding register will contain a larger value, and, therefore, the resolution of the system time will be less than the maximum provided by the underlying system clock.

If a user process requests the time, it generally does so via a system call which ultimately returns the value of the system time. The time interval between the initiation of this function call and its eventual return depends on such things as the CPU load at that particular time, the process scheduling algorithm in use, the priority of the user process, and so on. Thus, it is clear that this interval varies over time. The minimum magnitude of this variation represents the maximum precision provided by the operating system when reading the system time. Thus, the quality of a timestamp returned to a user process is dictated by both the underlying clock hardware and the higher layer system software.

2.2 The Crystal Oscillator

So far, the process of tracking time in a generic computing system has been detailed. It is now evident that the characteristics of both a clock's hardware and a system's software dictate the precision and granularity of a timestamp returned via a system call to a user process. Note that even without software precision errors, the time, as tracked by a system clock, even if it initially agrees with true time as recorded by a perfect clock, it might at some point in the future differ from it. To explain why this might occur one must focus on the oscillator, the core component of the system clock.

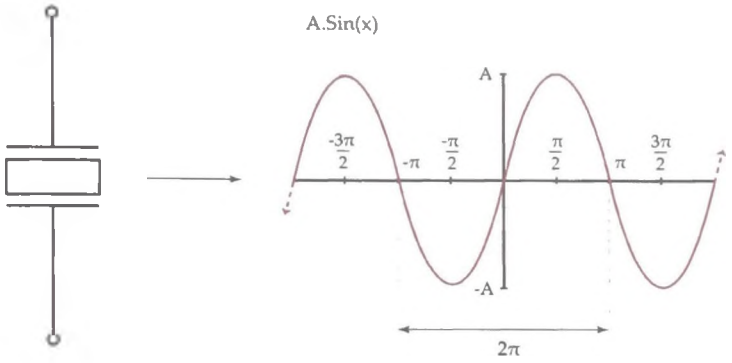


Figure 2.2: Oscillator signal

Oscillators used to drive clocks typically come in two varieties, namely *resistor-capacitor oscillators (RC)* and *crystal-controlled oscillators (XO)*, both of which generate a sinusoidal output signal at a relatively constant rate (see fig. 2.2). As detailed in [17], an ideal signal can be modelled using equation 2.1.

$$S(t) = A.sin[\Phi(t)] \tag{2.1}$$

In equation 2.1, A represents the amplitude coefficient and $\Phi(t)$ represents the total instantaneous phase. The total instantaneous phase can be modelled using equation 2.2, where V_{nom} represents the nominal frequency of the oscillator and t the true time.

$$\Phi(t) = 2\pi V_{nom} t \tag{2.2}$$

RC oscillators are composed of a network of resistors and capacitors, the structure and composition of which dictates the frequency of the output signal. Crystal-controlled oscillators employ a crystal as the frequency-determining device within the oscillator. Crystals exhibit a characteristic known as the *piezoelectric effect* whereby mechanical forces produce electrical charges and vice versa. Some crystal substances exhibit this effect to a greater degree than others, namely *quartz* and *Rochelle salt*, and, therefore, they are more commonly employed in oscillators, the former more so than the latter. With respect to time keeping, crystal-controlled oscillators provide superior frequency stability when compared

to their counterparts and, therefore, are commonly employed as the driving force of a system clock. In contrast, RC oscillators are generally used to generate clock signals in integrated circuits.

2.2.1 Crystal Accuracy

Crystals that are employed in oscillators are produced by cutting a piece of quartz at specific angles to specific axes. The angles of cuts together with their orientation have a dramatic effect on the characteristics of a crystal. In order to ensure a crystal resonates at a desired frequency, the cut, shape, and size of the crystal is chosen carefully so that its natural resonant frequency matches that of the desired frequency. This frequency in turn dictates the maximum resolution of the clock it drives. Once the crystal has been shaped, it is carefully mounted in holders within the oscillator circuitry and a specific voltage is applied to it. The result is mechanical vibrations which in turn produce an output voltage at the crystal's natural resonant frequency.

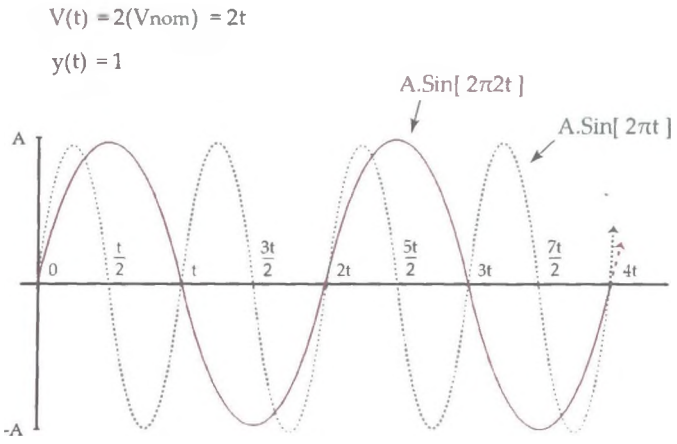


Figure 2.3: Oscillator accuracy

Since the natural resonant frequency of a crystal is highly dependent on its cut, the quality of the manufacturing process used to shape it will have a direct influence on the crystal's quality. Hence, a crystal's actual frequency may differ

from its stated frequency. This is generally true and in most cases manufactures will state the maximum error, which is typically less than 10ppm *(parts per million)*. The problem with frequency error becomes apparent when one considers the clock system an oscillator drives. If a crystal is stated as having a frequency of 1MHz, then one oscillation will be interpreted by the system as an interval of $1\mu s$. If, however, the crystal has a frequency offset of -10ppm, then for every million oscillations of the crystal, an extra $10\mu s$ will have passed in real time. While this may not seem significant, it results in a time error of almost a second a day. This frequency offset represents an important characteristic of an oscillator, namely the *accuracy*. Naturally, the lower the frequency offset of the oscillator, the more accurate it is. Formally, the accuracy of a clock can be expressed using a measurement termed the *fractional frequency deviation* and is calculated using equation 2.3.

$$y(t) = \frac{V(t) - V_{nom}}{V_{nom}} \tag{2.3}$$

In equation 2.3, $V(t)$ represents the actual frequency at true time t and V_{nom} represents the nominal, or expected, frequency. This is also illustrated in fig. 2.3. As expected, a value of zero represents a very accurate oscillator. Since that accuracy represents a very important characteristic of an oscillator, it must be factored in when modelling the total instantaneous phase of an oscillator as shown in equation 2.4.

$$\Phi(t) = \Phi_0 + 2\pi V_{nom}(1 + y_0)t \tag{2.4}$$

In equation 2.4, $\Phi(t)$ represents the total instantaneous phase, Φ_0 represents the initial phase offset, V_{nom} represents the nominal frequency, y_0 represents the initial fractional frequency offset from the nominal value and t represents the true time.

2.2.2 Crystal Stability

Another important characteristic of an oscillator is *stability*. Stability is a measure of an oscillator's ability to sustain a constant frequency over time. To understand

how the frequency of an oscillator could change, one must recognise that the crystal within an oscillator is influenced by physical factors. Temperature and pressure changes cause materials to expand and contract and, since the natural resonant frequency of a crystal is influenced by its shape and size, the issue becomes apparent. Slight variations in the angles of a crystal's cuts can have a dramatic impact on the stability of a crystal when operated within a specific temperature range. Thus, the specific shape of a crystal is not solely chosen so that it resonates at a desired frequency but also so that its temperature coefficient is suited to the temperature range of its future operating environment.

While temperature typically has the greatest impact on the frequency of an oscillator, other factors, such as pressure, voltage, gravity, and vibration. affect it too. Oscillator stability can also be classified in terms of time. Long-term stability refers to a gradual and permanent change in the oscillator's frequency over days or months. Long-term frequency change is referred to as *aging* and is usually relatively constant. Aging is the result of such things as contaminant build-up on the crystal and changes to the structure in/on which the crystal is housed/mounted. Short-term stability refers to short-term frequency changes which are typically the result of noise.

Factoring in stability when modelling the total instantaneous phase of an oscillator results in equation 2.5

$$\Phi(t) = \Phi_0 + 2\pi V_{nom}(1 + y_0)t + \pi D V_{nom} t^2 + \phi(t) \tag{2.5}$$

In equation 2.5, D represents the linear rate of change of the fractional frequency deviation, and $\phi(t)$ represents random phase deviations. In addition, it should be noted that $\phi(t)$ represents the stochastic component of equation 2.5, while the remaining terms represent deterministic components. Thus, without random phase deviations the instantaneous phase of the oscillator could be accurately predicted.

The typical noise types that compose the stochastic component $\phi(t)$ are termed *power-law noise processes* and are generally classified into five categories:

- *White phase modulation* (WPM)
- *Flicker phase modulation* (FPM)

- *White frequency modulation/ Random walk phase modulation* (WFM/RWPM)
- *Flicker frequency modulation* (FFM)
- *Random walk frequency modulation* (RWFM)

In some cases it can be useful to identify the dominant noise processes generated by an oscillator. This not only allows multiple oscillators to be compared but also allows one to determine an optimal observational interval for frequency estimations.

As detailed by Howe et al. [18], historically, the standard deviation was used to analyse fractional frequency fluctuation samples which were collected via a measurement technique that compared the signal generated by an oscillator under observation with the signal generated by a reference oscillator. However, it was found that in certain cases, such as when the fluctuations were characterised by non-white-noise frequency fluctuations, the standard deviation would increase as the number of samples increased, and without limit. Thus, other statistical methods were developed which were independent of the sample size.

The *Allan deviation (ADEV)* by Allan [19], expressed as $\sigma_y(\tau)$, represents the most recognised method used to characterise the random fluctuations of an oscillator's signal. It can be used to determine the dominant noise processes of an oscillator's signal at various sampling intervals and is defined in equation 2.6.

$$\sigma_y(\tau) = \sqrt{\frac{1}{2\tau^2}\left\langle \Big[x(t+2\tau) - 2x(t+\tau) + x(t)\Big]^2 \right\rangle} \tag{2.6}$$

In equation 2.6 the term τ represents the observational interval, $x(t)$ denotes a time error sample recorded at time t and the brackets $\langle\rangle$ represent an ensemble average. The Allan deviation can be estimated from a finite data set using equation 2.7 where τ_0 represents the sampling interval, $n\tau_0$ represents the observational interval ($\tau = n\tau_0$) and N represents the number of samples in the data set.

$$\sigma_y(n\tau_0) = \sqrt{\frac{1}{2n^2\tau_0^2(N-2n)} \sum_{i=1}^{N-2n} \Big(x_{i+2n} - 2x_{i+n} + x_i\Big)^2} \tag{2.7}$$

One might observe that while the standard deviation is calculated from the differences between fractional frequency samples and the average fractional frequency, which itself is calculated using the entire sample set, the Allan deviation, in contrast, is calculated from the differences between adjacent fractional frequency samples. An interesting aspect of the Allan deviation is that it converges for all the major noise processes. Thus, by graphing $\sigma_y(\tau)$ for various values of τ on a log-log plot, the noise processes can be discriminated between based on the slope of $log(\sigma_y(\tau))$. The slope associated with each noise process is illustrated in fig. 2.4.

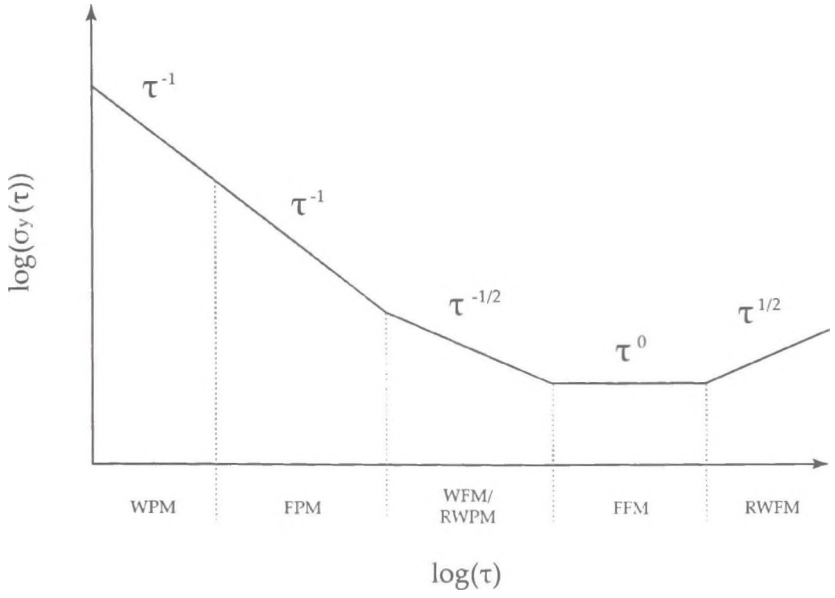


Figure 2.4: Convergence of ADEV for various noise processes

One may observe from fig. 2.4 that ADEV does not allow one to discriminate between white phase modulation noise and frequency phase modulation noise. Thus, other methods exist that allow all noise types to be distinguished from one another, such as the *Modified Allan Deviation (MDEV)* and *Time Deviation (TDEV)*. These are detailed in [17].

2.2.3 Crystal Categories

Quartz crystal oscillators are classified according to how they cope with frequency changes [20] which in turn is reflected in their cost.

The most basic type of oscillator is termed an *RTXO* or *room temperature crystal oscillator*. As its name suggests, it is designed to operate with minimum frequency changes over a defined temperature range. This is achieved by cutting the crystal such that its temperature coefficient is minimised for that particular range. This can result in accuracies of about 2.5 ppm over a temperature range of 50 °C [20]. Outside its designated temperature range, the frequency of the crystal may change more dramatically per degree Celsius.

The second type of oscillator is termed a *TCXO* or *temperature compensated crystal oscillator*. Such oscillators are designed by carefully choosing their individual components so that temperature changes have less of an impact on their nominal operational frequency. TCXOs may also include components that facilitate frequency tuning such as an adjustable capacitor. Unlike RTXOs which typically have a linear frequency vs. temperature graph over their designated temperature range, TCXOs may have a non-linear one. Thus, frequency excursions may occur at particular temperatures within the designated range. In general, TCXOs still out perform RTXOs with achievable accuracies in the order of 0.5 ppm [20].

The most accurate crystal oscillator is the *oven controlled crystal oscillator*. As its name suggests, such an oscillator has its crystal and temperature sensitive elements housed in a temperature controlled oven. Thus, the oscillator operates in a constant temperature environment. The particular temperature chosen is context dependent and is termed the *turn-over point*. This represents a point on the temperature vs. frequency curve with the lowest rate of change, that is, a minima or maxima. In reality the crystal will be cut such that a turn-over point occurs at about 15 to 20 °C above the maximum temperature that the oscillator will be exposed to.

2.3 Time Error

As is evident from the above discussion, the performance of a system clock is influenced to a large degree by the characteristics of the oscillator that drives it. In certain scenarios, a computing system may not require an accurate notion of true time, and in such cases time error will not be an issue. Conversely, a computing system must refer to an external source to determine its time error. The time error of a local clock relative to a reference clock, r, can be modelled using equation 2.8

$$x(t) = x_0 + (y_0 - y_{0,r})t + \frac{D - D_r}{2}t^2 + \frac{\phi(t) - \phi(t)_r}{2\pi V_{nom}} \tag{2.8}$$

where $x(t)$ represents the time error at true time t, x_0 represents the initial time error, y_0 represents the fractional frequency deviation, D represents the rate of change of the fractional frequency deviation and $\phi(t)$ denotes random phase deviations.

This formula can be simplified if the reference clock represents a perfect clock and, thus, never deviates from true time. Hence, the terms $y_{0,r}$, D_r, and $\phi(t)_r$ can then be set to zero leaving equation 2.9

$$x(t) = x_0 + y_0 t + \frac{D}{2}t^2 + \frac{\phi(t)}{2\pi V_{nom}} \tag{2.9}$$

Of course, no perfect clock exists. However, this formula will suffice if the time deviations of the reference clock are negligible in comparison to the required bounds of the time error. If it is required that a system adhere to a globally accepted time-scale that represents the best approximation of true time, then it should refer to a globally accepted time standard such as *Coordinated Universal Time* (UTC).

2.3.1 Coordinated Universal Time

While precise time akin to that recorded by a perfect clock is desirable, it is also necessary that an official timescale concur with the Earths rotation about its own axis. *Coordinated Universal Time (UTC)* offers a compromise between these two

requirements and has been the official time standard since January 1972.

Precise timekeeping necessitates that the basic unit of time, the second, is recorded as defined by the CGPM (*General Conference of Weights and Measures*): '*The second is the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium-133 atom.*'[21]. Precise timekeeping is made possible through the use of devices that can measure such atomic activity and are appropriately termed atomic clocks. With respect to the definition of the SI second (International System of Units), an atomic clock measures such atomic activity, and this activity occurs at a particular rate. In response, the atomic clock generates a signal with a frequency that concurs with the rate of this activity. This is subsequently used to increment a timescale of interest. One must note that the rate of this activity is influenced by the gravitational potential at the point in space the atomic clock is located, and this leads to time dilatation. Thus, in the case of two perfectly stable, accurate, and identical atomic clocks operating at different altitudes, the rate at which each generates ticks differs from the other. Velocity too leads to time dilation and, thus, the tick rate of two perfect, identical atomic clocks that are subjected to the same gravitational potential but travelling at velocities of different magnitudes also differ. Atomic clocks, however, are not perfect. Their stability and accuracy vary. As such, the primary atomic-based time standard *International Atomic Time (TAI)* is formulated using approximately 300 commercial atomic clocks located throughout the world. The signals generated by these clocks are combined in a weighted average to produce the best stability of frequency. This frequency is then steered to an ensemble of a small number of laboratory caesium clocks that have the best accuracy. This approach allows for a fractional frequency offset from the definition of the second as low as 10^{-16}. Thus, the TAI time standard is optimised to best realise the SI second. From this discussion one may observe that the actual time is a human artifact.

The Earth's rate of rotation about its axis is not constant and, therefore, does not represent a quality time reference. The timescale produced using the Earth as a reference is termed *Universal Time (UT)* and is equivalent to *Greenwich Mean Time (GMT)*. It would be unreasonable to expect people to function around the TAI timescale solely, and, thus, UTC combines both scales through the use of leap

seconds. By definition, a leap second, associated with the Earth's non-uniform rate of rotation, may be appended at the end of any month as recorded by TAI. The determination of required leap seconds is made by the *International Earth Rotation Service (IERS)* via data collected from numerous observatories.

2.3.2 Global Navigation Satellite System

UTC is concerned with the production of a precise global time-scale and leaves the distribution of this scale to other systems. One system that distributes high precision time very well is the *NAVSTAR Global Positioning System (GPS)*. GPS distributes both GPS time and UTC time. GPS time differs from UTC by an integer number of seconds. In 1980, when the GPS system began recording time, GPS time began to diverge from UTC time due to the systems inability to accommodate leap seconds. The GPS specifications dictate that it is kept to within a microsecond of UTC time although in reality it is kept within the nanosecond range minus the additional leap seconds which have been removed from TAI time to form UTC time since 1980.

The Global Positioning System itself consists of an array of satellites that span the globe at an altitude of approximately 20,200 Km. Each satellite contains four atomic clocks (see section 2.3.1), all of which are synchronised to each to form the GPS time-scale. The satellites are controlled by a *Master Control Station (MCS)* located in Colorado, U.S.A. The MCS, with the aid of a number of monitor stations strategically located around the globe, continuously monitor the position of each satellite along with the timing signals produced by each satellite. Any time or position related corrections are uploaded to each satellite by the MCS.

The GPS system is strictly a ranging system. Within a ranging system, a receiver of a signal determines its position relative to a transmitter of the signal by calculating the propagation time of the signal. In the GPS system, each satellite periodically broadcasts its own specific signal, and receivers use the propagation time of this signal, as governed by the speed of light, to determine their distance from a satellite. This process is referred to as three-dimensional *trilateration* and is illustrated in fig. 2.5.

A signal originating from a single satellite allows a recipient to construct

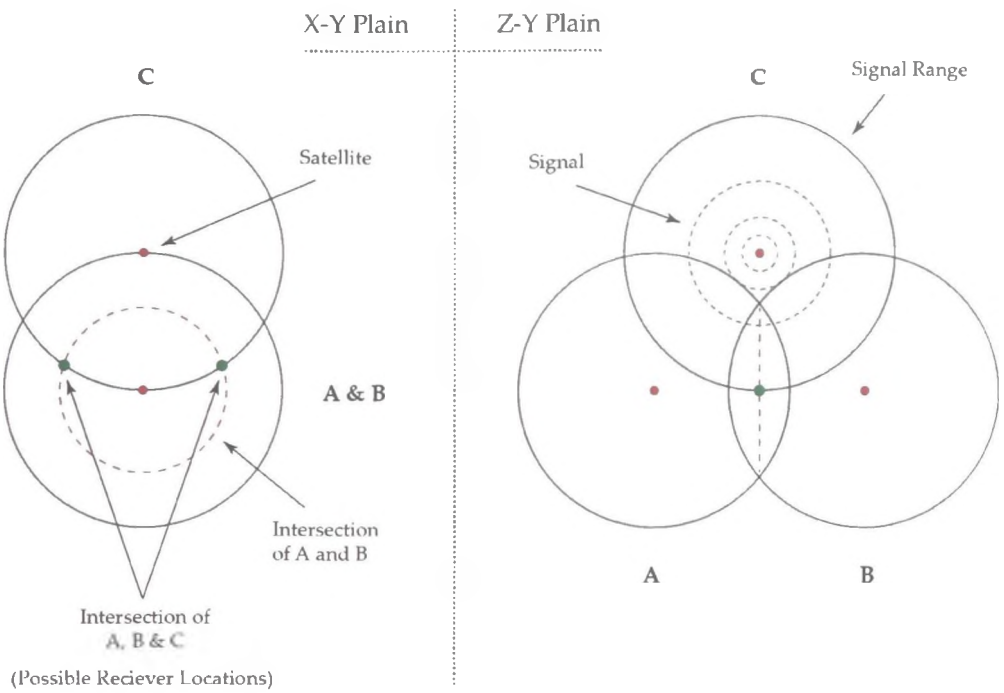


Figure 2.5: Trilateration

an imaginary sphere centred at that satellite. The recipient's location lies at a particular point on the surface of this sphere. With two signals originating from two separate satellites, the recipient can construct two imaginary spheres that intersect. The points of intersection of these spheres form a circle upon which the recipient is located. In order to determine its position in three-dimensional space, a third satellite is required. With a third signal the recipient can construct three spheres that intersect at two distinct points representing two possible locations. Signals obtained from three satellites will usually suffice, since one of these points will typically not lie on the Earths surface. The recipient uses the positional and orbital data (*Ephemeris*) superimposed on each satellites timing-signal to determine its position relative to the Earth.

This positional data, however, will only be accurate if the receiver is also synchronised to GPS time. Unlike GPS satellites, which employ atomic clocks, GPS receivers generally employ inexpensive quartz crystal clocks and, therefore,

will typically lack the relative accuracy. In this case, a fourth satellite can be used by the receiver to determine its time error. This is achieved by calculating the correction required to make the four respective spheres intersect at a single point. Thus, this eliminates the need for receivers to employ expensive atomic clocks.

2.4 Time Synchronisation

It is clear now that a system clock that relies on a crystal based oscillator as its frequency standard inevitably accumulates time error with respect to an official time standard such as UTC. This is also the case with atomic clocks that are synchronised to a UTC timescale, yet the rate at which an atomic clock accumulates error is orders of magnitude lower. The rate at which a clock driven by a crystal based oscillator accumulates error can be alleviated, to a certain extent, by employing expensive variants of crystal based oscillators. To improve further on this, and where a view of the sky is available, systems can employ a GPS receiver. An ideal solution just short of employing an atomic clock is a combination of both, that is, a GPS receiver used in conjunction with an oven controlled crystal oscillator. Such an infrastructure is relatively expensive but is widely deployed in time critical systems such as electric power systems and telecommunications systems. Nonetheless, there is a requirement for techniques that ensure that standard system clocks driven by inexpensive crystal based oscillators adhere to UTC time. A number of synchronisation techniques fulfil this role, though they can be subject to various sources of error. These techniques are described in the following sections.

2.4.1 Sources of Synchronisation Error

If two computing systems, hereafter referred to as node A and node B, connected via a communication link wish to synchronise their clocks, then they must exchange some information regarding the state of their clocks. Thus, node A might transmit a message containing a timestamp to node B. Node B can then set its clock to the value of that timestamp. This scenario is illustrated in fig. 2.6. Here,

node A is referred to as the *reference* and node B is termed the *host*. This represents one of the most basic synchronisation techniques and is prone to multiple sources of synchronisation error.

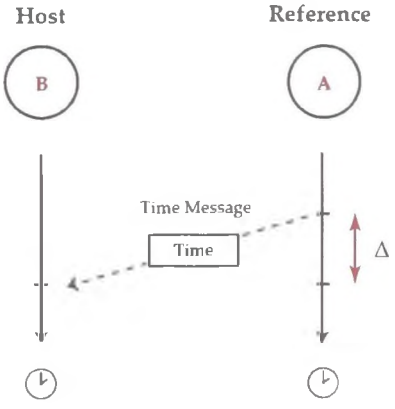


Figure 2.6: Uni-directional synchronisation

Firstly, node B does not accommodate for the time it takes the message to traverse the entire communication link between A and itself. This time is termed the *traversal time*. Thus, if node B sets its clock to the value received in the message, it will have a time error equivalent to the traversal time of the message. Node B could accommodate for the traversal time if it was known but, in most cases, it is non-deterministic. This is due to sources of non-deterministic latencies that exist along the communication path. They are categorised by Kopetz and Ochsenreiter [22] as the *send time*, *access time*, *propagation time* and *receive time* and are illustrated in fig. 2.7.

- The *send time* represents the time interval between the recording of a timestamp by a transmitter's synchronisation application and the delivery of a message, containing that timestamp, to the network interface for transmission. The time taken to construct the message is influenced by the underlying system. For instance, in a general purpose computing system, the process that constructs the message is subject to a scheduling algorithm

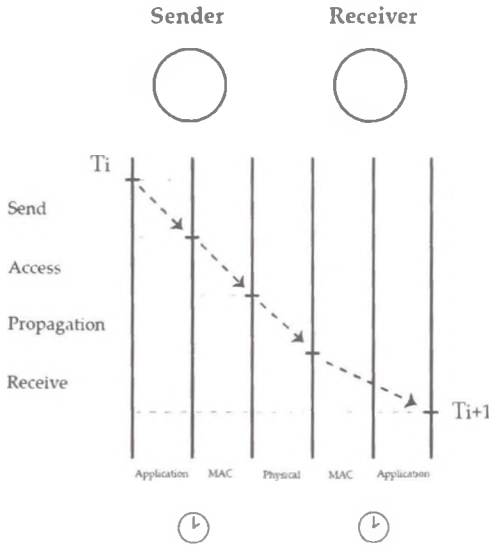


Figure 2.7: Components of message latency

which can block it numerous times during its operation. Furthermore, additional delays can be incurred due to system call overheads.

- The *access time* represents the delay incurred by the network interface while waiting to gain access to the communication medium. This is dictated by the *Medium Access Control (MAC)* protocol in use. For instance, the traditional wired Ethernet (CSMA-CD) and wireless Ethernet 802.11 (CSMA/CA) protocols use contention based approaches, meaning a node may not begin transmission until the channel is clear, thus, high traffic loads can lead to large access delays. While most wired Ethernet networks are switched and full duplex, thus, eliminating such contention, 802.11 networks are not and, therefore, contribute greatly to this problem (This topic is dealt with in detail in chapter 6).

- The *propagation time* represents the time taken for a message to traverse the communication link between two nodes. In a single hop network, where the sender and receiver are connected directly by the same physical medium,

the propagation time is dictated by the speed of light and, therefore, is deterministic if the distance between them is known. If, however, the sender and receiver are connected via multiple network nodes, this time is non-deterministic, since the message is subjected to multiple queue and access delays at each intermediate node.

- The *receive time* represents the time taken for the receiver's network interface to receive the message from the communication medium, decode it, and notify the host application that it has arrived.

While all of these components may be non-deterministic, much of this non-determinism can be eliminated by time stamping message transmission and reception events at lower levels in the communication hierarchy. If this is not possible, then an appropriate synchronisation technique should be employed to alleviate the negative effects of these delays.

2.4.2 Synchronisation Techniques

In general, one of three core synchronisation techniques/approaches are used to synchronise the clocks of distributed nodes to a common time reference. These techniques are detailed in [23, 24]. The simplistic synchronisation technique detailed in the previous section is referred to as *uni-directional synchronisation* by Römer et al. [23]. This technique is generally not effective at meeting the accuracy requirements of many time-sensitive applications, unless associated time messages are time stamped at lower layers in the communication stack. In fact, this technique is only appropriate if the accuracy requirements of the host are coarser than the maximum message latency between the host and reference.

A second more effective synchronisation technique is *round-trip synchronisation* [23], which is illustrated in fig. 2.8. Here, node B, the host, initiates the communication exchange by transmitting a *request message* to node A, the reference node. Node B records the transmission time, T_i, of this request message. The message traverses the link and arrives at A at which point A records the reception time, T_{i+1}, of the message. Node A then constructs a *response message* which it transmits back to B. This process requires that A records the transmission time of the response message, T_{i+2}, and places the timestamps T_{i+1} and T_{i+2}

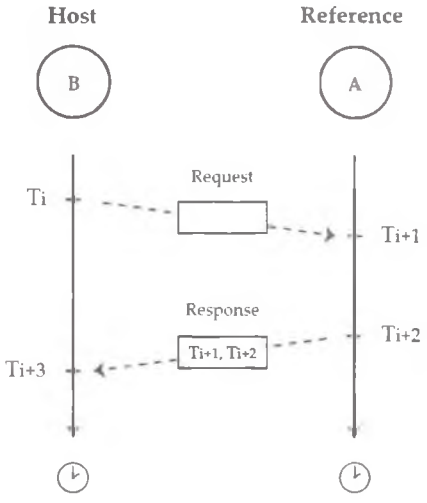


Figure 2.8: Round-trip synchronisation

into this message. Node B subsequently receives the response message, records its reception time, T_{i+3}, and uses the four acquired timestamps to determine the *round-trip delay (δ)* of the message. By eliminating the processing time at A and subtracting the down-link delay from the up-link delay, an approximation for B's time error relative to A can be determined. This error represents the *phase offset (θ)* between A and B and is calculated using equation 2.11.

$$\delta = (T_{i+3} - T_i) - (T_{i+2} - T_{i+1}) \tag{2.10}$$

$$\theta = \frac{(T_{i+1} - T_i) + (T_{i+2} - T_{i+3})}{2} \tag{2.11}$$

Round-trip synchronisation can, in certain circumstances, be an effective method for determining the traversal time of a message, thus, improving the estimate of a host's offset from its reference. Nevertheless, it makes one core assumption that turns out to be generally untrue in real world scenarios, that is, the latency of a request message is equal to the latency of the corresponding response message. The issue with this assumption is illustrated in fig. 2.9. In the scenario depicted in fig. 2.9, the latency of the response message exceeds

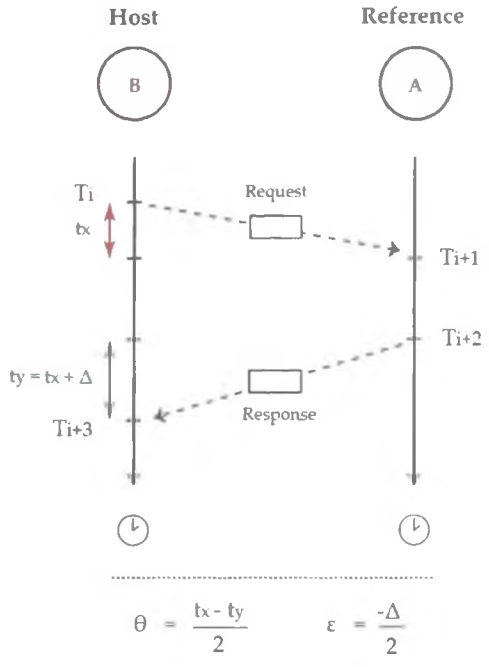


Figure 2.9: Asymmetric delay

that of the request message by a magnitude denoted by Δ. If the nodes are in fact synchronised, then this time difference results in an estimated offset error of $-\Delta/2$. Thus, although this technique outperforms uni-directional synchronisation, it is also subject to the negative effects of non-deterministic message latencies, although to a lesser extent.

A third type of synchronisation technique referred to as *reference broadcast synchronisation* [23] takes a very different approach to the aforementioned techniques. This technique takes advantage of the broadcast nature of certain communication links in an effort to eliminate the sources of non-deterministic message latencies at a transmitter/sender, namely the *send time* and *access time*.

The operation of this technique is illustrated in fig. 2.10. Here it is assumed that all nodes involved in the synchronisation operation are connected via a communication link that permits message broadcasts. A special node termed a *Reference Broadcast Node (RBN)* initiates the synchronisation process by broadcasting

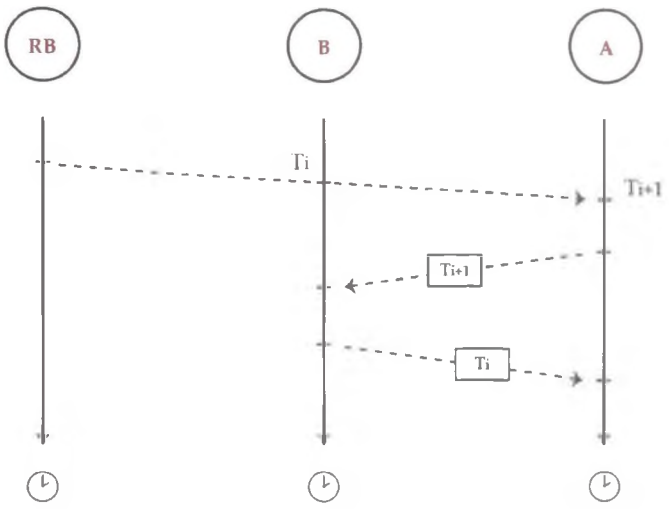


Figure 2.10: Reference broadcast synchronisation

a *beacon message*. The contents of the beacon message are not important. However, it is essential that all nodes involved in the synchronisation process can detect it. Nodes A and B receive the message at times T_i and T_{i+1} respectively. Subsequently, nodes A and B exchange their reception times. A comparison of the reception times allows each node to determine its offset from the other and, thus, provides a means to transform the timescale of one node into that of the other.

It must be noted that, while the send time and access time are eliminated, the reception time, and to a lesser extent, the propagation time are still present. Thus, they present possible sources of synchronisation error. Since the beacon message is broadcast, the magnitude of the propagation delay will be proportional to the difference in distance between node A and node B relative to node RB. In reality, non-deterministic delays associated with the send time and access time dominate so reference broadcast synchronisation can be relatively effective in particular scenarios.

2.4.3 Clock Skew and Drift

The synchronisation techniques detailed in the previous section provide a means for a host node to determine its phase offset from a reference node. If both nodes employ extremely accurate and stable oscillators, then it is sufficient to perform the synchronisation process only once. In reality, however, clocks will have a frequency error that may change over time necessitating multiple synchronisation rounds.

A frequency error that exists between a host and a reference results in *clock skew (λ)*. Clock skew is defined as the rate of change of a host's time with respect to a reference's time. A host may determine its clock skew relative to a reference by performing two synchronisation rounds. The skew is then calculated as the difference between the two estimated offset values (θ_i, θ_{i+1}) divided by the *synchronisation interval (τ)* (see equation 2.12). The synchronisation interval, τ, represents the time interval between two consecutive synchronisation rounds.

$$\lambda_i = \frac{\theta_i - \theta_{i+1}}{\tau} \tag{2.12}$$

While two synchronisation rounds is the minimum required to determine a host's skew, the accuracy of the estimated skew value depends on a number of factors. These include the time interval between synchronisation rounds and the magnitude of time error that is caused by non-deterministic message latencies. If the time interval between synchronisation rounds is too short, then message latency related errors typically dominate. If it is too long, then the stability of a clock becomes a factor, since the clock's frequency offset may change within the interval and contribute to a greater proportion of the time error.

The quality of a skew estimate can be improved by analysing multiple offset estimates using a technique termed *least squares linear regression* [25]. Linear regression provides a means to postulate a linear relationship between a host's time and reference's time. This relationship can be expressed using equation 2.13 and is illustrated in fig. 2.11.

$$T_r = \theta + \lambda T_h \tag{2.13}$$

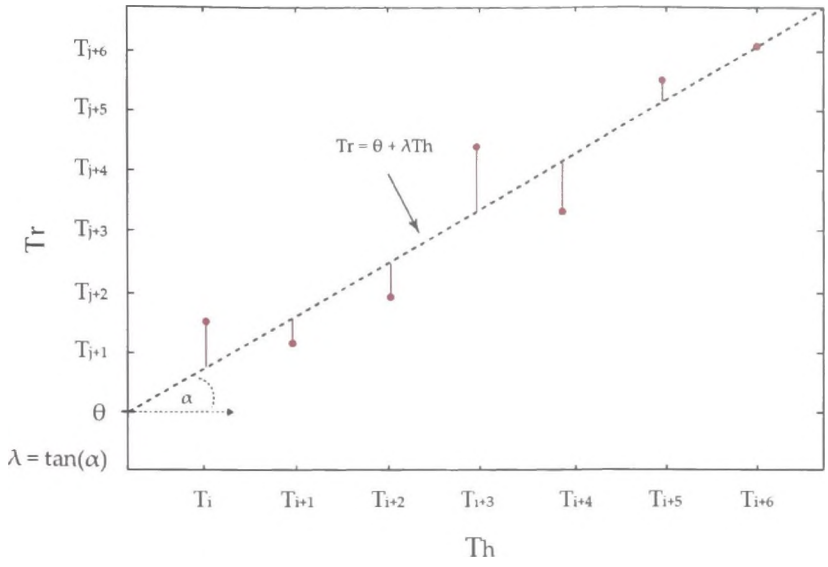


Figure 2.11: Linear regression

In equation 2.13 T_r denotes the time at the reference node and T_h denotes the time at the host node. The benefit of using linear regression with multiple data points is that it is more resistant to erroneous offset estimates. Consequently, the calculated skew value is generally more accurate.

The second derivative of a node's time with respect to a reference's time is referred to as *clock drift* (ϕ). Clock drift is the direct result of oscillator instability and represents the rate of change of a host's skew with respect to a reference's time. Clock drift can be determined with two or more skew estimates as shown in equation 2.14.

$$\phi_i = \frac{\lambda_i - \lambda_{i+1}}{\tau} \tag{2.14}$$

As detailed in section 2.2.2, oscillator frequency variations are the result of crystal aging, oscillator circuitry noise, and numerous environmental factors such as temperature changes. Thus, with respect to synchronisation, an accurate value for a host's clock drift at a particular point in time can be difficult to determine. Consequently, most synchronisation algorithms operate on the assumption that clock drift remains constant between synchronisation rounds. Thus, by frequently

updating the clock skew, the negative effects of clock drift can be indirectly alleviated. With an estimate of its clock's skew and knowledge of the maximum permitted time error, ϵ_{max}, a host can bound the synchronisation interval as expressed in equation 2.15.

$$\tau \leq \frac{\epsilon_{max}}{\lambda} \tag{2.15}$$

2.4.4 Summary

This chapter has presented the reader with an overview of the operation of a general purpose clock and explained why it accumulates time error and how it can be disciplined. The core component of a clock is its oscillator. The element that drives an oscillator is termed its frequency standard and typically consists of a quartz crystal that operates under a physical phenomenon referred to as piezo-electricity. Crystals are employed because of their low-cost and relative stability. Stability and accuracy represent two important characteristics of an oscillator and dictate its quality. A stable and accurate oscillator accumulates time error at a lower rate with respect to true time than an unstable and inaccurate oscillator. True time in this context is the time-scale recorded by a perfect clock. Since perfect clocks do not exist, this time-scale is theoretical. With respect to the formal definition of the second, the globally accepted representation of true time is UTC time. The UTC time-scale is tracked using a global array of atomic clocks and the GNSS system provides a means to distribute this time-scale.

For critical infrastructures, such as power systems and telecommunications systems, a combination of a GNSS receiver and oven controlled crystal oscillator may be employed. Such an approach can be infeasible and impractical in many situations. Thus, software synchronisation techniques are typically employed. To synchronise the clocks of two computing systems connected via a communication link, one of three techniques is generally used: uni-directional synchronisation, round-trip synchronisation, or reference-broadcast synchronisation. These techniques allow one to determine the phase offset between the two clocks. Due to the effects of clock skew and clock drift, which are the direct result of inaccurate and unstable oscillators, continuous synchronisation rounds are required to

ensure constant synchronisation.

Part II

Time Synchronisation in Wireless Sensor Networks

Chapter 3

Synchronisation Techniques for Wireless Sensor Networks

Chapter 2 of part I focused on the details of a generic system clock, in particular, its operation and its core component, the quartz crystal oscillator. The motivation for time synchronising the system clock of a computing system was highlighted and several generic time synchronisation techniques were outlined. This chapter elaborates on the aforementioned time synchronisation techniques by focusing on distinguished synchronisation techniques and protocols employed within the domain of *Wireless Sensor Networks (WSNs)*. The operation of these techniques and protocols are presented in detail and followed by a comparative analysis. The chapter concludes with a discussion that highlights the need for an energy efficient, responsive time synchronisation protocol that can operate in dynamic environments.

3.1 Introduction

Time synchronisation plays a critical role within real-time distributed systems. While this has always been the case, the continuous development of more advanced and diverse systems places an ever-increasing demand on time synchronisation facilities in terms of reliability, accuracy, and efficiency. Furthermore, security requirements may place additional demands on these facilities. Many

systems require that synchronisation facilities provide, or employ, an authentication system to ensure that time information is delivered across a communication link with integrity and from a recognised source.

Advances in integrated circuit design and fabrication techniques have given rise to the development of miniature computing devices which, with respect to their size, have relatively sophisticated capabilities, such as environmental sensing and radio communication. These devices are generally referred to as Microelectromechanical systems (MEMS). This technology in turn has spawned a new area in distributed data collection termed *Wireless Sensor Networks (WSN)*. While wireless sensor networks are nothing more than a number of dispersed miniature sensing devices connected via a wireless communication protocol, their applications are endless.

The individual nodes of a WSN are composed of four fundamental components. These components deal with data sensing/input, data communication, data processing and energy supply. Data input is provided by means of sensors that measure physical phenomena and convert the corresponding analogue signals (via an ADC) into their digital representation. Such physical phenomena include temperature, light intensity, pressure, humidity, magnetism, sound and many others.

Naturally, the collection of raw sensory data is an important facet but is usually inadequate if not combined with other sources of temporally or spatially related data. Data collected from multiple sources is typically collated, processed, and analysed by some central computing system in order to fulfil the goal of the application. Hence, it is necessary that nodes communicate with each other so that data can be collected and/or disseminated. A transceiver fulfils this role and is a critical component in such networks.

In the context of data processing, it can be beneficial to delegate some of the less intensive pre-processing tasks to the nodes themselves. This can significantly reduce the communication overhead of WSN applications. To do this, it is required that the nodes possess a relatively powerful microcontroller with a reasonable quantity of short term memory. The use of a microcontroller permits such pre-processing tasks as data validation, data filtration, and data collation, all of which can reduce the size of messages and the number of message exchanges.

Finally, in relation to energy, to power the aforementioned modules, a typical WSN node will contain a battery. Needless to say, this energy source is limited and, therefore, an important constraint in WSNs is low power consumption. Such a constraint dictates that the hardware and software components that comprise a WSN node are designed accordingly. Some WSNs may be deployed in inaccessible or hostile environments meaning that the length of their lives is dictated by their nodes' battery lives.

While a single WSN node is useful in its own right, the full potential of these miniature computing devices is revealed when a large number of them are networked together to form a distributed system, the applications of which are numerous. More often than not, the data collected for such applications must have temporal meta-data associated with it and, thus, a time synchronisation protocol of some sort must be employed.

When a time synchronisation protocol is required by a WSN application, its absence, or malfunction, either severely degrades the performance of the application or hinders its value completely. This is evident in the case of a simple structural health monitoring (SHM) application whereby static nodes record some physical phenomena such as structural vibration. Knowledge of the rate of vibration at one particular location would not suffice to determine the integrity of the structure as a whole. Instead, measurements collected from a large sub-set or possibly the entire structure at a single point in time would be required. Consequently, the collected data would need to be time-stamped by time synchronised nodes.

Time synchronisation services play an important role in many high level WSN applications, particularly those concerned with monitoring such as the environmental, structural, and habitat applications described by Cao et al. [26], Paek et al. [27] and Cerpa et al. [28] respectively. Their use, however, is not limited to the simple task of time-stamping data. Lower level applications, such as MAC scheduling applications and localisation applications, necessitate a common notion of time among WSN nodes and sometimes to a greater degree than higher level applications.

WSN Localisation techniques, such as the one described by Girod and Estrin [29], allow nodes to determine their location in space relative to each other or some

fixed point. Spatially enhanced data is particularly important when a WSN is comprised of mobile nodes or when the initial location of static nodes is unknown due to the deployment procedure employed. One such localisation technique referred to as *Time Of Flight (TOF)* ranging has a receiver node determine the relative position of a sender node by calculating the propagation time of a signal transmitted by the sender. Such a technique requires tight time synchronisation between the sender and receiver. The achieved synchronisation accuracy is highly influenced by the signal propagation speed. RF signals propagate through air at a much greater speed than acoustic signals and, thus, their use necessitates more precise synchronisation. As the synchronisation accuracy degrades so too does the node's estimate of its location, thus, degrading the performance of the WSN application.

MAC scheduling protocols, such as DMAC [30] and TRAMA [31], are another category of applications that necessitate time synchronisation among nodes. MAC scheduling techniques form a very important element of WSN networks. This is largely due to the energy constraints imposed on applications by battery-powered nodes. Wireless communication is an energy-intensive process, and, thus, MAC techniques used in conventional wireless networks, such as 802.11, are not especially suited to this domain. 802.11 CSMA techniques require that nodes remain in a state termed *busy listening* whereby they continuously listen to the wireless medium for incoming messages. Unfortunately, busy listening uses a substantial amount of energy that could otherwise be used more productively by battery-powered devices. Scheduling protocols, such as DMAC and TRAMA, remedy this issue by forcing nodes to enter *sleep* and *wake* states at regular intervals. This reduces, or eliminates, the time a node spends in the busy listening state. Such scheduling protocols may require time synchronisation accuracies in the order of milliseconds so that nodes can coordinate message exchanges among each other.

3.2 Static WSN Time Synchronisation

In the context of this work, time synchronisation protocols are classified into two categories: *static synchronisation* protocols and *dynamic synchronisation*

protocols. The distinction between each type is based on whether a particular synchronisation protocol possesses the ability to re-configure its transmission rate post-deployment. A static synchronisation protocol will not possess this functionality, whereas the latter will.

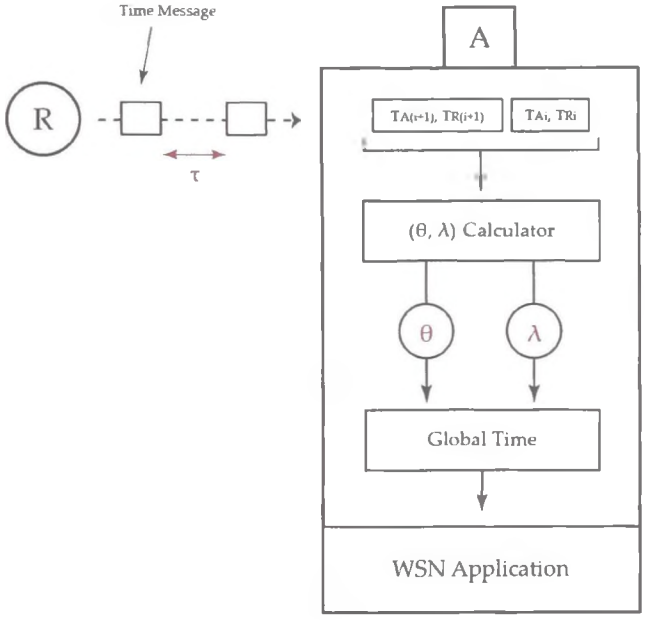


Figure 3.1: Static synchronisation

The structure of a static synchronisation protocol is depicted in fig. 3.1. Typically, a node, A, that wishes to synchronise with a reference node, R, does so by obtaining a reference point via a two-way message exchange (round-trip synchronisation) or a single broadcast (uni-directional synchronisation). Node A then uses this information to determine its phase offset, θ. A reference point consists of a local and reference timestamp pair, (T_{Ai}, T_{Ri}), whereby each timestamp relates to the same instant in real time. The phase offset between node A and, its reference, R at a particular point in time is the difference between the corresponding local and reference timestamps (see equation 3.1).

$$\theta_i = T_{Ai} - T_{Ri} \tag{3.1}$$

As explained in section 2.2.1, a quartz crystal oscillator typically has a fre-

quency which differs from its nominal/stated frequency. Consequently, the phase offset between any two nodes in a WSN will change over time. Thus, in order for node A to keep its timescale within a specific threshold of node $R's$, it must continuously obtain time messages from R. It is clear that in certain scenarios, for instance, when the accuracy requirements of a WSN application are quite stringent, the communication overhead between nodes can become unacceptably high. For this reason, a static synchronisation protocol may employ multiple reference points in conjunction with regression analysis to determine a node's frequency error relative to its reference. This frequency error, which is detailed in section 2.4.3, is referred to as clock skew (λ). If node A is aware of its clock skew it can use this information to correct its timescale and, thus, no further time messages are required from node R.

The key feature that distinguishes a static synchronisation protocol from a dynamic one is its static transmission interval, denoted τ. The value of this transmission interval is an important factor when a WSN is deployed in an environment that subjects nodes to varying ambient conditions, such as temperature or pressure fluctuations. The reason for this, as detailed in section 2.2.2, is that such conditions can alter the oscillation frequency of a quartz crystal oscillator. This change in frequency, referred to as clock drift (ϕ) in this text, can produce undesirable time errors if a node does not receive time messages at an adequate rate. Thus, an accuracy-energy trade off must be reached. A suitable transmission interval must be chosen so that a WSN's accuracy requirements are met with the least amount of energy consumption. Such a task is typically carried out via an empirical analysis of the environment, but this procedure can be time consuming, and it may not produce an optimal value, particularly if the environment is volatile.

3.3 Static Time Synchronisation Protocols

Having described the fundamentals of a generic static time synchronisation protocol, the following sub-sections outline some noteworthy examples.

3.3.1 Reference Broadcast Synchronisation

Reference Broadcast Synchronisation (RBS), by Elson et al. [11], employs the reference broadcast synchronisation technique described in section 2.4.2 to time synchronises a *cluster* of nodes, each of which shares a direct communication link with every other. Within a cluster, a particular node is elected the *beacon node* and, consequently, allocated the task of broadcasting a beacon message at regular intervals. The reception time of each beacon message is recorded by all nodes within the cluster. These nodes subsequently exchange their reception timestamps, thus, enabling them to determine the relationship between each other's time-scales. Thereafter, each node can construct a relative time-scale for every other node within the cluster, thus, permitting them to transform their time-scale into that of another.

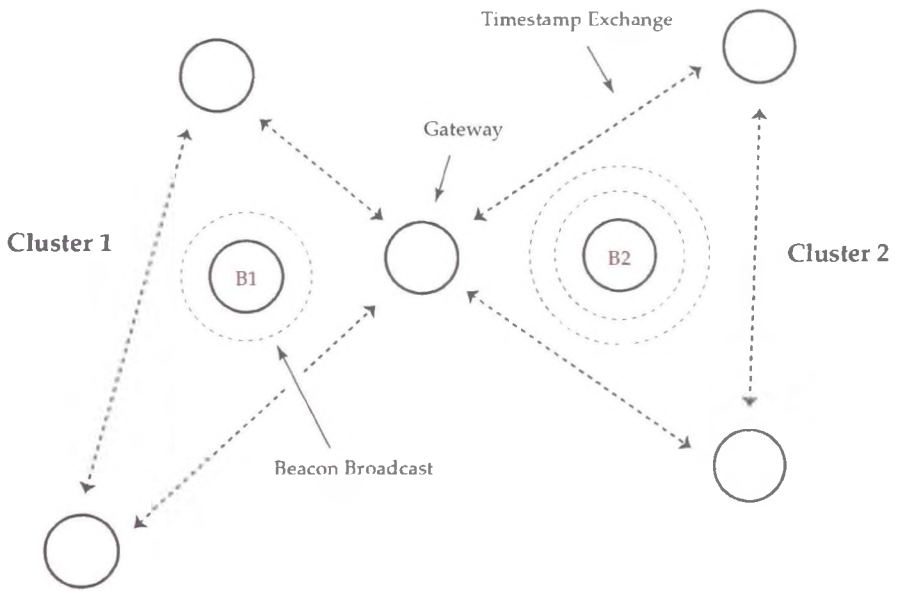


Figure 3.2: RBS operation

Since RBS is only concerned with the reception time of messages, it eliminates the two major sources of synchronisation error that other synchronisation techniques are more susceptible too, namely the *send time* and the *access time*.

Consequently, the two remaining sources of error, the *propagation time* and *receive time* of a beacon message, are the only issues RBS must contend with. The propagation time of a message is dictated solely by the speed of light and, in most scenarios, the associated errors are negligible, provided nanosecond precision is not required. Thus, the leading source of error is the receive time or, more specifically, non-deterministic delays associated with the receive time of a beacon message. Naturally, the magnitude of this delay is dependent on the layer in the communication hierarchy that a beacon reception event is time-stamped.

Studies to characterise the receive error at nodes that employ physical layer time-stamping reveal errors with a Gaussian distribution and a mean of zero. These results suggest that the accuracy achieved by a basic form of RBS can be improved further by forcing nodes to average multiple offset estimations in order to improve the estimate. RBS does this and, in addition, explicitly deals with clock skew by employing linear regression.

To extend RBS to operate in a multi-hop network comprised of nodes that do not all share a direct communication link with one another, nodes are organised into mutiple clusters, each of which contains one or more nodes that also belong to one or more neighbouring clusters (see fig. 3.2). Thus, these nodes act as gateways that translate time-scales between clusters. Studies performed on multi-hop networks of this type reveal that the synchronisation errors introduced at each hop are independent, and the total path error is equal to the sum of independent Gaussian variables ($\sigma\sqrt{N}$).

3.3.2 Timing-sync Protocol for Sensor Networks

The *Timing-sync Protocol for Sensor Networks (TPSN)*, by Ganeriwal et al. [12], organises a network into a tree hierarchy, the root of which acts as the source of time. Nodes at higher levels in this tree synchronise with nodes at lower levels using the round-trip synchronisation technique.

Construction of the tree hierarchy is termed the *Level Discovery phase* (see fig. 3.3). This phase is initiated by the root, a node that has been chosen explicitly because of its enhanced capabilities or by some automatic election process. The root assigns itself a level of zero in the hierarchy and then constructs

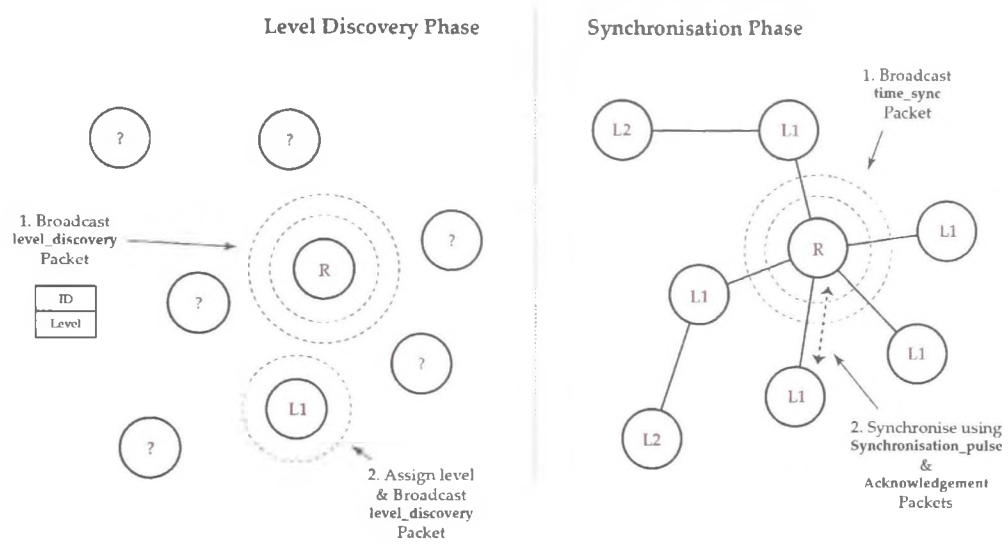


Figure 3.3: TSPN operation

a *level_discovery* packet which contains its *identity* and *level*. It subsequently broadcasts this packet. Recepients use this packet to determine their level and then broadcast their own *level_discovery* packets. Ultimately, this results in the construction of a synchronisation hierarchy. In order to accommodate special cases, such as when a node fails to determine its level due to packet collisions or when a new node joins a network after the level discovery phase, a node may transmit a *level_request* packet which in turn initiates the transmission of a *level_discovery* packet by a recipient. The level discovery phase is ultimately followed by the *Synchronisation Phase*.

As illustrated in fig. 3.3, the synchronisation phase is initiated by the root when it broadcasts what is termed a *time_sync* packet. The reception of this packet by a node at level 1 triggers a synchronisation round between that node and the root. The node transmits a *synchronisation-pulse* packet to the root and in turn the root responds with an *acknowledgement* packet. This procedure allows the node to collect the information required to determine its offset relative to the root and, thus, correct its clock. Synchronisation rounds at lower levels in the tree initiate synchronisation rounds at higher levels, since message exchanges are overheard at higher levels.

Since TPSN uses round-trip synchronisation, it is vulnerable to all the sources of synchronisation error. To mitigate the negative effects of non-determinism associated with the send time, access time, and receive time of a message, TPSN employs MAC layer time-stamping. The use of MAC layer time-stamping is practical in WSNs since the architecture of a WSN node is more coupled than that of a traditional computing system and, thus, the generation of timestamps at this layer is a trivial task. In terms of clock skew, TPSN does not employ regression analysis but relies on re-synchronisation which can in certain scenarios result in significant communication overhead.

3.3.3 Tiny-Sync and Mini-Sync

Tiny-Sync and Mini-Sync (TS/MS), by Sichitiu and Veerarittiphan [32], attempts to minimise the complexity involved in synchronising a WSN in terms of communication, processing, and storage overheads. Synchronisation between a node and its reference is accomplished using a *data collection* technique and a *line fitting* technique that produce a bounded estimate for the clock offset and skew. TS/MS makes no assumptions about the presence of components that lead to non-deterministic message latencies and provides a pure mathematical technique to mitigate error.

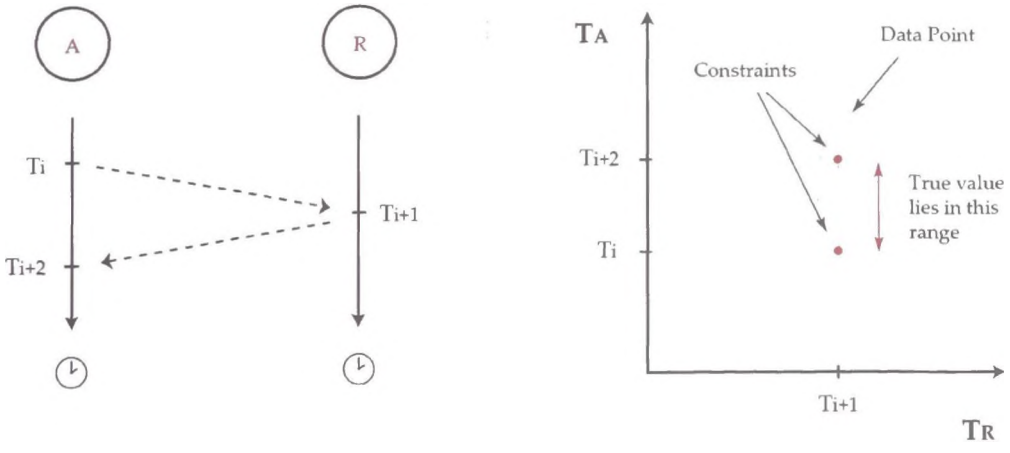


Figure 3.4: TS/MS: data collection (left) and data pont (right)

The data collection technique, illustrated in fig. 3.4, produces the three timestamps T_i, T_{i+1} and T_{i+2}. The timestamp, T_i, represents the transmission time of a request message sent by node A. Node R generates the timestamp T_{i+1} on receipt of the request message and immediately replies with a response message. Node A on receipt of the response message generates the timestamp T_{i+2}. These timestamps are used to produce a data point, (T_i, T_{i+1}, T_{i+2}), composed of the two constraints (T_{i+1}, T_i) and (T_{i+1}, T_{i+2}), as shown on the right of fig. 3.4. It is clear from fig. 3.4 that if the message latency between node A and R increases, then so too will the range of possible values that represent the true time at node A when node R records T_{i+1}.

Multiple message exchanges result in multiple data points which are used to bound the offset(θ) and skew(λ) of node A within the ranges $(\underline{\theta_{AR}}, \overline{\theta_{AR}})$ and $(\underline{\lambda_{AR}}, \overline{\lambda_{AR}})$ respectively. The upper and lower bounds for the offset and skew are calculated using those data points that minimise the bound ranges. They are illustrated in fig. 3.5. The final estimate of a node's offset and skew is calculated as the line that best fits within the bounds. This line is estimated using equations 3.2 and 3.3.

$$\theta = \frac{\underline{\theta_{AR}} + \overline{\theta_{AR}}}{2} \pm \frac{\overline{\theta_{AR}} - \underline{\theta_{AR}}}{2} \tag{3.2}$$

$$\lambda = \frac{\underline{\lambda_{AR}} + \overline{\lambda_{AR}}}{2} \pm \frac{\overline{\lambda_{AR}} - \underline{\lambda_{AR}}}{2} \tag{3.3}$$

The technique described so far represents the core mechanism employed by the two algorithms Tiny-sync and Mini-Sync. The distinction between Tiny-sync and Mini-Sync is based on the number of data points each one stores. Tiny-sync does not store more than two data points at a time and discards the two worst constraints after each data collection round (i.e. the constraints that maximise the bound ranges). While this is efficient, it does not guarantee optimal results. Mini-sync, on the other hand, remedies this by storing more data points and only eliminates those timestamps that do not satisfy a particular condition. Mini-sync, however, requires more storage and computations.

In relation to multi-hop synchronisation, the assumption is made that most WSNs are organised into tree hierarchies within which collected data is trans-

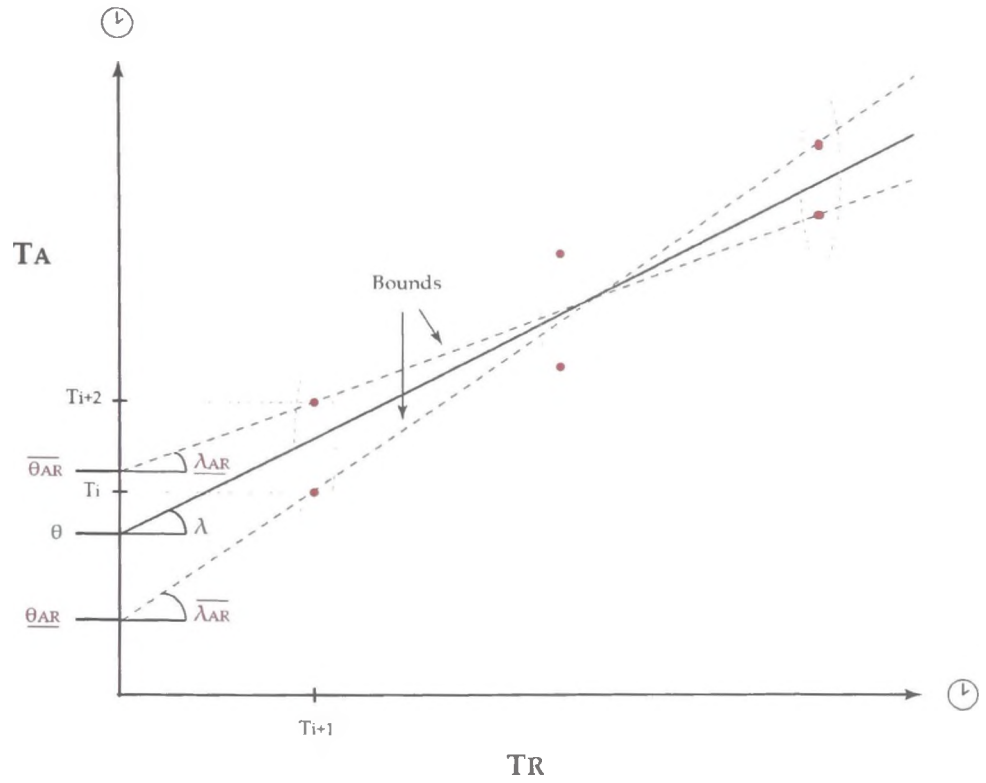


Figure 3.5: TS/MS: bounding the offset and skew

ported back to a root node with all intermediate nodes fusing/combining the sensed data along the way. Given this structure, Sichitiu and Veerarittiphan [32] propose that if nodes only synchronise with nodes that combine data, then this leads to better accuracy than if every node attempted to synchronise with a unique clock. Thus, if a node, A, is responsible for combining data received from one or more nodes, then those nodes should synchronise with A as this should provide better synchronisation accuracy. While this technique might prove favourable in such a network topology, it might not prove so in less conventional topologies.

3.3.4 Lightweight Time Synchronisation

The *Lightweight Time Synchronisation (LTS)* protocol, by van Greunen and Rabaey [33], attempts to keep a WSN time synchronised within a specified threshold using a minimum number of packet exchanges. Synchronisation between pairs of nodes is performed using round-trip synchronisation. Of course, this is nothing novel, but LTS focuses more on a network as a whole. It proposes two types of multi-hop synchronisation schemes, namely *centralised multi-hop LTS* and *distributed multi-hop LTS*. These are illustrated in fig. 3.6.

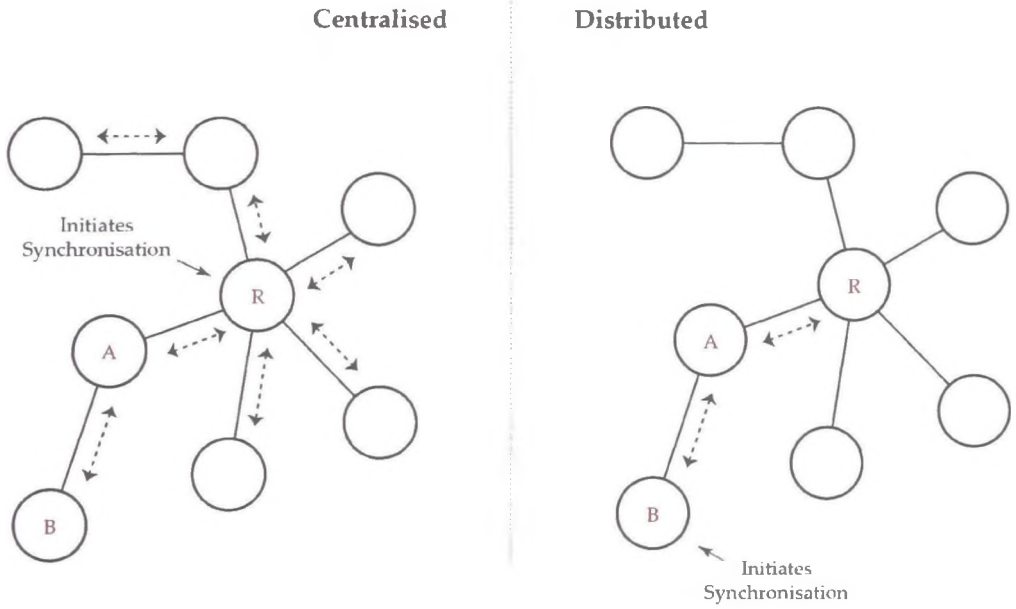


Figure 3.6: LTS operation

In the centralised scheme the network is constructed into a synchronisation spanning tree within which the root/reference node initiates synchronisation rounds. Nodes at consecutive hops are synchronised in turn until the entire spanning tree is synchronised. Since synchronised nodes initiate a message exchange, three messages as opposed to two are required, the third message being used to communicate the calculated offset back to the un-synchronised node. This is illustrated in fig. 3.7.

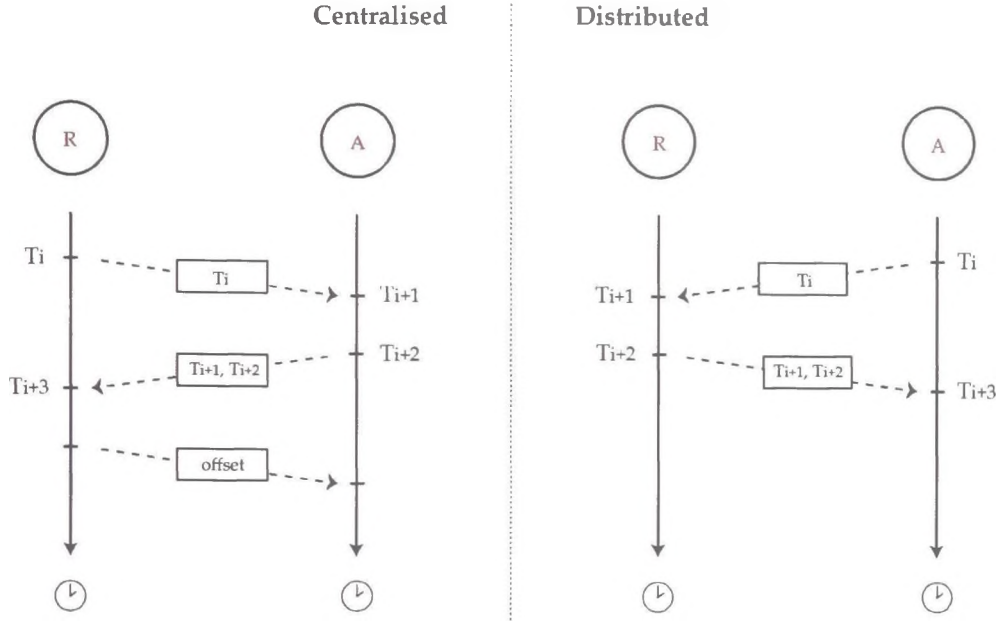


Figure 3.7: LTS operation

In view of the claim made by van Greunen and Rabaey [33] that the variance of the synchronisation error increases along each branch of the tree as a linear function of the number of hops, LTS promotes the construction of a minimum depth spanning tree before each synchronisation round in order to maximise accuracy throughout the network. In relation to time error that stems from clock skew, LTS alleviates such error by simply re-synchronising nodes at an appropriate rate which is based on knowledge of the worst case clock skew, the depth of the tree, and the WSN accuracy requirement. It should be noted that the worst case clock skew of each node is obtained from a specification sheet and programmed into each node before the network is deployed. The reference node, which initiates synchronisation rounds, must subsequently acquire all of this information and calculate a suitable periodic/static re-synchronisation rate.

The distributed scheme, in contrast to the centralised scheme, moves the responsibility of synchronisation from the reference node to the individual nodes themselves. A node determines when it requires synchronisation by using the same parameters that the reference node uses in the centralised scheme. To

accomplish synchronisation, a node sends a synchronisation request to its parent which in turn must synchronise itself. Thus, before a particular node can be synchronised, all nodes on the path from that node to the reference node must first be synchronised. An advantage of the distributed scheme is that not all nodes within the network must be synchronised when one particular node requires synchronisation. Thus, in certain scenarios, communication overhead can be significantly reduced.

3.3.5 Flooding Time Synchronisation Protocol

The *Flooding Time Synchronisation Protocol (FTSP)*, by Maróti et al. [10], organises a network into an ad-hoc synchronisation tree within which an elected root acts as the source of time. The synchronisation of nodes is performed using the uni-directional synchronisation technique detailed in section 2.4.2 and, thus, has senders distribute time to receivers.

The initial phase of FTSP involves the election of a root node which acts as the source of time for the network. The root election process is based on a simple algorithm whereby the node with the lowest assigned ID is elected the root. The election of the root is followed by synchronisation rounds which are initiated by the root and occur at periodic intervals denoted by τ. The choice of τ is dictated by the accuracy requirements of a WSN application and the state of a WSN's operating environment. If an environment subjects a node's clock to drift (changing skew), then lower values of τ result in more accurate estimates for that node's offset. This is true because the node receives time messages more frequently, and this allows it to update its estimate of its clock skew more regularly.

As illustrated in fig. 3.8, synchronisation messages contain 4 key fields: the *timestamp* field, the *rootID* field, the *nodeID* field, and the *seqNum* field. The *timestamp* field represents the sender's notion of time at the point of transmission; the *rootID* field represents the address of the root as recognised by the sender; the *nodeID* field represents the address of the sender and the *seqNum* field holds a sequence number that is incremented solely by the root at the beginning of each synchronisation round.

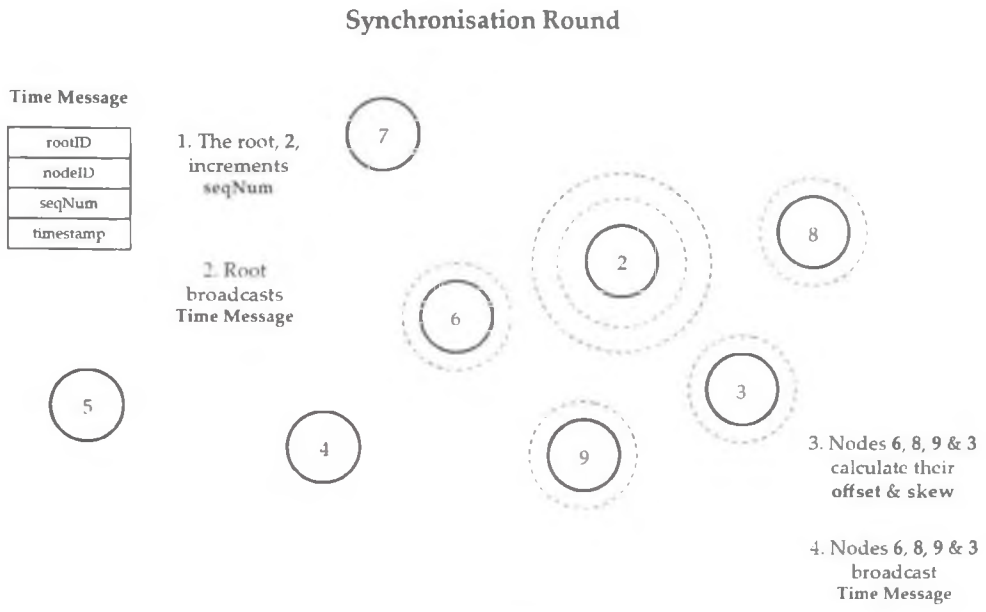


Figure 3.8: FTSP operation

A synchronisation round begins when the root broadcasts a synchronisation message. This message is received by all nodes within direct communication range of the root. The recipient nodes estimate their offset and skew and use these estimates to adjust their time-scale. They subsequently broadcast their own messages and place adjusted timestamps within the *timestamp* field. Thus, the root's time-scale is effectively flooded through the network. In order to manage redundant messages and ensure only the most recent messages are utilised by nodes, FTSP dictates that a node can only utilise the contents of a time message if it contains a lower *rootID* value or higher *seqNum* value than those received in the prior message. This mechanism, in addition to managing redundant messages, can also improve the synchronisation accuracy of those nodes located further away from the root. This is true because messages that arrive sooner encounter less delay en route through intermediate nodes and, therefore, more likely produce less error due to non-deterministic message latencies.

FTSP employs uni-directional synchronisation and, thus, is inherently susceptible to time errors that result from the propagation time of messages. This,

however, is shadowed by other larger delays that result from the MAC layer time stamping process. Traditional FTSP employs MAC layer time-stamping, which, although an improvement on application layer time-stamping, is still vulnerable to errors associated with non-deterministic message delays. To understand the source of these errors, the authors of FTSP analyse the tasks involved in communicating a message in detail. This analysis focuses on the transfer of an specific point in a message through the software and hardware components of a wireless sensor node. The assumption is made that the message is transferred to the radio one piece (one or more bytes) at a time whereby the radio requests each piece from the micro-controller via an interrupt request. The following delays are identified:

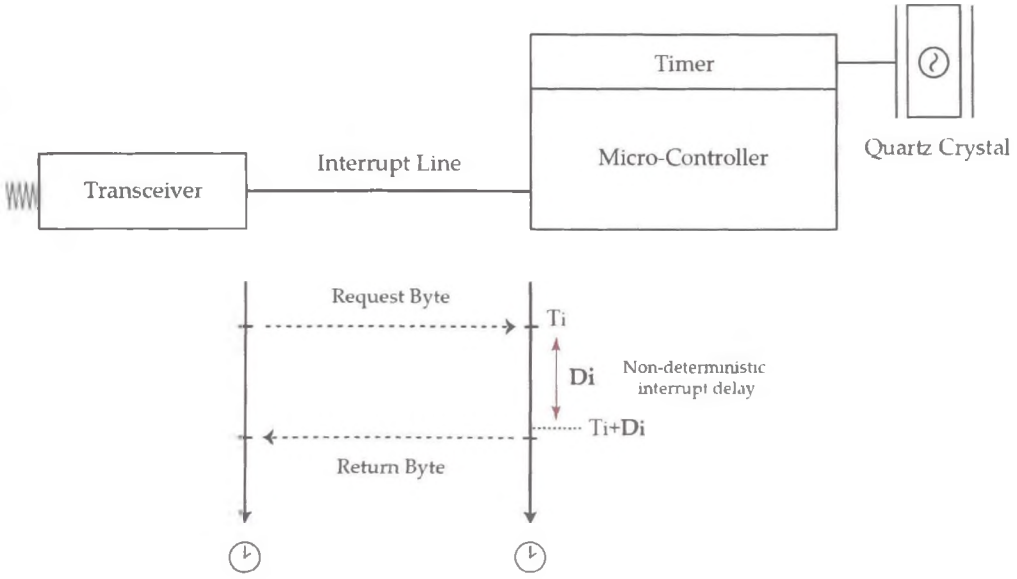
- *Interrupt handling time* - The time interval between the instant a transceiver raises an interrupt and the instant a microcontroller handles the interrupt by recording the time. This is non-deterministic but may be completely eliminated using capture registers, as declared by Maróti et al. [10].

- *Encoding time* - The time interval between the instant a transceiver raises an interrupt indicating the reception of a piece of data from a microcontroller and the instant the data is encoded into electromagnetic waves. This time is deterministic.

- *Decoding time* - The time required to decode the electromagnetic waves into a piece of binary data and then raise an interrupt to notify a microcontroller. This is mostly deterministic, but signal fluctuations and bit synchronisation errors can introduce jitter, that is, non-deterministic delays that are incurred when a receiver attempts to synchronise with a transmitter using the message preamble.

- *Byte alignment time* - This is a deterministic delay caused by differences in the byte alignment between a sender and receiver. This is applicable to receivers that cannot capture the byte alignment of a message and, therefore, must rely on the radio stack to determine the bit offset of the message using the synchronisation bytes located in the SYNC header of the message. The message must then be shifted accordingly.

This decomposition allows one to identify the greatest sources of error. These are identified by Maróti et al. [10] as the interrupt handling time and the encoding time. The FTSP method of reducing the errors associated with these processes entails the use of timestamps recorded at each byte boundary of a message. The assumption is made that a message is handed to a transceiver in a byte orientated fashion. The transceiver signals that it is ready to receive subsequent bytes via interrupt requests. Each interrupt request is handled by passing the next byte of the message to the transceiver and generating a timestamp. An FTSP sender records these timestamps and normalises them by taking an appropriate multiple of the nominal byte transmission time from each one, that is, the time it takes to transmit all bytes up to and including the byte associated with the timestamp. Thus, the transmission time is accounted for.



Ti	T1	T2	T3	T4	...
Di	10	23	5	8	

Figure 3.9: Interrupt delay

Interrupt delays vary and depend on the code being processed by the micro-controller when an interrupt is generated. Some code sections disable interrupts

which increases this delay. FTSP deals with interrupt delays by using the minimum of the recorded timestamps. The minimum timestamp provides a basis to deduce the interrupt delay associated with all other timestamps, thus, allowing them to be corrected. Consequently, those errors associated with the interrupt handling time are eliminated with high probability. Fig. 3.9 illustrates this mechanism. In fig. 3.9, the transceiver requests a byte from the microcontroller via an interrupt request at time T_i. The request is processed after a delay of D_i after which a timestamp is generated and a byte is returned to the transceiver. The value of D_i depends on the characteristics of the code being processed by the microcontroller at that time and, thus, varies for each request. In the table in fig. 3.9, the timestamps for each byte associated with a four byte message are presented. Here the request for the third byte results in the lowest interrupt delay of 5 ticks and, thus, produces the most accurate timestamp, T_3. Naturally, the identification of the minimum timestamp is performed after all timestamps have been normalised, since the value of D_i is unknown.

On the receiver side, any error associated with the byte-alignment time is corrected for. Since this delay is deterministic, it is calculated directly using the transmission time and the bit offset.

Ultimately, FTSP mitigates most of the errors associated with message delays with the exception of propagation delay, since it uses the uni-directional approach described in section 2.4.2. However, in this scenario, propagation time contributes to a very small proportion of error (less than $1\mu s$ for up to 300 meters). In contrast, the interrupt handling time can be as high as $30\mu s$.

In order to deal with clock skew, FTSP employs linear regression. Each node contains a regression table that holds the reference points related to the last N valid messages received. A node's skew value is updated each time a new, valid message is received. Subsequently, the oldest reference point contained in the regression table is shifted out and the new one, associated with the new message, is shifted in. The protocol also dictates that a node may only broadcast synchronisation messages when it has at least M entries in its regression table. This rule ensures synchronisation stability throughout the network.

3.3.6 Comparison

The protocols detailed in the previous sections all have a common goal, to time synchronise the resource limited computing systems found in WSNs. Each protocol, however, focuses more on one aspect of the task than another. A summary of the characteristics of each of the aforementioned protocols is presented in table 3.1. While protocols like RBS focus more on achieving the highest precision between nodes, others like TS/MS sacrifice maximum precision for low communication and computation overhead. TPSN, FTSP and LTS concentrate on multi-hop synchronisation more so than RBS and TS/MS which merely suggest multi-hop extensions. Applicability to commodity hardware and systems software also varies. FTSP and TPSN utilise MAC layer time-stamping and, therefore, access lower layers in the communication hierarchy. This can limit their use to specific platforms. RBS, LTS and TS/MS do not cross layers and are, therefore, completely platform independent. The technique used to synchronise a pair of nodes also varies. RBS, naturally, uses reference broadcast synchronisation, whereas FTSP uses uni-directional synchronisation, and the remainder use round-trip synchronisation.

Given the diversity of these time synchronisation protocols it is very difficult to rank them out of context. One must first define the network that they will be deployed on, the WSN accuracy requirements, the scale of the WSN, the capabilities of each node, and the dynamics of the environment. Suffice to say that each protocol is suited to a particular scenario. Having said this, FTSP has become a popular choice and is currently provided as the default synchronisation protocol in the *TinyOS* distribution.

Referring to table 3.1, it is true that FTSP makes assumptions about the nature of the communication channel and requires MAC layer accessibility, yet given the nature of WSNs these are not unreasonable assumptions. An important factor that separates FTSP from its rivals is its simplicity in terms of operation. Its simple root election policy allows it to dynamically re-configure its synchronisation topology in response to node failures. This makes it a more robust protocol in comparison to its rivals. This robustness proves favourable in WSNs where nodes may be inaccessible or expensive to maintain. In addition to this, FTSP

	RBS	TPSN	TS/MS	LTS	FTSP
Synchronisation Technique					
Reference Broadcast	X				
Uni-directional					X
Bi-directional		X	X	X	
Determined Clock Attributes					
Offset	X	X	X	X	X
Skew	X		X		X
Multi-hop Synchronisation					
All Nodes		X		X	X
Sub-set of Nodes	X		X		
Assumptions					
Broadcast Network	X				X
MAC Access		X			X

Table 3.1: WSN Time Synchronisation Protocol Comparison

employs uni-directional synchronisation and broadcasts messages. This drastically reduces the communication overhead required of other techniques. This does, however, result in minor errors, but these errors prove negligible in terms of the accuracy required by many WSN applications. Finally, since FTSP is provided as the default time synchronisation protocol in the *TinyOS* distribution, it seems logical to assume it has been tried and tested more frequently in the research community and, therefore, seems the more logical choice when faced with alternative options.

3.3.7 Parameter Configuration

In relation to the aforementioned synchronisation protocols, and given that they may be deployed in a variety of environments and employed by a variety of WSN applications, it is necessary before deployment to configure them appropriately.

Any time-sensitive WSN application has a specific accuracy requirement, and, generally, a WSN's nodes have energy constraints. This fact highlights an important configuration parameter, namely the *transmission interval*. Clearly a transmission interval that achieves the required accuracy efficiently is desired. If a protocol employs regression analysis to determine clock skew, then the number of data points used in the analysis affects its performance, thus, this highlights another important configuration parameter.

Choosing appropriate values for these parameters is not an trivial task, but Fujita et al. [34] propose a technique. It tackles the issue of determining synchronisation parameters that allow a time protocol to attain a particular accuracy in a particular environment most efficiently.

They first define a generic time protocol model that generalises a class of time synchronisation protocols, namely RBS, FTSP, and TPSN. This proposed model's operation consists of two phases. The first of these is the *Reference Point Creation* phase in which a node, A, that requires synchronisation produces the reference points (T_{Ai}, T_{Ri}) with the aid of a reference node, R. Since the model is a general one, the details of the process used to produce these reference points (i.e. uni-directional, round-trip, reference broadcast) is not specified. The *Reference Point Creation* phase highlights the first protocol parameter that must be determined. It is termed the *communication interval* (I_c) by Fujita et al. [34] and represents the time interval between the creation of two reference points.

The second phase, referred to as the *Clock Relation Inference* phase, uses the set of reference points collect in the first phase to infer a relationship between node $A's$ and node $R's$ clocks. This relationship is assumed to be a linear one and is computed via linear regression. A clock relation model is calculated from the reference points contained within a time interval termed the *synchronisation interval* (I_s). It is the continuous update of the clock relation model after each synchronisation interval that indirectly accounts for any clock drift. I_s, thus, denotes the second protocol parameter that must be determined.

The identification of suitable values for I_c and I_s in a particular operating environment is achieved by determining the total synchronisation error E_s for various values of I_c and I_s employed in that environment. The authors analyse the total error by identifying its composite elements, namely the *reference point*

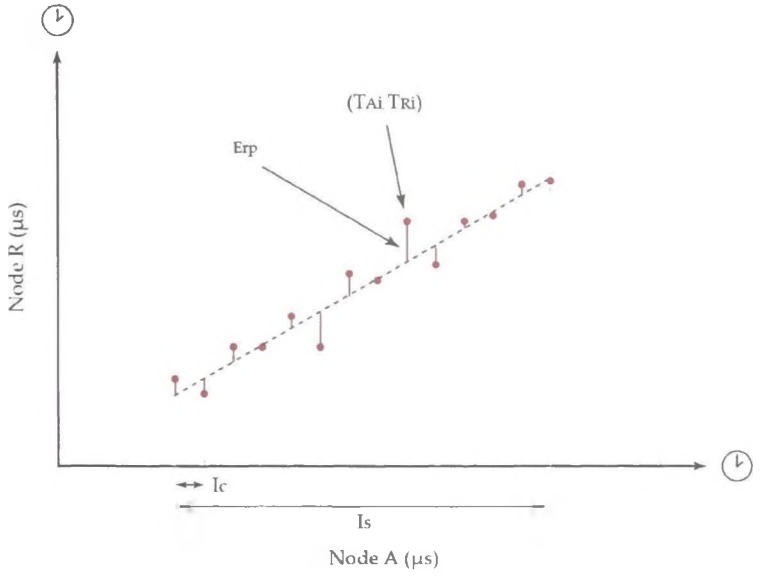


Figure 3.10: Protocol parameters

error E_{rp} and the *clock relation error* E_{cl}. Each of these errors is analysed individually. The *reference point error* represents the error that result from non-deterministic message delays during the *reference point creation* phase, that is, those delays described in section 2.4.1. The *clock relation error* represents the error that results from using an incorrect clock relation model, for example, a model that does not represent the true offset and/or skew of A's clock relative to R's clock. The addition of these errors produces the total error, $E_s = E_{rp} + E_{cl}$.

The verification of their method is performed using time data collected from a testbed over an appropriate interval. Plotting the standard deviation of the synchronisation errors against I_s for various values of I_c reveals optimal values for I_s and I_c that achieve specific levels of accuracy (see fig 3.11). Thus, the process of determining protocol parameters entails choosing a value for I_c that achieves the required accuracy and then choosing a value for I_s, below some upper bound $\overline{I_s}$, that achieves this accuracy most efficiently, that is, with the least amount of data and, thus, storage and computations.

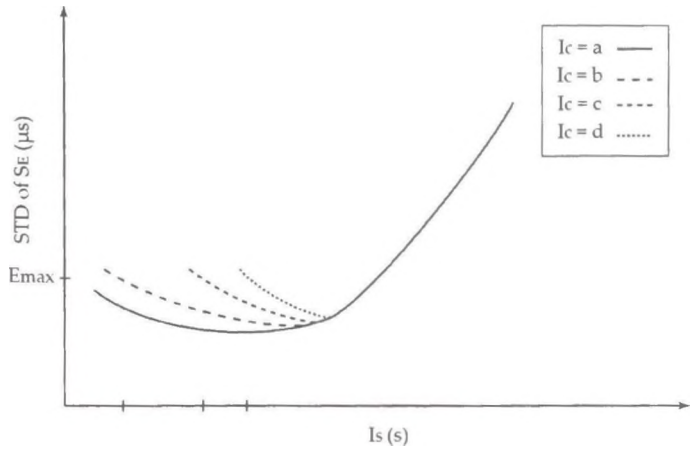


Figure 3.11: Synchronisation error

3.4 Dynamic WSN Time Synchronisation

Static synchronisation protocols are primarily concerned with a node's phase offset and clock skew. While determining the time offset between nodes found in traditional packet switched networks can be challenging, particularly when access to lower layers in the communication hierarchy is restricted, it is generally less of an issue in WSNs. The architecture of a WSN node is typically more coupled than that of a traditional computing system which means it is possible for higher layer software to interact directly with lower layer software. Thus, time stamping message reception and transmission events at lower layers in the communication hierarchy can be a simple task. In fact many modern microcontrollers employed in WSN nodes come equipped with capture registers that permit physical layer time stamping (see fig. 3.12). A capture register can be used to record the value of a timer when signalled to do so. In a typical case, a capture register is connected to the *start of frame delimiter (SFD)* interrupt line of a transceiver. The transceiver, once it has completed the transmission of the SFD field of a message, generates a signal that initiates the capture of the timer.

If a WSN's nodes possess capture registers, then errors associated with the send and access times of a message can be eliminated. The remaining sources of

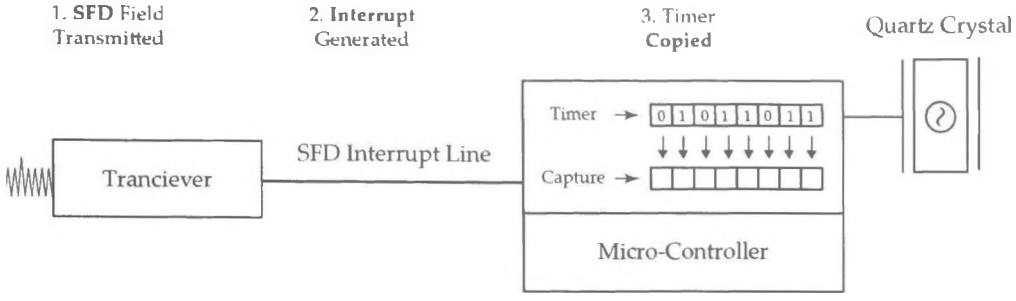


Figure 3.12: Capture register operation

error are delays associated with the propagation of the message and jitter at the receiver. These delays are generally quite small and, therefore, accuracies in the order of a microsecond can be achieved.

The next issue is clock skew. There are two methods that can be used to alleviate the effects of clock skew and, depending on the scenario, one will be favoured over the other. The simplest of these methods is periodic re-synchronisation. If the accuracy requirement of the WSN is not stringent, then such a method is justified since it will not result in significant communication overhead. The second method involves estimating clock skew using linear regression. This is preferred when the accuracy requirements are so harsh that the frequency of re-synchronisation would result in severe communication overhead. Thus, the extra computation and memory overhead required to employ linear regression is justified.

In either case, a suitable transmission interval must be chosen so that accuracy requirements are met and energy constraints adhered to. In a relatively stable environment within which variations in ambient conditions are negligible and, thus, have little effect on clock frequency, the transmission interval can be deduced using knowledge (if it exists) of the worst case relative clock skew between the nodes in the network. This, however, does not work well when the environment presents varying conditions. As outlined in section 2.4.3, temperature fluctuations can lead to clock drift which makes the task of determining an appropriate transmission interval more challenging.

In such cases, a suitable transmission interval might be determined via an

empirical analysis of the environment in combination with a technique like that suggested by Fujita et al. [34] (see section 3.3.7). However, this approach can be time consuming and may produce a sub-optimal result in highly dynamic environments. This issue promotes the employment of a mechanism that actively monitors the condition of nodes' clocks and alters their transmission interval in response to variations in clock frequency. The work in this thesis presents such a mechanism.

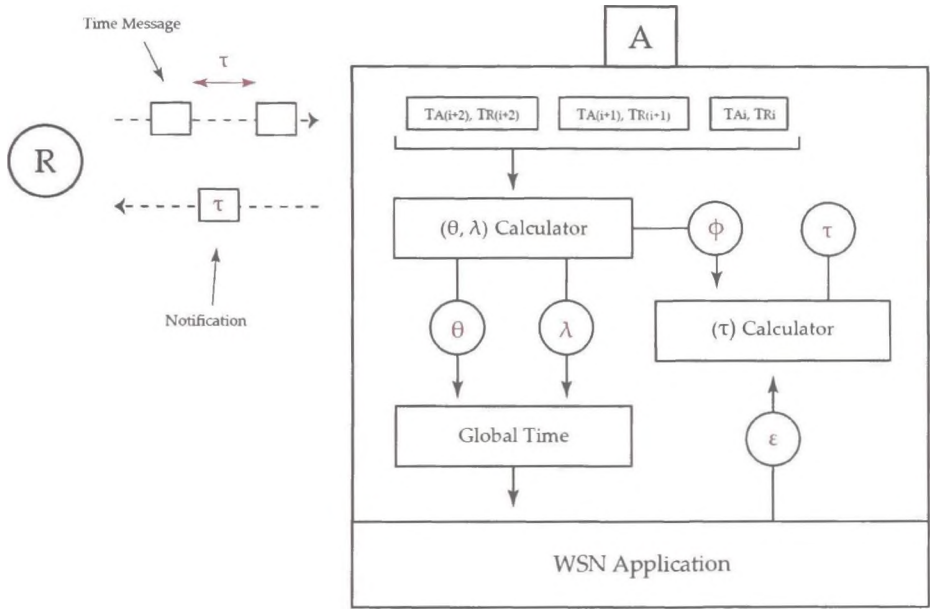


Figure 3.13: Dynamic synchronisation

Fig 3.13 illustrates how such a mechanism might work. In the example illustrated in fig 3.13, a WSN application's accuracy requirement is represented by an error bound, $[-\epsilon, \epsilon]$. The phase offset (θ) of every WSN node must be kept within this bound. A node, A, that synchronises with a reference node, R, does so by obtaining a reference point and uses this information to determine its phase offset. If A and R are capable of physical layer time stamping, then two reference points allow A to produce a reasonably accurate value for its clock skew (λ) relative to R. With three reference points A can estimate its clock drift (ϕ) relative to R. Knowledge of A's clock drift and the application error bound

can be used to estimate the future point in time that A's clock offset will exceed the error bound. To avoid exceeding the error bound, node A must receive a new time message from R before the aforementioned point in time so that it can update its estimate of its skew. Thus, the ideal time between message receptions, that is, the transmission interval (τ) of the reference node, is determined and the reference node is notified of this interval via a notification message.

This technique provides one approach that can be used to automatically determine an appropriate transmission interval in real-time. However, any approach that accomplishes the same goal is, in this text, classified as a *dynamic synchronisation protocol.*

3.5 Dynamic Time Synchronisation Protocols

The previous section detailed the fundamental concept of a dynamic time synchronisation protocol. The following subsections detail the design and operation of two WSN time synchronisation protocols that can be classified as dynamic.

3.5.1 Rate Adaptive Time Synchronisation

Rate Adaptive Time Synchronisation (RATS), by Ganeriwal et al. [14], uses a feedback control loop to determine an optimal transmission interval (S) to keep two nodes synchronised within an application defined error bound (E_{max}) (see fig. 3.14). A sample repository stores a past window of samples (T_{Ai}, T_{Ri}) corresponding to the transmission time and reception time of a message from a reference node, R, to a host node, A. From this repository of samples, a linear clock model is formulated and used to estimate the time at node A, denoted T_{Ae}. This time estimate is used to produce a prediction error/offset estimate (δ) by means of a confidence interval.

The employed confidence intervals imply that the errors produced by the linear clock model have a normal distribution. This of course is an incorrect assumption since physical phenomena, such as heat, influence the behaviour of a clock. To account for this bias, the confidence bands are altered using a scaling factor (Δ). The scaling factor is used to produce the final prediction error ($E_p = \Delta\delta$).

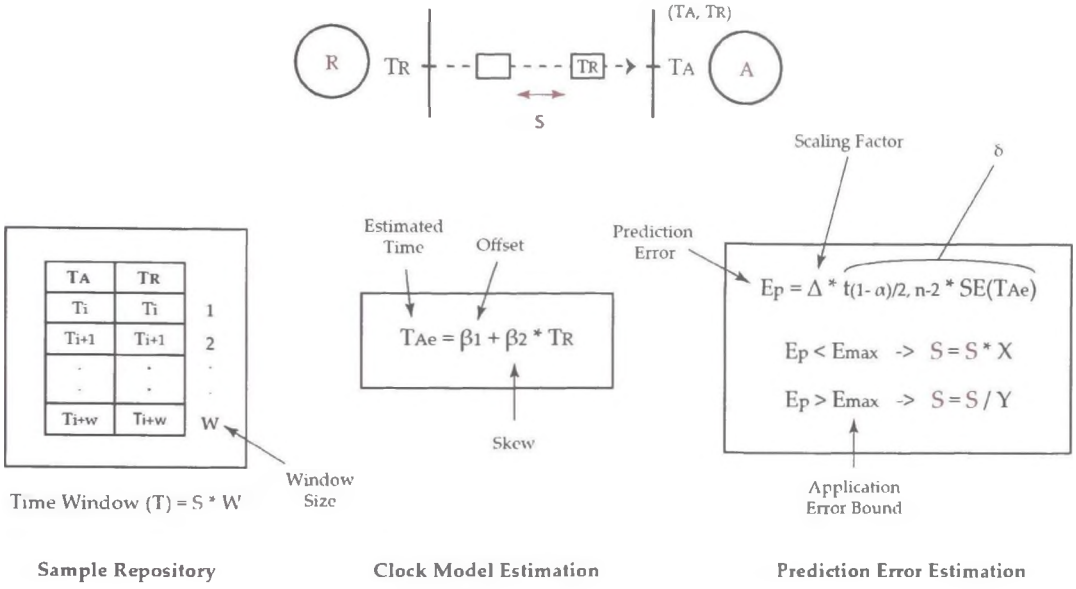


Figure 3.14: RATS operation

In order to determine the final transmission interval (S), the prediction error (E_p) is compared to the application error bound (E_{max}), and the transmission interval is multiplicatively decreased if the prediction error is greater and multiplicatively increased if the prediction error is smaller.

An important feature of RATS is its estimation of an optimal repository window size (W) for a given transmission interval (S), similar to that detailed in section 3.3.7. Through empirical analysis of data collected from sensors in different environments, it is observed that for a particular environment and sampling period there exists an optimal window size (W) that reduces the prediction error (E_p) (see fig 3.15). It is also observed that the *Time Window* (T), which is equal to the product of a given transmission interval (S) and its optimal window size (W), remains relatively constant for different transmission intervals and depends only on the environment. Of course because the clock is modelled using a linear relationship, at least 2 data points are required, thus, S can never be greater

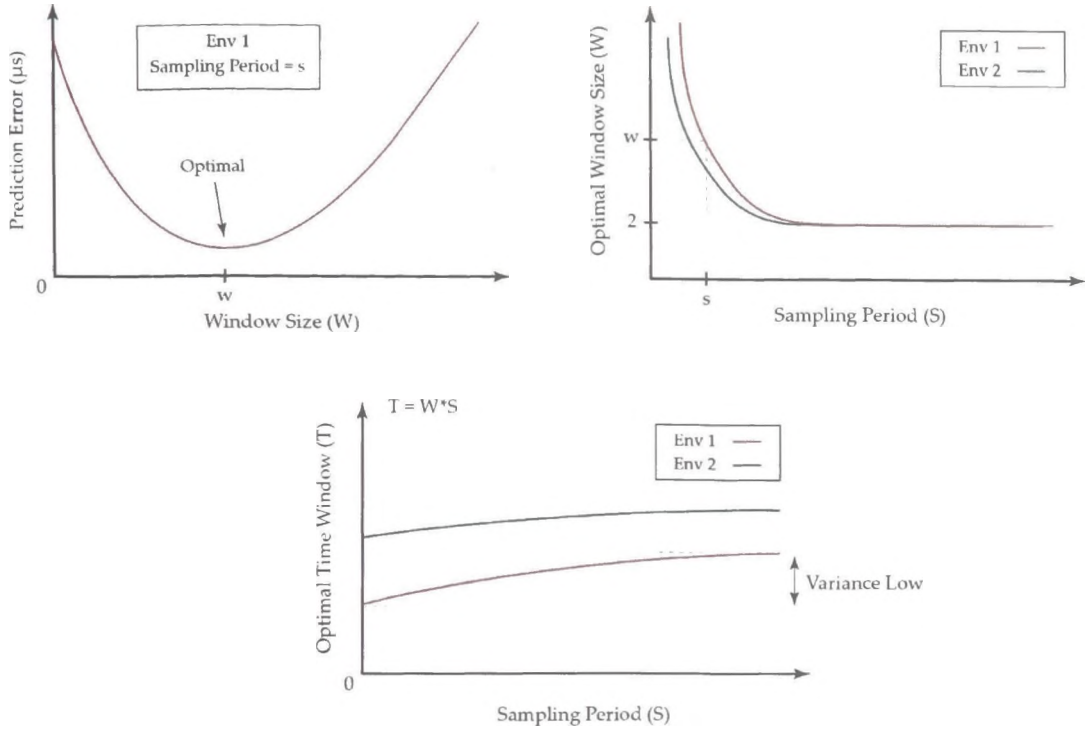


Figure 3.15: Window size, sampling period & time window

than $W/2$. All of these observations highlight an important learning parameter in RATS, namely the Time Window (T).

Thus, in order for RATS to determine an optimal transmission interval it must learn two key parameters, namely the scaling factor (Δ) and the time window (T), and these parameters are dependent on the environment. These parameters can be learned automatically by nodes and experiments indicate an average learning time of between 2 and 4 hours after which an optimal transmission interval is determined.

Ganeriwal et al. [14] confirm via various experiments that RATS outperforms static synchronisation techniques in terms of energy conservation and error reduction. It successfully determines a relatively optimal transmission interval in order to keep a node's time error within a specified application error bound (E_{max}). It, however, performs better in stable environments. The formulation of the RATS

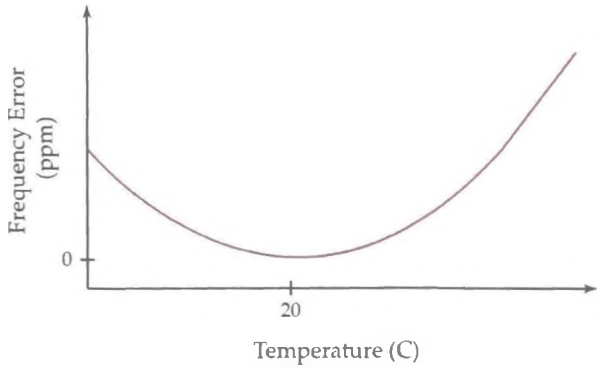
algorithm is based on datasets collected from environments with temperature variances no greater than $10°C$. The dataset from the environment with the greatest temperature change is the one that produces the most variable time window across a range of sampling periods. Given the behaviour of quartz crystals in response to temperature changes, this result seems somewhat expected and suggests that since RATS is quite dependent on the time window (T), it is more suited to relatively stable environments with gradual temperature changes. Of course, this is not a major disadvantage but rather suggests that it is not applicable to all scenarios.

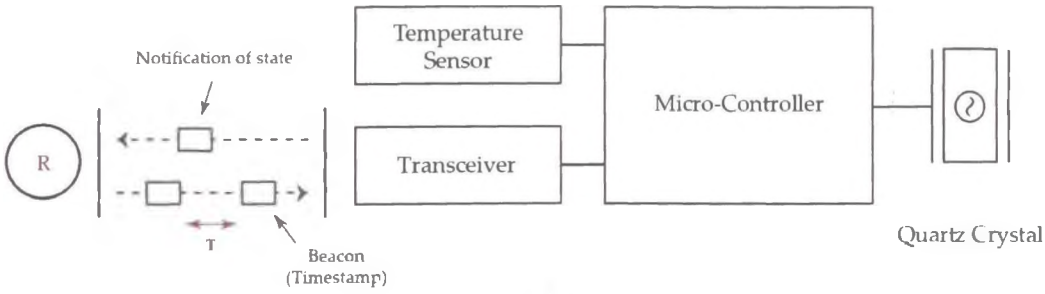
In response to this poorer performance in unstable environments, Ganeriwal et al. [14] investigate the effect of changing the clock model from a linear one to a quadratic one. The results, however, suggest no extra benefit to justify the additional complexity. This highlights the difficulties in modelling a clock.

3.5.2 Temperature Compensated Time Synchronisation

Temperature Compensated Time Synchronisation (TCTS), by Schmid et al. [15], focuses on the major source of clock drift, namely temperature. It characterises a node's clock based on its ambient temperature and corresponding frequency error. This is performed with the aid of a temperature sensor together with time measurements obtained from a reference node. Each temperature measurement is mapped to a frequency error measurement and placed in a calibration table which is subsequently used as a reference. This, in essence, allows a node to compensate for clock drift without the aid of further time messages, except for the occasional message to accommodate inaccurate measurements in the process.

The process, which is illustrated in fig 3.16, consists of two phases: the *calibration phase* and the *compensation phase*. In the *calibration phase* the node receives time-stamped beacon messages from a reference node at an interval termed the *calibration interval*. The *calibration interval* is an optimal interval that allows a node to produce the best estimate of its frequency error. The node determines this interval using knowledge of its nominal clock frequency together with information about the expected temperature environment. It then notifies its reference node of this interval.





Calibration Table

Freq. Error	Temperature
2	25
11	30
.	.
.	.
.	.

Calibration Phase

Determine frequency error at various temperatures
&
fill calibration table

T = Calibration interval

Compensation Phase

Measure temperature at regular intervals
&
use calibration table entries to correct for frequency error

T = Compensation interval

Figure 3.16: TCTS operation

At the reception event of each beacon message, the node takes a temperature reading, determines its frequency error and adds the entry to its calibration table. If the temperature does not change then the node notifies the reference node that it is calibrated and the reference node increases the time interval between beacon messages. The node then enters the *compensation phase* whereby it takes regular temperature readings and uses the calibration table to correct for the frequency error. If at any stage the temperature of the node changes and an entry is not found in the calibration table then the node switches back to the calibration state to acquire the required information.

A noteworthy strength of TCTS is its ability to keep its clock synchronised in scenarios where communication between nodes is lost. In addition to this, it can be used in conjunction with almost any legacy time synchronisation protocol. Conversely, it requires that nodes possess temperature sensors which limits its use to particular hardware implementations.

3.6 Summary

This chapter focused on time synchronisation in the context of Wireless Sensor Networks. Time synchronisation protocols play an important role in WSNs, since many WSN applications necessitate time-stamped data for data fusion purposes. Furthermore, lower layer applications, such as certain MAC protocols, require synchronisation for scheduling. Given its importance, there exist a range of time synchronisation protocols that attempt to fulfil this role. These protocols can be classified into two categories, namely static synchronisation and dynamic synchronisation protocols. This classification is based on whether a protocol has the capability to alter its communication parameters in response to changing clock characteristics. A static synchronisation protocol does not have this capability and employs a fixed transmission interval. A dynamic one continuously monitors this interval and adjusts it, selecting suitable values that achieve the required time accuracy with a minimum number of message transmissions.

The inefficiency of static time protocols stems from their use of a fixed transmission interval. This parameter specifies the time interval between subsequent message transmissions and its value is chosen based on the accuracy requirements

of the application. The need for continuous message exchanges between synchronising nodes is due to clock drift. While long term clock drift is generally the result of aging, short term clock drift is predominantly caused by temperature changes. An environment that does not subject WSN nodes to significant temperature changes, termed a stable environment in this text, does not cause significant clock drift. Thus, a synchronisation protocol operating in a stable environment can significantly reduce the number of message transmissions that are required to provide a similar level of accuracy in an unstable environment.

The value of the transmission interval employed by a static time protocol is predetermined before network deployment. Ideally, this transmission interval is selected after analysing the operating environment. The chosen interval will most likely be a value that achieves the required accuracy in a particular environment. After deployment, any temperature change in the environment that was not considered before deployment may lead to clock drift and, consequently, degrade the performance of the protocol. Conversely, if an interval is chosen such that a protocol can deal with rare temperature excursions, then for the majority of its lifetime, the protocol will be operating inefficiently.

Dynamic synchronisation protocols attempt to remedy this issue. The protocols detailed in section 3.5 try to maximise the transmission interval such that a particular level of synchronisation is achieved efficiently. TCTS does this by focusing on the dominant source of clock drift, that is, temperature. A node's temperature sensor is used in conjunction with time messages from a reference node in order to characterise a node's oscillator. The result is a collaboration table that maps an oscillator's frequency error to its ambient temperature. This is very powerful since the behaviour of the oscillator in response to temperature changes can be determined and the resulting time errors corrected for. In theory, a node in possession of a highly accurate calibration table that covers the full range of ambient temperatures could eliminate the communication overhead associated with the synchronisation process. TCTS, however, is not without its drawbacks. Its foremost shortcoming is its hardware requirements. Temperature sensors, although common, are not ubiquitous and this limits TCTS to specific settings. Another issue, although a minor one in many scenarios, is that TCTS focuses only on one aspect of the environment in order to characterise the oscillator.

Other features, such as pressure and voltage, are neglected.

RATS takes an alternative approach and relies solely on time messages in order to determine an ideal transmission interval for an operating environment. The RATS design is based on the observation that when using regression analysis to determine the skew of a clock, the data utilised must originate from a particular time interval or *Time Window*. This Time Window is related to both the transmission interval and the characteristics of the operating environment, and, thus, it must be learned before RATS can operate efficiently. This particular trait, the learning phase, highlights RATS's limitations in terms of its responsiveness to unforeseen environmental changes and suggests it is more suited to relatively stable environments with minor and gradual temperature changes.

The limitations of both TCTS and RATS in certain scenarios highlight a void to be filled. Thus, there is room for an alternative dynamic time protocol that does not have the aforementioned limitations and, thus, requires no additional hardware and can react relatively quickly in dynamic environments. One such scenario that necessitates an alternative solution is a WSN that consists of nodes with limited hardware that must operate in an environment that subjects nodes to large, rapid temperature or pressure changes. The protocol must achieve what RATS and TCTS achieve in their ideal environments, namely efficient time synchronisation. In particular it should perform the following:

- Reduce energy consumption
- Attain a pre-configured application-specific accuracy
- Eliminate a pre-deployment analysis of environment
- React rapidly in a changing environment
- Operate without any auxiliary hardware

A logical step towards this goal is to take the de facto static time synchronisation protocol, a protocol which has been tried and tested successfully in multiple WSN deployments, and extend it such that it operates dynamically. A comparison of the various static synchronisation protocols employed in WSNs is presented in section 3.3.6 and a strong case for FTSP is made. This conclusion is based

on the protocol's high accuracy capabilities, low communication overhead, and simple, efficient operation relative to other protocols. It is for this reason that this work presents and extension for the FTSP protocol. The extension produces a dynamic variant of FTSP termed the *Dynamic Flooding Time Synchronisation Protocol (D-FTSP)*. The next chapter presents the details of D-FTSP and verifies its effectiveness via appropriate experimentation.

Chapter 4

Dynamic Flooding Time Synchronisation Protocol

The *Dynamic Flooding Time Synchronisation Protocol (D-FTSP)*, by Shannon et al. [35], addresses a gap highlighted by the detailed analysis of other WSN time synchronisation protocols. It is designed for very dynamic and unpredictable environments that subject WSN nodes to rapid and substantial clock drift. Such extreme environments are not uncommon.

In relation to natural environments and according to Ahrens [36], the largest daily range of temperature occurs in deserts where day and night temperatures can vary by over 30 °C. Due to the lack of water vapour in the air in such regions and the lack of clouds, the sun's rays can rapidly heat surface air to as much as 40 °C. At night this heat energy is quickly radiated into space and temperatures can drop to 7 °C. Such extreme temperature variation, however, is not limited to desert regions. In other more habitable regions of the Earth, it is not unusual to observe an air temperature difference of 15 °C between air in contact with the ground and air a fraction of a meter above it [36]. This phenomenon occurs on calm sunny mornings when the rising sun heats the cool surface of the Earth. As the surface heats up, the air in contact with it is also heated by conduction, yet since air is a comparatively bad conductor of heat the temperature difference between surface air and air a number of centimetres above it can vary as much as 15 °C or more. With respect to an environmental based WSN operating in such

environments, temperature differences between nodes can vary likewise. Such temperature differences can cause significant variations in a node's clock skew throughout the day.

Artificial or indoor environments can present similar variations in temperature, and such changes can occur more rapidly. One example of an indoor environment that can display such characteristics is a Data Centre. ASHRAE *(American Society of Heating, Refrigerating and Air Conditioning Engineers)* recommends that critical data centres maintain a temperature in the range of 18 °C – 27 °C and also specifies an allowable range of 15 °C – 32 °C which translates to a 17 °C temperature variation [37]. For less critical data centres, an allowable range of 10 °C – 35 °C is specified. This is a substantial variation in temperature. If one considers a WSN application deployed in a data centre and assigned the task of monitoring the performance of a cooling system, due to the temperature dynamics in such an environment, WSN nodes might be subjected to rapid and localised temperature changes. The difference in the ambient temperature between subsets of nodes could vary as much as 20 °C or more, possibly as a result of a cold/hot aisle configuration with frequent VM (Virtual Machine) migrations. The resulting effect on nodes in terms of clock drift would be substantial and problematic if not alleviated.

As outlined in the previous chapter, D-FTSP is an extension of the *Flooding Time Synchronisation Protocol (FTSP)* described in section 3.3.5. FTSP is a static synchronisation protocol and, therefore, must be configured to achieve a specific synchronisation accuracy in a particular environment. The dynamic extension eliminates the need for configuration by directly monitoring the clock drift of nodes. This information is used to determine a suitable transmission interval that ensures accuracy requirements are attained and maintained efficiently.

Static synchronisation protocols that do not determine the clock skew of nodes must rely on re-synchronisation which can be extremely inefficient from an energy/bandwidth perspective and in some cases impractical. The majority of the protocols described in chapter 3 determine clock skew, thus, enabling them to significantly reduce their communication overhead. None of these protocols, however, determine the clock drift of nodes. Their operation is based on the assumption that the skew of a node remains constant over short intervals and, thus,

the continual re-calculation of a node's clock skew indirectly accounts for clock drift. This is true, but the interval over which skew remains relatively constant is dependent on the environment and so must be predetermined. This raises issues for WSNs that must be deployed in more dynamic environments.

Dynamic synchronisation protocols, such as RATS, go one step further and determine this interval by means of a feedback control loop. In order to do this, the protocol must determine a number of parameters, the values of which are dependent on the operating environment (see section 3.5). This requires a learning phase, thus, affecting its responsiveness to unforeseen environmental changes. TCTS, in contrast, does not determine an ideal transmission interval but instead employs a temperature sensor in order to determine the frequency error of a clock at various temperatures. Thus, TCTS requires additional hardware and it ignores other factors that might influence an oscillator, such as pressure and voltage.

In either case, the actual value of a node's clock drift is ignored. It is this feature that distinguishes D-FTSP from its counterparts. D-FTSP directly measures the drift of nodes and from this measurement determines the future point in time that a node should receive a time message in order to update its skew and avoid accumulating unacceptable time error. This approach offers particular advantages that make it more suited to volatile environments. By directly determining the drift of nodes, D-FTSP responds more rapidly to oscillator frequency changes. In addition, it does not focus on just one particular aspect of the environment but multiple (pressure, voltage etc.) and does so with no additional hardware. The result is efficient and accurate time synchronisation in dynamic environments.

4.1 Design and Operation

Before detailing the design and operation of D-FTSP, a brief overview of FTSP is necessary. At deployment time, FTSP uses a simple election policy to elect the root of the synchronisation hierarchy whereby the node with the lowest ID is chosen. After a root has been elected, it initialises a variable termed *seqNum* which is subsequently used to associate time messages with synchronisation rounds.

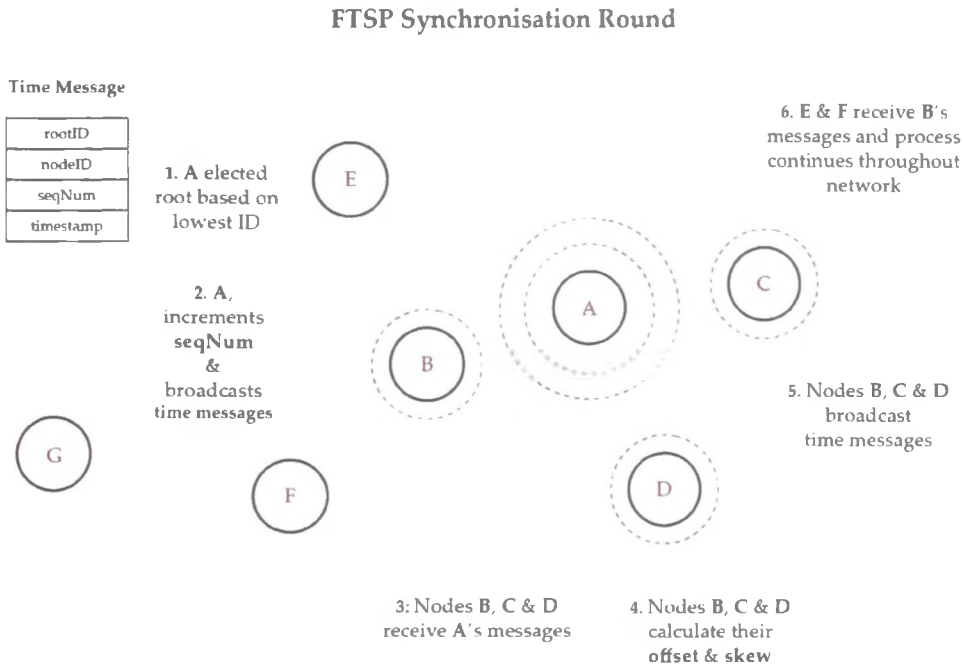


Figure 4.1: FTSP synchronisation round

The value of the *seqNum* variable is incremented by the root at the beginning of each synchronisation round. The steps involved in a synchronisation round are illustrated in fig. 4.1. In fig. 4.1, node A is elected the root and subsequently broadcasts time messages at a pre-configured interval termed the *transmission interval* (τ).

Fig. 4.1 also presents the format of an FTSP time message. It contains 4 fields: the *timestamp* field, the *rootID* field, the *nodeID* field, and the *seqNum* field. The *timestamp* field represents the sender's notion of time at the point of transmission; the *rootID* field represents the address of the root as recognised by the sender; the *nodeID* field represents the address of the sender and the *seqNum* field holds the value of the *seqNum* variable, as explained above.

Nodes B, C and D overhear A's broadcasts and use the received timestamps to estimate their offset and skew using linear regression. Nodes B, C and D subsequently broadcast their own time messages which contain timestamps that

have been modified using the aforementioned offset and skew estimations. Nodes E and F overhear messages broadcast by node B and use them to estimate their offset and skew. They too subsequently broadcast time messages. This process continues until the root's time-scale has been effectively *"flooded"* throughout the network.

The *seqNum* field plays an important role in FTSP. It allows nodes to distinguish between new messages and old, or repeated, messages. In a contention-based network, such as an 802.15.4 based WSN, nodes will compete with other nodes for the communication medium. If a node is busy transmitting, then competing nodes must wait. If message time stamping is performed at the application layer, as it is on particular sensor platforms, then this delay leads to time errors. Given this fact, it seems logical that recipients of time messages should only utilise the first message in a sequence of time messages with identical sequence numbers (*seqNum*), since this most likely contains a timestamp with the least amount of error. Thus, FTSP dictates that nodes can only utilise messages that contain a greater sequence number than the previously received message. For example, when nodes E and F broadcast time messages, node B will not utilise them because it has already encountered a message from A with an identical sequence number.

The above explanation summarises the core operation of FTSP. To extend FTSP to operate as D-FTSP does, the message structure must be modified such that a node can transmit more detailed information regarding the state of its clock, in particular, its clock skew. Furthermore, the FTSP protocol must be extended such that it can utilise this additional information in order to determine the clock drift of nodes.

Fig. 4.2 illustrates a D-FTSP synchronisation round and the modified message structure which contains two additional fields. The *hop* field holds the hop distance of a sender relative to its root and, as described below, enables recipients to determine if a sender is their child. The *skew* field holds the clock skew of a sender as estimated by the sender using messages received from a node at a lower hop, that is, messages received from a parent. It is the *skew* field that enables recipients to estimate the drift of neighbouring nodes, and it is the *hop* field that allows recipients to restrict this estimation to just its children, thus, reducing

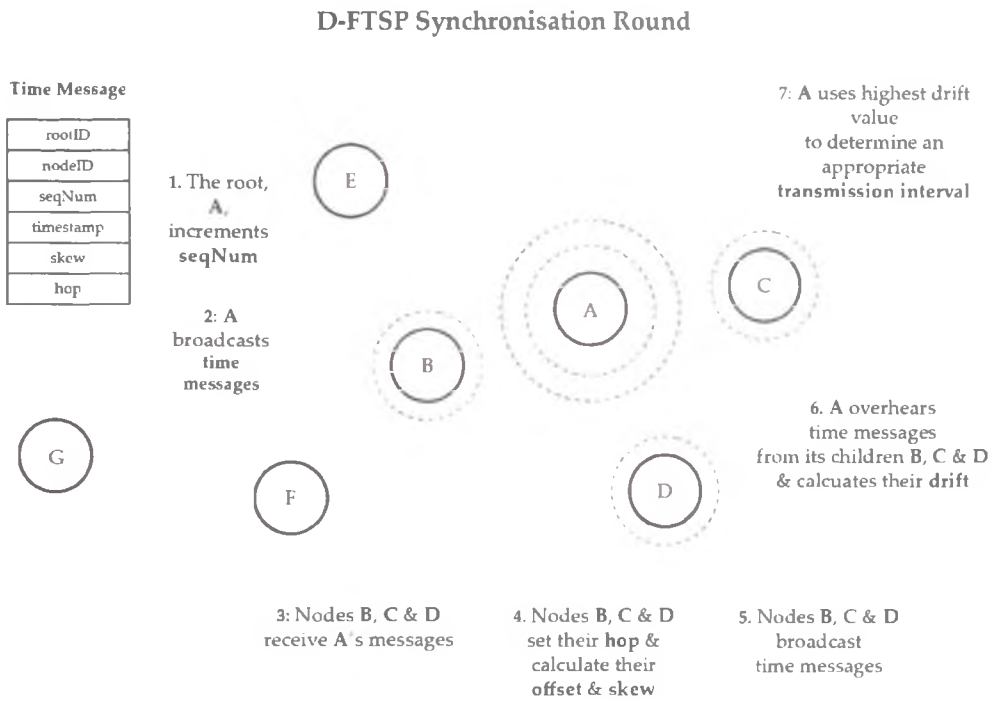


Figure 4.2: D-FTSP synchronisation round

processing and communication overhead.

Referring to fig. 4.2, a D-FTSP synchronisation round is similar to an FTSP synchronisation round in many respects. Thus, the elected root, A, broadcasts time messages which node's B, C and D use to estimate their clock offset and skew. However, nodes B, C and D also determine their hop distance from the root using the contents of the *hop* field. Nodes B, C and D subsequently broadcast time messages which contain both their estimated skew and their hop distance. Node A receives these broadcasts and determines from the value of the *hop* field that they are its children. A is now aware that it is directly responsible for ensuring B, C and D are synchronised within an error bound, $[-\epsilon_{avg}, \epsilon_{avg}]$, which is defined based on a WSN's application accuracy requirements. Thus, A estimates their individual clock drifts and uses the highest drift value together with the error bound to determine an appropriate transmission interval, that is, the interval between subsequent messages broadcast by A. This process continues

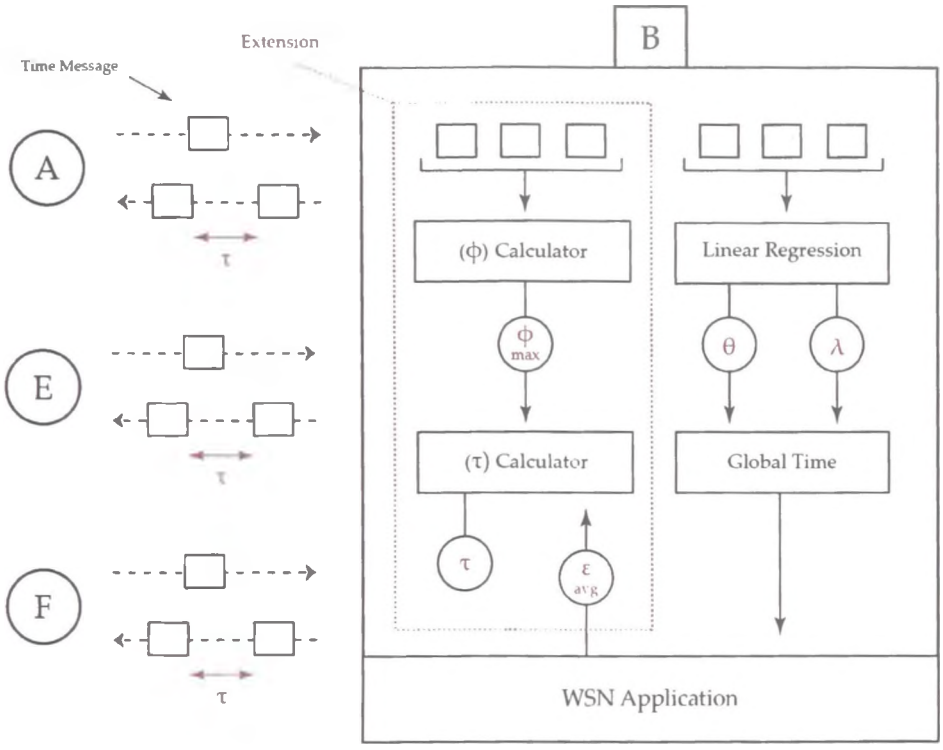


Figure 4.3: D-FTSP operation

throughout the network until all nodes are synchronised.

A more detailed analysis of D-FTSP's operation is illustrated in fig. 4.3. Here the focus is on node B which, from fig. 4.2, is a parent of nodes E and F and a child of node A. B initially receives broadcasted time messages from node A. It uses these time messages to create reference points from which it determines its offset (θ) and skew (λ) via linear regression. These parameters are subsequently used by a *Global Time* function to transform B's timescale into that of its reference/parent, A. This represents the core operation of FTSP.

The left section of fig. 4.3 illustrates how FTSP is extended to form D-FTSP. The extension enables node B to utilse time messages broadcasted by nodes E and F to calculate their clock drift (ϕ). The highest drift value, ϕ_{max}, and the application error bound, $[-\epsilon_{avg}, \epsilon_{avg}]$, are used to determine an appropriate transmission interval, τ, that ensures that the nodes receive a time message from

B before they accumulate an absolute error greater than ϵ_{avg}.

4.1.1 Transmission Interval

A node estimates the drift of each of its children and places these values in a table. It then uses the maximum drift, ϕ_{max}, and the application error bound to determine an appropriate transmission interval, τ. The application error bound, denoted ϵ_{avg}, is a preconfigured value representing the absolute average phase offset that a node should strive not to exceed. It should be noted that ϵ_{avg} is not a global error bound targeting the whole network but a local one, that is, within a single hop of a node. This is a limitation but is one that exists in FTSP and many other time synchronisation protocols that target multi-hop WSNs.

The value of the *transmission interval*, τ, is determined using equation 4.1.

$$\epsilon = \int_{t_i}^{t_i+\Delta} (\phi \cdot t)\, \mathrm{d}t \tag{4.1}$$

Δ represents the time interval over which a node that is subjected to a drift of magnitude ϕ will accumulate an error of magnitude ϵ. One can replace ϵ and ϕ with ϵ_{avg} and ϕ_{max} respectively. One must then determine Δ. Integration of equation 4.1 results in equation 4.3.

$$\epsilon_{avg} = \left[\phi_{max} \cdot \frac{t^2}{2}\right]_{t_i}^{t_i+\Delta} = \left(\phi_{max} \cdot \frac{(t_i+\Delta)^2}{2}\right) - \left(\phi_{max} \cdot \frac{t_i^2}{2}\right) \tag{4.2}$$

$$\epsilon_{avg} = \phi_{max} \cdot (t_i\Delta + \frac{\Delta^2}{2}) \tag{4.3}$$

Since one need only be concerned with relative time, the term t_i can be set to 0. The equation can then be reduced to that in equation 4.4.

$$\epsilon_{avg} = \phi_{max} \cdot \frac{\Delta^2}{2} \tag{4.4}$$

From this Δ can be derived using equation 4.5.

$$\Delta = \sqrt{\frac{2\epsilon_{avg}}{\phi_{max}}} \tag{4.5}$$

Since Δ represents the time interval before which a node's clock error exceeds ϵ_{avg}, the node must receive a new synchronisation message within this interval. Thus, one can let $\tau = \Delta$.

The calculation of square roots is a resource consuming process, particularly for low powered WSN motes which contain microcontrollers that do not typically possess a floating-point unit (FPU). Consequently, such calculations must be performed in software. To avoid this, the estimation of τ is performed using the formulated algorithm 1. In algorithm 1, τ is incremented by a constant value C. If the value of ϕ_{max} happens to be zero, then the operation completes (no drift detected). If not, then the error, ϵ, is calculated and compared to ϵ_{avg} in order to determine if the error exceeds the application error bound. If it doesn't, then the process repeats. An update of τ is performed after each transmission of a message or in the event that a higher value of ϕ is observed. This ensures nodes quickly determine an optimal transmission interval.

Algorithm 1 Calculation of τ

```
if φ_max ≠ 0 then
    ε ← 0
    τ ← τ_min
    while ε < ε_avg do
        τ ← τ + C
        ε ← φ_max × ((τ × τ)/2)
    end while
end if
```

4.2 Experimentation

As outlined in the preceding sections, D-FTSP is an extension of the FTSP protocol which transforms it from a static synchronisation protocol into a dynamic one. Thus, it is necessary to compare and contrast the performance of the two protocols in order to quantify the benefits of D-FTSP over FTSP, if any. The latter protocol requires that the parameter of interest, the *transmission interval*, is

manually configured, whereas the former automatically configures it. In order to provide an unbiased comparison, it is necessary to carefully design an appropriate experiment.

The first factor that must be considered is the operating environment. Since FTSP is a static synchronisation protocol, it must be configured with a suitable *transmission interval* so that it can operate more effectively within a stable environment, that is, an environment that does not subject nodes to clock drift values. Conversely, to best realise the benefits of D-FTSP, it must be deployed in an unstable environment, that is, an environment that does subject nodes to clock drift. Thus, to compare both protocols, it is necessary to test each in both a stable and unstable environment.

The second aspect that requires attention is the *transmission interval*. This parameter must be configured before deploying FTSP, whereas D-FTSP, using knowledge of one or more nodes' clock drift value(s), self-configures it during operation. For test purposes, with FTSP, this issue can be overcome by specifying a range, a maximum and minimum *transmission interval*, from which a number of discrete values are chosen to configure FTSP. This same range is used to bind D-FTSP's *transmission interval* so that test parameters are more consistent and results are more comparable.

The final factor is concerned with quantifying the performance of D-FTSP relative to FTSP. To accomplish this, suitable metrics must be chosen. An obvious feature of interest is the time accuracy provided by each protocol. To obtain this data within a WSN test-bed, the clock offset of each node for the duration of each experiment must be recorded. To compare the overall performance of each protocol in terms of accuracy, an average value that represents the synchronisation accuracy of the entire network is chosen. Thus, a metric termed the *Global Mean Offset* (μ) is selected. This represents the average absolute offset of all nodes from the root node for the duration of the experiment.

The calculation of a node's offset (θ) relative to its associated root node is presented in equation 4.6. In equation 4.6, θ_{it} represents the offset of node i at time t, while T_{it} and T_{rt} represent the timestamps as recorded by node i and its associated root r at time t. The calculation of the *Global Mean Offset* (μ) is presented in equation 4.7. In equation 4.7, N represents the number of nodes

in the WSN and D represents the duration of the experiment. It must be noted that the absolute value of a node's offset, $|\theta_{it}|$, is chosen because the magnitude of a node's offset from a root node is generally what impacts a WSN application, not the sign.

$$\theta_{it} = T_{rt} - T_{it} \tag{4.6}$$

$$\mu = \frac{\sum_{t=1}^{D} \cdot \left(\sum_{i=1}^{N} |\theta_{it}| \right)}{D \cdot N} \tag{4.7}$$

The second metric chosen quantifies the efficiency of each protocol, since this is the goal of D-FTSP. Given that the method by which a dynamic time protocol achieves efficiency is by altering the transmission interval, it seems natural that the most suitable metric is the number of message transmissions. Thus, a metric termed the *Global Number of Transmissions* is chosen which represents the total number of message transmissions that occur throughout the network for the duration of an experiment.

4.2.1 Testbed

Fig. 4.4 illustrates the testbed used in the experimentation. The entire testbed consists of thirteen WSN nodes. Twelve of these nodes are used to construct a multi-hop WSN whereas the thirteenth node is used to query the value of their clocks. The WSN nodes are configured with ID's 1 to 12 and, thus, node 1 represents the synchronisation root. Fig. 4.4 illustrates the communication links between each node. The root can communicate with nodes 2 and 3 both of which can also communicate with each other. This structure is repeated for subsequent pairs of nodes which adds additional hops to the network. Node 12 represents the final node and is located five hops from the root.

This structure does not represent the actual physical structure of the WSN, since all the nodes share a direct communication link. This structure is a logical one that is programmed into both the FTSP and D-FTSP protocols. Hence, the FTSP/D-FTSP application running on node 2 is programmed to except time messages from nodes 1, 3, 4 and 5 only, and so on. The reason for this logical

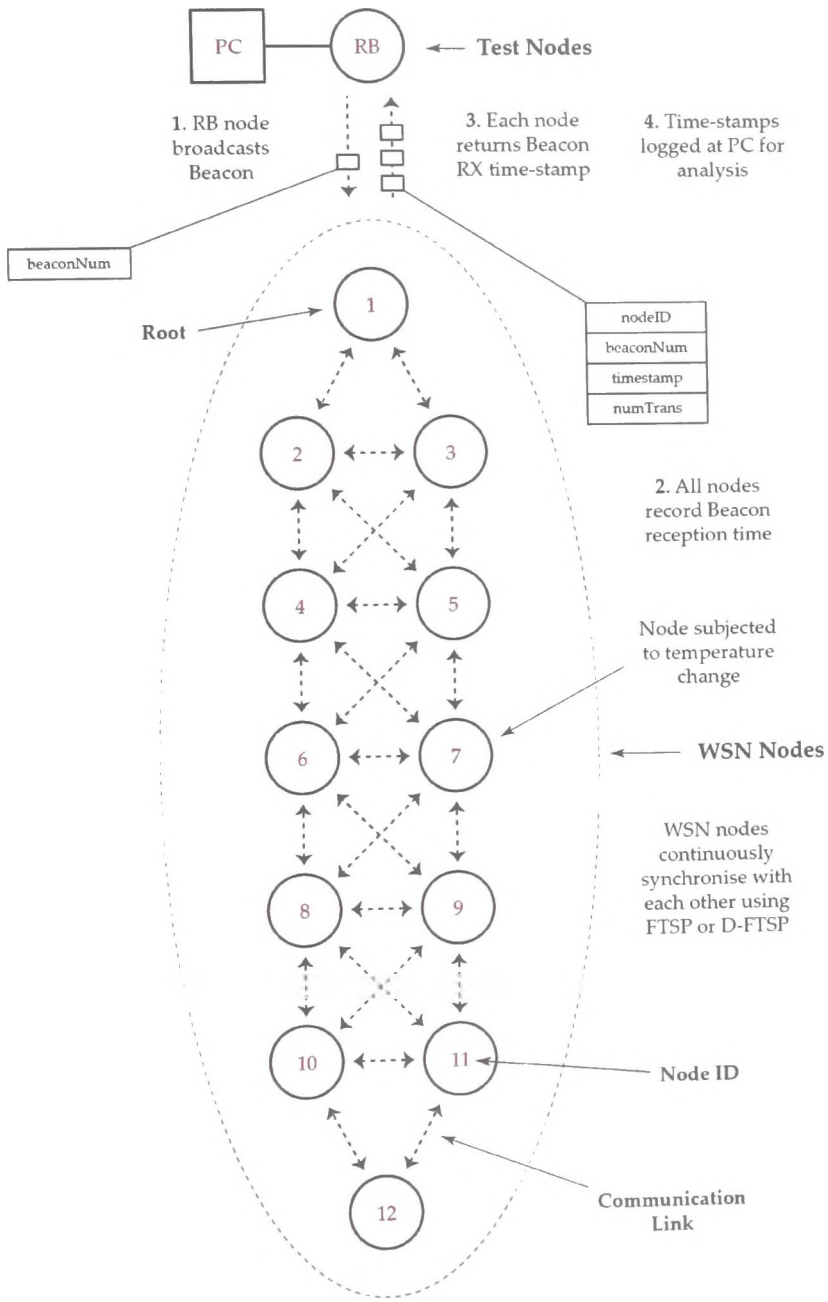


Figure 4.4: Testbed

network structure is to emulate a dispersed multi-hop WSN and permit convenient data acquisition from the WSN for analysis.

The technique by which data is acquired from the WSN, in order to measure the performance of FTSP/D-FTSP, partially imitates the reference broadcast technique described in section 2.4.2. The thirteenth node, termed the *RB* node (*Reference Broadcast node*), is positioned such that it shares a direct communication link with all twelve nodes that comprise the synchronisation network. This *RB* node is configured to periodically broadcast a beacon message which all twelve nodes can detect. Each node records the reception time of a beacon message and subsequently transmits this timestamp, along with a total value for the number of time messages it has transmitted up to that point, back to the *RB* node. The RB node also acts as base-station that transmits the received timestamps to a connected PC for logging and analysis.

The advantage of using reference broadcasting to acquire timestamps is that all nodes overhear a beacon message at almost the same instant. In fact, the actual time that each node detects the same beacon message differs by no more than a fraction of a microsecond and, therefore, produces errors that are negligible in terms of the accuracies a WSN time protocol can achieve (microseconds). Thus, a very accurate value for the offset of each node, at a particular point in time, can be obtained.

To enable simultaneous synchronisation and testing, the synchronising nodes must host more than one application. Fig. 4.5 illustrates the applications that run on each of the nodes in the testbed. Synchronising nodes run a synchronisation protocol (FTSP/D-FTSP) as well as a test application. The test application records beacon reception timestamps and transmits them back to the RB node. The RB node runs a single application that generates periodic beacon messages, collects the responses from each synchronising node, and transmits them to a PC. The application running on the PC analyses the generated log files and outputs values for the metrics used to compare the protocols.

The testbed presented so far is used to simulate a stable environment, but it must be modified such that it can simulate an unstable environment or, more specifically, cause node' clock's to drift. A simplistic and controllable method of achieving this entails altering the ambient temperature of the nodes. It is

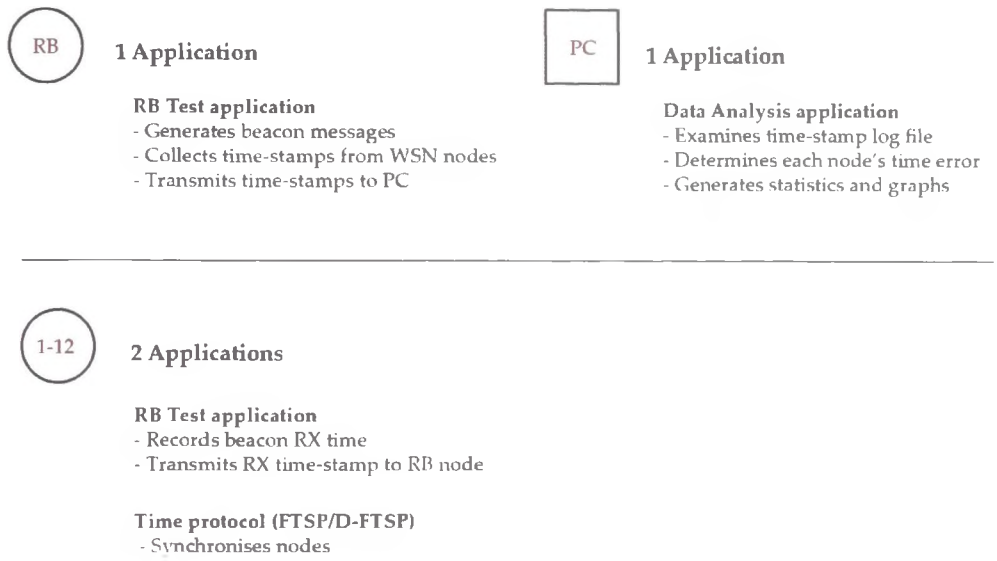


Figure 4.5: Testbed applications

also wise not to complicate the experiments such that too many variables are introduced, since this would make analysis difficult. Thus, rather than alter the ambient temperature of a subset of the nodes, a single node is chosen and solely subjected to temperature changes. Node 7 is considered a reasonable choice, since it is located mid-way between the root and the extremity of the network (node 12). Thus, when node 7's clock begins to drift, the error it accumulates affects nodes at higher hops but only affects half of the network. This makes the task of isolating causes and their associated effects, from the data, less complicated.

4.2.2 Hardware Details

The testbed detailed in the last section is a generic one and, therefore, details of the software and hardware employed in the actual experimentation are omitted. The hardware employed in the actual testbed consists of thirteen *TelosB* motes [38]. A TelosB mote is composed of an 8MHz MSP430 microcontroller, with 10KB RAM; an IEEE 802.15.4 compliant CC2420 transceiver, that operates at

250Kbps; and a 32.768KHz external quartz crystal. The MSP430 microcontroller has two timers, Timer A and B, each with multiple capture registers (see section 3.4). The Timer B counter register is driven by the external 32.768KHz quartz crystal. One of Timer B's associated capture registers, $B1$, is connected to the SFD (*Start Frame Delimiter*) interrupt line of the CC2420 transceiver. When the transceiver has fully received or transmitted the SFD field of a message, it sets its SFD line high, and the contents of Timer B's counter register is copied into the capture register ($B1$). This whole process allows the reception and transmission times of a time message to be recorded at the physical layer, thus, eliminating non-deterministic delays associated with microcontroller interrupts.

With regard to the particular experiments in this work, the advantage of using such hardware is that most of the time error that might result from non-deterministic message latencies is eliminated. Such error might otherwise be incorrectly associated with FTSP or D-FTSP, thus, affecting the interpretation of the results.

4.2.3 TinyOS

The TelosB nodes are operated by *TinyOS* [39], an open source embedded operating system designed for low-power wireless devices, particularly wireless sensors. TinyOS is a *non-blocking* OS that employs a single stack and, consequently, all I/O operations are *asynchronous*, otherwise known as *split-phase*.

The use of split-phase operations offers advantages that are suited to sensor nodes. Firstly, they enable several operations to run in parallel, since the system does not have to block until an operation completes. This results in a very responsive system, an important factor in WSNs. They're use can also lead to memory savings, since no state must be stored on a call stack. Nonetheless, this method of operation requires extremely efficient and optimised code. For this reason, TinyOS employs the *nesC* programming language which is a dialect of C that allows for the production of optimised code.

A TinyOS program is built out of constructs termed *components* which provide and use entities termed *interfaces*. An Interface is composed of two types of functions termed *commands* and *events*. The execution of a command initiates a

particular action and the completion of this action is signalled via a corresponding event which is called in response to an external interrupt. Thus, an event serves as a call-back function for a command and is passed information related to the outcome of the command. A command and its corresponding event split up invocation and completion, thus, permitting non-blocking/split-phase operations that do not seize stack memory.

Components are categorised into two types: *Modules* and *Configurations*. Modules provide the implementation code for one or more interfaces, that is, the implementation code for the commands of interfaces they provide and the implementation code for the events of interfaces they use. Conversely, configurations provide details related to how components are assembled together. They declare the connections between interfaces used by components and interfaces provided by components. This is termed *wiring*. It is this explicit static wiring by a developer that allows for the generation of efficient and optimised code. Of course, the disadvantage is that such code can be quite verbose and complex.

In addition to commands and events which are associated with I/O operations, TinyOS also defines an additional construct termed a *Task*. Unlike event handlers which are composed of *synchronised* code that cannot be pre-empted, tasks are composed of *asynchronous* code that can be deferred but solely by event handlers. Tasks generally consist of relatively lengthily computations that do not have strict time constraints. They are scheduled by TinyOS in FIFO order and, therefore, a task must fully complete before another task can be executed. Due to the scheduling algorithm employed, a developer must take care when implementing tasks so that the tasks do not consume a lot of processing time thereby delaying subsequent tasks.

In order to clarify the above explanation, fig. 4.6 presents the structure of a generic TinyOS application. Blocks *Ma*, *Mb* and *Mc* represent three module components. Module *Ma* uses two interfaces, namely *Ix* and *Iboot*, which are provided by modules *Mb* and *Mc* respectively. The configuration component defines the wiring between these interfaces by specifying their users, which are placed on the right of the arrow, and their corresponding providers, which are placed on the left of the arrow.

In this example, the interface *Iboot*, which is provided by module *Mc*, specifies

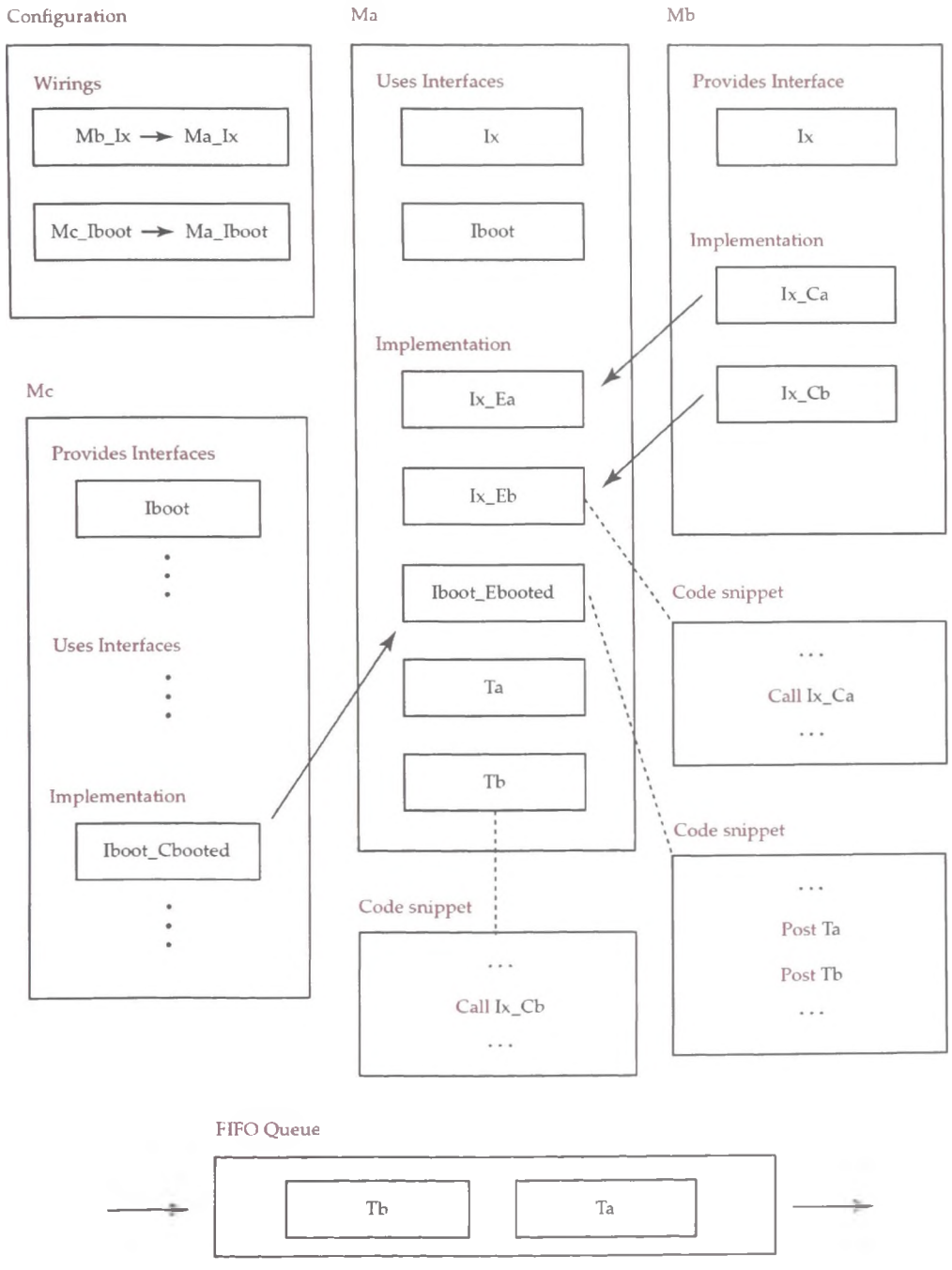


Figure 4.6: TinyOS application structure

a command called *Cbooted*, which is implemented by *Mc*. The *Cbooted* command signals *Ma*, a user of the *Iboot* interface, when the system has booted. It does so by calling the corresponding *Ebooted* event which is defined by *Ma*. The *Ebooted* event, defined by *Ma*, schedules two tasks which it has defined, *Ta* and *Tb*. This is achieved via nesC's *post* keyword which places the tasks into a FIFO queue. Task *Ta* is executed fully and when complete task *Tb* acquires the microcontroller. Task *Tb* in response calls the command *Cb* which is part of the *Ix* interface and provided by module *Mb*. The invocation of a command is preformed via nesC's *call* keyword. Although not illustrated in fig. 4.6, command *Cb* may schedule tasks and call subsequent commands, possibly obtaining data from an I/O device. When command *Cb* completes, it calls the corresponding *Eb* event which is defined by *Ma*. Event *Eb* in response calls command *Ca* provided by module *Mb* and when complete, the corresponding event *Ea* is triggered. This in essence represents the typical flow of a TinyOS application.

In relation to TinyOS and the experiments detailed in this chapter, one thing that should be mentioned is the mechanism by which messages are routed to the appropriate application running on a WSN node. TinyOS provides the *Active Message (AM)* communication layer to multiplex access to the radio. A lower layer message field termed *AM type* is responsible for this multiplexing. AM types are similar in function to the *UDP (User Datagram Protocol)* port in that they identify the application on the node to route the messages to. This explains how the synchronising nodes illustrated in fig. 4.5 can route beacon messages and time messages to their specific applications.

4.2.4 Experiments

D-FTSP is targeted at WSNs that must operate in dynamic environments. Such enviroments may subject clocks to substantial drift over relatively short intervals. In relation to the testbed and as detailed in section 4.2.1, the simulation of an unstable environment is accomplished by altering the ambient temperature of node 7. More specifically, node 7's ambient temperature is increased by 20 °C over a 30 second interval, kept at this temperature for 15 minutes, and then allowed to cool naturally. This results in a sharp spike in node 7's ambient

temperature followed by a gradual one. The values chosen represent an extreme temperature change over a relatively short interval. They are chosen with respect to those dynamic environments detailed at the beginning of this chapter, with a particular focus on data centres and the specifications detailed in [37]. A variation of this degree, although extreme, can occur in such artificial environments and the use of such extreme values for experimental purposes allows one to determine the potential of D-FTSP with respect to its effectiveness in its targeted environment.

With regards to the configuration of D-FTSP, a minimum and maximum transmission interval of 30 and 180 seconds, respectively, and an error bound, ϵ_{avg}, of 1 tick are chosen. 1 tick represents the highest accuracy achievable, since it also denotes the granularity of the physical clock. Since TelosB motes are used in the testbed, each of which employs a 32.768KHz clock, 1 tick is approximately equal to 30.5μs. A minimum transmission interval of 30s is chosen because it is found via experimentation that intervals below this do not result in any significant performance improvement. Thus, the elimination of intervals below this value simplifies the task of interpreting collected data. A maximum transmission interval of 180s is chosen because, due to the dynamic nature of the test environment and the targeted error bound, in a real world scenario it would be irrational to configure FTSP with a larger transmission interval. Thus, larger transmission intervals are eliminated, further simplifying the task of interpreting data.

In relation to FTSP and due to its static nature, multiple discrete values between D-FTSP's minimum and maximum *transmission interval* are chosen. More specifically, the values 30, 60, 120, and 180 seconds are selected.

The final experimentation procedure consists of ten individual experiments which are run for a duration of 100 minutes. Two of these entail D-FTSP operating in both the stable and unstable environments. The remaining eight entail FTSP, configured with *transmission intervals* (τ) of 30, 60, 120, and 180 seconds, operating in both the stable and unstable environment (see table 4.1).

	D-FTSP	FTSP	FTSP	FTSP	FTSP
Stable	(1) $\epsilon_{avg} = 1$	(2) $\tau = 30s$	(3) $\tau = 60s$	(4) $\tau = 120s$	(5) $\tau = 180s$
Unstable	(6) $\epsilon_{avg} = 1$	(7) $\tau = 30s$	(8) $\tau = 60s$	(9) $\tau = 120s$	(10) $\tau = 180s$

Table 4.1: Experiments and configuration parameters

4.2.5 Results

As stated in section 4.2, the two metrics chosen to compare and contrast D-FTSP and FTSP are the *global number of transmissions* and the *global mean offset*. The values of these metrics for each of the experiments performed are presented in table 4.2 and illustrated in fig. 4.7.

The total number of messages transmitted by FTSP is a deterministic value that is dependent on the value of the *transmission interval* (τ), the duration of the experiment (Δ), and the number of nodes in the network (N) (see equation 4.8). In these experiments the duration, Δ, is equal to 600s and the number of nodes, N, is equal to 12.

$$N \cdot (\Delta/\tau) \tag{4.8}$$

The total number of message transmissions for D-FTSP is non-deterministic and is revealed through experimentation. The results in table 4.2 reveal that D-FTSP transmits 502 messages in the stable environment and 634 in the unstable environment.

Naturally, the performance of D-FTSP cannot be determined based on the number of message transmissions alone. The accuracy achieved by both protocols must also be factored in. Table 4.2 reveals that D-FTSP and FTSP perform equally well in the stable environment. An important point is that the value of FTSP's transmission interval does not significantly influence the accuracy it achieves. This concurs with the logic that in the absence of clock drift the number of message transmissions can be reduced. This is exactly what D-FTSP does. Consequently, D-FTSP finds a suitable value for each node's *transmission interval* in order to adhere to the application error bound of 1 tick.

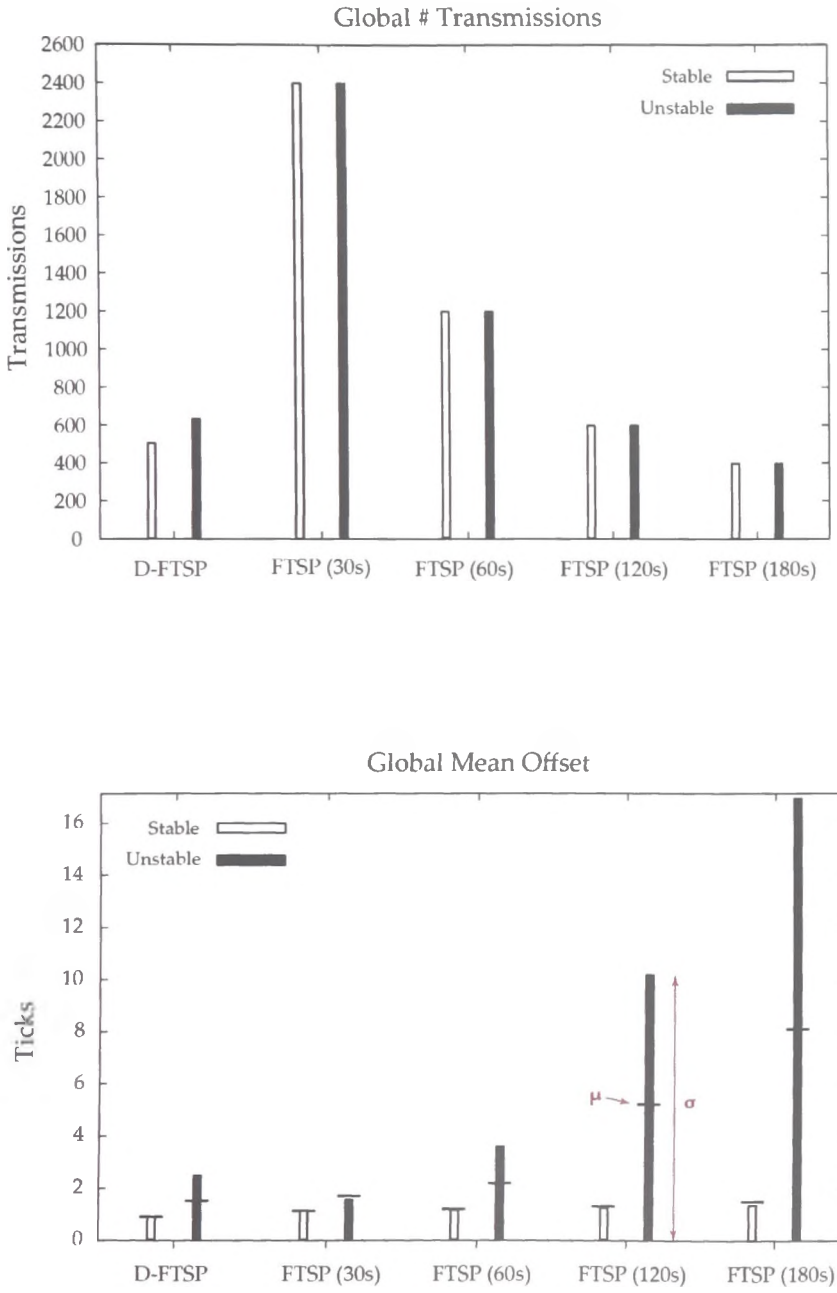


Figure 4.7: Global number of transmissions (top). Absolute global mean offset (μ) and standard deviation (σ) (bottom) (When interpreting the graph note that the height of the vertical bar represents the magnitude of σ)

	D-FTSP ($\epsilon_{avg} = 1$)	**FTSP** ($\tau = 30s$)	**FTSP** ($\tau = 60s$)	**FTSP** ($\tau = 120s$)	**FTSP** ($\tau = 180s$)
#					
Stable	502	2400	1200	600	400
Unstable	634	2400	1200	600	400
μ					
Stable	0.93	1.15	1.21	1.29	1.52
Unstable	1.50	1.61	2.18	5.18	8.12

Table 4.2: Global number of transmissions and absolute global mean offset (μ) (ticks)

The results associated with the unstable environment are more informative. They reveal that the value of FTSP's transmission interval significantly influences the accuracy it achieves. As expected, FTSP adheres to the error bound best when configured with the lowest transmission interval of 30s. When configured with the highest value of 180s, the mean accuracy achieved is over 8 ticks, that is, over five times less accurate. More importantly, the benefits of D-FTSP are revealed. D-FTSP adheres best to the application error bound and achieves a mean offset of 1.5 ticks with 634 message transmissions, just 26% more message transmissions than in the stable environment. Consequently, D-FTSP achieves a similar degree of accuracy to FTSP configured with a transmission interval of 30s and does so with almost 75% less message transmissions.

As detailed in section 4.1, D-FTSP deals with clock drift by forcing a node's parent(s) to transmit more time messages to it. Consequently, in the D-FTSP experiment performed in the unstable environment, because node 7's ambient temperature changes, its parents, nodes 4 and 5, should transmit significantly more messages than the remaining nodes. Fig. 4.8 presents the total number of messages transmitted by each node for the duration of the experiment. It reveals that this in fact does occur, since nodes 4 and 5 transmit up to 50% more messages than those nodes located at lower hops. D-FTSP, thus, ensures that only those nodes that must transmit more messages do so.

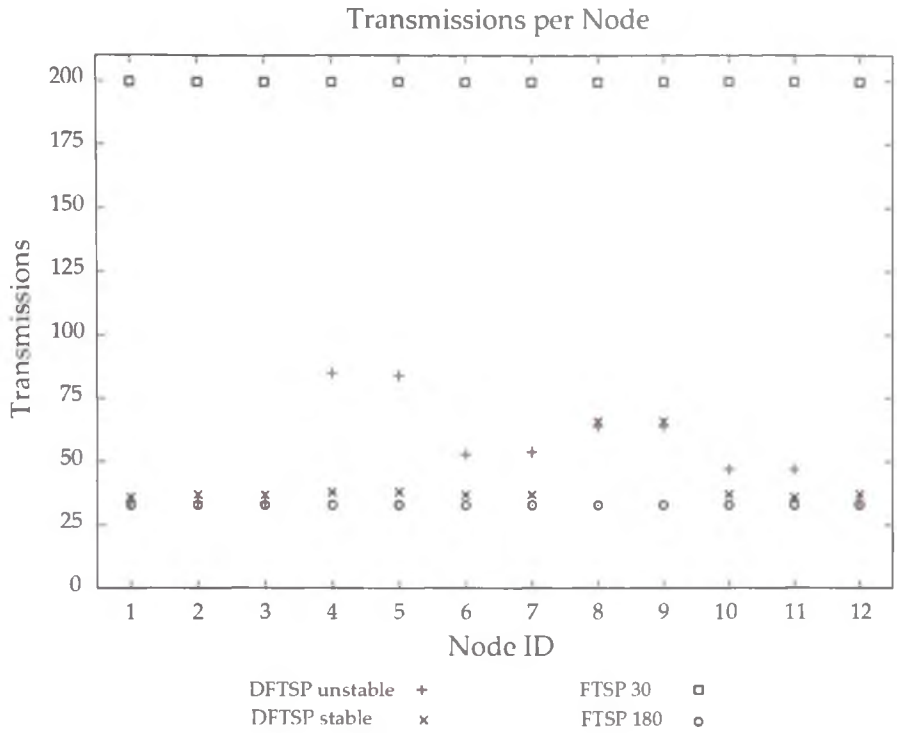


Figure 4.8: Number of transmssions per node

The metrics used to compare D-FTSP with FTSP show that D-FTSP outperforms FTSP in terms of accuracy and efficiency. The focus, in terms of accuracy, has been on the *global mean offset* statistic. Additional important statistics are the maximum offset and the variation of the offsets, which are presented in fig. 4.11 and fig. 4.7 respectively. When interpreting the standard deviation (σ) in fig. 4.7, note that the height of the vertical bars represent the magnitude of σ, thus, the graph should not be interpreted as a box-plot would be.

An example of the raw experimental data collected for two of the FTSP experiments is presented in fig. 4.9 and fig. 4.10. The histogram in fig. 4.9 illustrates the combined offsets of all nodes for the FTSP experiment performed in the stable environment with a transmission interval (τ) of 30s. The histogram in fig. 4.10 illustrates the combined offsets of all nodes for the FTSP experiment performed in the stable environment with a transmission interval (τ) of 180s. One may observe that the offset values in both experiments follow a slightly skewed nor-

mal distribution. This subtle skew is most likely due to the variation between the nodes' clocks and the root node's clock. If so, a different choice of root node might result in a more symmetric normal curve. In addition, the curve in fig. 4.10 is less peaked than that in fig. 4.9. This, of course, is due to the larger transmission interval (τ) of 180s which results in greater error. Of course, because these experiments are performed in a stable environment, the additional error is almost negligible with respect to a single clock tick.

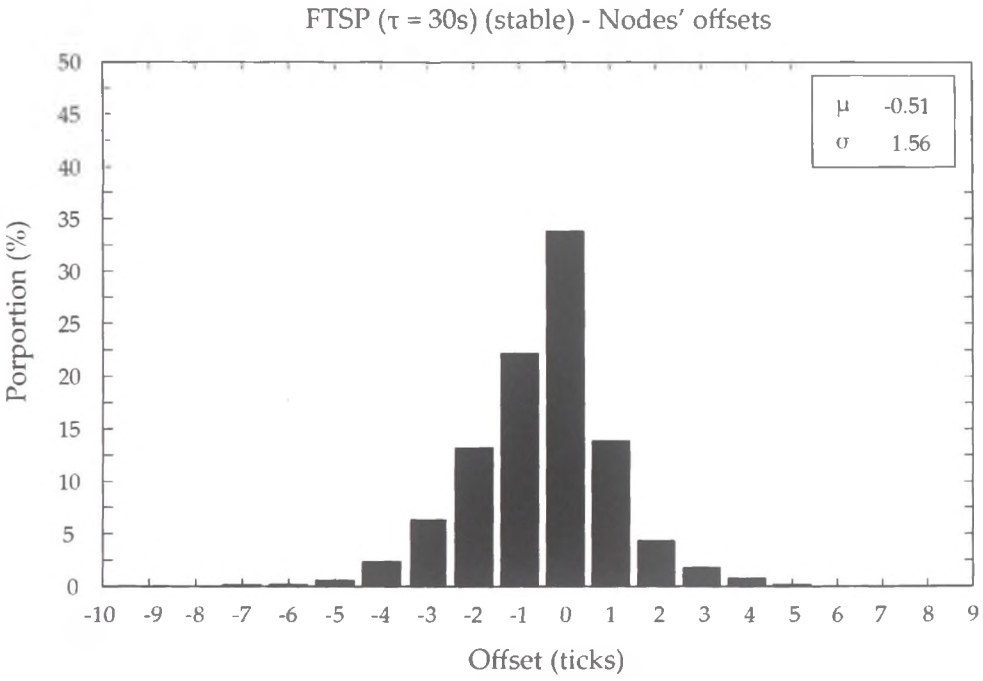


Figure 4.9: Raw data for FTSP experiment in stable environment with $\tau = 30$s

Finally, one may also observe that the mean offset is less than zero for both experiments. Of course, as stated in section 4.2, it is the magnitude of the offset that is important when judging a time synchronisation protocol, since the protocol's primary objective is to reduce this offset, regardless of its sign. When these values are converted to absolute values, the distribution of the offsets resemble a folded-normal curve and, thus, the mean and standard deviation differ as is evident in fig. 4.7.

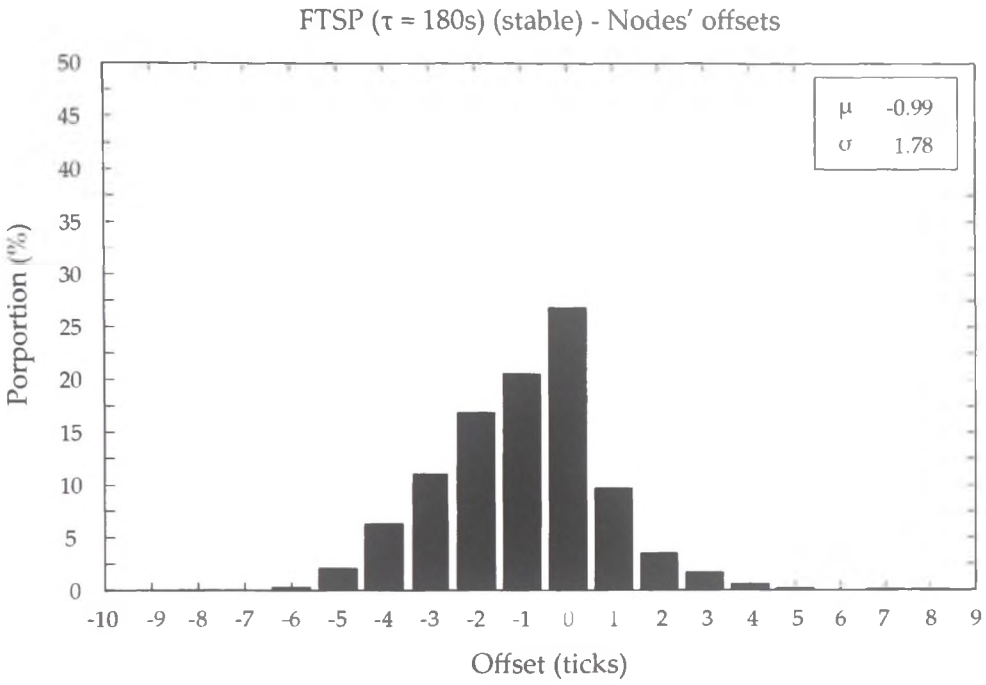


Figure 4.10: Raw data for FTSP experiment in stable environment with $\tau = 180$s

Figures 4.11 and 4.7 reveal that though D-FTSP achieves, on average, better accuracy in the unstable environment, the standard deviation (σ) of the offsets and, in particular, the maximum offset are relatively high. This can be attributed to the time required for nodes 4 and 5 to detect node 7's drift. This in turn is due to the fact that each node, in advance of the temperature disturbance, operates at the maximum transmission interval of 180s. Node 7, thus, in the worst case scenario, must wait this interval before it receives a time message and can update its clock skew.

Fig. 4.11 depicts the aforementioned scenario. It presents a graph of the ambient temperature, clock offset and clock skew of node 7 for the D-FTSP experiment performed in the unstable environment. As node 7's ambient temperature increases, so too does its offset. It eventually receives a time message from either node 4 or 5 and calculates its skew which it subsequently broadcasts in a time message. Nodes 4 and 5 determine from this message that node 7's clock

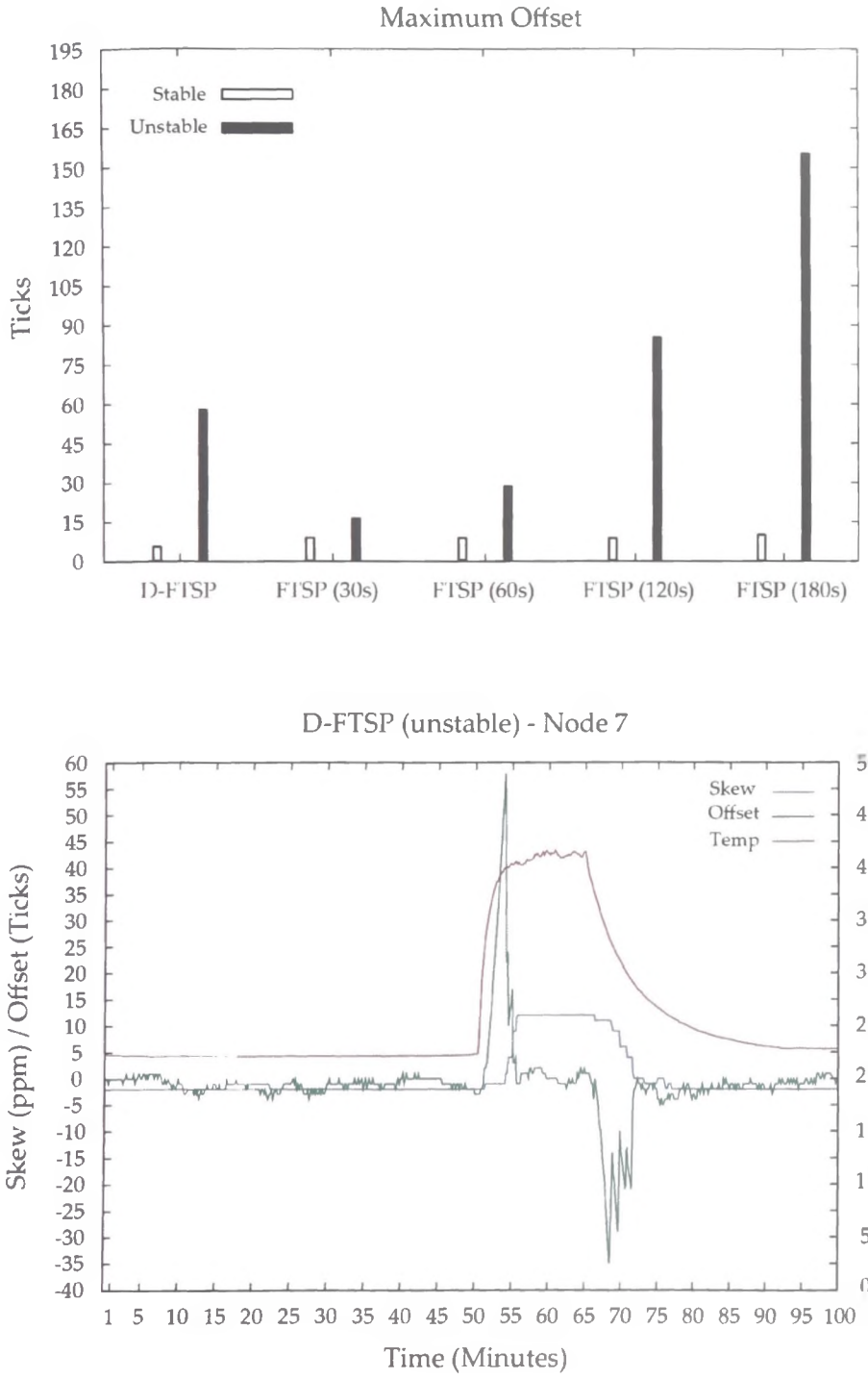


Figure 4.11: Maximum offset for each experiment (top). Node 7's temperature, offset, and skew during D-FTSP experiment in un-stable environment (bottom).

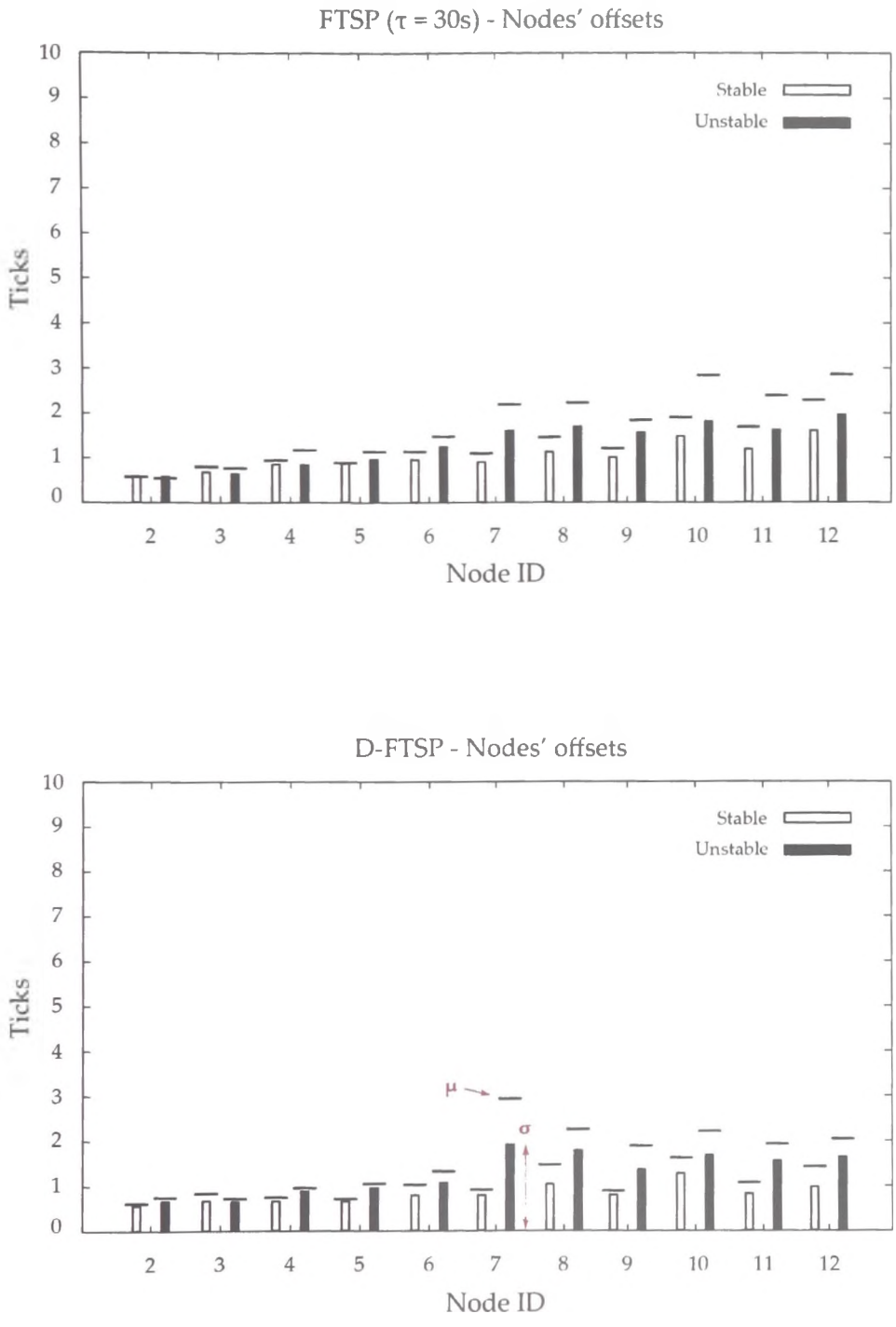


Figure 4.12: Nodes' mean offset (μ) and standard deviation (σ) for FTSP (top) and D-FTSP (bottom) in stable and unstable environment. (When interpreting the graph note that the height of the virtical bar represents the magnitude of σ)

is drifting and, in response, increase their transmission rate. Consequently, node 7 receives more data which it uses to determine an accurate value for its clock skew. Overall, the temperature excursion alters node 7's skew by a magnitude of $16ppm$.

Fig. 4.12 presents more detailed statistics. It displays the mean and standard deviation of each node's offset for both of the D-FTSP experiments. In addition, it displays the mean and standard deviation of each node's offset for both of the FTSP experiments associated with the *transmission interval* of $30s$. Referring to fig. 4.12 and in relation to those experiments performed in the unstable environment, the mean and standard deviation of node 7's offset is higher for D-FTSP. Thus, although D-FTSP, on average, performs better in terms of accuracy, there is a short minor degradation in accuracy during the temperature excursion. Of course, the magnitude of this degradation is dependent on the structure of the network and the maximum possible transmission interval permitted. The WSN testbed employed in the experimentation represents an extremely sparse network. If it were a dense one, a node experiencing clock drift would most likely have more parents and, thus, receive time messages more regularly. Such a scenario would definitely alleviate this affect.

4.2.6 Energy Analysis

The results in the previous section indicate that in an unstable environment D-FTSP can achieve similar accuracies to that of FTSP with significantly less message transmissions. More specifically, in these particular experiments, a 75% message transmission reduction was observed. This, however, does not translate directly into pro rata energy savings. D-FTSP transmits two extra fields, and this fact must be considered when quantifying D-FTSP's net energy saving capabilities.

The actual energy consumed by a transceiver engaged in a message exchange varies depending on factors such as the communication distance, the data rate, the message size and the modulation scheme employed. A detailed analysis of a transceiver's energy consumption is provided by Wang et al. [40]. Referring to this work and with respect to the broadcast nature of both FTSP and D-

FTSP, the energy required to transmit a single bit is dependent on the power consumption of the various circuitry components of the transceiver (i.e. *digital-to-analog converter*, *low pass filter*, *bandpass filter*, *mixer*, *frequency synthesiser*, *power amplifier*), the transmit power and the time it takes to transmit a single bit. To simplify the task of evaluating energy consumption, it is viewed as been directly proportional to the message size.

An FTSP time message contains a payload of 56 bits, whereas a D-FTSP message contains two extra fields, the *hop* and *skew* fields, each 8 bits in size. Hence, D-FTSP transmits 28% more bits per message and, thus, in the worst case, will consume 28% more energy per message. If one denotes e as the total energy required to transmit 1 bit of information on a particular platform, then the total energy required to transmit a D-FTSP message (E_{DM}) and FTSP message (E_{FM}) is $72e$ and $56e$ respectively. Thus, the relationship, in terms of energy consumption, between the transmission of a D-FTSP and FTSP message is represented by equations 4.9 and 4.10 respectively.

$$E_{DM} = 1.28E_{FM} \tag{4.9}$$

$$E_{FM} = 0.78125E_{DM} \tag{4.10}$$

This indicates that D-FTSP must transmit approximately 22% less messages than FTSP in order to consume the same amount of energy. In the experiment performed in the unstable environment, D-FTSP and FTSP achieve similar accuracies with 634 and 2400 message transmissions respectively. One can determine the equivalent number of FTSP message transmissions required to consume the same amount of energy that D-FTSP consumed using equation 4.11.

$$N(E_{DM}) = N(1.28E_{FM}) \tag{4.11}$$

Thus, in terms of energy, 634 D-FTSP message transmissions is equivalent to approximately 811 FTSP message transmissions. Thus, in that particular experiment, D-FTSP consumed 66% less energy as calculated in equation 4.13.

$$634(E_{DM}) = 811.52(E_{FM}) \tag{4.12}$$

$$100(1 - (811.52/2400)) = 66.2 \tag{4.13}$$

This is a substantial energy saving, but this analysis also highlights the fact that if energy savings is a priority, then D-FTSP is more suited to unstable environments.

In relation to CPU energy utilisation, the work by Shnayder et al. [41] provides a detailed breakdown of the energy consumed by various hardware components found in the Mica2 platform [42]. The authors run multiple experiments which measure the energy consumed by each hardware element when running particular applications provided by the TinyOS distribution. An interesting fact about microcontrollers employed by motes is that they consume approximately a constant amount of power while executing instructions. This is due to the fact that they do not incorporate the sophisticated energy saving techniques present in more advanced processors. According to [41] and [42], the microcontroller (*ATmega128L*) employed by a *Mica2* mote draws 8mA in active mode, and the mote's transceiver (*CC1000*) draws up to 27mA at maximum transmission power. In the case of a Telosb mote, according to [38] and [43], its microcontroller when active draws 1.8mA, whereas its transceiver when active can draw up to 17.4mA in transmit mode at maximum transmission power. As one can see, the type of platform has a large influence on the energy consumed. Additionally, the *ATmega128L* microcontroller when in *idle* state draws about a third of the current it does in the *active* state. This is interesting, since many of the applications analysed by Shnayder et al. [41] do not place the CPU from an *idle* state into a *Power-save* state and, therefore, the CPU, although rarely active, consumes a significant proportion of the overall power when these applications are running

In relation to the comparison of D-FTSP to FTSP and with regard to CPU energy consumption, any extra energy consumed by D-FTSP is directly proportional to the time required to execute its additional code. D-FTSP employs two additional blocks of code. The first block of code is concerned with the management of data related to a node's child/children. Data related to a particular child

is stored as a C *struct* which itself is stored in an array. When a D-FTSP node receives a new message from a child, it performs a linear search of an array for the appropriate child entry and updates the corresponding *struct*. In addition to this, a new drift value for the child is calculated, compared with the current highest drift value and updated if necessary. The time taken to perform this operation is dependent on the number of children (N_c) a node has, since this will represent the size of the table that must be searched. There are also other factors at play when this is analysed from a network point of view. This operation will be performed every time a node receives a message from a child. Since each child may have a different transmission interval, an accurate analysis of CPU energy consumption is a difficult task.

The second block of code is responsible for the update of a node's transmission interval. The algorithm was presented in algorithm 1. This operation is performed when a higher drift value for a child is observed. Thus, the number of times it is performed is dependent on the dynamics of the environment and the stability of the nodes.

To determine if the extra CPU operations performed by D-FTSP result in significant energy consumption relative to FTSP, one must formulate numerous models that model D-FTSP's energy consumption in a WSN. However, it is easier to analyse the problem by comparing the extra energy consumed by these operations in terms of the energy consumed by a packet transmission. If the relative energy consumed by the CPU is negligible, then it can be ignored since even a small reduction in the number of packet transmissions by D-FTSP relative to FTSP will account for it.

To perform the analysis, some notation is required and is presented in table 4.3. The objective is to determine R_E which represents the ratio of energy required by a CPU to execute the additional D-FTSP operations relative to the energy required to transmit a D-FTSP packet. The D-FTSP operation that is analysed is the one responsible for maintaining the child table. This involves a linear search of the child table followed by an update of a child entry. The number of CPU cycles required to perform each task is represented by ST_{cyc} and UT_{cyc} respectively. The size of the child table is directly related to the number of children a node possess and is represented by N_c. This will have a direct impact

Symbol	Definition	Unit
P_s	D-FTSP packet Size	bit(b)
T_{TXb}	Time to transmit a bit	Second(s)
ST_{cyc}	# Cycles to search child table	# Cycles
UT_{cyc}	# Cycles to update child table	# Cycles
N_c	Number of children	#
T_{cyc}	Time taken for 1 clock cycle	Second(s)
CPU_{AI}	CPU active current	Ampere (A)
TTX_{AI}	Transceiver active current (TX mode)	Ampere (A)
CPU_v	CPU voltage	Volt (V)
TTX_v	Transceiver voltage	Volt (V)
CPU_E	Energy consumed by CPU	Joule (J)
TTX_E	Energy consumed by Transceiver	Joule (J)
TX_{br}	Transceiver transmission rate	Bits per second (bps)
CPU_{clc}	CPU clock rate	Hertz (Hz)
R_E	Energy consumption ratio	#

Table 4.3: Notation

on the time required to update the table. Since the table is searched in a linear fashion, the average number of cycles required to maintain the table is calculated using equation 4.14.

The amount of time taken for one cycle, represented by T_{cyc}, is dependent on the clock speed of the CPU, denoted CPU_{clc}, and is calculated using equation 4.15. From equation 4.15, one can calculate the average amount of time required to maintain the child table which is given in equation 4.16.

$$(ST_{cyc} \cdot \frac{N}{2}) + UT_{cyc} \tag{4.14}$$

$$T_{cyc} = \frac{1}{CPU_{clc}} \tag{4.15}$$

$$T_{cyc} \cdot ((ST_{cyc} \cdot \frac{N}{2}) + UT_{cyc}) \tag{4.16}$$

Next one must determine the average energy consumed by the CPU when maintaining the child table. The power consumption of the CPU is given in equation 4.17. To resolve the average energy consumption, represented by CPU_E, one applies equation 4.18. Thus, an equation for the average energy consumed by a CPU for the aforementioned operation has been formulated.

$$CPU_{AI} \cdot CPU_V \tag{4.17}$$

$$CPU_E = (CPU_{AI} \cdot CPU_V) \cdot (T_{cyc} \cdot ((ST_{cyc} \cdot \frac{N}{2}) + UT_{cyc})) \tag{4.18}$$

Next one must formulate an equation for the energy consumed by a transceiver when it transmits a D-FTSP message. First, the time required to transmit a message is deduced. The time required to transmit a single bit, represented by T_{TXb}, is dependent on the data rate, denoted by TX_{br}, of the transceiver and is given in equation 4.19. From this, the time required to transmit a single message can be determined using equation 4.20. The power consumed by a transceiver in transmit mode is given in equation 4.21. With respect to equations 4.20 and 4.21, the total energy consumed by the transceiver when it transmits a D-FTSP message, represented by TTX_E, can be calculated using equation 4.22.

$$T_{TXb} = \frac{1}{TX_{br}} \tag{4.19}$$

$$P_s \cdot T_{TXb} \tag{4.20}$$

$$TTX_{AI} \cdot TTX_V \tag{4.21}$$

$$TTX_E = (TTX_{AI} \cdot TTX_V) \cdot (P_s \cdot T_{TXb}) \tag{4.22}$$

Finally, to calculate R_E, one calculates x, which represents the ratio of CPU power consumption to transceiver power consumption; and y, which represents

the ratio of time expended by the CPU to time expended by the transceiver. These are calculated using equations 4.23 and 4.24 respectively. It is reasonable to assume that the voltage applied to the CPU and transceiver is approximately equal and, thus, x can be simplified by cancelling out the voltage terms. R_E is calculated as the product of x and y and given in equation 4.25.

$$x = \frac{TTX_{AI} \cdot TTX_V}{CPU_{AI} \cdot CPU_V} = \frac{TTX_{AI}}{CPU_{AI}} \tag{4.23}$$

$$y = \frac{(p_s \cdot T_{TXb})}{T_{cyc} \cdot ((ST_{cyc} \cdot \frac{N}{2}) + UT_{cyc})} \tag{4.24}$$

$$R_E = \frac{TTX_{AI} \cdot (p_s \cdot T_{TXb})}{CPU_{AI} \cdot (T_{cyc} \cdot ((ST_{cyc} \cdot \frac{N}{2}) + UT_{cyc}))} \tag{4.25}$$

The formula in 4.25 can be applied to any particular platform. As stated previously, the relative energy consumption of the particular hardware elements varies between platforms. However, the focus here is on the *TelosB* platform which the experiments detailed in section 4.2.4 were performed on.

The *TelosB* platform employs an *MPS430* microcontroller and a *CC2420* transceiver. Referring to their corresponding data sheets [43, 44], the values for the required variables in equation 4.25 are presented in table 4.4. The determination of ST_{cyc} and UT_{cyc} is more challenging. One must determine the number of *MPS430* machine cycles required to execute the D-FTSP code block. This can be achieved by first compiling the code into *MPS430* assembly and then converting it to corresponding hex code. One can then use *naken430asm*, by Kohn [45], which disassembles the hex file and lists the number of machine cycles required for each operation. The values are presented in table 4.5. It takes approximately 28 machine cycles to search a position in the table and 113 machine cycles to update a child entry. In addition, the value of N_c is set to 10 which represents a relatively large number of children, indicative of an very dense WSN.

Plugging the values into equation 4.25 results in a value of approximately 872 (equation 4.26). This value can be interpreted as follows: the additional CPU operations required by D-FTSP relative to FTSP in relation to maintaining its child table will consume the same amount of energy as a D-FTSP packet trans-

Symbol	Value
CPU_{AI}	0.0018 A
TTX_{AI}	0.0188 A
TX_{br}	250,000 bps
CPU_{clc}	8×10^9 Hz

Table 4.4: TelosB parameters (MPS430 Microcontroller, CC2420 Transceiver)

Symbol	Value
P_s	72 b
ST_{cyc}	28
UT_{cyc}	113
N_c	10

Table 4.5: D-FTSP parameters

mission if performed 872 times. To put this into perspective one can consider a scenario whereby all 10 child nodes above are configured with a very low transmission interval of 30s. In this particular case, it would still take over 40 minutes for their parent node to consume the same amount of energy required to transmit a D-FTSP packet. Thus, CPU energy consumption is minimal for this particular operation.

$$R_E = \frac{0.0018 \times (72 \times (4 \times 10^{-6}))}{0.0188 \times ((1.25 \times 10^{-10}) \times ((28 \times (\frac{10}{2})) + 113))} \approx 872 \tag{4.26}$$

In relation to the second additional code block employed by D-FTSP, which is responsible for updating the transmission interval (τ) of a node, the number of machine cycles required to perform a single iteration of algorithm 1 was calculated using the technique mentioned above and was found to be 147 cycles. Again, this represents an extremely small proportion of additional CPU energy utilisation. Only in the case of large numbers of iterations of the loop (order of

thousands) would CPU energy consumption begin to compare to that of a message transmission, and such a case should never occur. Thus, again CPU energy consumption is minimal.

4.2.7 Memory Analysis

Similar to energy, memory is a limited and valuable resource in WSNs. The *TelosB* motes used in the experiments have just 10KB of RAM, whereas the *Mica2* has 128KB. Since WSN time synchronisation protocols are employed to aid higher level time-sensitive applications, it is important that they consume a very small proportion of the available memory. Because of the design of D-FTSP, in order to achieve the aforementioned accuracy and energy savings, it must store more state. The greatest source of additional memory consumption relative to FTSP is the table employed to store data describing a node's children. The data stored for each child includes a 2 byte *node ID* value, a 4 byte *timestamp* value and a 2 byte *skew* value. Thus, for each child, a D-FTSP node must consume 8 bytes. The total size of the table is directly related to the number of children a D-FTSP node has. A node with N_c children consumes approximately $N_c \times 8$ additional bytes relative to an FTSP node. In a real world scenario, a value of N greater than 10 would represent an unusual case. Thus, one can say that the maximum additional memory consumed by a D-FTSP node relative to an FTSP node would unlikely be greater than 80 bytes. In the case of a *TelosB* mote with just $10,000$ bytes of RAM, this represents a very small proportion of additional memory.

4.3 Summary and Contribution

This chapter presented the *Dynamic Flooding Time Synchronisation Protocol (D-FTSP)* an extension to the *Flooding Time Synchronisation Protocol (FTSP)*. D-FTSP is designed for dynamic and unpredictable environments that subject WSN nodes to rapid and extensive clock drift. It transforms FTSP from a static synchronisation protocol to a dynamic one by providing nodes with the ability to identify their child nodes and determine their clock drift. Knowledge of a

child's clock drift allows a parent node to determine an appropriate transmission interval that prevents a child's clock offset from exceeding an application defined error bound. In a real WSN deployment, a key criterion is often the level of synchronisation required and, thus, D-FTSP maps much more effectively to this scenario.

D-FTSP's objective is efficient and accurate time synchronisation in dynamic environments. In order to determine D-FTSP's ability to meet its objectives, it is compared and contrasted with FTSP in both a stable and unstable environment. The results indicate that in an unstable environment, D-FTSP can achieve similar average network-wide accuracies to that of FTSP with almost 75% less transmissions. An analysis of energy consumption indicates that this translates to a 66% energy saving. The analysis, however, also indicates that D-FTSP may, in certain rare scenarios, consume more energy in a stable environment which suggests that it is more suited to dynamic environments.

In conclusion, the contributions of D-FTSP can be summarised as follows:

- It provides nodes with the capability to directly determine the magnitude of a node's clock drift.
- It provides nodes with the capability to automatically self-configure their transmission interval with respect to a pre-configured WSN application error bound. This eliminates the pre-deployment stage required to determine suitable synchronisation parameters for static time synchronisation protocols such as FTSP.
- In a dynamic/unstable environment, it is capable of achieving similar network-wide accuracies to that of FTSP with a reduced number of transmissions, thus, conserving the limited energy supplies of WSN nodes.

Part III

Time Synchronisation in WiFi Networks

may serve thousands of MMOG clients simultaneously. The links between these clients and the MMOG server may vary drastically, some incorporating wireless technologies and others high speed fibre technologies. Consequently, subsets of clients may experience significantly greater packet delays than others, severely reducing the response time and, hence, fairness of the game. The use of time synchronisation can go a long way towards alleviating this unfairness. Similar to the VoIP scenario, MMOG clients can be synchronised such that clients may transmit time-stamped packets. An MMOG server can use this temporal information to equalise the delays experienced by all clients and, thus, improve fairness.

These examples highlight only a subset of applications that necessitate time synchronisation over PSNs. It is interesting to note that the majority of such applications are distributed in nature. Since distributed applications may have distributed databases housing related data, the correct sequencing of events proves crucial in ensuring the integrity of this distributed data. Time synchronisation may also play a significant role in enhancing the quality of service of applications, particularly multimedia services. Temporal enhanced data provides invaluable information that can be used to balance the quality of a service across multiple clients.

5.2 PSN Time Synchronisation Protocols

The two most dominant time synchronisation protocols employed in PSNs are the *Network Time Protocol (NTP)* and the *Precision Time Protocol (PTP)* (IEEE 1588). Each protocol is designed for a particular type of PSN. While NTP is designed for large dynamic networks, PTP is catered for relatively stable and controlled local area networks (LANs). The following sections describe each protocol in detail.

5.3 Network Time Protocol

The *Network Time Protocol (NTP)*, by Mills [8, 13, 50, 51, 52], is one of the oldest internet protocols in operation today. It has been in operation for over twenty-five

years. It is designed to synchronise the clocks of nodes connected over dynamic packet-switched networks that subject packets to varying latencies. The internet represents the best example of such a network. Multiple paths may exist between two particular nodes and either of these paths may become congested, or fail, at any time. Thus, the route a stream of packets take from one node to another may not remain fixed and, therefore, the latency of packets can vary over time.

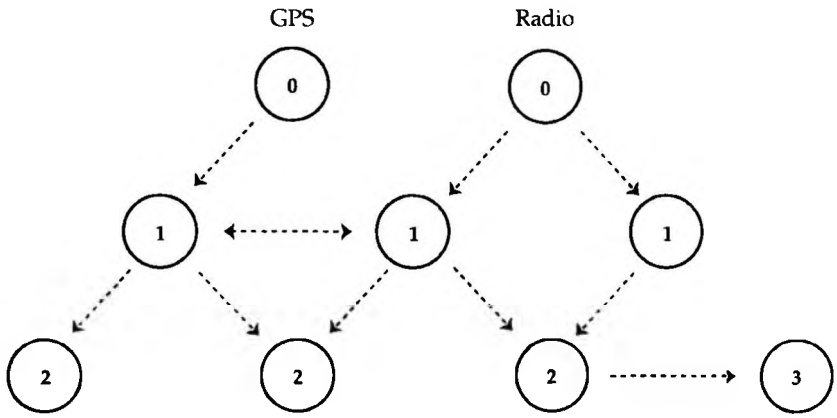


Figure 5.1: NTP hierarchy (stratums)

NTP uses the round-trip synchronisation technique described in section 2.4.2. Of course, since this technique assumes that the packet latencies between two nodes are symmetric, NTP's design incorporates techniques to mitigate the effect of asymmetric delays. The protocol uses redundant time references together with a suite of statistical algorithms to achieve this. The use of redundant time references increases the diversity of paths and, thus, increases the likelihood of acquiring high quality time data. In addition to this, the identification and elimination of low quality references is made possible. The statistical algorithms employed by NTP are used to filter the acquired data and choose the highest quality time references from which the final offset estimation is produced.

NTP hosts are organised into a hierarchical structure termed an NTP subnet (see fig. 5.1). Hosts at each level of the hierarchy are assigned a stratum number which represents their distance from the root of the hierarchy which itself has a stratum of zero. Stratum 0 references consist of extremely accurate time sources such as GPS, radio, and atomic clocks. Hosts that synchronise with stratum

0 references are classified as stratum 1 references, and those that synchronise with stratum 1 references are classified as stratum 2 references and so on. This structure, in addition to eliminating cyclical dependencies, provides a means of comparing the quality of references. For example, as detailed later, it is used by NTP's clustering algorithm to formulate a quality metric that is used to prune a list of references.

NTP's design and operation allows it to achieve millisecond accuracies within local area networks and less than 100 milliseconds in wide area networks. This is an impressive feat given the dynamic nature of the internet.

5.3.1 Packet Structure and Processing

The NTP protocol operates at the application layer of the OSI model. It uses the transport layer protocol UDP to distribute time. The NTP packet structure is illustrated in fig. 5.2. The interpretation of the packet fields varies depending on the operating mode of the NTP hosts that are interacting. The primary operating mode is *client/server* mode, and it represents the most intuitive one as it follows the classical *remote procedure call (RPC)* paradigm. The following description of the NTP fields relates to a NTP reply packet sent from an NTP server in response to a request from an NTP client (as illustrated in fig. 5.2) -

- *LI (Leap Indicator)* - Warns the client of an approaching leap second.
- *VN (Version Number)* - Specifies the version of NTP in use.
- *Mode* - Indicates the operating mode of the server.
- *Stratum* - Indicates the stratum number of the server.
- *Poll Interval* - Indicates the maximum interval in seconds between successive messages from the server to the client.
- *Precision* - Indicates the precision of the server's local clock in seconds.
- *Root Delay* - Indicates the total round-trip delay in seconds from the server to its primary reference. The primary reference represents the NTP host that the server has chosen as its preferred time reference.

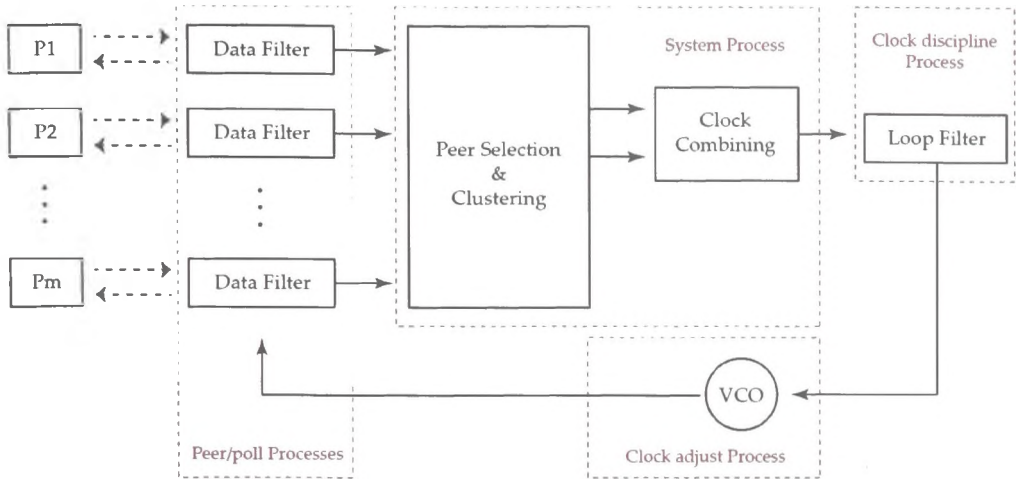


Figure 5.3: NTP processes and flow

packet contents that result in offset and round-trip delay estimates, collectively termed state variables. A peer and associated poll process are established for each peer. These processes together with their associated state variables form what is termed an *association*. An association is classified according to the duration of its lifetime which in turn is dependent on the operational mode of the peer and host. An association that exists indefinitely is classified as a *persistent* one, whereas one that exist briefly is classified as either *preemptable* or *ephemeral*. To clarify, in the case of a host and peer operating in *client/server mode*, the client establishes a persistent association for the server since it must continually poll it in order to stay synchronised. In contrast, the server, on receipt of an NTP request packet, generates an ephemeral association that only exists until the request is fulfilled. This is so because a server does not need to maintain state regarding the condition of a client.

The *system process* and its associated *selection*, *clustering* and *combining* algorithms are responsible for pruning the data passed to it from one or more peer process(es). The selection algorithm's core responsibility is the identification and elimination of data associated with erroneous peers, also termed *falsetickers*. The clustering algorithm prunes the remaining data further by identify the most accurate data and eliminating the remainder. If at this point data associated with

more than one peer remains then the final estimate of a host's offset is calculated using the combining algorithm. The final offset estimation is then passed to the *clock discipline process* which together with the *clock adjust process* adjust the clock in small appropriate increments insuring a continuous monotonic clock, that is, a non-decreasing clock.

5.3.3 Data Filter Algorithm

The data filter algorithm represents a core component of the peer process. Its primary function is the identification and selection of an accurate offset and RTD pair sample (δ, θ) from a list of recently collected samples associated with a particular peer.

The data filter algorithm's design is based on the characteristics of typical PSNs. Such networks are designed to alleviate packet congestion through the use of extra resources and sophisticated routing algorithms. Thus, while it is unlikely that a packet experiences significant delay in one direction, it is very unlikely that it also experiences a significant delay in the opposite direction. This can be verified by plotting a RTD (δ) vs offset (θ) scattergram for a particular peer. When the RTD (δ) and offset (θ) samples associated with a particular peer are plotted over time they form a scattergram, the shape of which highlights the characteristics of the path between a host and a particular peer. In general it resembles a *wedge*, as illustrated in fig. 5.4, and is termed a *wedge scattergram.*

Referring to fig. 5.4, one may observe that the majority of points are located at the top, bottom, and apex of the wedge which highlights the aforementioned characteristics of PSNs. Those points located at the bottom of the wedge indicate that the associated NTP request packets are subjected to greater delays than the corresponding reply packets. Correspondingly, those points located at the top of the wedge indicate that the associated NTP reply packets are subjected to greater delays than the corresponding request packets. Consequently, those points located at the apex of the wedge scattergram represent the most accurate estimates of the host's clock offset as they are associated with symmetric network delays. Those points are also associated with the lowest RTDs which suggest that when presented with a sequence of RTD and offset pair samples (δ, θ), the pair

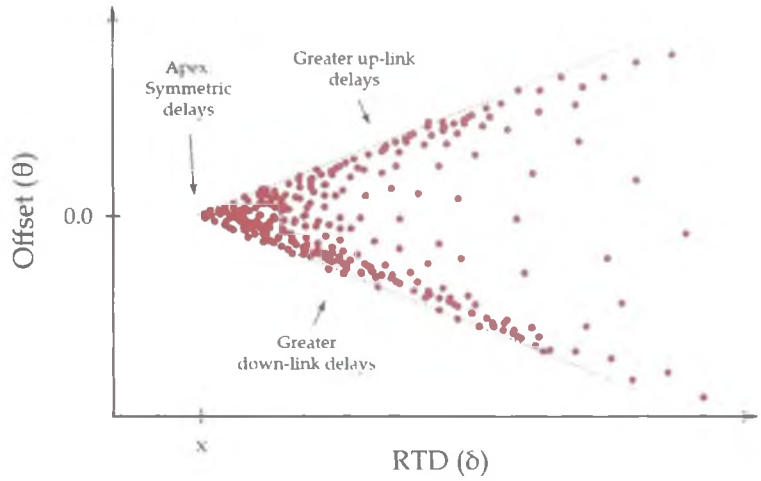


Figure 5.4: NTP wedge scattergram

with lowest RTD should be utilised as it represents the most accurate data.

Accordingly, the data filter algorithm, the core of which is illustrated in fig. 5.5, is employed by the peer process. The algorithm in fig. 5.5 is presented in relation to a peer and poll process associated with a peer m. The poll process generates an NTP request packet and sends it to m which responds with an NTP reply message, as illustrated in fig. 5.2. The timestamps $T_{(i+n)}$, $T_{(i+n)+1}$, and $T_{(i+n)+2}$ contained within the reply message together with the timestamp $T_{(i+n)+3}$ generated by the host are used to create the $(i+n)th$ sample $(\delta_{i+n}, \theta_{i+n}, \varepsilon_{i+n})$ which is placed into a shift register. The variable ε represents an important quality metric termed the *peer dispersion* that is used by a subsequent NTP algorithm. It is initialised with the resolution (ρ) of the host's clock and then increased at a constant rate of ϕ $(15ppm)$.

The shift register, which holds n samples in reverse chronological order, drops the oldest sample, $(\delta_{i-1}, \theta_{i-1}, \varepsilon_{i-1})$, upon receipt of a new sample. The n samples are then placed into a temporary list which is sorted by δ. The offset, θ_0, of the sample with the lowest value of δ is used together with the remaining offset values to calculate the *peer jitter* (φ) of m. The peer jitter represents another important quality metric that is subsequently used by the clustering algorithm to determine

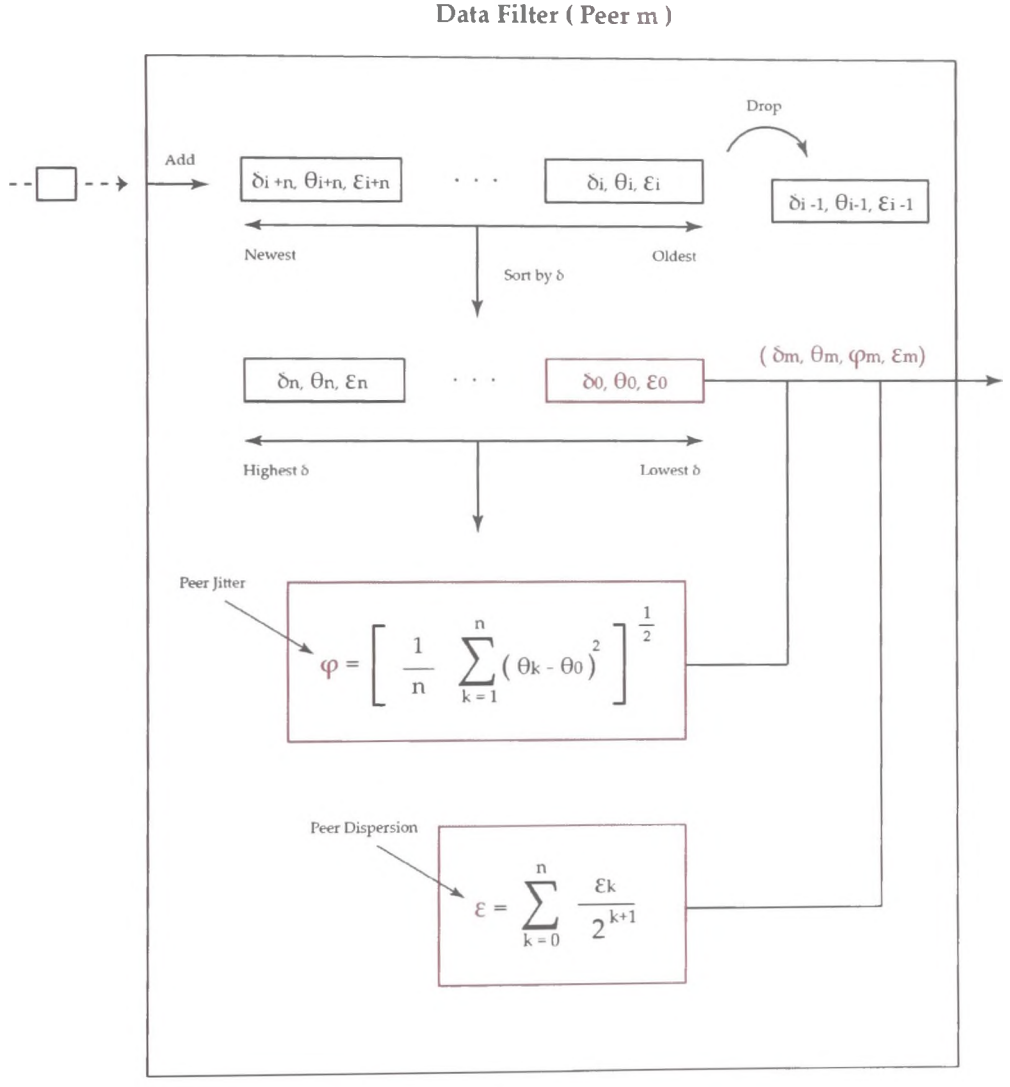


Figure 5.5: NTP data filter algorithm

the most accurate offset value associated with a group of m peers. Its equation resembles that of the *root mean square (RMS)* formula and, therefore, its value represents the magnitude of variation of a particular quantity, in this case, the magnitude of variation of each of the n previous values of θ associated with peer m.

Finally, the peer dispersion (ϵ) is calculated and the final output of the algorithm is a tuple of the form $(\delta_m, \theta_m, \varphi_m, \varepsilon_m)$ which corresponds to the lowest RTD sample, the jitter, and dispersion for peer m respectively. These variables are generated by each of the m peer processes associated with each of the m peers. They are subsequently referred to as *peer variables* and used by succeeding NTP algorithms.

5.3.4 Selection Algorithm

The peer/pool processes associated with each host's peer provide the system process with the peer variables produced by the data filter algorithm. Thus, the system process receives a tuple of the form $(\delta_i, \theta_i, \varphi_i, \varepsilon_i)$ for each of the m peers. Subsequent NTP algorithms are used to identify the highest quality data from amongst this collection. The selection algorithm begins this process by using the collection of peer variables to identify inaccurate peers.

A properly configured NTP client will employ numerous references/servers located across distinct network paths in order to synchronise its clock. The use of redundant references is beneficial in that data that originates from one or more incorrect server/s can be more easily identified assuming the majority of references are correct. The selection algorithm is responsible for separating incorrect peers, termed *falsetickers*, from correct peers, termed *truechimers*. It achieves this by first constructing confidence intervals for each of the m peers (see fig. 5.6). Subsequently it determines the smallest intersection interval within which at least $m - f$ of the peers' associated offsets lie.

The confidence interval of a peer represents a range of values within which the true offset associated with a peer must be located. In relation to a peer i, this interval is centred about the peer's associated offset estimate, θ_i, and is equal in magnitude to twice the value of the *root distance*, Λ_i, associated with that

peer. This interval is represented by the expression $[\theta_i - \Lambda_i, \theta_i + \Lambda_i]$. The root distance ($\Lambda$) associated with a peer represents the maximum time error of that peer relative to the root of the hierarchy. It is calculated using the *root dispersion* (E) and *root delay* (Δ) which are retrieved from the NTP reply packet associated with the peer (see fig. 5.2). The root distance for a particular peer i is calculated using equation 5.3.

$$\Lambda_i = \frac{\Delta_i}{2} + E_i \tag{5.3}$$

Before describing the selection algorithm in detail, it is first important to understand the rationale behind the construction of a peer's confidence interval. This is described in the subsequent section.

5.3.4.1 Confidence Interval Construction

Equation 5.3 stems from an analysis of the time error that can accumulate when two NTP nodes synchronise with each other. This error can be divided into two core components, the first of which is associated with timestamps and the second of which is associated with network delays.

Timestamp errors can be divided into *clock reading errors* and *clock frequency errors*. As explained in section 2.2, a clock consists of an oscillator that operates at a particular frequency f. Each oscillation increments a counter at intervals of $\rho = 1/f$ which represents the resolution of the clock. A timestamp, $T(t)$, produced by reading the clock at time t will have a clock reading error represented by a random variable x bounded by the interval $[-\rho, 0]$. An oscillator will also have a fractional frequency error (see section 2.2.1) which may change slowly over time. The NTP *clock discipline algorithm* ensures that the frequency error represented by the random variable y is bound by the interval $[-\phi, \phi]$ where ϕ represents the maximum frequency offset. Thus, the clock error τ seconds after a clock reading can be represented by the random variable z expressed in equation 5.4. The *probability density function (pdf)* of z resembles a bell-shaped curve centred at $\rho/2$ and grows in width with τ.

$$z = x + y\tau. \tag{5.4}$$

The fact that x and y are bounded allows one to determine the error bounds associated with timestamps. Referring to the scenario in fig. 5.2 where an NTP client, A, attempts to synchronise with an NTP server, B, by obtaining the timestamps T_i, T_{i+1}, T_{i+2}, and T_{i+3}, one can represent the *frequency error*, *frequency error bound*, and *clock reading error bound* of A and B as $(f_A, [-\phi_A, \phi_A], [-\rho_A, 0])$ and $(f_B, [-\phi_B, \phi_B], [-\rho_B, 0])$ respectively. Thus, the error bounds associated with each of the timestamps T_i, T_{i+1}, T_{i+2}, and T_{i+3} can be represented by equations 5.5, 5.6, 5.7 and 5.8 respectively.

$$\epsilon_1 = [-\rho_A, 0] \tag{5.5}$$

$$\epsilon_2 = [-\rho_B, 0] \tag{5.6}$$

$$\epsilon_3 = [-\rho_B, 0] + f_B(T_{i+2} - T_{i+1}) \tag{5.7}$$

$$\epsilon_4 = [-\rho_A, 0] + f_A(T_{i+3} - T_i) \tag{5.8}$$

Consequently, with respect to equations 5.2 and 5.1, the error inherent in determining the offset (θ) and RTD (δ) of A are contained within the bounds in equations 5.9 and 5.10 respectively. (NOTE: The expression $[i, j]+a$ is equivalent to $[i + a, j + a]$).

$$\{[-\rho_A - \rho_B, \rho_A + \rho_B] + f_B(T_{i+2} - T_{i+1}) - f_A(T_{i+3} - T_i)\}/2 \tag{5.9}$$

$$[-\rho_A - \rho_B, \rho_A + \rho_B] + f_B(T_{i+2} - T_{i+1}) + f_A(T_{i+3} - T_i) \tag{5.10}$$

To simplify the process of calculating the maximum absolute error inherent in determining θ and δ, the values ρ, ϕ, and T are set according to equations 5.11, 5.12, and 5.13 respectively. Thus, A and B's clocks are assumed to have the same resolution, ρ, and the frequency error of each clock is considered to be bound by the interval $[-\phi, \phi]$ due to the work of the *clock discipline algorithm*. These are assumptions but are adequate to determine the maximum errors.

$$\rho = max(\rho_A . \rho_B) \tag{5.11}$$

$$\phi = max(\phi_A, \phi_B) \tag{5.12}$$

$$T = T_{i+3} - T_i \tag{5.13}$$

The maximum absolute error in determining θ termed the *offset dispersion* (ε_θ) is then expressed using equation 5.14, and the maximum absolute error in determining δ termed the *delay dispersion* (ε_δ) is expressed using equation 5.15.

$$\varepsilon_\theta = \rho + \phi T \tag{5.14}$$

$$\varepsilon_\delta = 2(\rho + \phi T) \tag{5.15}$$

These maximum errors are associated with the timestamps. While they are important, they are typically overshadowed by network delay errors. In relation to the timestamps recorded by client A and server B, the upload delay (u) and download delay (d) of an NTP packet are $u = (T_{i+1} - T_i)$ and $d = (T_{i+3} - T_{i+2})$ respectively. Thus, θ and δ can be represented by equation 5.16 and 5.17

$$\theta = \frac{(u - d)}{2} \tag{5.16}$$

$$\delta = u + d \tag{5.17}$$

If θ_T represents the true offset of A relative to B, then the inequality $d \leq \theta_T \leq u$ must hold, since message delays are always positive. This inequality can also be expressed using equation 5.18 which from equations 5.16 and 5.17 is equivalent to equation 5.19

$$\frac{u + d}{2} - \frac{u - d}{2} \leq \theta_T \leq \frac{u + d}{2} + \frac{u - d}{2} \tag{5.18}$$

$$|\theta_T| \leq |\theta| + \frac{\delta}{2} \tag{5.19}$$

Equation 5.19 provides bounds for the error related to network delay, but the offset and delay dispersion presented in equations 5.14 and 5.15 must be factored in. When factored in, they result in equation 5.20 which can also be expressed using equation 5.21.

$$|\theta_T| \leq |\theta| + \varepsilon_\theta + \frac{\delta + \varepsilon_\delta}{2} \tag{5.20}$$

$$|\theta_T| \leq |\theta| + \frac{\delta}{2} + 2(\rho + \phi T) \tag{5.21}$$

The term $2(\rho + \phi T)$ represents the *dispersion*, ε, which was presented in section 5.3.3. Combining the network delay error and the dispersion results in a quantity termed the *synchronisation distance* (λ) (see equation 5.22).

$$\lambda = \frac{\delta}{2} + \varepsilon \tag{5.22}$$

The term λ represents the synchronisation distance between A and B. The synchronisation distance between B and its primary reference is represented by the *root distance*, Λ_R, which is expressed in equation 5.3 and is calculated using the *root delay* (Δ_R) and *root dispersion* (E_R). Client A obtains these quantities via an NTP reply message from B. Before A can provide synchronisation, it must recalculate these values. The calculation of Δ and E at A are performed using equations 5.23 and 5.24 respectively. In equation 5.24, μ represents the update interval and Θ represents the final offset value produced by the *clock combining algorithm* (described later).

$$\Delta = \Delta_R + \delta \tag{5.23}$$

$$E = E_R + \varepsilon + \phi\mu + \varphi + |\Theta| \tag{5.24}$$

5.3.4.2 Algorithm

The rationale behind the construction of a peer's confidence interval justifies the design of the selection algorithm. Since the confidence interval bounds the true offset of a host relative to a peer, the confidence interval of a particular peer that does not intersect with the majority of the intervals associated with the remaining peers is most likely a falseticker.

As explained earlier, the confidence interval of a peer i can be expressed as $[\theta_i - \Lambda_i, \theta_i + \Lambda_i]$ where θ_i represents the midpoint of the interval. In the case of m peers where up to f falsetickers are permitted and $f < m/2$, the NTP selection algorithm identifies the smallest intersection of $m - f$ confidence intervals that contains at least $m - f$ midpoints. This intersection interval is bounded by the lower limit, l, and the upper limit, u, and can be represented by the expression $[l, u]$.

This is illustrated in fig. 5.6 which depicts the confidence intervals associated with four peers. The selection algorithm produces an intersection interval which contains the midpoints θ_0, θ_1, and θ_2. The data associated with peer 3 lies outside this interval, and so the peer is considered a falseticker and removed from the group.

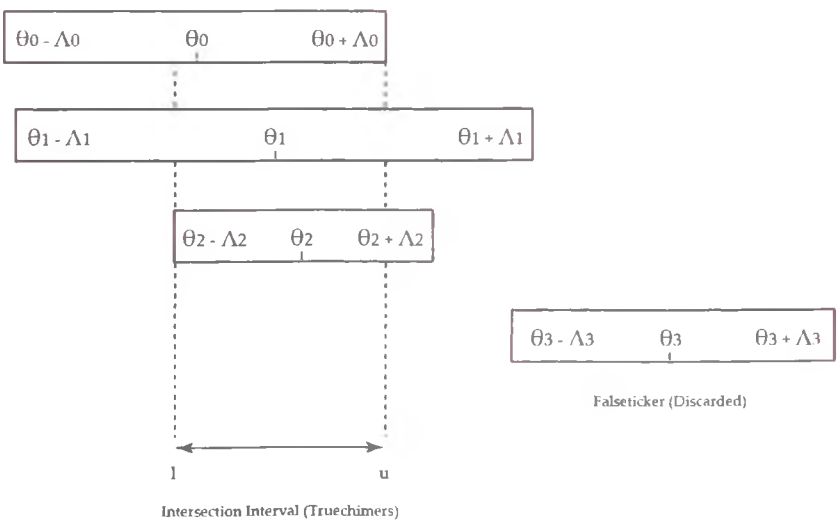


Figure 5.6: NTP selection algorithm

5.3.5 Clustering Algorithm

The selection algorithm is responsible for removing those peers that disagree with the majority and does this effectively. Nevertheless, the remaining peers and their associated offsets typically do not agree due to minor asymmetric delays. Thus, the most accurate peer among the remaining must be identified. This is the responsibility of the clustering algorithm.

In order for the clustering algorithm to identify this particular peer, it must use appropriate quality metrics. The first of these metrics is the *peer jitter* (φ) which is produced by the data filter algorithm presented in section 5.3.3. In relation to a particular peer, it represents the RMS of the differences between each of the offset samples associated with that peer and the most accurate offset sample. In essence, it represents the magnitude of variation of the offset samples associated with a particular peer and, therefore, is a good measure of the quality of data associated with that peer.

The second metric utilised is the *selection jitter* (φ_s) which represents the RMS of the differences between a particular peer's associated offset and the remaining peers' associated offsets. Thus, in the case of m remaining peers, the selection jitter for a peer i is expressed in equation 5.25.

$$\varphi_{s,i} = \left[\sum_{j=0}^{m-1} (\theta_i - \theta_j)^2 \right]^{\frac{1}{2}} \tag{5.25}$$

The final metrics employed are the *stratum* (s) and *root distance* (Λ) of the peer. These metrics are used to produce a sort metric denoted by the symbol λ. The sort metric for a particular peer i is calculated using equation 5.26 where Λ_{max} represents a bias factor termed the *maximum distance*.

$$\lambda_i = \Lambda_{max} s_i + \Lambda_i \tag{5.26}$$

Equation 5.26 is designed to give preference to those peers with lower stratum numbers. This is logical, since those peers with lower stratum numbers are typically more accurate than those with higher ones given the structure of the NTP hierarchy. Of course, this is not always the case, and so the root distance, Λ, of a peer is also utilised to handle such scenarios. Thus, in the case of two peers i

and j with stratum numbers 1 and 2 respectively, peer j will not precede peer i unless i's root distance is greater than the maximum distance, Λ_{max}, and j's root distance, Λ_j, combined.

The clustering algorithm commences by first constructing a list with tuples of the form $(\lambda_i, \theta_i, \varphi_i)$ for each of the m peers. It then sorts this list by the sort metric λ. On completion of the algorithm, this order represents their rank as time references. The value n_{min} is chosen to represent the minimum number of peers that must remain. Algorithm 2 is then employed. The first part of this algorithm calculates the selection jitter, φ_s, for each of the m peers. The selection jitter for a particular peer i is then multiplied by its associated sort metric λ_i to produce the statistic x_i. If the minimum number of peers has not been reached and the maximum selection jitter of all peers, $\varphi_{s,max}$ is greater than the minimum jitter of all peers φ_{min}, then the peer associated with the highest statistic x is removed, and the procedure repeats.

Algorithm 2 Selection Algorithm

loop
  for $i = 1 \rightarrow m$ **do**
    $sum \leftarrow 0$
    for $j = 1 \rightarrow m$ **do**
      $sum \leftarrow sum + (\theta_i - \theta_j)^2$
    end for
    $\varphi_{s,i} \leftarrow \sqrt{sum}$
    $x_i \leftarrow \lambda_i \varphi_{s,i}$
  end for
  if $m \leq n_{min}$ **then**
    exit loop
  else if $max(\varphi_{s,i}) \leq min(\varphi_j), (0 \leq i \leq m, 0 \leq j \leq m)$ **then**
    exit loop
  else
    remove peer with maximum x
    $m \leftarrow m - 1$
  end if
end loop

An example of the procedure is illustrated in fig. 5.7. Here four peers $1-4$ are presented along with their jitter and selection jitter which are represented by the width of their associated boxes. The first iteration, illustrated on the left of fig. 5.7, identifies the minimum jitter as φ_4 and the maximum selection jitter as $\varphi_{s,2}$ which are associated with peers 4 and 2 respectively. Because the minimum jitter is less than the maximum selection jitter, the peer with the maximum statistic x, here assumed to be peer 2, is removed. The second iteration identifies the minimum jitter as φ_4 and the maximum selection jitter as $\varphi_{s,3}$ which are associated with peers 4 and 3 respectively. This time the minimum jitter is greater than the maximum selection jitter leaving three survivors: peers 1, 3, and 4. The algorithm completes by ranking these peers based on the sort metric λ.

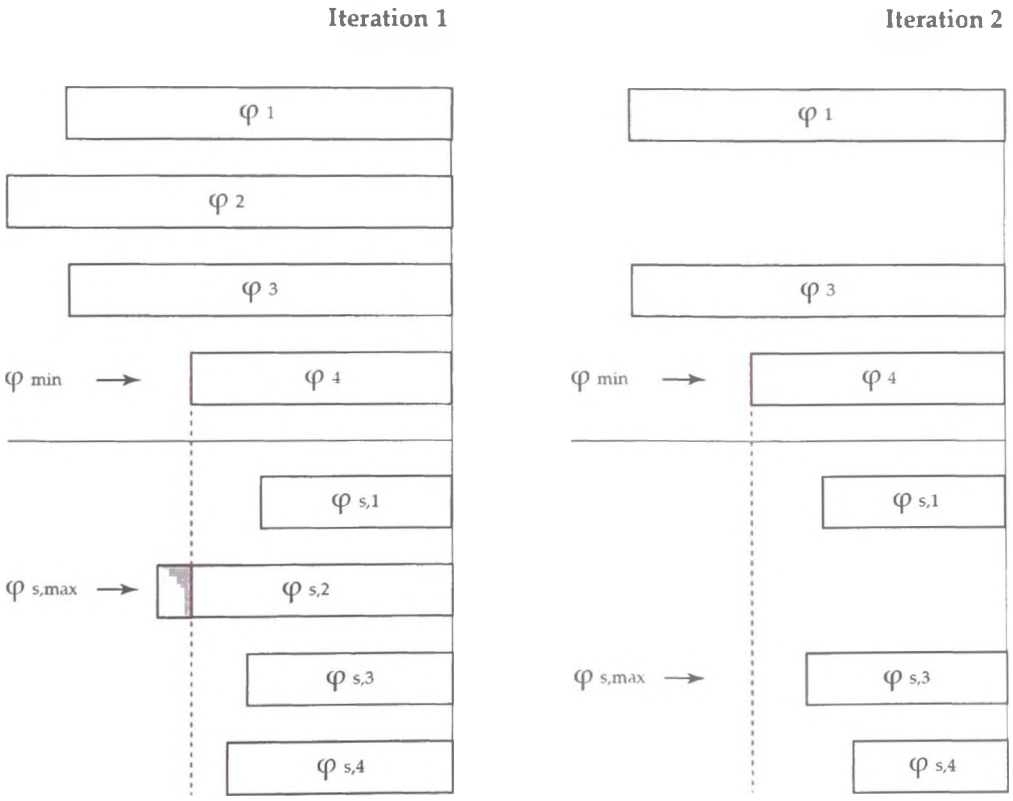


Figure 5.7: NTP clustering algorithm

5.3.6 Combining Algorithm

The output of the selection and clustering algorithm is a set of data associated with at least one peer. In the case of a single surviving peer, i, the variables associated with i will be used to update the core system variables Θ, Δ, and E. The *system offset*, Θ, will be set to the offset associated with i, that is, θ_i, while the *root delay*, Δ, and *root dispersion*, E, will be set according to equations 5.23 and 5.24 respectively.

In the case of multiple surviving peers, the *combining algorithm* is employed. The combining algorithm is based on an observation that it is possible to obtain a more accurate estimate of a host's offset if the offset of multiple surviving peers are averaged according to a suitable weighing scheme. This weighing scheme is based on the *root distance* (Λ) of a peer whereby those peers with lower values of Λ contribute more to the final estimation of Θ. This is expressed in equation 5.27.

$$\Theta = \left(\frac{1}{\sum_i \frac{1}{\lambda_i}} \right) \sum_i \frac{\theta_i}{\lambda_i} \tag{5.27}$$

The system variable, Θ, is subsequently used by the clock discipline algorithm which employs a *phase-locked loop* to correct the host's clock offset at an appropriate rate.

5.3.7 Clock Discipline Algorithm

The clock discipline algorithm forms the heart of the clock discipline process. It operates as a feedback control system that uses the offset value produced by preceding algorithms to calculate phase and frequency corrections which are subsequently used to control a *variable frequency oscillator (VFO)*.

The operation of the NTP algorithms in terms of a feedback control loop is illustrated in fig. 5.8. Here θ_c represents the control phase of the VFO and θ_r represents the phase of a reference server. The phase detector produces the signal V_d which represents the instantaneous phase difference. The output of one or more clock filters are processed by the intersection, clustering, and combining algorithms to produce the signal V_s. This is used by the loop filter to produce

phase and frequency corrections which are used by the clock adjust process to produce the signal V_c which controls the frequency of the VFO, thus, completing the loop.

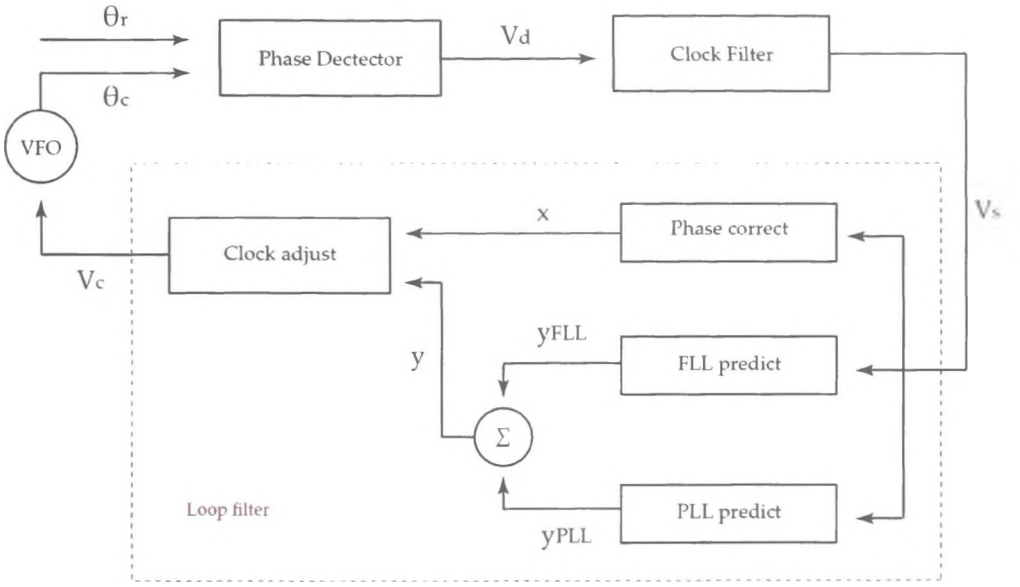


Figure 5.8: NTP clock discipline algorithm

The loop filter is responsible for producing both phase and frequency adjustments from the signal V_s. These are represented by the terms x and y respectively. The frequency adjustment, y, is produced using a combination of a *phase-locked loop (PLL)* and *frequency-locked loop (FLL)*. A PLL differs from an FLL in that it is used to directly minimise time error while indirectly minimising the frequency error. Conversely, an FLL directly minimises the frequency error while indirectly minimising the time error. Mills [13] shows that a PLL works better when system jitter dominates, whereas an FLL works better when oscillator wander dominates. By combining both techniques with the proper weightings, it is observed that better frequency adjustments can be obtained. The discipline algorithm, thus, employs a PLL/FLL hybrid approach.

Within the loop filter, the phase and frequency adjustments x and y are set according to equations 5.28 and 5.29. Thereafter, at intervals of one second,

the clock adjust process uses the adjustments to compute an appropriate phase increment which is then passed to a kernel function. This function, typically *adjtime()*, directly updates the system time. This continues until x and y are recomputed at the next update.

$$x = V_s \tag{5.28}$$

$$y = y + y_{PLL} + y_{FLL} \tag{5.29}$$

5.4 Precision Time Protocol

The network time protocol is designed to meet the accuracy requirements of distributed applications that are connected over wide area networks. NTP provides a sufficient degree of synchronisation to meet the requirements of a majority of these applications. When accuracies as low as 1 millisecond are required, NTP may also be employed but, ideally, the link that connects an NTP server to the devices that require synchronisation should not introduce delays that might negatively impact the protocol's performance. Thus, the link should be a managed one.

There are other applications that require synchronisation levels far beyond what NTP can deliver. Measurement, control, and operational applications utilised by manufacturing, utility, and telecommunication systems may require numerous devices to be synchronised within a few microseconds of each other in order to function properly. To provide such a degree of synchronisation requires the use of GPS receivers, and when this proves impractical, atomic clocks are the only remaining option. Of course, such approaches are extremely expensive. To meet an obvious demand, the Institute of Electrical and Electronics Engineers (IEEE) published the *IEEE 1588* standard in 2002. The IEEE 1588 standard, otherwise known as the *Precision Time Protocol (PTP)*, is designed to fill the niche that alternative synchronisation protocols cannot by providing a feasible means of time synchronising numerous interconnected computing devices within microseconds of each other.

5.4.1 Overview

PTP [9, 53, 54] is designed for systems that require microsecond to sub-microsecond accuracies. Such systems can be found in industrial, utility and communication sectors. Its design allows for administration free operation over spatially localised networks and can accommodate both high and low end devices.

In contrast to NTP and with regard to the accuracies it is designed to achieve, PTP makes some core assumptions about the state of the network it operates over. The first of these is associated with asymmetry. Similar to NTP, the protocol, at its core, employs a variant of the round-trip synchronisation technique and, thus, asymmetric two-way packet delays can impact its performance. Thus, PTP expects that the underlying network is managed and, thus, assumes that the network and its components are selected and configured to minimise asymmetry. In addition to this, PTP expects that the traffic patterns on the network are controlled so that traffic variation is minimised and, consequently, timing jitter. Ideally, the network should be configured such that PTP messages are prioritised and, in addition, where possible, internetworking devices should be replaced by PTP aware devices such as *transparent clocks* and *boundary clocks*. Finally, although it can tolerate them, PTP assumes that events associated with missed, duplicated, or out-of-order messages are rare.

The PTP protocol itself is a distributed one that organises nodes into a *master-slave hierarchy*, the root of which is termed the *grandmaster clock*. The grandmaster clock acts as the time reference for every clock within a PTP domain. The network elements that participate in the PTP synchronisation process are categorised into one of five PTP device types. Each PTP device type implements various features of the protocol. The role that a network element plays within a network typically dictates the type of PTP device it becomes within the PTP synchronisation hierarchy.

The devices in a PTP system interface with the network via ports. When the protocol constructs the synchronisation hierarchy, it places each port on each PTP device into one of three primary states: *slave*, *master* or *passive*. Thus, although a device typically contains a single physical clock, it is not the device itself that enters the slave, master, or passive state but its ports, each of which is

associated with a distinct *protocol engine* and a dataset that describes it. During the synchronisation process, a slave port determines its clock offset with respect to its associated master port using two-way PTP message exchanges.

5.4.2 PTP Synchronisation

Each slave port in a PTP device communicates with a master port on another PTP device via an independent communication link. This communication link may or may not contain other network elements. If it does, then these elements may introduce varying message latencies. If these elements are *"PTP aware"*, then they possess specialised PTP hardware/software that allows them to correct for any message latencies they are responsible for.

The goal of a port in the slave state is to determine the propagation delay of PTP messages that traverse the link between it and its associated master port. Knowledge of this propagation delay permits the port to determine the offset between its local clock and its master's local clock. To accomplish this, the standard describes two mechanisms that can be used, namely the *delay request-response mechanism* and the *peer delay mechanism*. Devices that employ the delay request-response mechanism use a variant of the round-trip synchronisation technique (described in section 2.4.2) in order to acquire the four timestamps $(T_1 - T_4)$ necessary to calculate their clock offset. Devices that employ the peer delay mechanism use a separate sequence of PTP messages in order to determine one-way link delays at each port. Using this mechanism, timestamps recorded at either end of the link can be corrected using the known link delay, thus, eliminating the need for a slave port to acquire all four timestamps $(T_1 - T_4)$.

5.4.2.1 PTP Messages

PTP messages can be categorised into two groups: PTP *event messages* and PTP *general messages*. The distinction between both is that PTP event messages generate timestamps when transmitted and received, whereas PTP general messages do not. Thus, the purpose of PTP event messages is to capture time information, whereas the purpose of PTP general messages is to communicate information.

The class of event messages include -

- *Sync*
- *Delay_Req*
- *Pdelay_Req*
- *Pdelay_Resp*

The class of general messages include:

- *Announce*
- *Follow_Up*
- *Delay_Resp*
- *Pdelay_Resp_Follow_Up*
- *Management*
- *Signaling*

To accommodate these message classes, every PTP port possess two logical interfaces: the *event interface* and the *general interface*. In contrast to the general interface, messages that are transmitted via the event interface trigger a reading of the local clock both at transmission and receipt. Depending on the type of event message and the hardware employed, the generated timestamp may or may not be placed into the outgoing message (see next section).

The generation of a timestamp occurs when a specific point in an event message crosses a particular boundary in one of the communication layers. This point is termed the *timestamp point*. The standard itself specifies a timestamp point in an event message for each of the communication layers. Thus, on transit through the protocol stack, a detection of the timestamp point at the elected layer triggers a reading of the local clock. Of course, the higher the elected layer is in the protocol stack, the less precise the recorded timestamp typically is due to various system latencies encountered during the clock reading process.

In order to achieve the maximum precision, timestamps should be recorded at the physical layer. In such cases, specialised PTP hardware is required which

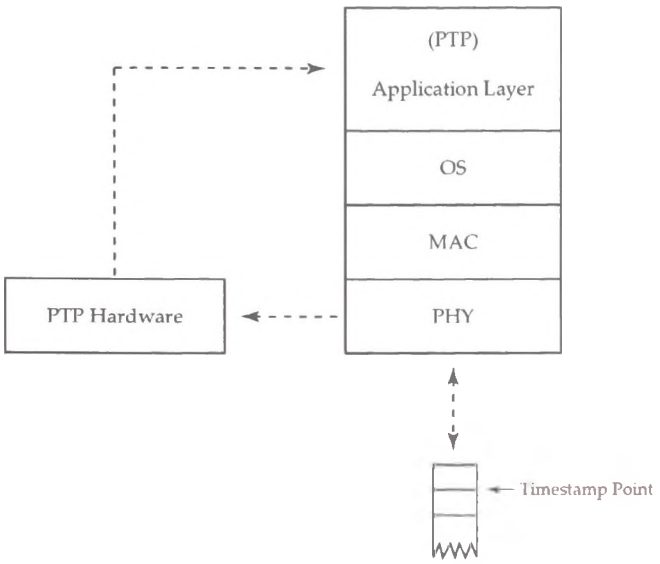


Figure 5.9: PTP time stamping

functions independently of the protocol stack (see fig. 5.9). This hardware, when it detects the timestamp point in the physical header, reads the local clock directly and, subsequently, passes the recorded timestamp via a direct and independent path to the protocol engine which runs at the application layer. Thus, all sources of non-deterministic system latencies are avoided.

5.4.2.2 Link Propagation Delay

PTP devices that employ the peer delay mechanism can determine the propagation delay of its ports' associated communication links where a communication link represents a connection between two distinct PTP ports. Every port on a PTP device that employs the mechanism uses it to determine a delay measurement for its associated link. Ports which share the same link, such as a master port on one device and a slave port on another, use the mechanism to calculate their own independent value for the link propagation delay.

The mechanism itself employs the *Pdelay_Req*, *Pdelay_Resp*, and if required, *Pdelay_Resp_Follow_Up* messages. The *Pdelay_Req* and *Pdelay_Resp* event messages generate the necessary timestamps, whereas the *Pdelay_Resp_Follow_Up*

message is used to communicate the timestamps.

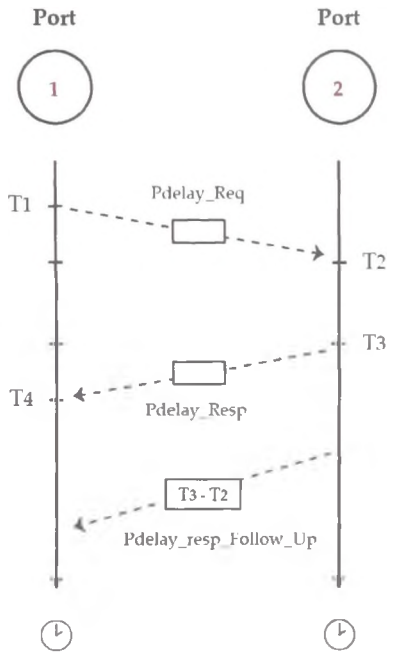


Figure 5.10: PTP peer delay mechanism

The link delay measurement process is illustrated in fig. 5.10. A port that wishes to determine the link propagation delay between it and another port that shares the same link, termed its *link peer*, initiates the communication exchange by transmitting a *Pdelay_Req* event message via the event interface. In fig. 5.10 port 1 and port 2 represent two ports that share a link. Port 1 wishes to determine the link delay between it and port 2 and, thus, initiates the communication exchange. This transmission triggers the generation of a timestamp denoted T_1 which represents the transmission time at port 1. The destination port, port 2, on receipt of the *Pdelay_Req* message via the event interface generates the timestamp T_2 which represents the reception time of the *Pdelay_Req* message. In response, port 2 transmits a *Pdelay_Resp* event message back to port 1 which generates the transmission timestamp T_3 at port 2 and the reception timestamp T_4 at port 1.

In the particular case depicted in fig. 5.10, at this stage in the process, port 1

only has access to timestamps T_1 and T_4. It must acquire either T_2 and T_3, or the difference between T_2 and T_3 in order to calculate the link delay. The acquisition of this information can be accomplished in one of three ways:

- The first approach, which is illustrated in fig. 5.10, has port 2 communicate the difference between T_2 and T_3 back to port 1 in a *Pdelay_Resp_Follow_Up* message.
- The second approach has port 2 communicate the difference between T_2 and T_3 back to port 1 in the *Pdelay_Resp* event message.
- The third approach has port 2 return timestamp T_2 in the *Pdelay_Resp* event message and timestamp T_3 in a *Pdelay_Resp_Follow_Up message.*

Once port 1 has acquired all four timestamps it can calculate the link delay and, subsequently, correct timestamps acquired using the synchronisation process (next section).

It must be noted that this mechanism assumes symmetric two-way delays. Any asymmetry will produce an incorrect estimation of the link delay. As mentioned earlier, a communication link between a master and slave can have various network elements that may introduce asymmetry. These elements, however, can be configured to proactively eliminate such asymmetry, as described in subsequent sections.

In addition to errors resulting from asymmetry, other less severe errors can be introduced if the frequency of port 1's associated clock differs from that of port 2's associated clock. In order to reduce such errors, port 2 should respond to a *Pdelay_Req* event message with a corresponding *Pdelay_Resp* message as quickly as possible after its reception.

5.4.2.3 Synchronisation Process

The general PTP synchronisation process is illustrated in fig. 5.11. The process itself employs the *Sync*, *Delay_Req*, *Delay_Resp*, and if required, *Follow_Up* messages. Each port in a PTP device uses the process independently of other ports and, thus, is responsible for constructing and communicating its own PTP messages. *Sync* and *Delay_Req* event messages are used to generate the necessary

timestamps required for a port in the slave state to determine its clock offset. *Delay_Resp* and *Follow_Up* general messages are used to communicate timestamps recorded by a master port back to a slave port.

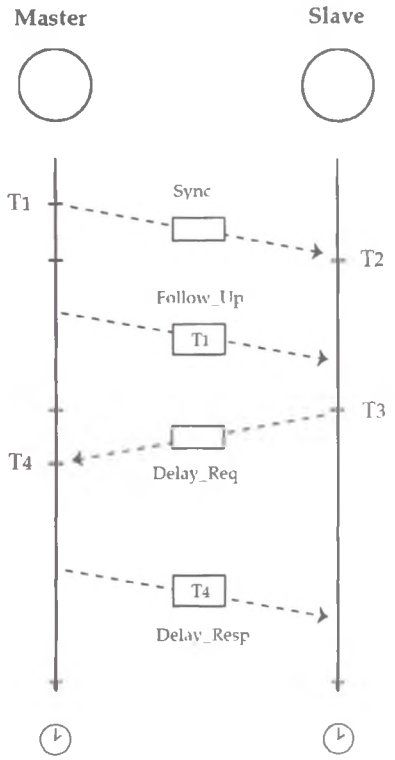


Figure 5.11: PTP synchronisation process

The synchronisation process is illustrated in fig. 5.11. The master port initiates the synchronisation process by transmitting a *Sync* message to the slave. This message is transmitted via the event interface which generates the timestamp T_1. The *Sync* message is received via the event interface at the slave triggering the generation of the timestamp T_2. The slave must have access to T_1 in order to accurately calculate its local clock's offset. The acquisition of T_1 can be accomplished in one of two ways:

- The timestamp T_1 can be placed into the *Sync* message before transmission by the master. This requires specialised hardware.

- A *Follow_up* general message can be used by the master to communicate T_1 to the slave (as illustrated in fig. 5.11)

If the slave and master employ the peer delay mechanism then at this stage in the process the slave has enough information to determine its clock offset. Hence, it will use the link propagation delay measurements it acquired via the peer delay mechanism to correct T_2 and, thus, calculate the clock offset.

If the slave and master employ the delay request-response mechanism, then the slave must acquire more time information. The slave transmits a *Delay_Req* event message which triggers the generation of timestamp T_3. The master on receipt of the *Delay_Req* message generates the timestamp T_4. This timestamp must be communicated back to the slave and is done so via a *Delay_Resp* message. At this point the slave has acquired the four timestamps ($T_1 - T_4$) necessary for it to compute its clock offset.

5.4.3 PTP Devices

Almost every network element that is said to be *"PTP aware"* is termed a PTP device. The role an element plays within a network is typically indicative of the PTP device it becomes within a PTP network. There are five such PTP devices: *ordinary clocks*, *boundary clocks*, *end-to-end transparent clocks*, *peer-to-peer transparent clocks*, and *management nodes*. Each of these devices implements various facets of the standard. Ordinary clocks and boundary clocks represent those devices that require synchronisation, whereas transparent clocks represent intermediate networks elements that are located on communication links between ordinary and boundary clocks. Management nodes, as the name suggests, are used for management and configuration of a PTP network. Each of these devices are discussed in detail in the following subsections.

5.4.3.1 Ordinary Clock

An ordinary clock represents the most basic device that requires synchronisation. This is typically a computing device with a single interface to a network and, therefore, possesses a single port. Depending on the role an ordinary clock plays

within a PTP network, its sole port is placed in either the master or slave state. If an ordinary clock is elected a grandmaster clock, then it acts as the primary time reference for every other ordinary and boundary clock within its domain. Such a clock is typically synchronised to an external time reference, such as GPS, and its port is placed in the master state which provides time information to other ports. If not elected a grandmaster clock, its port is placed in the slave state which acquires time information from another master port.

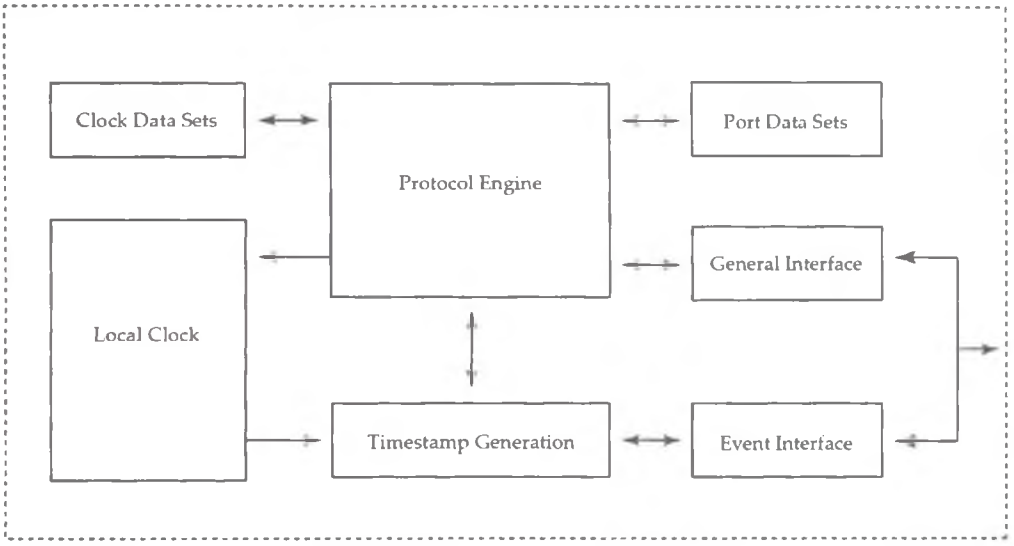


Figure 5.12: PTP ordinary clock

The structure of an ordinary clock is illustrated in fig. 5.12. It possesses a single *local clock*, port, and *protocol engine*. The protocol engine represents a core component of the PTP protocol and is typically associated with an individual port. It is responsible for determining the state of the port and for generating the necessary sequence of PTP messages required for synchronisation. As mentioned earlier, the port interfaces with its associated communication link via two logical interfaces: the event interface and the general interface. The event interface itself has access to the local clock via a *timestamp generation module* which timestamps incoming and outgoing messages. Access to this timestamp generation module distinguishes it from the general interface.

In addition to these core components, the ordinary clock contains two types

of data sets, one of which is associated with the port and the other which is associated with the ordinary clock itself. They are termed the *port data sets* and the *clock data sets* respectively. The clock data sets hold information that describes the local clock, parent clock (device to which it directly synchronises), and grandmaster clock. In addition to this, they also hold attributes related to synchronisation and the employed timescale. The port data sets hold information related to a specific port, in particular its state. It is the protocol engine that is responsible for maintaining these data sets. These data sets provide valuable information during the construction phase of the PTP master-slave hierarchy, as will be described later.

5.4.3.2 Boundary Clock

A boundary clock is the second and only other PTP device that synchronises its clock within a PTP network. The biggest distinction between it and an ordinary clock is the number of ports it contains (see fig. 5.13). It typically represents a network element, such as a bridge, router, or repeater and, therefore, may interface with a network via multiple physical links.

A boundary clock contains a single physical clock and a single collection of clock data sets. Every physical port on a boundary clock is associated with a separate protocol engine and a collection of port data sets. Since more than one protocol engine resides on a boundary clock, the protocol engines in a boundary clock have the ability to determine which one of the multiple ports on the device is responsible for synchronising the local clock.

5.4.3.3 End-to-end Transparent Clock

An end-to-end transparent clock represents a network element such as a bridge, router, or repeater that is said to be *"PTP aware"*. In contrast to ordinary and boundary clocks, an end-to-end transparent clock does not synchronise its local clock to any particular PTP timescale. The device is typically found on the communication link between a master port on one PTP device and the corresponding slave port on another PTP device. The purpose of the end-to-end transparent clock is to alleviate the negative effects that asymmetric two-way message laten-

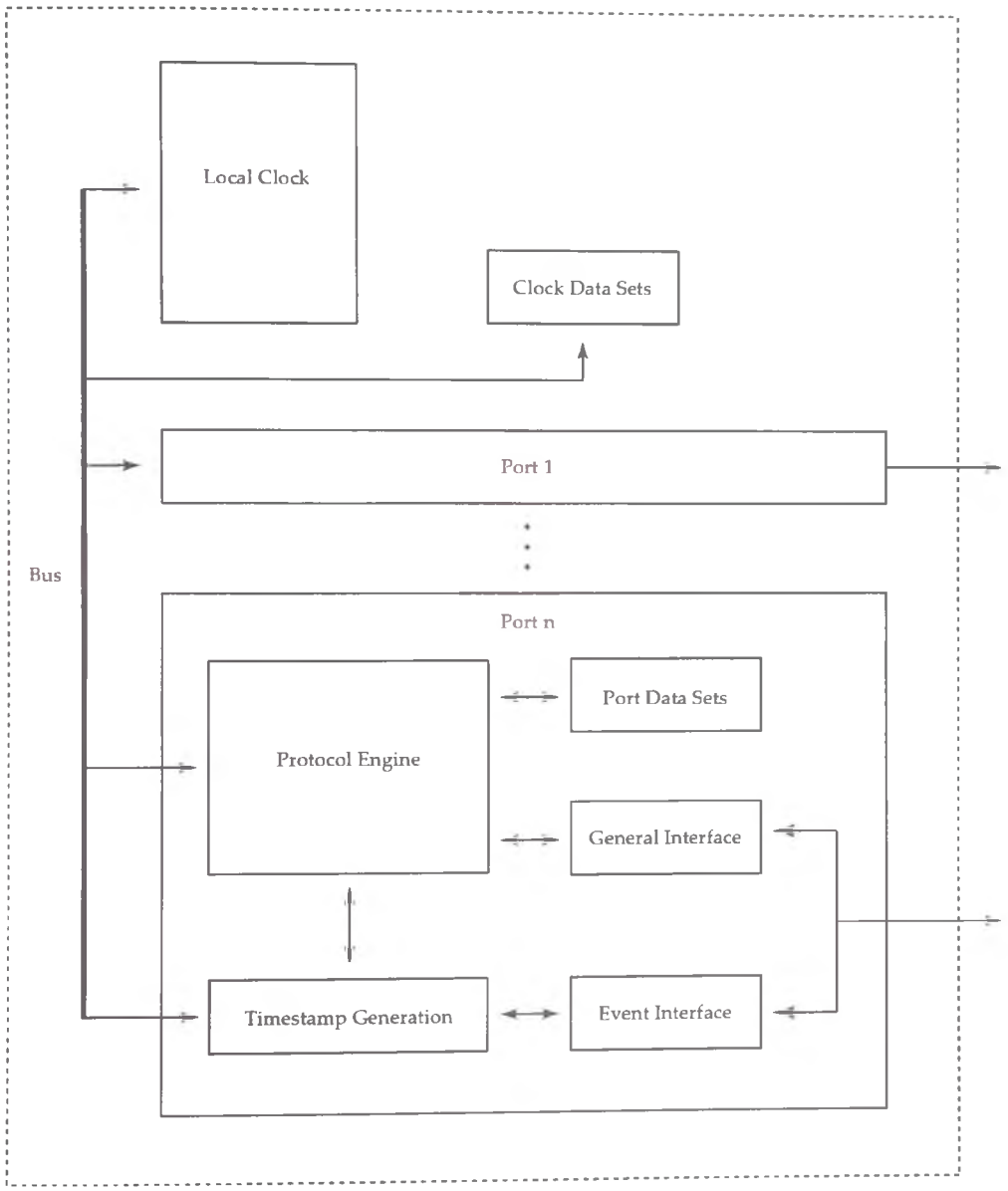


Figure 5.13: PTP boundary clock

cies have on the delay request-response mechanism. Each end-to-end transparent clock is responsible for correcting delays it directly subjects a PTP event message to. It accomplishes this by measuring the residence time of PTP event messages as they traverse the various communication layers of the device. This residence time is appended to the event message and subsequently used by a slave to alter/correct timestamps.

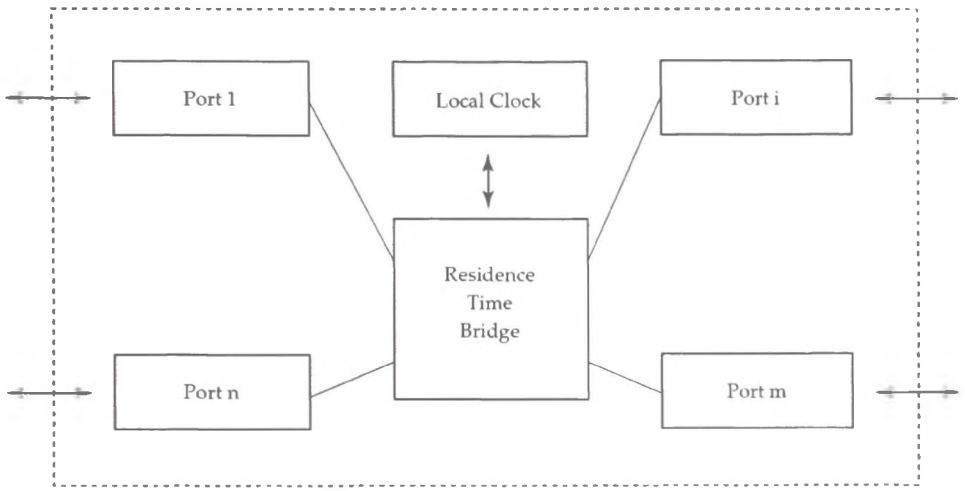


Figure 5.14: PTP transparent clock

As illustrated in fig. 5.14, the core component of an end-to-end transparent clock is the *residence time bridge*. The reception of a PTP event message by an end-to-end transparent clock triggers the generation of a timestamp at the reception port. This timestamp is referred to as an *ingress timestamp*. When the same PTP event message is transmitted by the transparent clock, it triggers the generation of a timestamp referred to as an *egress timestamp*. The difference between these timestamps, referred to as the *residence time*, is calculated by the *residence time bridge* and either appended to a special field in the event message called the *Correction field* or placed into the event message's corresponding *Follow_Up*. This residence time can then be used by the slave port on an ordinary or boundary clock to adjust timestamps generated by its associated master.

An important factor that must be considered when employing end-to-end transparent clocks is clock frequency. The residence time of an event message is

calculated using a transparent clock's local clock. This clock is not synthesised to the corresponding master and, therefore, may have a frequency offset that might introduce error into the residence time calculation. This may be acceptable if the error introduced does not impact the accuracy requirements of a time sensitive application. However, if not acceptable, methods exist to remedy this.

One such method described in the PTP standard allows a transparent clock to synthesise its local clock to that of a master clock. This is accomplished by forcing the transparent clock to analyse timestamps in *Sync* and/or *Follow_Up* messages sent by a master. By comparing the rate of change of its local time with respect to the master's local time, the ratio of one rate to the other can be estimated. This ratio of rates can then be used to adjust the frequency of the transparent clock's local clock (directly or via timestamp adjustments), thus, synthesising it to the master's clock. This whole process operates closed loop since the adjusted clock's timescale is used in subsequent estimations.

5.4.3.4 Peer-to-peer Transparent Clock

Peer to peer transparent clocks are almost identical to end-to-end transparent clocks with the exception that they support the peer delay mechanism. This allows them to measure associated link propagation delays as well as facilitate measurement of link delays by *link peers*.

A peer-to-peer transparent clock determines the link propagation delay between it and its link peer using the communication exchange detailed in section 5.4.2.2. Since a peer-to-peer transparent clock employs the peer delay mechanism, unlike an end-to-end transparent clock, it only needs to correct for the residence time of *Sync* messages used in the synchronisation process. The residence time of a *Sync* message and the link delay are placed into the correction field of the *Sync* or corresponding *Follow_Up* message. The link delay value placed into the correction field of a message by a transparent clock represents the delay of the link that the message was received on, not the sending link. The device that subsequently receives the message from the transparent clock, whether it is a boundary clock, an ordinary clock, or another transparent clock, makes the appropriate link corrections for its reception link.

It must be noted that peer-to-peer transparent clocks can only be connected to the ports of PTP devices that, like it, support the peer delay mechanism. Likewise, end-to-end transparent clocks should be connected to devices that only support the delay request-response mechanism. PTP devices that support different mechanisms can only be bridged via a boundary clock with ports that support both mechanisms.

5.4.3.5 Management Node

PTP management nodes, as their name suggests, provide a means to administer a PTP network through the use of management messages. A management node can manage a PTP network automatically and/or provide an interface for manual configuration. Unlike other PTP devices, management nodes can be combined with any of the other PTP devices.

5.4.4 PTP Network Construction

Once deployed, a PTP network automatically configures itself into a master-slave synchronisation hierarchy. In order to establish this synchronisation hierarchy, each port on each device must enter an appropriate state. The three states that determine the structure of the synchronisation hierarchy are the *master*, *slave*, and *passive* states. A port in the master state is the source of time for all ports on its associated link, whereas a port in the slave state synchronises to the master port on its associated link. A port in the passive state does not take part in the synchronisation process. Its purpose is to eliminate cyclic paths in the synchronisation topology.

Determining which state a port should enter is the responsibility of the *protocol state machine*. Every port on a PTP device is associated with an independent copy of this state machine. It determines which state a port should enter by analysing PTP *Announce* messages. Announce messages contain data describing other clocks in the network. By comparing the local clock's data sets with information contained in Announce messages, the future state of a port can be determined. The algorithm used to compare the data sets of clocks is termed the *best master clock algorithm*.

5.4.4.1 Best Master Clock Algorithm

The primary purpose of the best master clock algorithm is to compare the data sets of two clocks and determine the better clock. Multiple Announce messages, each containing a description of a remote clock, may be received by a particular port. It is the duty of the best master clock algorithm to compare such clock descriptions to that of the local clock and choose the next state of the port. The algorithm itself is composed of two distinct sub-algorithms: the *data set comparison algorithm* and the *state decision algorithm.*

The data set comparison algorithm is responsible for the actual comparison. It performs pair-wised comparisons of clock attributes describing the remote and local clock. Each of the attributes used in the comparison have a different precedence. These attributes in order of precedence are -

- *prority 1* - A value between 0 and 255 configured by a user to indicate the priority of a clock. It is used in situations where a master has been selected. Lower values take precedence.
- *clockClass* - A value that represents the traceability of a clock's time or frequency. For example, a value of 6 indicates that a clock is synchronised to a primary reference time source, while a value of 7 indicates that a clock was synchronised to a primary reference but is now in holdover mode.
- *clockAccuarcy* - A value in hex that represents the accuracy of a clock. Each hex value is mapped to an accuracy range. For example, the lowest value of 20_{hex} indicates that a clock is accurate to within $25ns$, while a value of 23_{hex} indicates that a clock is accurate to within $1\mu s$.
- *offsetScaledLogVariance* - Represents the stability of a clock as measured by an almost perfect clock.
- *prority 2* - Similar to the *priority 1* attribute but with lower precedence.
- *clockIdentity* - A unique identifier. This is used by the data set comparison algorithm in the event of a tie.

In the event that multiple Announce messages that describe the same remote clock are received because of cyclic paths in the underlying network topology, the data set comparison algorithm uses a distance attribute for comparison. This distance attribute which is deduced from a field (*stepsRemoved field*) in Announce messages, represents the number of hops traversed by the message on transit to the port.

After the data set comparison algorithm has determined the better clock, the state decision algorithm determines the next state of the port. The next state, or *recommended* state, of the port is determined based on the current state of the protocol state machine and its transition rules.

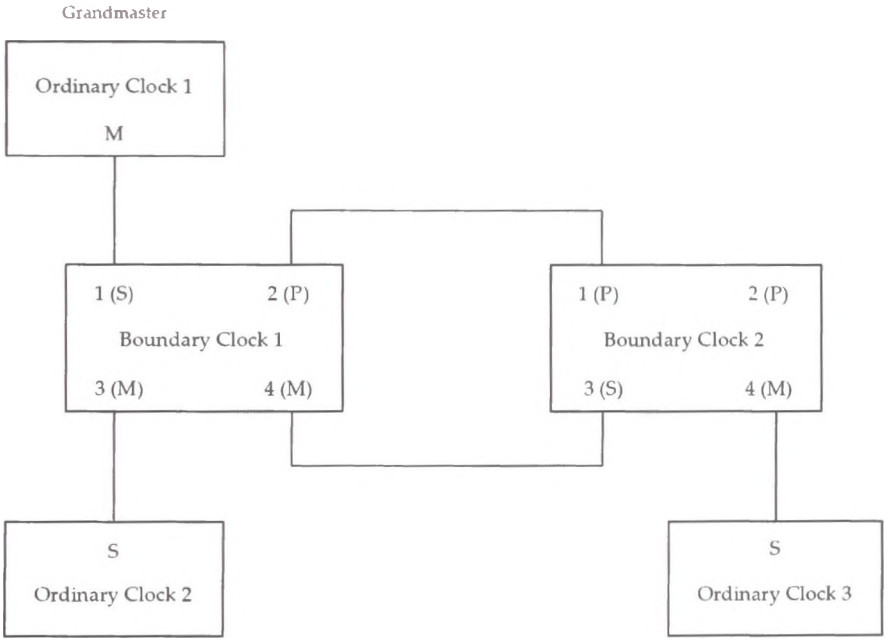


Figure 5.15: PTP master-slave hierarchy

An example of a PTP synchronisation hierarchy is illustrated in fig. 5.15. In fig. 5.15 ordinary clock 1 represents the root of the hierarchy and is, therefore, a grandmaster clock. It provides synchronisation to boundary clock 1. The port through which boundary clock 1 synchronises with ordinary clock 1 must be in the slave state. The remaining ports on boundary clock 1 must enter either the master or passive states in order to eliminate a cyclic path and yet still synchronise the

remaining network. Consequently, two of its remaining ports, 3 and 4, enter the master state providing synchronisation to ordinary clock 2 and boundary clock 2, while its remaining port, 2, enters the passive state in order to eliminate the cyclic connection between it and boundary clock 2. Similarly, boundary clock 2's ports, 1 and 2, enter the passive state, while its port 4 enters the master state to provide synchronisation to ordinary clock 3.

As is evident from fig. 5.15, ordinary clocks represent the leaves in the synchronisation hierarchy, whereas boundary clocks typically represent branches. The paths connecting the ordinary and boundary clocks may contain transparent clocks which do not participate in the synchronisation process but aid in eliminating asymmetric message delays.

5.5 Synchronisation over 802.11 Networks

The previous section detailed the two most dominant time protocols used in PSN networks. Each is designed to provide a particular degree of accuracy. NTP is designed for large, dynamic, latency-variable networks with nodes that host applications requiring millisecond synchronisation. In order to accommodate the dynamic nature of such networks, the protocol employs multiple time references/servers located over distinct links. To deal with the variable latencies associated with these networks, the protocol employs numerous statistical techniques to prune and identify quality data.

In contrast to NTP, PTP is designed for spatially localised networks that host applications which require sub-microsecond accuracies. In order to provide such a high degree of synchronisation, the network upon which it operates must be controlled. The network is assumed to be structured so as to reduce asymmetry. Any network elements that might cause variable packet latencies are equipped with specialised PTP hardware and software that eliminate the effects of asymmetry.

While both protocols are quite different in terms of their overall operation, they still, at their core, employ the round-trip synchronisation technique, albeit in different ways. Thus, both must provide methods to deal with asymmetry. PTP's approach proves most effective but more costly as devices must be equipped with specialised hardware. NTP's approach is more feasible, but in certain situations

it can lead to a significant reduction in accuracy.

One such situation in which accuracy can be degraded is when each NTP time reference shares a common communication link that subjects packets to large, non-deterministic, variable latencies. This situation may present itself when an NTP client operates on a wireless device. Since the wireless link represents the last leg of the journey, naturally, it is shared by all time references. In the case of an 802.11 wireless link, large traffic loads can lead to large variable packet latencies. The NTP protocol is designed to deal with asymmetric delays but does so by referring to its multiple time references which are assumed to be connected over distinct paths. In this particular scenario, this is not the case and, thus, each packet exchange between an NTP client and its time references may be subjected to the same asymmetric delays leading to significant time error, possibly in the order of hundreds of milliseconds.

To understand why and how 802.11 wireless links can degrade the performance of time protocols such as NTP, it is necessary to examine their operation. The following subsections detail the operation of an 802.11 network in the context of the NTP protocol. This will provide the reader with the facts necessary to appreciate the contribution presented in the next chapter.

5.5.1 802.11 Overview

IEEE 802.11 [3, 55] is a standard that specifies the operation of local area network wireless communication within the *ISM (industrial, scientific and medical)* radio bands. In essence, it adapts traditional *Ethernet (802.3)* to the wireless domain. It specifies the operation of two network layers: the physical layer and the MAC layer. While details of the MAC layer have remained relatively constant since the first standard was published in 1997, later revisions have added new physical layers to accompany the original *frequency-hopping spread-spectrum (FHSS)* and *direct-sequence spread-spectrum (DSSS)* physical layers. These new physical layers have significantly increased the bit rate of 802.11 networks.

Technically speaking, the adaptation of Ethernet technology to the wireless domain has been a challenging task. Relative to wired links, wireless links are very unreliable. Phenomena such as multipath propagation and signal attenuation

present issues that are not present in the wired domain. The use of a shared open-access medium with fuzzy undefined boundaries presents other issues in terms of hidden nodes and security. To deal with all of these issues, the MAC has undergone some major modifications relative to its wired counterpart.

Some of the methods used to deal with these issues include the use of positive acknowledgement for retransmission and an access method termed *CSMA/CA* to deal with hidden nodes. In addition to these, 802.11 networks must provide services that are innate in Ethernet networks. Examples of these include *association* and *authentication*. Association is used to register a station's MAC address with a *distribution system*, whereas authentication is used to prevent unauthorised access to the network. In a wired network, association is equivalent to plugging an Ethernet cable into a data jack, whereas authentication is equivalent to physically blocking off network equipment from users. This highlights the difficulties and challenges that wireless communication presents. The details of 802.11 wireless networks in terms of structure and operation follows.

5.5.2 Network Elements and Types

An 802.11 network is generally composed of four core elements. The first of these is a *wireless station* which typically represents a battery powered mobile device. The remaining elements provide the infrastructure through which data packets can be transmitted to and from stations and include a *wireless medium*, *access points*, and a *distribution system*. Access points (APs) perform wireless-to-wired bridging functions allowing stations to communicate with other stations not within their direct communication range. When several APs are employed to cover a large geographical area, they must be interconnected so that information related to the location and movement of stations can be communicated between them. The distribution system forms both the physical (network backbone) and logical components that facilitate this communication. Such communication is typically provided via an *inter-access point protocol (IAPP)*.

With these four elements a number of different network types can be constructed as illustrated in fig. 5.16. The core building block of an 802.11 network is a *Basic Service Set (BSS)* which operates within an area termed a *basic service*

area. A BSS consists of a number of stations that communicate with each other. Two variants of BSSs exist. The simplest of these is termed an *Independent Basic Service Set (IBSS)*. This consists of a number of stations that communicate directly with each other. The most basic IBSS consists of just two stations in direct communication with each other.

The second type of BSS is termed an *Infrastructure BSS.* In such a network, stations do not communicate with each other directly. All communication is relayed through an AP. Apart from the disadvantage related to communication overhead, this type of network offers a number of advantages. The first of these is that the basic service area is defined by the communication range of the AP only. This simplifies the process of defining network boundaries. Secondly, the AP is solely responsible for tracking what stations are in its associated BSS. This reduces the complexity that exists in the alternative network type where every station has to track the details of neighbouring stations. Finally, since APs act as a relay for all packets, they can aid in power saving by buffering packets and waking stations when necessary.

To extend the reach of an 802.11 network, a number of BSSs can be linked together to form an *Extended Service Set (ESS).* ESSs are formed by connecting APs together via a wired or wireless backbone. It is an ESS that employs the aforementioned distribution system to allow for inter-access point communication between APs. A station that wishes to send a packet to another station within the ESS but in a separate BSS simply sends it to its associated AP. This AP with the aid of the distribution system determines the AP associated with the receiver and delivers it to it via the distribution system medium or the ESS's backbone network.

5.5.3 MAC Challenges

The adaptation of a wired based technology such as Ethernet for the wireless domain presents many challenges. The leading issue is that concerning the quality of the link. Wireless links, particularly those within the ISM band, are subject to interference which can lead to attenuated signals. Other devices that operate within the same band can introduce noise and corrupt signals. In addition to this,

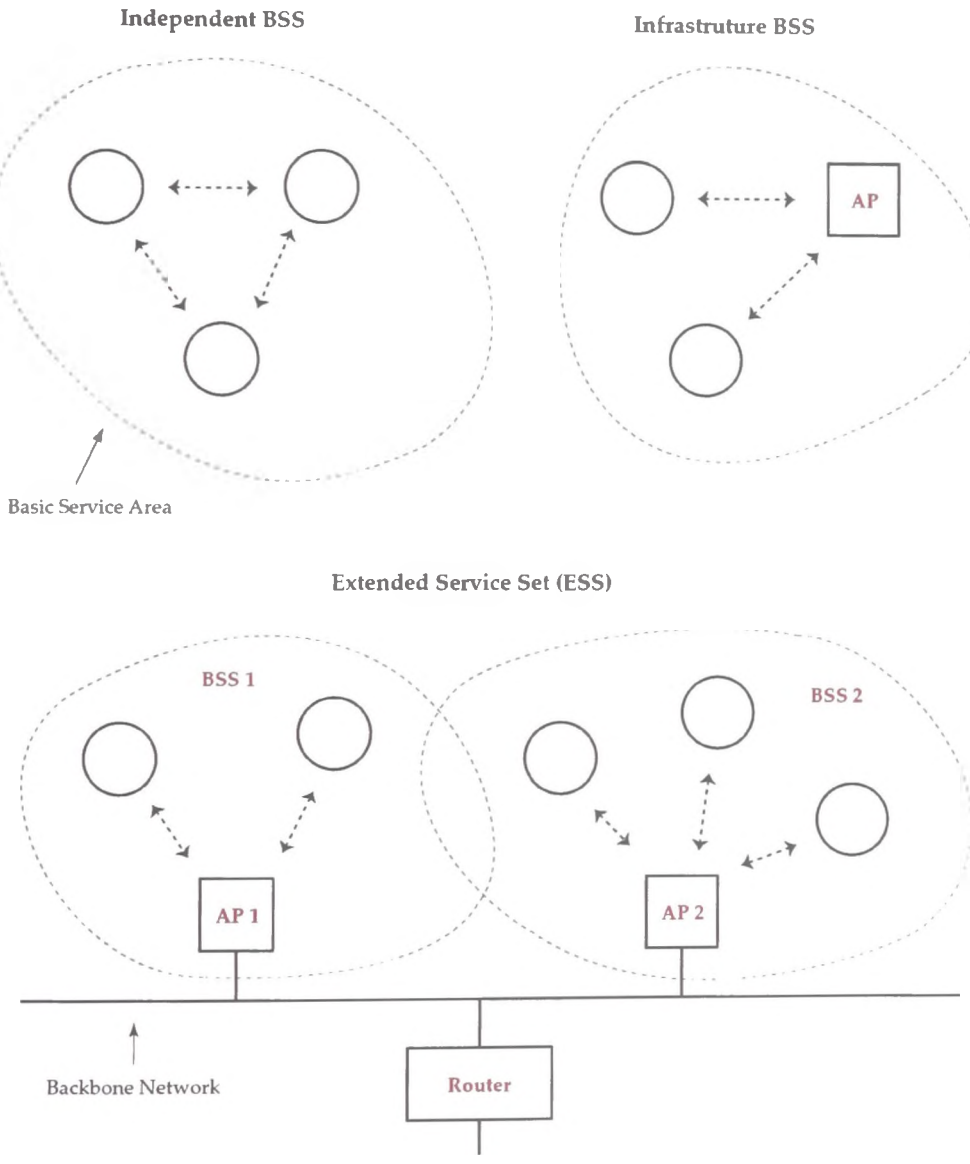


Figure 5.16: 802.11 network types

physical obstacles that obstruct direct communication between wireless devices can lead to multipath propagation and fading.

To deal with such issues, the 802.11 protocol incorporates *positive acknowledgement* of data packets (frames). All uni-cast frames transmitted between two devices must be acknowledged by the recipient. If an acknowledgement is not received by the sender, the communication is considered a failure and the sender attempts to retransmit the frame. This communication exchange between the sender and receiver is an *atomic operation* and must complete fully to be considered a success. In order to avoid the interruption of this atomic communication exchange by a third party, stations may reserve the medium for a period of time. This is done by advertising the reservation time to other nodes.

Another issue inherent in wireless networks is one termed the *hidden node problem.* This stems from the limited communication range of stations. For example, as illustrated in fig. 5.17, a station, A, in the process of transmitting a frame to another station, B, may not be overhead by a third station, C, due to its physical location. Thus, C is unaware that the medium has been reserved and initiates its own communication exchange with B which results in a packet collision.

To remedy this, 802.11 networks employ a variant of the network multiple access method used in Ethernet networks. This method is termed CSMA/CA or *carrier sense multiple access with collision avoidance.* Rather than just detect a collision as a station would do on an Ethernet network, 802.11 stations attempt to avoid collisions. This is accomplished by forcing a station, A, that wishes to transmit a frame to first transmit a short frame termed a *Request to Send (RTS)* frame. This is responded to by the recipient, B, with a corresponding *Clear to Send (CTS)* frame. The RTS and CTS frames notify all stations in A and B's communication range that the medium is reserved and, thus, any collisions that may have resulted are avoided. In reality an RTS/CTS exchange for every frame transmission can result in a lot of overhead and, thus, it is typically only employed in highly active networks. In addition to this, there exists a RTS threshold that specifies the maximum size a frame can be before the RTS/CTS exchange is employed. This exists because larger frames take longer to transmit and are, thus, more likely to collide with a hidden node's frame.

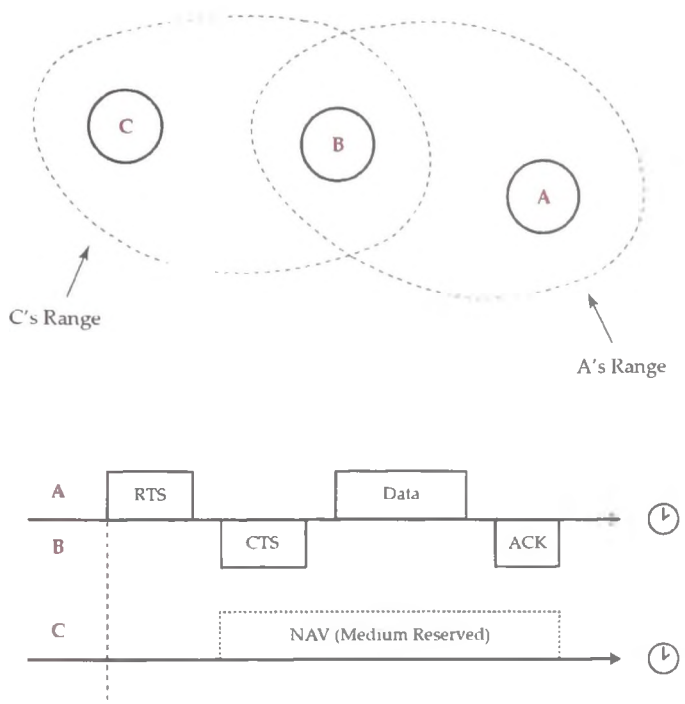


Figure 5.17: Hidden node and RTS/CTS exchange

5.5.4 Carrier Sensing

As mentioned in the previous section, 802.11 employs the CSMA/CA network access method. Collision avoidance is provided via the MAC using RTS and CTS frames. *Carrier sensing*, on the other hand, is performed at both the MAC and physical layers. Continuous physical layer carrier sensing is impractical in wireless networks sense most transceivers cannot transmit and receive simultaneously. Transceivers that can are generally expensive. Thus, when an 802.11 station is transmitting a frame, it cannot simultaneously sense the medium and so a collision may not be detected.

To remedy the limitations of physical carrier sensing, a second carrier sensing function is employed at the MAC layer. It is termed *virtual carrier sensing*, and it is facilitated via the *Network Allocation Vector (NAV)*. The NAV is a simple timer that is used by a station to reserve the medium so that atomic communication exchanges can complete uninterrupted. Dissemination of NAV

timing information is aided via the *duration* field of a frame. When a sender transmits a frame, it places the time period that it intends to reserve the medium for into the *duration* field of the frame. All stations within communication range of the sender inspect the *duration* field of the frame and set their NAV timer to its contents. The NAV subsequently counts down from this value. In relation to the virtual carrier sensing mechanism, it considers the medium to be busy while the NAV contains a non-zero value.

5.5.5 Inter-frame Spacing

Inter-frame spacing plays an important role in 802.11 networks. An inter-frame space represents the duration of time between the reception and transmission of two consecutive frames. It provides a means to prioritise traffic since a sequence of frames separated by a short inter-frame space gain access to a medium and reserve it quicker. This proves useful when frames must be fragmented. In such a case, the fragments are separated by a shorter inter-frame space than that of standard whole-frames. This reduces the chance of another station reserving the medium before all fragments have been transmitted.

The various inter-frame spaces (see fig. 5.18) and their duration are constant and independent of the physical layer employed with the exception of the EIFS (see below). The four types of inter-frame spaces are:

- *Short inter-fame space (SIFS)* - This has the shortest duration and is therefore used for the highest priority transmissions. After the medium becomes idle and the SIFS time has elapsed, high priority transmissions can commence. Examples of such transmissions include a RTS frame's associated CTS frame, a frame's corresponding acknowledgement and a fragmentation burst (sequence of fragments). Typically such transmissions should not be interrupted, particularly acknowledgement frames which form part of an atomic communication exchange.

- *DCF inter-fame space (DIFS)* - This is used by the *distributed coordination function (DCF)* which controls access to a contention based wireless medium (explained later). In such networks, whereby stations compete for

the medium, the DIFS represents the minimum duration the medium must be idle before a station can gain access to the medium. If the medium is still idle after this interval, then a station has the right to begin transmission immediately.

- *PCF inter-fame space (PIFS)* - The PIFS is shorter than the DIFS but longer than the SIFS. It is used by the *point coordination function (PCF)* which in contrast to the DCF function provides contention free services. In reality, the PCF is not employed very often.
- *Extended inter-frame space (EIFS)* - In contrast to the other inter-frame spaces, the EIFS is not a constant value. It is used in the event of an error, and its length varies depending on the circumstances.

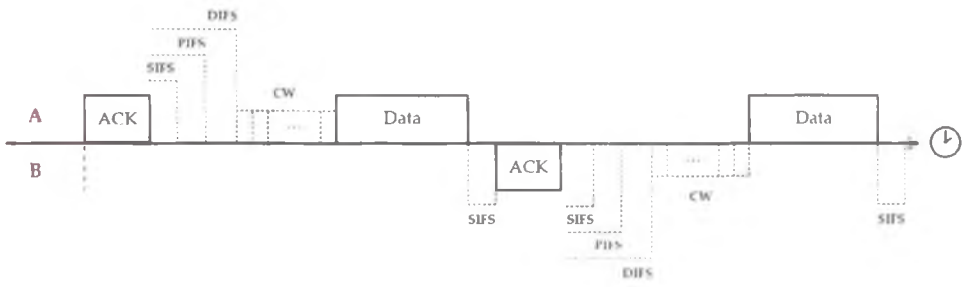


Figure 5.18: 802.11 interframe space

In contention based 802.11 networks, a station waits for the medium to be idle for a least the DIFS before it attempts to gain access. Once it gains access to the medium, all subsequent traffic may be transmitted after the SIFS. This ensures that atomic exchanges are not pre-empted. In addition, a station may continue to reserve the medium via the NAV, thus, allowing the delivery of a full sequence of frame fragments.

5.5.6 Distributed Coordination Function (DCF)

The 802.11 standard defines two coordination functions. These functions are responsible for controlling access to the medium. The first of these is the *Distributed*

Coordination Function (DCF) which provides a contention based service similar to Ethernet by means of the CSMA/CA mechanism mentioned in section 5.5.4. The second, termed the *Point Coordination Function (PCF)* which is built on top of the DCF, provides contention free services by using a centralised access control method. In reality, the PCF function is not widely employed and so is not discussed further.

The DCF, which is employed in the vast majority of 802.11 networks, provides a means for the stations within a BSS to coordinate access to the medium without any central management. By following a set of rules, access can be controlled in a relatively fair manner.

5.5.6.1 Basic Operation

Before a station can access the medium it must first perform physical and virtual carrier sensing (see section 5.5.4) to ensure that the medium is idle for a specific duration. This duration depends on the result of the previous transmission.

- If the previous transmission contained errors, then this duration is equal to the EIFS.
- If the previous transmission contained no errors, then this duration is equal to the DIFS.

Once this interval passes the station may commence transmission. If the physical or virtual carrier sense functions indicate that the medium is busy, then the station must defer access. When a station defers access, it waits for the medium to be idle for at the least the DIFS and then initiates the *exponential backoff procedure.*

5.5.6.2 Exponential Backoff Procedure

The period after the DIFS is termed the *backoff window/contention window (CW).* This window consists of slots, the number of which is based on the amount of successive access deferrals by a station. Each subsequent deferral results in double the previous number of slots plus an additional slot until the maximum

number of slots is reached (see fig. 5.19). Thereafter, the number of slots remains constant. Only after a successful transmission or in the event that a frame is discarded is the contention window reset. The actual length of a slot is dictated by the underlying physical layer. Naturally, those physical layers with higher speeds have shorter slots.

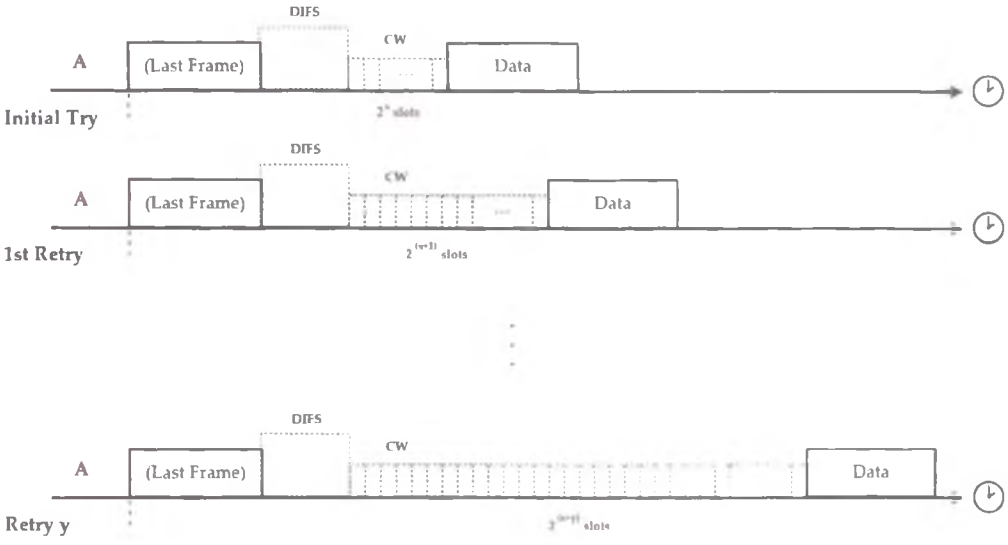


Figure 5.19: 802.11 exponential backoff procedure

When a station defers, it randomly picks a slot from the contention window and waits the corresponding time interval before sensing the medium again. Since each station chooses a slot randomly, access to the medium is conducted in a fair and orderly manner.

5.5.6.3 Error Recovery

The transmission of a frame and its corresponding acknowledgement frame is an atomic operation and must be completed in its entirety. Due to the manner in which the exchange is carried out, it is clearly the responsibility of the sender to determine if a transmission is a failure. In such cases, an acknowledgement is not received by a sender, and it must retransmit the frame. Obviously retransmissions cannot proceed indefinitely and so a *retry counter* is employed. This retry counter

is incremented in the event of a failed transmission. A failed transmission not only includes the lack of an acknowledgement but also includes a failure to gain access to the medium. Once the retry counter expires the frame is discarded.

5.5.7 Fragmentation

In particular cases, a higher layer data unit may be too large to place into a single frame. In such cases it is necessary to fragment the data unit into smaller units and place them into individual frames. This occurs when the higher layer data unit is larger than the network's configured *fragmentation threshold*. The fragmentation threshold in a network frequently subjected to interference is typically set to a low value in order to improve reliability. In such cases, when a data unit is fragmented and the fragment is corrupted by noise, only a small piece of the data unit needs to be retransmitted. This can ultimately lead to better throughput.

When a data unit is fragmented, it must be reassembled at the receiver in the correct order. Every frame in an 802.11 network has an associated *sequence number* which is located in the *sequence control* field of the frame. This sequence number allows stations to identify duplicate frames. When a frame is fragmented, it is also given a *fragment number* to allow a receiver to reconstruct the higher layer data unit.

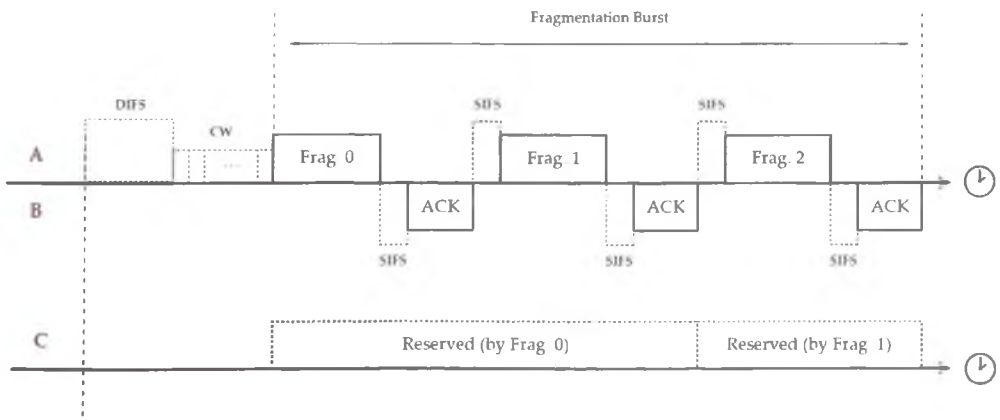


Figure 5.20: 802.11 fragmentation burst

When fragmentation does occur, it usually occurs in a burst or *fragmentation burst*. This burst is illustrated in fig. 5.20. In fig. 5.20, station A fragments a higher layer data unit and transmits the first fragment to B. After B acknowledges the fragment, station A since it has already acquired the medium can access the medium again after an SIFS period. Station A can continue to reserve the channel this way as it is permitted by the 802.11 standard. Since every other station must wait for A to finish transmitting all of the fragments, the exchange is considered a burst.

5.5.8 MAC Frame Types

Up to this point, the mechanisms that control access to the wireless medium have been described. The structure of an 802.11 frame and its fields form another important facet of an 802.11 network. Apart from encapsulating higher-level protocol data, 802.11 frames also aid in network management and control. This is facilitated via information transmitted in the various fields of frames. 802.11 MAC frames can be categorised into one of three groups:

- *Data Frames* - Data frames, as their name suggests, are used to transmit higher-level protocol data to remote stations. They represent the core 802.11 frame since every other type of frame either directly or indirectly facilitate the distribution of data frames.

- *Control Frames* - Control frames are employed to directly aid in the delivery of data frames. It is control frames that aid in reliability and medium access. Examples of control frames include acknowledgement frames and RTS/CTS frames.

- *Management Frames* - Management frames are used to provide 802.11 services. Examples of these services include association and authentication, which are provided with the aid of association and authentication frames. Other management frames such as *beacon* frames are used to advertise the existence of a network as well as disseminate specific information about the network so that stations can determine if they are compatible.

A detailed analysis of every type of frame is outside the scope of this work. However, in order to aid in one's understanding of the next chapter, the following sections examine the structure of a generic data frame and detail those fields that are relevant.

5.5.9 Data Frame

Fig. 5.21 illustrates the structure of a generic data frame. In comparison to the 802.3 Ethernet frame, there are a number of clear differences, the most dominant being the utilisation of two extra address fields. In addition, there exists a field termed the *frame control field* which, among other things, specifies the frame type. Also, one can observe a field termed the duration field which aids in medium reservation. A detailed analysis of each field follows.

5.5.9.1 Frame Control Field

Every 802.11 frame starts with a two byte long frame control field. Fig. 5.21 presents the various sub-fields that form the frame control field. Some of the more relevant sub-fields are:

- *Type and subtype fields* - The *type* and *subtype* sub-fields together identify the type of frame received. The type sub-field specifies the category of frame, i.e., data, control or management, while the sub-type specifies the type of frame in that category. In an 802.11 data frame, the type is set to *"10"* and the sub-type is set to *"0000"*.

- *ToDS and FromDS bits* - These bits indicate whether a frame is destined for the distribution system. The interpretation of the four address fields that comprise a data frame vary depending on where a transmitter and receiver are in the network. In a data frame the *ToDS* and *FromDS* bits are used to interpret the address fields.

The other 1 bit sib-fields also play an important role depending on the situation. For example, the *More Fragment* bit indicates that the frame has been fragmented and not all of the fragments have been received yet, while the *Retry*

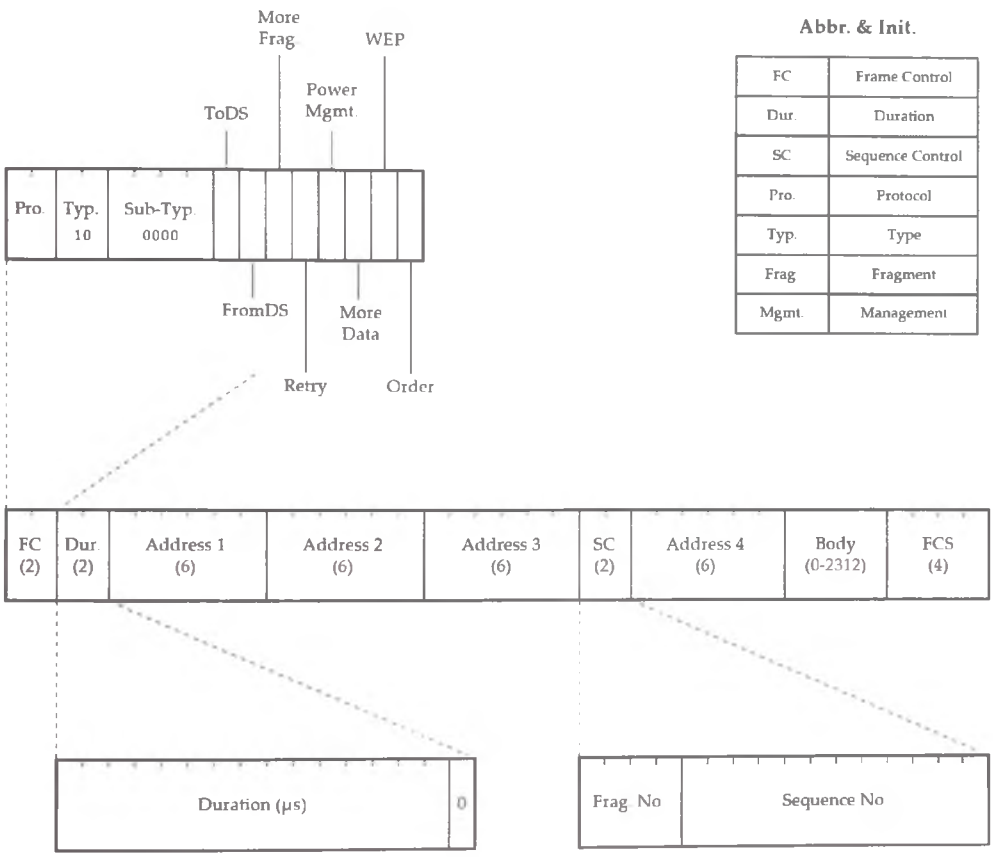


Figure 5.21: 802.11 data frame

bit, which is set for retransmitted frames, aids the receiver in identifying duplicate frames.

5.5.9.2 Duration/ID Field

The Duration/ID has a number of functions depending on the type of frame. In a data frame the 16th bit is set to zero indicating that it holds the amount of time, in microseconds, that the medium will remain busy for the current transmission. It is used by nearby stations/APs to set their network allocation vector (NAV) (see section 5.5.4). The value in the duration/ID field is added to the current contents of a NAV timer. It is subsequently used by a station's virtual carrier sensing function to determine when the medium is idle.

5.5.9.3 Address Fields

802.11 address fields, similar to Ethernet address fields, are 48 bits long. However, while an Ethernet frame has two address fields, an 802.11 frame may have up to four depending on the type of frame. An 802.11 data frame possess each of the four fields, but the interpretation of each field and whether they are all set depends on where the frame is coming from and going to. The four types of addresses that may be found in an address field are -

- *Transmitter address (TA)* - This represents the address of the station that physically transmitted the frame on to the medium.
- *Source address (SA)* - This is the address of the station that the message originated from, that is, the station that created the original message.
- *Destination address (DA)* - This is the address of the station which is the final recipient of the message, that is, the station that hands the message to a higher level protocol.
- *Receiver address (RA)* - The receiver address represents the address of the node that receives a message from the wireless medium but may or may not be the final destination.

- *BSSID* - The basic service set ID is an address used to differentiate messages from multiple BSSs located in the same area. In general, it is set to the MAC address of the AP that services the BSS. In an IBSS it is randomly assigned.

In an 802.11 data frame the actual type of address found in each of the address fields $1-4$ is determined by analysing the *ToDS* and *FromDS* bits of the frame control field. Table 5.1 lists all possible permutations of these bits along with the corresponding address field contents. It can be observed from table 5.1 that in an infrastructure network the BSSID takes the place of the receiver and transmitter address types, since it is also the address of the AP.

ToDS	FromDS	Addr. 1	Addr. 2	Addr. 3	Addr. 4
0	0	DA	SA	BSSID	N/A
1	0	BSSID	SA	DA	N/A
0	1	DA	BSSID	SA	N/A
1	1	RA	TA	DA	SA

Table 5.1: 802.11 data frame address fields

Determining whether a frame is destined for a single station or multiple stations within a BSS is based on the value of the first bit transmitted onto the medium. If it is set to 0, it is a *unicast* frame, whereas if it is set to 1, it is a *multicast* frame. A *broadcast* frame has all 48 bits set to 1.

5.5.9.4 Sequence Control Field

The *sequence control* field is 16 bits long and consists of a 4 bit *fragment number* sub-field and a 12 bit *sequence number* sub-field. As explain in section 5.5.7, a sequence number is associated with each frame in order to allow a receiver to detect and discard duplicate frames. The fragmentation field is used to determine the order of the fragments of a higher layer data unit so that they can be reassembled at the receiver.

5.5.9.5 Frame Body

The *frame body* field holds higher layer data units. The maximum amount of data that can be transmitted in a frame is 2,304 bytes with an extra 8 bytes used for *Wired Equivalent Privacy (WEP)* security.

5.5.9.6 Frame Check Sequence

The *frame check sequence (FCS)* field, also known as the *Cyclic Redundancy Check (CRC)*, is used to determine the integrity of a frame. The integrity check includes all the contents of the header together with the frame body, and 802.11 uses the same mathematical technique as Ethernet. Naturally, because of differences between their headers, the FCS must be recomputed by an AP when it transmits a frame from an 802.11 network to an Ethernet network and vice versa.

5.5.10 Issues with Synchronisation

The previous sections have detailed some of the core aspects of 802.11 networks with a particular focus on features that can affect the performance of higher layer time synchronisation protocols. One of those aspects that can lead to such performance degradation is an 802.11 network's use of a shared, low-bandwidth, unreliable medium. In contrast to wired networks, the throughput of 802.11 networks is significantly lower. When a large number of stations contend for the medium simultaneously, this can lead to large packet delays. With respect to a time synchronisation protocol, such as NTP, which employs a two-way message exchange, packet delays are not necessarily an issue unless the uplink and downlink delays of the request and reply packet respectively, differ. Thus, an issue arises when these delays are asymmetric. Unfortunately, in 802.11 networks, particularly those with high traffic loads, this is generally the case.

In 802.11 networks, asymmetric packet delays are caused by a number of different factors. In addition to the aforementioned factors associated with the quality of the medium, another important factor is that concerning the priority of wireless nodes. APs and stations both use the DCF to coordinate access to the medium and, therefore, have equal priority in terms of medium access. However,

an AP typically has more packets to transmit as it services all of the stations in its associated BSS. In terms of memory and processing resources, an AP is equipped to deal with such large traffic loads, but this solely means that rather than discard packets, an AP can queue them for longer. When the majority of traffic in a BSS is downlink traffic, the Tx (Transmission) queue in the AP's wireless interface far out-weights that of a stations Tx queue which leads to asymmetric delays in the network.

Although less usual, the same issue also arises in the case of a station which, in addition to hosting a time synchronisation client, also hosts another application that generates significant upload traffic. In this case, the station's Tx queue will out-weight that of the AP's Tx queue leading, again, to asymmetric two-way message delays between the station and AP

In relation to the synchronisation protocols described in this chapter, the performance degradation of NTP in the above scenario would far outweigh that of PTP. In fact, the performance of PTP would not be affected if an AP and station were equipped with specialised PTP hardware. NTP, although equipped with effective statistical techniques for pruning inaccurate data, relies on numerous servers located across diverse communication links. In the above scenario, the 802.11 link would represent the first and last leg of the journey and would, thus, be shared by all servers. Eventually, a significant clock error would result.

5.5.11 Summary

This chapter detailed the operation of two popular time synchronisation protocols employed in packet switched networks, namely NTP and PTP. Each protocol is designed to operate over a particular type of network, and each strives to provide a different level of synchronisation accuracy.

NTP is designed for large, dynamic, variable-latency networks and provides sub-millisecond synchronisation accuracies. To deal with the dynamic nature of such networks, the protocol employs multiple redundant time references located across diverse communication links. It uses a suite of statistical algorithms to prune a collected data set and employs a hybrid PLL/FLL control loop to make gradual clock adjustments ensuring a continuous monotonic local clock.

PTP, in contrast, is designed for local managed networks and is capable of providing microsecond level synchronisation or better under the right conditions. To provide such a high degree of synchronisation, the protocol employs an array of PTP devices with specialised PTP hardware capable of time-stamping time messages at the physical layer of the network. The utilisation of physical layer time-stamping eliminates the sources of non-deterministic message latencies and, thus. eliminates the negative effects of asymmetric two-way message delays.

Both NTP and PTP are designed for the wired domain. The vast deployment of 802.11 networks in the last decade has altered the composition of networks. Wireless links now represent a large subset of the extremities of networks. In relation to time synchronisation protocols, such networks present issues that are far less severe in wired networks. The use of a shared un-reliable communication medium can, in certain cases, result in large asymmetric two-way message delays degrading the performance of time protocols that employ round-trip synchronisation. While PTP can avoid this problem by employing physical layer time-stamping, the same is not true of NTP. Thus, a solution must be found.

The next chapter presents a technique that can be employed in an 802.11 network in order to alleviate the negative effects that asymmetric packet delays can have on time protocols such as NTP.

Chapter 6

Improving Time Synchronisation in 802.11 Networks

The previous chapter focused on the predominant PSN time synchronisation protocols, NTP and PTP. It also detailed the operation of 802.11 networks and explained why such networks can negatively impact the performance of a time protocol such as NTP. This chapter first quantifies the extent of this problem and then presents and validates a solution. With this solution, the latency of time messages as they traverse an 802.11 link can be determined in real-time. This information can be used to correct temporal data and, thus, increases its quality before it is employed by a time synchronisation protocol. Consequently, a time protocol such as NTP can utilise a higher quality data set which can lead to an improvement in its performance.

The following sections present a detailed analysis of the problem and then describe the operation of the aforementioned technique. The effectiveness of the technique is proven and quantified via an experimental testbed. The results indicate improvement gains that allow time protocols, such as NTP, to achieve synchronisation levels equivalent to that achievable over a wired link.

6.1 Problem Overview

To begin, a detailed analysis of the problem must be presented. The problem is presented with respect to the NTP protocol because of its widespread deployment in public IP networks and the increasing prevalence and need for mobile and wireless connectivity using 802.11 networks. As explained in section 5.3, NTP employs the round-trip synchronisation technique and, unlike PTP, does not employ physical layer time-stamping. Thus, it is susceptible to errors that stem from non-deterministic message delays. The components of such non-deterministic delays are detailed in section 2.4.1: the *send delay*, the *medium access delay*, the *propagation delay* and the *reception delay*. As explained in section 2.4.1, depending on the level in the protocol stack that a time message is time-stamped, some or all of the non-determinism associated with those components can be eliminated. While PTP benefits significantly from physical layer time-stamping, NTP employs application layer time-stamping and relies on multiple data sources and sophisticated statistical techniques to prune the data set.

With respect to 802.11 networks, the component that typically causes the greatest source of error is medium access delays. This is particularly the case when the network is subjected to large traffic loads. Section 5.5.1 details the operation of 802.11 networks and highlights the reason for such large medium access delays, namely the use of a low quality and low bandwidth radio link that multiple nodes must share. To handle contention in such networks, nodes must adhere to a strict set of access rules and those rules must be obeyed by all categories of nodes in the network, both station nodes and AP nodes. Since an AP services all of the stations in a particular BSS, it typically has a greater work load. However, it generally does not have higher priority in terms of medium access but instead is equipped with additional hardware capacity to deal with greater traffic loads. Equal priority between stations and APs coupled with the fact that the majority of traffic within a network is typically down-link traffic can lead to a situation whereby a time message spends a significant amount of time in an AP's *buffer*. The resulting asymmetry between the *up-link* and *down-link* delays of two associated NTP time messages can lead to a time error if not identified and eliminated by the NTP algorithms.

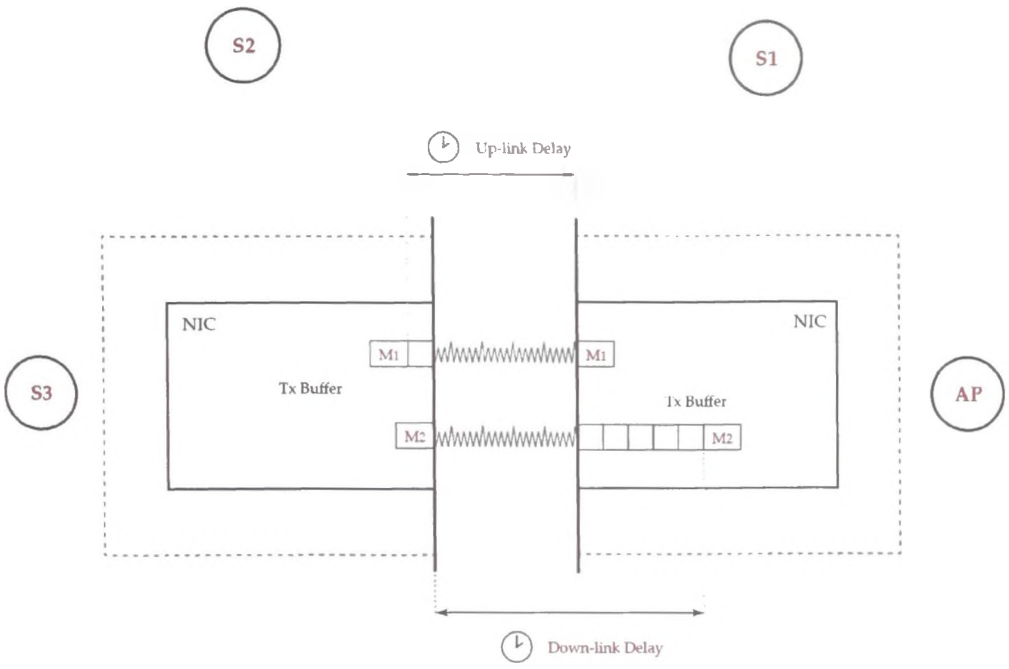


Figure 6.1: Up-link and down-link delays

This situation is illustrated in fig. 6.1. Here an AP and three stations, $S1$, $S2$, and $S3$, form a BSS. $S3$ hosts an NTP client which transmits an NTP request message at a particular poll interval. $S3$ must contend with nodes $S1$, $S2$, and the AP for access to the medium. When $S3$ gains access, it transmits the NTP request message, M_1, in an 802.11 data frame to the AP. The AP receives the frame and reroutes it to an appropriate node in the distribution system, possibly a router. At a later stage, the corresponding NTP reply message, M_2, is received by the AP from the distribution system. At this stage, the AP has a partially full *Tx buffer* and, therefore, it must queue M_2. The NTP reply message, M_2, will not be transmitted to $S3$ until the AP has gained access to the medium multiple times and transmitted earlier received messages to their appropriate destinations.

When $S3$ finally receives M_1, there may be a significant difference between the up-link delay of the request message and the down-link delay of the reply message. When the messages' associated timestamps are processed, an erroneous

offset estimate will be produced. If the NTP client on $S3$ is configured to poll multiple servers and the aforementioned scenario occurs infrequently, affecting only the occasional NTP message, then it is detected by the data filter algorithm, and the associated data is eventually discarded. If, however, it occurs frequently and the 802.11 link is used for all NTP exchanges between the client and its servers, then it is not detected and the local clock at $S3$ eventually becomes inaccurate.

This issue is not limited to delays at the AP. If instead delays are incurred by an outgoing NTP request message at $S3$, then a similar situation arises. In this case the up-link delay of the NTP request message may be much greater than the down-link delay of the associated NTP reply message. Such a situation could arise if $S3$ hosted an application that regularly produced large amounts of up-link traffic. Thus, outgoing NTP messages would be delayed in $S3$'s Tx buffer. In this particular situation, it would be more likely that $S3$ would drop frames before they incurred delays similar to that in an AP due to an AP's larger buffer size.

6.2 Evaluating Magnitude of Asymmetry

6.2.1 Experimental Overview

At this stage it should be clear what issues 802.11 links present to time protocols and where exactly they stem from. In order to justify formulating and contributing a solution to the problem, it is first necessary to quantify the magnitude of any asymmetry that might occur in an 802.11 network. With this information, an estimate of the possible time errors that could result at a client can be determined. The level of *"acceptable"* error is very much dependent on the application that requires synchronised time - this in turn informs the need for a solution.

In order to quantify the magnitude of asymmetric delays within an active 802.11 network, an appropriate simulation-based experiment is designed. The nodes that form an 802.11 network typically possess various communication capabilities and hardware resources. The 802.11 standard defines multiple physical layer modulation techniques, each of which offer a variety of data-rates. In the case of a BSS with multiple nodes, depending on the services offered by its asso-

ciated AP, stations may operate at a variety of speeds and use different physical layer modulation techniques. Furthermore, the hardware capabilities of nodes can vary significantly from one BSS to another. In particular, the buffer size of APs. Finally, individual BSSs may intersect quite extensively, and this adds further complexity to its operation. All of this variety and heterogeneity suggests that the up-link and down-link delays observed in one particular 802.11 network may differ significantly in comparison to another network.

Rather than attempt to model a typical 802.11 network, a homogenous 802.11 network is modelled whereby each node employs the same physical layer modulation technique and operates at the same data rate. To account for the various hardware capabilities of nodes across different networks, in particular the diverse buffer sizes of APs, a range of experiments are run whereby the buffer sizes of nodes are varied.

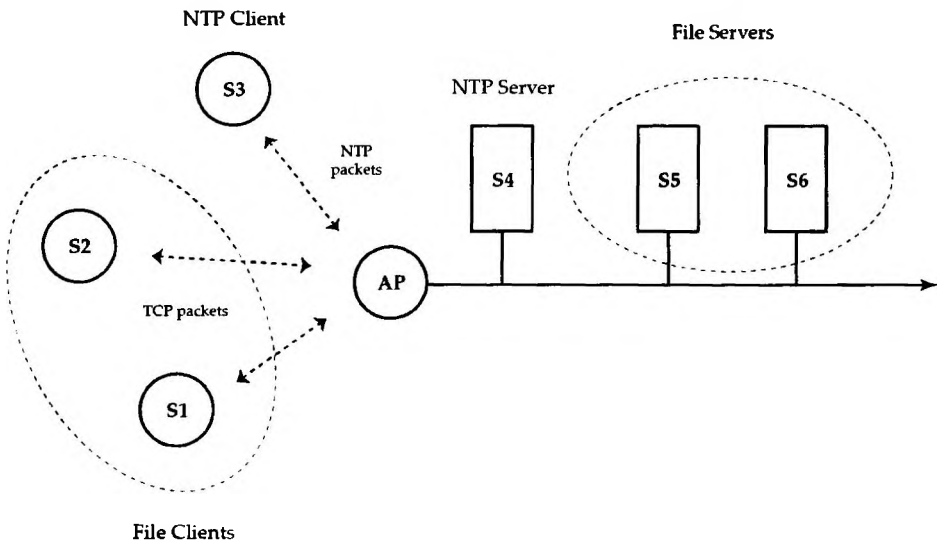


Figure 6.2: Experiment - evaluating magnitude of asymmetry

Fig. 6.2 presents the experimental design. It consists of both wired and wireless components. The wireless component represents a single BSS composed of 3 wireless stations and their associated AP. The wired component consists of 3

wired stations that are connected directly to the wired backbone that forms the AP's associated distribution system.

Wireless station, $S3$, hosts an NTP client that regularly polls an NTP server hosted by the wired station, $S4$. The objective is to create a scenario whereby NTP request messages are subjected to relatively minor delays at $S3$, and the corresponding NTP reply messages are subjected to relatively large delays at the AP. To realise this, wireless nodes $S1$ and $S2$ are configured with file clients that simultaneously download large files from file servers $S5$ and $S6$ respectively. The large amounts of down-link traffic generated by $S5$ and $S6$ should result in large wireless Tx buffer delays at the AP. Since the traffic is largely down-link traffic, the Tx buffer delays at $S3$ should remain relatively low in comparison to that of the AP's. The situation should produce asymmetric delays the magnitude of which can be determined from the experimental results.

All simulations are performed using the *NS3 simulator* [56, 57]. NS3 is a discrete-event simulator that can be used to accurately simulate a variety of network scenarios. The simulator allows one to define and construct topologies to which one can subsequently add entities termed *models* that accurately represent real-world network devices and protocols. The advantages of simulating a scenario such as that depicted in fig. 6.2 are numerous. Simulation eliminates factors which would otherwise be difficult to eliminate in a real-world testbed. Issues, such as interference and noise, caused by both intersecting 802.11 networks and other devices operating in the same ISM spectrum can be eradicated. In addition to this, simulations can be run at a fraction of the speed required to perform a real-world experiment and factors such as scalability can be analysed, a task which might otherwise prove impractical.

NS3 is designed for network simulation and, thus, provides models for a majority of network protocols. It does not, however, model application layer daemons and services. Thus, there is no model available to simulate the NTP protocol. However, this is not an issue because the only factors of concern are the size of the payload the NTP protocol generates and the transport layer protocol it employs. With this information one can use other methods to produce similar network traffic.

In general, the NTP protocol generates a 48 byte payload (section 5.3) and

employs the *UDP (User Datagram Protocol)* transport layer protocol to deliver this payload to its destination. Thus, it is necessary to install an application on $S3$ which generates a 48 byte UDP packet at regular intervals destined for $S4$. In addition to this, an application should be hosted on $S4$ which responds to each received packet with a similar size packet. This can be accomplished through the use of NS3's *UDPEchoClient* application model which can be configured to transmit a UDP packet of a specific size at a defined interval. The *UDPEchoClient* application works in combination with NS3's *UDPEchoServer* application model which simply returns all packets it receives from a *UDPEchoClient* application. These two relatively simple application models can be used to simulate NTP protocol traffic to a very high degree.

The next challenge entails the configuration of nodes $S1$, $S2$, $S5$ and $S6$ such that that the scenario depicted in fig. 6.2 can be simulated. These nodes must host appropriate NS3 application models that simulate $S1$ and $S2$ downloading large files from $S5$ and $S6$ respectively. As mentioned previously, the key reason for employing file client and server applications is to ensure that the majority of traffic generated within the network is in the down-link direction with respect to $S3$.

There is, however, another motive. Such applications generate *TCP (Transport Control Protocol)* traffic to reliably deliver files to their destination. The TCP protocol employs a *congestion control mechanism* which is designed to ensure that the network connecting the receiver to the sender does not become congested with TCP packets. The algorithm operates by analysing TCP acknowledgements at the sender. From this, it can deduce the number of TCP packets that are being dropped by the network and can calculate a suitable transmission rate. This algorithm operates in a closed-loop readjusting the transmission rate of the sender in response to changes in the network.

The advantage of this in relation to the experiment depicted in fig. 6.2 is that the TCP congestion algorithm ensures that the network operates at its peak and, thus, ensures that the AP's wireless interface Tx buffer is relatively full most of the time. Consequently, NTP reply packets sent from $S4$ to $S3$ are queued at the AP for a longer interval than that of NTP request packets at $S3$. This results in two-way asymmetric delays between $S3$ and the AP.

With respect to the simulation of a file download in NS3, this can be accomplished by employing NS3's *PacketSink* application. The purpose of the *PacketSink* application is to consume all traffic of a specific type and specific destination address. The application is *"attached"* to nodes $S1$ and $S2$ and is configured to consume all TCP traffic received on a particular port. With respect to traffic generation at $S4$ and $S5$, this can be simulated by first creating an NS3 *TCP socket* at both nodes and then connecting these sockets to their respective wireless clients, $S1$ and $S2$. Once configured, a specific amount of data can be transmitted to the wireless clients via the socket API.

6.2.2 Simulations and Results

The previous section presented the experimental setup that is used to quantify the extent of asymmetric delays that arise from varying traffic profiles in 802.11 networks. With respect to the previous section, an NS3 simulation of the experiment illustrated in fig. 6.2 is implemented using the aforementioned NS3 models. In order to analyse the impact of factors such as data rate and hardware capabilities, numerous simulations are run whereby the physical layer data rate and AP buffer size is varied for each. In particular, both the 802.11b and 802.11a physical layer standards which operate at a maximum data rate of 11Mbps and 54Mbps, respectively, are employed. In addition, the maximum buffer size of the AP is varied from 10 to 100 packets which, with respect to Li et al. [58], represents a reasonable range of sizes. The NTP client (*UDPEchoClient*) is configured with a transmission interval of 200ms and each simulation is run for 60s, a sufficient amount of time to collect the required data.

The combined results of these simulations are presented in fig. 6.3. The graph presents the average up-link and down-link delays incurred by NTP request and reply messages, respectively, as they traverse the link between $S3$ and the AP. In relation to these simulations, the up-link delay represents the time difference between when an NTP request message is added to the Tx buffer at $S3$ and detected at the physical layer of the AP. Conversely, the down-link delay represents the difference between when an NTP reply message is added to Tx buffer at the AP and detected at the physical layer of $S3$ (see fig. 6.1). In fig. 6.3 these delays

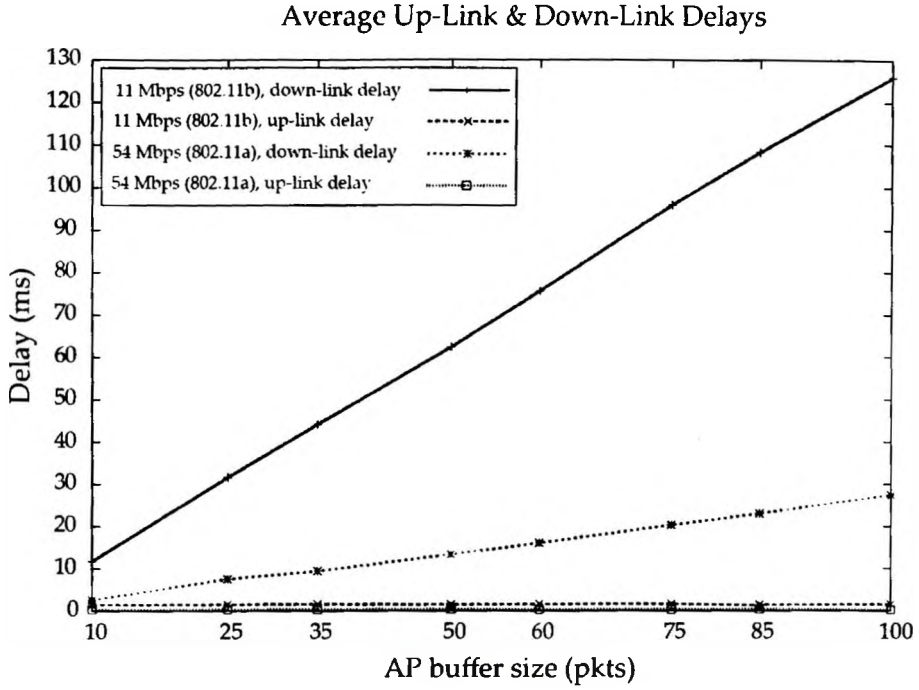


Figure 6.3: Simulation results - average up-link and down-link delays

are presented for both 802.11b and 802.11a wireless networks with AP buffer sizes ranging from 10 to 100 packets.

The results highlight the significant influence that data rate and buffer size have on packet delays. In relation to the buffer size, the results seem to suggest that simply reducing the size of the AP's buffer would alleviate the problem but, of course, this would lead to a reduction in the throughput of the network due to packet loss. This is verified in fig. 6.4 which presents the percentage of packet loss associated with the various AP buffer sizes for each of the simulations. A thorough analysis of the effects of buffer size is detailed by Li et al. [58].

In relation to time synchronisation, the factor of interest is the magnitude of the asymmetry, that is, the difference between the average up-link and down-link delays presented in fig. 6.3. These delays are presented in table 6.1. The impact of these delays on a time protocol that employs round-trip time synchronisation without any filtering mechanism can be examined by applying equation 5.2. The

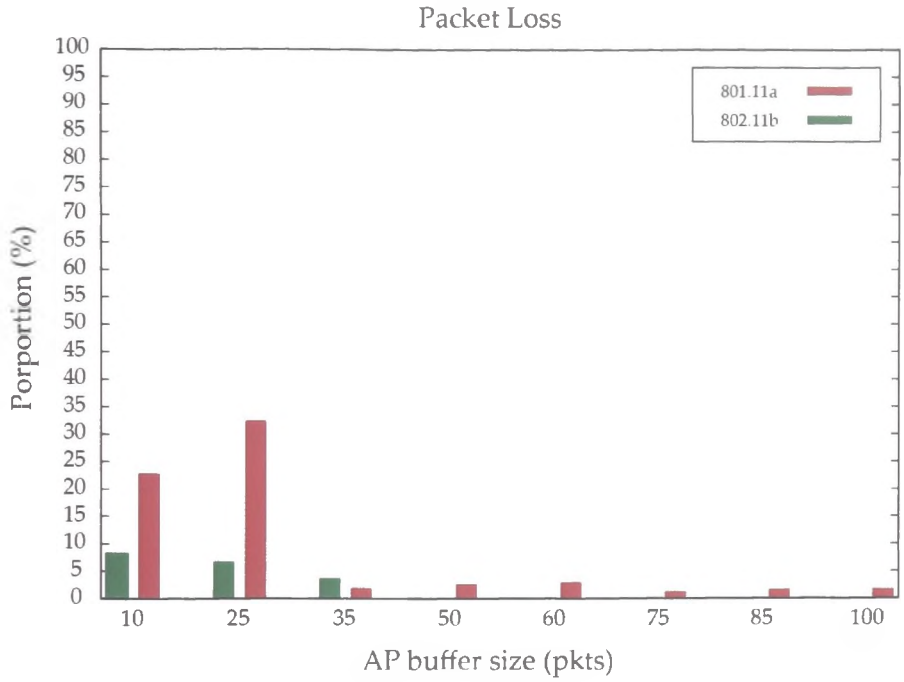


Figure 6.4: Packet loss

resulting time errors would be equal in magnitude to half those delays presented in table 6.1. These errors are illustrated in fig. 6.5.

	10	**25**	**35**	**50**	**60**	**75**	**85**	**100**
802.11a	2.28	7.19	9.05	12.96	15.73	19.91	22.67	27.15
802.11b	10.26	30.11	42.53	60.91	74.05	94.16	106.92	124.42

Table 6.1: Simulation results - asymmetric delays (ms)

The errors presented in table 6.1 are quite significant particularly with respect to a time-sensitive application that requires sub-millisecond accuracies. Of course, if such asymmetries were to arise infrequently, then a protocol like NTP would be capable of filtering and discarding this erroneous data. If, however, such traffic profiles persisted then those errors would remain. The protocol would perceive these values as correct and, thus, make an incorrect adjustment to the local

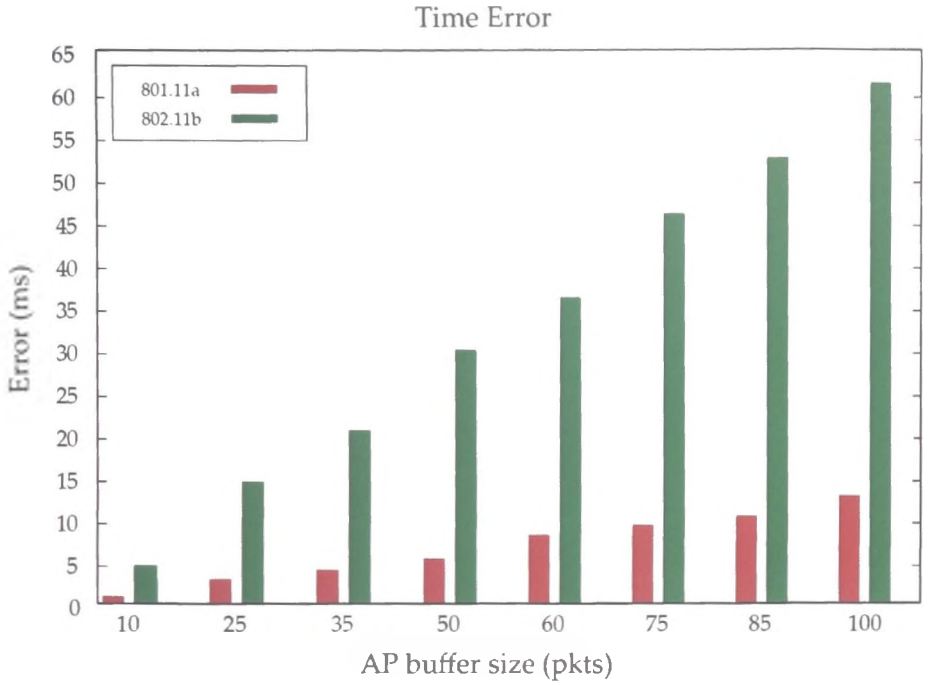


Figure 6.5: Time error

clock. One must also note that the results displayed in fig. 6.3 are average values. Maximum asymmetric delays in the order of 190 milliseconds are observed in the 802.11b simulation with an AP buffer size of 100 packets. A graph illustrating the raw up-link and down-link delays for one of the simulations (802.11b with AP buffer size of 50) is presented at the top of fig. 6.6. A distinctive feature of this graph is the 'sawtooth' pattern formed by the down-link delay values. This is indicative of the TCP congestion control mechanism as it attempts to determine an appropriate transmission rate. One can see that although the average downlink delay is 60ms, delays as high as 85ms are observed. The maximum up-link and down-link delays recorded for each of the simulations is presented at the bottom of fig. 6.6.

The experiments up to this point have focused on the effects of large down-link delays relative to up-link delays. This represents the common scenario in a network whereby down-link traffic (WWW, media streaming traffic, etc.) dwarfs

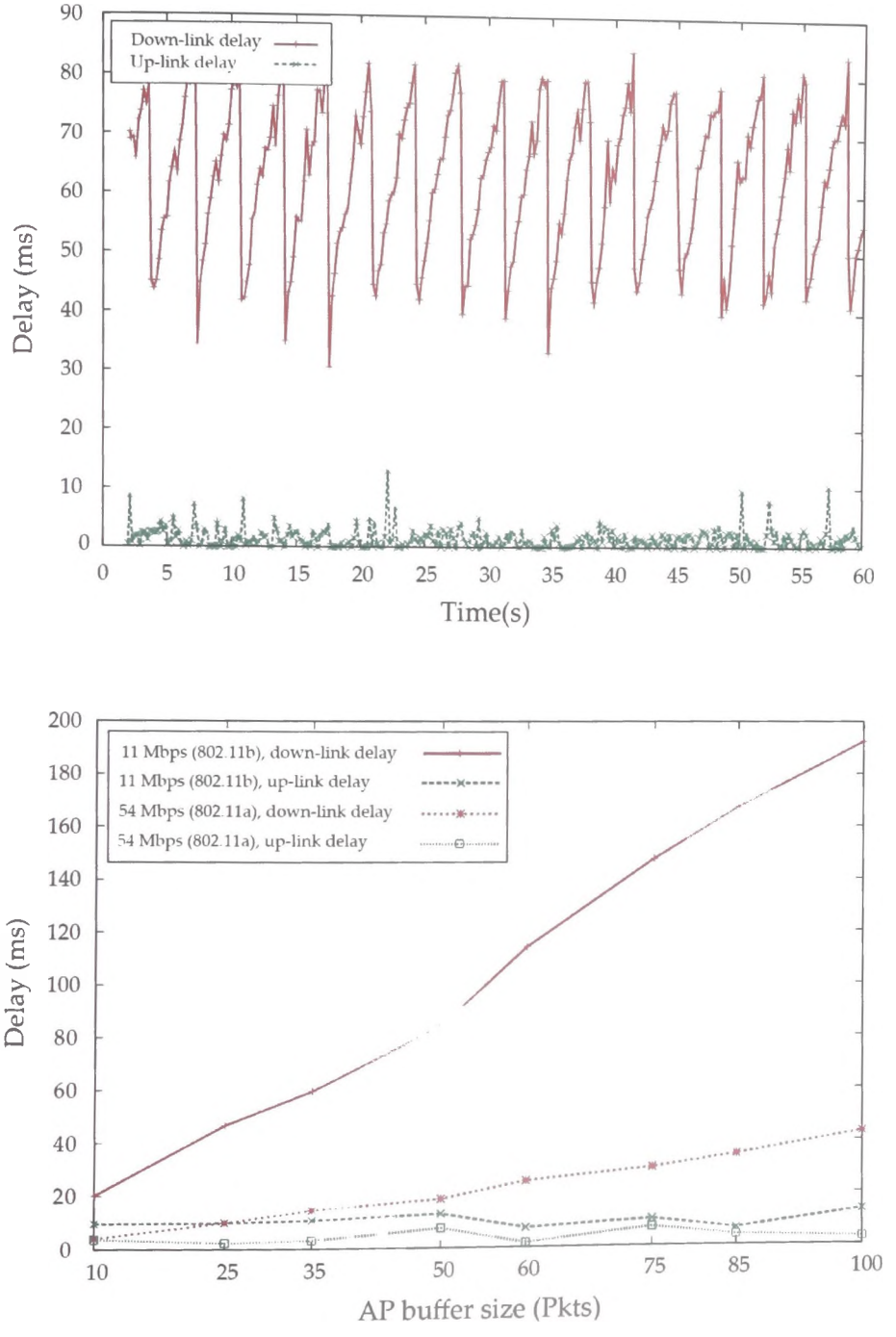


Figure 6.6: Raw delay values (802.11b network with AP buffer size of 50 packets) (top). Maximum up-link and down-link delays (bottom)

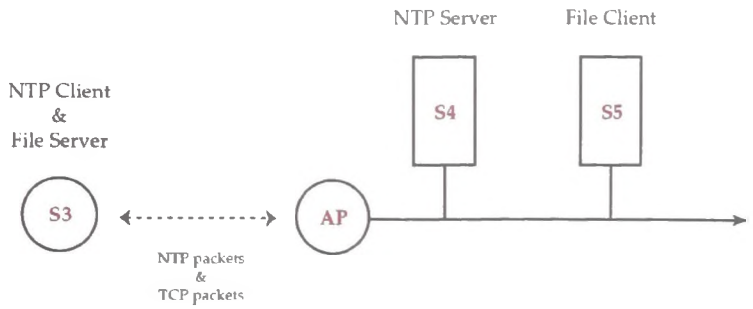


Figure 6.7: Simulation to determine magnitude of up-link delays

up-link traffic. Nevertheless, situations may arise whereby the opposite scenario occurs, that is, the amount of data transmitted by a wireless node far exceeds the amount of data it receives. To determine the impact of such a traffic profile on NTP performance, the experiment illustrated in fig. 6.7 is also simulated. In this experiment, nodes S1, S2, and S6 are removed. The wireless node S3 in addition to hosting an NTP client is also configured to host a file server that uploads a file to the file client S5. Similar to the preceding simulations, the effects of data rate and buffer size are also analysed and, thus, varied over numerous simulations.

The results of these simulations are illustrated in fig. 6.8. The graph displays almost identical characteristics to that of fig. 6.3. These simulations verify that a similar issue in terms of asymmetry can occur when the NTP host's Tx buffer becomes congested, and the AP's buffer remains relatively empty. The result in terms of absolute time error would be similar to the previous case, although as is evident from section 5.3, the NTP offset would be opposite in sign.

Although these simulations are quite thorough with regard to the variety of factors considered, namely data rate and buffer size, data rates below 11Mbps are not considered. As detailed by Li et al. [58], delays as large as 1 second can be observed in lower data rate (1-2Mbps) networks. These are quite significant in comparison to the results obtained here and would prove detrimental to a time protocol assigned the task of providing millisecond accuracies.

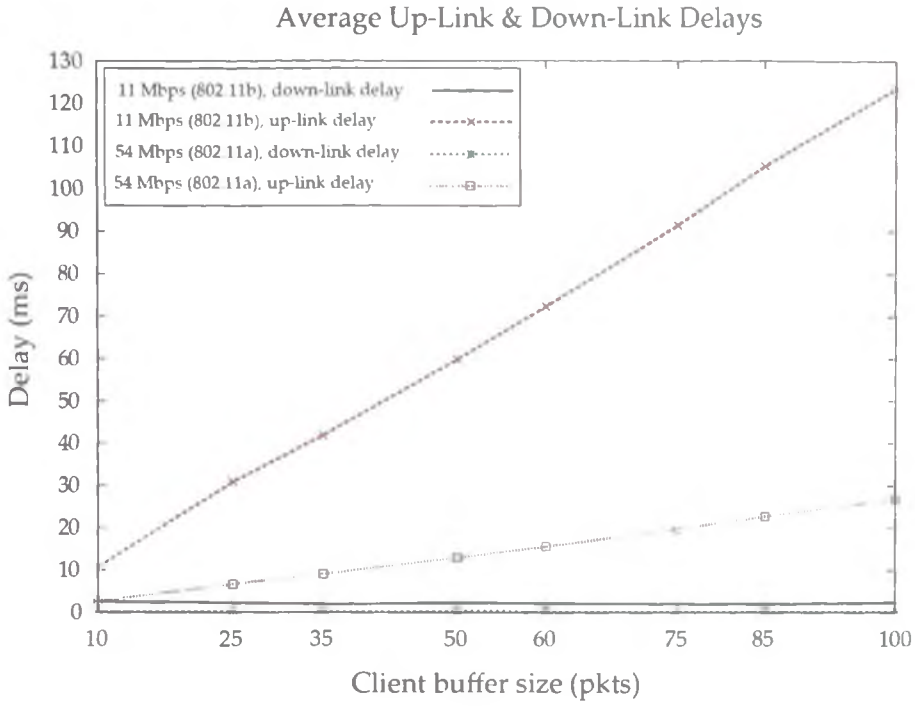


Figure 6.8: Simulation results

6.3 Delay Determination Mechanism

The results in section 6.2.2 clearly show that the effects of asymmetric traffic profiles coupled with a default AP priority in a highly active 802.11 network can have a significant negative impact on the operation of time protocols and, thus, can lead to undesirable time errors at a host. The results of the simulations presented in the previous section confirm the need for a solution where application requirements are more stringent than NTP can achieve.

The objective here is to improve the quality of the dataset that is presented to a time protocol operating on a wireless node. An improved or more accurate dataset in turn permits a protocol to produce a more accurate estimation of its host's clock offset and skew. In terms of the NTP protocol, the task involves making appropriate adjustments to recorded timestamps (see section 6.3.5) before they are processed by the protocol's algorithms. The appropriate adjustments or corrections must be determined by some method. This represents the challenge

and core contribution of this work.

The mechanism proposed in this work determines the appropriate adjustments to NTP timestamps by resolving the up-link and down-link delay of the associated request and reply messages respectively. This is accomplished by accessing lower layers in the network hierarchy and exploiting the operation of 802.11 networks. Clearly, time-stamping NTP packets at lower layers in the hierarchy, similar to PTP, allows for the elimination of errors that result from non-deterministic delays at higher layers. However, unlike PTP, a general purpose computing system does not employ hardware that recognises and automatically timestamps NTP packets. Thus, one must utilise more common facilities.

Fortunately, many modern wireless *NICs (Network Interface Cards)* permit MAC layer time-stamping and packet injection, and most current OSs (Operating Systems) provide libraries with standard APIs that allow higher layer applications to interface with lower layer NIC drivers. Combining these facilities with a detailed knowledge of the 802.11 standard allows one to design a feasible mechanism that can be used to determine/estimate message delays and, thus, significantly improve the degree to which a clock's time can be synchronised over 802.11 networks.

6.3.1 Terminology

To begin some basic terminology must be clarified. The up-link delay and down-link delay are denoted by Δ_U and Δ_D respectively (see fig. 6.9 and fig. 6.11). At the highest level, they refer to those delays associated with the link between a wireless NTP host and its associated AP. Their exact definitions vary depending on the layer within the protocol stack that the transmission and reception event of a time message is time-stamped. With respect to the NS3 simulations presented in the previous section, these timestamps are recorded at the MAC layer at the transmitter and the physical layer at the receiver, respectively (section 6.2.2). These simulations, however, serve to demonstrate and quantify the magnitude of time errors that could be associated with an 802.11 link in times of high traffic loads. In reality, the transmission event of an NTP request message and the reception event of an NTP reply message are time-stamped at the application

layer of the wireless NTP host. The AP does not record a timestamp, but this is dealt with by the proposed mechanism. In any case, one should note that in the subsequent text the definition of both the up-link delay and down-link delay differ slightly from that of the simulations. The up-link delay now represents the time difference between when an NTP request message is time-stamped by the NTP protocol at the application layer and when it is received at the MAC layer of the AP. The definition of the down-link delay represents the time difference between when an NTP reply message is added to the Tx buffer at the AP and received at the MAC layer of the NTP host. Furthermore, while the proposed mechanism is capable of determining an accurate value for the up-link delay of an NTP request message, it is only possible for it to provide an estimate for the down-link delay of an NTP reply message.

6.3.2 Determining Up-link Delays

Determining the up-link delay of an NTP time message entails the combined use of the facilities described earlier. It is possible to place a modern NIC into a mode termed *monitor mode*. In this state an NIC is capable of routing all 802.11 network traffic to an application that requests it. This is performed by means of an operating system's packet capture facility. This packet capture facility can be interfaced with using a standard API provided by a PCAP library as detailed by Garcia [59]. Thus, one can develop a user application that can retrieve and analyse 802.11 frames. In addition to providing an application with raw 802.11 frames, it also provides metadata detailing the size and reception time of the frames as recorded by the NIC driver.

Access to 802.11 frames allows one to exploit the operation of 802.11 networks for the purpose of determining associated delays. As explained in section 5.5.1, the 802.11 standard employs positive acknowledgements to improve the reliability of RF links. The standard specifies that the transmission of a unicast data frame and the reception of its associated *acknowledgement (ACK)* frame is an autonomous operation. Thus, it must complete fully or the transmission is considered a failure. In addition, the time interval between the reception event of a data frame and the transmission event of the corresponding ACK frame is a

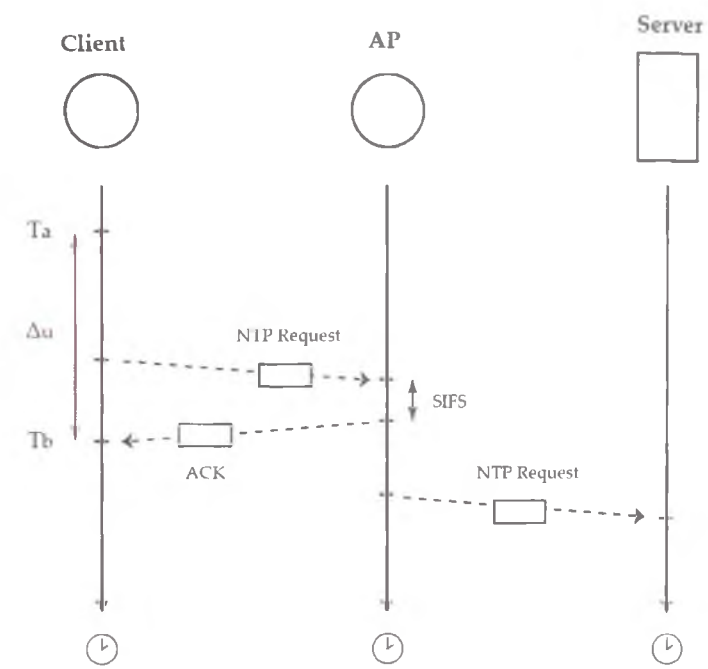


Figure 6.9: Up-link delay determination

constant value termed the SIFS. With this information it is possible to determine the time that an AP receives an NTP request message at its MAC layer. This is achieved by identifying and time-stamping the corresponding 802.11 ACK frame transmitted by the AP back to the NTP host.

This process is illustrated in fig. 6.9. Here the NTP client residing on its wireless host constructs an NTP request message. It then records and stores the timestamp T_a and transmits the message via an operating system call to an NTP server. The message is processed by lower layers and eventually presented to the NIC driver which appends the appropriate MAC headers and adds it to the Tx queue. Once access to the medium is acquired, it is transmitted to the AP. The AP, on receipt, responds with an ACK after an SIFS interval. While this process occurs, another application monitors incoming and outgoing 802.11 frames. When it detects an NTP request message, it acquires the subsequent and associated ACK and retrieves its MAC timestamp, T_b, via the PCAP interface.

The difference between the application layer timestamp, T_a, and the MAC timestamp, T_b, is an approximation of the up-link delay (see equation 6.1). There is some noise in this data due to both latencies incurred during the time-stamping process and propagation delays, but these are negligible in the context of medium access delays.

$$\Delta_{UP} \approx T_b - T_a \tag{6.1}$$

6.3.3 Estimating Down-link Delays

As explained in section 6.1, WWW and media streaming traffic profiles tend to be asymmetric and consist predominantly of downlink traffic. This fact along with the general case, whereby an Access Point has equal priority in terms of medium access, can lead to a bottleneck at the Access Point resulting in greater down-link delays and, thus, NTP synchronisation errors.

In section 6.3.2, up-link delays are determined using a passive approach whereby 802.11 ACKs are used to deduce the approximate transmission times of NTP request messages at the MAC layer and, thus, determine the up-link delay of these messages. Determining down-link delays is a more difficult task since they are the result of queues at the AP. However, by using similar techniques to the ones already outlined in combination with active packet injection, a relatively accurate estimate of the delay incurred by an NTP reply message at an AP can be acquired.

6.3.4 Mechanism

Modern wireless NICs permit packet injection. Thus, it is possible to construct raw 802.11 frames at the application layer and deliver them to an NIC driver via the PCAP interface. The NIC driver subsequently computes the CRC for the raw frame and injects it into the network. This facility, when employed in a particular way, can allow one to estimate delays at an AP. This can be accomplished by constructing a specific data frame at an NTP host. This particular frame when received by an AP is added directly to its Tx buffer for delivery back to the NTP host. By determining the transmission and reception time of this *'crafted'* frame,

a relatively good estimate of the delay incurred by frames at the AP at that particular point in time can be obtained. The key structure of this crafted frame is illustrated in fig. 6.10

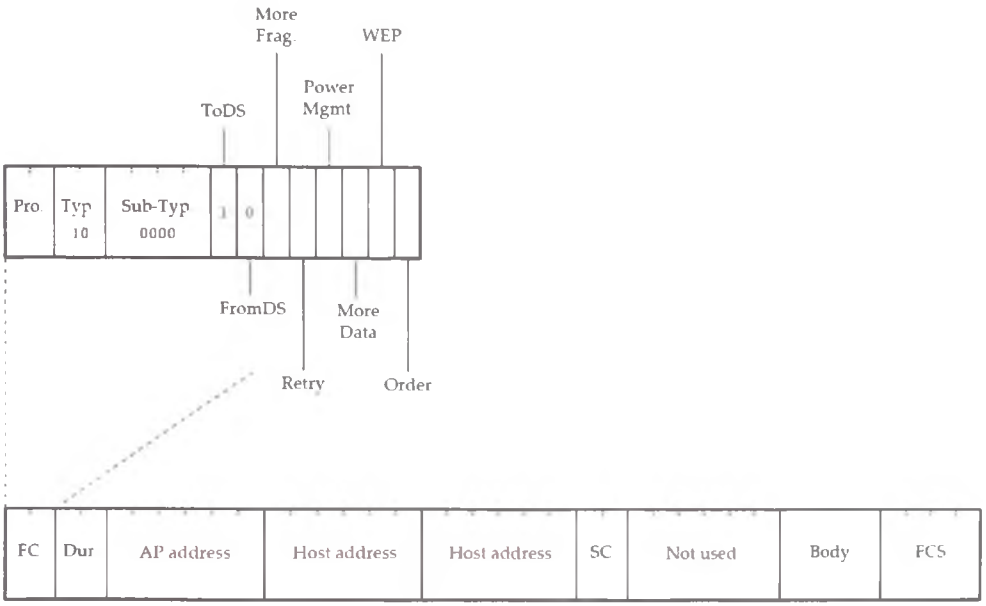


Figure 6.10: Crafted frame

The structure of a general 802.11 data frame is detailed is section 5.5.9. Referring to section 5.5.9, in order to construct an appropriate frame, two key field types must be configured correctly, namely the *frame control* field, and the *address* fields. Since the frame is a data frame, the *type* and *sub-type* subfields of the frame control field should be set to *'10'* and *'0000'* respectively. In addition to this, the *ToDS* and *FromDS* subfields should be set to *'1'* and *'0'* respectively, since an NTP host must transmit it to an AP which is part of the distribution system. When the *ToDS* and *FromDS* fields are configured with these values, the interpretation of the address fields follows that presented in table 5.1. Thus, the first address field (*receiver address*) holds the address of the AP and the second address field (*transmitter address*) holds that of the NTP host. Finally, the third address field (*destination address*) also holds that of the NTP host since the AP must deliver it back to the transmitter.

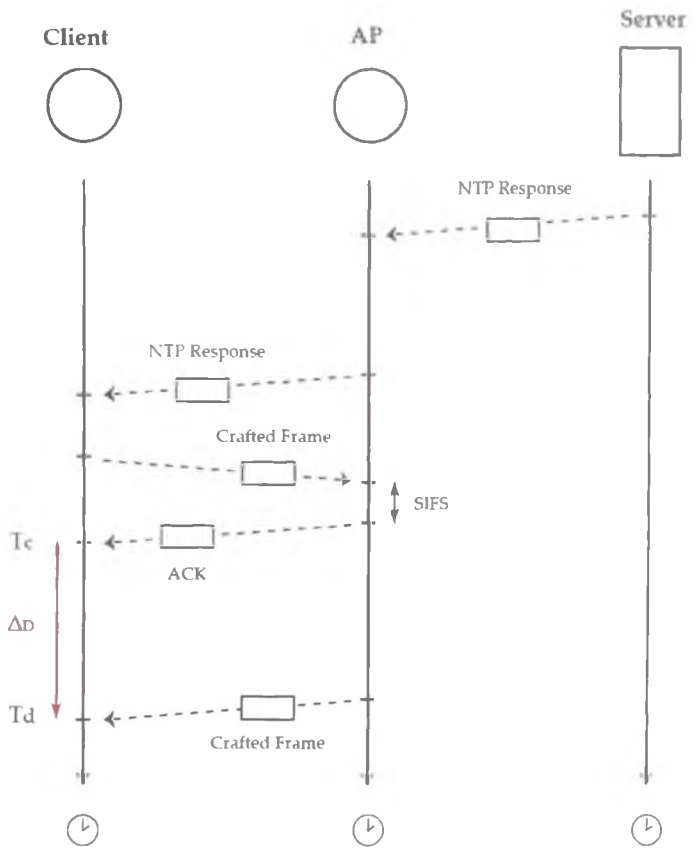


Figure 6.11: Down-link delay estimation

The mechanism used to estimate the queue and access delays of NTP reply messages at an AP is illustrated in fig. 6.11. When an NTP reply message is received by an NTP host, it is detected by a monitoring application. This initiates the immediate injection of the aforementioned crafted frame into the network. The associated ACK returned by the AP is identified and its MAC timestamp, T_c, is recorded. At a later stage the AP returns the crafted frame to the NTP host, as described above. It is detected by the monitoring application which records its associated MAC timestamp, T_d. At this stage, an estimation of the down-link delay associated with the NTP reply message can be calculated using equation 6.2. Since a crafted frame is actively injected after the NTP reply packet is received by a host, it can only be described as an estimate of the down-link

delay, due to the dynamic nature of AP traffic.

$$\Delta_D \approx T_d - T_c \tag{6.2}$$

6.3.5 Timestamp Correction

The two techniques described so far are implemented as an application layer module that runs as a background process continuously determining the up-link delay and estimating the down-link delay of NTP request and reply messages respectively. This information must be used to correct NTP timestamps before they are passed to the NTP daemon running on a wireless host.

The formula used by the NTP protocol to calculate the clock offset of a host is presented in equation 6.3. In 6.3, θ_U denotes the *un-corrected offset* whereby the up-link and down-link measurements have not been incorporated into the equation and, thus, any asymmetry has not been corrected for. If one modifies timestamp T_i by adding the value of the up-link delay, Δ_U, to it one gets TC_i. This represents a more accurate value for the transmission time of the NTP request message, since the delay it encounters before it is transmitted onto the medium is removed. Similarly, if one modifies timestamp T_{i+3} by taking the value of the down-link delay, Δ_D, from it one gets TC_{i+3}. This represents a more accurate value for the actual transmission time of the NTP reply message, since much of the delay encountered by the message at the AP is removed.

When these corrected values are incorporated into equation 6.3, one gets equation 6.6. In equation 6.6, θ_C denotes the *corrected offset*. Altering the equation allows one to express it in the form 6.7. From equation 6.7, one can observe that only one of the timestamps require modification before being processed by the NTP daemon. Thus, the module need only modify one timestamp, T_{i+3}, by adding the result of $(\Delta_U - \Delta_D)$ to it.

$$\theta_U = \frac{(T_{i+1} - T_i) + (T_{i+2} - T_{i+3})}{2} \tag{6.3}$$

$$T_i = TC_i - \Delta_U \tag{6.4}$$

$$T_{i+3} = TC_{i+3} + \Delta_D \tag{6.5}$$

$$\theta_C = \frac{(T_{i+1} - (T_i + \Delta_U)) + (T_{i+2} - (T_{i+3} - \Delta_D))}{2} \tag{6.6}$$

$$\theta_C = \frac{(T_{i+1} - T_i) + (T_{i+2} - (T_{i+3} - \Delta_D + \Delta_U))}{2} \tag{6.7}$$

6.3.6 Implementation

As stated in the previous sections, the aforementioned mechanism is implemented as an application layer software module. The module, which is composed of a packet *sniffer* and *injector*, is implemented in plain C. The motivation for employing plain C is one regarding speed and efficiency. In order for a sniffer to detect both application layer NTP packets and their associated MAC layer ACK frames, it is necessary to monitor all traffic transmitted and received on a wireless interface. Thus, all 802.11 frames received via the *pcap* interface must be analysed. In times of high traffic loads, this may mean analysing thousands of frames per second. In order for the module to stay responsive and cope with such demand, it should execute a minimum number of instructions when analysing frames and, thus, any language that might present unnecessary overhead through additional but unnecessary features (garbage collection) is not be suitable. Additionally, the libpcap library is designed to be used with C and C++. Its use with another language, such as *Java* or *Perl*, requires a wrapper which presents further overhead. Finally, in this work, the module is used in conjunction with the NTP daemon which itself is developed in C. Developing the module in C rather than C++ allows for natural integration of the module with the NTP daemon if required.

While there are many performance benefits to developing such a module in plain C, there are disadvantages in terms of the development process itself. Programming large applications in a *procedural* language as opposed to an *object orientated (OO)* language can be more difficult and typically requires additional lines of code which in turn increases the risk of producing *"buggy"* code. With

respect to the work in this thesis, it is vital to decrease the chance of producing erroneous code that might affect the outcome of subsequent experiments. Thus, it is preferable to obtain the performance benefits of plain C while still possessing the development benefits of an OO language. In this respect, the module is implemented in C but developed in an OO fashion.

The development of objects in plain C is achieved through the use of *struct* constructs in conjunction with function pointers. An example of a *String* object developed in this manner is presented in figures 6.12, 6.13 and 6.14.

Fig. 6.12 presents the contents of a *Sting.h* file which defines a *String* structure and declares the function prototypes used by an initialised *String* 'object'. Figures 6.13 and 6.14 present snippets from a *String.c* file which contain the implementation details for the function prototypes declared in *Sting.h*. The most interesting function is the *newString* function which is responsible for returning instances of *String* objects. This function allocates any required memory from the heap and assigns declared function pointers to corresponding function definitions. Another interesting function is the *String_free* function which de-allocates memory used by a *String* object, thus, destroying it. This eliminates the need for a garbage collector but places responsibility on a programmer to explicitly destroy objects. Finally, the bottom of fig. 6.14 illustrates the code required to instantiate, utilise, and destroy a *String* object.

Using this approach, the application is developed in a modular fashion and each module/object is tested individually, minimising the chance of producing erroneous code. The core objects constructed in this manner include:

- *LinkedList* - A generic collection object used to store other objects in a linked list
- *TimeTools* - An object with a set of functions for manipulating, converting, and formatting NTP timestamps
- *IEEE80211RadioTap* - An object for analysing 802.11 RadioTap headers
- *IEEE80211* - An object for analysing 802.11 headers
- *LLC* - An object for analysing Logical link control headers

```
typedef struct String {

        //Variables (Object Attributes)
        char * value;
        int length;
        char * currentpos;

        //Function pointers (Object Methods)
        void ( * set ) ( struct String *, char * );
        void ( * append ) ( struct String *, char * );
        int ( * equals ) ( struct String *, char * );
         int ( * same ) ( struct String *, struct String * );
        int ( * compare ) ( struct String *, struct String *);
        int ( * hasMoreInt ) ( struct String * );
        int ( * getNextInt ) ( struct String * );
        void ( * free ) ( struct String * );
        int ( * comparator )( void *, void *);
        void ( * freer )( void * );

} String;


//Function prototypes
String * newString( char * );
void String_set( String * , char * );
void String_append( String *, char * );
int String_equals( String *, char * );
int String_same ( String *, String * );
int String_compare ( String *, String * );
int hasMoreInt( String * );
int String_getNextInt( String * );
void String_free( String * );
int String_comparator( void *, void * );
void String_freer( void * );
```

Figure 6.12: Definition of a *String* object using C *structs* and function pointers

```
String * newString( char * value ) {

        //Allocate memory for 'String' struct (String Object)
        String * s = NULL;
        s = ( String * ) malloc( sizeof( String ) );

        //Initialise variables with default values
        s->value = NULL;
        s->length = 0;
        s->currentpos = NULL;

        //Initialise function pointers
        s->set = String_set;
        s->equals = String_equals;
        s->same = String_same;
        s->append = String_append;
        s->compare = String_compare;
        s->hasMoreInt = String_hasMoreInt;
        s->getNextInt = String_getNextInt;
        s->free = String_free;
        s->comparator = String_comparator;
        s->freer = String_freer;

        //Set specified value
        s->set( s, value );
        return s;
}
```

Figure 6.13: Instantiation of a *String* object

```
//Sets the contents of a 'String Object' to a specified value
void String_set( String * s, char * value ) {
        int size = strlen( value );
        if( s->value == NULL ) {
                s->value = ( char * ) malloc( sizeof( char[size+1] ) );
                strcpy( s->value, value );
        }
        else if( s->length != size ) {
                free( s->value );
                s->value = ( char * ) malloc( sizeof( char[size+1] ) );
        }
        strcpy( s->value, value );
        s->currentpos = s->value;
        s->length = size;
}


//Free up memory used by a 'String Object'
void String_free( String * s ) {

        if( s->value != NULL )
                free( s->value );

        free( s );
}
```

```
//Create a new 'String Object'
String * s = newString( "Hello" );

//Alter contents of 'String Object'
s->set( s, "Bye" );

//Destroy 'String Object'
s->free( s );
```

Figure 6.14: *String* object method implementation snippet (top). *String* object instantiation, employment, and destruction (bottom).

- *IP* - An object for analysing IP headers
- *UDP* - An object for analysing UDP headers
- *NTP* - An object for analysing NTP headers
- *IEEE80211RadioTapCrafter* - An object for crafting 802.11 RadioTap headers
- *IEEE80211Crafter* - An object for crafting 802.11 headers
- *LLCCrafter* - An object for crafting Logical link control headers
- *Stat* - An object for producing statistics from a list of numerical data
- *GnuPlotGrapher* - An object for producing various types of gnuplot generated graphs
- *NTPData* - An object for storing all timestamps associated with an NTP exchange, including the timestamps of associated crafted packets and ACKs. This object also contains functions that calculate the associated up-link and down-link delays.

The application itself is developed using the aforementioned objects and the *libpcap* library. The *libpcap* library provides a function called *pcap_loop* with the signature *pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)*. The function allows a user application to provide a handler function which is called whenever the interface captures an incoming or outgoing packet. This handler function is passed a pointer to a byte array which holds a copy of the captured packet. The flowchart in fig. 6.15 illustrates the process flow on receipt of a captured packet/frame from a wireless interface. The frame is first analysed by decoding the various protocol headers. On receipt of an *NTP Request*, *NTP Response*, *crafted*, or *ACK* frame, the respective process flows illustrated in figures 6.16 and 6.17 commence. The reception of any other packet type completes the process until the next packet capture.

Fig. 6.16 illustrates the process flow on receipt of an *NTP Request* packet. All timestamps associated with an NTP exchange are stored within an *NTPData*

object. Since an NTP client may poll multiple servers, each NTP association is assigned a distinct *NTPData* object instance. Each *NTPData* object is stored in a linked list. When an *NTP Response* packet is detected, the list is searched for the relevant *NTPData* object, and it is updated. If an *NTPData* object does not exist for the particular NTP association, then a new one is created and appended to the list. In addition to this, a reference to the object is created so that the list does not need to be searched when the NTP Response packet's associated 802.11 ACK is received (see fig. 6.17).

Fig. 6.16 also illustrates the process flow on receipt of an *NTP Response* packet. In this case, the list is searched again, and the relevant *NTPData* object is retrieved and updated. Subsequently, a frame is crafted in accordance with section 6.3.4 and injected into the network.

Fig. 6.17 illustrates the process flow on receipt of the *crafted* frame from the AP as well as its corresponding ACK. On receipt of the crafted frame's associated ACK, all relevant *NTPData* objects in the list are updated. Relevant objects are those that pass specific validity checks and have an *NTP Response* Rx timestamp associated with them. Subsequently, a *timestamp file*, containing modified $T4$ timestamps (in accordance with equation 6.7) for each NTP association, is updated using the respective *NTPData* objects.

Communication between the module and the NTP daemon is carried out through use of the timestamp file. The NTP daemon process, on receipt of an *NTP Response* packet from a peer, refers to this file for any available corrections to the $T4$ timestamp. This requires some modifications to the NTP code base, in particular the *ntp_proto.c* file. Here a new function named *correctT4* is defined and called within the NTP function *process_packet* which among other things is responsible for calculating the peer offset. The flow of the function *correctT4* is illustrated in fig. 6.18. The function iterates through each line in the timestamp file searching for corrections to the $T4$ timestamp associated with the packet been currently processed. If a correction is found, then the offset is calculated using the corrected $T4$ timestamp. If not, then the process sleeps for a number of milliseconds. A number of attempts are made to find a relevant correction, and if not successful then the function completes and the original $T4$ timestamp is employed.

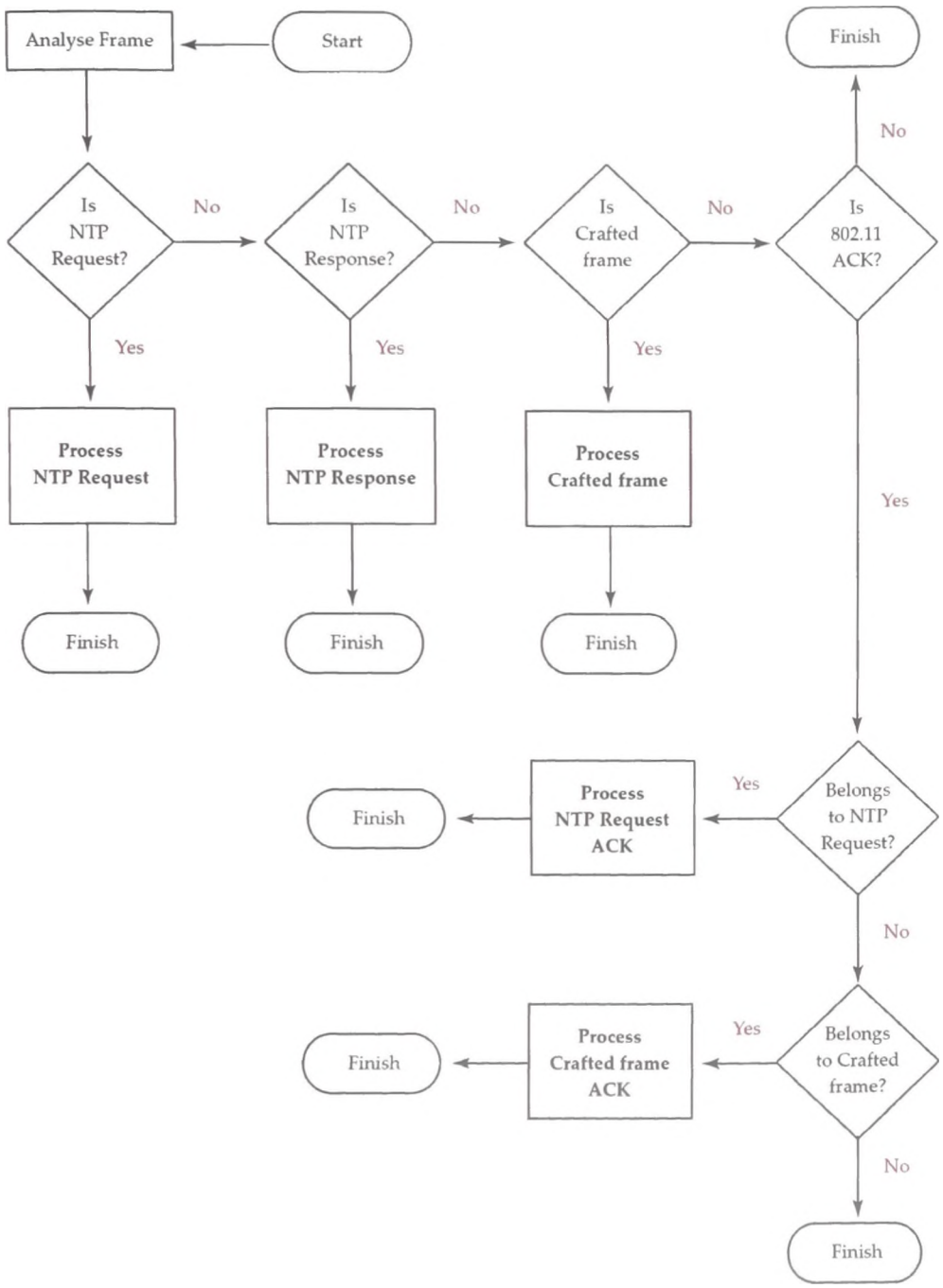


Figure 6.15: High level process flow on receipt of a captured packet/frame from the wireless interface

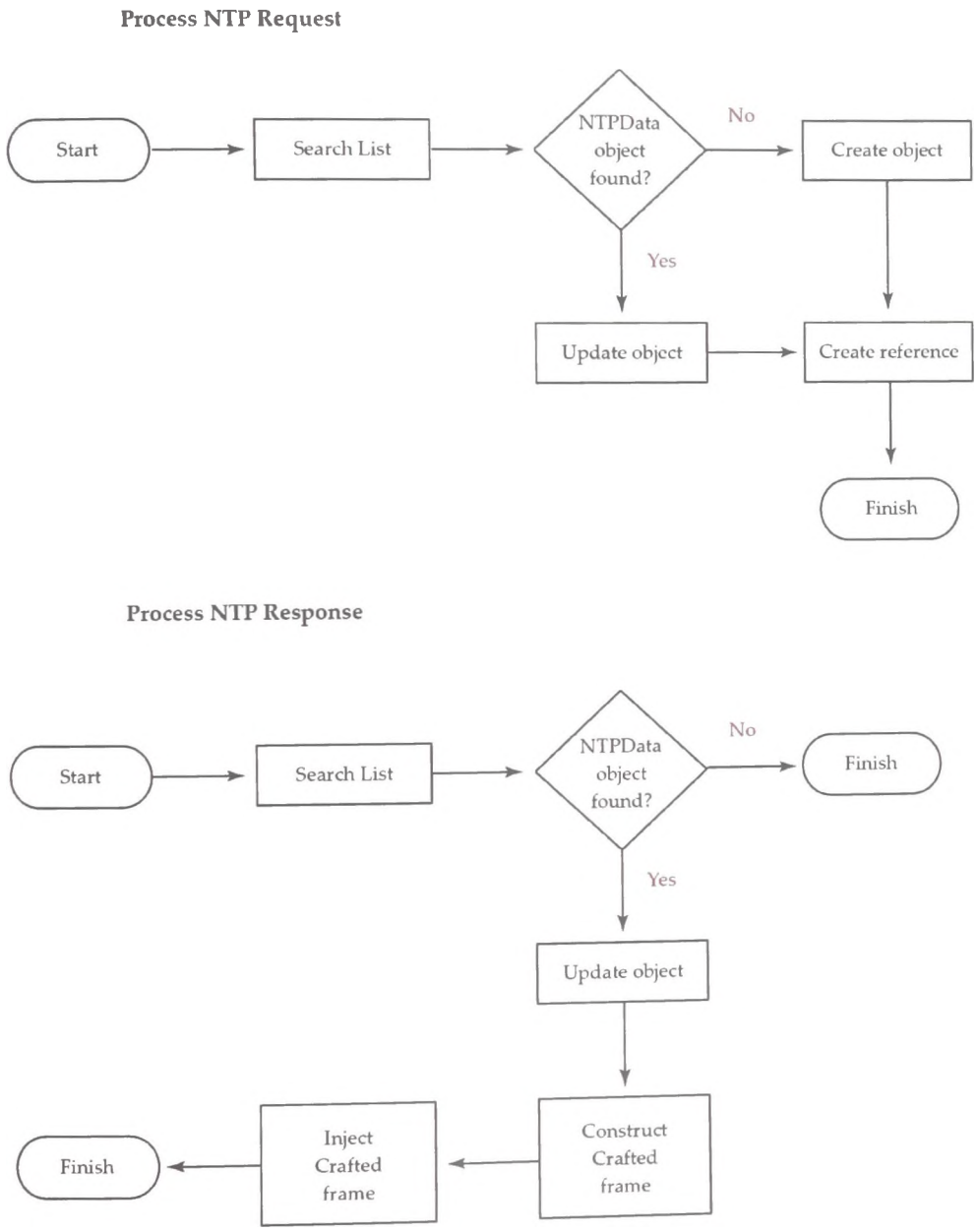


Figure 6.16: Process flow on receipt of NTP request and response packets

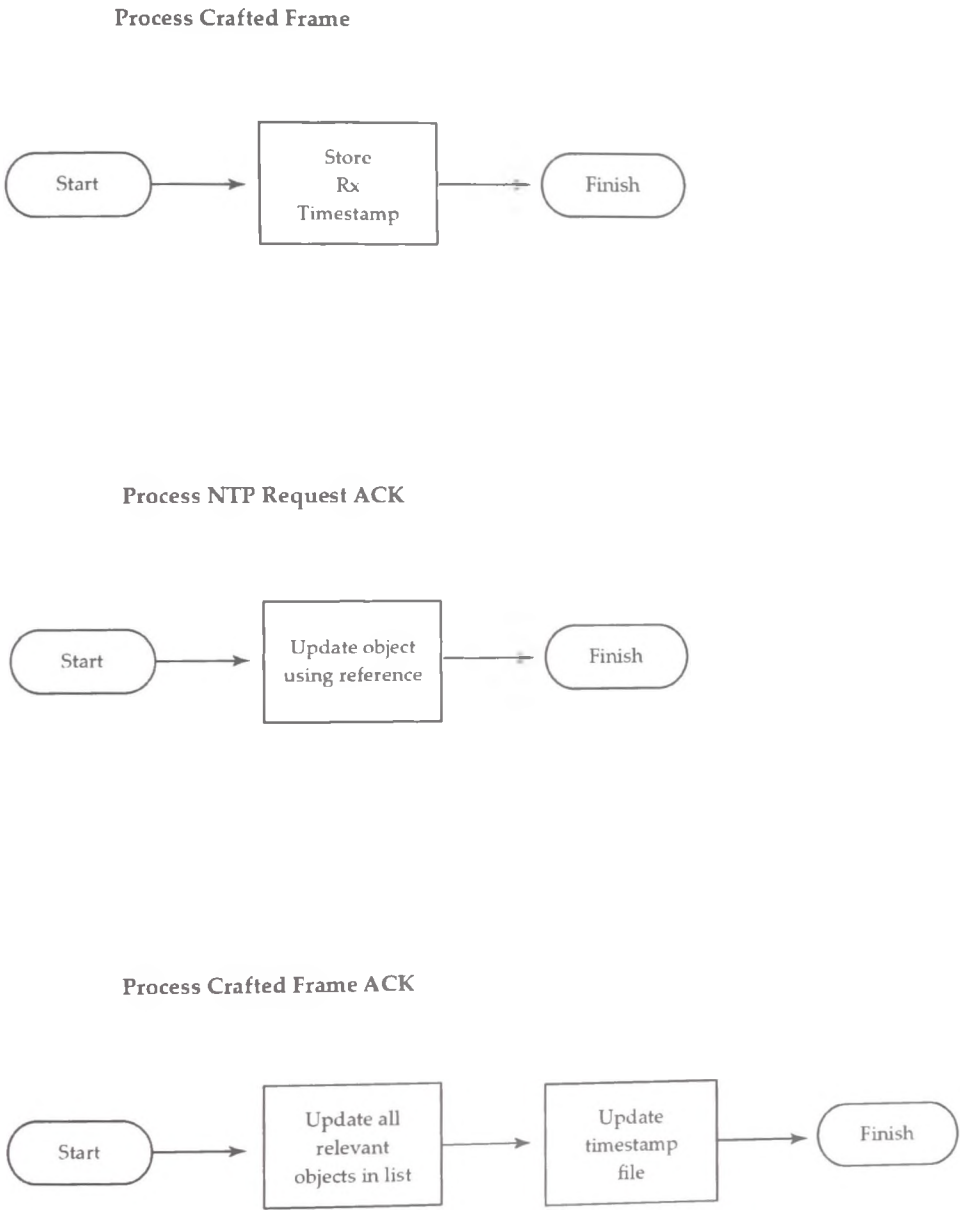


Figure 6.17: Process flow on receipt of crafted frame and relevant ACKs

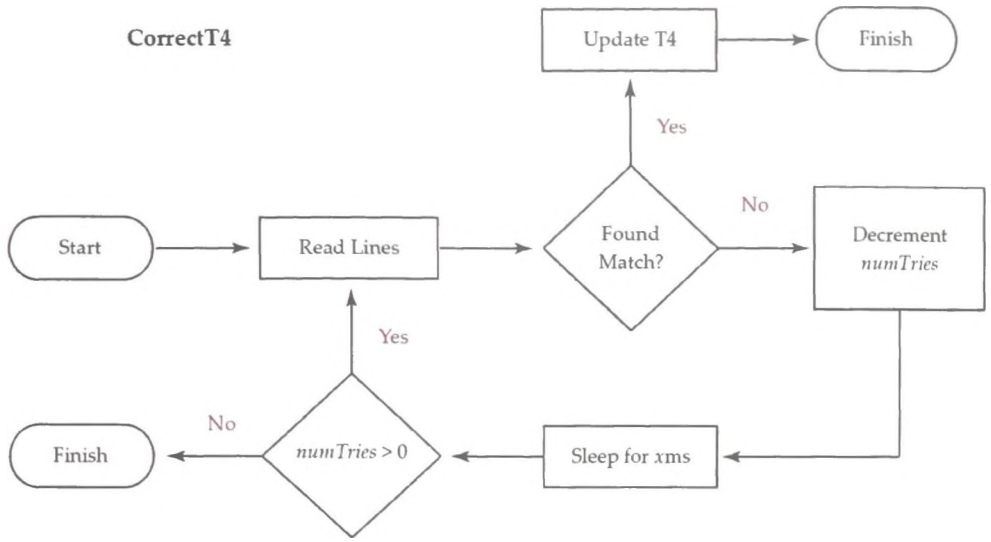


Figure 6.18: *CorrectT4* function flow

The rational for searching the file multiple times with sleep intervals in between is one related to process scheduling and system delays. Since the module runs as a separate process relative to the NTP daemon, there is no guarantee that it can receive an NTP Response packet and carry out all subsequent operations before the daemon refers to the timestamp file. Thus, the module is given a relevant amount of time to carry out these operations by placing the NTP daemon into numerous sleep states.

6.4 Mechanism Validation

6.4.1 Testbed Overview

The previous sections outline a mechanism to significantly increase the quality of temporal data employed by time synchronisation protocols that reside on wireless hosts operating in busy and contentious 802.11 networks. The mechanism is tested in a real wireless environment in order to determine its effectiveness. To do this the experimental testbed illustrated in fig. 6.19 is deployed.

The testbed is somewhat similar to that illustrated in fig. 6.2 for the simula-

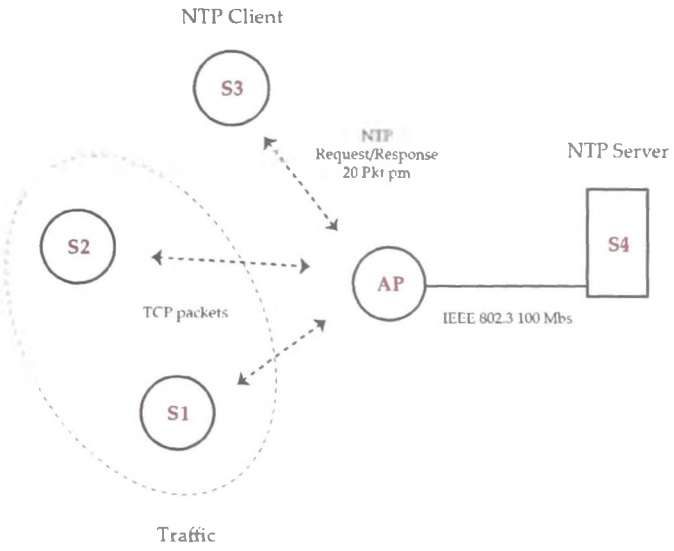


Figure 6.19: Validation testbed

tions but with a number of modifications. The wired component of the testbed consists of a single station, $S4$, which hosts an NTP server and is connected directly to the AP via a 100Mbps wired link. The wireless component of the testbed consists of 3 wireless clients $S1$, $S2$, and $S3$ which together with their associated AP represent a small BSS. Wireless station, $S3$, hosts an NTP client that transmits NTP request packets to $S4$ at a rate of 20 packets per minute. The NTP server hosted by $S4$ responds to each NTP request packet with a corresponding NTP response packet. It should be noted that the NTP transmission rate in this testbed is much greater than that which could be used in a real world deployment. NTP only permits a maximum configurable rate of 3 packets per minute. In this testbed it is purposefully set to a higher rate in order to increase the amount of data collected during the experiment. Nevertheless, it is much lower than the rate used in the simulations presented in section 6.2.2. The simulations, however, only model simple network elements and traffic and, thus, the rate can be set higher without impacting the results of the experiments.

Similar to the design of the simulation experiment detailed in section 6.2.1, the objective here is to recreate delay asymmetries typical of real-world conditions in order to test the effectiveness of the mechanism. To accomplish this, station

S1 hosts a traffic generator. The traffic generator operates by first opening up a TCP connection with station $S2$. It subsequently initiates the transmission of raw TCP packets (header only) to $S2$ at a rate of 1500 packets per second. $S2$ responds to each TCP packet with a corresponding TCP acknowledgement packet. Since all traffic generated by $S1$ and $S2$ is routed through the AP, the wireless Tx buffer at AP remains relatively full producing the desired scenario.

It should be noted that the reason that this testbed differs from that used in the simulations (see section 6.2.1) is for control reasons. One of the advantages of using a simulator to run an experiment is that certain attributes of the network can be set to unrealistic values. This allows one to eliminate factors that might add noise to results. In the simulation, the wired link connecting the AP to the servers is configured such that NTP packets do not incur relatively significant delays while traversing the wired link. In a real testbed, this is not possible so in order to ensure NTP packets do not incur significant delays over the wired link, the real-world experiment is redesigned so that no additional traffic traverses the wired link. Thus, all non-NTP traffic in the real-world testbed is generated within the wireless component of the network.

With respect to the hardware employed in this testbed, all stations excluding the AP run the *Ubuntu* distribution of the *Linux OS*. Each of the wireless nodes employ an *802.11g* compliant wireless network interface card (NIC). The mechanism/module runs on $S3$ and, thus, requires an NIC capable of packet *sniffing* and *injection*. Therefore, $S3$ employs an NIC with an *Atheros* chipset which is run by the *ath5k* driver and provides the required functionality. In relation to traffic generation at $S1$, the *hping3* packet generator is employed. Its interface allows one to specify a TCP packet transmission rate. With regard to the chosen rate of 1500 packets per second, this value is chosen carefully via trial and error so that the AP's buffer remains relatively full while still coping with the traffic load.

Finally, this testbed focuses on generating large down-link delays between $S3$ and the AP. A testbed that focuses on the alternative scenario of larger up-link delays is considered less important for two reasons: firstly, in a real world scenario down-link delays typically dominate; secondly, with respect to the mechanism detailed in previous sections, the task of determining the down-link delay of time

messages is more challenging than determining their up-link delay and, thus, if successful then it adequately highlights the module's capability.

6.4.2 Experiments and Results

An experiment is run for a duration of 1 hour. It is initiated by the NTP client at time T_0 which begins transmitting NTP request packets to the NTP server. The server responds with NTP response packets. Non-NTP traffic is not introduced until time T_1 which represents the 16th minute. The traffic generation continuous until the experiment completes at time T_2. The traffic produced by $S1$ and $S2$ is illustrated in fig. 6.20.

Non-NTP Traffic is not introduced into the experiment until a later time so that the clock skew between the NTP client's host and the NTP server's host can be calculated. Thus, the clock offset between $S3$ and $S4$ is recorded between times T_0 and T_1. These values are then used to determine their relative clock skew using linear regression. Provided that both clocks remain stable, which is a reasonable assumption given that their temperature remains stable, the relative clock skew can be used to determine their *true relative clock offset*, denoted θ_T, at a future point in time. It is important to note that although the NTP protocol is employed in this experiment the NTP algorithms are disabled at the client so that the delay determination mechanism can be validated. Thus, the clock offset and clock rate are not modified by the NTP daemon. In addition, the NTP server at $S4$ does not synchronise with another NTP server, and so its clock is not altered by the protocol.

When a value for θ_T is obtained at T_1, the network is loaded with TCP traffic. The offset values produced by the NTP client with and without the aid of the module are logged. Offset values produced without the aid of the module are termed *un-corrected offsets* and are denoted by θ_U, whereas those produced with the aid of the module are termed *corrected offsets* and denoted by θ_C. These values are collected from time T_1 to T_2. The absolute difference between the true offset, θ_T, and the un-corrected offset θ_U is denoted by ϵ_U and is calculated using equation 6.8. The absolute difference between the true offset, θ_T, and the corrected offset, θ_C, is denoted by ϵ_C and is calculated using equation 6.9.

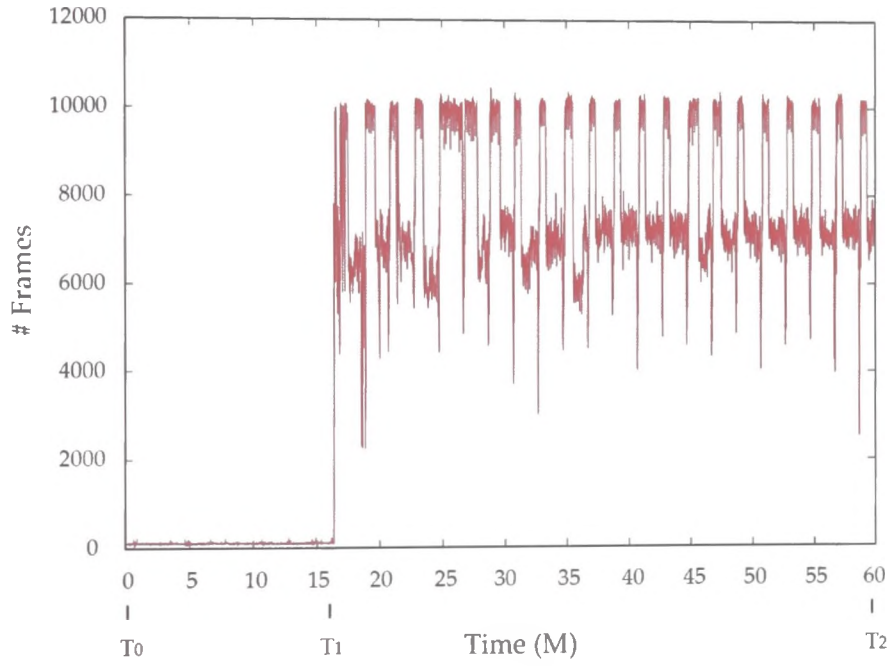


Figure 6.20: Traffic - number of 802.11 frames detected on network during experiment

$$\epsilon_U = |(\theta_T - \theta_U)| \tag{6.8}$$

$$\epsilon_C = |(\theta_T - \theta_C)| \tag{6.9}$$

The results of the experiment are presented in fig. 6.21. The graph in fig. 6.21 is a combination of the results displayed in figures 6.22, 6.23. It 6.24 and plots the values θ_T, θ_C. and θ_U for the duration of the experiment. In addition, it presents the average (μ), maximum, and standard deviation (σ) of the errors ϵ_U and ϵ_C.

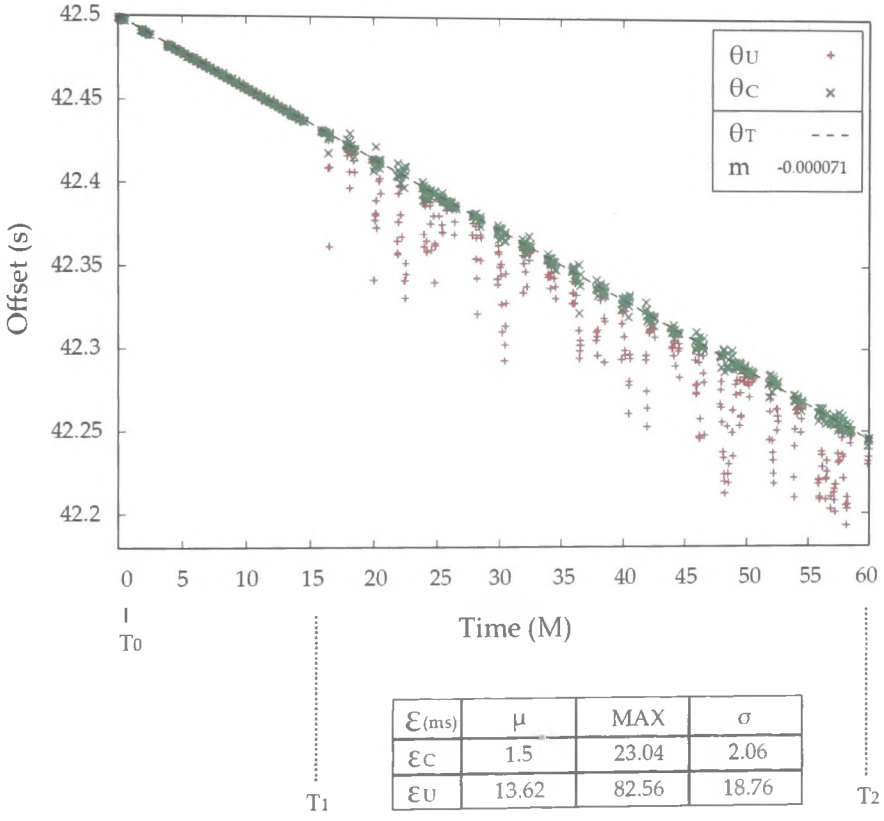
The first interesting feature of the graph illustrated in fig. 6.21 is the line produced by θ_T. The line has a clear slope which indicates that $S3$'s clock operates at a different frequency to $S4$'s clock, that is, there is a relative skew offset. The slope of the line represents the magnitude of the frequency difference/error which is equal to 71ppm. This is equivalent to a rate of 6.13 seconds per day, a value not atypical of computer clocks that employ relatively cheap quartz crystal oscillators.

Prior to time T_1, the point at which non-NTP traffic is introduced into the network, each offset value represented by θ_U and θ_C is, in terms of the degree of synchronisation under focus, accurately determined by the NTP protocol and has minor or negligible error. However, once traffic is introduced at T_1, the offset values corresponding to θ_U deviate considerably from θ_T indicating significant error. In contrast, the offset values corresponding to θ_C deviate far less.

A quantitative analysis of the results indicates that, in this particular scenario, the mean error (μ) produced with the module is 90% less than the mean error produced without the module as it is reduced from 13.6ms to 1.5ms. In addition to this, the maximum error is reduced from 82.5ms to 23ms, and the standard deviation (σ) of the error is reduced from 18.7ms to 2ms, a 60% and 90% reduction respectively. The standard deviation of the errors ϵ_U and ϵ_C are illustrated in fig. 6.25 and fig. 6.26 respectively.

6.4.3 Limitations

The mechanism detailed in this work, while validated above, has some limitations of use that need to be understood. The first and foremost limitation is one con-



$\mathcal{E}_{(ms)}$	μ	MAX	σ
ε_C	1.5	23.04	2.06
ε_U	13.62	82.56	18.76

Figure 6.21: Results - offsets (θ_T, θ_U and θ_C) and errors (ϵ_U and ϵ_C)

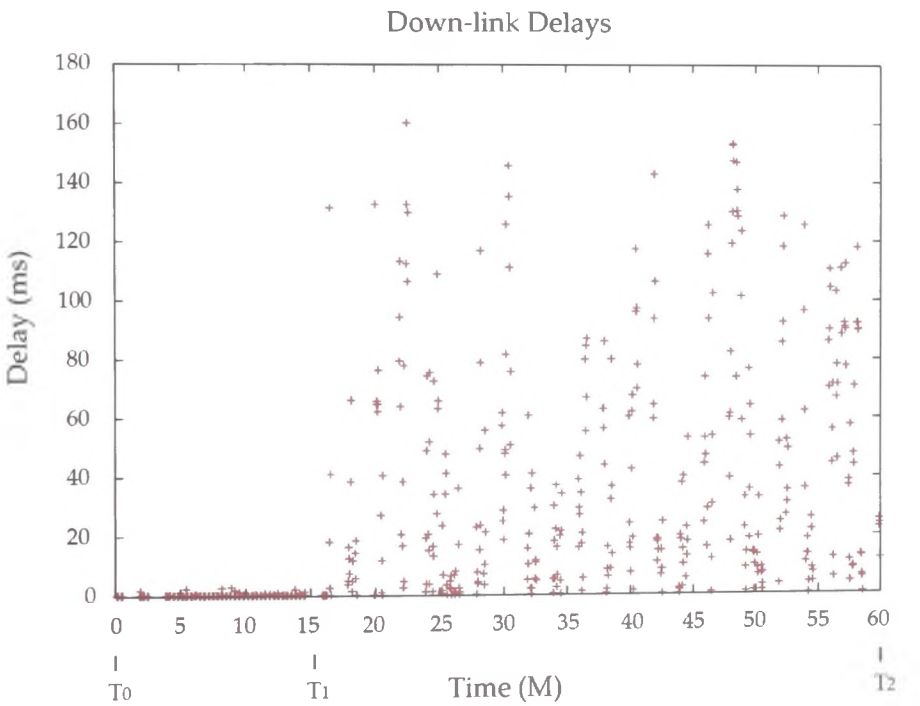


Figure 6.22: Down-link delays

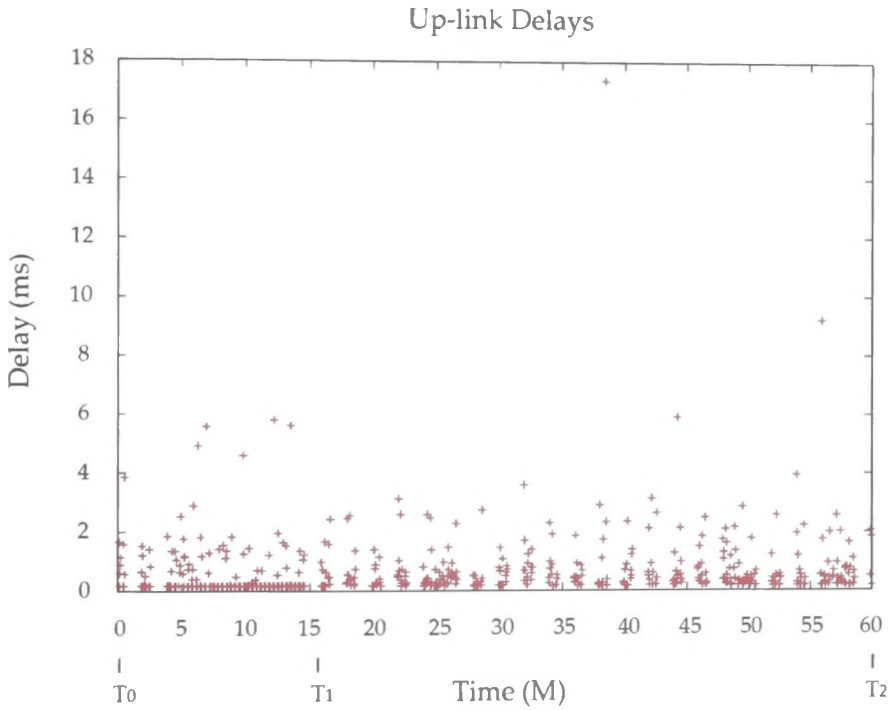


Figure 6.23: Up-link delays

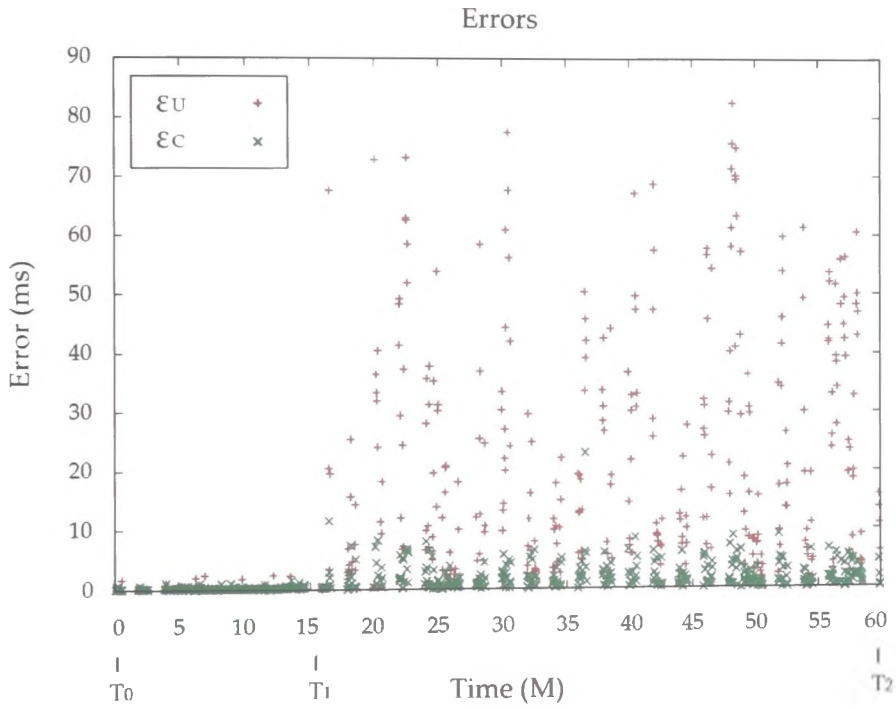


Figure 6.24: Errors with (ϵ_C) and without (ϵ_U) module

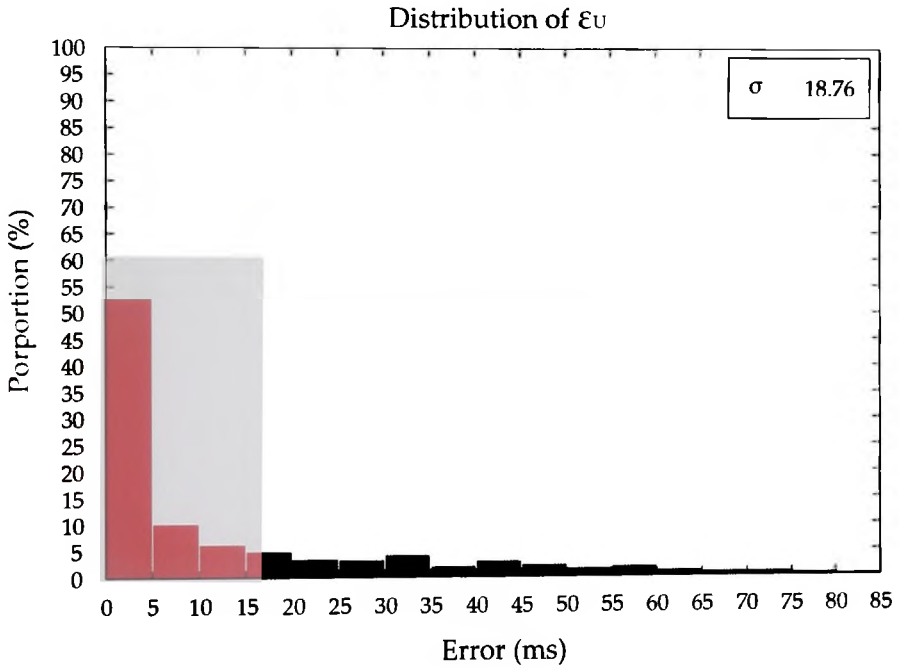


Figure 6.25: Distribution for ϵ_U

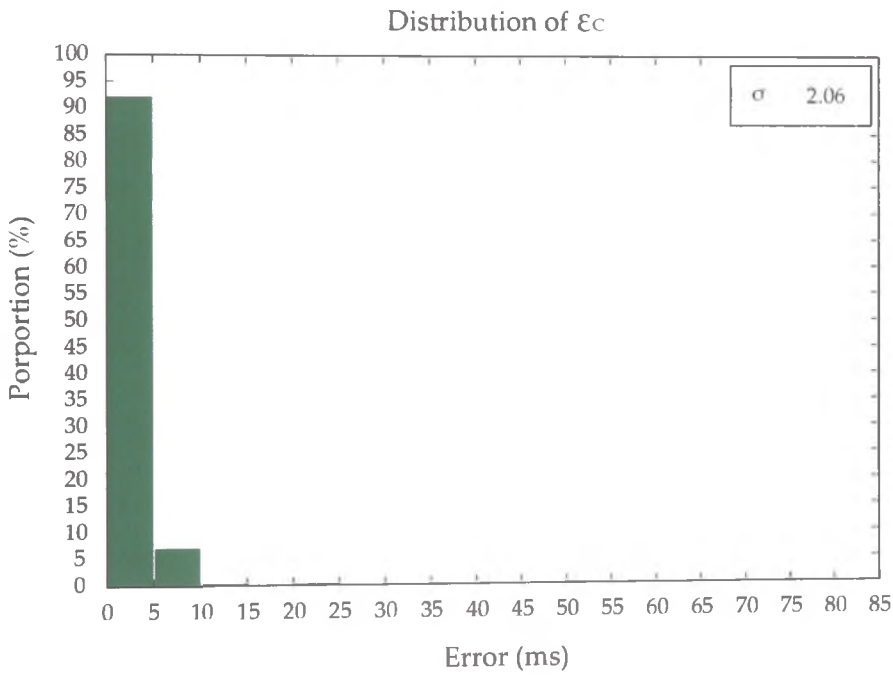


Figure 6.26: Distribution for ϵ_C

cerning the availability of appropriate hardware. While many chipsets currently on the market are natively capable of providing the required functionality, they may be restricted by the driver that runs them. Thus, in general it is the driver that must support *"monitor"* mode and be capable of packet *"injection"*. The Linux OS supports many of these chipsets. Out of 127 chipset variants known to be supported by Linux, 72% of them support monitor mode while 87% of those support packet injection. Thus, 63% of the chipsets support both mechanisms. Those chipsets are employed by 17 adapter manufacturers (*Atheros, Broadcom, Intel, Intersil, Ralink, Realtek, ST-NXP, ZyDAS*) 8 of which provide adapters that possess the required functionality. This is a large proportion and, thus, the hardware and software limitations are not so much a limitation in a Linux environment but rather other environments such as Windows. Details regarding Windows support are difficult to find, yet it is probably fair to say that it does not provide the support to the same extent as Linux.

Regardless of this limitation, the benefits of this mechanism with regard to synchronisation improvement gains justify the effort involved in purchasing the appropriate adapter in environments where time-sensitive applications or applications that can benefit from time synchronisation are deployed. As outlined in section 5.1, a wide range of such applications either exist or are emerging, and it is reasonable to assume that as the demand for improved synchronisation over wireless increases, manufactures will respond by employing appropriate chipsets in their adapters as well as providing appropriate drivers.

Another limitation of this mechanism is one concerning wireless encryption. The mechanism does not handle encryption, and the experiments presented in section 6.4.2 are performed in an open access network. Such an issue can be overcome if the employed driver provides an interface that permits one to specify that an injected packet should be encrypted before transmission. The implementation of such a facility would be trivial and could be provided with ease if required.

6.4.4 Summary and Contribution

This chapter detailed the design, operation, and validation of a mechanism that improves the performance of time synchronisation protocols operating on 802.11

based wireless hosts. The extent to which highly active 802.11 networks can degrade the effectiveness of time synchronisation techniques, in particular round-trip synchronisation, was quantified via appropriate experimental designs and associated simulations. The results of these simulations indicated that the data rate and buffer size of 802.11 nodes directly influence the magnitude of delays in such networks. Thus, in high traffic, low data rate networks composed of nodes that possess large buffers, these delays can be quite significant and can lead to considerable time errors with respect to time-sensitive applications that require single millisecond accuracies. It is also noted that reducing the buffer size of nodes whilst reducing delays does so at expense of increased packet loss.

The design of a mechanism used to combat this issue was described in detail. The mechanism consists of two components that leverage facilities provided by current wireless NICs and exploit particular 802.11 protocol operations in order to determine the up-link and down-link delays of time messages respectively. These values can subsequently be used to improve the temporal dataset employed by time protocols, such as NTP, thereby improving their performance.

The verification and quantification of this mechanism's effectiveness in a real-world scenario was performed via experimentation in a real-world testbed. The results indicated significant improvement gains with as much as a 90% reduction in the mean error of temporal data when the mechanism was employed. In addition, reductions in the order of 60% and 90% for the maximum error and standard deviation of the error respectively were observed.

The experimental results clearly highlight the benefits of employing this mechanism in a wireless network. Its use in conjunction with a time synchronisation protocol can result in considerable improvements in accuracies. This significant mitigation of time error makes possible the successful migration of several time sensitive applications from the wired domain to the wireless domain (see section 5.1). In addition, it opens up the possibility for new types of applications and techniques whose correct operation on 802.11 wireless hosts would not be possible without millisecond synchronisation.

The mechanism is currently limited to particular scenarios. The required hardware facilities are not yet provided by all NICs on the market and there are issues related to security. These limitations, however, can be overcome. They

are primarily associated with NIC driver limitations, and such drivers could be modified in response to consumer demand.

Part IV

Conclusions

Chapter 7

Conclusions and Future Work

This thesis has outlined and validated two significant research contributions within the scope of time synchronisation over wireless networks. The first of these focused on energy efficient time synchronisation in WSNs. It presented the design, operational details, and benefits of an alternative WSN time protocol, namely the *Dynamic Flooding Time Synchronisation Protocol (D-FTSP)*, an extension of the *Flooding Time Synchronisation Protocol (FTSP)*. D-FTSP is designed to minimise energy wastage that results from communication overhead in WSNs operating in dynamic/temperature-varying environments. Its deployment in such environments can lead to substantial energy savings, thus, pro-longing the battery-life of WSNs. Energy use/battery life is often a key constraint in WSN deployments and, thus, this contribution is significant.

The second contribution focused on significantly improving the performance of two-way PSN time synchronisation protocols such as NTP over IEEE 802.11 networks, in particular, busy networks. A solution was presented in the form of a technique/mechanism that leverages facilities provided by current *Wireless Network Interface Cards (WNICs)* and exploits the standard operational details of 802.11 networks in order to determine delays incurred by time packets traversing such networks. These delays are subsequently used to reduce the error in datasets used by time protocols such as NTP. Experiments reveal reductions in errors as great as 90% which can lead to a significant improvement in performance, similar to that achievable over wired links. This proves a significant contribution since the improvement of synchronisation levels over 802.11 networks is currently a big

issue.

7.1 Core Contributions

The core research contributions of this thesis can be summarised as follows:

7.1.1 D-FTSP

- *Saves energy* - D-FTSP, a dynamic WSN time synchronisation protocol capable of providing accuracies equivalent to those of current static WSN time protocols but with a reduced number of transmissions in dynamic environments. Its use in dynamic environments can lead to significant energy savings, pro-longing the battery-life of WSN's.

- *Saves time* - D-FTSP configures the transmission rate of WSN nodes automatically post-deployment and, therefore, eliminates the need for pre-deployment configuration. This speeds up the network deployment phase since an analysis of the dynamics of the operating environment is not required.

- *Responsive* - D-FTSP is capable of providing accuracies equivalent to those of current dynamic WSN time protocols but may react much more rapidly to changing skew and with no additional hardware requirements. In addition, D-FTSP monitors a node's clock drift directly and, therefore, can deal with multiple sources of drift, including that which results from relatively unstable oscillators.

7.1.2 Improved Synchronisation over 802.11 Networks

- The core contribution is a mechanism in the form of a software module that delivers synchronisation levels over 802.11 networks similar to those achievable over wired networks, thus, representing a significant improvement. It dynamically determines/estimates delays incurred by time messages as they

traverse an 802.11 link that connects a wireless station and its associated access point.

- Knowledge of network delays associated with a time packet can be used to reduce those errors associated with non-deterministic message latencies. This leads to greatly improved synchronisation accuracy.
- Although the contribution is validated using the NTP protocol, the module can be used in conjunction with any internet time protocol that employs the round-trip or uni-directional synchronisation technique.

7.2 Future Work

While both research contributions were validated through rigorous experimentation, some scope for further research was identified, as detailed below.

7.2.1 D-FTSP Maximum and Minimum Intervals

It may be possible to further improve the efficiency of D-FTSP by having it dynamically alter the minimum and maximum transmission intervals of nodes with respect to the node density of a WSN. Referring to chapter 4, D-FTSP, as it stands, is statically configured with a minimum and maximum transmission interval. The maximum interval is necessary to prevent the transmission interval from incrementing indefinitely. The difference between the minimum and maximum interval forms a range from which the actual transmission interval is dynamically chosen in response a node's clock drift. The minimum and maximum interval could be dynamically chosen with respect to an application's accuracy requirements, thus, delivering a more responsive and efficient solution.

To clarify, one can consider a scenario whereby an ideal D-FTSP maximum transmission interval is chosen, and a node that receives messages at this interval accumulates an error less than the application error bound. If this node has one parent, it will eventually receive messages at the maximum transmission interval and still remain synchronised. If, however, this node has two parents, then it will

most likely receive more messages than is necessary for it to remain synchronised. In this case, if the node's parents increase their maximum transmission intervals such that the maximum time difference between both parent's message transmissions never exceeds the originally configured maximum transmission interval, then further energy savings could be gained.

This, however, is not a trivial task. For it to work, the parents of a node would first have to determine how many additional parents their child has. This could be achieved by having D-FTSP nodes broadcast the number of parents they have in time messages. Of course, the number of parents alone would not be enough information to guarantee an error free solution, since it is possible that all parents could be transmitting at almost the same time. The node's parents would also have to know the maximum time interval between any two consecutive messages received by their child. Thus, the problem could get relatively complex with respect to any energy savings gained.

7.2.2 Module API

The mechanism detailed in chapter 6 was implemented as a software module that runs as a separate process relative to the NTP daemon process. For experimental purposes, the NTP application code was modified such that it could retrieve delay measurements from the module process via a file and modify/correct generated timestamps. This, however, was the extent of the interaction between the processes. If the module was implemented in the form of a library with a standard API that time protocols could employ, then this would make deployment more transparent and simple and improve the performance of the module in terms of efficiency as the module/synch protocol would exist as a single process.

To elaborate, the module implemented in the form of a library with a standard API would allow a time protocol to alter the behaviour of the module directly in response to certain events and/or requirements. Even the most basic API with simple *start* and *stop* functions could drastically improve the modules efficiency by reducing overhead. Thus, rather than the current situation whereby the module monitors traffic continuously, it could be started and stopped by the NTP daemon in response to NTP packet transmission and reception events.

Consequently, wireless traffic would only be monitored between transmission and reception events, thus, significantly reducing overhead.

In addition to this, the module could be configured to inject a controlled sequence of *"crafted"* packets into the network between NTP packet transmission and reception events. This burst of packet injections would provide numerous down-link delay measurements that when averaged, or processed by some other means, could provide an improved estimate for the down-link delay of an NTP response packet. This could potentially improve the accuracy of down-link delay estimations by the module. Such data could also be used to analyse trends in the down-link delay measurements and possibly make predictions about future delays.

Appendix A

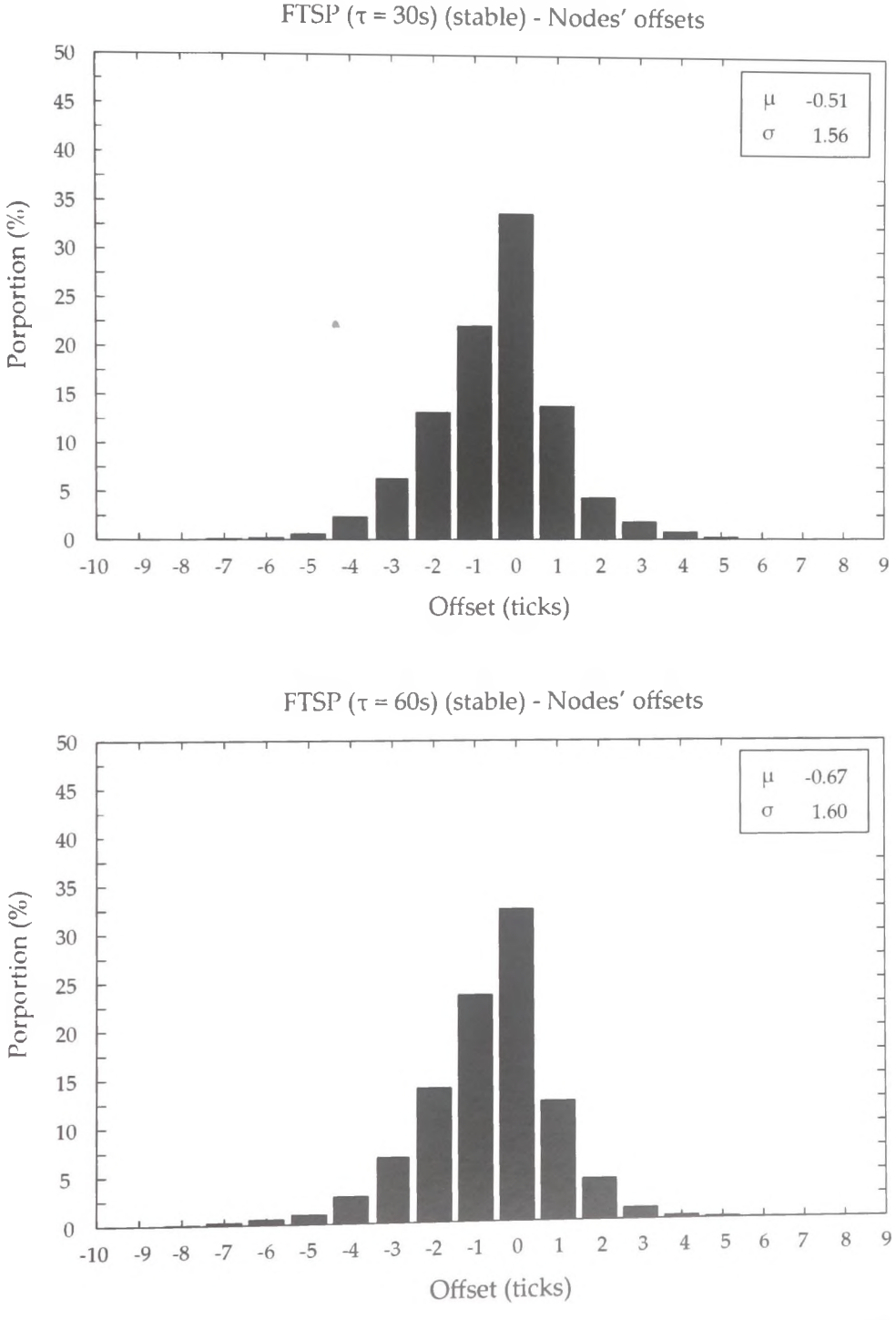


Figure 1: Raw data for FTSP experiments in stable environment with $\tau = 30$s (top) and $\tau = 60$s (bottom)

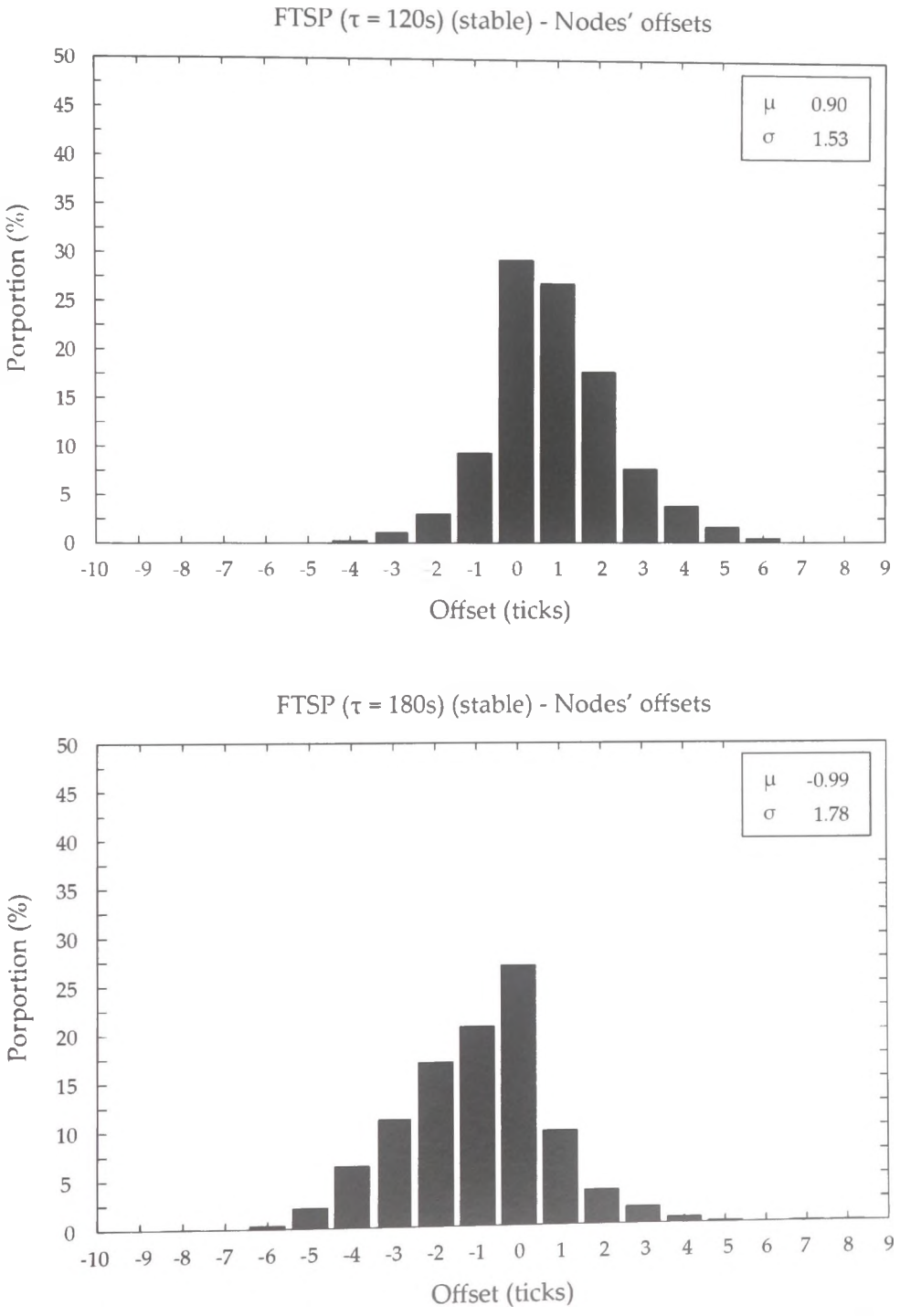


Figure 2: Raw data for FTSP experiments in stable environment with $\tau = 120$s (top) and $\tau = 180$s (bottom)

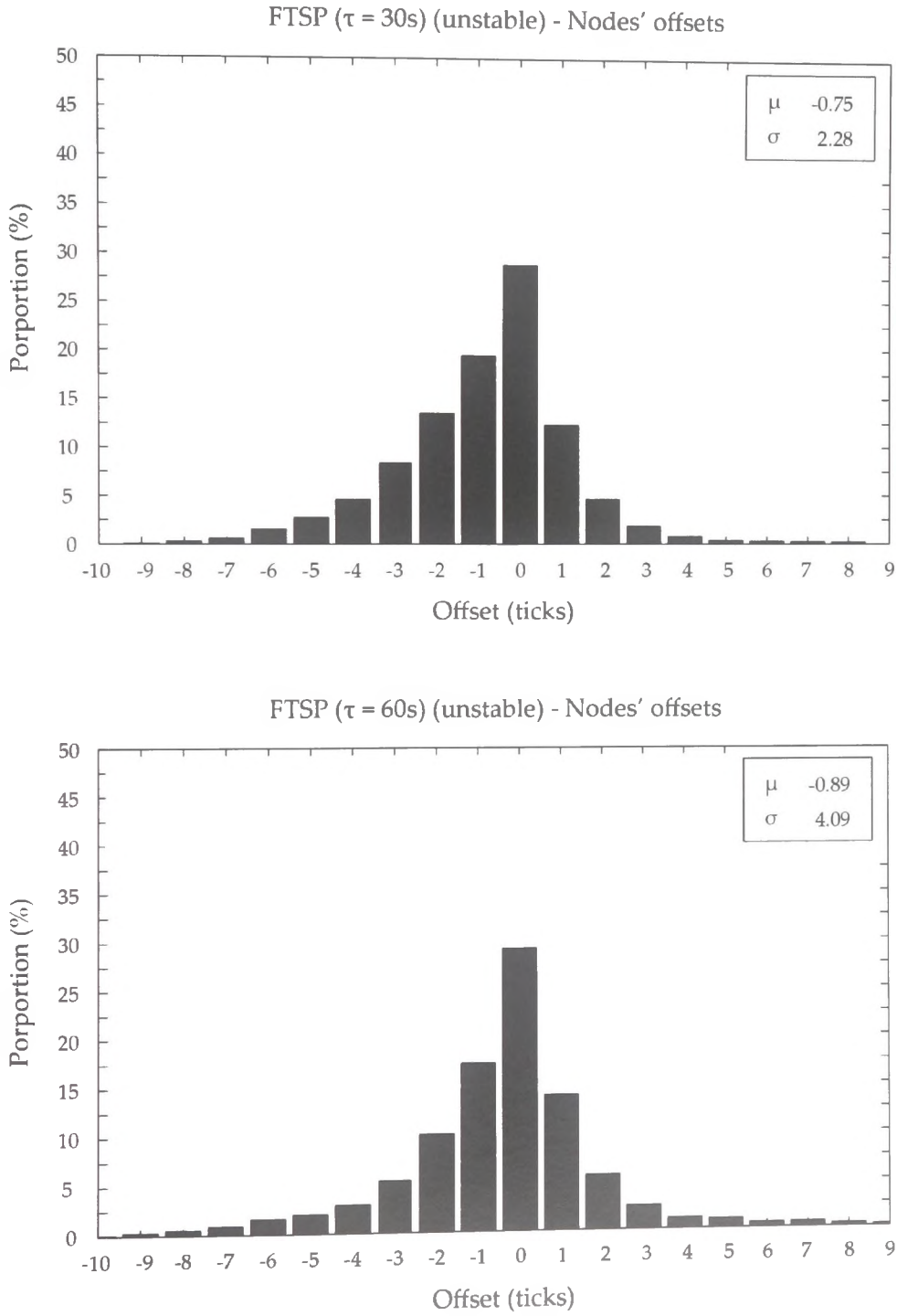


Figure 3: Raw data for FTSP experiments in unstable environment with $\tau = 30$s (top) and $\tau = 60$s (bottom)

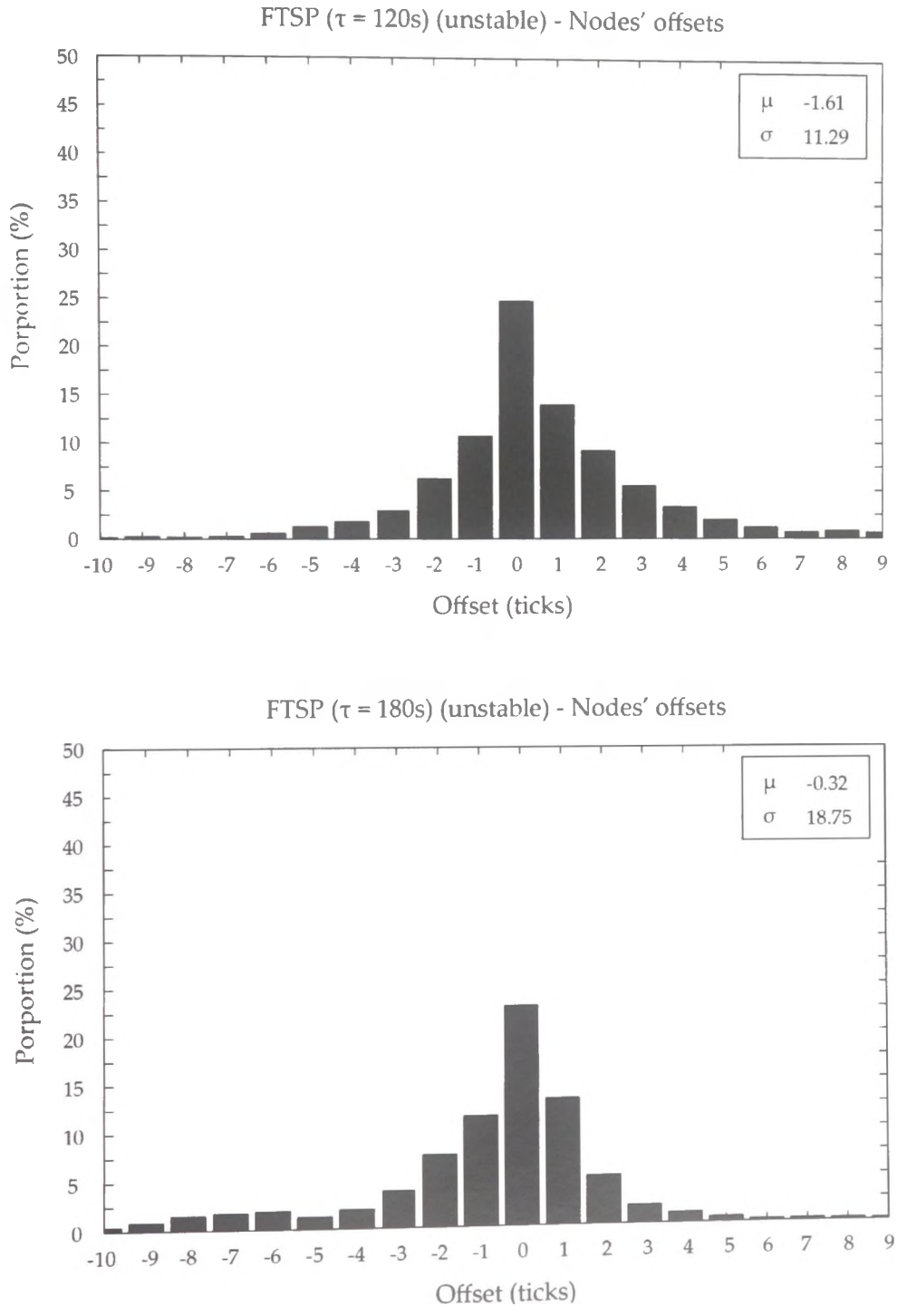


Figure 4: Raw data for FTSP experiments in unstable environment with τ = 120s (top) and τ = 180s (bottom)

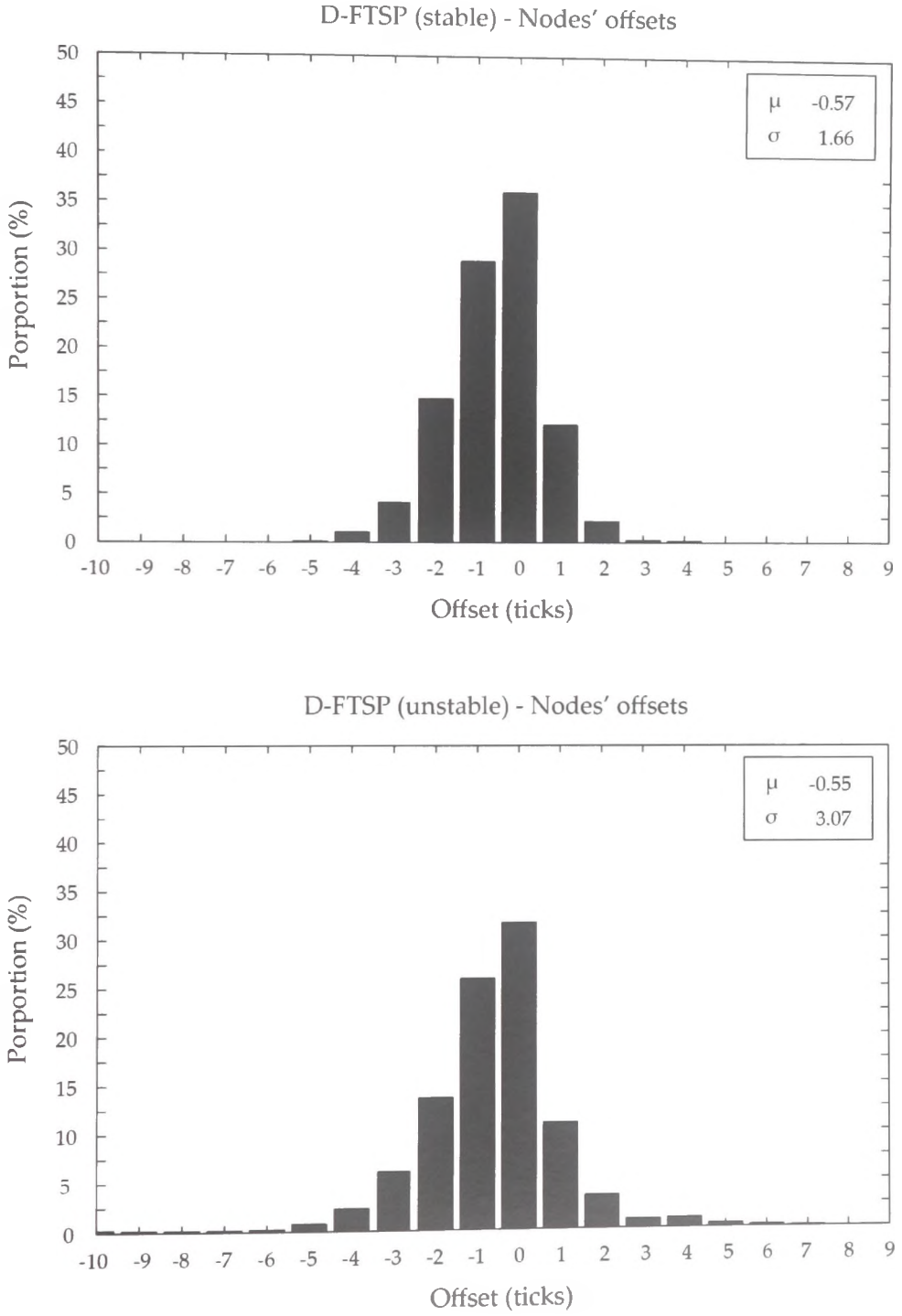


Figure 5: Raw data for D-FTSP experiments in stable (top) and unstable (bottom) environment with $\epsilon_{avg} = 1$ tick

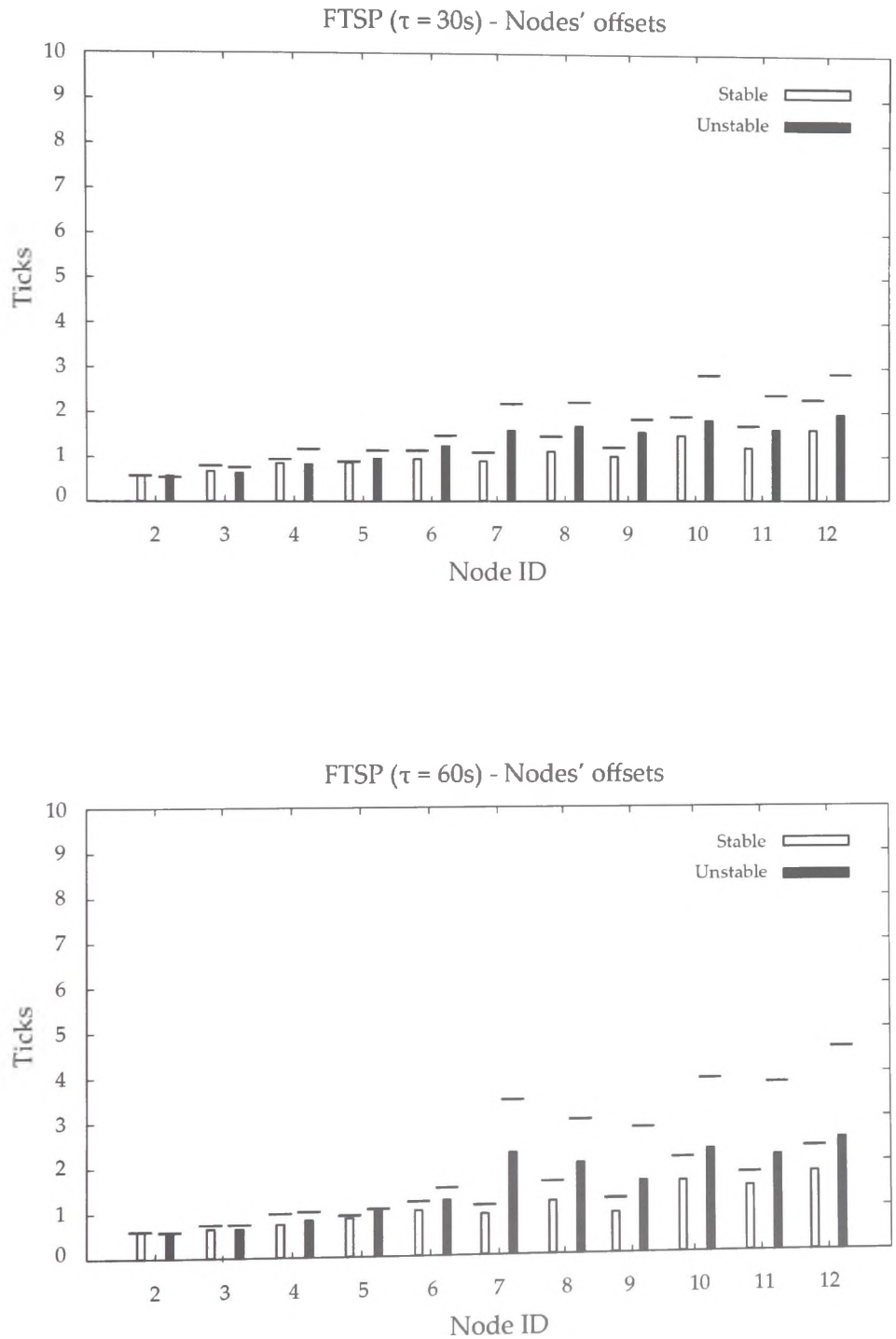


Figure 6: Nodes' mean offset (μ) and standard deviation (σ) for FTSP experiments in stable and unstable environment with τ = 30s (top) and τ = 60s (bottom).

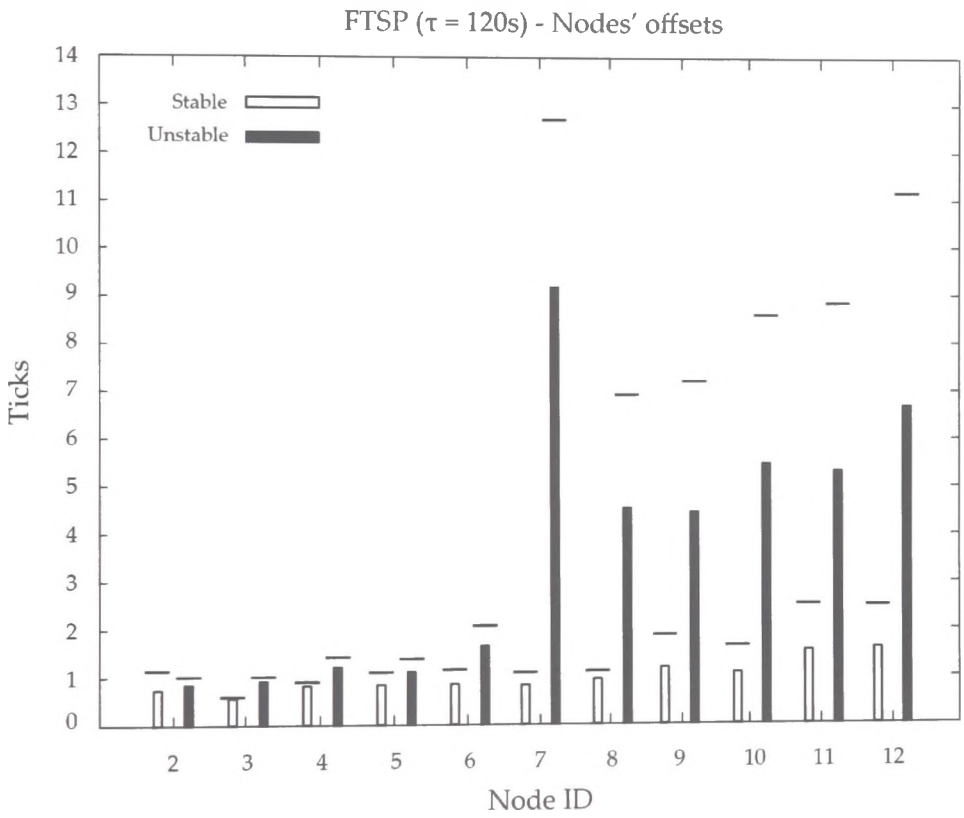


Figure 7: Nodes' mean offset (μ) and standard deviation (σ) for FTSP experiments in stable and unstable environment with $\tau = 120$s

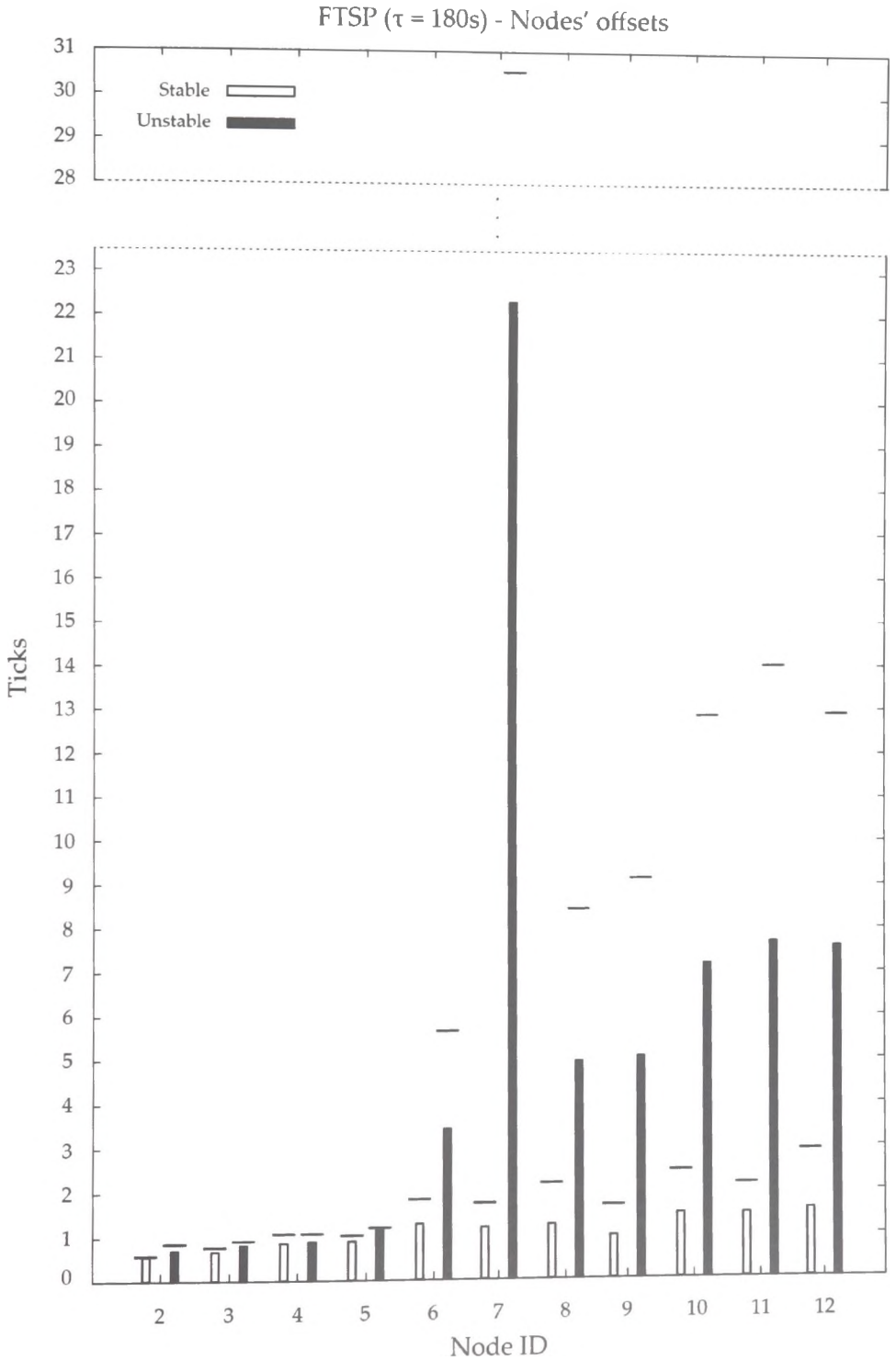


Figure 8: Nodes' mean offset (μ) and standard deviation (σ) for FTSP experiments in stable and unstable environment with $\tau = 180$s

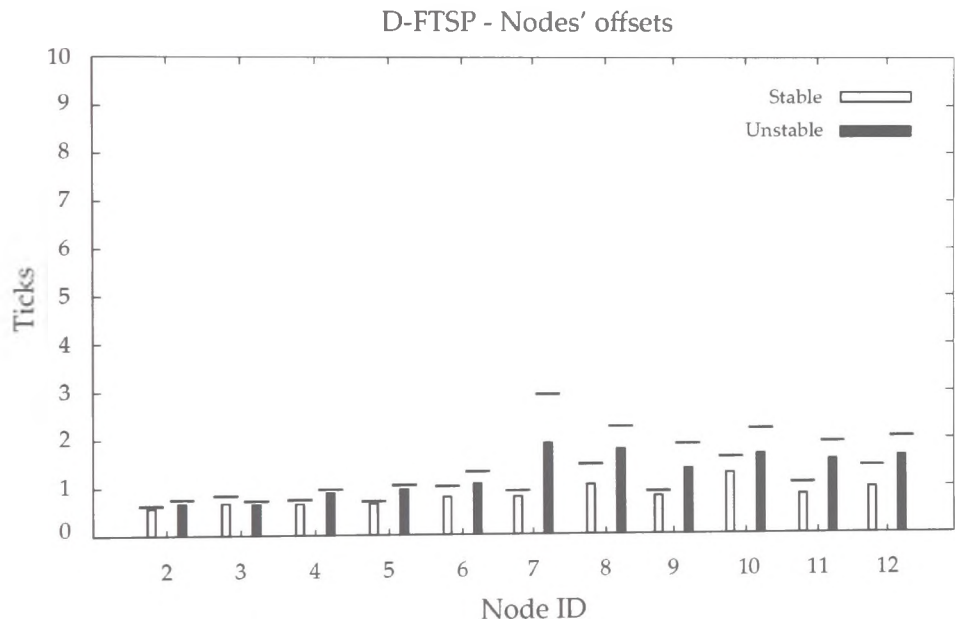


Figure 9: Nodes' mean offset (μ) and standard deviation (σ) for D-FTSP experiments in stable and unstable environment with $\epsilon_{avg} = 1$ tick

Bibliography

[1] Dave Evans. Internet of things: How the next evolution of the internet is changing everything. Technical report, CISCO, Apr 2011.

[2] More than 50 billion connected devices - taking connected devices to mass market and profitability. Technical report, ERICSSON, Feb 2011.

[3] IEEE. Draft standard for local and metropolitan area networks - specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications - amendment 3: Enhancements for very high throughput in the 60 ghz band. *IEEE P802.11ad/D8.0, May 2012 (Draft Amendment based on IEEE 802.11-2012)*, pages 1–667, 2012.

[4] Ed Callaway, Paul Gorday, Lance Hester, Jose A. Gutierrez, Marco Naeve, Bob Heile, and Venkat Bahl. Home networking with ieee 802.15.4: a developing standard for low-rate wireless personal area networks. *Communications Magazine, IEEE*, 40(8):70–77, 2002.

[5] Stacey Higginbotham. Zigbee and z-wave are out. broadcoms new chips bet on bluetooth and wi-fi for iot, May 2013. URL `http://gigaom.com/`. Online.

[6] Daniel M. Dobkin and Bernard Aboussouan. Low power wi-fi for ip smart objects. Technical report, GainSpan Corporation, 2013.

[7] Serbulent Tozlu, Murat Senel, Wei Mao, and Abtin Keshavarzian. Wi-fi enabled sensors for internet of things: A practical approach. *Communications Magazine, IEEE*, 50(6):134–143, 2012. ISSN 0163-6804. doi: 10.1109/MCOM.2012.6211498.

[50] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. 1992.

[51] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking (TON)*, 3(3):245–254, 1995.

[52] David L. Mills. Precision synchronization of computer network clocks. *ACM SIGCOMM Computer Communication Review*, 24(2):28–43, 1994.

[53] John C. Eidson. *Measurement, Control, and Communication Using IEEE 1588 (Advances in Industrial Control)*. Springer, 2006. ISBN 978-1846282508.

[54] John C. Eidson and Kang Lee. Ieee 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*, pages 98–105. IEEE, 2002.

[55] Matthew Gast. *802.11 wireless networks: the definitive guide*. O'Reilly Media, Inc., 2002. ISBN 978-0596001834.

[56] Thomas R. Henderson, Mathieu Lacage, George F. Riley, C. Dowell, and J.B. Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 2008.

[57] Thomas R. Henderson, Mathieu Lacage, George F. Riley, C. Dowell, and J.B. Kopena. Network simulator 3, Apr 2013. URL `http://www.nsnam.org/`. Online.

[58] Tianji Li, Douglas Leith, and David Malone. Buffer sizing for 802.11-based networks. *IEEE/ACM Transactions on Networking (TON)*, 19(1):156–169, Feb 2011. ISSN 1063-6692. doi: 10.1109/TNET.2010.2089992. URL `http://dx.doi.org/10.1109/TNET.2010.2089992`.

[59] Luis Martin Garcia. Programming with libpcap±sniffing the network from our own application. *Hakin9-Computer Security Magazine*, pages 2–2008, 2008.